Conflict-free Replicated Data Types have Abstract Data Types

Abel Nieto Rodriguez

Ph.D. Dissertation

Department of Computer Science
Aarhus University
Denmark

Conflict-free Replicated Data Types have Abstract Data Types

A dissertation
presented to the Faculty of Natural Sciences
of Aarhus University
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

by
Abel Nieto Rodriguez
July 6, 2023

Abstract

This dissertation develops a unified approach to the verification of *Conflict-free Replicated Data Types* (CRDTs). In a separate but related line of inquiry, the dissertation explores a *client-centric* view of the *encoding of causality* in separation logic that underpins the verification of CRDTs.

CRDTs are a class of distributed data structures that are weakly consistent and highly available. CRDTs achieve high availability by avoiding inter-replica synchronization before updates; instead, updates are immediately executed locally, which leads to inconsistencies among replica states. These inconsistencies are later automatically resolved through custom conflict-resolution policies.

There are two main families of CRDTs, depending on the implementation technique used: *operation-based* (op-based) and *state-based*. They differ in their replication mechanism and how they meet their consistency model, which typically is causal consistency. Op-based CRDTs propagate *individual* local updates to other replicas and rely on an underlying communication protocol that guarantees causal delivery. State-based CRDTs propagate their *entire* state to other replicas, and ensure causal delivery by drawing states from a join semilattice, so that states are merged through joins.

Because the semantics of CRDTs are complicated by their concurrent nature and the need for conflict resolution, there has been widespread interest in formal method techniques for specifying and verifying CRDTs. Existing techniques, however, are unable to abstract over the two kinds of CRDT designs. *What is missing is a view of CRDTs as abstract data types.* Such abstract view of CRDTs would both clarify their semantics (just *what* is a CRDT?) and unlock modular verification of client programs against an abstract CRDT specification, so that clients can reason about a CRDT without knowing how it is implemented. This would put CRDTs on par with traditional sequential data structures like sets or maps, which admit different implementations yet are understood abstractly as a collection of axiomatically-specified operations.

This dissertation fills in the missing abstraction gap by developing a unified approach to specifying and verifying both classes of CRDTs. This unified approach uses separation logic to give functional correctness specifications for both op-based and state-based CRDTs. Drawing on prior work on *encoding causality* in separation logic, as well as an expressive specification style that uses CRDT *denotations*, the dissertation presents CRDT specifications that enjoy a number of desirable properties: they are *modular*, support verification of whole programs including *client code*, describe realistic CRDT *implementations*, and are *formally verified* in the Coq proof assistant in a *foundational* style.

To demonstrate feasibility of the proposed techniques, the dissertation presents a set of libraries for building formally verified CRDTs. Using these libraries we go on to verify an extensive family of example CRDTs, including higher-order combinators. Finally, the dissertation shows how we can give the *same* specification to two implementations of a CRDT, so that a client program can be verified against an abstract specification without knowledge of how the CRDT is implemented.

In a related line of work, the dissertation shows how the encoding of causality in separation logic that is key to specifying CRDTs is also strong enough to imply a number of *session guarantees*. These guarantees serve as reasoning principles for clients of weakly consistent replicated data types. Additionally, this result is evidence that the modelling of causality in separation logic correctly captures trace properties of causal consistency.

Together, the material in this dissertation expands our ability to formally reason about weakly consistent replicated data types in a modular, foundational style.

Resumé

Denne Ph.D.-afhandling udvikler en samlet metode og tilgang til at verificere *Konflikfrie Replikerede Datatyper* (KRDT'er). I et særskilt men relateret arbejde udforsker afhandlingen et *klientfokuseret* syn på, hvordan den *kausalitet*, der underbygger verifikationen af KRDT'er, enkodes i separationslogik.

KRDT'er er en klasse af distribuerede datastrukturer, der er svagt konsistente og højt tilgængelige. KRDT'er opnår høj tilgængelighed ved at undgå synkronisering mellem replikaer, før operationer på datastrukturen udføres. Operationerne bliver i stedet for lokalt udført med det samme, hvilket kan medføre uoverensstemmelser replikaernes tilstande imellem. Disse uoverensstemmelser bliver senere automatisk løst gennem brugerdefinerede konfliktløsningspolitikker.

Der findes to primære familier af KRDT'er, der adskiller sig fra hinanden ved den anvendte implementationsteknik: *operationsbaserede* og *tilstandsbaserede*. De to familier adskiller sig ved deres replikationsmekanisme og ved, hvordan de realiserer deres konsistenmodel, som typisk er kausal konsistens. Operationsbaserede KRDT'er propagerer *individuelle* lokale opdateringer til andre replikaer og afhænger af en underlæggende kommunikationsprotokol, der garanterer kausal levering. Tilstandsbaserede KRDT'er propagerer *hele* deres tilstand til andre replikaer og garanterer kausal levering ved at tage tilstande fra en semilattice, hvor tilstande forenes ved hjælp af en "join"-operation.

Da semantikken af KRDT'er kompliceres af deres samtidige, distribuerede natur og behovet for konfliktløsning, har der været vidtstrakt interesse for at udvikle formelle teknikker til at specificere KRDT'er. Eksisterende teknikker er dog ikke i stand til at abstrahere over de to typer af KRDT-design. *Det, der mangler, er at kunne betragte KRDT'er som abstrakte datatyper.* Et abstrakt syn på KRDT'er vil både klarlægge deres semantik (*hvad* er en KRDT?), men også gøre det muligt at verificere klientprogrammer modulært op mod en abstrakt KRDT-specifikation, så klienter kan ræsonnere omkring en KRDT uden at vide, hvordan den er implementeret. Dette ville placere KRDT'er på niveau med traditionelle sekventielle datastrukturer, som har forskellige implementationer, men forstås abstrakt via en række af operationer, der er specificeret aksiomatisk.

Denne Ph.D.-afhandling udfylder dette abstraktionshul ved at udvikle en forenet tilgang til at specificere og verificere begge klasser af KRDT'er. Denne forenede tilgang gør brug af separationslogik til at specificere funktionel korrekthed for både operationsbaserede og tilstandsbaserede KRDT'er. Med udgangspunkt i tidligere arbejde, der har vist, at kausalitet kan enkodes i separationlogik, og eksisterende ekspressive specifikationsteknikker, der benytter KRDT "denoteringer", udvikler denne PhD-afhandling KRDT-specifikationer, der nyder godt af en række ønskværdige egenskaber: de er *modulære*, de understøtter verifikationen af hele programmer, der inkluderer *klientkode*, de beskriver realistiske KRDT-implementaioner, og de er formelt verificerede i bevisassistenten Coq.

For at demonstrere at de fremsatte teknikker er mulige at realisere, udvikler denne afhandling en række softwarebiblioteker til at bygge formelt verificerede KRDT'er. Ved hjælp af disse biblioteker verificerer vi en vidstrakt familie af KRDT'er, inklusiv højereordens kombinatorer. Afhandlingen viser også, hvordan vi kan give *den samme* specifikation af to KRDT-implementationer, så et klientprogram kan verificeres op mod en abstrakt specifikation, der er uviden om, hvordan KRDT'en er implementeret.

I et særskilt, men relateret arbejde, viser afhandlingen, hvordan enkodningen af kausalitet i separationslogik, der er nøglen til specificeringen af KRDT'er, også er stærk nok til at vise en række af sessionsgarantier. Disse garantier tjener som ræsonneringsprincipper for klienter af svagt konsistente replikerede datatyper. Derudover er resultatet bevis for, at modellen af kausalitet i separationslogik indfanger de afledte egenskaber af kausal konsistens.

I sin helhed udvikler materialet i denne Ph.D.-afhandling vores evne til at ræsonnere modulært og formelt omkring svagt konsistente replikerede datatyper.

Acknowledgments

Thank you to all the following:

- My advisor, Lars Birkedal, for his strategic leadership and for assembling the exceptional team of kind and smart researchers that work on Aneris. Equally importantly, for saving my skin twice: first when my family and I had to re-locate to Canada mid-PhD, and then when I had to write this dissertation earlier than originally planned. Lars, you have been a class act and it has been a pleasure working together.

- My co-advisor, Amin Timany, who knows more about Aneris and Coq than anyone else I know. Technical discussions with Amin were always illuminating and his tenacity when tackling hard problems is an example I look up to.

- The members of my thesis commitee, Jesper Bengtson and Ilya Sergey, who generously agreed to read this dissertation.

- Léon Gondelman, without whom this dissertation would not have happened. Léon developed the technical fundamentals on which the material in this thesis rests, and directed me towards the study of CRDTs.

- Simon Gregersen, who fielded many a technical question on Iris, Aneris, and Coq, and gracefully agreed to translate the abstract of this dissertation into Danish.

- My co-authors Alban Reynaud and Arnaud Daby-Seesaram, who got up to speed with Aneris in no time and made our CRDT papers possible. Special thanks to Arnaud for writing most of the Coq proofs for one of the chapters of this dissertation.

- My friends Magnus, Daniela, Marit, Thor, Aleš, Zahra, Amin and Léon, who brightened our stay in Denmark.

- My parents, Pedro and Irina, and my brother Alejandro, for their ever-present love and for helping shepherd the kids so I had time to write this dissertation.

- The biggest thank you of all goes to Marianna, who supported me in completing this PhD amidst *two* inter-continental moves, a global pandemic, and two very happy and demanding children. Let's do it again!

Abel Nieto Rodriguez
Waterloo, Ontario, Canada, July 6, 2023

To Marianna, Sam, and Jane

Contents

Part I

Overview

1 Introduction

This dissertation argues that it is possible to formally reason about Conflict-free Replicated Data Types, or *CRDTs*, as Abstract Data Types; that is, we can treat a CRDT as a collection of operations, together with laws that the operations must satisfy, but crucially we need not worry about *how* the operations are implemented. The slogan is thus

Conflict-free Replicated Data Types have Abstract Data Types

Specifically, I show how to specify a wide class of both operation- and state-based CRDTs using modern separation logic. These specifications and their associated proofs have a number of desirable properties:

- They are *modular*: we can specify CRDTs and their clients independently, and then compose their correctness proofs.
- They are *abstract*: a specification can hide implementation details of the CRDT; this allows us to show that an operation-based CRDT and its state-based counterpart both implement the same abstract data type.
- They are *foundational*: the correctness proofs do not rely on any axioms beyond the definition of the operational semantics of the language we implement the CRDTs in.
- They are *machine-verified*: all the proofs are checked in the Coq proof assistant, giving us additional certainty of their correctness.

1.1 Structure of the Dissertation

This dissertation is divided in two parts.

Part I contains two chapters. Chapter 1 is an introduction to causal consistency, CRDTs, abstract data types, and the use of separation logic for verifying distributed systems. Chapter 2 surveys prior work on verification of CRDTs, as well as summarizes the dissertation's content and contributions.

Part II is based on three published papers that I co-authored during my PhD, and consists of three chapters. The first two chapters develop a unifying methodology for the specification and verification of the two types of CRDTs: Chapter 3 deals with *operation-based* CRDTs and Chapter 4 is about their *state-based* counterparts. In reasoning about CRDTs the notion of causal consistency is key. Chapter 5 explores the specification of a causally-consistent distributed database from a *client-centric* perspective, arriving at a set of four *session guarantees.*

1.2 An Informal Asynchronous System Model

Throughout this chapter we work in an asynchronous system model with an unreliable network, which we now informally describe. Our model is a state-transition system where each state describes N *processes*, *replicas*, or *nodes*, as well as the state of the network, which contains all in-flight messages. The number of processes N is fixed before execution starts.

After each transition, the state of zero or more processes can change. Additionally, new messages can be added or removed from the network as they are sent or received. Because the network is *unreliable* messages can be dropped, duplicated, or re-ordered while in-flight.

The system is *asynchronous* because there is no upper bound on how long a message might take before it is delivered, nor on how slow a process might be relative to others [Lyn96]. The asynchronous model can represent *network partitions* by arbitrarily delaying delivery of messages between certain pairs of replicas.

In Part II we will use not an informal model, but the formal operational semantics of AnerisLang, the OCaml-like language of the Aneris separation logic [Kro+20].

1.3 Causality in Distributed Systems

In an asynchronous system model, it is not always possible to tell which of two events (sending a message, mutating a local variable, etc.) happened before the other one. This is because there is no global notion of time that can be used to totally-order events. Instead, we can develop a *partial* order on events, called the *happens-before* relation and denoted with $\rightarrow$ [Lam78]:

- If e is the event corresponding to sending a message, and e' is the event corresponding to receiving said message, then $e \rightarrow e'$.
- If a process executes event e before it executes another event e', then $e \rightarrow e'$.
- We can take the transitive closure of the above two rules: if $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$.

