Principles of Eventual Consistency

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: 978-1-60198-858-4

Foundations and Trends® in Programming Languages

Volume 1, Issue 1-2, 2014

Editorial Board

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 resonably 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.

2

Preliminaries

One of our main themes is to use mathematical language to describe expected or actual behaviors of distributed systems. In this chapter, we give careful explanations of the technical foundations we use throughout the book. Readers may read it from beginning to end, but are encouraged to skim or skip through it and refer back when needed.

We rely mostly on standard notations that are commonly used in textbooks, but we also introduce some custom notations and concepts that are particularly useful for our purpose, most notably event graphs (§2.2).

2.1 Sets and Functions

We use standard notations for working with sets. Note that we write $A \subseteq B$ to denote $\forall a \in A : a \in B$. In particular, the notation $A \subseteq B$ does neither imply nor rule out either $A = B$ or $A \neq B$. We let $\mathbb{N}$ be the set of all natural numbers (starting with number 1), and $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$.

We write $A \to B$ to denote the set of functions from A to B, and $A \rightharpoonup B$ to denote the set of partial functions. Following tradition, we write $f : A \to B$ to mean $f \in (A \to B)$. For a function $f : A \to B$,

we define $\mathsf{dom} f \stackrel{\text{def}}{=} A$, and for a partial function $f : A \rightharpoonup B$, we define $\mathsf{dom} f \stackrel{\text{def}}{=} \{a \in A \mid \exists b \in B : f(a) = b\}$. When working with a partial function f, we write $f(a) = \bot$ to mean $a \notin \mathsf{dom} f$. The symbol $\bot$ is used exclusively for this purpose, *i.e.* is not an element of any set.

Functions and partial functions can be interpreted as relations: $(A \to B) \subseteq (A \rightharpoonup B) \subseteq (A \times B)$, and we take advantage of this in our notations. For instance, we write $\emptyset$ for the partial function with empty domain. For any (partial) function $f : A \to B$ and elements $a \in A$, $b \in B$, we define the (partial) function $f[a \mapsto b]$ as

$$f[a \mapsto b](x) \stackrel{\text{def}}{=} \begin{cases} b & \text{if } x = a \\ f(x) & \text{otherwise.} \end{cases}$$

The power set $\mathcal{P}(A) = \{A' \mid A' \subseteq A\}$ is the set of all subsets of A. We can lift a function $f : A \to B$ to a function $\lfloor f \rfloor : \mathcal{P}(A) \to \mathcal{P}(B)$ by $\lfloor f \rfloor(A') = \{f(a) \mid a \in A'\}$ and may sometimes do so implicitly, *i.e.* using the same symbol f to denote $\lfloor f \rfloor$.

For a finite or infinite set A, we write $|A| < \infty$ or $|A| = \infty$, respectively. We define $A \subseteq_{\text{fin}} B \stackrel{\text{def}}{\iff} (A \subseteq B \wedge |A| < \infty)$, $\mathcal{P}_{\text{fin}}(A) \stackrel{\text{def}}{=} \{B \mid B \subseteq_{\text{fin}} A\}$, and $A \rightharpoonup_{\text{fin}} B \stackrel{\text{def}}{=} \{f : A \rightharpoonup B \mid |\mathsf{dom} A| < \infty)$.

2.1.1 Finite Sequences

Given a set A, we let A^* be the set of finite sequences (or "words") of elements of A, including the empty sequence which is denoted ϵ. We identify sequences of length one with the element they contain, thus $A \subseteq A^*$. We let $A^+ \subseteq A^*$ be the set of nonempty sequences of elements of A. Thus, $A^* = A^+ \cup \{\epsilon\}$.

For two sequences $u, v \in A^*$, we write $u \cdot v$ to denote the concatenation (which is also in A^*). If $f : A \to B$ is a function, and $w \in A^*$ is a sequence, then we let $f(w) \in B^*$ be the sequence obtained by applying f to each element of w.

We define operators sort, map, and foldr as follows: (1) Given a finite set A and a total order rel on A, we let $A.\mathsf{sort}(\mathsf{rel}) \in A^*$ be the sequence obtained by arranging the elements of A in ascending $<_{\mathsf{rel}}$-order. (2) Given a sequence $w \in A^*$, and a function $f : A \to B$, we define $w.\mathsf{map}(f) \in B^*$ to be the sequence obtained by applying f to each

element of w. (3) Given an element $a_0 \in A$, a function $f : A \times B \to A$, and a sequence $w \in B^*$, we define

$$\mathsf{foldr}(a_0, f, w) = \begin{cases} a_0 & \text{if } w = \epsilon \\ f(\mathsf{foldr}(a_0, f, w'), b) & \text{if } w = w'b \end{cases}$$

2.1.2 Relations

A binary relation rel over A is a subset $\mathsf{rel} \subseteq A \times A$. For $a, b \in A$, we use the notation $a \xrightarrow{\mathsf{rel}} b$ to denote $(a, b) \in \mathsf{rel}$, and the notation $\mathsf{rel}(a)$ to denote $\{b \in A \mid a \xrightarrow{\mathsf{rel}} b\}$. We use the notation rel^{-1} to denote the inverse relation, i.e. $(a \xrightarrow{\mathsf{rel}^{-1}} b) \Leftrightarrow (b \xrightarrow{\mathsf{rel}} a)$. Therefore, $\mathsf{rel}^{-1}(b) = \{a \in A \mid a \xrightarrow{\mathsf{rel}} b\}$ (we use this notation frequently).

Given two binary relations $\mathsf{rel}, \mathsf{rel}'$ over A, we define the composition $\mathsf{rel}\,;\mathsf{rel}' = \{(a, c) \mid \exists b \in A : a \xrightarrow{\mathsf{rel}} b \xrightarrow{\mathsf{rel}'} c\}$. We let id_A be the identity relation over A, i.e. $(a \xrightarrow{\mathsf{id}_A} b) \Leftrightarrow (a \in A) \wedge (a = b)$. For $n \in \mathbb{N}_0$, We let rel^n be the n-ary composition $\mathsf{rel}\,;\mathsf{rel}\ldots;\mathsf{rel}$, with $\mathsf{rel}^0 = \mathsf{id}_A$. We let $\mathsf{rel}^+ = \bigcup_{n\geq 1} \mathsf{rel}^n$ and $\mathsf{rel}^* = \bigcup_{n\geq 0} \mathsf{rel}^n$. We let $\mathsf{rel}^? = \mathsf{rel}^0 \cup \mathsf{rel}^1$. For some subset $A' \subseteq A$, we define the restricted relation $\mathsf{rel}|_{A'} \stackrel{\text{def}}{=} \mathsf{rel} \cap (A' \times A')$.

We often abbreviate conjunctions of relations when convenient; for example, $a = b = c$ is short for $(a = b \ \wedge \ b = c)$, and $a \xrightarrow{\mathsf{rel}} b \xrightarrow{\mathsf{rel}} c$ is short for $(a \xrightarrow{\mathsf{rel}} b \ \wedge \ b \xrightarrow{\mathsf{rel}} c)$

2.1.3 Orders and Equivalences

Relations can represent many different things. In our context, we are particularly interested in the cases where relations represent some kind of ordering of events, or an equivalence relation. We define various properties of relations in Figure 2.1.

Partial orders are irreflexive and transitive, which implies acyclic (because any cycle in a transitive relation implies a self-loop). We often visualize partial orders as directed acyclic graphs. Moreover, in such drawings, we usually omit transitively implied edges, to avoid overloading the picture.

A partial order does not necessarily order all elements, which distinguishes it from a total order. All total orders are also partial or-

Property	Element-wise Definition $\forall x, y, z \in A$:	Algebraic Definition
symmetric	$x \xrightarrow{\mathsf{rel}} y \Rightarrow y \xrightarrow{\mathsf{rel}} x$	$\mathsf{rel} = \mathsf{rel}^{-1}$
reflexive	$x \xrightarrow{\mathsf{rel}} x$	$\mathsf{id}_A \subseteq \mathsf{rel}$
irreflexive	$x \not\xrightarrow{\mathsf{rel}} x$	$\mathsf{id}_A \cap \mathsf{rel} = \emptyset$
transitive	$(x \xrightarrow{\mathsf{rel}} y \xrightarrow{\mathsf{rel}} z) \Rightarrow (x \xrightarrow{\mathsf{rel}} z)$	$(\mathsf{rel}\,;\mathsf{rel}) \subseteq \mathsf{rel}$
acyclic	$\neg(x \xrightarrow{\mathsf{rel}} \ldots \xrightarrow{\mathsf{rel}} x)$	$\mathsf{id}_A \cap \mathsf{rel}^+ = \emptyset$
total	$x \neq y \Rightarrow (x \xrightarrow{\mathsf{rel}} y \vee y \xrightarrow{\mathsf{rel}} x)$	$\mathsf{rel} \cup \mathsf{rel}^{-1} \cup \mathsf{id}_A = A \times A$

Property	Definition
natural	$\forall x \in A : \lvert\mathsf{rel}^{-1}(x)\rvert < \infty$
partialorder	irreflexive $\wedge$ transitive
totalorder	partialorder $\wedge$ total
enumeration	totalorder $\wedge$ natural
equivalencerelation	reflexive $\wedge$ transitive $\wedge$ symmetric

Figure 2.1: Definitions of common properties of a binary relation $\mathsf{rel} \subseteq A \times A$.

ders. For a partial or total order rel, we sometimes use the notation $a \leq_{\mathsf{rel}} b \overset{\text{def}}{\iff} [(a \xrightarrow{\mathsf{rel}} b) \vee (a = b)]$.

An equivalence relation is a transitive, reflexive, and symmetric relation. If rel is an equivalence relation, we sometimes use the notation $a \approx_{\mathsf{rel}} b \overset{\text{def}}{\iff} [a \xrightarrow{\mathsf{rel}} b]$. An equivalence relation rel on A partitions A into equivalence classes $[x]_{\mathsf{rel}} = \{y \in A \mid y \approx_{\mathsf{rel}} x\}$. The equivalence classes are pairwise disjoint and cover A. We write $A/\!\approx_{\mathsf{rel}}$ to denote the set of equivalence classes.

2.1.4 Countable Sets

A total order that is also natural (*i.e.* for each element x, there are only finitely many elements that are ordered before x) is called an *enumeration*. If there exists an enumeration for a set, that set is called *countable*. Countable sets can be finite or infinite.

If rel is an enumeration on a set A, we can choose elements $a_i \in A$ such that $A = \{a_0, a_1, \ldots\}$ with $(a_i \xrightarrow{\mathsf{rel}} a_j \Leftrightarrow i < j)$, by defining a_i

to be the (uniquely) determined element of A that has rank i, where $\mathsf{rank}(A, \mathsf{rel}, a) \stackrel{\text{def}}{=} |\{x \in A \mid x \xrightarrow{\mathsf{rel}} a\}|$. Also, we can define notations for a successor function and a predecessor partial function: $\mathsf{succ}(A, \mathsf{rel}, a_i) = a_{i+1}$, $\mathsf{pred}(A, \mathsf{rel}, a_{i+1}) = a_i$, and $\mathsf{pred}(A, \mathsf{rel}, a_0) = \bot$.

All total orders on finite sets are enumerations. However, not all total orders on infinite sets are enumerations: for example, the lexicographic order on $\mathbb{N}_0 \times \mathbb{N}_0$, defined as $(a, b) < (c, d) \stackrel{\text{def}}{\iff} (a < c) \vee (a = c \wedge b < d)$, is not natural.

Lemma 2.1. Subsets and products of countable sets are countable.

Proof. For subsets, the claim follows easily from the fact that relations remain total and natural when restricted to a subset. For products $\{a_0, a_1, \dots\} \times \{b_0, b_1, \dots\}$, we can enumerate the tuples (a_i, b_j) by first lexicographically enumerating the finite set of tuples whose indexes add up to 0, then the finite set of tuples whose indexes add up to 1, and so on. This process yields the enumeration $(a_0, b_0), (a_0, b_1), (a_1, b_0), (a_0, b_2), (a_1, b_1), (a_2, b_0), (a_0, b_3), \dots$ □

2.1.5 Order-Extension Principle

Sometimes, we want to take a partial order and add just enough edges to turn it into a total order. For finite sets, the topological sort algorithm (a standard algorithm, which can be found in textbooks such as Cormen et al. [2003]) does just that: it provides a way to sort all elements of a directed acyclic graph (the partial order) into a sequence (a total order) such that there are no backward edges (thus, the total order extends the partial order). This process is also possible for infinite sets. We prove it for arbitrary countable sets (*i.e.* a set for which an enumeration exists) using the following deterministic construction.

Proposition 2.1 (Deterministic Totalization)**.** Let A be a countable set and en be an enumeration of A. Let rel be a partial order on A. Then we can define a total order $\mathsf{totalize}(\mathsf{rel}, \mathsf{en})$ on A such that $\mathsf{rel} \subseteq \mathsf{totalize}(\mathsf{rel}, \mathsf{en})$.

Proof. Since A has an enumeration en, we can enumerate it as $A = \{a_0, a_1, \dots\}$ where $(a_i \xrightarrow{\mathsf{en}} a_j) \Leftrightarrow (i < j)$. Define the set of pairs $P =$

$\{(a_i, a_j) \subseteq A \times A \mid i < j\}$ and enumerate P as defined in Lemma 2.1. Then, define a sequence of relations $\mathsf{rel}_0, \mathsf{rel}_1, \ldots$ as

$$\mathsf{rel}_0 \stackrel{\text{def}}{=} \mathsf{rel} \qquad \mathsf{rel}_{k+1} \stackrel{\text{def}}{=} \begin{cases} \mathsf{rel}_k & \text{if } p_k \in \mathsf{rel}_k^{-1} \\ (\mathsf{rel}_k \cup \{p_k\})^+ & \text{otherwise} \end{cases}$$

Then clearly, the rel_k are monotonic: $\mathsf{rel}_k \subseteq \mathsf{rel}_{k+1}$. Moreover, each rel_k is a partial order: for $k = 0$ this is assumed of rel. For the induction step: transitivity is easy (we either use the previous relation which is transitive by induction, or we use transitive closure); irreflexivity and acylicity hold because any newly formed cycle or self-loop has to contain the newly added edge (by induction, the previous relation is acyclic and irreflexive), and we only add the edge if its converse is not in the previous relation, and the edge is never a self-loop, thus no cycle or self-loop can form.

Finally, define $\mathsf{totalize}(\mathsf{rel}, \mathsf{en}) \stackrel{\text{def}}{=} \bigcup_k \mathsf{rel}_k$. This satisfies the conditions in the claim, since (1) it contains rel because $\mathsf{rel}_0 = \mathsf{rel}$, (2) it is total because for any a_i, a_j with $i < j$ there exists a k such that $p_k = (a_i, a_j)$, and then a_i and a_j are ordered in rel_{k+1}, (3) similarly, it is transitive because for any three $a_i \xrightarrow{\mathsf{rel}} a_j \xrightarrow{\mathsf{rel}} a_n$ we can find k large enough so that a_i, a_n are ordered in rel_k, which is a partial order so it must order them as $a_i \xrightarrow{\mathsf{rel}} a_n$ to not form a cycle, and (4) it is irreflexive and acyclic because any self-loop or cycle would have to be contained in some rel_k for k large enough, contradicting our finding that each rel_k is a partial order. □

The above proof is deterministic (we always construct the same total order if starting with the same partial order and enumeration). The order extension principle is true for general (non-countable) sets as well, but the proof requires the axiom of choice [Marczewski, 1930].

2.2 Event Graphs

To work with all the various specifications and guarantees, we need techniques and notations that let us conveniently reason about executions at various abstraction levels. We use *event graphs* for that pur-

pose, since event graphs can be easily projected (to remove information) and extended (to add information).

An event graph represents an execution of the system, and encodes information about that execution in the form of vertices, attributes, and relations.

- *Vertices* represent events that occurred at some point during the execution. The number of vertices can be infinite, which allows us to reason about properties of infinite executions (in particular, liveness properties). Events are drawn from some universe Events which we leave unspecified, but assume large enough to contain any sets we may encounter in a concrete situation.

- *Attributes* label vertices with information pertinent to the corresponding event, such as the operation performed, or the value returned.

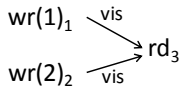
- *Relations* represent orderings or groupings of events; we visualize relations in various ways, such as by arrows (well suited for partial orders), or by aligning events vertically (well suited for total orders representing the real-time succession of events), or by adding numeric subscripts to event labels (well suited for total orders representing arbitration timestamps), or by grouping related events into dashed boxes (well suited for equivalence relations).

Definition 2.1. An *event graph* G is a tuple $(E, d_1, \ldots, d_n)$ where $E \subseteq \mathsf{Events}$ is a finite or countably infinite set of events, $n \geq 1$, and each d_i is an attribute or a relation over E.

To give an advance impression of the flexibility of event graphs, we show four examples where event graphs represent concepts that we will develop in this tutorial (histories, operation contexts, abstract executions, and concrete executions) in Figure 2.2. We will explain what they mean once we reach the corresponding definition later on.

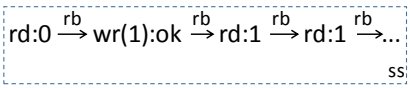
Isomorphisms. Event graphs are meant to carry information that is independent of the actual elements of Events chosen to represent the events.

(a) An event graph for an operation context (Definition 4.4):



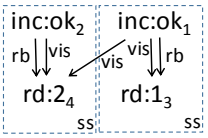
$$\left[\begin{array}{l} \text{graph is } (\{e_1, e_2, e_3\}, \mathsf{op}, \mathsf{vis}, \mathsf{ar}) \text{ where} \\ \mathsf{op} = \{(e_1, \mathsf{wr}(1)), (e_2, \mathsf{wr}(2)), (e_3, \mathsf{rd})\} \\ \mathsf{vis} = \{(e_1, e_3), (e_2, e_3)\} \\ \mathsf{ar} = \{(e_i, e_j) \mid i < j\} \end{array}\right]$$

(b) An event graph for an infinite history (Definition 3.1):



$$\left[\begin{array}{l} \text{graph is } (\mathbb{N}_0, \mathsf{op}, \mathsf{rval}, \mathsf{rb}, \mathsf{ss}) \text{ where} \\ \mathsf{op} = \{(0, \mathsf{rd}), (1, \mathsf{wr}(1))\} \\ \qquad \cup \{(n, \mathsf{rd}) \mid n \geq 2\} \\ \mathsf{rval} = \{(0, 0), (1, ok)\} \\ \qquad \cup \{(n, 1) \mid n \geq 2\} \\ \mathsf{rb} = \{(a, b) \mid a < b\} \\ \mathsf{ss} = \mathbb{N}_0 \times \mathbb{N}_0 \end{array}\right]$$

(c) An event graph for an abstract execution (Definition 3.3):



$$\left[\begin{array}{l} \text{graph is } (\{a_1, a_2, b_1, b_2\}, \mathsf{op}, \mathsf{rval}, \mathsf{rb}, \mathsf{ss}, \mathsf{vis}, \mathsf{ar}) \text{ where} \\ \mathsf{op} = \{(a_1, \mathsf{inc}), (a_2, \mathsf{rd}), (b_1, \mathsf{inc}), (b_2, \mathsf{rd})\} \\ \mathsf{rval} = \{(a_1, ok), (a_2, 2), (b_1, ok), (b_2, 1)\} \\ \mathsf{rb} = \{(a_1, a_2), (b_1, b_2)\} \\ \mathsf{ss} = \{(a_1, a_2), (a_2, a_1), (b_1, b_2), (b_2, b_1)\} \\ \mathsf{vis} = \{(a_1, a_2), (b_1, b_2), (b_1, a_2)\} \\ \mathsf{ar} = b_1 < a_1 < b_2 < a_2 \end{array}\right]$$

(d) An event graph for a concrete execution (Definition 7.5):

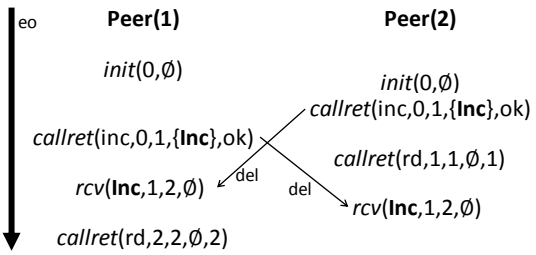


Figure 2.2: Four examples of event graphs used for different purposes throughout this tutorial.

Definition 2.2. Two event graphs $G = (E, d_1, \ldots, d_n)$ and $G' = (E', d'_1, \ldots, d'_n)$ are *isomorphic*, written $G \simeq G'$, if d_i and d'_i are of the same kind (attribute vs. relation) and if there exists a bijection $\phi : E \to E'$ such that for all d_i where d_i is an attribute, and all $x \in E$, we have $d_i(\phi(x)) = d'_i(x)$, and such that for all d_i where d_i is a relation, and all $x, y \in E$, we have $x \xrightarrow{\mathsf{d_i}} y \Leftrightarrow \phi(x) \xrightarrow{\mathsf{d'_i}} \phi(y)$.

Lemma 2.2. Let $G = (E, \mathsf{rel})$ be an event graph where rel is an enumeration. Then, G is isomorphic to $(N, <)$ where N is either $\{0, 1 \ldots, n\}$ for some n, or $N = \mathbb{N}_0$.

Proof. See § A.1.1 in the appendix. □

2.2.1 Projection and Extension

It is often convenient to remove information from event graphs, either by carving out a subgraph (*i.e.* restricting to a subset of vertices), or by removing some attributes or relations, or both.

Definition 2.3. Let $G = (E, d_1, \ldots, d_n)$ be an event graph, let $E' \subseteq E$ be a subset of the vertices in G, and let $\{d'_1, \ldots d'_k\} \subseteq \{d_1, \ldots, d_n\}$ be a subset of the attributes/relations. Then, we let $G|_{E', d'_1, \ldots, d'_k}$ denote the projected graph $(E', d'_1, \ldots, d'_k)$ where for all d'_i where $d'_i = d_j$: $d'_i|_{E'} = d_j|_{E'}$.

We say an event graph G_1 is a *projection* of an event graph G_2 if $G_1 = G_2|_{E, d_1, \ldots, d_n}$ for some E, d_i. Conversely, we say an event graph G_2 is an *extension* of an event graph G_1 if G_1 is a projection of G_2.

Lemma 2.3 (Structure Preservation)**.** Projection preserves the character of a relation: partial orders, total orders, natural orders, enumerations, and equivalence relations remain partial orders, total orders, natural orders, enumerations, and equivalence relations, respectively, when projected to a subset.

The lemma is easily proved by checking that each property in Fig. 2.1 is preserved.

3

Consistency Specifications

A consistency model, just like any other specification, serves as a contract between the system architects and the client programmers. Such a contract simplifies the task of each side: system architects need not know the details about how client programs intend to use the system, and client programmers need not understand how the system is implemented. The key to a successful specification is the appropriate use of abstraction.

The following steps provide a brief overview of our technical approach to defining a consistency model and checking conformance of an implementation.

1. Define a set of all observable behaviors $\mathcal{H}$, called histories (§3.1). This amounts to defining the interface on which the clients interact with the system, and specifying what information to record about this interaction.

2. Define the subset $\mathcal{H}_{\text{good}} \subseteq \mathcal{H}$ of *correct behaviors*. This is how we specify the consistency. In general, one may do so operationally (by a reference model) or declaratively (by a collection of consistency guarantees). The methodology we present here is declara-

tive and based on abstract executions (§3.2), which are histories enriched by additional information about visibility and arbitration.

3. For a particular implementation Π, determine its observable behaviors $\mathcal{H}(\Pi)$, and verify correctness by proving $\mathcal{H}(\Pi) \subseteq \mathcal{H}_{\text{good}}$. We demonstrate how to verify protocols in Chapters 8 and 10.

3.1 Histories

We capture the observable behavior of a system by defining its *histories*. A history records all the interactions between clients and the system. We include the following information in each history:

- The operations performed. To keep the formalism simple, we define a single set Operations to be the set that contains all operations of all data types (more about this in §4.1.2).

- Whether the operation completed, and what value was returned. We define the set Values broadly to include values of all types (more about this in §4.1.1).

- The relative order of non-overlapping operations. Operations are non-overlapping if one of them returns before the other one is called, in real time.

- The *session* an operation belongs to. Sessions represent distinct, sequential interfaces to the system: there can be at most one operation pending per session. Our sessions correspond to what are often called processes or clients (on shared memory) or contexts or connections (for replicated stores or databases). In this tutorial, sessions are always bound to a particular role, but in general, systems may allow sessions to be migrated [Terry et al., 1994].

Histories are commonly visualized using timeline diagrams as shown at the top of Fig. 3.1. Time goes from left to right, and operations are drawn as horizontal intervals, whose left end represents the operation

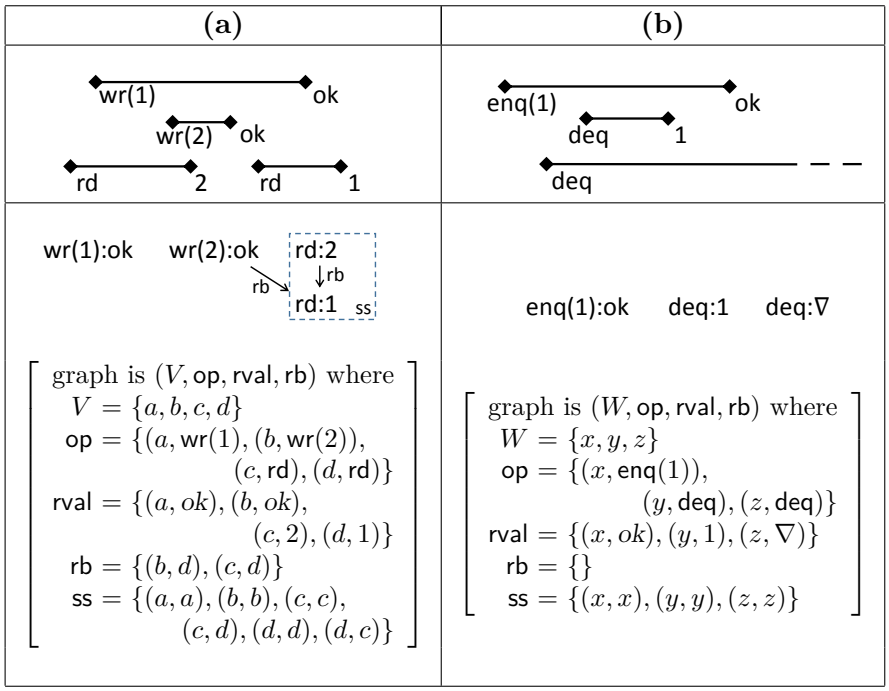


Figure 3.1: Two examples of conventional timeline diagrams that represent histories (top), and the corresponding event graphs (bottom).

call and whose right end (if present) represents the operation return, and is labelled by the returned value.

Figure (a) shows a history of an integer register supporting a read operation rd and write operations $\{\mathsf{wr}(n) \mid n \in \mathbb{N}\}$. There are four operations, two writes by two different sessions (writing 1 and 2, respectively), and two reads by the same session (returning 1 and 2, respectively). Update operations return the constant *ok* (where $ok \in \mathsf{Values}$) to indicate completion. Figure (b) shows a history of a FIFO queue supporting a enqueue operation $\{\mathsf{enq}(n) \mid n \in \mathbb{N}\}$ and a dequeue operation deq. There are three operations in three separate sessions, one enqueue and two dequeues, one of which is pending.

Formally, we represent histories using *event graphs* (as described in §2.2). Each event in the event graph corresponds to an interval in the timeline diagram, as shown in Figure 3.1 (bottom).

Definition 3.1 (History)**.** A *history* is an event graph $(E, \mathsf{op}, \mathsf{rval}, \mathsf{rb}, \mathsf{ss})$ where

(h1) $\mathsf{op} : E \to \mathsf{Operations}$ describes the operation of an event.

(h2) $\mathsf{rval} : E \to \mathsf{Values} \cup \{\nabla\}$ describes the value returned by the operation, or the special symbol ∇ ($\nabla \notin \mathsf{Values}$) to indicate that the operation never returns.

(h3) rb is a natural partial order on E, the *returns-before* order.

(h4) ss is an equivalence relation on E, the *same-session* relation.

The relation rb captures the ordering of non-overlapping operations. To represent session information, we use a single relation ss, which we call the *same-session* relation. It is an equivalence relation that indicates that two operations have been issued as part of the same session. Not surprisingly, we call the equivalence classes of ss (*i.e.* the sets $[e]_{\mathsf{ss}} = \{e' \in E \mid e' \approx_{ss} e\}$) *sessions*.

To ensure that histories are meaningful, we need a few additional conditions, as captured in the following definition.

Definition 3.2 (Well-formed History)**.** A history $(E, \mathsf{op}, \mathsf{rval}, \mathsf{rb}, \mathsf{ss})$ is *well-formed* if

(h5) $x \xrightarrow{\mathsf{rb}} y$ implies $\mathsf{rval}(x) \neq \nabla$ for all $x, y \in E$.

(h6) for all $a, b, c, d \in E$: $(a \xrightarrow{\mathsf{rb}} b \ \wedge \ c \xrightarrow{\mathsf{rb}} d) \Rightarrow (a \xrightarrow{\mathsf{rb}} d \ \vee \ c \xrightarrow{\mathsf{rb}} b)$.

(h7) For each session $[e] \in E/{\approx_{ss}}$, the restriction $\mathsf{rb}|_{[e]}$ is an enumeration.

Condition (h5) says that an operation that does not return at all cannot return before any operation. Condition (h6) ensures that rb is an *interval order* [Greenough, 1976], *i.e.* consistent with a timeline interpretation where operations correspond to segments. Condition (h7) ensures that sessions are indeed sequential — it implies that any two operations in the same session are ordered by the returns-before relation, thus they cannot overlap.

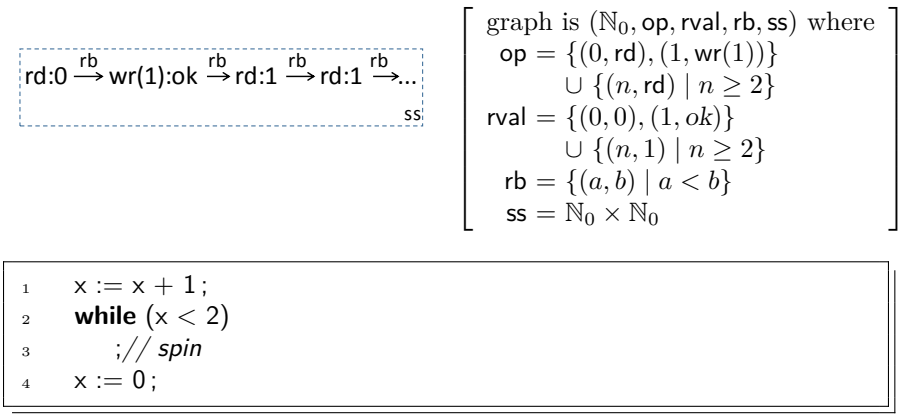


Figure 3.2: Example of an infinite history (top) and a snippet of client code that could cause it to happen (bottom).

Although we will not pursue the topic of transactions further, we would like to point out that it is straightforward to extend histories to also record transaction information, using an equivalence relation to capture which operations belong to the same transaction (*e.g.* as formalized by [Burckhardt et al., 2013]).

Infinite Histories. Since histories correspond to executions of some client program, they can be infinite if that program does not terminate (which is the desired behavior of services, for example). Fig. 3.2 shows an example of an event graph that is an infinite history, with a single session. The first event (leftmost vertex) is a read operation that returns the value 0. The second vertex is a write operation that writes the value 1. All remaining (infinitely many) vertices are read operations that all return the value 1. We can imagine this execution to be the result of a client program that reads and writes a single shared variable as shown at the bottom of Fig. 3.2. Note that the statement on line 1 produces two events, a read and a write that depends on the read. Also, note that line 4 is never executed because the spin loop on lines 2,3 continues forever. This example illustrates the difference between (dynamic) events and (static) statements: although each event is the result of some statement, there is not a one-to-one correspondence.

3.2 Abstract Executions

How do we tell good histories from bad histories? The most common approach is to require linearizability of histories. Intuitively, a history is *linearizable* if it is possible to insert commit points in the timeline, where we must place a commit point for each operation that completed somewhere between its call and its return. The role of the commit points is to serve as an explanation, or *justification*, that confirms the correctness of the observed return values. If no such witness exists, the history is invalid.