Another name for $\rightarrow$ is *causal order*, and if $e \rightarrow e'$ we say that e is a *causal dependency* of e', or equivalently that e' *causally depends on* e. The notion of causality appears in different parts of this dissertation; two I would like to highlight are *causal broadcast* and *causal consistency*.

Causal Broadcast

Causal broadcast originates with the ISIS project as a kind of operationalization of Lamport's abstract happens-before relation. ISIS provides a set of building blocks for fault-tolerant distributed systems [Bir86]; central to this is a broadcast (one-to-many) communication protocol called CBCAST [BSS91] ("causal broadcast"). CBCAST provides `broadcast` and `deliver` primitives that guarantee that when the latter returns a message M it must have previously returned *all causal dependencies* of M. In other words, CBCAST delivers messages in a way that respects the causal order of events. Chapter 3 presents a verified implementation of causal broadcast.

$$\begin{array}{lll} p_1: & w(x)0 & r(x)1 \\ p_2: & w(x)1 & r(x)0 \\ & \text{(a)} & \end{array} \qquad \begin{array}{lll} p_1: & w(x)0 & w(x)1 \\ p_2: & r(x)1 & r(x)0 \\ & \text{(b)} & \end{array}$$

Figure 1.1: A process history (a) that is causally consistent but not sequentially consistent, and one (b) that is not causally consistent. The notation $w(x)v$ stands for writing value v at memory location x, while $r(x)v$ means reading v from location x.

Causal Consistency

Causal consistency arose in the context of building a shared memory abstraction for distributed systems [Aha+95]. The idea is to allow behaviours ruled out under sequential consistency [Lam79] in the name of scalability. Figure 1.1 shows a process history that is not sequentially consistent but is causally consistent. History (a) is not sequentially consistent because any serialization has to decide whether $w(x)0$ happened before or after $w(x)1$. If we choose the former, then $r(x)0$ must happen after $w(x)0$ but before $w(x)1$, so we end up with e.g. $w(x)0, r(x)0, w(x)1, r(x)1$, but this violates p_2's program order.

In causal consistency, the requirement that the *same* serialization must be consistent with every process' program order is relaxed so that we can produce a serialization for each process, as long as each serialization is consistent with the program order of writes as well as a partial order that connects writes to corresponding reads. In history (a) from Figure 1.1 this *writes-into* order connects $w(x) \mapsto r(x)0$ and $w(x)1 \mapsto r(x)1$. Given that, we can produce the two requisite serializations:

$$w(x)0, w(x)1, r(x)1$$
$$w(x)1, w(x)0, r(x)0$$

By contrast, in history (b) (still Figure 1.1) we are not allowed to reorder the two writes in the serialization of p_2's history, because $w(x)0$ happening before $w(x)1$ is fixed by p_1's program order. We cannot serialize p_2's history as $w(x)0, w(x)1, r(x)1, r(x)0$ because then the second read must return 1 instead of 0.

Zooming out from the technical details, the notion of causal consistent implies four *session guarantees* that provide a more intuitive understanding of what assumptions a program can make about reads and write under this model [Ter+94]. For example, the *monotonic writes* guarantee says that because $w(x)0$ happens before $w(x)1$ in p_1 according to program order, the two writes will be "propagated" in the same order to all other processes. This makes it clear that history (b) is an invalid behaviour under causal consistency so a client program need not guard against it. Chapter 5 deals precisely with showing that the definition of causality we use in this dissertation implies the four session guarantees.

1.4 *Conflict-free Replicated Data Types*

Conflict-free Replicated Data Types are a class of distributed data structures that are highly-available and weakly-consistent. By *highly-available* I mean that a CRDT can continue responding to user requests even in the presence of network partitions, or if one or more replicas are down. By *weakly-consistent* I mean that, unlike in a traditional data structure, two replicas

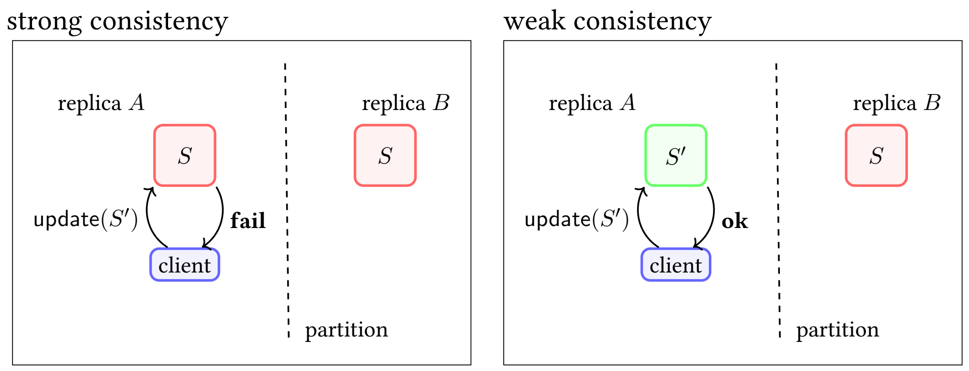


Figure 1.2: Strong vs weak consistency model for a replicated data type.

can concurrently observe the CRDT being in different states. The combination of the above two features is forced by an impossibility result known as the CAP theorem [GL02], which says that any distributed data store can only provide at most two of the following: strong consistency, availability, and partition tolerance.

Figure 1.2 shows how a replicated data type responds to client updates under strong and weak consistency, given a network partition. On the left, a client tries to update replica A's state from S to S'. Because strong consistency requires that the data type behave as if it were not replicated, then replica A must reject the client's update. The system remains strongly consistent, but is no longer available.

By contrast, weak consistency favours availability over agreement, so replica A accepts the client's update. This means that the data type's state has now diverged, because each replica has a different local state. CRDTs adopt this approach. Once the partition heals, we need an (automated) mechanism for resolving conflicts, hence the term "conflict-free".

Since their introduction in Shapiro et al. [Sha+11a], distributed systems practitioners have identified two main classes (or implementation strategies) of CRDTs: *operation-based* (or *op-based*) CRDTs, and *state-based* CRDTs. The distinction between the two kinds has to do with how local updates to a CRDT are propagated to other replicas: op-based CRDTs propagate *individual operations*, while state-based CRDTs propagate the *entire local state*.

1.4.1 Strong Eventual Consistency

Weak consistency is a catch-all term that includes any consistency model weaker than sequential consistency. A more precise and useful consistency model for CRDTs is *strong eventual consistency* (SEC) [Sha+11b], which gives two guarantees:

- *Convergence*: two replicas that have processed the same set of updates must end up in the same state.
- *Eventual delivery*: all updates are eventually delivered to all correct replicas.

Note that the former is a *safety* property ("nothing bad ever happens") and the latter a *liveness* property ("good things eventually happen"). The techniques in this dissertation only prove safety properties, so when we talk about SEC we usually mean just convergence.

SEC does not imply causal consistency. Figure 1.3 shows a replicated register called x that satisfies SEC. The register resolves conflicting writes by taking the write that happened

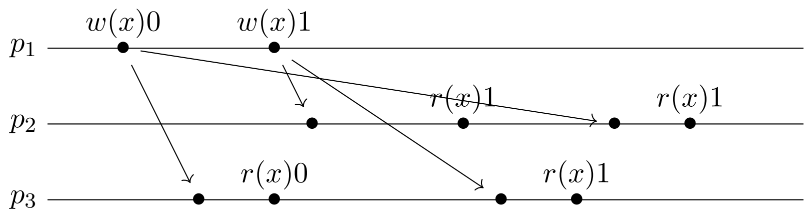


Figure 1.3: SEC does not imply causal consistency.

```
module type GCOUNTER = sig
  val init : addr list -> int -> unit
  val inc : int -> unit
  val read : unit -> int
end

module Client (C: GCOUNTER) = struct
  let test addrs i =
    C.init addrs i;
    C.inc 1;
    C.inc 2;
    let v = C.read () in
    assert (v >= 3)
end
```

Figure 1.4: Interface of a GCounter CRDT and client code.

last according to program order; it is a "last writer wins" register. This history satisfies SEC because whenever a process has seen both writes, they choose $w(x)1$ over $w(x)0$, so that p_2 and p_3's state eventually converge. However, the history is not causally consistent because $w(x)0$ is delivered *after* $w(x)1$ to p_2, violating p_1's program order and monotonic writes.

Nor is SEC implied by causal consistency: history (a) in Figure 1.1 shows an execution that is causally-consistent, yet both process end up in different states. This is because causal consistency imposes no restrictions on the order of concurrent writes, but SEC requires that write conflicts be uniformly resolved across all replicas.

Given the above, the consistency requirements on CRDTs are often extended to imply not only SEC but also causal consistency.

1.4.2 *Example: Grow-Only Counter*

We illustrate the differences between op-based and state-based CRDTs via an example: the *grow-only counter* (GCounter). A GCounter is a simple replicated counter that can only be incremented. Its API, shown in OCaml in Figure 1.4 consists of three functions:

- The client should first call `init` with a list of addresses where the data structure is replicated, as well as the index of the current replica.
- `inc` increments the counter's value by $n \in \mathbb{N}$.

- `read` reads the counter's current value.

Figure 1.4 also shows client code that interacts with the counter, including an assertion about the counter's semantics. Because the counter is monotonic, we know that after executing the two increments its value must be at least 3. The correctness of the assertion requires causal consistency; specifically, it requires that reads observe the effects of previous local writes (this is the "read your writes" session guarantee). This is an example of the kind of reasoning we would like to be able to carry out formally when working with CRDTs.

1.4.3 Operation-Based CRDTs

Op-based CRDTs propagate individual operations using a causal broadcast algorithm. Because causal broadcast can reorder concurrent operations, op-based CRDTs need to be designed so that operations that can happen concurrently are commutative and associative (to guarantee SEC). For some CRDTs these properties hold naturally: e.g. the GCounter has only one operation, addition, which satisfies both requirements. In other cases operations are not naturally commutative. For example, a set-like CRDT might have operations for inserting and removing an element. In such situations, we need to force commutativity by tweaking the CRDT's semantics. For example, we could favour insertions over deletions so that if an element is concurrently inserted and removed the deletion has no effect. This is the so-called *add-wins set*. Dually, we can have deletions prevail over concurrent insertions, so that a concurrent insertion and deletion results in the element being deleted. This is a *remove-wins* set. The bigger point is that CRDT semantics are tricky and some work is required to turn a sequential datatype into a CRDT.

Figure 1.5 sketches out the implementation of a GCounter as an op-based CRDT. The counter's state is initialized to 0. Because the state is mutable and concurrently accessed by more than one thread, state mutations are guarded by a lock. The `read` operation simply retrieves the current value of the counter. The `inc(n)` operation increments the counter by `n` and then broadcasts the increment to all other replicas using a `broadcast` primitive provided by the `Broadcast` module.

In a background thread, the counter executes the `receive` function. The function executes an infinite loop that first checks whether any operations from other replicas need to be processed. It does so using a non-blocking `deliver` primitive that returns an option with the contents of the remote operation, if one exists. If a remote operation has been received, `receive` updates the value of the counter accordingly.

Finally, the `init` function initializes the broadcasting layer with the addresses of participating replicas, as well as the index of the current replica within the addresses array. It also spawns a background thread that executes `receive`.

Requirements on `broadcast` and `deliver`

Two points of note. First, the GCounter easily satisfies SEC because addition is commutative and associative. Second, the GCounter's implementation is simple, but relies on the existence of the `broadcast` and `deliver` primitives. These should satisfy the following requirements:

- Because the network is unreliable, `broadcast` needs to handle dropped messages by resending a message if needed.

```
module OpCounter : GCOUNTER = struct
  let st = ref 0
  let m = Mutex.create ()

  let read () = !st

  let inc n =
    Mutex.lock m;
    st := !st + n;
    Mutex.unlock m;
    Broadcast.broadcast n

  let receive () =
    while (true) do
      let msg = Broadcast.deliver () in
      match msg with
      | None -> ()
      | Some n ->
        lock m;
        st := !st + n;
        unlock m
    done

  let init addrs i =
      Broadcast.init addrs i;
      let _ = Thread.create receive () in ();
end
```

Figure 1.5: Op-based GCounter.

- Dually, `deliver` needs to prevent re-delivery of a duplicate message. There needs to be a way of uniquely identifying a message.
- To guarantee causal consistency, `deliver` needs to return messages in causal order: that is, a message can only be delivered once all its causal dependencies have been delivered.

In other words, the correct implementation of op-based CRDTs relies on the existence of a causal broadcast algorithm in the style of CBCAST.