Our methodology follows the same general principle: a history is correct if and only if we can justify it, by augmenting it with some additional information that explains the observed return values. However, instead of adding commit points to a history, we add *visibility and arbitration* relations, which allow us to define not just linearizability, but the whole spectrum of consistency models that are commonly used for eventually consistent systems.

Visibility tells us about the relative timing of update propagation and operations. It is an acyclic relation. If an operation a is visible to b (written $a \xrightarrow{\mathsf{vis}} b$), it means that the effect of a is visible to the client performing b. In a system where updates are communicated by messages, this may mean that the message about operation a reached the client that performed operation b before the operation b was performed. We call two updates *concurrent* if they cannot see each other, *i.e.* are not ordered by visibility.

Arbitration is used to indicate how the system resolves *update conflicts*, *i.e.* how it handles concurrent updates that do not commute. It is a total order on operations. If an operation a is arbitrated before b (written $a \xrightarrow{\mathsf{ar}} b$), it means that the system considers the operation a to happen earlier than operation b. In practice, systems can arbitrate operations in various ways; most often, arbitration is represented by some kind of timestamp. The timestamp can be taken from a physical or logical clock on the node that performs the operation, or it can be affixed to the operation later, for example when it is processed on a single server that provides a serialization point.

Definition 3.3 (Abstract Executions)**.** An *abstract execution* is an event graph $(E, \mathsf{op}, \mathsf{rval}, \mathsf{rb}, \mathsf{ss}, \mathsf{vis}, \mathsf{ar})$ such that

(a1) $(E, \mathsf{op}, \mathsf{rval}, \mathsf{rb}, \mathsf{ss})$ is a history.

(a2) vis is an acyclic and natural relation.

(a3) ar is a total order.

We let $\mathcal{A}$ be the set of all abstract executions. For some abstract execution $A = (E, \mathsf{op}, \mathsf{rval}, \mathsf{rb}, \mathsf{ss}, \mathsf{vis}, \mathsf{ar})$, we get the corresponding history by removing visibility and arbitration: $\mathcal{H}(A) = (E, \mathsf{op}, \mathsf{rval}, \mathsf{rb}, \mathsf{ss})$.

Example. Fig. 2.2(c) shows an example of an event graph that is an abstract execution.

3.3 Consistency Guarantees

We can define a large variety of consistency guarantees simply by formulating conditions on the various attributes and relations appearing in the abstract executions.

Definition 3.4 (Consistency Guarantee)**.** A *Consistency Guarantee* $\mathcal{P}$ is a predicate or property of an abstract execution A, *i.e.* a statement that is true or false, depending on the particulars of A up to isomorphism. We write $A \models \mathcal{P}$ if $\mathcal{P}$ is true for A.

To define a consistency model, we simply collect all the guarantees needed, and then specify that histories must be justifiable by an abstract execution that satisfies them all:

Definition 3.5 (Correct History)**.** Let $H \in \mathcal{H}$ be a history, and let $\mathcal{P}_1, \ldots \mathcal{P}_n$ be a collection of consistency guarantees. We say that H satisfies the guarantees if it can be extended (by adding visibility and arbitration) to an abstract execution that satisfies them:

$$H \models \mathcal{P}_1 \wedge \cdots \wedge \mathcal{P}_n \quad \overset{\text{def}}{\iff} \quad \exists A \in \mathcal{A} : \mathcal{H}(A) = H \quad \wedge \quad A \models \mathcal{P}_1 \wedge \cdots \wedge \mathcal{P}_n.$$

A consistency model is now simply a combination of consistency guarantees. In the next two chapters, we introduce various consistency models, and the guarantees they are made of, which include:

- *Data type semantics* that give meaning to operations. We discuss how to specify both sequential data types (with traditional single-copy semantics) and replicated data types (which can provide conflict resolution) in Chapter 4.
- *Ordering guarantees* (such as causality) that rule out anomalies caused by reordering of operations. We define a comprehensive list of such guarantees in Chapter 5 (Fig. 5.1).
- *Convergence guarantees* that ensure eventual consistency. We define quiescent consistency in §4.2.1, and eventual visibility in §5.1.

3.4 Background

Operational Consistency Models. An *operational consistency model* provides a reference implementation Π_{ref} whose behaviors provide the specification. In this case, an implementation is deemed correct if it *refines* the specification:

$$\Pi_{\text{impl}} \models \Pi_{\text{ref}} \quad \stackrel{\text{def}}{\iff} \quad \mathcal{H}(\Pi_{\text{impl}}) \subseteq \mathcal{H}(\Pi_{\text{ref}})$$

The main advantage of this approach is that it produces robust specifications and requires no additional formalization work. Also, there is a well-studied methodology (e.g. the use of simulation relations or refinement mappings) to prove correctness.

Operational models for strong consistency (such as linearizability, sequential consistency) are easy to write and understand. However, when considering weaker consistency, operational models can become unwieldy. First, they may become difficult to reason about once they have a large number of small transitions that interleave in complicated ways that are hard to visualize. Second, operational models are often too specific, *i.e.* over-specify the behavior and do not accommodate a large number of different implementations and optimizations.

Axiomatic Consistency Models. An axiomatic model is declarative, *i.e.* it uses logical conditions (traditionally called axioms, but here we call them consistency guarantees) to define the valid behaviors. The advantage is that we can easily customize the consistency model, choosing any number and any combination of consistency guarantees. Perhaps more importantly, the consistency guarantees can have meaning for the user of the system, thus allowing a user to reason about the correctness of their programs at a higher abstraction level.

The main disadvantage of axomatic models is that it is often difficult to fully understand the effect of the axioms. It is easy to make them accidentally too weak or too strong, thus we need to be careful to spend some effort on validating them to ensure they serve the intended purpose. For example, we can prove that an operational model implements them (thus, they are not too strong), or we can use them to prove a client program correct (thus, they are not too weak).

4

Replicated Data Types

To understand eventually consistent data, we need to understand data, and we need to understand conflict resolution. We start this chapter by revisiting *sequential data types*, based on the notions of state, operations, and values. Then, we show how to generalize them to *replicated data types*, which can specify various conflict resolution strategies. The key insight is to interpret state not as a value, but as a graph of prior operations, called the *operation context*.

Replicated data types can specify simple types (e.g. counters, registers) or container types (sets, lists, key-value stores), or even recursive compositions of other replicated data types (such as replicated object stores). This chapter deals with the specification aspect only; how to implement replicated data types is discussed in Chapter 6.

4.1 Basic Definitions

Before diving into the challenges of reading and updating shared data in a distributed system, we start with the basics: what is data, and what does it mean to read and update it? We set the foundation of our formalization by defining sets to represent values and operations.

4.1.1 Values

We define Values to be a set of values. It is meant to contain any piece of data we use in programs, such as constants, tuples, lists, and other recursive data types. For example, we find it reasonable to include

- Natural numbers $\mathbb{N}_0 = \{0, 1, 2 \dots\}$.
- Character strings, *e.g.* "", "a", "boo", ...
- Object identifiers, from some set Objects $= \{x, y, z, \dots\}$.
- Various constants used for specific purposes, *e.g. ok*, *false*, *true*, *undef*, *error*, ... (to be explained when needed).
- Finite sequences of values, such as the empty sequence ϵ, or a nested sequence $[1, 2, [3, \text{``far''}, \text{``boo''}]]$.
- Tagged tuples, *e.g.* $\mathit{Timestamp}(3, 4)$, $\mathit{Message}(\text{“}hi\text{''}, [1, 2]) \dots$.
- Finite sets of values, *e.g.* the empty set $\emptyset$.

4.1.2 Operations

We define Operations to be a set of operations. It is meant to contain operations invoked on data of any data type, whose consistency we may want to discuss. For example, we find it reasonable to include

- A read operation rd
- A write operation wr(v) writing a value $v \in$ Values.
- An increment operation inc.
- Operations add(v) and rem(v) to add/remove a value from a set.
- Operations on objects, fields, or indexed components

  $$x.\mathsf{rd}, \quad y.f.\mathsf{wr}(0), \quad z.2.\mathsf{inc}, \quad \dots$$

  of the general form $v.o$ where $v \in$ Values and $o \in$ Operations.

Values and operations can be used to model any imaginable kind of shared data. However, they are purely syntactic so far. To proceed, we need a better idea of what the operations are supposed to mean, *i.e.* a way to specify the semantics of a data type.

4.2 Sequential Data Types

Conventionally, we reason about data by means of its *state*. We assume that at any point of time, the data is in a particular state. This state determines the return value of read operations, and is modified when we perform update operations.

Definition 4.1 (Sequential Data Type)**.** A *sequential data type* $\mathcal{S}$ is a tuple $(\Sigma, \sigma_0, \delta)$ where $\Sigma \subseteq \mathsf{Values}$ is a set of states with an initial state $\sigma_0 \in \Sigma$, and δ is a function $\mathsf{Operations} \times \Sigma \to (\mathsf{Values} \times \Sigma) \cup \{\nabla\}$.

The function δ describes the effect of operations: $\delta(o, \sigma) = (v, \sigma')$ means that the operation o returns v when called in state σ, and changes the state to σ'.

Blocking. If $\delta(o, \sigma) = \nabla$, where $\nabla \notin \mathsf{Values}$ is a special symbol, it means that the operation o blocks in state σ. For example, a dequeue operation blocks if the queue is empty.

Determinism. We assume determinism, *i.e.* the specification completely describes the behavior of the data type. The reason is that we model nondeterminism due to concurrency explicitly using visibility and arbitration, as we will discuss in §4.3.

Register Example. We define a register data types that stores a value (initially the undefined value *undef*) that can be read by a read operation rd and updated by a write operation $\mathsf{wr}(v)$, as

$$\mathcal{S}_{reg} \stackrel{\text{def}}{=} (\mathsf{Values}, \mathit{undef}, \delta) \quad \text{with } \delta(o, v) = \begin{cases} (v, v) & \text{if } o = \mathsf{rd} \\ (\mathit{ok}, v') & \text{if } o = \mathsf{wr}(v') \\ (\mathit{error}, v) & \text{otherwise} \end{cases}$$

Any operation other than rd or $\mathsf{wr}(v)$ is not applicable to this data type, thus returns the constant *error* which is an element of Values.

State (and initial state)	Oper.	Returned value	Updated state	Condition

Counter $\mathcal{S}_{ctr}$

$n \in \mathbb{N}_0$ (initially 0)	rd	n	same	
	inc	ok	$n+1$	

Register $\mathcal{S}_{reg}$

$v \in$ Values (initially *undef*)	rd	v	same	
	wr(v')	ok	v'	

Key-Value Store $\mathcal{S}_{kvs}$

f : Values $\rightharpoonup_{\text{fin}}$ Values (initially $\emptyset$)	rd(k)	*undef*	same	if $f(k) = \perp$
		f(k)	same	if $f(k) \neq \perp$
	wr(k, v)	ok	$f[k \mapsto v]$	

Set $\mathcal{S}_{set}$

$A \in \mathcal{P}_{\text{fin}}$(Values) (initially $\emptyset$)	rd	A	same	
	add(v)	ok	$A \cup \{v\}$	
	rem(v)	ok	$A \setminus \{v\}$	

Queue $\mathcal{S}_{queue}$

$w \in$ Values* (initially ϵ)	enq(v)	ok	$w \cdot v$	
	deq	v	w'	if $w = v \cdot w'$
		∇		if $w = \epsilon$

Wall $\mathcal{S}_{wall}$

$w \in$ Values* (initially ϵ)	post(v)	ok	$w \cdot v$	
	rd	w	same	

Figure 4.1: Definitions of some sequential data types. Implicitly, we define $\delta(o, \sigma) = (error, \sigma)$ for all the operations that are not listed.

More Examples. We show four more examples of sequential data types in Fig. 4.1, including a counter, a key-value store, a set, a queue, and a *wall*. The latter is what we may use for chat rooms or the like: it is a list of values supporting a read operation rd, and an append operation called $\mathsf{post}(v)$. We adopt this terminology (wall, post) because it is commonly used by social network applications; "append-only list" would be a more descriptive name.

Operation Categories. It is often convenient to categorize operations. Read-only operations are operations which have no side effects, update-only operations are operations that return no information, mixed operations are operations that are neither read-only nor update-only, and blocking operations are operations that may block.

Definition 4.2. Let $\mathcal{S}$ be sequential data type.

$$\mathsf{readonlyops}(\mathcal{S}) \stackrel{\text{def}}{=} \{o \in \mathsf{Operations} \mid \forall \sigma, v, \sigma' \colon \delta(o,\sigma) = (v,\sigma') \Rightarrow \sigma = \sigma'\}$$

$$\mathsf{updateonlyops}(\mathcal{S}) \stackrel{\text{def}}{=} \{o \in \mathsf{Operations} \mid \forall \sigma : \exists \sigma' : \delta(o,\sigma) = (ok,\sigma')\}$$

$$\mathsf{mixedops}(\mathcal{S}) \stackrel{\text{def}}{=} \mathsf{Operations} \setminus (\mathsf{readonlyops}(\mathcal{S}) \cup \mathsf{updateonlyops}(\mathcal{S}))$$

$$\mathsf{blockingops}(\mathcal{S}) \stackrel{\text{def}}{=} \{o \in \mathsf{Operations} \mid \exists \sigma : \delta(o,\sigma) = \nabla\}$$

For example, $\mathsf{deq} \in \mathsf{mixedops}(\mathcal{S}_{queue}) \cap \mathsf{blockingops}(\mathcal{S}_{queue})$.

4.2.1 Quiescent Consistency

We can define *quiescent consistency*, a weak form of eventual consistency, by requiring that in any execution where the updates stop at some point (*i.e.* where there are only finitely many updates), there must exist some state, such that each session converges to that state (*i.e.* all but finitely many operations e in each session $[f]$ see that state).

Definition 4.3. Let $\mathcal{S} = (\Sigma, \sigma_0, \delta)$ be a sequential data type, and let $A = (E, \mathsf{op}, \mathsf{rval}, \mathsf{rb}, \mathsf{ss}, \mathsf{vis}, \mathsf{ar})$ be an abstract execution. Then, define

$$\begin{aligned}
&\textsc{QuiescentConsistency}(\mathcal{S}) \quad \stackrel{\text{def}}{\iff} \\
&\quad \Big(\ \big|\{e \in E \mid \mathsf{op}(e) \notin \mathsf{readonlyops}(\mathcal{S})\}\big| < \infty \quad \Longrightarrow \quad \exists \sigma \in \Sigma : \\
&\qquad \forall [f] \in E/\!\approx_{ss} \colon \ \big|\{e \in [f] \mid \delta(\mathsf{op}(e),\sigma) \neq (\mathsf{rval}(e),\sigma)\ \}\big| < \infty \ \Big)
\end{aligned}$$

Note that we express this guarantee on a per-session basis, rather than for all sessions at once, because there may be infinitely many sessions in an execution. This is important because for many protocols (*e.g.* the counter protocols in Fig. 6.3 and Fig. 6.5), new clients may join throughout the (infinite) execution, and execute a finite number of steps before converging to the stable value.

Limitations. Quiescent consistency is sometimes used as general definition of eventual consistency, such as when Terry et al. [1995] coined the term originally. However, it is too weak of a specification. For one, it does not guarantee anything in a situation where updates keep arriving. Thus it cannot be used for services where there is no guarantee that the updates ever stop. Worse, quiescent consistency does not say anything about what values should be returned *before* convergence. For example, a register could (temporarily) return an arbitrary bogus value that was never written, which may surprise clients and have negative consequences. We introduce a stronger definition of eventual consistency in §5.1, which does not suffer from these limitations.

4.3 Replicated Data Types

Sequential state-based semantics are sufficient to define behaviors of data types as long as the system maintains single-copy-semantics, *i.e.* the illusion that there is but one current state of the data. However, this is only true for strong consistency models such as linearizability and sequential consistency. Under eventual consistency, the user may observe the effects of replication and conflict resolution, and state-based semantics are no longer sufficient to describe such behavior. Our solution is to move from a state-based definition of data types to a more general definition that is based on operations.

Instead of defining the current state as a value, we think of the current state as the *graph of prior operations*, called the *operation context*.

Then, the effect and return value of an operation are determined by its context rather than by the current state.

Definition 4.4 (Operation Context)**.** An *operation context* is a finite event graph $C = (E, \mathsf{op}, \mathsf{vis}, \mathsf{ar})$ where $\mathsf{op} : E \to \mathsf{Operations}$ describes the operation of each event, vis is an acyclic relation representing visibility among the elements of E, and ar is a total order representing arbitration of the elements in E. We let $\mathcal{C}$ be the set of all operation contexts.

We now specify a replicated data type by a function $\mathcal{F}_\tau(o, C)$, which takes as a parameter the operation o and an operation context C. This generalizes the specification of a sequential data type using a function $\delta(o, \sigma)$ that takes as a parameter the operation o and the current state σ (Definition 4.1).

Definition 4.5 (Replicated Data Type)**.** A *replicated data type* $\mathcal{F}$ is a function $\mathsf{Operations} \times \mathcal{C} \to \mathsf{Values}$ that, given an operation o and an operation context C, specifies the expected return value $\mathcal{F}(o, C)$ to be used when performing o in context C, and which does not depend on the events, *i.e.* is the same for isomorphic (as in Definition 2.2) contexts: $C \simeq C' \Rightarrow \mathcal{F}(o, C) = \mathcal{F}(o, C')$ for all o, C, C'.

The events in E capture what prior operations are visible to the operation that is about to be performed. E is always finite because no operation can happen after an infinite number of other operations. The total order ar is called the *arbitration order*; it is used to resolve conflicts between prior operations in a deterministic way. The visibility order vis represents the mutual visibility of the operations in E, and is needed if the effect of an operation in E depends on it. To clarify how these all work together, we discuss examples in the next few sections.

Determinism. Note that $\mathcal{F}$ is deterministic: two events that perform the same operation in the same context produce the same return values. This is necessary to ensure convergence, as we will see when proving quiescent consistency in §5.4. Of course, a call to a replicated data type is still highly nondeterministic due to unpredictable scheduling and timing of message delivery. Working with a deterministic specification $\mathcal{F}$ simply means that all the non-determinism arising due to scheduling and message delivery is already captured by vis and ar that are passed as arguments to $\mathcal{F}$.

4.3.1 Replicated Counter

To generalize the sequential counter $\mathcal{S}_{ctr}$ (Figure 4.1, top) to a replicated counter $\mathcal{F}_{ctr}$, we specify that the number returned by a read operation is exactly the number of increment operations in the context:

$$\mathcal{F}_{ctr}(\mathsf{rd}, (E, \mathsf{op}, \mathsf{vis}, \mathsf{ar})) = \left|\{e' \in E \mid \mathsf{op}(e') = \mathsf{inc}\}\right|$$

This is a simple example: all operations commute, and no conflict resolution is needed. Thus, the value returned by $\mathcal{F}_{ctr}$ depends only on E and op, but not on vis or ar.

4.3.2 Replicated Registers

To generalize the sequential register $\mathcal{S}_{reg}$, we need to decide how to resolve conflicts between concurrent write operations, because write operations do not commute. There are multiple options.

Last-Writer-Wins Register. The simplest and most common solution is to use some sort of timestamp to determine the order of the writes. In our formalization, this order is called the arbitration order. To specify the last-writer-wins register, we say that a read sees the last write in the visible context (where last means last in terms of arbitration order), or *undef* if there is no write. We use the notation $\mathsf{writes}(E) \stackrel{\text{def}}{=} \{e \in E \mid \mathsf{op}(e) = \mathsf{wr}(v) \text{ for some } v\}$, and specify:

$$\mathcal{F}_{reg}(\mathsf{rd}, (E, \mathsf{op}, \mathsf{vis}, \mathsf{ar})) = \begin{cases} \mathit{undef} & \text{if } \mathsf{writes}(E) = \emptyset \\ v & \text{if } \mathsf{op}(\max_{\mathsf{ar}} \mathsf{writes}(E)) = \mathsf{wr}(v) \end{cases}$$

Since ar is always a total order and E is finite, a maximal write $\max_{\mathsf{ar}} \mathsf{writes}(E)$ is uniquely determined if there is any write at all, *i.e.* $\mathsf{writes}(E) \neq \emptyset$. This is an example of arbitration-based conflict resolution — operations do not commute, but we use the arbitration order to order them consistently.

Multi-Value Register. Another conflict resolution strategy is to report conflicting writes to the user, and rely on some application-dependent resolution. We can define a multi-value-register [G. DeCandia et al., 2007] where read operations do not return an individual value, but a

set of values, one for each write preceding the read that has not been overwritten by some other write:

$$\mathcal{F}_{mvr}(\mathsf{rd}, (E, \mathsf{op}, \mathsf{vis}, \mathsf{ar})) = \\ \{v \mid \exists e \in E : \mathsf{op}(e) = \mathsf{wr}(v) \text{ and } \forall e' \in \mathsf{writes}(E) : e \overset{\mathsf{vis}}{\not\longrightarrow} e'\}$$

Note that to determine which writes are overwritten by a write, we use the visibility relation between writes.

4.3.3 Standard Conflict Resolution

For any sequential data type $\mathcal{S}$ without blocking operations, we can define a corresponding replicated data type $\mathcal{F}$ by specifying that the effect of the updates in the context is determined by applying them sequentially to the initial state, in the order specified by the arbitration order. We call this the *standard conflict resolution*.

Definition 4.6 (Standard Conflict Resolution)**.** Let $\mathcal{S}_\tau = (\Sigma, \sigma_0, \delta)$ be a sequential data type without blocking operations. Let δ_{rval} and δ_Σ be the two components of δ (meaning that $\delta(o, \sigma) = (\delta_{\mathsf{rval}}(o, \sigma), \delta_\Sigma(o, \sigma))$ for all o, σ). Then we define the replicated data type $\mathcal{F}\langle\mathcal{S}_\tau\rangle$ as

$$\mathcal{F}\langle\mathcal{S}_\tau\rangle(o, (E, \mathsf{op}, \mathsf{vis}, \mathsf{ar})) \overset{\text{def}}{=} \delta_{\mathsf{rval}}(\mathsf{foldr}(\sigma_0, \delta_\Sigma, E.\mathsf{sort}(\mathsf{ar})))$$

where sort and foldr are the operators we defined in §2.1.1 on p. 20.

In fact, the replicated data type for the counter and the last-writer-wins register defined above is equivalent to the standard conflict resolution: $\mathcal{F}_{ctr} = \mathcal{F}\langle\mathcal{S}_{ctr}\rangle$ and $\mathcal{F}_{reg} = \mathcal{F}\langle\mathcal{S}_{reg}\rangle$.

For the key-value store, the set, and the wall data types (Fig. 4.1) we define replicated data types using standard conflict resolution: $\mathcal{F}_{kvs} = \mathcal{F}\langle\mathcal{S}_{kvs}\rangle$, $\mathcal{F}_{set} = \mathcal{F}\langle\mathcal{S}_{set}\rangle$, and $\mathcal{F}_{wall} = \mathcal{F}\langle\mathcal{S}_{wall}\rangle$.

4.3.4 Replicated Sets

For sets, the interesting question is how to handle conflicts between a concurrent add and remove of the same element [Bieniusa et al., 2012b]. Using standard conflict resolution $\mathcal{F}\langle\mathcal{S}_{set}\rangle$ means the last operation

wins. Sometimes, that is however not what we want, for example if we are concerned about losing data: a remove operation may cancel out an add operation even if that add operation is not visible at the time the remove operation is performed.

Add-Wins Set. For the add-wins-set (also known as observed-remove set [Shapiro et al., 2011b]), we specify that an add always wins against a concurrent remove, by requiring that an element is in the set if it has been added by some operation that was not superseded by a later remove operation:

$$\mathcal{F}_{awset}(\mathsf{contains}(v), (E, \mathsf{op}, \mathsf{vis}, \mathsf{ar})) = true \quad \overset{\text{def}}{\iff}$$
$$\exists e \in E : \ \mathsf{op}(e) = \mathsf{add}(v) \ \wedge \ \neg(\exists e' \in E : \mathsf{op}(e') = \mathsf{rem}(v) \wedge e \xrightarrow{\mathsf{vis}} e')$$

4.3.5 Replicated Object Stores

We can compose multiple replicated objects into an *object store*. At the specification level, this amounts to defining a type for each value that represents an object. We define a *typing* to be a partial function $\mathcal{D}$ that maps values v to a replicated data type $\mathcal{D}(v)$.

For example, consider an online chat application, with an object chat with fields (1) wall, a list containing posted entries, and (2) access, a set containing the people who have permission to view them. We can define the corresponding typing as

$$\mathcal{D}(v) = \begin{cases} \mathcal{F}_{wall} & \text{if } v = \mathsf{chat.wall} \\ \mathcal{F}_{set} & \text{if } v = \mathsf{chat.access} \\ \bot & \text{otherwise} \end{cases}$$

The typing then determines the replicated data type of the store.

Definition 4.7. For a typing $\mathcal{D}$, define the replicated object store $\mathcal{F}_{\mathcal{D}}$:

$$\mathcal{F}_{\mathcal{D}}(o, (E, \mathsf{op}, \mathsf{vis}, \mathsf{ar})) =$$
$$\begin{cases} \mathcal{D}(v)(o', (E_v, \mathsf{op}_v, \mathsf{vis}|_{E_v}, \mathsf{ar}|_{E_v})) & \text{if } o = v.o' \text{ and } \mathcal{F}_{\mathcal{D}(v)} \neq \bot \\ \mathit{error} & \text{otherwise} \end{cases}$$

where $E_v = \{e \in E \mid \exists o : \mathsf{op}(e) = v.o\}$, and $\mathsf{op}(e) = v.\mathsf{op}_v(e)$.

The object composition we defined here is an *independent* composition: there is no interaction between the replicated objects. Compositions are not always independent in practice. For example, if the chat object supports a delete operation del, then invoking chat.del should delete chat.wall and chat.access as well, thus a subsequent chat.posts.rd operation should return an empty sequence or some error message.

4.3.6 Quiescent Consistency

We now generalize operation categories and quiescent consistency (§4.2.1) from a sequential data type $\mathcal{S}$ to a replicated data type $\mathcal{F}$. Update-only and blocking operations are easy to generalize. However, determining read-only operations is a bit more tricky: all operations change the context, thus we need to determine whether they do so in a non-observable way. To this end, we define an operation o to be a read-only operation if for all observer operations o' the return value does not change when we remove o from the context:

$$o \in \mathsf{readonlyops}(\mathcal{F}) \overset{\text{def}}{\iff} \forall o' \in \mathsf{Operations} : \forall C = (E, \dots) \in \mathcal{C} : \forall e \in E : \\ \mathsf{op}(e) = o \quad \Rightarrow \quad \mathcal{F}(o', C) = \mathcal{F}(o', C|_{E \setminus \{e\}, \mathsf{op}, \mathsf{vis}, \mathsf{ar}})$$

We can straightforwardly generalize quiescent consistency now:

$$\textsc{QuiescentConsistency}(\mathcal{F}) \quad \overset{\text{def}}{\iff} \\ \Big(\ \big|\{e \in E \mid \mathsf{op}(e) \notin \mathsf{readonlyops}(\mathcal{F})\}\big| < \infty \quad \Longrightarrow \quad \exists C \in \mathcal{C} : \\ \forall [f] \in E/{\approx_{ss}} : \ \big|\{e \in [f] \mid \mathcal{F}(\mathsf{op}(e), C) \neq \mathsf{rval}(e) \ \}\big| < \infty \ \Big)$$

4.4 Return Value Consistency

Finally, we define consistency of return values, as a predicate on abstract executions, for a given replicated data type $\mathcal{F}$.

Definition 4.8. For a replicated data type $\mathcal{F}$, we define the return value consistency guarantee as

$$\textsc{RVal}(\mathcal{F}) \quad \overset{\text{def}}{=} \quad \forall e \in E : \quad \mathsf{rval}(e) = \mathcal{F}(\mathsf{op}(e), \mathsf{context}(A, e))$$

where context is defined as follows:

Definition 4.9. Let $A = (E, \mathsf{op}, \mathsf{rval}, \mathsf{rb}, \mathsf{ss}, \mathsf{vis}, \mathsf{ar})$ be an abstract execution containing an event $e \in E$. Then

$$\mathsf{context}(A, e) \stackrel{\text{def}}{=} A|_{\mathsf{vis}^{-1}(e),\mathsf{op},\mathsf{vis},\mathsf{ar}}$$

Note that $\mathsf{context}(A, e)$ is always an operation context: $\mathsf{vis}^{-1}(e)$ is finite because vis is natural (condition (a2) on p. 35), and the restrictions of vis and ar are acyclic and total, respectively, because acyclicity and totality are preserved under restrictions (Lemma 2.3).

5

Consistency

The level of consistency afforded by replicated data types and quiescent consistency is usually not strong enough to write correct programs, because it still permits many "anomalies" that can surprise programmers.

Boss Example [Lloyd et al., 2011]. Consider an online chat, and consider that a user wants to post the message "time for a new job!" to the wall, but first removes the boss from the access list so that the boss does not see the message. With an object store as defined in §4.3.5, the user would perform these two operations in a session:

chat.access.rem(*boss*); chat.wall.post("Time for a new job!");

Since the boss is removed before the message is posted, the user has a reasonable expectation that the boss is not able to see the message. However, this is not always guaranteed — the second operation may take effect on the computer of the boss before the first one does.

Ordering and Atomicity. In general, a programmer can prevent such problems in various ways, depending on the consistency model and the features of the system. *Ordering guarantees* can ensure the order of operations is preserved under some conditions. *Transactions* can ensure that operation sequences are atomic, *i.e.* never become visible sepa-

rately. *Synchronization* operations, such as fence or flush operations, can enforce ordering selectively.

We discuss the ordering guarantees provided by common consistency models in the remainder of the chapter. Transactions and synchronization operations are out of scope of this tutorial, but are discussed in Burckhardt et al. [2013].

Summary. For easier reference, we start with a summary of the consistency models and ordering guarantees, without much explanation upfront, and provide discussion and examples later. Figure 5.1 (top half) shows the most common consistency models, ordered from strong to weak. Each model is parameterized by the replicated data type $\mathcal{F}$, and consists of several consistency guarantees. These guarantees are conjunctions of predicates that include return value consistency $\text{RVAL}(\mathcal{F})$ as defined in §4.4, the ordering guarantees shown in Figure 5.1 (bottom half), and the eventual-visibility guarantee, which we define in §5.1 below.

Overview. We proceed by introducing a series of successively stronger models (having already started with quiescent consistency in the previous chapter). For each consistency model, we demonstrate an anomaly that it permits, and how it can be fixed by adding more guarantees. We thus finally arrive at linearizability, the strongest model. After that, we conclude the chapter with proving the hierarchy of the models.

5.1 Basic Eventual Consistency

Under eventual consistency, convergence is expressed as *eventual visibility*: a completed operation e must eventually become visible to all sessions. We express this guarantee by requiring that in each session, almost all operations that start after e has returned (*i.e.* all but finitely many) must see e.