1.4.4 State-Based CRDTs

State-based CRDTs do not assume that the underlying communication protocol implements causal broadcast. They instead guarantee SEC by representing the data structure's state as an element of a *join semi-lattice*. A join semi-lattice (which I will sometimes refer to just as a *lattice*) is a poset $(S, \leq, \sqcup)$ equipped with a join (or least upper bound) operator $\sqcup$ such that $x \sqcup y \in S$ is defined for all x and y in S, satisfying the following:

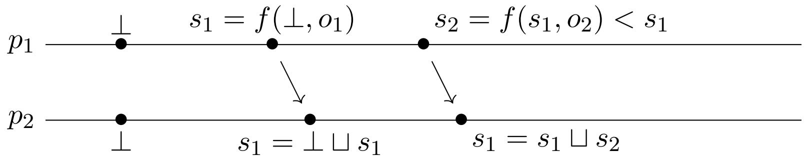


Figure 1.6: A non-monotonic mutator leads to lost updates.

- $x \sqcup y$ is an upper bound: $x \leq x \sqcup y$ and $y \leq x \sqcup y$
- $x \sqcup y$ is the least upper bound:

$$\forall u \in S, x \leq u \wedge y \leq u \implies x \sqcup y \leq u$$

The following properties of $\sqcup$ are crucial for state-based CRDTs:

- Commutativity: $x \sqcup y = y \sqcup x$.
- Associativity: $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$.
- Impotence: $x \sqcup y \sqcup y = x \sqcup y$.

To define a state-based CRDT we then start with a lattice L that can encode the desired semantics. The CRDT's initial state is some value $v \in L$: often the initial value is $\perp$, if L has a minimum.

We define a *merge* function that updates the local state with the contents of remote states. The merge function is just the join operator of the underlying lattice. Commutativity and associativity of joins ensure that given a sequence of states (updates) $S = s_1, \ldots, s_n$, merging any permutation of S yields the same resulting state. This means the network can re-order messages sent by different replicas while maintaining convergence. Impotence means that a replica can merge the same remote state multiple times without effect, so the network is allowed to duplicate messages.

Finally, we define a *mutator* function $f : L \to O \to L$, where O is the type of supported operations. The mutator is used to update the local state before the latter is propagated to other replicas. The mutator should be *monotonic*: $\forall l \in L, o \in O, l \leq f(l, o)$. Monotonicity of the mutator is needed because otherwise local updates are lost during merges. Figure 1.6 shows a state-based CRDT replicated over two processes. The initial state is $\perp$ for both processes. The first process performs two local updates, but because f is not monotonic we have $s_2 < s_1$. The states s_1 and s_2 are propagated to p_2: s_1 is successfully propagated because $\perp \leq s_1$, but when we try to propagate s_2 we run into issues because $s_2 < s_1$ implies $s_1 \sqcup s_2 = s_1$. Now the two replicas have diverged, and p_1's second update is lost. If f were monotonic, we would have $s_1 \leq s_2$, $s_1 \sqcup s_2 = s_2$ and the second update is correctly propagated.

State-Based GCounter

Figure 1.7 shows an OCaml implementation of the state-based GCounter. The main point is that both the op-based and state-based GCounter satisfy the same interface (`COUNTER` signature), so client code such as in Figure 1.4 can use a GCounter without knowing whether it is op-based or state-based.

```
module StateCounter : GCOUNTER = struct
  let st = ref []
  let m = Mutex.create ()
  let addrs = ref []
  let i = ref 0

  let read () =
    Mutex.lock m;
    let r = List.fold_left (+) 0 !st in
    Mutex.unlock m;
    r

  let inc n =
    Mutex.lock m;
    st := List.mapi (fun j x -> if j = !i then x + n else x) !st;
    Mutex.unlock m

  let receive_thread () =
    while (true) do
      let msg = receive () in
      match msg with
      | None -> ()
      | Some (vs) ->
        Mutex.lock(m);
        st := List.map2 Int.max !st vs;
        Mutex.unlock(m)
    done

  let send_thread () =
    while (true) do
      List.iter (fun addr -> send addr !st) !addrs
    done

  let init addrs' i' =
    i := i';
    addrs := addrs';
    st := List.init (List.length !addrs) (fun _ -> 0);
    let _ = Thread.create receive_thread () in
    let _ = Thread.create send_thread () in
    ()
end
```

Figure 1.7: State-based GCounter.

In this implementation the GCounter's state is not a number, but a list of of N entries tracking the contributions of every replica to the value of the counter, where N is the number of replicas. Joins in this lattice consist of taking pointwise maximums. The initial state is the list where all entries are 0. The mutator increments the counter's value by incrementing the entry corresponding to the local replica.

There are now two background threads: as before the `receive_thread` incorporates changes sent from other replicas, but instead of executing individual operations `receive_thread` now merges the remote state with the current one. The second background thread is `send_thread`, which repeatedly messages the current local state to other replicas for merging. In a realistic implementation, `send_thread` would take care to not spam other replicas, using for example acknowledgements to avoid sending out the same state multiple times.

1.4.5 Trade-offs

The decision of whether to use an op-based or a state-based design incurs a number of trade-offs. Op-based CRDTs replicate by messaging individual operations. This is attractive if the CRDT state is large but operations can be represented compactly, for example in text editing applications [Sha+11a]. The dual situation also occurs: if the state is small it might be more efficient to batch updates by merging a remote state, as opposed to executing individual operations. Op-based designs require that the underlying communication protocol guarantee causal delivery, so they might not be an option in situations where e.g. causal broadcast is not available. By contrast, state-based CRDTs are compatible with an unreliable communication protocol that can re-order, delete, and duplicate messages. Because causal delivery of operations is a strong guarantee, op-based CRDTs often have simpler designs than their state-based counterparts, where one needs to construct a lattice that encodes the right datatype semantics. There is work on coalescing the advantages of both designs: *δ-CRDTs* [ASB18] are lattice-based (so work well with unreliable networks) but replicate by sharing only small fragments of their state.

1.4.6 Equivalence of CRDT Designs

Since their introduction in Shapiro et al. [Sha+11a] it has been clear that op-based and state-based CRDTs are "equivalent", with Shapiro et al. [Sha+11a] remarking:

> Interestingly it is always possible to emulate a state-based object using the operation-based approach, and vice-versa.

To emulate the state-based approach using an op-based CRDT, a replica A broadcasts an "operation" `merge(st)`, where `st` is A's current state. Upon receipt, remote replicas execute the operation by merging `st` using the lattice's join, as in the state-based case.

To emulate the op-based approach using a state-based CRDT, define the lattice to be

$$(\mathcal{P}(\mathsf{Op} \times \mathsf{Ts}), \subseteq, \cup)$$

That is, a lattice element is a set of pairs (o, t) where o is an operation of the op-based CRDT and t is a *timestamp* that is unique, totally-ordered and consistent with causality. Such a timestamp can be generated at every replica using a local counter and a unique replica identifier (e.g. an IP address) and is known as a *Lamport timestamp* [Lam78]. The timestamp's role is twofold: to prevent an operation from being delivered twice (both copies will have the same timestamp),

```
module type S = sig
  type elt (* element type *)
  type t   (* type of sets *)
  val empty : t
  val mem : elt -> t -> bool
  val add : elt -> t -> t
  val remove : elt -> t ->
end
```

Figure 1.8: Set ADT in OCaml, abridged.

as well as to establish an order between delivered operations that is consistent with causality (to emulate causal delivery). The lattice's partial order is subset inclusion, and join is set union.

One can think of the above design as a "lazy" version of the op-based CRDT, where operations are not immediately applied upon receipt, but instead are kept around so that causal order can be preserved. Once we need to query the CRDT, we can re-run the operations in timestamp order to compute a resulting state. This design is further described in the *pure op-based CRDT* approach of Baquero et al. [BAS14].

The emulation-based argument for equivalence of CRDTs is enlightening but ultimately is lacking in explanatory power because we seem to be saying "these two things are the same abstract thing", *without saying what the abstract thing is.*

A related problem is that because we do not have abstract specifications, we focus on guaranteeing eventual consistency, which is a crucial safety property but only *part* of functional correctness (does the data structure produce the right output for the given input).

1.5 Abstract Data Types

A different perspective on the equivalence of both kinds of CRDTs comes from the notion of *abstract data type* (ADT), given by Liskov and Zilles [LZ74]:

> An abstract data type defines a class of abstract objects which is completely characterized by the operations available on those objects. This means that an abstract data type can be defined by defining the characterizing operations for that type.

ADTs are ubiquitous in the standard libraries of modern programming languages; in OCaml they are represented using module system signatures. Figure 1.8 shows an example set ADT, abridge from OCaml's standard library. The ADT declares a type for elements and sets of elements, as well as operations on sets (e.g. add an element, remove an element, etc.). In addition to describing the set of available operations, an ADT also describes how to use them. For example, the set ADT might specify that removing an element that is not present in the set is a no-op, or alternatively that it might trigger an exception.

Figure 1.4 shows part of the definition of the GCounter ADT. I say "part of" because we are missing the specifications for the different operations. This example suggests that it might be possible to show "equivalence" of the two kinds of CRDTs by proving that they can both implement the same ADTs.

By expressing CRDTs as ADTs we benefit in at least two ways:

- We end up with implementations *and* correctness proofs that are more modular. We can write client code that uses a CRDT without knowing how it is implemented. This means we can swap out the implementation at a later point for e.g. optimization purposes, without modifying the client. Additionally, we can prove the client correct against an abstract specification that is satisfied by both kinds of CRDTs. Then if we do swap out the CRDT's implementation we do not need to modify the client's proof either.

- We can eschew SEC as the main correctness criterion, instead adopting full functional correctness (of which SEC is part). The specifications for the ADT operations are more abstract in that they hide away implementation details. Because the specifications are more abstract, it is easier to say whether they capture properties of interest. For example, a specification for a state-based CRDT might talk about lattices and least upper bounds, but these are implementation details that obscure what data structure we are implementing (e.g. a counter). If we can write more abstract specifications for the GCounter ADT and then show that the state-based counter satisfies those specifications, then that gives us confidence that the state-based implementation is correct.

1.6 Separation Logic for Distributed Systems

To specify and verify that the CRDT implementations in this dissertation meet their desired safety properties my main tool is the Aneris separation logic [Kro+20]. Aneris is a higher-order, concurrent, and distributed separation logic built using the Iris program logic framework [Jun+18]. Because Aneris is higher-order, Hoare triples themselves are propositions, so we can specify higher-order functions like filter and map. By *concurrent* I mean that Aneris can reason about multiple threads concurrently executing in the same node and interacting via shared memory. By *distributed* I mean that Aneris can reason about multiple nodes that execute concurrently and communicate solely via message passing. More generally, the requirement to reason about message passing, as opposed to just shared memory, is what sets distributed algorithms apart from single-node concurrent algorithms.

This section assumes basic familiarity with separation logic at the level of Reynolds [Rey02] and sketches out how Aneris can be used to reason about distributed systems. The Aneris project was started in Krogh-Jespersen et al. [Kro+20] and since then has been used to verify a variety of distributed algorithms: a load balancer, two-phase commit, a replicated log, a causally-consistent key-value store, single-decree Paxos, as well as the family of CRDTs described in this dissertation [Kro+20; Tim+21; Gon+21; Nie+22; Nie+23]. A separate line of work has extended Aneris to establish not only partial correctness but also a specific kind of simulation of transition systems that can be used to obtain liveness results, in addition to safety results [Tim+21]. Finally, because Aneris is an instantiation of the Iris program logic framework, Aneris proofs can be mechanized in Coq using MoSeL [Kre+18a], a Coq framework for developing interactive proofs using separation logic. MoSeL provides built-in tactics as well as general Coq infrastructure (e.g. pretty-printing of the separation logic proof context) that make writing Aneris proofs very similar to writing regular Coq proofs. As a consequence, all the safety proofs in this dissertation are mechanized in Coq in a *foundational* style: the program logic rules are proven (in Coq) safe against the operational semantics of Aneris, which is itself mechanized. There are no assumptions that enlarge the trusted code base beyond those embedded in the operational semantics, as well as a few standard classical axioms (functional extensionality, propositional extensionality, and choice).

$$
\begin{aligned}
z &\in \mathbb{Z} \\
s &\in \mathit{String} \\
x \in Var,\ sh \in \mathit{Handle},\ \ell &\in \mathit{Loc} \triangleq \text{(infinite countable set)} \\
l \in \mathit{BaseLit} &::= z \mid \texttt{true} \mid \texttt{false} \mid () \mid \ell \mid s \mid z \\
v \in \mathit{Val} &::= l \mid \texttt{rec}\ f\ x = e \mid (v_1, v_2) \mid \texttt{inl}\ v \mid \texttt{inr}\ v \\
\circledcirc_1 &::= \ \sim\ \mid\ -\ \mid \texttt{i2s} \mid \texttt{s2i} \mid \texttt{len} \\
\circledcirc_2 &::= \ +\ \mid\ -\ \mid\ *\ \mid \texttt{quot} \mid \texttt{rem} \mid \& \mid | \mid \hat{}\ \mid \\
&\quad\ << \ \mid\ >> \ \mid\ \leq\ \mid\ <\ \mid\ =\ \mid\ ++ \\
e \in \mathit{Expr} &::= v \mid x \mid e_1\ e_2 \mid\ \circledcirc_1 e \mid e_1 \circledcirc_2 e_2 \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \\
&\quad \mid \texttt{find}\ e_1\ e_2\ e_3 \mid \texttt{substring}\ e_1\ e_2\ e_3 \mid \texttt{rand}\ e \mid (e_1, e_2) \mid \texttt{fst}\ e \mid \texttt{snd}\ e \\
&\quad \mid \texttt{inl}\ e \mid \texttt{inr}\ e \mid \texttt{match}\ e\ \texttt{with}\ \texttt{inl}\ x_1 \Rightarrow e_1 \mid \texttt{inr}\ x_2 \Rightarrow e_2\ \texttt{end} \\
&\quad \mid \texttt{fork}\ \{e\} \mid \texttt{ref}\ \ e \mid\ \texttt{!}\ e \mid e_1 \texttt{:=} e_2 \mid \texttt{CAS}\ e_1\ e_2\ e_3 \\
&\quad \mid \texttt{makeaddress}\ e_1\ e_2 \mid \texttt{getaddress}\ e \mid \texttt{socket}\ \mid \texttt{socketbind}\ e_1\ e_2 \\
&\quad \mid \texttt{sendto}\ e_1\ e_2\ e_3 \mid \texttt{receivefrom}\ e \mid \texttt{settimeout}\ e_1\ e_2\ e_3 \mid \texttt{start}\ l\ e
\end{aligned}
$$

Figure 1.9: Abstract syntax of AnerisLang [Tea23].