Definition 5.1 (Eventual Visibility)**.**

$$\text{EVENTUALVISIBILITY} \quad \stackrel{\text{def}}{\iff} \quad \forall e \in E : \forall [f] \in E/\approx_{ss} : \\ \left|\{e' \in [f] \mid (e \xrightarrow{\mathsf{rb}} e') \wedge (e \not\xrightarrow{\mathsf{vis}} e')\}\right| < \infty$$

Consistency Models

$\textsc{Linearizability}(\mathcal{F}) \stackrel{\text{def}}{=} \textsc{SingleOrder} \wedge \textsc{Realtime} \wedge \textsc{RVal}(\mathcal{F})$

$\textsc{SequentialConsistency}(\mathcal{F}) \stackrel{\text{def}}{=} \textsc{SingleOrder} \wedge \textsc{ReadMyWrites} \wedge \textsc{RVal}(\mathcal{F})$

$\textsc{CausalConsistency}(\mathcal{F}) \stackrel{\text{def}}{=} \textsc{EventualVisibility} \wedge \textsc{Causality} \wedge \textsc{RVal}(\mathcal{F})$

$\textsc{BasicEventualConsistency}(\mathcal{F}) \stackrel{\text{def}}{=} \textsc{EventualVisibility} \wedge \textsc{NoCircularCausality} \wedge \textsc{RVal}(\mathcal{F})$

$\textsc{QuiescentConsistency}(\mathcal{F}) \stackrel{\text{def}}{=}$ (see page 49)

Ordering Guarantees

$\textsc{ReadMyWrites} \stackrel{\text{def}}{=} (\mathsf{so} \subseteq \mathsf{vis})$

$\textsc{MonotonicReads} \stackrel{\text{def}}{=} (\mathsf{vis}\,;\mathsf{so}) \subseteq \mathsf{vis}$

$\textsc{ConsistentPrefix} \stackrel{\text{def}}{=} (\mathsf{ar}\,;(\mathsf{vis} \cap \neg\mathsf{ss})) \subseteq \mathsf{vis}$

$\textsc{NoCircularCausality} \stackrel{\text{def}}{=} \mathsf{acyclic}(\mathsf{hb})$

$\textsc{CausalVisibility} \stackrel{\text{def}}{=} (\mathsf{hb} \subseteq \mathsf{vis})$

$\textsc{CausalArbitration} \stackrel{\text{def}}{=} (\mathsf{hb} \subseteq \mathsf{ar})$

$\textsc{Causality} \stackrel{\text{def}}{=} \textsc{CausalVisibility} \wedge \textsc{CausalArbitration}$

$\textsc{SingleOrder} \stackrel{\text{def}}{=} \exists E' \subseteq \mathsf{rval}^{-1}(\nabla) : \mathsf{vis} = \mathsf{ar} \setminus (E' \times E)$

$\textsc{Realtime} \stackrel{\text{def}}{=} \mathsf{rb} \subseteq \mathsf{ar}$

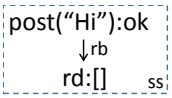
Figure 5.1: Overview of the definitions for commonly used consistency models and ordering guarantees. All the formulas shown above are predicates over an abstract execution $A = (E, \mathsf{op}, \mathsf{rval}, \mathsf{rb}, \mathsf{ss}, \mathsf{vis}, \mathsf{ar})$, with session order $\mathsf{so} \stackrel{\text{def}}{=} \mathsf{rb} \cap \mathsf{ss}$ and happens-before order $\mathsf{hb} \stackrel{\text{def}}{=} (\mathsf{so} \cup \mathsf{vis})^+$.

Eventual visibility is sufficiently strong to write correct client programs. For example, an eventually visible replicated counter F_{ctr} can be used to reliably count events in a distributed systems. However, if eventual visibility is our only guarantee, we may have to deal with a number of confusing anomalies. We now discuss these anomalies, and introduce more guarantees that can be used to eliminate them.

5.1.1 Session Guarantees

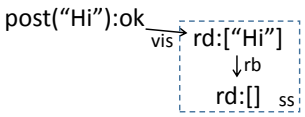
When a user issues several operations as part of the same session, there is often an expectation that the order in which they are issued is preserved. We define the session order to be the relation $\mathsf{so} \stackrel{\text{def}}{=} \mathsf{rb} \cap \mathsf{ss}$. Note that within each session, so is an enumeration of the operations in the session because of condition (h7) on p. 32.

Read My Writes



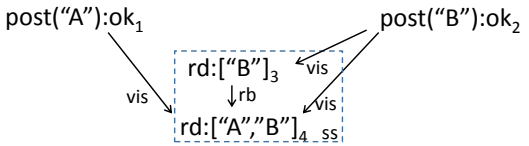
Consider a user who posts a message "Hi" to the wall, and then reads the wall: clearly, they expect to see their own message there. However, eventual consistency alone does not enforce it and allows the anomaly shown on the right. The guarantee ReadMyWrites (Fig. 5.1) ensures that for any two events $x \xrightarrow{\mathsf{so}} y$, we have $x \xrightarrow{\mathsf{vis}} y$, and thus prevents this anomaly.

Monotonic Reads



When issuing more than one read in the same session, we would expect that we see more operations over time, not fewer. However, eventual consistency alone does not enforce it and allows the anomaly shown on the right, where the second read in the session on the right does not see the post of "Hi", even though the first one did. The guarantee MonotonicReads (Fig. 5.1) ensures that if $x \xrightarrow{\mathsf{vis}} y$ and $y \xrightarrow{\mathsf{so}} z$, then also $x \xrightarrow{\mathsf{vis}} z$, which prevents this anomaly.

Consistent Prefix



When issuing more than one read in the same session, we may expect that operations become visible in arbitration order (e.g. timestamp order). However, eventual consistency alone does not enforce it and allows the anomaly shown above on the right: the operation that posts "A" becomes visible after the post of "B", but precedes it in arbitration order (as indicated by the numeric subscripts which represent timestamps); thus, "A" appears before "B" in the returned value. This is potentially confusing, since in the sequential semantics, a post appends only at the end, yet here it looks like someone posted at the beginning. The guarantee ConsistentPrefix (Fig. 5.1) ensures that whenever we see an operation from a different session, we also see all operations that precede it in arbitration order, which prevents this anomaly.

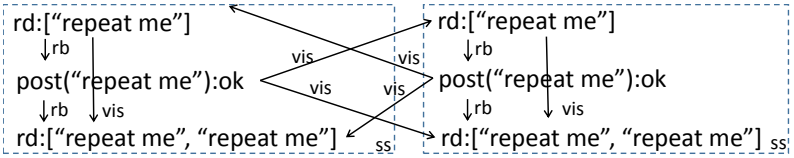
ConsistentPrefix implies that the remote operations contained in each operation context form a contiguous prefix of the sequence of all operations (ordered by arbitration order), hence the name.

5.1.2 Causality Guarantees

It is common in distributed systems to use the notion of *happens-before* for events that have a potentially causal relationship. If two operations happen in the same session, they may be causally related: the program or the user may have decided to issue an operation based on values returned by an earlier operation. Also, if an operation A is visible to an operation B, the value returned by B may depend on A. Therefore, we define the happens-before relation hb to be the transitive union of session order and visibility: $\mathsf{hb} \stackrel{\text{def}}{=} (\mathsf{so} \cup \mathsf{vis})^+$.

Note that the happens-before relation does not indicate true causality, but only potential causality. Distinguishing the two can be surprisingly subtle, and is beyond the scope of this tutorial.

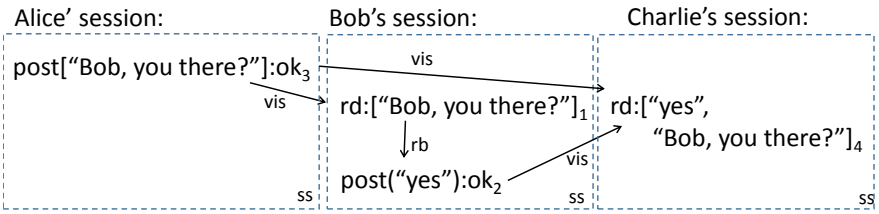
Circular Causality



Weird things can happen if the happens-before order is allowed to have cycles. Consider this anomaly involving two users. The first user reads the wall and sees a single post saying "repeat me" (perhaps also including some promises about good things that will happen to you if you abide), and obeys by posting the same string. Then, it reads the wall and sees both the original post, and the repeated post. The second user goes through the exact same experience. Nothing seems wrong from the perspective of each individual user. However, something is definitely wrong: where did the original post come from? It spookily materialized out of thin air. This is an example of *circular causality*.

The guarantee NoCircularCausality (Fig. 5.1) ensures that the happens-before relation contains no cycles, which prevents circular causality, and in particular rules out this anomaly. Since circular causality is usually not an issue in the consistency algorithms we study here (although it sometimes is when considering compiler optimizations for shared memory), we include it as a guarantee in our baseline, the basic eventual consistency model (Fig. 5.1).

Causal Arbitration

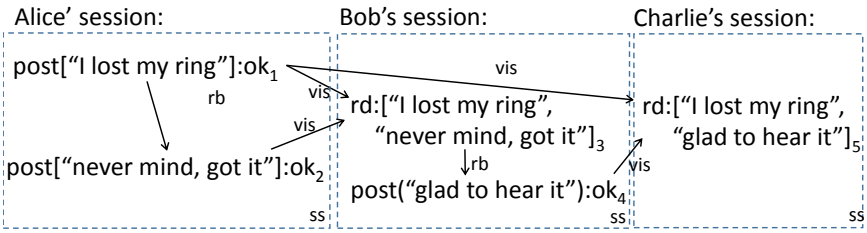


If the happens-before order is not the same as the arbitration order, it can get confusing. Consider this anomaly: Alice posts the string "Bob, you there?". Bob reads the wall, sees Alice's post, and replies "yes". Charlie is a quiet observer and sees both posts. However, the arbitra-

tion order (indicated by numeric subscripts representing timestamps) orders Bob's reply before Alice's question, so Charlie sees them in the "wrong" order.

The guarantee CausalArbitration (Fig. 5.1) ensures that if $x \xrightarrow{\mathsf{hb}} y$, then $x \xrightarrow{\mathsf{ar}} y$, which prevents this anomaly. In practice, causal arbitration can be ensured by obtaining timestamps from sufficiently well synchronized physical clocks, from logical clocks, or from a single arbitration node.

Causal Visibility



If the happens-before order does not enforce visibility, it can get confusing. Consider this anomaly, which is a standard example for causality [Lloyd et al., 2014]: Alice posts the string "I lost my ring", and quickly follows it up with a second post "never mind, got it". Bob reads the wall, sees both posts, and replies "glad to hear it". Charlie is a quiet observer and sees Bob's post and Alice's first post, but not Alice's second post. Thus, it appears to him that Bob is happy about Alice's loss.

The guarantee CausalVisibility (Fig. 5.1) ensures that if $x \xrightarrow{\mathsf{hb}} y$, then $x \xrightarrow{\mathsf{vis}} y$, which prevents this anomaly. In practice, causal visibility can be enforced by tracking dependencies, or by using vector clocks, or by using ordered broadcast.

5.2 Causal Consistency

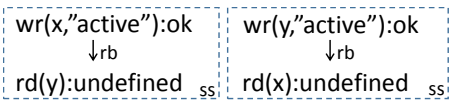
We define causal consistency (Fig. 5.1) as the combination of causal arbitration and causal visibility. Of the models listed in the top half of Fig. 5.1, causal consistency is the strongest one that can be imple-

mented in such a way as to remain available under partitions.[1] Causal consistency implies all the ordering guarantees discussed earlier, except consistent prefix:

Lemma 5.1. If $A \models \textsc{CausalConsistency}(\mathcal{F})$, then $A \models \textsc{ReadMyWrites} \wedge \textsc{MonotonicReads} \wedge \textsc{NoCircularCausality} \wedge \textsc{CausalArbitration} \wedge \textsc{CausalVisibility}$.

The combination $\textsc{CausalConsistency}(\mathcal{F}) \wedge \textsc{ConsistentPrefix}$ is slightly stronger than $\textsc{CausalConsistency}(\mathcal{F})$ alone, and can also be implemented while fully available under partitions (see the protocol $\mathsf{BufferedSequencer}\langle\mathcal{F}\rangle$ of Fig. 6.12 and Theorem 10.14).

5.2.1 Dekker Anomaly



The *Dekker* anomaly (on the right) is possible under causal consistency. Two sessions each write "active" to two different locations, then read the location that the other one is writing. Neither one, however, sees the write by the other one. The Dekker anomaly is ruled out by strong models like sequential consistency or linearizability, as discussed in the next section.

5.3 Strong Models

The last two models we discuss are linearizability and sequential consistency. We consider both of them to be *strong* models, because they ensure that there is a single global order of operations that determines both visibility and arbitration. This is expressed by the guarantee SingleOrder (Fig. 5.1, bottom) which requires that arbitration and visibility are the same except for some subset $E' \subseteq \mathsf{rval}^{-1}(\nabla)$ that represents incomplete operations that are not visible to any other

[1] To support this statement we (1) reproduce a causally consistent protocol (for arbitrary $\mathcal{F}$ without blocking operations) that is fully available under partitions in Fig. 6.13, and (2) prove in theorem 9.1 that there exist no fully available, sequentially consistent protocol implementations for arbitrary $\mathcal{F}$.

operations. In particular, SingleOrder implies

$$\forall e, e' \in E: \quad [\ e \xrightarrow{\mathsf{vis}} e' \quad \Rightarrow \quad e \xrightarrow{\mathsf{ar}} e'\ ] \ \wedge \\ [e \xrightarrow{\mathsf{ar}} e' \quad \wedge \quad (\mathsf{rval}(e) \neq \nabla) \quad \Rightarrow \quad e \xrightarrow{\mathsf{vis}} e']$$

For example, if we have an arbitration order that corresponds to timestamps, then this guarantee implies that an operation can only see operations with earlier timestamps, and must see all complete operations with earlier timestamps. This is the essence of strong consistency.

Besides SingleOrder, linearizability and sequential consistency both require one additional ordering guarantee.

- Linearizability requires Realtime. This guarantee states that the returns-before partial order (§3.1) must be consistent with the arbitration order: $\forall e, e' \in E: \ e \xrightarrow{\mathsf{rb}} e' \ \Rightarrow \ e \xrightarrow{\mathsf{ar}} e'$.

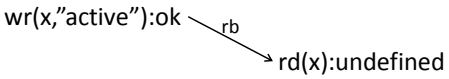
  For example, if arbitration is based on timestamps, it means that if e returns before e', the timestamp of e must be before the timestamp of e'. While this is always true for perfectly synchronized physical timestamps, it may not be true for imperfectly synchronized physical timestamps, or for logical timestamps.

- Sequential consistency requires ReadMyWrites. This guarantee ensures that the session order of operations is respected: For two operations in the same session, their order implies visibility:

$$\forall e, e' \in E: \quad e \approx_{\mathsf{ss}} e' \ \wedge \ e \xrightarrow{\mathsf{rb}} e' \quad \Rightarrow \quad e \xrightarrow{\mathsf{ar}} e'$$

5.3.1 Sequential Consistency versus Linearizability

Sequential consistency is slightly weaker than linearizability. For example, the abstract execution on the right is not linearizable,



but sequentially consistent: we can order the read before the write in visibility and arbitration, and this ordering is consistent with the session order, but is not consistent with the returns-before order.

To observe the difference between sequential consistency and linearizability, clients must be able to communicate over a side channel.

For example, consider the abstract execution shown above, and suppose some user A does the write and some user B does the read. After the write returns, user A may call user B on the phone before user B starts the read operation. Thus, the users can observe that the returns-before order fails to guarantee visibility. For that reason, our REALTIME condition is sometimes called *external* consistency.

We show two examples of implementations that are sequentially consistent, but not linearizable, in §1.3.2 and §6.5.1.

5.3.2 Dekker Test

A common way to test whether a protocol is sequentially consistent is to run the following Dekker test.

Program A:

```
1 x := "active";
2 if (y = undef)
3     print "A wins";
```

Program B:

```
1 y := "active";
2 if (x = undef)
3     print "B wins";
```

When running the program above under sequential consistency, it should never print both "A wins" and "B wins":

Lemma 5.2. Let A be the abstract execution of the Dekker anomaly shown in §5.2.1. Then $A \not\models \text{SEQUENTIALCONSISTENCY}(\mathcal{F}_{kvs})$.

Proof. Let $E = \{w_x, w_y, r_y, r_x\}$ where w_x, w_y are the writes to x and y, respectively, and r_y, r_x are the reads from y, x, respectively. Since ar is total, it must order w_x and w_y. We can assume $w_x \xrightarrow{\mathsf{ar}} w_y$ without loss of generality (otherwise, proceed symmetrically). Since $w_y \xrightarrow{\mathsf{ro}} r_x$, we have $w_y \xrightarrow{\mathsf{vis}} r_x$ (by READMYWRITES), and thus $w_y \xrightarrow{\mathsf{ar}} r_x$ (by SINGLEORDER and $\mathsf{rval}(w_y) \neq \nabla$), and thus $w_x \xrightarrow{\mathsf{ar}} r_x$ (by transitivity of ar), and thus $w_x \xrightarrow{\mathsf{vis}} r_x$ (by SINGLEORDER and $\mathsf{rval}(w_x) \neq \nabla$). But then $\text{RVAL}(\mathcal{F}_{kvs})$ requires that $\mathsf{rval}(r_x) =$ "active", contradicting $\mathsf{rval}(r_x) = \mathit{undef}$. □

5.4 Hierarchy of Models

It is customary to compare the *strength* of consistency models. We say that a model $\mathcal{P}_1$ is *stronger* than a model $\mathcal{P}_2$, written $\mathcal{P}_1 > \mathcal{P}_2$, if $H \models \mathcal{P}_1$ implies $H \models \mathcal{P}_2$, for all histories $H \in \mathcal{H}$.

Proposition 5.1. For any replicated data type $\mathcal{F}$, the following hierarchy between consistency models holds:

$$\begin{aligned} &\textsc{Linearizability}(\mathcal{F}) \\ >\ &\textsc{SequentialConsistency}(\mathcal{F}) \\ >\ &\textsc{CausalConsistency}(\mathcal{F}) \\ >\ &\textsc{BasicEventualConsistency}(\mathcal{F}) \\ >\ &\textsc{QuiescentConsistency}(\mathcal{F}) \end{aligned}$$

Proof. Let $A = (E, \mathsf{op}, \mathsf{rval}, \mathsf{rb}, \mathsf{ss}, \mathsf{vis}, \mathsf{ar})$ be an abstract execution.

$\Big[$ (SingleOrder $\wedge$ Realtime) $\Rightarrow$ ReadMyWrites. $\Big]$ Let $e \xrightarrow{\mathsf{so}} e'$. Then, by definition of so, $e \xrightarrow{\mathsf{rb}} e'$. By Realtime, this implies $e \xrightarrow{\mathsf{ar}} e'$, and by condition (h5) on p. 32, it implies $\mathsf{rval}(e) \neq \nabla$. Thus, by SingleOrder, we get $e \xrightarrow{\mathsf{vis}} e'$, which proves ReadMyWrites.

$\Big[$ (SingleOrder $\wedge$ ReadMyWrites) $\Rightarrow$ CausalArbitration. $\Big]$ $\mathsf{so} \subseteq \mathsf{vis}$ by ReadMyWrites; and $\mathsf{vis} \subseteq \mathsf{ar}$ by SingleOrder. Thus, $\mathsf{hb} = (\mathsf{so} \cup \mathsf{vis})^+ \subseteq (\mathsf{vis} \cup \mathsf{vis})^+ = \mathsf{vis}^+ \subseteq \mathsf{ar}^+ = \mathsf{ar}$.

$\Big[$ (SingleOrder $\wedge$ ReadMyWrites) $\Rightarrow$ CausalVisibility. $\Big]$ As in the previous case, we deduce $\mathsf{hb} \subseteq \mathsf{vis}^+$. It remains to show that $\mathsf{vis}^+ \subseteq \mathsf{vis}$. Let E' be the set of incomplete operations as in SingleOrder; then, $x \xrightarrow{\mathsf{vis}} y \xrightarrow{\mathsf{vis}} z$ implies $x, y \notin E'$ and thus $x \xrightarrow{\mathsf{ar}} y \xrightarrow{\mathsf{ar}} z$. Since ar is transitive, we get $x \xrightarrow{\mathsf{ar}} z$, and thus $x \xrightarrow{\mathsf{vis}} z$.

$\Big[$ SingleOrder $\Rightarrow$ EventualVisibility. $\Big]$ Given an arbitrary event $e \in E$ and a session $[f] \in E/\approx_{ss}$, we need to show

$$\left|\{e' \in [f] \mid (e \xrightarrow{\mathsf{rb}} e') \wedge (e \not\xrightarrow{\mathsf{vis}} e')\}\right| < \infty. \tag{5.1}$$

If $\mathsf{rval}(e) = \nabla$, (5.1) follows immediately because of condition (h5) on p. 32. Otherwise, if $[f]$ contains an e' such that $\mathsf{rval}(e') = \nabla$, it

implies that $[f]$ is finite (by condition (h5) and condition (h7)) and (5.1) follows. Otherwise, define the set $A \stackrel{\text{def}}{=} \mathsf{vis}^{-1}(e)$, which must be finite because vis is natural. Then, for any $e' \in [f] \setminus A$, we know that $e' \not\xrightarrow{\mathsf{vis}} e$, and thus $e' \not\xrightarrow{\mathsf{ar}} e$ (by SINGLEORDER and because $\mathsf{rval}(e') \neq \nabla$), and thus $e \xrightarrow{\mathsf{ar}} e'$ (by totality of ar), and thus $e \xrightarrow{\mathsf{vis}} e'$ (by SINGLEORDER and because $\mathsf{rval}(e) \neq \nabla$). Therefore, $e \xrightarrow{\mathsf{vis}} e'$ for almost all $e' \in [f]$ (all but the finitely many ones in A), which implies (5.1).

$\Big[$ CAUSALITY $\Rightarrow$ NOCIRCULARCAUSALITY. $\Big]$ Since $\mathsf{hb} \subseteq \mathsf{vis}$, any cycle on hb would imply a cycle on vis which contradicts Def. 3.3.

$\Big[$ EVENTUALVISIBILITY$\wedge$RVAL$(\mathcal{F})$ $\Rightarrow$ QUIESCENTCONSISTENCY$(\mathcal{F})$. $\Big]$ Let $U = \{e \in E \mid \mathsf{op}(e) \notin \mathsf{readonlyops}(\mathcal{F})\}$ be the set of all updates. If U is infinite, QUIESCENTCONSISTENCY$(\mathcal{F})$ is vacuously satisfied. Otherwise, let $C \stackrel{\text{def}}{=} A|_{U,\mathsf{op},\mathsf{vis},\mathsf{ar}}$. Given $[f] \in E/\approx_{ss}$, we need to show

$$\big|\{e \in [f] \mid \mathcal{F}(\mathsf{op}(e), C) \neq \mathsf{rval}(e)\ \}\big| < \infty. \tag{5.2}$$

For each update event u, there exists a finite set E'_u such that u is visible for all events in $[f] \setminus E'_u$ (by EVENTUALVISIBILITY). Then $E' \stackrel{\text{def}}{=} \bigcup_{u \in U} E'_u$ is also finite, and for all $e \in [f] \setminus E'$, and all $u \in U$, we have $u \xrightarrow{\mathsf{vis}} e$. We prove two auxiliary claims:

1. For all operation contexts $C'' = A|_{E'',\mathsf{op},\mathsf{vis},\mathsf{ar}}$ with $U \subseteq E'' \subseteq E$ and all operations $r \in \mathsf{readonlyops}(\mathcal{F})$, we have $\mathcal{F}(r, C'') = \mathcal{F}(r, C)$. Proof: by induction over $d \stackrel{\text{def}}{=} |E'' \setminus U|$. If $d = 0$ then $C'' = C$ and thus $\mathcal{F}(r, C'') = \mathcal{F}(r, C)$ as claimed. If $d > 0$, then there exists an event $e \in E'' \setminus U$; since e is not in U, it is not an update event, thus we know $\mathcal{F}(r, C'') = \mathcal{F}(r, C''|_{E'' \setminus \{e\},\mathsf{op},\mathsf{vis},\mathsf{ar}})$, and the latter is equal to $\mathcal{F}(r, C)$ by induction.
2. For all $e \in [f] \setminus E'$, we have $\mathsf{rval}(e) = \mathcal{F}(\mathsf{op}(e), C)$. Proof: since e sees all events in U, we know $\mathsf{vis}^{-1}(e) \supset U$ and e is a read-only operation; using the first claim, we get $\mathcal{F}(\mathsf{op}(e), \mathsf{context}(A, e)) = \mathcal{F}(\mathsf{op}(e), A|_{\mathsf{vis}^{-1}(e),\mathsf{op},\mathsf{vis},\mathsf{ar}}) = \mathcal{F}(\mathsf{op}(e), C)$, which implies the claim by RVAL$(\mathcal{F})$.

The second claim then implies (5.2) because E' is finite. $\square$

6

Implementations

In this chapter, we take a look at more protocol examples. The examples represent three general patterns used for propagating updates: epidemic protocols, broadcast protocols, and global sequence protocols. For each of these categories, we present several variations that illustrate the tradeoff between the consistency model and the speed/availability of operations. The accompanying proofs of correctness can be found in chapter 10.

6.1 Overview

We show an overview of the 10 implementations in Fig. 6.1. The protocols are organized into three categories: *epidemic*, *broadcast*, and *global sequence* protocols. We describe these categories below (§6.1.1).

For each implementation, we show the data type, the consistency model, and the offline availability of operations, *i.e.* whether operations remain available in the presence of arbitrary network partitions.

Four of the implementations are generic templates, *i.e.* they work for any replicated data type $\mathcal{F}$, while the other six are specialized for a specific data type (counter, register, or key-value store).

Data Type	Implementation Name	Page	Consistency	Offline Availability
Epidemic Protocols				
$\mathcal{F}_{reg}$	EpidemicRegister	(p. 14)	sequential	yes
$\mathcal{F}_{ctr}$	EpidemicCounter	(p. 70)	eventual	yes
Broadcast Protocols				
$\mathcal{F}_{ctr}$	BroadcastCounter	(p. 69)	eventual	yes
$\mathcal{F}_{kvs}$	EventualStore	(p. 72)	eventual	yes
$\mathcal{F}_{kvs}$	CausalStore	(p. 74)	causal	yes
$\mathcal{F}$	CausalStreams$\langle\mathcal{F}\rangle$	(p. 82)	causal	yes
Global Sequence Protocols				
$\mathcal{F}_{reg}$	SingleCopyRegister	(p. 11)	linearizable	no
$\mathcal{F}$	Sequencer$\langle\mathcal{F}\rangle$	(p. 77)	sequential	reads
$\mathcal{F}$	AsyncSequencer$\langle\mathcal{F}\rangle$	(p. 78)	eventual	yes
$\mathcal{F}$	BufferedSequencer$\langle\mathcal{F}\rangle$	(p. 80)	causal	yes

Figure 6.1: Overview of the 10 protocol implementations presented in this chapter, grouped by their update propagation pattern. For a more precise comparison of the consistency and availability guarantees, see Fig. 6.2

6.1.1 Protocol Categories

We distinguish the following three categories.

In **Epidemic Protocols** (also known as "state-based" protocols), nodes propagate local updates to remote nodes by periodically disseminating a "summary" representing the *cumulative effect*. For example, in the epidemic register protocol described in the introduction (§1.3.2), the latest written value and the timestamp can be thought of a summary of the effect of all past updates. of all the updates known to this node. We call this propagation style *epidemic* because nodes "infect" other nodes with information, spreading information indirectly. Importantly, nodes do not need to broadcast each update to all other nodes, and messages may be lost, duplicated, or reordered without harm.

In **Broadcast Protocols** (also known as "operation-based" protocols), each node sends an update message to all remote nodes whenever it does a local update. The update messages are guaranteed to arrive, but may arrive in unspecified order.[1] Broadcast protocols are particularly easy to use when updates commute with each other, because then the order in which they are applied is irrelevant.

In **Global Sequence Protocols**, the nodes eventually agree on a common sequence of updates. This sequencing can be done by a server in a client-server topology, or by using a distributed algorithm[2] in a peer-to-peer system (*e.g.* Cachin et al. [2011], Nakamoto [2008]).

6.1.2 General Discussion

Performance. None of the categories or implementations is *a priori* better than others, not even if we fix the specific data type considered. Whether a particular implementation is well suited for an application depends on the frequency of operations, the number of replicas, and the sizes of summary messages, update messages, and data type content.

Consistency and Availability. As is to be expected due to the CAP theorem (§9.1), all implementations either (1) exhibit weak consistency (*i.e.* are neither sequentially consistent nor linearizable), or (2) are not fully available under partitions, or (3) use a very limited data type (*e.g.* EpidemicRegister).

The three sequencer protocols Sequencer$\langle\mathcal{F}\rangle$, AsyncSequencer$\langle\mathcal{F}\rangle$, and BufferedSequencer$\langle\mathcal{F}\rangle$ demonstrate how we can start with a conservative protocol that guarantees sequential consistency, but is potentially slow and does not work under arbitrary network partitions, and then add caching/queueing optimizations that turn it into a weakly consistent protocol that is available under partitions.

[1] We sometimes strengthen this requirement slightly, without fundamentally changing the performance tradeoffs, by requiring pairwise ordering, *i.e.* any two messages sent by the same origin to the same destination are delivered in order.

[2] This type of algorithm is usually called total-order broadcast, because it can be understood as a type of broadcast where all messages, including messages by different origins, are delivered in the same order to all receivers.

	Single-Copy Reg.	Epidemic Reg.	Broadcast Ctr.	Epidemic Ctr.	Eventual Store	Causal Store
data type $\mathcal{F}$	$\mathcal{F}_{reg}$		$\mathcal{F}_{ctr}$		$\mathcal{F}_{kvs}$	
LINEARIZABILITY($\mathcal{F}$)	✓	—	—	—	—	—
SEQUENTIALCONSISTENCY($\mathcal{F}$)	✓	✓	—	—	—	—
CAUSALCONSISTENCY($\mathcal{F}$)	✓	✓	—	✓	—	✓
BASICEVENTUALCONSISTENCY($\mathcal{F}$)	✓	✓	✓	✓	✓	✓
READMYWRITES	✓	✓	✓	✓	✓	✓
CONSISTENTPREFIX	✓	✓	—	—	—	—
MONOTONICREADS	✓	✓	✓	✓	✓	✓
CAUSALARBITRATION	✓	✓	✓	✓	✓	✓
CAUSALVISIBILITY	✓	✓	—	✓	—	✓
available under partitions	—	✓	✓	✓	✓	✓

	S.	*A.S.*	*B.S.*	*C.S.*
LINEARIZABILITY($\mathcal{F}$)	—	—	—	—
SEQUENTIALCONSISTENCY($\mathcal{F}$)	✓	—	—	—
CAUSALCONSISTENCY($\mathcal{F}$)	✓	—	✓	✓
BASICEVENTUALCONSISTENCY($\mathcal{F}$)	✓	✓	✓	✓
READMYWRITES	✓	—	✓	✓
CONSISTENTPREFIX	✓	✓	✓	—
MONOTONICREADS	✓	✓	✓	✓
CAUSALARBITRATION	✓	✓	✓	✓
CAUSALVISIBILITY	✓	—	✓	✓
availableunderpartitions(o)				
where $o \in$ readonlyops($\mathcal{F}$)	✓	✓	✓	✓
where $o \in$ mixedops($\mathcal{F}$)	—	—	✓	✓
where $o \in$ updateonlyops($\mathcal{F}$)	—	✓	✓	✓

Figure 6.2: Detailed breakdown of the consistency guarantees for of the counter, register, and store protocols (top) and the Sequencer (S.), AsyncSequencer (A.S.), BufferedSequencer (B.S.), and CausalStreams (C.S.) protocols (bottom). For the latter a check mark indicates $\Pi(\mathcal{F}) \models \mathcal{P}$ for all $\mathcal{F}$, and a dash indicates $\Pi(\mathcal{F}) \not\models \mathcal{P}$ for some $\mathcal{F}$.

Guarantee Details. The precise consistency and availability guarantees for each implementation are shown in Fig. 6.2, which also shows some finer consistency distinctions. For example, both BufferedSequencer$\langle\mathcal{F}\rangle$ and CausalStreams$\langle\mathcal{F}\rangle$ satisfy CausalConsistency($\mathcal{F}$), but only BufferedSequencer$\langle\mathcal{F}\rangle$ satisfies ConsistentPrefix.

Other Names. Database systems are often categorized as primary replication (typically corresponding to global sequence protocols) or multi-master replication (typically corresponding to epidemic or broadcast protocols). Epidemic protocols are also similar to gossip protocols, where pairs of nodes periodically exchange information bidirectionally (as formalized by Princehouse et al. [2014], for example). In the literature on conflict-free replicated data types (CRDTs) , such as in Shapiro et al. [2011a,b,c], Roh et al. [2011], Bieniusa et al. [2012a], or Burckhardt et al. [2014a], epidemic protocols are called state-based (because the summary that is sent is usually the same as the local state stored on each replica), broadcast protocols are called operation-based, and global sequencing is generally avoided.

Realism. The example algorithms in this chapter demonstrate techniques that are commonly used in real systems. However, real systems often contain additional complications that we have omitted for now (such as partitioning of servers using a distributed hash table, and the use of quorums).

6.2 Pseudocode Semantics

To reason about correctness and consistency with mathematical precision, a protocol must define exactly what distributed executions are possible. At the same time, we want protocols to be readable, concise, and mostly self-explanatory. Unfortunately, these goals are conflicting! As a compromise, we present protocols in pseudocode, but later on describe the meaning carefully by means of a compilation process (§8.4) that compiles the pseudocode into a formal protocol.

Our pseudolanguage liberally includes syntax and features that are common in modern programming languages, in particular functional

languages, such as pattern matching. We assume the reader is mostly familiar with those. The following concepts, on the other hand, are somewhat nonstandard.

Default Values. All our types have a default value, which is the initial value of all state variables. The default value of nat is 0.