1.6.1 The Language of Aneris

Aneris proofs reason about programs written in AnerisLang, a programming language with support for higher-order functions, mutation, concurrency and message-passing, among other features. AnerisLang can be thought of as a small, but expressive, untyped subset of OCaml. In particular, it is eagerly evaluated.

Figure 1.9 shows the abstract syntax of AnerisLang. I would like to highlight a few of the constructs of the language: as in OCaml `ref e` creates a mutable reference initialized to the value of `e`; a reference can be de-referenced via `!e` and assigned to via `e := e'`. AnerisLang supports intra-node concurrency via the `fork e` expression, which evaluates `e` in a new thread that runs concurrently with the originating one. Message passing is done via the `sendto` and `receivefrom` primitives, both of which can be configured to be blocking or non-blocking. These two mimic a UDP-style, connectionless communication channel that is unreliable, where messages can be lost, ordered, or duplicated. `sendto skt msg dest` sends the message `msg` on socket `skt` to the socket address `dest`; a socket address is an (IP address, port number) pair. `receivefrom skt` returns the next message buffered at socket `skt` if one exists, or nothing if the socket is empty. Finally, the `start ip e` expression starts a new node running at IP address `ip` that evaluates expression `e`. Said node runs concurrently with the current node; as a result, there are two forms on concurrency: intra-node concurrency of multiple threads running on the same node, and inter-node concurrency of multiple node nodes executing at different IP addresses. In the operational semantics of Aneris both forms of concurrency are represented uniformly: the semantics keeps a list of all running threads as a list of (IP address, expression) pairs. The `start` expression can only be run in a privileged "system node": a typical setup for AnerisLang programs is to have the system node set up the constituents of the distributed system (via `start`) but perform no other steps.

```
Definition ping : val := fun: "addr" "server",
  let: "skt" := NewSocket ... in
  SocketBind "skt" "addr";;
  SendTo "skt" #"PING" "server";;
  let: "msg" := unSOME (ReceiveFrom "skt") in
  assert: ((Fst "msg") = #"PONG").

Definition pong : val := fun: "addr",
  let: "skt" := NewSocket ... in
  SocketBind "skt" "addr";;
  let: "msg" := unSOME (ReceiveFrom "skt") in
  let: "sender" := Snd "msg" in
  assert: ((Fst "msg") = #"PING");;
  SendTo "skt" #"PONG" "sender".

Definition ping_pong_runner : expr :=
  let: "pongaddr" := MakeAddress #"0.0.0.0" #80 in
  let: "pingaddr" := MakeAddress #"0.0.0.1" #80 in
  Start "0.0.0.0" (pong "pongaddr") ;;
  Start "0.0.0.1" (ping "pingaddr" "pongaddr").
```

Figure 1.10: Ping-pong example in AnerisLang, adapted from the Aneris Coq formalization [Teab].

An Example Program

Figure 1.10 shows an AnerisLang program that illustrates these features in action and serves as the "hello world" example for AnerisLang. The example shows two nodes: one messages the other with "ping" and the other replies with "pong".

The `ping` function takes as arguments the socket addresses of the two sockets through which the nodes communicate. It then creates a new socket and binds it to the given local socket address. The socket is set to be *blocking*, meaning that `receivefrom` waits until a message is available before returning. The `ping` function then sends the message `PING` to the remote address. It then waits for a response and asserts that the response is `PONG`.

The `pong` function is the dual of `ping`. It waits (and asserts) for a `PING` message and then replies with `PONG`.

Finally, the `ping_pong_runner` (the system node) function creates two socket addresses at different IP addresses using port 80. It then starts the two nodes running `ping` and `pong` respectively. Notice that the system node then terminates, but evaluation continues because the two spawned nodes are still running.

Errors in AnerisLang are modelled by having the semantics get stuck on an expression. Specifically, the expression `assert false` is stuck. Verifying the ping-pong example consists of proving, using Aneris' program logic rules, that evaluation of the `ping_pong_runner` expression never gets stuck (i.e. that the assertions never fail).

Ht-fork

$$\frac{\{P\}\ \langle n;\ e\rangle\ \{w.\,\mathsf{True}\}}{\{P\}\ \langle n;\ \texttt{fork}\ \{e\}\rangle\ \{v.\,v = ()\}}$$

Ht-Protocol

$$\frac{e \notin \mathit{Val} \qquad \{P * a \Mapsto \psi\}\ \langle n;\ e\rangle\ \{v.\,Q\}}{\{P * \mathsf{Unallocated}(\{a\})\}\ \langle n;\ e\rangle\ \{v.\,Q\}}$$

Ht-start

$$\frac{\{P * \mathsf{FreePorts}(n, \mathcal{P})\}\ \langle n;\ e\rangle\ \{w.\,\mathsf{True}\}}{\{P * \mathsf{FreeIp}(n)\}\ \langle \mathfrak{S};\ \texttt{start}\ n\ e\rangle\ \{v.\,v = ()\}}$$

Ht-newsocket

$$\{\mathsf{True}\}\ \langle n;\ \texttt{socket}\ ()\rangle\ \{z.\, z \overset{n}{\hookrightarrow} (\mathsf{None}, \texttt{true})\}$$

Ht-send

$$\left\{\begin{array}{l}(n,p) = m.\mathsf{orig} * z \overset{n}{\hookrightarrow} (\mathsf{Some}\ m.\mathsf{orig}, b) * \\ m.\mathsf{orig} \rightsquigarrow (R, T) * m.\mathsf{dest} \Mapsto \Phi\ * \Phi(m)\end{array}\right\}$$

$$\langle n;\ \texttt{sendto}\ z\ m.\mathsf{body}\ m.\mathsf{dest}\rangle$$

$$\left\{\begin{array}{l}v.\,v = |m.\mathsf{body}| * z \overset{n}{\hookrightarrow} (\mathsf{Some}\ m.\mathsf{orig}, b) * \\ m.\mathsf{orig} \rightsquigarrow (R, T \cup \{m\})\end{array}\right\}$$

Ht-send-duplicate

$$\left\{\begin{array}{l}(n,p) = m.\mathsf{orig} * m \in T * z \overset{n}{\hookrightarrow} (\mathsf{Some}\ m.\mathsf{orig}, b) * \\ \qquad\qquad m.\mathsf{orig} \rightsquigarrow (R, T) * m.\mathsf{dest} \Mapsto \Phi\end{array}\right\}$$

$$\langle n;\ \texttt{sendto}\ z\ m.\mathsf{body}\ m.\mathsf{dest}\rangle$$

$$\left\{\begin{array}{l}v.\,v = |m.\mathsf{body}| * z \overset{n}{\hookrightarrow} (\mathsf{Some}\ m.\mathsf{orig}, b) * \\ m.\mathsf{orig} \rightsquigarrow (R, T)\end{array}\right\}$$

Ht-recv-non-blocking

$$\left\{\begin{array}{l}z \overset{n}{\hookrightarrow} (\mathsf{Some}\ (n,p), \mathsf{False}) * (n,p) \rightsquigarrow (R, T) * \\ (n,p) \Mapsto \Phi\end{array}\right\}$$

$$\langle n;\ \texttt{receivefrom}\ z\rangle$$

$$\left\{\begin{array}{l}v.\ \Big(v = \mathsf{None} * z \overset{n}{\hookrightarrow} (\mathsf{Some}\ (n,p), \mathsf{False}) * \\ \quad (n,p) \rightsquigarrow (R, T)\Big) \vee \\ \quad \Big(\exists m.\ m.\mathsf{dest} = (n,p) * v = \mathsf{Some}\,(m.\mathsf{body}, m.\mathsf{orig}) * \\ \quad \big((m \notin R * z \overset{n}{\hookrightarrow} (\mathsf{Some}\ (n,p), \mathsf{False}) * \\ \quad (n,p) \rightsquigarrow (R \cup \{m\}, T) * \Phi(m)\big)\ \vee \big(m \in R * \\ \quad z \overset{n}{\hookrightarrow} (\mathsf{Some}\ (n,p), \mathsf{False}) * (n,p) \rightsquigarrow (R, T)\big)\Big)\end{array}\right\}$$

Figure 1.11: Selected program logic rules of Aneris, adapted from the Aneris documentation [Tea23].

1.6.2 *Program Logic Rules*

We next take a look at a selection of Aneris program logic rules that reason about concurrency and message passing. The main judgment in Aneris is the Hoare triple $\{P\}\ \langle n;\ e\rangle\ \{v.\,Q\}$. As usual, this is a partial correctness assertion saying that if P holds, then e can execute at IP address n without getting stuck (without runtime errors) and, if e terminates, then Q holds. Notice that e is executed in the context of a specific IP address, and that v is bound in Q.

The Ht-fork says that to prove $\langle n;\ \texttt{fork}\ \{e\}\rangle$ safe it suffices to prove $\langle n;\ e\rangle$ safe at the same IP address and with the same precondition P.

The rule for starting a new node (as opposed to a new thread within an existing node) is similar. First, note that we can only prove safety of $\texttt{start}\ n\ e$ when the expression executes from the system node $\mathfrak{S}$. Additionally, we need knowledge of the proposition $\mathsf{FreeIp}(n)$ which says that no other node has been previously started at IP n. In exchange for "giving in" the $\mathsf{FreeIp}(n)$ resource, we get back a $\mathsf{FreePorts}(n, \mathcal{P})$ resource saying that all ports in $\mathcal{P}$ are free at IP n (ports are identified with natural numbers). A port is free if no socket is bound to it. To summarize, the rule for starting a node and forking a thread are virtually the same, except that nodes have unique identifiers (IP addresses) and state (free ports), so additional resources are needed to track those.

Allocating a socket with $\langle n;\ \texttt{socket}\ ()\rangle$ returns a socket handle z, as well as the resource $z \overset{n}{\hookrightarrow} (\mathsf{None}, \texttt{true})$ indicating that the handle is not yet bound to a port and that the socket is blocking. It is also possible to create non-blocking sockets.

The HT-Protocol rule introduces one of the key concepts of Aneris: *socket protocols*. A socket protocol is a predicate that all messages incoming at a given socket address must satisfy. A given socket address a is either *unallocated*, represented by the proposition $\mathsf{Unallocated}(a)$, or has an associated socket protocol, modelled via the proposition $a \Mapsto \psi$. This rule allows us to "allocate" a socket protocol provided that the address is previously unallocated. Notice that the allocation is purely logical: effectively we are saying that from this point in the proof onward, we guarantee that all messages *sent* to a satisfy ψ. There is also a dual implication, *receiving* from a socket bound at address a must yield a message that satisfies ψ (or a duplicate message, as we will see below). The restriction that e not be a value is technical in nature and has to do with the definition of weakest precondition (in terms of which Hoare triples are defined) in Aneris.