Maps. We use two kinds of maps: partial maps and total maps. Partial maps pmap<Key,Value> work exactly like maps in Java, or dictionaries in C#: mathematically, a partial map m is a partial function with finite domain m.keys and range m.values. Total maps tmap<Key,Value>, however, are total functions: all keys (*i.e.* all elements of the type Key) are always present, but only finitely many map to a non-default value. If m is a total map, then m.keys returns all the keys that map to a non-default value, and m.values returns all the non-default values in the range of m.

6.3 Counters

A broadcast-based counter is shown in Fig. 6.3. Each role stores the current value of the counter in current (line 6). On a read operation (line 8), this current value is returned. On an increment operation (line 11), the value is incremented, an Inc message is broadcast, and *ok* is returned. When receiving an Inc message (line 16), we increment the current value. Since increment operations commute, the fact that messages arrive in nondeterministic order is not a problem.

We show an epidemic counter in Fig. 6.5. As for any epidemic protocol, the key design question is how to construct a summary that can concisely represent the cumulative effect of all known updates (in this case, increment operations) and that can be merged with other summaries. The solution is to use a vector that counts the number of increments *separately for each replica* [Shapiro et al., 2011b]. This vector[3] is stored in counts (line 6) and sent in the epidemic messages (line 15). Its initial value is the default value of tmap<nat,nat>, which

[3]Since the number of replicas is infinite (to model unbounded dynamic creation of replicas during execution), the vector is infinite but sparse. We represent it as a total map tmap<nat,nat>.

```
1  protocol BroadcastCounter {
2
3    message Inc() : reliable
4
5    role Peer(nr: nat) {
6      var current: nat;
7
8      operation read() {
9        return current;
10     }
11     operation inc() {
12       current := current +1;
13       send Inc();
14       return ok;
15     }
16     receive Inc() { current++; }
17   }
18 }
```

Figure 6.3: An implementation of the counter $\mathcal{F}_{ctr}$ based on reliable broadcast.

$$\Pi_{\mathsf{BroadcastCounter}} \stackrel{\text{def}}{=} (R, M, O, \Sigma, P, S, T) \text{ where}$$

$$R = \{Peer(i) \mid i \in \mathbb{N}_0\}$$
$$M = \{Inc\}$$
$$O_{Peer(i)} = \{\mathsf{rd}, \mathsf{inc}\}$$
$$\Sigma_{Peer(i)} = \{SPeer(c) \mid c \in \mathbb{N}_0\}$$
$$P_{Peer(i)} = \{bg\}$$
$$S = reliable(\{Inc\})$$
$$T_{Peer(i)} = \Big\{ \begin{array}{l} init(SPeer(0), \emptyset), \\ callret(\mathsf{inc}, SPeer(c), SPeer(c+1), \{Inc\}, ok), \\ callret(\mathsf{rd}, SPeer(c), SPeer(c), \emptyset, c), \\ rcv(Inc, SPeer(c), SPeer(c+1), \emptyset) \\ step(bg, SPeer(c), SPeer(c), \emptyset) \end{array} \Big| \quad c \in \mathbb{N}_0 \quad \Big\}$$

Figure 6.4: Formal protocol (as defined in §8.3) for the broadcast-based counter, corresponding to the pseudocode in Figure 6.3.

```
1  protocol EpidemicCounter {
2
3    message Latest(c: tmap<nat,nat>) : dontforge, eventualindirect
4
5    role Peer(n: nat) {
6      var counts: tmap<nat,nat>;
7
8      operation read() {
9        return counts.values().sum();
10     }
11     operation inc(val: boolean) {
12       counts[n] := counts[n] + 1;
13       return ok;
14     }
15     periodically { send Latest(counts); }
16
17     receive Latest(c) {
18       foreach (k in c.keys) { counts[k] = max(counts[k], c[k]); }
19 } } }
```

Figure 6.5: An epidemic implementation of the counter $\mathcal{F}_{ctr}$.

$$\Pi_{\mathsf{EpidemicCounter}} \stackrel{\text{def}}{=} (R, M, O, \Sigma, P, S, T) \text{ where}$$

$$R = \{Peer(i) \mid i \in \mathbb{N}_0\}$$
$$M = \{Latest(c) \mid c \in (R \to_{\text{fin}} \mathbb{N}_0)\}$$
$$O_{Peer(i)} = \{\mathsf{rd}, \mathsf{inc}\}$$
$$\Sigma_{Peer(i)} = \{SPeer(c) \mid c \in (R \to_{\text{fin}} \mathbb{N}_0)\}$$
$$P_{Peer(i)} = \{bg\}$$
$$S = dontforge(M) \wedge eventualindirect(M)$$
$$T_{Peer(i)} = \Big\{ \begin{array}{l} init(SPeer(\lambda r.0), \emptyset), \\ callret(\mathsf{inc}, SPeer(c), SPeer(c[r \mapsto c[r]+1]), \emptyset, ok), \\ callret(\mathsf{rd}, SPeer(c), SPeer(c), \emptyset, \sum_{r \in \mathsf{dom}\sigma} c(r)), \\ rcv(Latest(c'), SPeer(c), SPeer(\lambda r. \max(c(r), c'(r))), \emptyset), \\ step(bg, SPeer(c), SPeer(c), \{Latest(c)\}) \end{array} \Big| \quad c, c' \in (R \to_{\text{fin}} \mathbb{N}_0) \Big\}$$

Figure 6.6: Protocol for the epidemic implementation of the counter, compiled from the pseudocode in Figure 6.5.

maps each role to zero. On read operations (line 8), the sum of all counts is returned. On increment operations (line 11), the entry of the count vector corresponding to this role is incremented. When receiving a vector (line 17), the two vectors are merged by taking the point-wise maximum.

6.4 Stores

Key-value stores are the most widely used eventually consistent replicated data types. We discuss two implementations. The first one is eventually consistent, and achieves convergence by applying updates only if they have a more recent timestamp (known as Thomas' write rule, going back to Thomas and Beranek [1979]). The second one tracks dependencies to maintain causality and is a variant of the algorithm described in Lloyd et al. [2011]. Both protocols implement the general data type $\mathcal{F}_{kvs}$, and are parameterized by types $\mathsf{Key} \subseteq \mathsf{Values}, \mathsf{Val} \subseteq \mathsf{Values}$, which specify the range of keys and values.

6.4.1 Eventual Store

Fig. 6.7 shows a key-value store implementation based on reliable broadcast. It uses Lamport logical clocks to create timestamps. Each role stores a copy of the entire store, with timestamps attached to each value in the store.

When a read operation is called (line 14), we simply return the value stored, or the constant *undef* if there is none for this key (as required by the definition of $\mathcal{F}_{kvs}$).

When a write operation is called (line 20), we create a new logical timestamp for this operation by advancing the local clock, store the new value and the new timestamp in store, and broadcast an Update message containing the timestamp, the key, and the value.

When receiving an Update message (line 26), it is applied to the local store only if the store has no entry for this key yet, or the timestamp in the message is larger than what is in the store. This ensures that updates are "only applied forward".

```
1  protocol EventualStore⟨Key,Val⟩ {
2
3    struct Timestamp(number: nat; rid: nat);
4    function lessthan(Timestamp(n1,rid1), Timestamp(n2,rid2)) : boolean {
5      return (n1 < n2) ∨ (n1 == n2 ∧ rid1 < rid2);
6    }
7
8    message Update(key: Key, val: Val, ts: Timestamp) : reliable
9
10   role Replica(rid: nat) {
11     var localclock: nat;
12     var store: pmap⟨Key, pair⟨Val,Timestamp⟩⟩;
13
14     operation read(key: Key) {
15       match store[key] with
16         ⊥→ { return undef;}
17         (val,ts) → { return val; }
18       }
19     }
20     operation write(key: Key, val: Val) {
21       localclock++; // advance logical clock
22       store[key] := (val,ts);
23       send Update(key,val,Timestamp(localclock,rid));
24       return ok;
25     }
26     receive Update(key,val,ts) {
27       if (store[key] = ⊥∨ store[key].second.lessthan(ts))
28             store[key] := (val, ts);
29       if (ts.number > localclock) // keep up with time
30             clock := ts.number;
31     }
32   }
33 }
```

Figure 6.7: A broadcast-based eventually consistent key-value store with last-writer-wins resolution $\mathcal{F}_{\text{kvs}}$.

Consistency. The EventualStore<K,V> protocol is eventually consistent, because the reliable delivery implies that all updates become eventually visible to all nodes. Also, it satisfies causal arbitration, because the timestamps increase with the happens-before order (this is always guaranteed when using Lamport clocks). It does, however, not satisfy causal visibility (and is thus not causally consistent) because updates reach different roles at different times, thus the anomaly shown on p. 57 is possible.

6.4.2 Causal Store

Figures 6.8 and 6.9 show a key-value store implementation that tracks dependencies to ensure causal consistency. The best way to understand this is to see it as a tweaked version of the EventualStore protocol in Fig. 6.7. The differences are:

- Each role keeps track of what has been read in a session, and stores it in deps. Note that deps is a partial map: its keys are the keys of the store that have been read in this session, and its values are the timestamps of what was read for each key.

- When writing, the dependencies are included in the Update message that is broadcast.

- When receiving an update, it is *not* immediately applied. Instead, it goes into a processing queue. Each role stores one such queue per source role, in a structure InBuffer, which also stores the timestamp of the latest update processed from that queue.

- Periodically, for each buffer, we check if the next update in the queue is ready to process, which is the case iff all of its dependencies have timestamps that are less than or equal to the timestamp number of the latest processed update from the same role.

```
1  protocol CausalStore⟨Key,Val⟩ {
2
3    struct Timestamp(number: nat ; rid: nat)
4    function lessthan(Timestamp(n1,rid1), Timestamp(n2,rid2)) {
5      return (n1 < n2) ∨ (n1 == n2 ∧ rid1 < rid2) ;
6    }
7    struct Update(ts: Timestamp, key: Key, val: Val,
8                              deps: pmap⟨Key,Timestamp⟩)
9
10   message Notify(update:Update) : reliablestream
11
12   role Replica(rid: nat) {
13
14     struct InBuffer {
15       updates: queue<Update> ;
16       lastprocessed: nat ;
17     }
18
19     var store: pmap⟨Key, pair⟨Val,Timestamp⟩⟩ ;
20     var buffers: tmap⟨nat, InBuffer⟩ ;
21     var deps: pmap⟨Key, Timestamp⟩ ;
22
23     ... continued in Fig.6.9 ....
24   }
25 }
```

Figure 6.8: A causally consistent implementation of the key-value store with last-writer-wins resolution $\mathcal{F}_{\text{kvs}}$.

6.5 Protocol Templates

We now study protocol templates that work for generic replicated data types $\mathcal{F}$ that do not have any blocking operations. They are usually (but not always) less efficient than protocols optimized for that particular data type, because they often store or propagate more information than necessary. However, they are excellent starting points for implementations, to which we can then add space reduction optimizations. We divide these templates into two categories: three are based on single-server serialization, and one is based on reliable broadcast.

```
1  ...
2
3  operation read(key: Key) {
4      match store[key] with {
5          ⊥→ { return undef;}
6          (val,ts) → { deps[key] := ts; return val; }
7      }
8  }
9
10 operation write(key: Key, val: Val) {
11     buffers[rid].lastprocessed++; // advance logical clock
12     var ts := Timestamp(buffers[rid].lastprocessed,rid);
13     store[key] := (val, ts);
14     send Notify(Update(ts,key,val,deps));
15     return ok;
16  }
17
18 receive Notify(update) {
19     buffers[update.ts.rid].updates.enqueue(update);
20 }
21
22 function readytoapply(u) {
23     return forall (k,ts) in u.deps :
24         buffers[ts.rid].lastprocessed ⩾ ts.number;
25 }
26
27 periodically {
28     foreach(r in buffers.keys)
29       if (!buffers[r].updates.empty
30                                 ∧ readytoapply(buffers[r].updates.next)) {
31         var u := buffers[r].updates.dequeue();
32         if ((store[u.key] = ⊥) ∨ store[u.key].second.lessthan(u.ts))
33           store[u.key] := (u.val, u.ts);
34         buffers[r].lastprocessed := u.ts.number;
35         if (u.ts.number > buffers[rid].lastprocessed) //keep up with time
36           buffers[rid].lastprocessed := update.ts.number;
37     }
38 }
39
40 ...
```

Figure 6.9: Code for the Replica Role in Fig. 6.8.

6.5.1 Single-Server Protocols

The first three protocols are based on the simple idea that all updates go through a single server, which provides a serialization point. We start with a conservative version that is sequentially consistent and then gradually introduce caching and queueing optimizations that improve the latency/availability of operations at the expense of consistency.

Sequential Consistency, Local Reads. The protocol $\mathsf{Sequencer}\langle\mathcal{F}\rangle$ is shown in Fig. 6.10. Each client stores a log of confirmed operations in confirmed. Only operations that are received from the server are appended to the log. All messages sent from the server are delivered reliably and in order (as indicated by the transport requirement reliablestream), thus all logs show the same order of operations.

To perform a read-only operation, we need not communicate with the server, but can determine the correct return value based on the replicated data type $\mathcal{F}$ and the log. This is done in the function computeresult, which computes the context $(E, \mathsf{op}, \mathsf{vis}, \mathsf{ar})$ where E is the set of positions in the log, op is the operation at that position, and vis and ar are the order of the operations in the log.

To perform an update operation, we send the operation to the server and wait for the response. Note that while we are waiting for a response, the server may send us other messages. When we receive the response (which is different from other messages because its cid field matches this client), we compute the result based on the current log.

Note that the protocol is sequentially consistent (we prove this in Theorem 10.14 on page 127). But it is not linearizable: a read operation starting after a write operation returns does not necessarily see that write operation, if they are on different clients, since the server may not have delivered the update to the client performing the read yet.

Asynchronous Update-Only Operations. It may appear to be a waste of time to wait for responses of update-only operations, since the returned value is always just *ok*. A popular (but consistency-weakening, thus not really correct) optimization is to return *ok* immediately and perform the update asynchronously. The protocol $\mathsf{AsyncSequencer}\langle\mathcal{F}\rangle$ in Fig. 6.11 shows this optimization. It ensures that all update messages

```
1  protocol Sequencer⟨F⟩ {
2
3    message ToServer(cid: nat, op: Operation) : reliable
4    message ToAll(cid: nat, op: Operation) : reliablestream
5
6    role Server {
7      receive(ToServer(cid,op)) {
8        send ToAll(cid,op);
9      }
10   }
11
12   function computeresult(op: Operation, s: seq⟨Operation⟩) {
13     return F(op, ( s.positions, //E
14                    (p) ⇒ s[p], // op
15                    (p1,p2) ⇒ (p1 < p2), // vis
16                    (p1,p2) ⇒ (p1 < p2)); // ar
17   }
18
19   role Client(cid: nat) {
20     confirmed: seq⟨Operation⟩;
21     operation perform(op: Operation) where [op ∈ readonlyops(F)] {
22       return computeresult(op,confirmed);
23     }
24     operation perform(op: Operation) where [op ∉ readonlyops(F)] {
25       send ToServer(cid, op);
26       // does not return to caller yet
27     }
28     receive ToAll(c, op) where [c = cid] {
29       var rval = computeresult(op,confirmed);
30       confirmed.append(op);
31       return rval; // returns earlier call
32     }
33     receive ToAll(c, op) where [c != cid] {
34       confirmed.append(op);
35     }
36   }
37 }
```

Figure 6.10: Global sequence protocol Sequencer$\langle\mathcal{F}\rangle$ that guarantees sequential consistency. Read-only operations are available under partitions.

```
1  protocol AsyncSequencer⟨F⟩ {
2
3    message ToServer(cid: int, op: Operation) : reliablestream
4    message ToAll(cid: int, op: Operation) : reliablestream
5
6    role Server {
7      receive(ToServer(cid,op)) {
8        send ToAll(cid,op);
9      }
10   }
11
12   function computeresult(op: Operation, s: seq⟨Operation⟩) {
13     return F(op, ( s.positions, // E
14                    (p) ⇒ s[p], // op
15                    (p1,p2) ⇒ (p1 < p2), // vis
16                    (p1,p2) ⇒ (p1 < p2))); // ar
17   }
18
19   role Client(cid: int) {
20     confirmed: seq⟨Operation⟩;
21     operation perform(op: Operation) where [op ∈ readonlyops(F)] {
22       return computeresult(op, confirmed);
23     }
24     operation perform(op: Operation) where [op ∈ updateonlyops(F)] {
25       send(ToServer(cid,op));
26       return ok;
27     }
28     receive ToAll(c, op) {
29       confirmed.append(op);
30     }
31   }
32 }
```

Figure 6.11: Global sequence protocol AsyncSequencer$\langle\mathcal{F}\rangle$. Read-only and update-only operations are available under partitions.

sent to the server remain ordered, thus the annotation reliablestream on ToServer messages.

This optimization means that both read-only and update-only operations are available under partitions. However, the protocol is no longer sequentially consistent: the Dekker anomaly (§5.2.1) is possible, since we do not wait for writes to propagate to the server. Perhaps worse, it does not guarantee read-my-writes (§5.1.1), and thus no longer guarantees causal consistency.

Pending Operations Buffer. Our final version of the sequencer protocol, the BufferedSequencer$\langle\mathcal{F}\rangle$, is shown in Fig. 6.12. It is similar to the AsyncSequencer$\langle\mathcal{F}\rangle$ (in particular, all operations are available under partitions), but has stronger consistency because it takes pending updates into account when computing the result of read operations. The idea is to buffer pending updates locally in a sequence pending, and make them visible to subsequent local operations even before they are confirmed[4]. We discuss below how the protocol works, and prove in Theorem 10.14 that it guarantees causal consistency and consistent prefix.

When an update operation is performed, we create an update record that captures the operation, the size of the current log, and the client id (line 27). The size of the log is recorded so that we know later on what operations in the log were visible to this operation. When implementing a replicated data type $\mathcal{F}$ that does not depend on the vis in the context, such as any replicated data type using standard conflict resolution, this field is not needed and can be removed.

The update record is then appended to the pending sequence and sent to the server (line 29). Update records arriving from the server are appended to the confirmed sequence (line 34), and removed from the pending sequence if they originated on this client (line 36).

When computing the return value of an operation (line 25), the function computeresult constructs an operation context containing the operations in both sequences confirmed and pending, with arbitration order corresponding to the concatenated sequence confirmed $\cdot$ pending, and with visibility determined by the principle that (1) an operation

[4]This idea is very similar to the use of store buffers in the TSO memory model. It is however not exactly equivalent, as shown in Burckhardt et al. [2014b].

```
1  protocol BufferedSequencer⟨F⟩ {
2
3    message ToServer(u: Update) : reliablestream
4    message ToAll(u: Update) : reliablestream
5
6    role Server {
7      receive(ToServer(u)) {
8        send ToAll(u);
9      }
10   }
11
12   struct Update(op: Operation, visprefix: nat, cid: nat);
13
14   function computeresult(op: Operation, s: seq⟨Update⟩) {
15       return F(op, (s.positions, (p) ⇒ s[p].op, // E, op
16           (p1,p2) ⇒ (p1 < p2) ∧
17                     (s[p1].cid = s[p2].cid ∨ p1 < s[p2].visprefix), // vis
18           (p1,p2) ⇒ (p1 < p2))); // ar
19   }
20
21   role Client(cid: nat) {
22     confirmed: seq⟨Update⟩;
23     pending: queue⟨Update⟩;
24     operation perform(op: Operation) {
25       var rval = computeresult(op, confirmed · pending);
26       if (op ∉ readonlyops(F)) {
27         var u = Update(op, confirmed.size, cid);
28         pending.enqueue(u);
29         send ToServer(u);
30       }
31       return rval;
32     }
33     receive ToAll(u: Update) {
34       confirmed.append(u);
35       if (u.cid = cid)
36           pending.dequeue(); // remove confirmed update
37     }
38   }
39 }
```

Figure 6.12: Global sequence protocol BufferedSequencer$\langle\mathcal{F}\rangle$ that guarantees causal consistency and consistent prefix. All operations are available under partitions.

sees all preceding updates by the same node, and (2) it sees all operation in the log prefix up to the recorded length visprefix.

6.5.2 Broadcast Protocol: CausalStreams

The protocol CausalStreams$\langle\mathcal{F}\rangle$ in Fig. 6.13 satisfies roughly the same guarantees as protocol BufferedSequencer$\langle\mathcal{F}\rangle$; in particular, it satisfies causal consistency, and all operations are available under partitions. However, it does not use a single server for serialization, but uses direct, reliable broadcast. The general idea is somewhat similar to the causally consistent key-value store in Fig. 6.8: when receiving an update, we do not apply it immediately, but put it into a queue and wait until it is ready (*i.e.* all updates it depends on have arrived). To determine whether that is the case, we do not attach the actual dependencies, but use *vector clocks* (Fidge [1988], Mattern [1989]).

Each node stores the set of all updates it has heard of in known, and stores a current vector clock in vc. The entries of the vc vector clock represent (1) the current clock of this node (at vc[cid]), and (2) the timestamp of the latest update received from each node y (at vc[y]).

When an operation is performed, we compute a new timestamp, but first advance vc[cid] to a value that is larger than any other value in the vector clock. Thus, we know that (a) we can always tell where an update came from by looking at its timestamp - it must be the role whose number is largest, and (b) the new timestamp is ordered after all timestamps of updates known to this role. An update structure (with the operation and the timestamp) is then broadcast to all other roles.

To compute the return value, we create an operation context by taking all updates in known and using their vector clocks to determine both visibility and arbitration. Arbitration is determined by the size of the maximal entry (and the role id if there is a tie); visibility is determined by checking if each entry is less or equal.

Updates received are put into a queue and later processed by a periodic background process. An update is processed once its timestamp's secondary elements (*i.e.* all elements that are not maximal, and are thus not the origin of the update) are no larger than the current vector clock. This ensures causality.

```
1  protocol CausalStreams⟨F⟩ {
2
3    type VClock = tmap⟨int,int⟩;
4    function max(vc: VClock) { return vc.values().max(); }
5
6    struct Update(op: Operation, vc: tmap<int,int>)
7
8    function origin(u: Update) {
9      return u.vc.key_of_max_value; }
10   function arbitedbefore(u1 : Update, u2 : Update) {
11     return max(u1.vc) < max(u2.vc) ∨
12              max(u1.vc) = max(u2.vc) ∧ origin(u1.vc) < origin(u2.vc); }
13   function visibleto(vc1 : VClock, vc2 : VClock) {
14     return forall i : vc1[i] ⩽ vc2[i]; }
15
16   message Notify(u: Update, vc: VClock) : reliablestream
17
18   role Peer(pid: int) {
19     var known: set⟨Update⟩;
20     var vc: VClock;
21     var pending: tmap⟨nat,queue⟨Update⟩⟩;
22     operation perform(op: Operation) {
23       var rval = F(op, makecontext(known, arbitedbefore, visibleto));
24       vc[pid] = max(vc)+1; // advance logical clock
25       var u = Update(op,vc);
26       known.add(u);
27       send Notify(u,vc);
28       return rval;
29     }
30     receive Notify(u) { pending[origin(u.vc)].add(u); }
31     periodically for (sender: int) {
32       if (forall i : (i = sender) ∨ vc[i] ⩾ pending[sender].next.vc[i]) {
33         var u := pending[sender].dequeue();
34         known.add(u);
35         vc[sender] := u.vc[sender];
36       }
37     }
38 } }
```

Figure 6.13: Broadcast protocol CausalStreams$\langle\mathcal{F}\rangle$ that uses vector clocks for causality. All operations are available under partitions.

7

Concrete Executions

To verify protocol implementations, we need a model of execution with an appropriate level of detail: it must include the sending and receiving of messages (which are not present in abstract executions), and show exactly how they interleave with operation calls, operation returns, and background processing steps on each role. In the next three chapters, we present a formal model of executions and protocols.

7.1 Transitions

We model concrete executions based on atomic events called *transitions*. Transitions happen on a particular role instance, may receive or send messages, and may modify the local state of the role. To define transitions, we assume the following sets of entities: A set of role instances Roles, a set of messages Messages, a set of local states States, and a set of processes Processes.

Each transition represents an atomic step a role can take. Transitions happen in response to an external stimulus. This stimulus can be either an operation call by the client program, an incoming message from a remote role, or the scheduler choosing to perform a processing

step. When transitions execute, they can do two things: send a (finite) number of messages to other roles, and/or return a value to the client program.

Definition 7.1. Define the set of transitions Transitions to consist of all syntactic expression of the form of one of

$$\begin{array}{lll} \mathit{init}(\sigma', M) & \mathit{call}(o, \sigma, \sigma', M) & \mathit{callret}(o, \sigma, \sigma', M, v) \\ & \mathit{rcv}(m, \sigma, \sigma', M) & \mathit{rcvret}(m, \sigma, \sigma', M, v), \\ & \mathit{step}(p, \sigma, \sigma', M) & \mathit{stepret}(p, \sigma, \sigma', M, v) \end{array}$$

where $\sigma, \sigma' \in \mathsf{States}$, $o \in \mathsf{Operations}$, $M \subseteq_{\text{fin}} \mathsf{Messages}$, $v \in \mathsf{Values}$, $m \in \mathsf{Messages}$, and $p \in \mathsf{Processes}$.

The meaning of these transitions is as follows.

- The transition $\mathit{init}(\sigma', M)$ is an initialization transition, which puts the automaton in an initial state σ'. M is a finite set of messages, describing the messages that are sent by this transition (possibly empty).
- The transition $\mathit{call}(o, \sigma, \sigma', M)$ means the role accepts a call to operation o by the client program. It starts in state σ (called the pre-state), ends in state σ' (the post-state), and sends the set of messages M.
- The transition $\mathit{rcv}(m, \sigma, \sigma', M)$ means the role accepts a message m from the network. Again, σ and σ' are the pre- and post-state, respectively, and M is the set of messages sent by the transition.
- The transition $\mathit{step}(p, \sigma, \sigma', M)$ means that process p of this role takes a spontaneous step. The meaning of σ, σ', and M is as before. Roles can be composed of multiple internal processes, each of which is guaranteed to be scheduled fairly (meaning that each process in a non-crashed role executes an infinite number of steps).

For the latter three transitions we also include a variant that additionally returns a value v to the client program, with the following meaning.

The *callret* transition models an operation that returns instantly (the call and return are part of the same atomic transition). The transitions *rcvret* and *stepret* model situations where a currently pending operation returns, in response to a an arriving message or some internal processing step.

Implicit Broadcast. When a message is sent, no particular destination is specified — in our model, all sent messages go to all roles. This simplifies the reasoning about protocols that use broadcast. It does not limit generality: for protocols using point-to-point communication, we can add destination information into the message and ignore the message on non-destination nodes (using a dummy transition of the form $rcv(m, \sigma, \sigma, \emptyset)$).

Definition 7.2. Define the partial functions op, rcv, proc, pre, post, snd, rval on transitions as follows:

t	op(t)	rcv(t)	proc(t)	pre(t)	post(t)	snd(t)	rval(t)
$init(\sigma', M)$	$\bot$	$\bot$	$\bot$	$\bot$	σ'	M	$\bot$
$call(o, \sigma, \sigma', M)$	o	$\bot$	$\bot$	σ	σ'	M	$\bot$
$rcv(m, \sigma, \sigma', M)$	$\bot$	m	$\bot$	σ	σ'	M	$\bot$
$step(p, \sigma, \sigma', M)$	$\bot$	$\bot$	p	σ	σ'	M	$\bot$
$callret(o, \sigma, \sigma', M, v)$	o	$\bot$	$\bot$	σ	σ'	M	v
$rcvret(m, \sigma, \sigma', M, v)$	$\bot$	m	$\bot$	σ	σ'	M	v
$stepret(p, \sigma, \sigma', M, v)$	$\bot$	$\bot$	p	σ	σ'	M	v

For a set E of events, we define the subsets of calls and returns as $\mathsf{calls}(E) \stackrel{\text{def}}{=} \{e \in E \mid \mathsf{op}(e) \neq \bot\}$ and $\mathsf{returns}(E) \stackrel{\text{def}}{=} \{e \in E \mid \mathsf{rval}(e) \neq \bot\}$.

7.2 Trajectories

During an execution, each role performs a sequence of transitions, subject to a few conditions: each transition sequence must start with an initialization transition; all but the first transition must start in the state that the previous transition ended at; and calls and returns must be properly matched. We call such a sequence a *trajectory*. Formally,

Definition 7.3 (Trajectories)**.** A *trajectory* is an event graph $(E, \mathsf{eo}, \mathsf{tr})$ such that

(t1) eo is an enumeration of E.

(t2) $\mathsf{tr} : E \to \mathsf{Transitions}$ specifies the transition of each event. We write $\mathsf{op}(e)$ short for $\mathsf{op}(\mathsf{tr}(e))$, and similarly for rcv, proc, pre, post, snd, and rval.

(t3) The first (and only the first) transition is an initialization transition, and the pre-state of each transition matches the post-state of the previous transition:

$$\forall e \in E : \quad \big(\ \mathsf{pre}(e) = \bot = \mathsf{pred}(E, \mathsf{eo}, e) \\ \vee \quad \mathsf{pre}(e) = \mathsf{post}(\mathsf{pred}(E, \mathsf{eo}, e)) \ \big)$$

where $\mathsf{pred}(E, \mathsf{eo}, e)$ is the partial function that returns the predecessor of e in E with respect to the enumeration eo, or is undefined ($\bot$) if there is no predecessor.

(t4) A call transition may not follow another call transition unless there is a return transition in between them:

$$\forall c_1, c_2 \in \mathsf{calls}(E) : \quad c_1 <_{\mathsf{eo}} c_2 \quad \Rightarrow \\ \exists r \in \mathsf{returns}(E) : \quad c_1 \leq_{\mathsf{eo}} r <_{\mathsf{eo}} c_2$$

We let $\mathcal{T}$ be the set of all trajectories.

Note that the conditions above do not quite enforce proper alternation of calls and returns: they allow spurious returns, because buggy protocol implementations may perform too many return transitions. Correct protocols should always produce well-formed trajectories:

Definition 7.4 (Well-formed Trajectories)**.** A trajectory $(E, \mathsf{eo}, \mathsf{tr})$ is *well-formed* if each event is preceded by no more returns than calls:

$$\forall e \in E : \big|\{r \in \mathsf{returns}(E) \mid r \leq_{\mathsf{eo}} e\}\big| \leq \big|\{c \in \mathsf{calls}(E) \mid c \leq_{\mathsf{eo}} e\}\big|$$

7.3 Concrete Executions

Trajectories describe the execution of a single role. To describe an execution of the entire system, we use (1) one trajectory per participating role, and (2) a message delivery relation del that matches send and receive events.

Definition 7.5 (Concrete Executions)**.** A *concrete execution* is an event graph $G = (E, \mathsf{eo}, \mathsf{tr}, \mathsf{role}, \mathsf{del})$ such that

(c1) eo is an enumeration of E.

(c2) $\mathsf{tr} : E \to \mathsf{Transitions}$ specifies the transition of each event. As before, we write $\mathsf{op}(e)$ short for $\mathsf{op}(\mathsf{tr}(e))$, etc.

(c3) $\mathsf{role} : E \to \mathsf{Roles}$ specifies the role of each event.
We write $E(r) \stackrel{\text{def}}{=} \{e \in E \mid \mathsf{role}(e) = r\}$.

(c4) The events for each role are a trajectory:

$$\forall r \in R: \quad G|_{E(r),\mathsf{eo},\mathsf{tr}} \ \in \ \mathcal{T}$$

(c5) del is a binary, injective relation that satisfies

$$\forall s, r \in E: \quad s \xrightarrow{\mathsf{del}} r \quad \Rightarrow \quad s \xrightarrow{\mathsf{eo}} r \ \ \wedge \ \ \mathsf{rcv}(r) \in \mathsf{snd}(s)$$

We denote the set of all concrete executions by $\mathcal{E}$, and the set of all finite concrete executions by $\mathcal{E}_{\text{fin}}$.

Examples. Fig. 2.2(d) on page 26 shows an example of a finite concrete execution. Each event is represented by its transition. Events are aligned in columns according to which replica they are on, and aligned vertically according to the execution order eo which is the timeline (higher up means earlier). The delivery relation del is indicated by arrows, and connects sending transitions to receive transitions. Two examples of infinite concrete executions are shown on pages 91 and 92, on the left, and discussed in the first paragraph below the diagrams.