Recall that in Aneris messages can be lost in transit. This means that a node might want to re-send an old message until receipt is confirmed by the destination. We have seen that when sending a message we must satisfy the destination's socket protocol, but socket protocols are Aneris propositions so they can demand not only knowledge of but also *ownership* over a logical resource. For example, the $\mathsf{FreeIp}(n)$ resource is *non-duplicable*: it is *not* the case that $\mathsf{FreeIp}(n) \vdash \mathsf{FreeIp}(n) * \mathsf{FreeIp}(n)$. If a socket protocol requires $\mathsf{FreeIp}(n)$ we must give up ownership of the resource when sending the message. Ownership of the resource is conceptually transferred to the network and, ultimately, to the recipient once they call `receive`. This motivates the need for two different rules for sending a message, depending on whether the message is a duplicate or not. In the latter case, when sending message m we must prove $\Phi(m)$ for a destination with socket protocol Φ. However, if the message is a duplicate we do not need to re-prove Φ, since Φ could involve non-duplicable resources.

The Ht-Send rule reasons about non-duplicate messages. Notice that the precondition requires that the socket by bound to a port, and as mentioned we must satisfy the recipient's socket protocol (in particular, this means we must know what their socket protocol is). Finally, we have the *message history* resource $m.\mathsf{orig} \rightsquigarrow (R, T)$. This resource tracks the sets R and T of received and transmitted messages at address $m.\mathsf{orig}$. After sending a message m we obtain back the updated history $m.\mathsf{orig} \rightsquigarrow (R, T \cup \{m\})$.

The Ht-Send-Duplicate rule does not take the socket protocol $\Phi(m)$ in the precondition, but does require that we prove that the message m was previously sent: $m \in T$, with $m.\mathsf{orig} \rightsquigarrow (R, T)$.

Finally, we have Ht-Recv-Non-Blocking, the non-blocking version of the receive rule. In order to verify $\langle n;\ \texttt{receivefrom}\ z\rangle$ we must know that the socket is bound to an address (n, p) and that it is non-blocking, as well as the message history (R, T) and socket protocol Φ for that address. The postcondition is a disjunction with three cases. One possibility is that the socket's buffer is empty, in which case `receivefrom` z returns `None` and the message history is unchanged. Another option is that receive returns a *new* message that has not been previously received. In this case, we learn that $m \notin R$, $m.\mathsf{orig} \rightsquigarrow (R \cup \{m\}, T)$ and crucially $\Phi(m)$. That is, in this case the socket protocol holds for the received message. Finally, it could be that the message was duplicated by the network, so the returned message was already in R. In this case, the message history is unchanged and we do not learn that socket protocol holds.

The rules above are but a small part of Aneris. We direct the reader to Jung et al. [Jun+18] and Birkedal and Bizjak [BB17] for an introduction to Iris, and to Krogh-Jespersen et al. [Kro+20] and the Aneris documentation [Tea23] for further discussion of Aneris. The papers Krebbers et al. [KTB17], Krebbers et al. [Kre+18b] and the Aneris website [Teaa] are starting points for working with Aneris' Coq formalization. Part II describes additional Aneris features that are relevant to formalization of CRDTs.

2 Contributions

This chapter outlines the state of the art in formal verification of CRDTs and, against that backdrop, explains this dissertation's contributions.

2.1 Landscape of CRDT Verification

A comprehensive, up-to-date bibliography on CRDTs, including their specification and verification, can be found at `https://crdt.tech/papers.html`. Examining it suggests several themes:

1. *CRDTs are amenable to machine-checked verification.* Some of the early works on formalizing CRDTs relied on pen-and-paper proofs [Bur+14; Got+16; Att+16], but more recent projects virtually all employ some form of machine-checked verification [ZBP14; Li+14; Gom+17; Liu+20; NPS20; DFG22; Lad+22; ZWS23]. I speculate that CRDTs make an attractive target for formal methods due to their self-contained nature: research prototypes of CRDTs are small (up to hundreds of lines of code) and some of the properties of interest, notably convergence, the algebraic properties of state-based CRDTs, and commutativity of operations in op-based CRDTs, apply not just to one data structure but entire classes of them, making it so their formalization "pays off".

2. *Formal methods practitioners have deployed a wide array of techniques towards formalizing CRDTs.* These include program logics [LF21; Tim+21], static analysis [Li+14], transition-system models [ZBP14; Bur+14; NPS20], bespoke programming languages [DFG22; ZWS23], type systems [ZWS23], as well as program synthesis [Lad+22]. As usual, there is often tension between the expressiveness and automation level of a given technique. Fully-automated techniques tend to verify specific properties (e.g. convergence) [ZWS23; DFG22], while those are that are more interactive or require user annotations can accommodate more general safety invariants [NPS20]. The use of SMT solvers is ubiquitous among both kinds of techniques [NPS20; Liu+20; Liu+20; Sou+22; Lad+22; DFG22].

3. *There is no agreement on what it means for a CRDT to be correct and, therefore, on what one ought to prove to verify one.* In other words, the set of verified properties varies across different techniques. There are roughly three possible options:

    - The paper introducing CRDTs [Sha+11a] proposes *strong eventual consistency* as the defining correctness property for CRDTs, and some verification efforts have focused on proving SEC [ZWS23; DFG22; Liu+20; Gom+17]. More specifically, they prove *convergence*, the safety component of SEC. With the exception of the Trillium logic [Tim+21], which has one small case study on CRDTs, none of the works I am familiar with try to prove eventual delivery or other liveness properties. SEC is a fine property that emerges naturally from relaxing strong consistency: we go from "replicas agree on state all the time" to "replicas eventually agree on state". But it does not tell the whole story: for example, we can "implement" a replicated grow-only counter so that every increment increases the value of the counter by 2 instead of by 1. This does not violate SEC, but neither does it capture the counter's intended semantics.

- This suggests that we care about is not just SEC but *functional correctness*. Functional correctness is specifying what the output of an operation on a data structure should be given its inputs. The word *input* here should be taken to include the data structure's state as well as its prior history of operations.

  The seminal work in this camp is Burckhardt et al. [Bur+14]. They observe that sequential data types can be specified with a function that maps lists of operations to the resulting data type state. They generalize this to the CRDT setting by relaxing the requirement that operations be sequentially sorted. Instead of a list of operations, we start with a *set* of operations together with causality metadata. That is, we specify CRDTs via partial functions from DAGs to their resulting data type state. The edges in the DAG describe the happens-before relation between CRDT operations. Notice this specification style gives us converge "for free", because two replicas that have observed the same set of operations necessarily end up, as per the specification (a function), in the same resulting state. Follow up works that target functional correctness and use this specification style include Zeller et al. [ZBP14], Kleppmann et al. [Kle+18], and Leijnse et al. [LAB19].

- The final kind of verification task that appears in the literature is *invariant preservation*. This appears in the context of *hybrid* CRDT designs where some operations require coordination while others do not. For example, a CRDT modelling an auction service needs to guarantee that after the auction is closed no replica can continue bidding. The associated invariant would be "at most one replica is declared as winner of the auction". Another way to think about these hybrid designs is as CRDTs where some operations are disabled some of the time. Works that focus on proving invariant preservation include Li et al. [Li+14], Gotsman et al. [Got+16], and Nair et al. [NPS20].

4. *There is variance in how "realistic" the verified CRDTs are.* In other words, some verification attempts target *models* of CRDTs [Bur+14; ZBP14; Got+16; NPS20; LF21], while others verify runnable code that is either extracted from a theorem prover or directly written and verified in an executable language [Gom+17; DFG22; ZWS23; Li+14; Liu+20].

5. *Verification techniques mostly specialize in one class of CRDT, targeting either op-based [Gom+17; Kle+18; LF21] or state-based [ZBP14; Liu+20; NPS20; Lad+22; ZWS23] designs, but not both.* Sometimes papers express a desire to generalize their technique to the other kind of CRDT, but to-date this has proven to be an elusive goal. The one notable exception is Burckhardt et al. [Bur+14], whose methodology can specify both kinds of CRDTs, but it should be noted that this paper operates at the level of a highly-abstracted model, where the differences between both kinds of CRDTs are less clear.

6. *Not many techniques support verification of clients of CRDTs.* As every software component, in the real world CRDTs do not operate in isolation and work together with other components. One natural question is whether the verification technique used to certify that the CRDT is e.g. causally-consistent extend to proving correctness of users or clients of the CRDT. For example, given a verified GCounter, can we prove that a client that increments the GCounter twice and then queries will read a value of at least 2? A follow-up question is whether the client can be verified against the CRDT's *specification* (as opposed to against the CRDT's implementation). This is desirable because it allows us to modify the CRDT's correctness proof without changing the client's proof, as long as the specification remains constant. In other words, the question is whether the verification technique produces correctness proofs that are *modular*.

   Liang and Feng [LF21] develop the first and only modular verification technique for CRDTs (with the exception of the work described in this dissertation). They propose *Abstract Converging Consistency* (ACC) as a correctness criterion for CRDTs. ACC is as an extension of SEC that implies not only convergence but also functional correctness. It does so by relating a concrete implementation of an op-based CRDT to an abstract specification whose operational semantics allow for re-ordering of operations, and is therefore less realistic but more obviously correct. They show that ACC directly implies convergence, as well as prove an abstraction theorem that shows that ACC implies contextual refinement of the abstract CRDT by its implementation. Finally, they present a rely-guarantee style logic for verification of clients against the abstract CRDT model. Composing their techniques gives an end-to-end safety proof of clients and CRDT implementations.

2.2 *Overview of the Dissertation*

The main body of the dissertation in Part II is thematically divided in two: *specification and verification of op-based and state-based CRDTs* in Chapters 3 and 4, and *session guarantees for client-centric consistency* in Chapter 5.

2.2.1 *Verification of CRDTs*

This dissertation presents a unified approach to specifying and verifying op-based and state-based CRDTs using separation logic. By *unified approach* I mean two different things. First, we can specify both kinds of CRDTs using the same techniques: an extension of the *encoding of causality* in separation logic developed by Gondelman et al. [Gon+21], specifications based on CRDT *denotations* [Bur+14], and a modular proof technique that can transform purely-functional, sequential implementations of data structures into fully-fledged CRDTs. Second, not only can we specify both kinds of CRDTs, but we can also give an op-based CRDT and its state-based counterpart the *same specification.* In other words, I show that even though op-based and state-based CRDTs use wildly different implementation techniques (and specifically, they achieve causal consistency through two very different mechanisms), we can abstract away from these implementation details. As explained earlier in this introduction

Conflict-Free Replicated Data Types have Abstract Data Types

Looking at the rubric in Section 2.1, the work in this dissertation

1. *Is machine checked* using the Coq theorem prover. Furthermore, all the results in Part II are *foundational*: that is, they depend only on the "foundations of mathematical logic, without additional assumptions and axioms" [App01]. In particular, unlike other works which *assume* causal broadcast as part of their network model when verifying op-based CRDTs [Gom+17], I implement and verify in Chapter 3 a general Reliable Causal Broadcast (RCB) algorithm [BSS91] that is later used as a building block for verifying op-based CRDTs.

2. *Falls within the program logics camp.* Specifically, I use the Aneris distributed separation logic [Kro+20]. Unlike some of the related work [LF21], I do not develop a bespoke program logic that is specifically targeted towards reasoning about CRDTs. Instead, we can express all of the required technical gadgets (safety invariants, state tracking, denotational specifications, causal consistency) through existing features of Aneris (inherited from Iris) together with the metalogic (Coq): in particular we use Iris *invariants* and ghost state based on *partial commutative monoids* (PCMs). This has the advantage that our verified CRDT libraries remain fully-compatible with other Aneris developments (e.g. the causal broadcast algorithm and client code that uses the CRDTs).

3. *Verifies not only convergence, but also full functional correctness for both kinds of CRDTs.* I use CRDT denotations to specify both kinds of CRDTs. Even though denotations have been used in this way before [Bur+14], this is the first time that denotations are used to describe *runnable code*, as opposed to high-level descriptions of CRDTs based on transition systems [Bur+14] or pseudocode [ZBP14].

4. *Verifies runnable code, as opposed to abstract models or algorithms.* The implementations verified in this dissertation are written in AnerisLang, which can be thought of a subset of OCaml. In fact, the CRDT implementations were first written in OCaml and then translated into AnerisLang via an (unverified) transpiler. AnerisLang has many of the features of modern programming languages. The CRDT implementations make us of all of the following: mutation, higher-order functions, intra-node concurrency, locks, and (realistic) network primitives.

5. *Handles both op-based and state-based CRDTs.* This is one of the key contributions of the dissertation, since no prior work provides modular specifications of both kinds of CRDTs. The techniques of Liang and Feng [LF21] can also produce modular specifications of op-based CRDTs, but the authors acknowledge that extending their approach to the state-based setting is challenging:

   > This paper considers only operation-based CRDTs. Our results may be adapted to support state-based CRDTs when assuming causal delivery, but it seems nontrivial to build abstractions that on the one hand reflect the algorithms' resistance to unreliable networks, and on the other hand are still useful for client reasoning. [LF21]

   What enables the generality of the techniques presented here is a combination of (a) the expressive power of modern separation logics like Aneris, (b) the encoding of causality in separation logic by Gondelman et al. [Gon+21] and (c) the abstract nature of denotational specifications. With regards to point (a), for example, one challenge of verifying CRDTs is that updates to a replica's state can happen concurrently, but specifying concurrent

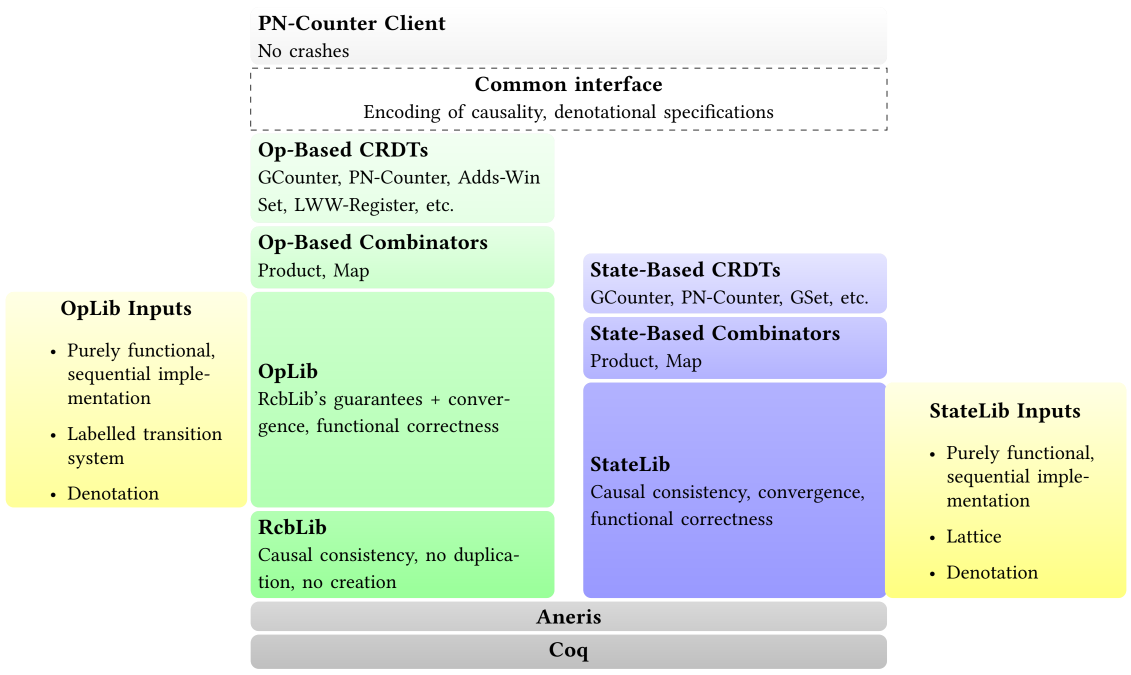

PN-Counter Client
No crashes
Common interface
Encoding of causality, denotational specifications
Op-Based CRDTs
GCounter, PN-Counter, Adds-Win Set, LWW-Register, etc.
Op-Based Combinators
Product, Map
OpLib Inputs
Purely functional, sequential implementation
Labelled transition system
Denotation
OpLib
RcbLib's guarantees + convergence, functional correctness
RcbLib
Causal consistency, no duplication, no creation
State-Based CRDTs
GCounter, PN-Counter, GSet, etc.
State-Based Combinators
Product, Map
StateLib
Causal consistency, convergence, functional correctness
StateLib Inputs
Purely functional, sequential implementation
Lattice
Denotation
Aneris
Coq

Figure 2.1: Overview of the material in Chapters 3 and 4. Green boxes are for op-based CRDTs, blue boxes for state-based CRDTs, and yellow-boxes for user inputs.

data structures is the bread-and-butter of concurrent separation logic [BB17], so we can use standard techniques like *logically atomic triples* [Jun+15] to give specifications in the face of concurrency. Regarding point (b), Gondelman et al. [Gon+21] observed that one can reason modularly about causality within separation logic by tracking *sets of writes* to a key-value store together with causality metadata (vector clocks). This dovetails perfectly with (c) the use of denotations for specifying CRDTs as in Burckhardt et al. [Bur+14], because in their telling the "essence" of CRDTs (their specification) is a mapping from sets of operations plus causality metadata to a resulting state.

6. *Supports verification of client programs.* Because we can give functional correctness specifications to CRDTs we can then verify client programs that *use* those CRDTs. We show a client program that uses a counter CRDT while making runtime assertions. We then go on to prove the client safe (the assertions never fail) by using the counter's specification, and without considering the client's implementation.

Part II of the dissertation substantiate the claims above. Chapters 3 and 4 are about verification of op-based and state-based CRDTs, respectively, and are summarized in Figure 2.1.

On Chapter 3 (green component stack in Figure 2.1) I implement and verify a *Reliable Causal Broadcast* (RCB) algorithm [BSS91]. RCB is a communication abstraction that strengthens the minimal guarantees provided by Aneris' message passing primitives to include causal delivery. The RcbLib library implements a modification of RCB known as *tagged reliable causal broadcast* that makes it more suitable for building CRDTs [BAS14].

Using RCB, I then verify OpLib: a library for building op-based CRDTs. OpLib turns a user-provided, purely functional and sequential implementation of a data structure into its corresponding CRDT. OpLib handles the mutation, concurrency control, and message passing that are part of every CRDT implementation. In addition to the sequential implementation, users also provide a denotation that serves as the CRDT's specification, as well as a labelled-transition system model that connects the denotation to its implementation. Using OpLib, I verify twelve example CRDTs from the literature, including two CRDT *combinators*. These combinators allow us to modularly combine CRDT implementations and their proofs to automatically build more complex verified CRDTs.

Similarly, Chapter 4 builds the verification infrastructure needed for verifying state-based CRDTs (blue component stack in Figure 2.1). This starts with a StateLib library that is the dual of OpLib. StateLib also takes as input a sequential implementation of a data structure, as well as its denotation-based specification and a lattice model. Instantiated thus, StateLib produces a state-based CRDT with the desired semantics. Because state-based CRDTs are automatically causally consistent, which I formally prove for the first time, StateLib does not rely on RCB. Using StateLib, I verify five example CRDTs, again including the map and product combinators. For example, I verify a GCounter CRDT and then instantiate the PN-Counter as a pair of GCounters.

Both developments in Chapters 3 and 4 use the same encoding of causality in separation logic that I generalize from Gondelman et al. [Gon+21]. Specifically, this encoding is expressed as a set of Aneris propositions together with lemmas the propositions satisfy (laws of causality). Each of the developments in Chapters 3 and 4 defines these propositions differently: in other words, they each implement the same "causality interface". Together with the fact that both developments use specifications based on denotations, this means we can build abstract specifications for CRDTs that hide whether the CRDTs are op-based or state-based. As proof of concept, I verify a client program that uses a PN-Counter CRDT (the client uses causal consistency to reason about the results of the counter's operations). I then implement and verify two versions of the PN-Counter: one version is built on top of OpLib and RcbLib, and the other is implemented using the state-based combinators and StateLib. I am able to verify the client code against the abstract specification of a PN-Counter, so that I can swap out the op-based code for the state-based code while making only minimal (proof engineering-related) changes.

2.2.2 Session Guarantees for Client-Centric Consistency

The distributed systems literature distinguishes between two kinds of consistency models: *data-centric* and *client-centric* [TS07, Chapter 7].

A data-centric consistency model offers a *global* view of the system, specifying the order in which operations are propagated across replicas. In particular, the model specifies which operations are concurrent and so can be re-ordered.

By contrast, a client-centric consistency model is concerned only with operations delivered at a *single* node (either local operations generated by the node, or operations propagated from remote replicas) within the context of a *session*, a sequence of local reads and writes. The model then specifies the relationship between operations in the session, as well as their propagation order to other replicas.

The client-centric view of consistency can be broken up into four *session guarantees*: *read your writes*, *monotonic reads*, *monotonic writes*, and *writes follow reads* [Ter+94]. These guarantees are intuitive reasoning principles for clients interacting with a weakly-consistent data type that implements them. For example, the monotonic reads guarantee states that reads always return values that are "more up-to-date" than values returned by previous reads in the same session.

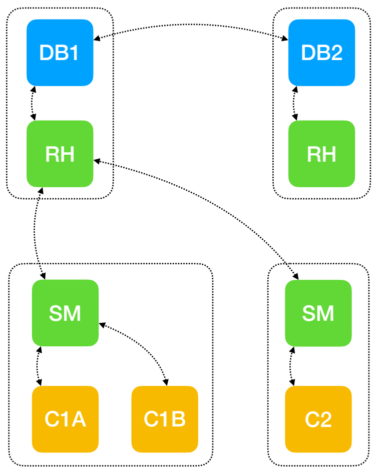


Figure 2.2: Clients using the distributed database via the session manager library. RH is the request handler, and SM is the client stub. Copied from Gondelman et al. [Gon+21].

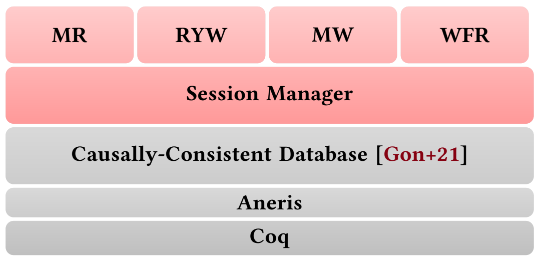


Figure 2.3: Overview of the verified components in Chapter 5. Each box depicts a component modularly verified against the *specification* of the component below.

Chapter 5 proves that the four session guarantees follow from the encoding of causality in separation logic developed by Gondelman et al. [Gon+21]. In that paper, my co-authors and I presented a verified implementation of a causally-consistent distributed database (a distributed key-value store), together with the first modular specifications of the same. Motivated the modular nature of the specifications and prior results showing that causal consistency implies all four session guarantees [BSW04], we wondered whether one could use the database in a setting where the session guarantees are relevant.

The result, presented in Chapter 5, is a *session manager* library that allows clients to use the key-value store even when the client and the database are located in different nodes (i.e. the typical case for client-server designs). Figure 2.2 shows the session manager being used by remote clients to interact with the database. To that effect, the session manager is divided into two parts: a *request handler* component that is co-located with the database and proxies client requests to it, and a *client* stub that implements a simple form of RPC for sending client requests to the server.

Just like the implementation, the proofs of the session guarantees are modular (Figure 2.3). On top of the specifications for the causally-consistent database in Gondelman et al. [Gon+21], Chapter 5 builds specifications for the session manager library. These specifications use only *persistent* resources, both facilitating client reasoning and reflecting the fact that no client has exclusive access to the database. Finally, using the session manager specifications I prove the

Table 2.1: The four session guarantees. Edited from Gondelman et al. [Gon+21].

Guarantee	**Program**		
Read Your Writes	sconnect(ip);	swrite(ip,k,v);	sread(ip,k)
Monotonic Reads	sconnect(ip);	sread(ip,k);	sread(ip,k)
Monotonic Writes	sconnect(ip);	swrite(ip,k1,v1);	swrite(ip,k2,v2)
Writes Follow Reads	sconnect(ip);	sread(ip,k1);	swrite(ip,k2,v)

four session guarantees, which appear as client programs (Table 2.1).

Chapter 5 is an extended, self-contained version of Section 5 of Gondelman et al. [Gon+21].

2.3 Contributions

Because the previous sections have been rather discursive, it bears succinctly re-stating the main contributions of this dissertation.

Chapter 3 and 4

- Present the first unified account of specification and verification of op-based and state-based CRDTs. In particular, Chapter 4 presents the first modular specifications of state-based CRDTs.
- Chapter 3 presents the first formally-verified implementation of causal broadcast, a common distributed systems building block.
- Chapter 4 contains the first formal proof that state-based CRDTs are causally-consistent.
- Develop a proof technique for CRDTs that has a number of desirable properties:
  - Proves functional correctness and not just eventual consistency.
  - Supports reasoning about clients of CRDTs.
  - Reasons at the level of runnable code, as opposed to high-level models.
  - Is conducted in a general-purpose separation logic that is not specialized to reason about CRDTs.
  - Is foundationally verified in Coq.

Chapter 5

- Presents the first formal verification of the four session guarantees for client-centric consistency. Specifically, I show that these guarantees hold for Gondelman et al. [Gon+21]'s causally-consistent database.
- Shows how to encode the guarantees in separation logic, therefore allowing client-side reasoning using the guarantees.
- All results are foundationally verified in Coq.