Definition 7.6 (Notations for Executions)**.** For convenience, we introduce a few notations. For a concrete execution $G = (E, \mathsf{eo}, \mathsf{tr}, \mathsf{role}, \mathsf{del})$, we define:

- $\mathsf{traj}(G, r)$ is the trajectory of role r in G: $\mathsf{traj}(G, r) \stackrel{\text{def}}{=} G|_{E(r),\mathsf{eo},\mathsf{tr}}$.
- $\approx_{\mathsf{role}}$ is an equivalence relation on events relating events by the same role: $e \approx_{\mathsf{role}} e' \stackrel{\text{def}}{\iff} \mathsf{role}(e) = \mathsf{role}(e')$.
- ro is the role-order relation, defined to be the execution order restricted to events by the same role: $\mathsf{ro} \stackrel{\text{def}}{=} (\mathsf{eo} \cap \approx_{\mathsf{role}})$. Note that $\mathsf{ro}|_{E(r)}$ is always an enumeration of $E(r)$.
- $\mathsf{roles}(G)$ is the set of all roles that appear in the execution: $\mathsf{roles}(G) \stackrel{\text{def}}{=} \{r \in \mathsf{Roles} \mid |E(r)| > 0\}$.
- $\mathsf{del}(M)$ is the delivery relation del restricted to $M \subseteq \mathsf{Messages}$: $s \xrightarrow{\mathsf{del(M)}} r \quad \Leftrightarrow \quad s \xrightarrow{\mathsf{del}} r \ \wedge \ \mathsf{rcv}(r) \in M$.

7.3.1 Silent Crashes

The system guarantees that each correct role gets scheduled repeatedly unless it crashes. In our model, crashes simply correspond to finite trajectories - because a role that does not crash will execute forever. We call a role *correct* in an execution if it never crashes.

Definition 7.7 (Correct Roles)**.** Let $G = (E, \mathsf{eo}, \mathsf{tr}, \mathsf{role}, \mathsf{del})$ be a concrete execution. We define the set of correct (*i.e.* non-crashed) roles to be the set of roles whose trajectories are infinite, and the crashed roles to be the set of roles whose trajectories are nonempty and finite:

$$\mathsf{correct}(G) = \{r \in \mathsf{Roles} \mid |E(r)| = \infty\}$$
$$\mathsf{crashed}(G) = \{r \in \mathsf{Roles} \mid 0 < |E(r)| < \infty\}$$

We say an execution G is *complete* if $\mathsf{crashed}(G) = \emptyset$. We define $\mathcal{E}_{\text{complete}}$ to be the set of complete executions.

An important observation is that there is no difference between a crashed node and a node that is simply "frozen" (and may all of a sudden decide to thaw and take more steps). This type of failure is thus sometimes called a *silent crash*. In general, silent crashes are more difficult to handle than detectable crashes. For example, Fischer et al. [1982] show that consensus cannot be solved in an asynchronous system with silent crashes.

7.3.2 Well-formed Executions

For correct implementations and client programs, calls and returns on each role should alternate properly.

Definition 7.8 (Well-formed Executions)**.** A concrete execution $G = (E, \mathsf{eo}, \mathsf{tr}, \mathsf{role}, \mathsf{del})$ is *well-formed* if all its trajectories $\mathsf{traj}(G, r)$ are well-formed. We define $\mathcal{E}_{\text{wellformed}}$ to be the set of well-formed executions.

For protocols where all calls return immediately (as part of the same transition), all executions are well-formed (Lemma 7.1). In general, however, an incorrect protocol implementation may contain spurious return transitions. When proving the correctness of protocols, we thus need to also prove that all its executions are well-formed.

Lemma 7.1. Let $G = (E, \mathsf{eo}, \mathsf{tr}, \mathsf{role}, \mathsf{del})$ be an execution such that all calls return immediately: $\mathsf{calls}(E) = \mathsf{returns}(E)$. Then G is well-formed.

7.4 Observable History

We now show how to characterize the observable behavior of an execution using histories as introduced in §3.1 and defined in Definition 3.1 on page 32. We obtain a history from a concrete execution by selecting the operation call events, and recording information about them.

Auxiliary notations. The following notations help us to express properties of events in a concrete execution $G = (E, \mathsf{eo}, \mathsf{tr}, \mathsf{role}, \mathsf{del})$:

- We define the set of operation call events

$$\mathsf{ops}(E) \stackrel{\text{def}}{=} \{e \in E \mid \mathsf{op}(e) \neq \bot \}$$

- For an event $e \in E$, we define $\mathsf{call}(e)$ to be the preceding call in the same role, $\mathsf{ret}(e)$ to be the succeeding return in the same role:

$$\mathsf{call}(e) = \max_{\mathsf{ro}} \left\{ c \in E \;\middle|\; c \leq_{\mathsf{ro}} e \;\wedge\; \mathsf{op}(c) \neq \bot \right\}$$
$$\mathsf{ret}(e) = \min_{\mathsf{ro}} \left\{ r \in E \;\middle|\; e \leq_{\mathsf{ro}} r \;\wedge\; \mathsf{rval}(r) \neq \bot \right\}$$

  Note that $\mathsf{call}(e) = e$ if e is a call, and $\mathsf{call}(e) = \bot$ if there is no preceding call, and similarly for $\mathsf{ret}(e)$.

We can now define how we obtain the observable history $\mathcal{H}(G)$ of a concrete execution G.

Definition 7.9. Let $G = (E, \mathsf{eo}, \mathsf{tr}, \mathsf{role}, \mathsf{del})$ be a concrete execution. Then we define the observable history of G to be the event graph $\mathcal{H}(G) \stackrel{\text{def}}{=} (E_A, \mathsf{op}_A, \mathsf{rval}_A, \mathsf{rb}_A, \mathsf{ss}_A)$ that records the following information:

(x1) $E_A \stackrel{\text{def}}{=} \mathsf{ops}(G)$ (the events are the operation calls in G)

(x2) $\mathsf{op}_A(e) \stackrel{\text{def}}{=} \mathsf{op}(\mathsf{tr}(e))$ (we record the operation performed)

(x3) $\mathsf{rval}_A(e) \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } \mathsf{op}(e) = \bot \\ \mathsf{rval}(\mathsf{ret}(e)) & \text{if } \mathsf{ret}(e) \neq \bot \\ \nabla & \text{otherwise} \end{cases}$
(we record for an operation call e the value returned by the matching return, or ∇ if it did not return)

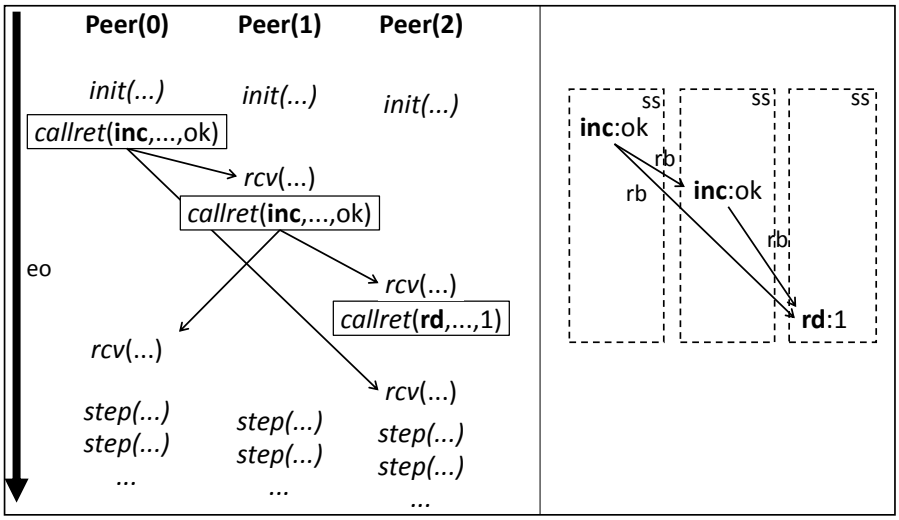
(x4) $e \xrightarrow{\mathsf{rb_A}} e' \quad \stackrel{\text{def}}{\iff} \quad (\mathsf{ret}(e) \neq \bot) \wedge (\mathsf{ret}(e) \xrightarrow{\mathsf{eo}} e')$
(the returns-before relation captures whether operation e returned before operation e' was called)

(x5) $e \approx_{ss_A} e' \quad \stackrel{\text{def}}{\iff} \quad e \approx_{\mathsf{role}} e'$
(the same-session relation captures whether operations happened on the same role)

If there is no risk of confusion, we abbreviate op_A as op, and so on (but note that $\mathsf{rval}_A \neq \mathsf{rval}$ in general). The following lemma shows that the definition above works as intended, *i.e.* produces histories as defined in Definition 3.1 on page 32.

Lemma 7.2. For any concrete execution $G \in \mathcal{E}$, $\mathcal{H}(G)$ is a history, and if G is well-formed, then $\mathcal{H}(G)$ is well-formed.

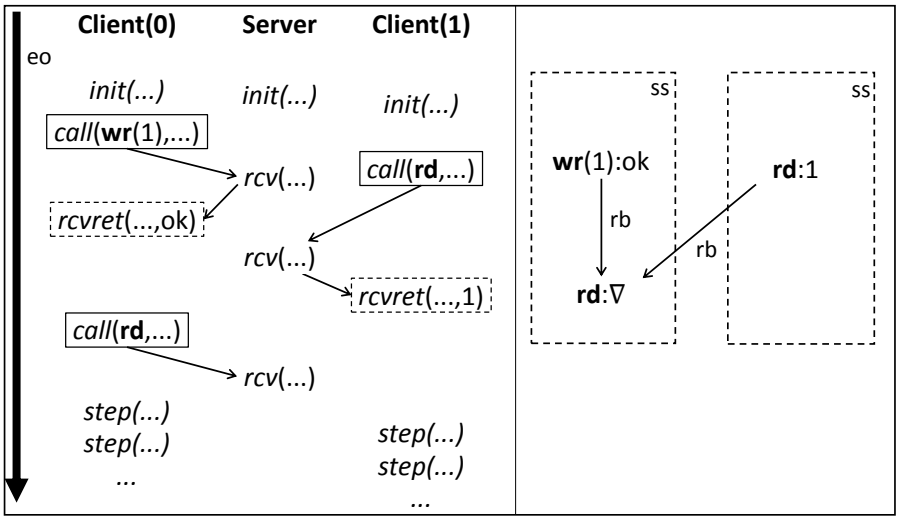
Example 1: Broadcast Counter



Concrete Execution. On the left, we show a graphical representation of a concrete execution $G = (E, \mathsf{eo}, \mathsf{tr}, \mathsf{role}, \mathsf{del})$ of the BroadcastCounter (see code in Fig. 6.3 on page 69). The event graph contains a vertex for each transition of each role. Transitions are aligned in columns according to the role they belong to, and are aligned vertically according to execution order. We have put frames around the call transitions $\mathsf{ops}(G)$. In this implementation, all operation calls return immediately, thus all call events are a transition of the form $\mathit{callret}(o, \sigma, \sigma', M, v)$. To reduce the level of detail, we have omitted all arguments except o (the operation being called) and v (the returned value). This execution is infinite, and complete: all participating roles (*i.e.* all roles that execute at least one transition) execute infinitely many transitions (indicated by dots at each column's bottom).

History. On the right, we show a graphical represention of the corresponding history $\mathcal{H}(G) = (E_A, \mathsf{op}_A, \mathsf{rval}_A, \mathsf{rb}_A, \mathsf{ss}_A)$. The events correspond to the operation calls $E_A = \mathsf{ops}(G)$ (framed transitions), the operation and return values are copied from the concrete execution, and the returns-before order matches the order in which the operations appear in the concrete execution.

Example 2: Single-Copy Register



Concrete Execution. On the left, we show a concrete execution of the single-copy register (Fig. 1.1 on page 11). The call and return of an operation happens in separate transitions (we highlight calls and returns using solid or dashed frames, respectively), and there are no calls or returns on the server, which only responds to messages by the clients. This execution is not complete: the server crashes (*i.e.* executes only a finite number of transitions and then stops) and the client on the left waits forever for a response that does not arrive.

History. On the right, we show the corresponding history. The history shows the three operations that are called in the concrete executions. It combines information from the call and return events into a single event. The fact that the second read did not return is indicated by rval_A being ∇. Note that the first write and the first read are not ordered by ar because their durations overlap.

This history is not a valid history for the register data type: it cannot be extended to an abstract execution that satisfies $\textsc{RVal}(\mathcal{F}_{reg})$ because read operations are not allowed to return ∇ (unlike, for example, dequeue operations on queues), as specified in Fig. 4.1 on page 42.

7.5 Infinite Executions

Finite executions represent the behavior of the system during a limited time interval, starting at the beginning (before any roles are initialized) and ending at some arbitrary point of time (perhaps in the middle of some pending operations). In contrast, infinite executions represent what the system does when left running forever. To define our consistency models, we need to work with infinite executions, because the consistency guarantees include liveness aspects such as "each operation call eventually returns" or "each operation eventually becomes visible to all sessions".

In this section, we examine (1) how finite and infinite executions are related to each other, via the notion of prefixes and limits, and (2) how to distinguish between safety properties and liveness properties. In addition to their philosophical interest, these concepts are also useful when we prove general implementability results, as in chapter 9.

7.5.1 Prefixes and Limits

We can easily take a prefix of any execution, and of any finite length desired (less than or equal to the length of the original execution). This prefix is always a finite execution itself.

Definition 7.10. Let $G = (E, \mathsf{eo}, \mathsf{tr}, \mathsf{role}, \mathsf{del})$ be a concrete execution, with an enumerable set of events $E = \{e_0, e_1, \dots\}$ (where $e_i \xrightarrow{\mathsf{eo}} e_j \Leftrightarrow i < j$), and let $E' = \{e_0, \dots, e_k\} \subseteq_{\text{fin}} E$. Then, the event graph $G' = G|_{E',\mathsf{eo},\mathsf{tr},\mathsf{role},\mathsf{del}}$ is called a *prefix* of G, written $G' \sqsubseteq G$. We define $\mathsf{prefixes}(G)$ to be the set of all prefixes of G.

Lemma 7.3. Let $G \in \mathcal{E}$ and $G' \sqsubseteq G$. Then $G' \in \mathcal{E}_{\text{fin}}$.

Conversely, we can take the limit of a sequence of prefixes. This limit is always a concrete execution.

Definition 7.11. Let $G_0 \sqsubseteq G_1 \sqsubseteq G_2 \dots$ be an infinite ascending sequence of finite concrete executions. Then, define its limit as:

$$\lim_i G_i \stackrel{\text{def}}{=} (\bigcup_i E_i, \bigcup_i \mathsf{eo}, \bigcup_i \mathsf{tr}, \bigcup_i \mathsf{role}, \bigcup_i \mathsf{del}).$$

Lemma 7.4. If $G_0 \sqsubseteq G_1 \sqsubseteq \cdots$ and $G_i \in \mathcal{E}_{\text{fin}}$, then $\lim_i G_i \in \mathcal{E}$.

7.5.2 Safety and Liveness

When specifying and verifying properties, we often distinguish between safety properties (which describe "bad" things that should never happen) and liveness properties (which describe "good" things that should eventually happen). We formally define safety and liveness as follows.

Definition 7.12. Let $\mathcal{P} \subseteq \mathcal{E}$ be a subset of executions. Then,

- $\mathcal{P}$ is called a *safety property* if the membership of an execution is completely determined by its finite prefixes:

  $$\forall G \in \mathcal{E} : \big(G \in \mathcal{P} \quad \Leftrightarrow \quad \forall G' \in \mathsf{prefixes}(G) : G' \in \mathcal{P}\big)$$

  or equivalently stated, if membership can always be refuted by a finite prefix:

  $$\forall G \in \mathcal{E} : \big(G \notin \mathcal{P} \quad \Leftrightarrow \quad \exists G' \in \mathsf{prefixes}(G) : G' \notin \mathcal{P}\big)$$

- $\mathcal{P}$ is called a *liveness property* if any finite execution can be extended to an execution that satisfies the property:

  $$\forall G \in \mathcal{E}_{\text{fin}} : \exists G' \in \mathcal{P} : G \in \mathsf{prefixes}(G').$$

Interestingly, one can show that *any* property is a conjunction of a safety property and a liveness property [Alpern and Schneider, 1985].

Examples. We show a number of interesting safety and liveness properties in the section on transport layer guarantees §8.2. Also, well-formedness (Definition 7.8) is a safety property.

Lemma 7.5. $\mathcal{E}_{\text{wellformed}}$ is a safety property.

8

Protocols

In the previous chapter, we have shown how to model executions of any asynchronous distributed protocol. The next step is to formalize the notion of a *protocol*. This will allow us to not only study individual protocols, but to ask deeper questions, such as whether a protocol with certain properties can even exist.

A protocol has several components: it defines what roles participate, the format of messages, a state machine (called role automaton) for each participating role, and the required message delivery guarantees.

8.1 Role Automata

The transitions taken by each role must be consistent with the *role automaton* that the protocol defines for that role. A role automaton is a set of transitions, subject to conditions that ensure that a role correctly models the asynchronous sending and receiving of messages, the asynchronous operation calls and returns (from and to the client program), and the scheduling of processes.

Definition 8.1 (Role Automaton)**.** Let $O \subseteq \mathsf{Operations}$ be a set of operations, let $\Sigma \subseteq \mathsf{States}$ be a set of states, let $M \subseteq \mathsf{Messages}$ be a set of

messages, and let $P \subseteq \mathsf{Processes}$ be a set of processes. A *role automaton* T over (O, Σ, M, P) is a set of transitions $T \subseteq \mathsf{Transitions}$ that satisfies the following properties:

(r1) The operations, states, and messages range over O, Σ, M, P:

$$\begin{array}{lll} \mathsf{op}(T) \subseteq O & \mathsf{pre}(T) \subseteq \Sigma & \mathsf{post}(T) \subseteq \Sigma \\ \mathsf{snd}(T) \subseteq \mathcal{P}(M) & \mathsf{rcv}(T) \subseteq M & \mathsf{proc}(T) \subseteq P \end{array}$$

(r2) There is at least one initialization transition:

$$\exists \sigma, M : \mathit{init}(\sigma, M) \in T$$

(r3) All messages can be received in all states:

$$\forall m \in M : \forall \sigma \in \Sigma : \exists t \in T : (\ \mathsf{rcv}(t) = m \ \wedge \ \mathsf{pre}(t) = \sigma\ )$$

(r4) All operations can be called in all states:

$$\forall o \in O : \forall \sigma \in \Sigma : \exists t \in T : (\ \mathsf{op}(t) = o \ \wedge \ \mathsf{pre}(t) = \sigma\ )$$

(r5) All processes can take a step in all states:

$$\forall p \in P : \forall \sigma \in \Sigma : \exists t \in T : (\ \mathsf{proc}(t) = p \ \wedge \ \mathsf{pre}(t) = \sigma\ )$$

Condition (r2) expresses that there must be at least one initialization transition. There can be more than one if we want to initialize the automaton nondeterministically.

Condition (r3) expresses that a role must always accept *any* message. This requirement can be easily satisfied by including transitions of the form $\mathit{rcv}(m, \sigma, \sigma)$ that "ignore" unwanted messages, such as messages that aren't intended for this particular receiver. Note that if a node would like to process an incoming message at a later point of time, it can store the message in a buffer that is part of its state, and use a dedicated process with its own step transition to process it later.

Condition (r4) expresses that a role must always accept *any* calls from the client. Just as for messages, the automaton may ignore calls that are invalid in some sense by using a transition $\mathit{call}(o, \sigma, \sigma)$, or it may produce an error message using a transition along the lines of $\mathit{callret}(o, \mathit{Error}(\text{invalid call}), \sigma, \sigma)$.

Condition (r5) expresses that each process in a role must be able to "take a step". There may be multiple such transitions — if the implementation is nondeterministic. If a process p has no work to do in state σ, it can satisfy condition (r5) by providing an "idle" transition $\mathit{step}(\sigma, \sigma, p)$. Idle transitions correspond to what is called "stuttering" in state-oriented models (such as used by Abadi and Lamport [1991]).

Fairness. At runtime, the system must ensure that each process in a role gets scheduled repeatedly to give it a chance to make progress (such as sending a message, or returning from a pending operation).

Definition 8.2. We say a trajectory $G = (E, \mathsf{eo}, \mathsf{tr})$ is *fair* to a process $p \in \mathsf{Processes}$ if it is either finite, or contains infinitely many transitions for p:

$$\mathsf{fair}(G, p) \quad \stackrel{\text{def}}{\iff} \quad |E| < \infty \quad \vee \quad |\{e \in E \mid \mathsf{proc}(e) = p\}| = \infty.$$

Our processes are simply labels on transitions that establish fair scheduling of transitions within a single role. Thus, process labels have no meaning across multiple roles. Processes cannot crash, but the role containing them may (which could be considered a simultaneous crash of all its processes). Our process labels correspond to the equivalence relation used in I/O automata (Lynch and Tuttle [1989]) to ensure fairness. For our processes, strong and weak fairness (*e.g.* as described in Lamport [1994]) coincide, because condition (r5) guarantees that each process is always enabled.

8.2 Transport Guarantees

No meaningful protocol can work correctly without some system-wide guarantees about message delivery. Such guarantees include (1) safety guarantees, which prevent the forging, loss, duplication, or reordering of messages, and (2) liveness guarantees, which ensure that messages are eventually delivered, or eventually indirectly delivered.

Definition 8.3 (List of Transport Guarantees)**.** We define a *transport guarantee* to be one of the following predicates over a concrete execution $G = (E, \mathsf{eo}, \mathsf{tr}, \mathsf{role}, \mathsf{del})$, parameterized by some set of messages $M \subseteq \mathsf{Messages}$:

- *dontforge*(M) requires that messages in M that are received must have been sent, *i.e.* not invented out of thin air:

$$\forall e : ((\mathsf{act}(e) = rcv \;\; \wedge \;\; \mathsf{rcv}(e) \in M) \;\; \Rightarrow \;\; \exists e' : (e' \xrightarrow{\mathsf{del}} e))$$

- *dontduplicate*(M) requires that no message in M sent by an event e is delivered twice to the same role:

$$\forall e, e_1, e_2 : \forall m \in M :$$
$$((e \xrightarrow{\mathsf{del(m)}} e_1) \wedge (e \xrightarrow{\mathsf{del(m)}} e_2) \wedge (e_1 \approx_{\mathsf{role}} e_2)) \quad \Rightarrow \quad (e_1 = e_2)$$

- *dontlose*(M) requires that messages in M with correct sender are eventually delivered to all correct receivers (other than the sender):

$$\forall s, r \in \mathsf{correct}(G) \text{ where } s \neq r : \quad \forall e \in E(s) : \forall m \in M :$$
$$m \in \mathsf{snd}(e) \quad \Rightarrow \quad \exists e' \in E(r) : \;\; e \xrightarrow{\mathsf{del(m)}} e'$$

  This guarantee applies only to correct (*i.e.* non-crashed) roles, because a crashed receiver cannot perform a receive transition, and for a crashed sender, the crash may prevent the message from propagating (even if the crash occurs after the send operation).

- *pairwiseordered*(M) requires that the order of messages in M by the same sender to the same receiver is preserved:

$$\forall s_1, s_2, r_1, r_2 :$$
$$((s_1 \xrightarrow{\mathsf{ro}} s_2) \wedge (s_1 \xrightarrow{\mathsf{del(M)}} r_1) \wedge (s_2 \xrightarrow{\mathsf{del(M)}} r_2) \wedge (r_1 \approx_{\mathsf{role}} r_2)$$
$$\Rightarrow \quad (r_1 \xrightarrow{\mathsf{ro}} r_2))$$

- *reliable*(M) and *reliablestream*(M) require that all messages in M are delivered exactly once, and in-order (for the latter):

$$\textit{reliable}(M) \stackrel{\text{def}}{\Longleftrightarrow} \textit{dontforge}(M) \wedge \textit{dontduplicate}(M) \wedge \textit{dontlose}(M)$$
$$\textit{reliablestream}(M) \stackrel{\text{def}}{\Longleftrightarrow} \textit{reliable}(M) \wedge \textit{pairwiseordered}(M)$$

- *eventual*(M) requires that a correct, stubborn sender (*i.e.* a sender sending infinitely many messages in M) can eventually get a message through to any correct receiver:

$$\forall s, r \in \mathsf{correct}(G) \text{ where } s \neq r : \\ \left|\{e \in E(s) \mid \mathsf{snd}(e) \cap M \neq \emptyset\}\right| = \infty \quad \Rightarrow \\ \forall e \in E(s) : \exists e' \in E(r) : (e \xrightarrow{\mathsf{ro}\,;\ \mathsf{del(M)}} e')$$

- *eventualindirect*(M) requires that a correct, stubborn sender (*i.e.* a sender sending infinitely many messages in M) can eventually reach any correct receiver, possibly via other roles, through a chain of messages:

$$\forall s, r \in \mathsf{correct}(G) \text{ where } s \neq r : \\ \left|\{e \in E(s) \mid \mathsf{snd}(e) \cap M \neq \emptyset\}\right| = \infty \quad \Rightarrow \\ \forall e \in E(s) : \exists e' \in E(r) : (e \xrightarrow{(\mathsf{ro} \cup \mathsf{del(M)})^*} e')$$

The guarantee *eventualindirect*(M) is slightly weaker (and slightly easier to provide) than *eventual*(M). Yet it is still sufficient for the important category of epidemic protocols (§6.1.1), where direct delivery of messages between all pairs is not required.

Lemma 8.1 (Safety and Liveness)**.** The three transport guarantees *dontforge*(M), *dontduplicate*(M), and *pairwiseordered*(M) are safety properties. The three transport guarantees *dontlose*(M), *eventual*(M), and *eventualindirect*(M) are liveness properties.

8.3 Protocols

We can now assemble our definition of a protocol, by combining the concepts of a role automaton (Definition 8.1 on page 95), and transport guarantees (Definition 8.3 on page 97).

Definition 8.4. A *protocol* is a tuple $(R, M, O, \Sigma, P, S, T)$ where

- $R \subseteq \mathsf{Roles}$ is a set of role instances.
- $M \subseteq \mathsf{Messages}$ is a set of messages.
- O maps each $r \in R$ to a set $O_r \subseteq \mathsf{Operations}$ of operations.
- Σ maps each $r \in R$ to a set of states $\Sigma_r \subseteq \mathsf{States}$.
- P maps each $r \in R$ to a set of processes $\emptyset \neq P_r \subseteq_{\text{fin}} \mathsf{Processes}$.
- $S = S_1 \wedge \cdots \wedge S_n$ is a conjunction of transport guarantees.
- T maps each $r \in R$ to a role automaton T_r over (O_r, M, Σ_r, P_r).

We show two examples of protocols in Fig. 6.4 (the broadcast counter protocol) and Fig. 6.6 (the epidemic counter protocol) on pp. 69–70.

Protocol Executions. Protocols precisely define what executions are possible. These executions are of the form as defined in Definition 7.5 on p. 87, and consistent with the specified role automata and the transport guarantees:

Definition 8.5. For a protocol $\Pi = (R, M, O, \Sigma, P, S, T)$, define the set of its executions $\mathcal{E}(P)$ to be the set of executions $G = (E, \mathsf{eo}, \mathsf{tr}, \mathsf{role}, \mathsf{del})$ such that

(e1) The transitions of each role are as specified by its role automaton: $\forall e \in E : \mathsf{tr}(e) \in T_{\mathsf{role}(e)}$.

(e2) The execution satisifes each transport guarantee: $\forall i : G \models S_i$.

(e3) All trajectories are fair, for all processes of that role: $\forall r \in R : \forall p \in P_r : \mathsf{fair}(\mathsf{traj}(G, r), p)$.

We define the sets of finite / complete protocol executions as

$$\mathcal{E}_{\text{fin}}(\Pi) \stackrel{\text{def}}{=} \mathcal{E}(\Pi) \cap \mathcal{E}_{\text{fin}} \qquad \mathcal{E}_{\text{complete}}(\Pi) \stackrel{\text{def}}{=} \mathcal{E}(\Pi) \cap \mathcal{E}_{\text{complete}}$$

8.4 Pseudocode Compilation

Throughout this tutorial, and in particular in chapter 6, we have described many different protocols using pseudocode. To achieve reasonable precision of language without providing a formal definition of a language syntax and semantics (which is beyond the scope of this tutorial), we resort to an informal, but precise description of a *compilation process* that takes a pseudocode protocol definition and compiles it into a formal protocol (as in Definition 8.4).

We now discuss this compilation process. Much of it can be gleaned from the examples given in Figures 6.4 and 6.6 on pp. 69–70, which show the input and the output of the compilation process.

- The set R of role instances is determined by the role definitions in the protocol. For example, a definition **role** Peer(nr:nat) translates to the set $\{Peer(i) \mid i \in \mathbb{N}_0\}$.
- The set M of messages is determined by the message definitions in the protocol. For example, a definition **message** Latest(c: tmap<nat,nat>) translates to the set $\{Latest(c) \mid c \in (R \to_{\text{fin}} \mathbb{N}_0)\}$.
- The sets O_r, for $r \in R$, are determined by the operations that appear in the definition of the role of which r is an instance.
- The sets Σ_r, for $r \in R$, represents the state of a role. We prefix the role identifier with an S to denote the state. For example, $\Sigma_{Peer(i)} = \{SPeer(c) \mid c \in \mathbb{N}_0\}$ for a role whose state is an integer (Fig. 6.4).
- The set of processes P_r is defined to contain a process for each periodic task, or if there aren't any, a single background process $\{bg\}$.
- The transport guarantees S match the reliability attributes of the messages.

The interesting part is how we obtain the role automaton T_r. The key insight here is that every handler (preceded by the keyword **operation**,

receive, **periodically**, or **init**) executes atomically, as a single transition. Thus, we obtain the set of transitions as follows:

- For every operation $o \in O_r$ and state $\sigma \in \Sigma_r$, we create a call transition. The body of the handler determines the set of sent messages (one per **send** statement in the body), updated state (side effect of executing the code), and return value (if it contains a **return** statement).

- For every message $m \in M$ and state $\sigma \in \Sigma_r$, we create a receive transition. If there is a handler matching m, we use the body of the handler to determine sent messages, updated state, and return value (if any). Otherwise, we use a dummy transition $\mathit{rcv}(m, \sigma, \sigma, \emptyset)$.

- For every process $p \in P_r$ and state $\sigma \in \Sigma_r$, we create a step transition. If the process corresponds to some **periodically** clause, the latter determines sent messages, updated state, and return value (if any). Otherwise, we create the idle transition $\mathit{step}(p, \sigma, \sigma, \emptyset)$.

- We create a transition $\mathit{init}(\sigma, \emptyset)$ where σ is the state where all fields assume the default value defined by their type, or, if the role specifies an initialization handler, we use that handler to create the transition.

9

Implementability

We can now state precisely what it means for a protocol to be correct, by connecting the formalisms we use for consistency guarantees (Chapters 3 – 5) and for executions of protocols (Chapters 7 and 8).

Definition 9.1 (Correctness)**.** We say that a protocol Π *guarantees* a consistency model $\mathcal{P}_1 \wedge \cdots \wedge \mathcal{P}_n$ if it produces only well-formed executions (Def. 7.8, p. 89), and if all of its complete executions satisfy the consistency guarantees (Def. 3.5, p. 35):

$$\Pi \models \mathcal{P}_1 \wedge \cdots \wedge \mathcal{P}_n \quad \stackrel{\text{def}}{\iff} \quad \mathcal{E}(\Pi) \subseteq \mathcal{E}_{\text{wellformed}} \ \wedge \\ \forall G \in \mathcal{E}_{\text{complete}}(\Pi): \quad \mathcal{H}(G) \models \mathcal{P}_1 \wedge \cdots \wedge \mathcal{P}_n$$

With this definition in place, it becomes possible to ask and answer many interesting questions. For example, we can formalize protocols and verify rigorously whether they do indeed provide the guarantees we think they should. We show correctness proofs for the protocols introduced earlier in chapter 10.

But beyond that, we can also ask the question whether certain guarantees are even implementable by *any* protocol. We start this section by proving a general version of the CAP theorem.

9.1 CAP

To provide clients continous access to data even when the network is temporarily unavailable, roles must complete client operations without first waiting for responses from remote nodes. Whenever this is possible, we can make such operations atomic.