2.4 Included Publications

Part II consists of three chapters, each of which is based on a publication that I co-authored during my PhD:

- Chapter 3 is about verification of op-based CRDTs, and is a copy of the OOPSLA'22 paper by Nieto et al. [Nie+22]

  > Abel Nieto, Léon Gondelman, Alban Reynaud, Amin Timany, and Lars Birkedal. 2022. Modular Verification of Op-Based CRDTs in Separation Logic. Proc. ACM Program. Lang. 6, OOPSLA2, Article 188 (October 2022), 29 pages. https://doi.org/10.1145/3563351

  The text and figures have only been minimally edited to fit this dissertation's template.

  I was the main author of the paper and had the lead role in all aspects of the research project, including scoping, conceptualization, proof engineering, and paper writing.

- Chapter 4 is about verification of state-based CRDTs, and is a copy of the ECOOP'23 paper by Nieto et al. [Nie+23]

  > Abel Nieto, Arnaud Daby-Seesaram, Léon Gondelman, Amin Timany, and Lars Birkedal. 2023. Modular Verification of State-Based CRDTs in Separation Logic. In 37th European Conference on Object-Oriented Programming (ECOOP 2023). Schloss Dagstuhl-Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2023.12

  The text and figures have only been minimally edited to fit this dissertation's template.

  I was the main author of the paper and had the lead role in all aspects of the research project, including scoping, conceptualization, proof engineering, and paper writing. Arnaud Daby-Seesaram wrote the majority of the Coq proofs in this Chapter's formalization.

- Chapter 5 is about verification of session guarantees for client-centric consistency. It is based on the POPL'21 paper by Gondelman et al. [Gon+21]

  > Léon Gondelman, Simon Oddershede Gregersen, Abel Nieto, Amin Timany, and Lars Birkedal. 2021. Distributed causal memory: modular specification and verification in higher-order distributed separation logic. Proc. ACM Program. Lang. 5, POPL, Article 42 (January 2021), 29 pages. https://doi.org/10.1145/3434323

  I was the lead author of Section 5 of the paper, on session guarantees, as well as the associated Coq development. I also contributed to technical discussions and general editing of the paper. Chapter 5 is an extended, self-contained, version of Section 5. Parts of the chapter's introduction, the entirety of Section 5.1, and some figures are copied and edited from the paper, as well as the accompanying technical appendix [Gon+20]. The rest is original material.

2.5 *Coq Formalizations*

The material in Part II is formalized using Mosel [Kre+18a], a Coq framework for Iris (and Aneris) developments. The formalizations span over 28000 lines of Coq code, and all three received a "Reusable" badge after assessment by the corresponding artifact evaluation committees:

Chapter	**Artifact Link**	**Artifact Size (LOC)**
Chapter 3	`https://zenodo.org/record/7055010`	14748
Chapter 4	`https://zenodo.org/record/7718868`	12064
Chapter 5	`https://zenodo.org/record/4139601`	1883

Part II

Publications

3 Modular Verification of Op-Based CRDTs in Separation Logic

This chapter is a copy of the OOPSLA'22 paper "Modular Verification of Op-Based CRDTs in Separation Logic" [Nie+22], of which I was the lead author. The text and figures have been lightly edited for formatting.

Abstract

Operation-based Conflict-free Replicated Data Types (op-based CRDTs) are a family of distributed data structures where all operations are designed to commute, so that replica states eventually converge. Additionally, op-based CRDTs require that operations be propagated between replicas in causal order. This paper presents a framework for verifying safety properties of CRDT implementations using separation logic. The framework consists of two libraries. One implements a Reliable Causal Broadcast (RCB) protocol so that replicas can exchange messages in causal order. A second OpLib library then uses RCB to simplify the creation and correctness proofs of op-based CRDTs. OpLib allows clients to implement new CRDTs as purely-functional data structures, without having to reason about network operations, concurrency control and mutable state, and without having to each time re-implement causal broadcast. Using OpLib, we have implemented 12 example CRDTs from the literature, including multiple versions of replicated registers and sets, two CRDT combinators for products and maps, and two example use cases of the map combinator. Our proofs are conducted in the Aneris distributed separation logic and are formalized in Coq. Our technique is the first work on verification of op-based CRDTs that satisfies both of the following properties: it is *modular* and targets executable *implementations*, as opposed to high-level protocols.

To an outside observer, a distributed system ideally appears to function as a single computer, and the fact that the system is composed of multiple collaborating processes is an implementation detail hidden inside the proverbial black box. This behaviour is formally captured by the notion of *linearizability* [HW90], which says that concurrent execution histories of a linearizable data structure can be re-ordered so that operations appear to take place (a) atomically and (b) in a manner that is consistent with sequential order.

Alas, the CAP[1] theorem [GL02] shows that, in the presence of network partitions, a system can be either linearizable or available, but not both. *Available* in this context means that the nodes in different network partitions can (independently) continue to service client requests, without waiting for the partitions to heal.

[1]Consistency, Availability, Partition tolerance

Confronted with this consistency vs availability dilemma, practitioners have developed systems that trade off stronger forms of consistency (e.g. linearizability and sequential consistency) in favour of better availability (e.g. Chang et al. [Cha+08], Sivasubramanian [Siv12], Tyulenev et al. [Tyu+19], Bailis et al. [Bai+13], Lloyd et al. [Llo+11], and Chodorow and Dirolf [CD10]). This is possible by adopting weaker consistency models; among such models are *strong eventual consistency* (SEC) [Sha+11b] and *causal consistency* [Aha+95]. For example, in SEC two processes that read from a replicated register might observe different values even though no intervening writes have occurred locally (something not possible when reading from sequentially-consistent local memory from within a process). Eventually, however, the state of the replicated register at different replicas must converge. More precisely, SEC requires the following two properties (note the first is a liveness property while the latter is a safety property):

- *(Eventual Delivery)* An update delivered to a correct replica is eventually delivered to all replicas.
- *(Convergence)* Replicas that have delivered the same updates eventually reach equivalent states.

Conflict-free Replicated Datatypes (CRDTs) [Sha+11a] are a class of distributed systems where a data structure (e.g. register, set, or map) is replicated over multiple replicas that mutate its state via local operations. Because replicas are allowed to invoke operations without coordinating with others, different replicas might arrive at conflicting states. CRDTs resolve such conflicts automatically. There are two main ways of going about this. One option is to model the replica state as a (join) semilattice, so that merges are accomplished by taking least upper bounds (joins); these are *state-based* or *convergent* CRDTs. Changes are then propagated by sending the entire state to other replicas on the (possibly unreliable) network. Another option is to propagate, instead of the entire state, just the effect of each individual update. It becomes then necessary to enforce that each operation is executed exactly once (at most once for the convergence and at least one for the eventual delivery properties above), which typically requires broadcasting primitives that offer reliable delivery. Furthermore, it is also necessary to enforce that some or all operations commute so that concurrent operations can be applied in any order. This last class, known as operation-based (*op-based*) or *commutative* CRDTs, is the focus of this paper. [2]

Consider the following example of a counter data structure replicated over two nodes:

```
(* Node A *)
add 1; add 200

(* Node B *)
add 2; let v = read () in
assert((v = 2) || (v = 3) || (v = 203))
```

The counter exports two operations: `add(z)`, which adds an integer `z` to the counter, and `read()`, which returns the counter's current value. This CRDT is known as a *positive-negative counter* (PN-Counter)[Sha+11a].

[2] From now on whenever we use the term *CRDT* the reader can safely assume that we mean *op-based* CRDT, unless explicitly noted otherwise.

One question of interest for the example above is what are the possible values of `v`. Because the counter should remain available even if A and B are partitioned, A's `add(1)` should execute without trying to synchronize with B. This means that A's and B's `add` operations potentially happen concurrently. By contrast, when A's two operations are broadcast to B, they should be applied by B following A's program order. Finally, when B reads, we do not know whether A's updates have been received, but we do expect that the `add(2)` has been recorded locally. This means that the possible values for `v` are `2` (only the local `add` has been applied), `3` (only A's first `add` has been applied), and `203` (all three `add`s have been applied). Results like `0`, `200` and `202` are not valid answers.

Causal Delivery Our intuitions about valid execution traces in the example above can be captured by a *happens-before* or *causality* relation on events [Lam78]. Let a and b be two events (possibly taking place at different processes). Then a *happens before* b (and b is *causally dependent* on a), written $a \rightarrow b$, if one of the following holds:

- a and b take place in the same process, and $a < b$ according to *program order*.
- a is the event of sending a message m and b is the corresponding event where m is received.
- $a \rightarrow c$ and $c \rightarrow b$ for some other event c (the transitive closure of the above two rules).

If neither $a \rightarrow b$ nor $b \rightarrow a$, then we say they are *concurrent*, written $a||b$. Informally, we say that events are *causally delivered* if the following property holds: if an event e is delivered[3] to a replica p, then all events on which e causally depends must have been previously delivered to p. We can then require that valid PN-counter execution traces satisfy causal delivery of operations. Indeed, this is a common requirement for many CRDTs in the literature [BAS14].

Reliable Causal Broadcast One way to realize the guarantees of causal delivery is to implement a one-to-many communication protocol known as *Reliable Causal Broadcast* (RCB) [CGR11]. In RCB, a group of N replicas send each other messages. The protocol's interface consists of two functions: $\mathsf{broadcast}(msg)$, which sends message msg to all other $N - 1$ replicas, and $\mathsf{deliver}()$, which returns a received message (if one exists) while respecting causal order.

Verifying CRDTs Because CRDTs are data structures replicated across multiple processes, each of which is allowed to reorder concurrent operations, they are challenging to specify and verify.

The main property of interest for verification is SEC [Sha+11b] which as we mentioned can be divided into convergence and eventual delivery. [4] However, convergence does not say how the CRDT's final state is computed from the set of received operations. Burckhardt et al. [Bur+14] addressed this question by showing how to give *functional correctness* specifications for CRDTs. Another consideration is whether the verified properties can be reused by components other than the CRDT: that is, whether the verification technique is *modular*. The recent

[3]Delivery occurs when the event processing layer makes its clients aware of the event; this can take different forms depending on the specific application.

[4]The terminology is not universal: Shapiro et al. [Sha+11a] refers to both properties together as *eventual convergence*.

work of Liang and Feng [LF21] presents the first modular verification technique for op-based CRDTs.

An additional design decision is the level of detail at which to model the CRDT that is the target of verification. There are roughly two options: one can model the CRDT as a high-level protocol, perhaps assuming that the network is reliable or ignoring node-local concurrency. Alternatively, we can implement the CRDT in a general-purpose programming language where we have to deal with a plethora of low-level (but realistic) details such as an unreliable network, concurrency-control, and mutation.

Our work This paper is about proving SEC and functional correctness of op-based CRDTs. To the best of our knowledge, all prior work on verification of op-based CRDTs consists of techniques that produce modular specifications but work at the protocol level, or techniques that work for implementations but are non-modular (see Section 3.6 for a classification of prior work). The main contribution of our work is to lift that restriction: we can produce *modular* specifications of CRDT *implementations*. Additionally, unlike prior work which assumes causal delivery by the network, our CRDTs include a general-purpose implementation of reliable causal broadcast. All our proofs are mechanized in Coq. More precisely, the contributions of this work are as follows:

1. We implemented and verified an RcbLib library for reliable causal broadcast (RCB). To the best our knowledge, this is the first time a formalization of op-based CRDTs includes a general-purpose implementation of RCB, as opposed to assuming causal broadcast.

2. On top of the RcbLib library, we implemented and verified an OpLib library for building op-based CRDTs. Using OpLib, one can create op-based CRDTs as purely-functional data structures, without having to reason about low-level details like mutation, concurrency control, and network operations. Similarly, by proving only simple sequential specifications, OpLib users obtain from the library rich specifications for their CRDTs, enabling modular reasoning about convergence, causality, and functional correctness.

3. We evaluated OpLib by implementing a collection of 12 CRDTs, including multiple versions of registers and sets, as well as two combinators for products and maps. We further evaluated the modularity of our specifications by verifying a client program that uses a CRDT obtained via OpLib.

4. We wrote our libraries in a subset of OCaml that is then automatically translated to AnerisLang, the programming language of the Aneris [Kro+20] distributed separation logic. Our proofs were conducted in Aneris and are mechanized in Coq.