Definition 9.2. Let $\Pi = (R, M, O, \Sigma, P, S, T)$ be a protocol. We say that an operation o is *available under partitions* on a role r if $o \in O_r$ and all calls are atomic:

$$\mathsf{availableunderpartitions}(\Pi, o, r) \quad \stackrel{\text{def}}{\iff} \quad (o \in O_r) \wedge (\forall t \in T_r : \mathsf{op}(r) = o \;\Rightarrow\; \mathsf{rval}(t) \neq \bot).$$

9.1.1 CAP Sketch

We start with a sketch of the proof structure that is helpful to understand the mechanics of the proof. Consider a key-value store. Is it possible to make both reads and writes available under partitions while guaranteeing sequential consistency? The answer is no. To see why, assume that it is indeed possible, and derive a contradiction. Consider the Dekker test (§5.3.2), and consider a situation where we have two roles (A and B), and run three experiments called A, B, and AB. In each experiment, we simulate a network partition by not delivering any messages until all code has finished executing. In experiment A, we run program A only (on role A). In experiment B, we run program B only (on role B), and in experiment AB, we run both. Now, observe that (1) in experiment A, the program must print `"A wins"`, and (2) in experiment B, the program must print `"B wins"`. But since there are no messages exchanged until the program finished executing, it means that in experiment AB, we must see both `"A wins"` and `"B wins"`, contradicting sequential consistency.

9.1.2 General CAP

Since CAP does not apply to all data types (for example, we show that it is possible to ipmlement a sequentially consistent register that is

available under partitions in §10.2.2), we first characterize what property of the data type causes the problem.

Definition 9.3. Given some replicated data type $\mathcal{F}_\tau$, we say that two operations $w_a, w_b \in \mathsf{Operations}$ are *independently observable* by two operations $r_a, r_b \in \mathsf{Operations}$ if for all operation contexts of the form

$$C_a = (\{e_a\}, \{(e_a, w_a)\}, \emptyset, \emptyset) \qquad C_b = (\{e_b\}, \{(e_b, w_b)\}, \emptyset, \emptyset)$$
$$C_{ab} = (\{e_a, e_b\}, \{(e_a, w_a), (e_b, w_b)\}, \mathsf{vis}, \mathsf{ar}) \quad (\mathsf{vis}, \mathsf{ar} \text{ are arbitrary})$$

the operations r_a, r_b can distinguish between the contexts:

$$\mathcal{F}_\tau(r_a, C_{ab}) \neq \mathcal{F}_\tau(r_a, C_b) \quad \text{and} \quad \mathcal{F}_\tau(r_b, C_{ab}) \neq \mathcal{F}_\tau(r_b, C_a)$$

Both the key-value store $\mathcal{F}_{\text{kvs}}$ and the counter have independently observable updates:

- $w_a = \mathsf{inc}$ and $w_b = \mathsf{inc}$ are independently observable updates for the counter $\mathcal{F}_{ctr}$, because $\mathcal{F}_{ctr}(\mathsf{rd}, C_a) = \mathcal{F}_{ctr}(\mathsf{rd}, C_b) = 1 \neq 2 = \mathcal{F}_{ctr}(\mathsf{rd}, C_{ab})$.
- $w_a = \mathsf{wr}(a, 1)$ and $w_b = \mathsf{wr}(b, 1)$ for two objects $a \neq b$ are independently observable.

However, the register $\mathcal{F}_{reg}$ does not have independently observable updates; even if choosing two different writes $w_a = \mathsf{wr}(1)$ and $w_b = \mathsf{wr}(2)$, we still have $\mathcal{F}_\tau(\mathsf{rd}, C_{ab}) = \mathcal{F}_\tau(\mathsf{rd}, C_b)$ if we let ar in C_{ab} order the events as $e_a \xrightarrow{\mathsf{ar}} e_b$. In fact, this is a good thing: since we presented a sequentially consistent register in the introduction, we better not prove now that it is impossible.

We are now ready to state and prove the CAP theorem for any data type with independently observable writes.

Theorem 9.1 (SC-CAP Core)**.** Let $\Pi = (R, M, O, \Sigma, P, S, T)$ be a sequentially consistent protocol for a data type $\mathcal{F}_\tau$ with at least two roles, *i.e.*

$$\Pi \models \textsc{SequentialConsistency}(\mathcal{F}_\tau) \quad \text{and} \quad |R| \geq 2,$$

and let $w_a, w_b \in O$ be independently observable by $r_a, r_b \in O$. Then, there exists a role $r \in R$ on which at least one of the operations w_a, w_b, r_a, r_b is not available under partitions.

Corollary 9.2. For neither the counter $\mathcal{F}_{ctr}$, nor the key-value store $\mathcal{F}_{\text{kvs}}$, do there exist any sequentially consistent implementations that make all operations available under partitions on more than one role.

Theorem 9.1. We proceed indirectly: assuming that w_a, w_b, r_a, r_b are available under partitions on all roles, we construct a concrete execution that is not sequentially consistent. We break this down into three steps. In the first two steps, we separately construct trajectories J_x and J_y for two roles $x, y \in R$ such that $x \neq y$, and show that trajectory J_x ends with an r_b operation that returns $\mathcal{F}_\tau(r_b, C_a)$ (for C_a as in Definition 9.3), and trajectory J_y ends with a r_a operation that returns $\mathcal{F}_\tau(r_a, C_b)$. Then, in a third step, we construct a complete concrete execution G'_{xy} such that $\mathcal{H}(G'_{xy})$ is not contained in $\mathcal{H}(\textsc{SequentialConsistency}(\mathcal{F}_\tau))$, which is a contradiction that completes the proof.

Step 1. Consider the role automaton T_x. By condition (r2) on p. 96, it must contain an initialization transition $t_0 = \mathit{init}(\sigma_0, M_0)$. Now, by condition (r4), and by the assumption that w_a is available under partitions, there must exist a transition $t_1 = \mathit{callret}(w_a, M_1, \sigma_0, \sigma_1, v_1) \in T_x$. Similarly, by the assumption that r_b is available under partitions, there must exist a transition $t_2 = \mathit{callret}(r_b, M_2, \sigma_1, \sigma_2, v_2) \in T_x$. Then the tuple $J_x \stackrel{\text{def}}{=} (\{x_0, x_1, x_2\}, \mathsf{eo}, \mathsf{tr})$ where $x_0 <_{\mathsf{eo}} x_1 <_{\mathsf{eo}} x_2$ and $\mathsf{tr}(x_i) = t_i$ is a trajectory for the role automaton T_x.

We now argue that $v_2 = \mathcal{F}_\tau(r_b, C_a)$ for some operation context C_a as in Definition 9.3. To see why, note that we can extend the trajectory to a finite concrete execution $G_x = (\{x_0, x_1, x_2\}, \mathsf{eo}, \mathsf{tr}, \mathsf{role}, \mathsf{del})$ where $\mathsf{role}(x_i) = x$ and $\mathsf{del} = \emptyset$ (no other roles perform transitions, no messages are delivered). Then $G_x \in \mathcal{E}(\Pi)$ because (1) the events for each role are a trajectory for T_x, (2) all transport guarantees are satisfied because no messages are delivered (thus safety is not violated) and the execution is finite (thus $\mathsf{correct}(G_x) = \emptyset$ and thus liveness is not violated), and (3) the execution is fair because it is finite. Its observable history is $\mathcal{H}(G_x) = (\{x_1, x_2\}, \mathsf{op}, \mathsf{rval}, \mathsf{rb}, \mathsf{ss})$ where $\mathsf{op}(x_1) = w_a$, $\mathsf{op}(x_2) = r_b$, $\mathsf{rval}(x_i) = v_i$, $x_1 \xrightarrow{\mathsf{rb}} x_2$, and $x_1 \approx_{ss} x_2$. Because of the progress theorem 9.3, which we prove in the next chapter, we can extend G_x to a complete execution $G_x \sqsubseteq G'_x \in \mathcal{E}_{\text{complete}}(\Pi)$ such that $\mathcal{H}(G_x) = \mathcal{H}(G'_x)$.

Now, because $\Pi \models \text{SEQUENTIALCONSISTENCY}(\mathcal{F}_\tau)$, we know there exists an abstract execution A_x that is the same as $\mathcal{H}(G'_x) = \mathcal{H}(G_x)$, but with added relations vis and ar (where vis is acyclic and ar is total), and that satisfies conditions SINGLEORDER,READMYWRITES and $\text{RVAL}(\mathcal{F}_\tau)$. Because $x_1 \xrightarrow{\mathsf{rb}} x_2$ and $x_1 \approx_{ss} x_2$, we have $x_1 \xrightarrow{\mathsf{so}} x_2$, and thus by READMYWRITES, $x_1 \xrightarrow{\mathsf{vis}} x_2$. Thus the context (as defined on page 50) for x_2 is $\mathsf{context}(A_x, e) = (\{x_1\}, \mathsf{op}, \emptyset, \emptyset)$ with $\mathsf{op}(x_1) = w_a$. It is thus of the form stated in Definition 9.3, which concludes step 1.

Step 2. Consider the role automaton T_y. Symmetric to step 1, using the same reasoning but swapping a and b in the indexes of the operations w_a and r_b, we obtain a trajectory $J_y = (\{y_0, y_1, y_2\}, \mathsf{eo}, \mathsf{tr})$ of the role automaton T_y, and where $y_0 \xrightarrow{\mathsf{eo}} y_1 \xrightarrow{\mathsf{eo}} y_2$, $\mathsf{tr}(y_1) = \mathit{callret}(w_b, v'_1, \sigma'_0, \sigma'_1)$, $\mathsf{tr}(y_2) = \mathit{callret}(r_a, v'_2, \sigma'_1, \sigma'_2)$ is a trajectory for the role automaton T_y, and where $v'_2 = \mathcal{F}_\tau(r_a, C_b)$ for some operation context C_b as in Definition 9.3.

Step 3. We take the two trajectories J_x, J_y from steps 1 and 2 and combine them into a single, finite concrete execution $G_{xy} = (\{x_0, x_1, x_2, y_0, y_1, y_2\}, \mathsf{eo}, \mathsf{tr}, \mathsf{role}, \mathsf{del})$ where $x_0 <_{\mathsf{eo}} x_1 <_{\mathsf{eo}} x_2 <_{\mathsf{eo}} y_0 <_{\mathsf{eo}} y_1 <_{\mathsf{eo}} y_2$, tr and role are as defined before in steps 1 and 2, and $\mathsf{del} = \emptyset$ (no messages are delivered). Then, $G_{xy} \in \mathcal{E}(\Pi)$ because (1) the events for each role are a trajectory, (2) all transport guarantees are satisfied because no messages are delivered (thus safety is not violated) and the execution is finite (thus $\mathsf{correct}(G_x) = \emptyset$ and thus liveness is not violated), and (3) the execution is fair because it is finite. The observable history is $\mathcal{H}(G_{xy}) = (\{x_1, x_2, y_1, y_2\}, \mathsf{op}, \mathsf{rval}, \mathsf{rb}, \mathsf{ss})$ with $\mathsf{op}, \mathsf{rval}, \mathsf{ss}$ the same as defined previously in steps 1 and 2, and with $\mathsf{rb} = \mathsf{eo}$. Using the progress theorem 9.3, we can extend to a complete concrete execution $G_{xy} \sqsubseteq G'_{xy} \in \mathcal{E}_{\text{complete}}(\Pi)$ with $\mathcal{H}(G'_{xy}) = \mathcal{H}(G_{xy})$.

Finally, it remains to be shown that $\mathcal{H}(G_{xy}) \not\models \text{SEQUENTIALCONSISTENCY}(\mathcal{F}_\tau)$ which concludes the proof by contradicting $\Pi \models \text{SEQUENTIALCONSISTENCY}(\mathcal{F}_\tau)$. We proceed indirectly: assume that $\mathcal{H}(G_{xy}) \models \text{SEQUENTIALCONSISTENCY}(\mathcal{F}_\tau)$. This means there exists an abstract execution A_{xy} that is the same as $\mathcal{H}(G_{xy})$, but with added relations vis and ar, and that satisfies both conditions SINGLEORDER, READMYWRITES and $\text{RVAL}(\mathcal{F}_\tau)$. Then we

must have $x_1 \xrightarrow{\mathsf{vis}} x_2$ and $y_1 \xrightarrow{\mathsf{vis}} y_2$ (because ReadMyWrites implies $\mathsf{so} \subseteq \mathsf{vis}$). Also, we must have either $x_1 \xrightarrow{\mathsf{vis}} y_1$ or $y_1 \xrightarrow{\mathsf{vis}} x_1$ (because SingleOrder implies that vis is a total order on all operations $e \in \mathsf{ops}(G)$ such that $\mathsf{rval}_G(e) \neq \nabla$). But both lead to a contradiction:

- If $y_1 \xrightarrow{\mathsf{vis}} x_1$, then also $y_1 \xrightarrow{\mathsf{vis}} x_2$ because vis is transitive. Thus, $\mathsf{context}(A_{xy}, x_2) = (\{x_1, y_1\}, \dots)$ and therefore (by Definition 9.3) $\mathcal{F}_\tau(\mathsf{context}(A_{xy}, x_2)) \neq v_2 = \mathsf{rval}(x_2)$, contradicting RVal$(\mathcal{F}_\tau)$.
- If $x_1 \xrightarrow{\mathsf{vis}} y_1$, then also $x_1 \xrightarrow{\mathsf{vis}} y_2$ because vis is transitive. Thus, $\mathsf{context}(A_{xy}, y_2) = (\{x_1, y_1\}, \dots)$ and therefore (by Definition 9.3) $\mathcal{F}_\tau(\mathsf{context}(A_{xy}, y_2)) \neq v_2' = \mathsf{rval}(y_2)$, contradicting RVal$(\mathcal{F}_\tau)$.

□

9.2 Progress

Our definition of protocols and executions is meant to model systems including roles and a network. Thus, it is intuitively clear that we should always be able to take a finite execution and continue taking steps forever, to obtain an infinite execution. However, in our formalization, it is not immediately clear that such a completion exists for arbitrary protocols.

For example, if we removed condition (r3) on p. 96, we could construct some protocol Π_{pathetic} that contains a role that refuses to receive a message (*i.e.* whose automaton does not have a transition to receive it), which means the *dontlose* guarantee may be unsatisfiable in all executions of Π. This would have odd consequences: it implies that $\mathcal{E}_{\text{complete}}(\Pi_{\text{pathetic}}) = \emptyset$, which in turn implies that Π_{pathetic} vacuously satisfies *any* consistency guarantee (Def. 9.1). Clearly, such nonsense must be ruled out.

We now prove that our definition of a protocol is crafted carefully enough. First, consider the notion of completing a history by filling in some absent return values:

Definition 9.4. A history $H = (E, \mathsf{op}, \mathsf{rval}, \mathsf{rb}, \mathsf{ss})$ is a *completion* of a

history H' if the latter is of the form $H' = (E, \mathsf{op}, \mathsf{rval}', \mathsf{rb}, \mathsf{ss})$ where for all $e \in E$, either $\mathsf{rval}(e) = \mathsf{rval}'(e)$ or $\mathsf{rval}'(e) = \nabla$.

The following theorem proves that any finite execution can be completed, without requiring external input (such as additional operation calls). We need it for the proof of Theorem 9.1.

Theorem 9.3 (Progress)**.** Let Π be a protocol, and let G_0 be a finite execution of Π. Then, there exists a complete execution G of Π such that $G_0 \sqsubseteq G$ and $\mathcal{H}(G)$ is a completion of $\mathcal{H}(G_0)$.

Proof. Let $\Pi = (R, M, O, \Sigma, P, S_1 \wedge \cdots \wedge S_n, T)$. The idea is to construct an infinite sequence of executions $G_0 \sqsubseteq G_1 \sqsubseteq G_2 \ldots$ and then show that $G \stackrel{\text{def}}{=} \lim_i G_i$ (as defined on page 93) is an execution of Π. For this to work, we must choose valid transitions from the role automata specified by T, we must be fair to all processes (required by condition (e3) on p. 100), and we must respect the transport guarantees $S_1 \wedge \cdots \wedge S_n$. We can achieve this by always scheduling/delivering a process/message that has been waiting the longest. We now describe this proof idea in more detail.

For a finite execution $G = (E, \mathsf{eo}, \mathsf{tr}, \mathsf{role}, \mathsf{del})$, with eo enumerating E in the order $E = \{e_0, \ldots, e_n\}$, we define $\mathsf{rank}(e_i) = i$, and for each $r \in \mathsf{roles}(G)$, we let $i_r \in E$ be the initialization transition of $\mathsf{traj}(G, r)$. Now we define the set of scheduling obligations $Q_G \subseteq (E \times M \times R) \cup (E \times R \times P)$ and the function $\mathsf{birthday}_G : Q_G \to \mathbb{N}_0$ as follows:

- Q_G contains all tuples (e, m, r) such that $m \in \mathsf{snd}(e) \wedge r \in \mathsf{roles}(G) \wedge \mathsf{role}(e) \neq r \wedge \neg(\exists e' \in E(r) : e \xrightarrow{\mathsf{del}} e' \wedge \mathsf{rcv}(e') = m)$. Each tuple represents an obligation to deliver the message m sent by e to role r. We define $\mathsf{birthday}_G(e, m, r) = \max(\mathsf{rank}(e), \mathsf{rank}(i_r))$.
- Q_G contains all tuples (e, r, p) such that $r \in \mathsf{roles}(G) \wedge p \in P_r \wedge e = \max_{\mathsf{eo}}\{e' \mid \mathsf{role}(e') = r \wedge (\mathsf{pre}(e') = \bot \vee \mathsf{proc}(e') = p)\}$. We define $\mathsf{birthday}_G(e, r, p) = \mathsf{rank}(e)$.

The following lemma clarifies that that the birthday of obligations does not change if we extend an execution, and can thus be used to find out how old an obligation is:

Lemma 9.4. If $G \sqsubseteq G'$, and if $q \in Q_G \cap Q_{G'}$, then $\mathsf{birthday}_G(q) = \mathsf{birthday}_{G'}(q)$.

Proof. If $q = (e, m, r) \in Q_G \cap Q_{G'}$, then both G and G' contain the events e and i_r, which have the same rank in both, so the birthday is the same. Similarly, if $q = (e, r, p)$, then both contain e, thus the rank is the same also. □

To construct the sequence $G_0 \sqsubseteq G_1 \ldots$, in each step, we pick a scheduling obligation of minimal birthday, and choose a transition to service that obligation. As we do so, we may add finitely many new obligations of a higher birthday.

Lemma 9.5. Given a finite execution $G_n = (\{e_0, \ldots, e_n\}, \ldots) \in \mathcal{E}(\Pi)$ and a scheduling obligation $q \in Q_{G_n}$, there exists an execution $G_{n+1} = (\{e_0, \ldots, e_{n+1}\}, \ldots) \in \mathcal{E}(\Pi)$ such that $G_n \sqsubseteq G_{n+1}$, and $\mathcal{H}(G_{n+1})$ is a completion of $\mathcal{H}(G_n)$, and $Q_{G_{n+1}} = ((Q_{G_n} \setminus \{q\}) \cup X)$ for some finite set X such that $\mathsf{birthday}(x) = n + 1$ for all $x \in X$.

Proof. If $q = (e, m, r)$, then by condition (r3) on p. 96, the automaton T_r must contain a transition t such that $\mathsf{rcv}(t) = m$, and whose pre-state $\mathsf{pre}(t)$ matches the post-state of the last transition of $\mathsf{traj}(G_n, r)$. Thus, we can add an event e_{n+1} with this transition to the execution. Then $q \notin Q_{G_{n+1}}$ (because the message has now been delivered) and X contains all tuples (e_{n+1}, m', r') where m' is a message in $\mathsf{snd}(t)$ and r' is a role in $\mathsf{roles}(G_n)$, and is thus finite.

If $q = (e, r, p)$, then by condition (r5) on p. 96, the automaton T_r must contain a step transition t for process p whose pre-state matches the post-state of the last transition of $\mathsf{traj}(G_n, r)$. Thus, we can add an event e_{n+1} with this transition to the execution. Then $q \notin Q_{G_{n+1}}$ (because e is no longer the latest step/init event) and X contains the finite set of tuples (e_{n+1}, m', r') where m' is a message in $\mathsf{snd}(t)$ and r' is a role in $\mathsf{roles}(G_n)$, plus the tuple (e_{n+1}, r, p), and is thus finite. □

Since $Q_{G_0} = \emptyset$, and we add only finitely many obligations in each step, Q_{G_i} is finite for all i. This means that for all obligations, there exist only finitely many obligations that have an equal or older birthday, which means that all obligations get serviced after finitely many steps.

Then, $G \stackrel{\text{def}}{=} \lim_i G_i$ is a complete execution of Π because:

- G is a concrete execution by Lemma 7.4.
- It satisfies condition (e1) on p. 100 (which is a safety property) as well as all the safety guarantees among the S_i, because any safety property satisfied by all G_i carries over to the limit (Definition 7.12).
- G satisfies fairness (condition (e3)) because for each $r \in \mathsf{roles}(G)$ and process $p \in P_r$, there is always some obligation (e, r, p), thus p gets scheduled again after finitely many steps.
- G satisfies the liveness conditions among the S_i, because for every event e sending a message m and each role $r \in \mathsf{roles}(G)$, there is a scheduling obligation (e, m, r) that gets eventually serviced, thus the message is delivered.

And it is easy to see that $G_0 \sqsubseteq G$ and $\mathcal{H}(G)$ is a completion of $\mathcal{H}(G_0)$. □

10

Correctness

The best way to develop a good understanding about how the various building blocks used in protocols (such as timestamps, logical clocks, vector clocks, and so on) ensure its proper function is to prove correctness of a variety of protocols that use different design principles. In this chapter, we study five of the protocol examples from Chapter 6 and prove that they satisfy the consistency guarantees claimed in Fig. 6.2.

10.1 Proof Structure

All of our proofs follow the same general structure. Our goal is to prove an obligation of the form

$$\Pi \models \text{RVAL}(\mathcal{F}) \wedge \mathcal{P}_1 \wedge \cdots \wedge \mathcal{P}_n,$$

which states that the protocol Π correctly implements the replicated data type $\mathcal{F}$ and provides the ordering and liveness guarantees $\mathcal{P}_i$. To prove this goal, we proceed as follows.

1. *The execution.* We start with a single, arbitrary, complete execution of the protocol $G = (E, \mathsf{eo}, \mathsf{tr}, \mathsf{role}, \mathsf{del}) \in \mathcal{E}_{\text{complete}}(\Pi)$.

2. *Auxiliary notations.* We introduce auxiliary notations as needed, to help us reason about the execution G. For example, we often define functions that correlate send and receive events in G.

3. *State Invariant.* We write down a state invariant $I(e)$, which for each event $e \in E$ expresses the state $\mathsf{post}(e)$ as a function of the event graph G. We then prove $\forall e \in E : I(e)$ by induction over the execution order eo.

4. *Well-formed.* We prove that G is well-formed (Definition 7.8). This step is often trivial since for most of our protocols, all call operations return immediately.

5. *Witness.* We define arbitration and visibility relations $\mathsf{ar}_A, \mathsf{vis}_A$ on the set of operation call events $E_A \stackrel{\text{def}}{=} \mathsf{ops}(G)$, and show that ar_A is a total order, and that vis_A is an acyclic natural relation.

6. *Abstract Execution.* By extending the observable history $\mathcal{H}(G) = (E_A, \mathsf{op}_A, \mathsf{rval}_A, \mathsf{rb}_A, \mathsf{ss}_A)$ (Definition 7.9) with the witness visibility and arbitration relations, we obtain an event graph $A = (E_A, \mathsf{op}_A, \mathsf{rval}_A, \mathsf{rb}_A, \mathsf{ss}_A, \mathsf{vis}_A, \mathsf{ar}_A)$ that meets the requirements for an abstract execution (Definition 3.3).

7. *Return Values.* We prove that $A \models \textsc{RVal}(\mathcal{F})$. This step typically relies on the state invariant.

8. *Ordering and Liveness Guarantees.* We prove for all $\mathcal{P}_i$ that $A \models \mathcal{P}_i$, by reasoning about ar_A and vis_A.

Lemma 10.1. The steps above are sufficient to prove the goal.

Proof. See § A.1.2 in the appendix. □

One advantage of this proof approach is that all reasoning takes place on a single, fixed infinite execution. We can state and prove arbitrary properties of the execution, possibly involving both future and past events, without adding history or prophecy variables to the implementation [Abadi and Lamport, 1991], and without needing backward simulations [Lynch and Vaandrager, 1995, Colvin et al., 2006].

Accessing future events is sometimes necessary to prove consistency: for example, in all of our global sequence protocol implementations, the arbitration order of two events is decided at a later point of time than when the events are originally performed.

Organization. We organize the correctness proofs in this chapter based on the protocol category, because protocols in the same category tend to have similar proofs. We start with two epidemic protocols (counter and register), continue with a broadcast protocol (counter) and conclude with two global sequence protocols (single-copy register and buffered sequencer).

10.2 Epidemic Protocols

For an execution $G = (E, \mathsf{eo}, \mathsf{tr}, \mathsf{role}, \mathsf{del})$, we define the visibility cone $V(e)$ of an event e to be the set of operation call events in $E_A = \mathsf{ops}(G)$ from which we can reach e along a path consisting of ro and del edges:

$$V(e) \quad \stackrel{\text{def}}{=} \quad \{x \in E_A \mid \quad x \xrightarrow{\mathsf{ro} \cup \mathsf{del}}{}^{*} e\}.$$

For most epidemic protocols, the set $V(e)$ captures exactly the operations that are visible to e, since the "cumulative summary" is stored in the state of each role (thus flows along ro edges) and transmitted in messages (thus flows along del edges). The following lemma is useful when doing induction proofs involving $V(e)$.

Lemma 10.2 (Visibility Cone Update)**.** Let $G = (E, \mathsf{eo}, \mathsf{tr}, \mathsf{role}, \mathsf{del})$ be a concrete execution that satisfies *dontforge*. Let $e \in E$ be an event. Then

$$V(e) = \begin{cases} \emptyset & \text{if } \mathsf{pre}(e) = \bot \\ V(p) \cup \{e\} & \text{if } \mathsf{op}(e) \neq \bot \\ V(p) \cup V(s) & \text{if } \mathsf{rcv}(e) \neq \bot \\ V(p) & \text{otherwise} \end{cases}$$

where $p = \mathsf{pred}(E, \mathsf{ro}, e)$ the predecessor event by the same role (exists if $\mathsf{pre}(e) \neq \bot$ by condition (t3) on p. 86), and where $s = \mathsf{del}^{-1}(e)$ the sender of the message received by e (exists if $\mathsf{rcv}(e) \neq \bot$ by *dontforge*, and is unique by injectivity of del).

Proof. See § A.1.3 in the appendix. □

10.2.1 Epidemic Counter

Theorem 10.3. $\Pi_{\mathsf{EpidemicCounter}} \models \textsc{CausalConsistency}(\mathcal{F}_{ctr})$.

We follow the proof structure described in §10.1 to prove $\Pi_{\mathsf{EpidemicCounter}} \models \textsc{RVal}(\mathcal{F}_{ctr}) \wedge \textsc{Causality} \wedge \textsc{EventualVisibility}$.

Auxiliary Definitions. We define the set of increment operations by a specific replica r that are visible to an event e as

$$\mathsf{visinc}(e, r) \quad \stackrel{\text{def}}{=} \quad \{x \in V(e) \mid (\mathsf{role}(x) = r) \wedge (\mathsf{op}(x) = \mathsf{inc})\}$$

State Invariant. We now define the state invariant that expresses the post-state of an event as a function of the concrete execution G. For this implementation, the state is a vector that counts the increment operations per replica.

Lemma 10.4. The following invariant $I(e)$ holds for all events $e \in E$:

$$\mathsf{post}(e) = \mathit{SPeer}(\ \lambda r.\ |\mathsf{visinc}(e, r)|\ )$$

Proof. To prove $\forall e \in E : I(e)$ we use induction over eo, and do a case distinction over $\mathsf{tr}(e)$. Looking at each transition (listed in Fig. 6.6) separately, we get:

$\left[\ \mathsf{tr}(e) = \mathit{init}(\mathit{SPeer}(\lambda r.0), \emptyset).\ \right]$ By Lemma 10.2, $V(e) = \emptyset$, thus $\mathsf{visinc}(e, r) = \emptyset$ for all r, thus $\mathit{SPeer}(\lambda r.0) = \mathsf{post}(e)$.

$\left[\ \mathsf{tr}(e) = \mathit{callret}(\mathsf{rd}, \sigma, \sigma, \emptyset, v) \text{ or } \mathsf{tr}(e) = \mathit{step}(bg, \sigma, \sigma, \{m\}).\ \right]$ These are the "boring" cases where the invariant is preserved because neither the state nor the number of visible increments change. Let $p = \mathsf{pred}(E, \mathsf{ro}, e)$. Then $I(e)$ holds because:

$$\mathsf{post}(e) \stackrel{1}{=} \mathsf{post}(p) \stackrel{2}{=} \mathit{SPeer}(\lambda r.\ |\mathsf{visinc}(p, r)|\ ) \stackrel{3}{=} \mathit{SPeer}(\lambda r.\ |\mathsf{visinc}(e, r)|\ )$$

1. The event e does not modify the state ($\mathsf{pre}(e) = \mathsf{post}(e)$) and the prestate of e must match the poststate of e' (condition (c4) on p. 87).
2. By induction $I(p)$ is true (because $p \xrightarrow{\mathsf{vis}} e$, thus $p \xrightarrow{\mathsf{eo}} e$).

3. By Lemma 10.2, either $V(e) = V(p)$ or $V(e) = V(p) \cup \{e\}$. In either case $\mathsf{visinc}(e, r) = \mathsf{visinc}(p, r)$ for all r because e is not an increment operation.

$\left[\ \mathsf{tr}(e) = \mathit{callret}(\mathsf{inc}, \epsilon, \mathit{SPeer}(c), \mathit{SPeer}(c[r \mapsto c[r] + 1]), \emptyset, \mathit{ok}).\ \right]$ This is the case where a role increments the local count and leaves the other counts unchanged. Let $p = \mathsf{pred}(E, \mathsf{ro}, e)$. By Lemma 10.2, $V(e) = V(p) \cup \{e\}$. Thus, $\mathsf{visinc}(e, r) = \mathsf{visinc}(p, r)$ for $r \neq \mathsf{role}(e)$ and $\mathsf{visinc}(e, r) = \mathsf{visinc}(p, r) + 1$ for $r = \mathsf{role}(e)$. This implies $I(e)$.

$\left[\ \mathsf{tr}(e) = \mathit{rcv}(\mathit{Latest}(c'), \mathit{SPeer}(c), \mathit{SPeer}(\lambda r.\, \max(c(r), c'(r))), \emptyset).\ \right]$ This is the case where something interesting happens. So far, everything we have proved would work just as well for a naive implementation where we use a single counter for counting all increments, instead of a separate counter for each replica. Now, however we see why we need to count per replica: so that the maximum of two counts matches the count of the union — as expressed in Lemma 10.5 below. Let $p = \mathsf{pred}(E, \mathsf{ro}, e)$ and $s = \mathsf{del}^{-1}(e)$. Let $r \in R$ be arbitrary. Then $c(r) = \mathsf{visinc}(p, r)$ (because $I(p)$ by induction), and $c'(r) = \mathsf{visinc}(s, r)$ (because $I(s)$ by induction). By Lemma 10.2, $V(e) = V(p) \cup V(s)$, and thus $\mathsf{visinc}(e, r) = \mathsf{visinc}(p, r) \cup \mathsf{visinc}(s, r)$. By Lemma 10.5 below (using that ro is a total order on $E_A(r)$), this implies $|\mathsf{visinc}(e, r)| = \max\{|\mathsf{visinc}(p, r)|, |\mathsf{visinc}(s, r)|\} = \max\{c(r), c'(r)\}$ (by induction) which implies $I(e)$. □

Lemma 10.5. Let C be some set that is totally ordered by rel and let $A, B \subseteq C$ be two finite subsets that are predecessor-closed under rel (*i.e.* $\mathsf{rel}^{-1}(A) \subseteq A$ and $\mathsf{rel}^{-1}(B) \subseteq B$). Then $|A \cup B| = \max\{|A|, |B|\}$.

Proof. See § A.1.4 in the appendix. □

Well-formed. G is trivially well formed because all operations return immediately (Lemma 7.1).

Witness. As explained above, in epidemic protocols an operation e is visible to an operation e' if we can reach e' from e along a path consisting of role-order and delivery edges. As for the arbitration, it

plays no role for this data type so any total order extending visibility will do: we can use the totalization function defined in proposition 2.1 on page 23 to obtain ar_A. We define visibility and arbitration as follows:

$$\mathsf{vis}_A \stackrel{\text{def}}{=} (\mathsf{ro} \cup \mathsf{del})^+|_{E_A} \qquad \mathsf{ar}_A \stackrel{\text{def}}{=} \mathsf{totalize}(\mathsf{vis}_A, \mathsf{eo}|_{E_A})$$

Our witness satisfies the requirements:

- vis_A is an acyclic, natural relation on E_A: because $\mathsf{del} \subseteq \mathsf{eo}$ (condition (c5) on p. 87), $\mathsf{ro} \subseteq \mathsf{eo}$ (by Definition 7.6, pg. 87), and eo is an enumeration (condition (c1) on p. 87) and thus transitive, acyclic, and natural. Thus $(\mathsf{ro} \cup \mathsf{del})^+ \subseteq \mathsf{eo}$. This implies that vis_A cannot have cycles and must be natural (by Lemma 2.3).
- ar_A is a total order on E_A, by proposition 2.1.

Return Values. To prove $A \models \textsc{RVal}(\mathcal{F}_{ctr})$, we need to show that $\forall e \in E : \mathsf{rval}_A(e) = \mathcal{F}_{ctr}(\mathsf{op}(e), \mathsf{context}(A, e))$. Since all operations return immediately, $\mathsf{rval}_A(e) = \mathsf{rval}(e)$. If $\mathsf{op}(e) = \mathsf{inc}$, then $\mathsf{tr}(e) = \mathit{callret}(\mathsf{inc}, \sigma, \sigma', \emptyset, \mathit{ok})$, thus $\mathsf{rval}(e) = \mathit{ok} = \mathcal{F}_{ctr}(\mathsf{inc}, \dots)$. If $\mathsf{op}(e) = \mathsf{rd}$, then $\mathsf{tr}(e) = \mathit{callret}(\mathsf{rd}, \mathit{SPeer}(c), \mathit{SPeer}(c), \emptyset, v)$ for some c and for $v = \sum_r c(r)$. Using the invariant (Lemma 10.4), we get

$$\begin{aligned}
\mathsf{rval}(e) = \sum_r c(r) &\stackrel{10.4}{=} \sum_r \left|\{e' \in E(r) \mid e' \xrightarrow{\mathsf{ro} \cup \mathsf{del}}{}^* e \text{ and } \mathsf{op}(e') = \mathsf{inc}\}\right| \\
&= \sum_r \left|\{e' \in E(r) \mid e' \xrightarrow{\mathsf{vis}_A} e \text{ and } \mathsf{op}(e') = \mathsf{inc}\}\right| \\
&= \left| \bigcup_r \{e' \in E(r) \mid e' \xrightarrow{\mathsf{vis}_A} e \text{ and } \mathsf{op}(e') = \mathsf{inc}\} \right| \\
&= \left|\{e' \in E \mid e' \xrightarrow{\mathsf{vis}_A} e \text{ and } \mathsf{op}(e') = \mathsf{inc}\}\right| \\
&= \mathcal{F}_{ctr}(\mathsf{op}(e), \mathsf{context}(A, e)).
\end{aligned}$$

Ordering Guarantees. To prove $A \models \textsc{Causality}$, we need to show that the relation $\mathsf{hb}_A = (\mathsf{so}_A \cup \mathsf{vis}_A)^+$ is contained in both vis_A and ar_A. This is easy: all events by the same role are in the same session (condition (x5) on p. 90), thus $\mathsf{so}_A = \mathsf{ro}|_{E_A}$; therefore,

$$\mathsf{hb}_A = (\mathsf{ro}|_{E_A} \cup (\mathsf{ro} \cup \mathsf{del})^+|_{E_A})^+ \subseteq (\mathsf{ro} \cup \mathsf{del})^+|_{E_A} = \mathsf{vis} \subseteq \mathsf{ar}.$$

Liveness Guarantee. To prove $A \models \textsc{EventualVisibility}$, let $e \in E$ and $[f] \in E/{\approx_{ss}}$ as in Definition 5.1 (page 52). Let $s \stackrel{\text{def}}{=} \mathsf{role}(e)$ and $r \stackrel{\text{def}}{=} \mathsf{role}(f)$. Then $s, r \in \mathsf{correct}(G)$ since $G \in \mathcal{E}_{\text{complete}}$ implies $\mathsf{crashed}(G) = \emptyset$. In particular, $E(s)$ is infinite. Since $bg \in P_s$ and *snd* is the only transition for process *bg* (Fig. 6.6) we know that s performs the *snd* transition infinitely many times by fairness (condition (e3) on p. 100). Since the protocol specifies the *eventualindirect*(M) guarantee (defined on page 97), this implies that $\forall e \in E(s) : \exists e' \in E(r) : (e \xrightarrow{(\mathsf{ro} \cup \mathsf{del}(\mathsf{M}))^*} e')$. Thus there exists an $e' \in E(r)$ such that $e \xrightarrow{\mathsf{ro} \cup \mathsf{del}} + e'$. Since $E(r)$ is totally ordered by ro, this implies that for any $e'' \in E(r)$, $e \overset{\mathsf{vis}}{\not\longrightarrow} e'' \Rightarrow e'' \xrightarrow{\mathsf{ro}} e'$. Thus there can be at most finitely many such e'', because $\mathsf{ro}^{-1}(e')$ is finite.

10.2.2 Epidemic Register

We now prove correctness of the epidemic register (Fig. 1.2). This is an unusual example of an epidemic protocol because it is sequentially consistent, not just causally consistent. For that reason, we do not use the cone $V(e)$ to define visibility, but instead use a total timestamp order.

Theorem 10.6. $\Pi_{\mathsf{EpidemicRegister}} \models \textsc{SequentialConsistency}(\mathcal{F}_{reg})$.

We follow the proof structure described in §10.1 to prove $\Pi_{\mathsf{EpidemicRegister}} \models \textsc{RVal}(\mathcal{F}_{reg}) \wedge \textsc{SingleOrder} \wedge \textsc{ReadMyWrites}$.

Auxiliary Definitions. Timestamps are the key for ordering the events in the execution. We let $\mathsf{TS} \stackrel{\text{def}}{=} \mathbb{N}_0 \times \mathbb{N}_0$ be the set of timestamps, ordered by the lexicographic order (as defined by the pseudocode Fig. 1.2) which is a total order. For any event $e \in E$, we define the corresponding timestamp $\mathsf{ts}(e) \stackrel{\text{def}}{=} \mathsf{post}(e).\mathsf{written}$. We define the timestamp order on E as $e \xrightarrow{\mathsf{ts}} e' \stackrel{\text{def}}{\iff} \mathsf{ts}(e) < \mathsf{ts}(e')$. It is a partial order (because it is irreflexive and transitive), but not a total order (read events do not advance the logical clock, thus multiple events can have the same timestamp). We define the set of write operations $W \stackrel{\text{def}}{=} \{e \in E_A \mid \mathsf{op}(e) = \mathsf{wr}(v) \text{ for some } v \in \mathsf{Values}\}$.

Lemma 10.7. The following conditions are true:

1. Timestamps for each role are monotonic:
   $\forall e, e' \in E : \quad e \xrightarrow{\mathsf{ro}} e' \;\Rightarrow\; \mathsf{ts}(e) \leq \mathsf{ts}(e')$.
2. $\mathsf{ts} \cup \mathsf{ro}$ is acyclic.
3. Timestamps for writes by the same role are strictly ordered:
   $\forall w, w' \in W : \quad w \xrightarrow{\mathsf{ro}} w' \;\Rightarrow\; \mathsf{ts}(w) < \mathsf{ts}(w')$.
4. ts is injective for writes:
   $\forall w, w' \in W : \quad w \neq w' \;\Rightarrow \mathsf{ts}(w) \neq \mathsf{ts}(w')$.

Proof. The first claim is proved easily by an induction over the transitions. The second claim follows from the first. The third claim follows from the first and the fact that we increase written.number when creating a new timestamp. The fourth claim follows because the second claim ensures uniqueness per replica, and uniqueness across replicas is also guaranteed because the originating replica is included in the timestamp. □

State Invariant. What we prove about the state is that current,written are either meaningless (if written.number $= 0$) or describe some write that actually took place earlier in the execution.

Lemma 10.8. The following invariant $I(e)$ holds for all events $e \in E$:

$$\begin{aligned}\big[\ (\mathsf{post}(e).\mathsf{written.number} = 0) \wedge (\mathsf{post}(e).\mathsf{current} = \mathit{undef})\ \big] \quad \vee \\ \big[\exists w \in W :\ \mathsf{post}(e).\mathsf{written} = \mathsf{ts}(w) \wedge \\ (w \leq_{\mathsf{eo}} e) \ \wedge\ \mathsf{op}(w) = \mathsf{wr}(\mathsf{post}(e).\mathsf{current})\big]\end{aligned}$$

Proof. Induction over eo and case distinction on the transition. For the initial transition, the invariant is established because its first clause is true. For read, send, and idle transitions, and for write transitions where the incoming message does not have a larger timestamp than current, the local state is not modfied. By induction, the invariant is true for the predecessor $e' = \mathsf{pred}(E, \mathsf{ro}, e)$ and thus carries over to e. Write transitions where the incoming timestamp is larger establish the second clause because the invariant is true for the sender $s = \mathsf{del}^{-1}(e)$

by induction, from where we copy the fields $\mathsf{current}, \mathsf{written}$, and who satisfies $s \leq_{\mathsf{eo}} e$. □

Well-formed. G is trivially well formed because all operations return immediately (Lemma 7.1).

Witness. We define visibility and arbitration by using the timestamp partial order ts and breaking ties using the execution order eo:

$$\mathsf{vis}_A \stackrel{\text{def}}{=} \mathsf{ar}_A \stackrel{\text{def}}{=} \{(e, e') \mid \mathsf{ts}(e) < \mathsf{ts}(e') \ \vee \ (\mathsf{ts}(e) = \mathsf{ts}(e') \ \wedge \ e \xrightarrow{\mathsf{eo}} e')\}$$

Note that this is indeed a total order because it is transitive, and irreflexive (for any cycle or self-loop $e_1 \xrightarrow{\mathsf{ar}} e_2 \ldots \xrightarrow{\mathsf{ar}} e_n \xrightarrow{\mathsf{ar}} e_2$, $n \geq 1$, all events would have to have the same timestamp, implying a cycle or self-loop in eo which is a contradiction) and orders any two events (because timestamps are totally ordered, and eo is a total order). It is also natural, because the number of clients is finite. In fact, this is the reason we bounded the number of clients in this protocol (line 10 of Fig. 1.2 specifies that there are at most $N + 1$ clients in an execution).

Return Values. Since all operations return immediately, $\mathsf{rval}_A(e) = \mathsf{rval}(e)$. Writes obviously return the correct value *ok*. For reads, consider e with $\mathsf{op}(e) = \mathsf{rd}$ and with visible write set $W(e) \stackrel{\text{def}}{=} \{w \in W \mid w \xrightarrow{\mathsf{vis}} e\}$, for which we need to show

$$\mathsf{rval}(e) = \begin{cases} \mathit{undef} & \text{if } W(e) = \emptyset \\ v & \text{if } \left(\max_{\mathsf{ar}} W(e)\right) = \mathsf{wr}(v) \end{cases} \tag{10.1}$$

To see why this is true, consider both cases of the disjunction in the invariant.

- Assume $(\mathsf{post}(e).\mathsf{written.number} = 0) \wedge (\mathsf{post}(e).\mathsf{current} = \mathit{undef})$. Then $\mathsf{rval}(e) = \mathit{undef}$. Since the first component of $\mathsf{ts}(e)$ is zero, we know that $e' \xrightarrow{\mathsf{vis}} e$ implies that the first component of $\mathsf{ts}(e')$ is also zero, which means e' cannot be a write; thus $W(e) = \emptyset$, and (10.1) is true.

- Assume $\exists w \in W : \mathsf{post}(e).\mathsf{written} = \mathsf{ts}(w) \wedge (w \leq_{\mathsf{eo}} e) \wedge \mathsf{op}(w) = \mathsf{wr}(\mathsf{post}(e).\mathsf{current})$. Then $w \in W(e)$ and $\mathsf{op}(w) = \mathsf{wr}(\mathsf{rval}(e))$. So,

all that remains to show is that w is maximal in $W(e)$. But this follows because: we know that $\mathsf{ts}(w) = \mathsf{ts}(e)$, and this implies that $\mathsf{ts}(w') \leq \mathsf{ts}(w)$ for any $w' \in W(e)$. But writes are totally ordered by claim 4 of the lemma above, thus $\mathsf{ts}(w') < \mathsf{ts}(w)$, which means that w is indeed maximal in $W(e)$.

Ordering Guarantees. It is easy to see that $A \models$ SingleOrder: $\mathsf{ar}_A \cap (E_c \times E) \subseteq \mathsf{vis}_A$ is obvious since $\mathsf{ar}_A = \mathsf{vis}_A$. To prove $A \models$ ReadMyWrites, we need to show $\mathsf{so} \subseteq \mathsf{vis}_A$ which follows from the monotonicity of the timestamps in a role (claim one of the lemma above) and the fact that $\mathsf{so} \subseteq \mathsf{eo}$.

10.3 Broadcast Protocols

For the protocols based on reliable broadcast introduced earlier, each message contains information about one operation. Visibility is determined by the existence of a path of the form $(\mathsf{del}^? ; \mathsf{ro}^*)$, *i.e.* zero or one delivery edge, followed by zero or more role-order edges. For an event e, we define the delivery set $D(e)$ of operations that have been delivered to the replica performing e:

$$D(e) \quad \stackrel{\text{def}}{=} \quad \{x \in E_A \mid \quad x \xrightarrow{\mathsf{del}^? ; \mathsf{ro}^*} e\}$$

The use of the delivery set $D(e)$ for broadcast protocols is somewhat analogous to the use of the visibility cone $V(e)$ for epidemic protocols. The following lemma is useful when doing inductions involving $D(e)$.

Lemma 10.9 (Delivery Set Update)**.** Let $G = (E, \mathsf{eo}, \mathsf{tr}, \mathsf{role}, \mathsf{del})$ be a concrete execution that satisfies *dontforge*. For all $e \in E$:

$$D(e) = \begin{cases} \emptyset & \text{if } \mathsf{pre}(e) = \bot \\ D(p) \cup \{e\} & \text{if } \mathsf{op}(e) \neq \bot \\ D(p) \cup \{s\} & \text{if } \mathsf{rcv}(e) \neq \bot \\ D(p) & \text{otherwise} \end{cases}$$

where $p = \mathsf{pred}(E, \mathsf{ro}, e)$ the predecessor event by the same role, and where $s = \mathsf{del}^{-1}(e)$ the sender of the message received by e.

The proof is analogous to Lemma 10.2.

10.3.1 Broadcast Counter

We now prove correctness of the broadcast counter (see pseoudocode in Fig. 6.3 and formal definition in Fig. 6.4).

Theorem 10.10.

$$\Pi_{\mathsf{BroadcastCounter}} \models \textsc{BasicEventualConsistency}(\mathcal{F}_{ctr}) \\ \wedge \textsc{ReadMyWrites} \wedge \textsc{CausalArbitration}$$

We follow the proof structure described in §10.1. The proof is similar to the proof for the state-based counter (§10.2.1), but a little easier.

State Invariant. The state invariant says that the current count is the sum of all increments in the delivery set.

Lemma 10.11. The following invariant $I(e)$ holds for all events $e \in E$:

$$\mathsf{post}(e) = \mathit{SPeer}(|\{x \in D(e) \mid \mathsf{op}(x) = \mathsf{inc}\}|)$$

The proof of the lemma has the same structure as for the state-based counter; we use induction and case distinction on the transition. All cases are easy, thus we do not reproduce them in detail.

Well-formed. G is trivially well formed because all operations return immediately (Lemma 7.1).

Witness. We define visibility and arbitration as follows:

$$\mathsf{vis}_A \stackrel{\text{def}}{=} ((\mathsf{ro} \cup \mathsf{del})\,;\mathsf{ro}^*)|_{E_A} \quad \mathsf{ar}_A \stackrel{\text{def}}{=} \mathsf{totalize}((\mathsf{ro} \cup \mathsf{del})^*|_{E_A}, \mathsf{eo}|_{E_A})$$

Visibility is natural and acyclic because $\mathsf{ro} \subseteq \mathsf{eo}$ and $\mathsf{del} \subseteq \mathsf{eo}$ and thus $\mathsf{vis}_A \subseteq \mathsf{eo}$, and eo is natural and acyclic. As in the state-based counter, the arbitration is irrelevant for the values returned because $\mathcal{F}_{ctr}$ does not depend on it, so we choose a totalization of the happens-before order (to get causal arbitration).

Return Values. To prove $A \models \textsc{Rval}(\mathcal{F}_{ctr})$, we need to show $\mathsf{rval}_A(e) = \mathcal{F}_{ctr}(\mathsf{op}(e), \mathsf{context}(A, e))$ for all $e \in \mathsf{ops}(E)$. Since all operations return immediately, $\mathsf{rval}_A(e) = \mathsf{rval}(e)$. For $\mathsf{op}(e) = \mathsf{inc}$, $\mathsf{rval}(e) = \mathit{ok}$ as desired. For $\mathsf{op}(e) = \mathsf{rd}$, we know $\mathsf{tr}(e) = \mathit{callret}(\mathsf{rd}, \mathit{SPeer}(c), \mathit{SPeer}(c), \emptyset, c)$ for

some number c. Using the invariant (Lemma 10.11), we get $\mathsf{rval}(e) = \big|\{x \in D(E) \mid \mathsf{op}(x) = \mathsf{inc}\}\big| = \big|\{x \in \mathsf{ops}(E) \mid x \xrightarrow{\mathsf{vis}} e \text{ and } \mathsf{op}(x) = \mathsf{inc}\}\big| = \mathcal{F}_{ctr}(\mathsf{op}(e), \mathsf{context}(A, e))$.

Ordering Guarantees. To prove $A \models$ ReadMyWrites, we need to show $\mathsf{so} \subseteq \mathsf{vis}$ which is immediate since $\mathsf{so} = \mathsf{ro}|_{\mathsf{ops}(E)}$. To prove $A \models$ CausalArbitration $\wedge$ NoCircularCausality, we need to show that the relation $\mathsf{hb} = (\mathsf{so} \cup \mathsf{vis}_A)^+$ is contained in ar_A and is acyclic. The former follows directly from how we defined ar_A. Acyclicity is also easy because $\mathsf{del} \subseteq \mathsf{eo}$ and $\mathsf{ro} \subseteq \mathsf{eo}$, and eo is acyclic.

Liveness Guarantee. To prove $A \models$ EventualVisibility, let $e \in E$ and $[f] \in E/\approx_{ss}$ as in Definition 5.1 (page 52). Let $s \stackrel{\text{def}}{=} \mathsf{role}(e)$ and $r \stackrel{\text{def}}{=} \mathsf{role}(f)$. Then $s, r \in \mathsf{correct}(G)$ since $G \in \mathcal{E}_{\text{complete}}$. Since the protocol requires the *reliable*(M) guarantee and thus the *dontlose*(M) guarantee (defined on page 97), this implies that there exists an $e' \in E(r)$ such that $e \xrightarrow{\mathsf{del}} e'$. This implies that for any $e'' \in E_A(r)$ such that $e' \xrightarrow{\mathsf{ro}} e''$, we have $e \xrightarrow{\mathsf{vis}} e''$. Thus there can be at most finitely many $e'' \in E_A(r)$ such that $e \not\xrightarrow{\mathsf{ro}} e''$.

10.4 Global-Sequence Protocols

In global sequence protocols, the arbitration order is not immediately determined, but requires some communication. In our examples, the global sequence is constructed on the server, and the arrival order on the server determines the arbitration order.

10.4.1 Single-Copy Register

We now prove that the single-copy-register (Fig. 1.1) is a linearizable implementation (Fig. 5.1) of the last-writer-wins register $\mathcal{F}_{reg}$ (§4.3.2).

Theorem 10.12. $\Pi_{\mathsf{SingleCopyRegister}} \models$ Linearizability$(\mathcal{F})$.

We follow the proof structure described in §10.1 to prove $\Pi_{\mathsf{SingleCopyRegister}} \models$ RVal$(\mathcal{F}_{reg}) \wedge$ SingleOrder $\wedge$ Realtime.

Auxiliary Definitions. Each operation sends a message to the server,

and the server then replies with an acknowledgment. Because delivery is specified to be reliable, and $G \in \mathcal{E}_{\text{complete}}$, the following events are uniquely defined for each operation $a \in \mathsf{ops}(G)$:

$$a^{\text{srv}} \stackrel{\text{def}}{=} \text{the } e \in E \text{ such that } a \xrightarrow{\mathsf{del}} e$$
$$a^{\text{ack}} \stackrel{\text{def}}{=} \text{the } e \in E \text{ such that } a^{\text{srv}} \xrightarrow{\mathsf{del}} e$$

We now define the global order go on $\mathsf{ops}(E)$ as the order in which the server processes operations:

$$a \xrightarrow{\mathsf{go}} b \quad \stackrel{\text{def}}{\iff} \quad a^{\text{srv}} \xrightarrow{\mathsf{go}} b^{\text{srv}}.$$

For some server event $e \in E(\mathit{Server})$, we define the set of operations that were done on the server up to and including e as $\mathsf{done}(e) = \{a \in \mathsf{ops}(G) \mid a^{\text{srv}} \leq_{\mathsf{eo}} e\}$. For some set of operations $E' \subseteq \mathsf{ops}(G)$, we define the subset of write operations $\mathsf{writes}(E') \stackrel{\text{def}}{=} \{e \in E' \mid \mathsf{op}(e) = \mathsf{wr}(v) \text{ for some } v\}$.

State Invariant. On the server, the implementation stores the latest committed write. Thus, our invariant is:

Lemma 10.13. For all $e \in E(\mathit{Server})$,

$$\mathsf{post}(e).\mathsf{current} = \begin{cases} \mathit{undef} & \text{if } \mathsf{writes}(\mathsf{done}(e)) = \emptyset \\ v & \text{if } \mathsf{op}(\max_{\mathsf{go}} \mathsf{writes}(\mathsf{done}(e)) = \mathsf{wr}(v) \end{cases}$$

Proof. By induction over eo, and case distinction over the type of transition. For initialization transitions, the invariant is established because $\mathsf{done}(e)$ is empty and $\mathsf{post}(e).\mathsf{current}$ is *undef*. For non-initialization transitions, let $p = \mathsf{pred}(E(\mathit{Server}), \mathsf{eo}, e)$ be the predecessor transition on the server. The invariant holds for p by induction, and is preserved: (1) for receive transitions that receive a read request, $\mathsf{post}(e) = \mathsf{post}(p)$ and $\mathsf{writes}(\mathsf{done}(e)) = \mathsf{writes}(\mathsf{done}(p))$, thus the invariant is preserved; (2) for receive transitions that receive a write request, let s be the sending operation event. Then $s = \max_{\mathsf{go}} \mathsf{writes}(\mathsf{done}(e))$, and $\mathsf{current}$ contains the value written by s as required. □

Well-formed. We need to show that each trajectory is well-formed. This is trivial for the server role since it contains no calls or returns. Consider

the events $E' = E(\mathit{Client}(i))$ by client i and an arbitrary event $e \in E$. By Definition 7.4, we need to show that in E', the number of returns preceding e is less than or equal to the number of calls preceding e. To prove this, consider that given a return transition $r \in \mathsf{returns}(E')$, by *dontforge* and injectivity of del we can track it back to a uniquely determined operation $a \in \mathsf{ops}(G)$ such that $r = a^{\mathrm{ack}}$. Since the client number i is tracked by the messages, we know $a \in E'$. Therefore, $\big|\{r \in \mathsf{returns}(E') \mid r \leq e\}\big| = \big|\{a \in \mathsf{calls}(E') \mid a^{\mathrm{ack}} \leq e\}\big| \leq \big|\{a \in \mathsf{calls}(E') \mid a \leq e\}\big|$. Together with condition (t4) on p. 86, this implies that calls and returns alternate.

Witness. Visibility and arbitration are both defined by the global order

$$\mathsf{vis}_A \stackrel{\mathrm{def}}{=} \mathsf{go} \qquad \mathsf{ar}_A \stackrel{\mathrm{def}}{=} \mathsf{go}$$

To prove that vis is acyclic and natural, and that ar is a total order, consider that go is an enumeration (*i.e.* a natural total order) on $\mathsf{ops}(G)$: it is isomorphic to $(E(\mathit{Server}), \mathsf{eo}|_{E(\mathit{Server})})$ which is an enumeration by condition (c4) on p. 87 and condition (t1) on p. 86.

Return Values. Write operations return the correct value *undef*. Consider a read operation $r \in E(\mathit{Client}(i))$. Then, $\mathsf{rval}_A(r)$ (as constructed in condition (x3) on p. 90) is equal to $\mathsf{rval}(r^{\mathrm{ack}})$, because calls and returns alternate (see paragraph on well-formedness above). Therefore, it is the value sent to the transition r^{ack} by the transition r^{srv}. Using the invariant on r^{srv}, and matching it with the definition of $\mathcal{F}_{reg}$ on page 46, we know that this value equals $\mathcal{F}_{reg}(\mathsf{rd}, \mathsf{done}(r^{\mathrm{srv}}), \mathsf{op}, \mathsf{go}, \mathsf{go}) = \mathcal{F}_{reg}(\mathsf{rd}, \mathsf{vis}^{-1}(r) \cup \{r\}, \mathsf{op}, \mathsf{vis}, \mathsf{ar}) = \mathcal{F}_{reg}(\mathsf{rd}, \mathsf{vis}^{-1}(r), \mathsf{op}, \mathsf{vis}, \mathsf{ar}) = \mathcal{F}_{reg}(\mathsf{rd}, \mathsf{context}(A, r))$.

Ordering Guarantees. To prove $A \models \textsc{SingleOrder}$, choose $E' = \emptyset$, then $\mathsf{vis} = \mathsf{go} = \mathsf{ar} = \mathsf{ar} \setminus (E' \times E)$. To prove $A \models \textsc{Realtime}$, consider two operations $a, b \in E_A$. If $a \xrightarrow{\mathsf{rb}} b$, it means that $a^{\mathrm{ack}} <_{\mathsf{eo}} b$. Thus $a^{\mathrm{srv}} <_{\mathsf{eo}} a^{\mathrm{ack}} <_{\mathsf{eo}} b <_{\mathsf{eo}} b^{\mathrm{srv}}$, thus $a \xrightarrow{\mathsf{go}} b$, thus $a \xrightarrow{\mathsf{ar}} b$.

10.4.2 Buffered Sequencer

We now prove correctness of the protocol $\mathsf{BufferedSequencer}\langle\mathcal{F}\rangle$ of Fig. 6.12.

Theorem 10.14.

$$\Pi_{\mathsf{BufferedSequencer}\langle\mathcal{F}\rangle} \models \textsc{CausalConsistency}(\mathcal{F}) \\ \wedge \textsc{ConsistentPrefix}.$$

We follow the proof structure described in §10.1 to prove the required consistency guarantees, which are $\textsc{RVal}(\mathcal{F}_{reg}) \wedge \textsc{CausalVisibility} \wedge \textsc{CausalArbitration} \wedge \textsc{ConsistentPrefix} \wedge \textsc{EventualVisibility}$.

Auxiliary Definitions. For the given concrete execution $G = (E, \mathsf{eo}, \mathsf{tr}, \mathsf{role}, \mathsf{del})$, define the sets U_G of updates and C_G of client roles:

$$U_G \stackrel{\text{def}}{=} \{o \in E \mid \mathsf{op}(o) \neq \bot \ \wedge \ \mathsf{op}(o) \notin \mathsf{readonlyops}(\mathcal{F})\}$$
$$C_G \stackrel{\text{def}}{=} \{r \in \mathsf{roles}(G) \mid r = \mathsf{Client}(i) \text{ for some } i\}$$

The only transition to create an update event $u \in U_G$ is the perform transition by a client $\mathsf{role}(u) \in C_G$, which sends an update message $\mathsf{ToServer}(u)$. The server transition that receives the message in turn broadcasts a $\mathsf{ToAll}(u)$ message. Because delivery is specified to be reliable, and $\mathsf{crashed}(G) = \emptyset$, the following events are uniquely defined for each $u \in U_G$ and $c \in C_G$:

$$u^{\text{srv}} \stackrel{\text{def}}{=} \text{the } e \in E \text{ such that } u \xrightarrow{\mathsf{del}} e$$
$$u^{\text{tell } c} \stackrel{\text{def}}{=} \text{the } e \in E \text{ such that } u^{\text{srv}} \xrightarrow{\mathsf{del}} e$$

We now define the update order uo_G (abbreviated uo) as the order in which the server processes updates:

$$a \xrightarrow{\mathsf{uo}} b \quad \stackrel{\text{def}}{\iff} \quad a^{\text{srv}} \xrightarrow{\mathsf{eo}} b^{\text{srv}}.$$

Then, uo is a total order on U_G, and satisfies for all $c \in C_G$:

$$a \xrightarrow{\mathsf{uo}} b \quad \Leftrightarrow \quad a^{\text{tell } c} \xrightarrow{\mathsf{ro}} b^{\text{tell } c}$$

(where ro is the role order as defined for concrete executions on page 87) because delivery of messages is specified to be pairwise in-order.

State Invariant. The implementation stores information about updates by means of Update structs. For some update $u \in U_G$, we define the corresponding tuple

$$\mathsf{desc}(u) \stackrel{\text{def}}{=} \mathsf{Update}(\mathsf{op}(u), \left|\{x \in U_G \mid x^{\text{tell } \mathsf{role}(u)} \leq_{\mathsf{ro}} u\}\right|, \mathsf{role}(u)).$$

The second entry, the field visprefix, counts how many updates have been delivered by the server to the node peforming u at the time it performs u. This information is used in computeresult to determine visibility when computing a return value.

We can now formulate the state invariant, which clarifies the meaning of the sequences confirmed and pending (the latter is called a queue in the pseudocode, which we model as a sequence to which we append elements on the right and remove elements on the left).

Lemma 10.15 (State Invariant for Clients)**.** For all $e \in E$ such that $\mathsf{role}(e) \in C_G$,

$$\begin{aligned}&\mathsf{post}(e).\mathsf{confirmed}\\ &\qquad = \{u \in U_G \mid u^{\mathrm{tell}\ \mathsf{role}(e)} \leq_{\mathsf{ro}} e\}.\mathsf{sort}(\mathsf{uo}).\mathsf{map}(\mathsf{desc})\\ &\mathsf{post}(e).\mathsf{pending}\\ &\qquad = \{u \in U_G \mid u \leq_{\mathsf{ro}} e <_{\mathsf{ro}} u^{\mathrm{tell}\ \mathsf{role}(e)}\}.\mathsf{sort}(\mathsf{uo}).\mathsf{map}(\mathsf{desc})\end{aligned}$$

where sort and map are the operators we defined in §2.1.1 on p. 20.

The following lemma is an easy consequence of the state invariant.

Lemma 10.16 (Update Descriptors)**.** Let $u \in U_G$ with $\mathsf{role}(u) = \mathit{Client}(i)$. Then $\mathsf{desc}(u) = (\mathsf{op}(e), |\{x \in U_G \mid x^{\mathrm{tell}\ \mathit{Client}(i)} \leq_{\mathsf{ro}} u\}|, i)$.

Witness. We now define visibility and arbitration relations vis_A and ar_A. To this end, we must extend the update order uo in some way, so it orders not only the update operations U_G, but also the read-only operations. We define a "last-update" partial function $\mathsf{lu} : \mathsf{ops}(G) \rightharpoonup U_G$ that, given some operation event e on a client $\mathsf{role}(e) \in C_G$, returns the last update that precedes e:

$$\mathsf{lu}(e) \quad \stackrel{\mathrm{def}}{=} \quad \max_{\mathsf{uo}} \{u \in U_G \mid u \leq_{\mathsf{ro}} e \ \vee\ u^{\mathrm{tell}\ \mathsf{role}(e)} \leq_{\mathsf{ro}} e\} \tag{10.2}$$

Note that $\mathsf{lu}(e) = \bot$ if the set on the right-hand-side is empty. Also, note that for update events $u \in U_G$, $\mathsf{lu}(u) = u$.

Using the last-update function, we can now define arbitration and

visibility. First, we define the arbitration witness ar_A (abbreviated ar):

$$\begin{aligned} a \xrightarrow{\mathsf{ar}} b \quad &\overset{\text{def}}{\iff} \quad \mathsf{lu}(a) <_{\mathsf{uo}} \mathsf{lu}(b) \quad \vee \quad \mathsf{lu}(a) = \bot \neq \mathsf{lu}(b) \\ &\vee \quad (a <_{\mathsf{eo}} b \quad \wedge \quad (\mathsf{lu}(a) = \mathsf{lu}(b) \quad \vee \quad \mathsf{lu}(a) = \bot = \mathsf{lu}(b))) \end{aligned} \tag{10.3}$$

The idea is that update operations are ordered by the update order, read-only operations with different last updates are ordered the same way as those updates, and otherwise (if the updates do not establish the order because they are the same or not defined), we use execution order eo to order the operations.

Next, we define the visibility witness vis_A (abbreviated vis):

$$\begin{aligned} a \xrightarrow{\mathsf{vis}} b \quad &\overset{\text{def}}{\iff} \quad a <_{\mathsf{eo}} b \ \wedge \\ &(a \xrightarrow{\mathsf{ro}} b \quad \vee \quad \mathsf{lu}(a) = \bot \quad \vee \quad \mathsf{lu}(a)^{\text{tell } \mathsf{role}(\mathsf{lu}(b))} \leq_{\mathsf{ro}} \mathsf{lu}(b)) \end{aligned} \tag{10.4}$$

The idea is that operations are visible to other operations if they precede them in execution order, and their respective last updates are either on the same role, or notification of the first update was delivered to the role performing the second update before performing the second update.

The following lemmas imply that arbitration and visibility satisfy the basic requirements for a witness.

Lemma 10.17. ar is a total order on $\mathsf{ops}(G)$.

Proof. See § A.1.5 in the appendix. □

Lemma 10.18. vis is a natural partial order on $\mathsf{ops}(G)$.

Proof. See § A.1.6 in the appendix. □

Return Values. We need to prove that return values are consistent with the replicated data type $\mathcal{F}$ and the visibility and arbitration witness:

$$\forall e \in \mathsf{ops}(G) : \quad \mathsf{rval}_A(e) = \mathcal{F}(\mathsf{op}(e), A|_{\mathsf{vis}^{-1}(e),\mathsf{op},\mathsf{vis},\mathsf{ar}}) \tag{10.5}$$

The following lemma establishes that the return value is indeed correctly computed by the implementation. It implies equation (10.5) because $\mathcal{F}$ is insensitive to isomorphisms, and to the removal of read-only

operations, and because $\mathsf{rval}_A(e) = \mathsf{rval}(e)$. Its proof uses the state invariant, Lemma 10.16, and Lemma 10.20.

Lemma 10.19. Let $e \in \mathsf{ops}(G)$ be an operation transition, and let s be the sequence of update structs that is passed to the computeresult operation (*i.e.* $s = \mathsf{pre}(e).\mathsf{confirmed} \cdot \mathsf{pre}(e).\mathsf{pending}$). Then, the operation context

$$\begin{aligned}&(\{0, 1, \ldots, |s| - 1\},\\ &\lambda i.s[i].\mathsf{op},\\ &\{(i,j) \mid i < j \ \wedge \ (s[i].\mathsf{cid} = s[j].\mathsf{cid} \ \vee \ i < s[j].\mathsf{visprefix})\},\\ &\{(i,j) \mid i < j\})\end{aligned}$$

is isomorphic to the operation context $(\mathsf{vis}^{-1}(e), \mathsf{op}, \mathsf{vis}, \mathsf{ar})|_{U_G}$.

Lemma 10.20. Let $e \in \mathsf{ops}(G)$. Then

$$\begin{aligned}\{u \in U_G \mid u \xrightarrow{\mathsf{vis}} e\}.\mathsf{sort}(\mathsf{ar}) \ = \\ \{u \in U_G \mid u^{\text{tell } \mathsf{role}(e)} <_{\mathsf{ro}} e\}.\mathsf{sort}(\mathsf{uo}) \cdot \\ \{u \in U_G \mid u <_{\mathsf{ro}} e <_{\mathsf{ro}} u^{\text{tell } \mathsf{role}(e)}\}.\mathsf{sort}(\mathsf{uo})\end{aligned}$$

Ordering Guarantees. We prove the three ordering guarantees in the three lemmas below.

Lemma 10.21. $A \models$ CausalVisibility.

Proof. Since vis is transitive by Lemma 10.18, it suffices to show so $\subseteq$ vis. This is guaranteed because $a \xrightarrow{\mathsf{so}} b$ implies $a <_{\mathsf{eo}} b$ and $a \xrightarrow{\mathsf{ro}} b$ and thus $a \xrightarrow{\mathsf{vis}} b$. □

Lemma 10.22. $A \models$ CausalArbitration.

Proof. Since ar is transitive (Lemma 10.17), it is enough to show (a) so $\subseteq$ ar and (b) (vis$\backslash$so) $\subseteq$ ar. To prove so $\subseteq$ ar: If $a \xrightarrow{\mathsf{so}} b$ then $a <_{\mathsf{eo}} b$. If $\mathsf{lu}(a) = \bot$, $a \xrightarrow{\mathsf{ar}} b$ follows directly. Otherwise, we know $\mathsf{lu}(a) \leq_{\mathsf{uo}} \mathsf{lu}(b)$, because $\mathsf{lu}(b)$ takes the maximum of a larger set than $\mathsf{lu}(a)$ (10.2). Therefore, $a \xrightarrow{\mathsf{ar}} b$ (10.3). To prove (vis $\setminus$ so) $\subseteq$ ar: If $a \xrightarrow{\mathsf{vis}} b$ but not $a \xrightarrow{\mathsf{so}} b$, then $a <_{\mathsf{eo}} b$ and either (a) $\mathsf{lu}(a) = \bot$ or (b) $\mathsf{lu}(a)^{\text{tell } \mathsf{role}(\mathsf{lu}(b))} \leq_{\mathsf{ro}}$

$\mathsf{lu}(b)$. (a) implies $a \xrightarrow{\mathsf{ar}} b$ directly. (b) on the other hand implies $\mathsf{lu}(a) \in \{u \in U_G \mid u \leq_{\mathsf{ro}} \mathsf{lu}(b) \ \vee \ u^{\text{tell } \mathsf{role}(\mathsf{lu}(b))} \leq_{\mathsf{ro}} \mathsf{lu}(b)\}$, which by maximality of $\mathsf{lu}(b)$ (10.2) implies $\mathsf{lu}(a) \leq_{\mathsf{uo}} \mathsf{lu}(b)$, thus $a \xrightarrow{\mathsf{ar}} b$. □

Lemma 10.23. $A \models$ ConsistentPrefix.

Proof. We need to show that for all $a, b, c \in \mathsf{ops}(G)$ such that $a \xrightarrow{\mathsf{ar}} b \xrightarrow{\mathsf{vis}} c$ and $\mathsf{role}(b) \neq \mathsf{role}(c)$, we have $a \xrightarrow{\mathsf{vis}} c$. Distinguish cases based on the disjunction in (10.4) for $b \xrightarrow{\mathsf{vis}} c$. [$b \xrightarrow{\mathsf{ro}} c$.] Not possible since $\mathsf{role}(b) \neq \mathsf{role}(c)$. [$\mathsf{lu}(b) = \bot$.] Then by (10.3), $\mathsf{lu}(a) = \bot$ and $a \xrightarrow{\mathsf{eo}} b$, thus $a \xrightarrow{\mathsf{eo}} c$, thus $a \xrightarrow{\mathsf{ar}} c$. [$\mathsf{lu}(b)^{\text{tell } \mathsf{role}(\mathsf{lu}(c))} \leq_{\mathsf{ro}} \mathsf{lu}(c)$.] Then $\mathsf{lu}(b) \leq_{\mathsf{uo}} \mathsf{lu}(c)$, thus $b \xrightarrow{\mathsf{ar}} c$, thus $a \xrightarrow{\mathsf{ar}} c$ by transitivity of ar. □

Liveness Guarantee. To prove $A \models$ EventualVisibility, let $e \in \mathsf{ops}(G)$, and let $[f] \in E/{\approx_{ss}}$ be a session. We need to show that e is visible to almost all $e' \in [f]$. If $\mathsf{lu}(e) = \bot$, then $e <_{\mathsf{eo}} e'$ for almost all e' (eo is an enumeration). Otherwise, let $x = \mathsf{lu}(e)^{\text{tell } \mathsf{role}(f)}$. Then $x <_{\mathsf{eo}} e'$ for almost all $e' \in E(\mathsf{role}(f))$. Thus $e \xrightarrow{\mathsf{vis}} e'$ for almost all $e' \in [f]$.

11

Related Work

Consistency models and distributed protocols are relevant for programmers across many disciplines. Not surprisingly, related work appears in many different communities. We briefly examine the most important connections below. For a deeper study, we recommend specialized surveys and collections such as Terry [2008], Saito and Shapiro [2005], Bernstein and Das [2013], or Charron-Bost et al. [2010].

11.1 Distributed Systems

Replication, lazy update propagation, and conflict resolution are commonly used in distributed systems, often without using the term eventual consistency. We give an overview of some applications in §1.2.

The Bayou system [Terry et al., 1995, 1994, Petersen et al., 1997], which provides eventually consistent replicated databases on mobile devices, stands out for its clear articulation of the underlying challenges of update propagation and conflict resolution, and for formulating abstract consistency properties called session guarantees [Terry et al., 1994, Terry, 2011] (which correspond to the ordering guarantees in §5). It coined the term eventual consistency, and defined it to mean

what we here call quiescent consistency: replicas eventually converge once updates stop (§4.2.1).

Brewer's articulation of the CAP conjecture [Brewer, 2000] sparked interest in eventual consistency as it relates to the construction of scalable, available, and reliable services (IEEE Computer CAP retrospective, 2012). Reformulations such as PACELC [Abadi, 2012] include the quantitative aspect: availability in a strongly consistent system is hampered by slow connections, not just by broken connections.

Key-Value Stores. Amazon's Dynamo system [DeCandia et al., 2007] demonstrated the utility and effectiveness of eventual consistency for the purpose of building scalable, available, and reliable storage systems, and has become widely cited. Eventual consistency has been extensively used for this purpose, fueled by the trend of providing virtual compute and storage services in the cloud, and the noSQL movement.

Replicated Data Types. Developing optimized consistency protocols for datatypes like counters, sets, and lists has the potential to further improve scalability and availability [Burckhardt et al., 2014a, Shapiro et al., 2011a,c,b, Roh et al., 2011, Bieniusa et al., 2012a]. Of particular interest are conflict-free replicated data types (CRDT's) [Shapiro et al., 2011a,c,b] which provide scalable symmetric implementations for many common data types.

11.2 Databases

In databases, consistency is not just about individual operations, but about sequences of operations called *transactions*. Conceptually, we can think of transactions as augmenting a data type (such as a key-value store, or a relational database) with the capability of executing several operations atomically, *i.e.* as if they were a single operation. This gives programmers the freedom to invent new operations at any time, which is important for situations where the precise set of queries and updates cannot be known *a priori*.

The strongest consistency model for databases is one-copy serializability (1SR) [Alsberg and Day, 1976]. It corresponds to linearizability

at the transaction level. Note that in the database literature, the abstract consistency model used for transactions is generally called the *isolation level*, not to be confused with data consistency (meaning compliance with data invariants specified by the schema).

Research on database replication began in the 1970's, and commercial database management systems (DBMS) started supporting replication in the late 1980's (as recollected by Bernstein and Das [2013]). The unavoidable trade-off between consistency, availability, and partition tolerance had been known to the database community long before the CAP formulation became popular [Rothnie and Goodman, 1977]. Eventually consistent replication (or multi-master replication) appears early on in Thomas' majority consensus algorithm [Thomas and Beranek, 1979].

Note that replication is not the only motivation to consider weaker consistency models: efficient concurrency control is important even for non-distributed databases. In fact, it is very common for commercial database systems to use isolation levels weaker than 1SR, such as read-committed (the default setting of commercial databases) or snapshot isolation [Fekete et al., 2005, Berenson et al., 1995].

Our formalism can be extended with transactions to formalize isolation levels, as demonstrated in Burckhardt et al. [2013]. This style of formalization is also recommended by Fekete and Ramamritham [2010], using executions augmented with a justification similar to the visibility and arbitration relations in our abstract executions.

Parallel snapshot isolation [Sovran et al., 2011], which is slightly weaker than snapshot isolation, can help to mitigate the cost of wide-area communication in georeplicated databases.

Weak forms of transactional guarantees can be made available under partitions, using consistency models such as eventually consistent transactions [Burckhardt et al., 2012a, 2014b], causally consistent transactions [Li et al., 2012, Lloyd et al., 2013], or highly available transactions [Bailis et al., 2013, 2014].

11.3 Shared-Memory Multiprocessors

Researchers recognized early on that the use of caching in multiprocessors can lead to consistency problems, even for programs that perform only simple loads and stores. In a brief paper on this topic, Lamport [1979] describes the Dekker anomaly (see §5.2.1) and sketches an axiomatic definition of sequential consistency (SC).

Since architects are generally unwilling to pay the cost of SC (never mind the even stronger linearizability), most multiprocessor architectures use weaker consistency models [Adve and Gharachorloo, 1996], despite the challenge they pose to developers. In practice, however, most programmers can steer clear of non-sequentially-consistent behaviors simply by writing data-race-free programs [Adve and Hill, 1993, Boehm and Adve, 2008]. Where this is not possible (e.g. in lock-free algorithms where deliberate data races are needed for optimal performance), programmers need to insert appropriate fences, which can be challenging. To solve this problem, researchers have proposed to manually check programs for the presence of certain types of cycles [Shasha and Snir, 1988], or to run static tools to detect problems caused by the weak consistency [Burckhardt et al., 2007] and/or automatically insert fences [Kuperstein et al., 2012, Abdulla et al., 2012].

Formalizations of memory consistency models can be categorized into axiomatic models and operational models, as described in §3.4. For research and programming purposes, it is often desirable to have equivalent formulations of a model in both operational and axiomatic styles. Proofs of equivalence can be difficult to obtain; one direction of such equivalence proofs (showing that an operational model satisfies an axiomatic specification) corresponds to the correctness proofs in chapter 10.

Early proposals for weak memory consistency models include the SPARC TSO/PSO/RMO models [Weaver and Germond, 1994], Processor Consistency (PC) and Release Consistency (RC) [Gharachorloo, 2005], and Collier's family of models [Collier, 1992], among many others. The TSO model, which is only slightly weaker than SC, is now used by Intel's x86 architecture [Sewell et al., 2010]. The PowerPC model, which is particularly weak, has proven to be challenging to understand

and formalize [Sarkar et al., 2011]. The C++ memory model [Batty et al., 2011, 2013] and the Java memory model [Manson et al., 2005] are language-level memory models that provide a hardware-independent layer. Validating programs and compilers against these models remains an active area of research.

Formalization style. The use of ordering and equivalence relations over events to formulate axiomatic memory consistency models is a recurring theme in the literature [Shasha and Snir, 1988, Collier, 1992, Lamport, 1979, Burckhardt, 2007, Yang, 2005, Batty et al., 2011]. These models follow a style that is very similar to ours, with some differences:

- The data type $\mathcal{F}$ is "memory", which is essentially a key-value store (§4.3.3), with keys corresponding to memory addresses, augmented with fences and atomic read-modify-write instructions.

- Histories use different names and record slightly different information: operations are called instructions, sessions are called processes, session order is called program order, and there may be additional relations that capture control- and data-dependencies.

- Abstract executions do typically not use visibility and arbitration relations, but reads-from and coherence relations (the reads-from relation expresses how values flow from stores to load, and the coherence relation expresses how to order stores with respect to a single location).

The connection between these approaches, *i.e.* how to compare an axiomatic model for eventual consistency with with the C/C++ relaxed memory model [Batty et al., 2011, 2013] is examined in detail in Burckhardt et al. [2013].

11.4 Distributed Algorithms

The material in this tutorial intersects substantially with classic themes of Distributed Computing, most noteably linearizability, asynchronous network protocols, and verification.

Linearizability as a consistency criterion that is independent of the particular data type was introduced by Herlihy and Wing [1990]. Linearizable objects (also known as atomic objects, as discussed *e.g.* in Lynch [1996]) have been extensively studied, both in terms of general constructions or impossibility results, as well as in terms of efficient network protocols or shared-memory algorithms for specific data types. Because linearizability is easy to explain and has good properties (e.g. compositionality), it remains the golden standard for implementations of shared objects where communication latency and reliability is not a major problem (in particular, on shared-memory multiprocessors) [Heller et al., 2005, Harris, 2001, Harris et al., 2002].

The observation that linearizability cannot be guaranteed while remaining available under network partitions (*i.e.* the CAP theorem) was proved by Gilbert and Lynch [2002] (including both an asynchronous and a synchronous variant). Our version of CAP (§9.1) is a more general version of the asynchronous case: it applies to sequential consistency, not linearizability, and applies for any data type with independently observable writes. A quantitative version of CAP (a lower bound on read and write latency in sequentially consistent systems) can also be found in Attiya and Welch [1998].

Our consistency protocols illustrate the concept of *layering* of transport abstractions. This is a standard theme in distributed algorithms: for example, we can construct reliable protocols on top of unreliable ones, ordered ones on top of unordered ones, and secure ones on top of insecure ones [Cachin et al., 2011, Lynch, 1996, Attiya and Welch, 1998]. In this sense, our consistency models can be understood as high-level delivery guarantees for update propagation. Logical clocks [Lamport, 1978] and vector clocks [Fidge, 1988, Mattern, 1989] are techniques that are commonly used in this context.

11.5 Verification

Simply stated, verification is about separating the 'what' (the specification) from the 'how' (the implementation). This separation is an essential skill, required for modularizing complex systems, and therefore

relevant for audiences beyond formal methods. Thus, how to best specify and verify concurrent or distributed systems is a recurring theme in computer science. Contributions to this area include many Turing award winners, most noteably Leslie Lamport. His award citation mentions causality, logical clocks, safety, liveness, replicated state machines, and sequential consistency, all of which are topics in this tutorial.

We can categorize verification broadly based on the nature of specifications: *property verification*, where we prove that all reachable states or all executions satisfy a certain property, or *refinement verification*, where we prove that all executions are observationally equivalent to some execution of an operational specification. Our method is most similar to property verification (because our consistency guarantees are a conjunction of formulas) but has also some similarity to refinement verification (because we use an existential quantification over abstract executions).

The idea of modeling protocol participants using automata (states and atomic transitions) is standard and can be found both in algorithm textbooks [Cachin et al., 2011, Lynch, 1996, Attiya and Welch, 1998] and verification tools [Dill, 1996, Lamport, 1994]. However, the terminology and formal details exhibit many minor variations. Compositional reasoning can be tricky to get right, in particular when including fairness and liveness, but has been solved by the formulation of I/O automata [Lynch and Tuttle, 1987, Lynch, 1996]. Our formalization of protocols is a specialization of the latter.

Verification of linearizable objects in the shared-memory setting has received much attention over the years, including mechanically-verified interactive proofs and automatic model checking tools [Colvin et al., 2006, Vafeiadis et al., 2006, Burckhardt et al., 2010].

Contrary to most work on protocol verification, we are not primarily concerned with automation (yet). In its current form, our methodology is meant for manual proofs, and is intended more as an education than a certification tool: for now, the main purpose of our proofs is to convey insight into the workings of an implementation, rather than to provide infallible (*i.e.* mechanically checked) evidence of correctness.

12

Conclusion

We have reached the end of this tutorial, yet in some sense, this is a beginning: what we have built is a foundation upon which to build. By delineating the basic principles of eventual consistency, we have erected a framework for reasoning about specifications and implementations, thus informing the development of globally distributed systems.

Our formalization of consistency models brings systems and definitions from across space, time, and research communities together in one place. We hope it will promote a wider understanding and a common terminology.

Our emphasis was on combining four activities that we consider essential to a principled approach: formulating consistency specifications, writing protocol implementations, proving correctness of implementations, and proving impossibility results. We have carried out all of these in a level of detail that we consider appropriate for an audience with an appreciation of mathematics.

Developing correct distributed protocols can be quite daunting in general, and our protocol examples may appear overly simplistic. However, the best preparation for complexity is a good understanding of simplicity: In our own experience, proving the correctness of a number

of simple protocols leads to a deep understanding of recurring design principles and techniques, which then opens the door for correctly composing more complex protocols.

Acknowledgements

I want to thank my collaborators Alexey Gotsman, Hongseok Yang, and Marek Zawirski, as well as Marc Shapiro, for the many insightful discussions that have inspired me to pursue this line of work. The material presented here is closely aligned with our collaborative work on formalizing eventual consistency and replicated data types [Burckhardt et al., 2013, 2014a]. I also want to thank my wife Andi Wolff for her help and support.

Appendices

A

Selected Proof Details

We include some proof details in this appendix.

A.1 Lemmas

We include proofs of several minor lemmas in this section, for reference. All of these proofs are relatively easy and perhaps somewhat boring — whatever is interesting is included in the main text.

A.1.1 Proof of Lemma 2.2

Choose $N = \mathbb{N}_0$ if $|E| = \infty$ or $N = \{0, \ldots, |E| - 1\}$ otherwise. Define $\phi : E \to N$ by $\phi(e) = |\mathsf{rel}^{-1}(e)|$. Then, ϕ preserves order: if $e \xrightarrow{\mathsf{rel}} e'$ then $\mathsf{rel}^{-1}(e) \subseteq \mathsf{rel}^{-1}(e')$, and since $e' \not\xrightarrow{\mathsf{rel}} e$ (by acyclicity), $\mathsf{rel}^{-1}(e) \subsetneq \mathsf{rel}^{-1}(e')$, thus $\phi(e) = |\mathsf{rel}^{-1}(e)| < |\mathsf{rel}^{-1}(e')| = \phi(e')$. Also, ϕ is a bijection because it is injective ($\forall e, e' : e \neq e' \Rightarrow \phi(e) \neq \phi(e')$) and surjective ($\forall n \in N : \exists e \in E : \phi(e) = n$). First, we show that ϕ is injective: for $e \neq e'$, without loss of generality $e \xrightarrow{\mathsf{rel}} e'$ because rel is total. Thus $\phi(e) < \phi(e')$ implies $\phi(e) \neq \phi(e')$. Finally, we show that ϕ is surjective: if N is finite, this follows from injectivity because both sets are the same size; otherwise, let n be the smallest number

not in $\phi(E)$. Since $|E|$ is infinite and injective, $\phi(E)$ must be infinite, thus there must exist an $n_0 > n$ such that $n_0 \in \phi(E)$. Pick a minimal such n_0. Let $e \in E$ be the element such that $\phi(e) = n_0$, and let $e' = \max_{\mathsf{rel}}\{e' \in E \mid e' \xrightarrow{\mathsf{rel}} e\}$. Then, $\mathsf{rel}^{-1}(e') = \mathsf{rel}^{-1}(e) \setminus e'$, thus $\phi(e') = \phi(e) - 1$. But this contradicts either minimality of n_0 (if $\phi(e') \neq n$) or $n \notin \phi(E)$ (if $\phi(e') = n$).

A.1.2 Proof of Lemma 10.1

By Definition 9.1 we need to prove $\mathcal{E}(\Pi) \subseteq \mathcal{E}_{\text{wellformed}}$ and $\forall G \in \mathcal{E}_{\text{complete}}(\Pi) : \quad \mathcal{H}(G) \models \text{RVAL}(F) \wedge \mathcal{P}_1 \wedge \cdots \wedge \mathcal{P}_n$. The latter follows because we chose an arbitrary $G \in \mathcal{E}_{\text{complete}}(\Pi)$ and constructed an A such that $\mathcal{H}(A) = \mathcal{H}(G)$ and $\mathcal{H}(A) \models \text{RVAL}(F) \wedge \mathcal{P}_1 \wedge \cdots \wedge \mathcal{P}_n$. For the former, progress (Theorem 9.3) guarantees that any $G' \in \mathcal{E}(\Pi)$ is a prefix $G' \sqsubseteq G$ of a complete execution $G \in \mathcal{E}_{\text{complete}}(\Pi)$, for which we prove $G \in \mathcal{E}_{\text{wellformed}}$, thus also $G' \in \mathcal{E}_{\text{wellformed}}$ (because it is a safety property by Lemma 7.5).

A.1.3 Proof of Lemma 10.2

If $\mathsf{pre}(e) = \bot$, it is an initialization transition, thus there is no x such that $x \xrightarrow{\mathsf{del}} e$ because only receive transitions can receive messages (condition (c5) on p. 87). Also, there is no x such that $x \xrightarrow{\mathsf{ro}} e$ because an initialization transitions is the first transitions by each role (condition (c4) and condition (t3) on p. 86). Thus $V(e) = \emptyset$. If $\mathsf{pre}(e) \neq \bot$, the claimed equality of sets follows from $\supseteq$ and $\subseteq$, which we each discuss in the following two paragraphs.

Observation (1): the right-hand side is contained in the left-hand side, because:

- $V(p) \subseteq V(e)$ because $p \xrightarrow{\mathsf{ro}} e$ implies that any path $x \xrightarrow{\mathsf{ro} \cup \mathsf{del}} * p$ can be extended to a path $x \xrightarrow{\mathsf{ro} \cup \mathsf{del}} * e$.
- if $\mathsf{op}(e) \neq \bot$, then $e \in V(e)$ because trivially $e \xrightarrow{\mathsf{ro} \cup \mathsf{del}} * e$.
- since s satisfies $s \xrightarrow{\mathsf{del}} e$, $V(s) \subseteq V(e)$ because any path $x \xrightarrow{\mathsf{ro} \cup \mathsf{del}} * s$ can be extended to a path $x \xrightarrow{\mathsf{ro} \cup \mathsf{del}} * e$.

Observation (2): the left-hand side is contained in the right-hand side. Let $x \in V(e)$. Then $\mathsf{op}(x) \neq \bot$ and $x \xrightarrow{\mathsf{ro} \cup \mathsf{del}}{}^* e$. We now show this implies that x is contained in the right-hand side. If $x = e$ then $\mathsf{op}(e) \neq \bot$, thus x is contained in the right-hand side (second line). Otherwise, there exists a y such that we have a path of the form $x \xrightarrow{\mathsf{ro} \cup \mathsf{del}}{}^* y \xrightarrow{\mathsf{ro} \cup \mathsf{del}} e$. Now we do a case distinction on the kind of the last edge in this path.

- Consider $y \xrightarrow{\mathsf{ro}} e$. Then, because p is the ro-maximal event satisfying $p \xrightarrow{\mathsf{ro}} e$ (by definition of pred), we must have $y \xrightarrow{\mathsf{ro}}{}^* p$. Thus $x \in V(p)$, and thus x is contained in the right-hand side (second, third, or fourth line).

- Consider $y \xrightarrow{\mathsf{del}} e$. Then $s = y$, and $\mathsf{rcv}(e) \in \mathsf{snd}(y)$ (condition (c5) on p. 87). Thus, $V(y)$ is contained in the right-hand side (third line), and thus also x.

A.1.4 Proof of Lemma 10.5

If $A \subseteq B$ or $B \subseteq A$ then the claim follows easily. We now show that one of those is always the case, by reductio ad absurdum: if neither of those is true, then there exist $x \in A \setminus B$ and $y \in B \setminus A$. Now, $x \xrightarrow{\mathsf{rel}} y$ is not possible since it would imply $x \in B$ because B is predecessor closed, contradicting $x \in A \setminus B$. Symmetrically, $y \xrightarrow{\mathsf{rel}} x$ is not possible. Thus rel is not total, which is a contradiction.

A.1.5 Proof of Lemma 10.17

(1) ar is *irreflexive*: never $\mathsf{lu}(a) <_{\mathsf{uo}} \mathsf{lu}(a)$, nor $a <_{\mathsf{eo}} a$. (2) ar is *total*: given arbitrary a, b. If $\mathsf{lu}(a) = \bot = \mathsf{lu}(b)$ or $\mathsf{lu}(a) = \mathsf{lu}(b)$, then a, b are ordered by ar because eo is total. If exactly one of $\mathsf{lu}(a), \mathsf{lu}(b)$ is $\bot$, then they are ordered. If none is $\bot$, they are ordered because uo is total. (3) ar is *transitive*: suppose $a \xrightarrow{\mathsf{ar}} b \xrightarrow{\mathsf{ar}} c$. Distinguish cases. $[\ \mathsf{lu}(a) = \bot, \mathsf{lu}(c) \neq \bot.\ ]$ $a \xrightarrow{\mathsf{ar}} c$ is immediate. $[\ \mathsf{lu}(a) = \bot, \mathsf{lu}(c) = \bot.\ ]$ Then also $\mathsf{lu}(b) = \bot$, thus it follows from transitivity of eo. $[\ \mathsf{lu}(a) \neq \bot, \mathsf{lu}(b) = \bot.\ ]$ Impossible. $[\ \mathsf{lu}(a) \neq \bot, \mathsf{lu}(b) \neq \bot, \mathsf{lu}(c) = \bot.\ ]$ Impossible. $[\ \mathsf{lu}(a) \neq \bot, \mathsf{lu}(b) \neq \bot, \mathsf{lu}(c) \neq \bot.\ ]$ Do a second-level case distinction. $[\![\ \mathsf{lu}(a) = \mathsf{lu}(b) = \mathsf{lu}(c).\ ]\!]$ Then $a <_{\mathsf{eo}} c$, thus $a \xrightarrow{\mathsf{ar}} c$. $[\![\ \mathsf{lu}(a) = \mathsf{lu}(b) <_{\mathsf{uo}}$

$\mathsf{lu}(c)$. $\rrbracket$ Then $\mathsf{lu}(a) <_{\mathsf{uo}} \mathsf{lu}(c)$, thus $a \xrightarrow{\mathsf{ar}} c$. $\llbracket$ $\mathsf{lu}(a) <_{\mathsf{uo}} \mathsf{lu}(b) = \mathsf{lu}(c)$. $\rrbracket$ Same argument. $\llbracket$ $\mathsf{lu}(a) <_{\mathsf{uo}} \mathsf{lu}(b) <_{\mathsf{uo}} \mathsf{lu}(c)$. $\rrbracket$ Same argument.

A.1.6 Proof of Lemma 10.18

(1) vis is *natural* and *irreflexive*: follows because $\mathsf{vis} \subseteq \mathsf{eo}$ and eo is natural and irreflexive. (2) vis is *transitive*: suppose $a \xrightarrow{\mathsf{vis}} b \xrightarrow{\mathsf{vis}} c$. Then $a <_{\mathsf{eo}} b <_{\mathsf{eo}} c$. Do a case distinction on the disjunctions in (10.4). $[$ $a \xrightarrow{\mathsf{ro}} b, b \xrightarrow{\mathsf{ro}} c$. $]$ Then $a \xrightarrow{\mathsf{ro}} c$. $[$ $a \xrightarrow{\mathsf{ro}} b, \mathsf{lu}(b) = \bot$. $]$ Then $\mathsf{lu}(a) = \bot$ also (Lemma A.1), thus $a \xrightarrow{\mathsf{vis}} c$. $[$ $a \xrightarrow{\mathsf{ro}} b, \mathsf{lu}(b)^{\mathsf{tell}\ \mathsf{role}(\mathsf{lu}(c))} \leq_{\mathsf{ro}} \mathsf{lu}(c)$. $]$ Apply Lemma A.1 to $a \xrightarrow{\mathsf{ro}} b$, consider subcases. $\llbracket$ $\mathsf{lu}(a) <_{\mathsf{uo}} \mathsf{lu}(b)$. $\rrbracket$ Then, because notifications are sent from server to client in order, $\mathsf{lu}(a)^{\mathsf{tell}\ \mathsf{role}(\mathsf{lu}(c))} \leq_{\mathsf{ro}} \mathsf{lu}(b)^{\mathsf{tell}\ \mathsf{role}(\mathsf{lu}(c))} \leq_{\mathsf{ro}} \mathsf{lu}(c)$, thus $a \xrightarrow{\mathsf{vis}} c$. $\llbracket$ $\mathsf{lu}(a) = \bot$. $\rrbracket$ Then $a \xrightarrow{\mathsf{vis}} c$.

Lemma A.1. $\forall a, b \in \mathsf{ops}(G) :\ a \xrightarrow{\mathsf{ro}} b \Rightarrow \mathsf{lu}(a) \leq_{\mathsf{uo}} \mathsf{lu}(b)\ \vee\ \mathsf{lu}(a) = \bot$.

Proof. $a \xrightarrow{\mathsf{ro}} b$ implies that $\mathsf{lu}(b)$ takes the maximum of a larger set than $\mathsf{lu}(a)$, thus either $\mathsf{lu}(a) \leq_{\mathsf{uo}} \mathsf{lu}(b)$ or $\mathsf{lu}(a) = \bot$. □

Lemma A.2. Arbitration, when restricted to updates, matches the update order: $\mathsf{ar}|_{U_G} = \mathsf{uo}$.

Lemma A.3. Arbitration, when restricted to updates by a specific role r, matches the role order: $\forall r \in \mathsf{Roles} : \mathsf{ar}|_{E(r)} = \mathsf{ro}$.

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.