Structure of the paper The rest of the paper is organized as follows: Section 3.1 gives a quick primer to the Iris and Aneris program logics. Section 3.2 provides an overview of the key ideas of our work and presents the concepts that CRDT implementers need to use our libraries. Section 3.3 describes in more detail RcbLib's implementation and correctness proof. Section 3.4 then does the same for OpLib. Section 3.5 discusses our case studies (the implemented CRDTs). We then take a look at prior work on Section 3.6, and conclude in Section 3.7.

$$\begin{array}{lll} P, Q \in \mathit{iProp} ::= \mathsf{True} \mid \mathsf{False} \mid P \wedge Q \mid P \Rightarrow Q \mid P \vee Q \mid \forall x.\ P \mid \exists x.\ P \mid \cdots & & \text{higher-order logic} \\ \qquad \mid P * Q \mid P \twoheadrightarrow Q \mid \ell \mapsto_{ip} v \mid \{P\}\ \langle ip; e \rangle\ \{x.\ Q\} \mid \Box P & & \text{separation logic} \\ \qquad \mid \boxed{P}^{\mathcal{N}} \mid {}_{\mathcal{E}_1}{\Rrightarrow}_{\mathcal{E}_2} & & \text{Iris resources and invariants} \end{array}$$

Figure 3.1: The fragment of Iris and Aneris relevant to this paper.

3.1 Aneris Primer

Iris [Jun+18] is a state-of-the-art program logic designed to reason about concurrent programs based on separation logic. Aneris [Kro+20] is a program logic built on top of Iris for reasoning about distributed systems. Figure 3.1 shows the fragment of Iris and Aneris logic that we need in this paper:

First and foremost Iris is a higher-order logic with the usual connectives. Note how we can quantify, both existentially and universally, over any domain, including *iProp* itself (we write *iProp* for the universe of Iris propositions). Iris is a separation logic. Iris propositions can assert ownership of resources and express their *disjointness*. The proposition $P * Q$ holds if the owned resources can be split into two disjoint parts where one satisfies P and the other Q. The *magic wand*, $P \twoheadrightarrow Q$, also called separating implication, asserts ownership over resources that when combined with (disjoint) resources satisfying P would satisfy Q. The so-called points-to proposition, $\ell \mapsto_{ip} v$, asserts exclusive ownership over the memory location ℓ stating that the value stored in this location is v. This proposition differs from the standard separation logic points-to proposition only in that it is annotated with the *Ip* address of the node to which it belongs — this is necessary as we are working with a distributed system in Aneris. Similarly, in Aneris a Hoare-triple $\{P\}\ \langle ip; e \rangle\ \{x.\ Q\}$, in addition to the program, also takes the *Ip* address of the node the program is running on.

The persistently modality, $\Box$, captures duplicability of propositions. It allows us to distinguish between propositions that are duplicable and those that are not, *e.g.*, points-to propositions: $\ell \mapsto_{ip} v * \ell \mapsto_{ip} w \vdash \mathsf{False}$. Here, $\vdash$ is the logical entailment relation of Iris. Intuitively, $\Box P$ holds if P does and furthermore, P does not assert ownership of any non-duplicable resources. We say a proposition is persistent if $P \vdash \Box P$; note that for any proposition P we always have $\Box P \vdash P$. Persistent propositions are duplicable, *i.e.*, $\Box P \vdash \Box P * \Box P$, and hence they merely express knowledge as opposed to expressing (exclusive) ownership over resources. An example of a persistent proposition is Iris invariants. The invariant $\boxed{P}^{\mathcal{N}}$ asserts that P must hold at all times throughout program execution. Hence, throughout a proof, for the duration of an atomic step of computation, we can access invariants, *i.e.*, we get to know that the invariant holds before the step of computation and need to guarantee that it also holds afterwards. The name of the invariant $\mathcal{N}$ is used to track accesses to invariants and prevent them from being accessed in an unsound manner, *e.g.*, accessing the same invariant twice during the same atomic step of computation which could result in duplicates of non-duplicable propositions like the points-to proposition. The update modality,[5] ${}_{\mathcal{E}_1}{\Rrightarrow}_{\mathcal{E}_2}$, allows manipulation of invariants and resources in Iris. The masks $\mathcal{E}_1$ and $\mathcal{E}_2$ are sets of invariant names and respectively indicate which invariants hold before and after the "update" takes place. We write ${\Rrightarrow}_{\mathcal{E}}$ for ${}_{\mathcal{E}}{\Rrightarrow}_{\mathcal{E}}$. The update modality is the primary way of working with invariants in Iris. They

[5]In Iris jargon this modality is called the fancy update modality; see Jung et al. [Jun+18] for more details.

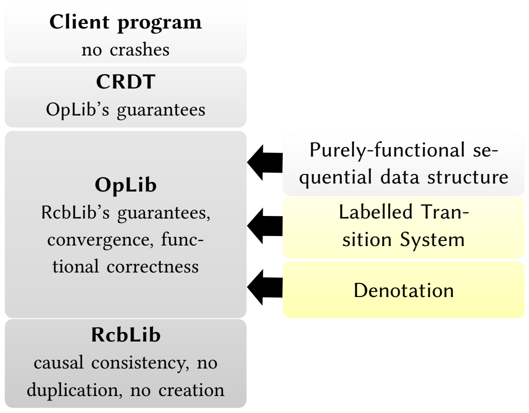

Client program
no crashes
CRDT
OpLib's guarantees
OpLib
RcbLib's guarantees, convergence, functional correctness
RcbLib
causal consistency, no duplication, no creation
Purely-functional sequential data structure
Labelled Transition System
Denotation

Figure 3.2: Overview of our development. The OpLib library is parameterized by a CRDT specification given by the components in the right column. Grey boxes are written in OCaml/AnerisLang; yellow boxes are written in Coq.

are used in the definition of Iris Hoare-triples in such a way as to enforce the aforementioned invariant policy of only allowing access to invariants during atomic steps of computation. Intuitively, the proposition ${}_{\mathcal{E}_1}{\Rrightarrow}_{\mathcal{E}_2} P$ holds if we can manipulate resources (allocate new resources, or update the existing ones) and manipulate invariants (create new invariants, access invariants, or reestablish invariants) so as to make sure that P holds. Furthermore, during this update we can access all invariants in $\mathcal{E}_1$ but must ensure that all invariants in $\mathcal{E}_2$ hold after the update is done.

3.2 *Main Ideas*

This section provides a birds-eye view of the paper, focusing on concepts users need to use our libraries. Figure 3.2 shows an overview of our work. We structured our development as a tower of components, each exporting a modular specification.

Higher-level components can then be verified using solely the specifications of its dependencies, without knowledge of the dependency's implementation. Each box in Figure 3.2 lists a component and the safety properties guaranteed by its specification. Grey boxes are written in OCaml; [6] yellow boxes are written in Coq.

3.2.1 *RcbLib*

At the base of our verified tower of components we have a library implementing a reliable causal broadcast protocol [CGR11]. This library is built on top of UDP, so it makes minimal assumptions about network guarantees. In particular, messages can be dropped, re-ordered, and duplicated by the network. The library deploys a suite of techniques, such as sequence ids, acknowledgments, retransmissions, and a delay queue, to offer three main guarantees: broadcast messages are delivered in causal order, without duplicates, and ensuring that any message delivered was previously broadcast by another participant (the *no creation* property in Figure 3.2). These are the three safety properties of RCB [CGR11].

[6]Later automatically translated to AnerisLang, the programming language of the Aneris distributed separation logic.

Verifying RcbLib The main idea for verifying RcbLib is to generalize the treatment of causality in Gondelman et al. [Gon+21] to the causal broadcast setting. We now briefly outline our approach and expand on it in Section 3.3.

The first step is to define separation logic resources tracking the set of broadcast messages between replicas in two ways: the $\mathsf{OwnGlobal}(h)$ resource provides a *global view* tracking the set h of all messages broadcast by any replica, while the $\mathsf{OwnLocal}(i, s)$ resource provides a *local view* tracking the set s of all messages that has been delivered by replica i. Here, messages are triples $(\mathsf{p}, \mathsf{vc}, \mathsf{o})$ consisting of the message's payload p, vector clock vc, and id of the originating replica o.

The next step is to craft separation logic specifications for RcbLib's broadcast and deliver functions. Below, we show a simplified specification for broadcast :

$$\begin{array}{l}\{\mathsf{OwnGlobal}(h) * \mathsf{OwnLocal}(i, s)\}\\ \quad \langle ip_i; \mathsf{broadcast}(p)\rangle\\ \{m.\, \mathsf{payload}(m) = p * \mathsf{OwnGlobal}(h \uplus \{m\}) * \mathsf{OwnLocal}(i, s \uplus \{m\})\}\end{array}$$

This spec states that in order to broadcast a message with payload p, we need to provide both the global view and the local view of the broadcasting replica. broadcast can then execute without errors and return a message m with payload p. Logically, we know that the global set of broadcast messages now includes m, and also that node i has delivered (is aware of) the new message.

In addition to the broadcast and deliver specifications, following Gondelman et al. [Gon+21] we provide to the user of RcbLib a set of laws governing the above resources. Notably, the causality law states that, given the ownership of $\mathsf{OwnGlobal}(h)$ and $\mathsf{OwnLocal}(i, s)$, we can conclude that

$$\forall m \in s, m' \in h.\ \mathsf{vc}(m') < \mathsf{vc}(m) \Rightarrow m' \in s$$

i.e., for any message m that has been delivered at node i, if we know of another message m' that has been broadcast by any other node such that m' happened before m, [7] then it must be the case that node i has previously delivered m' as well. All laws are proven in Coq and provided as lemmas.

3.2.2 *OpLib*

Conceptually, an op-based CRDT implementation can be seen as an infinite loop that maintains the CRDT's state at a given replica. This loop has a number of responsibilities:

1. accept *local* operations invoked by the user at the replica
2. modify the CRDT's state as per the effects of local operations
3. propagate local operations to other replicas
4. listen for *remote* operations communicated via the network
5. modify the CRDT's state as per the effects of remote operations

[7] $\mathsf{vc}(m)$ stands for m's *vector clock*, a mechanism for tracking causal dependencies.

One can then observe that there are a number of derived responsibilities that flow from the ones above: for example, since steps (2) and (5) can happen concurrently, some form of concurrency control (e.g. locking) is needed. Additionally, because the network is unreliable, step (3) requires that the CRDT is be able to tolerate dropped messages. Another observation is that most of the steps above are agnostic to the semantics of the specific CRDT: only when modifying the CRDT's state (steps (2) and (5)) do we need to know the inner workings of the data type's operations.

These observations suggest a design where the generic responsibilities are factored out as a library that is parametric on the CRDT's operations and their effects. Inspired by the approach in Baquero et al. [BAS14], we instantiate a CRDT via the OpLib library that we have implemented on top of RcbLib. In our library, all that the user needs to provide is the data type's *initial state* and an *effect* function that can process new operations. This design allows a CRDT implementer to focus on the core logic of their data type as a purely-functional data structure, while delegating to OpLib all the gritty details of inter-replica communication, concurrency control, and mutation. Because OpLib uses RcbLib for propagating operations between replicas, clients can rely on the guarantees of causal broadcast. Once instantiated with the user's purely functional data type, OpLib turns it into a fully-fledged CRDT that exports two functions: $\mathsf{get\_state}()$, which returns (a copy of) the CRDT's current state, and $\mathsf{update}(\mathsf{op})$ which updates the state via a new operation *op*.

Verifying OpLib To verify OpLib we adapt the notion of CRDT *denotations* [LAB19; Bur+14] to separation logic. A CRDT denotation $[\![\cdot]\!] : 2^{Msg} \rightharpoonup St$ is a (partial) function from sets of messages (a message contains an operation plus causality metadata) to the CRDT state that results from executing said operations. Both *Msg* and *St* vary depending on the specific CRDT. For example, the denotation for a PN-Counter is a function that maps a set of messages to the sum of its payloads: $[\![s]\!] = \sum_{m \in s} \mathsf{payload}(m)$.

Denotations have been previously used to give high-level specifications for CRDTs as well as CRDT combinators (e.g. products of CRDTs and maps where the value type is an arbitrary CRDT) [LAB19; Bur+14]. However, those works do not use denotations to verify implementations. We adapt denotations by constructing a separation logic resource $\mathsf{LocSt}(i, \bullet\, s, \circ\, r)$[8] which tracks the sets s and r of local and remote operations, respectively, processed by replica i. The key insight behind the resource $\mathsf{LocSt}(i, \bullet\, s, \circ\, r)$ is that it tracks *precisely* the set of processed local operations s, but provides only *a lower bound* on the set of processed remote operations r. This captures the intuition that while a CRDT user can control which local operations they perform, they do not know which additional remote operations have been propagated from other replicas at a given moment in time. The simplified spec for get_state below shows how the resource is used:

$$\{\mathsf{LocSt}(i, \bullet\, s, \circ\, r)\}\ \mathsf{get\_state}()\ \{m.\, \exists r', r \subseteq r' * m = [\![s \cup r']\!] * \mathsf{LocSt}(i, \bullet\, s, \circ\, r')\}$$

The spec says that prior to calling get_state we must know that replica i has processed exactly the local messages in s, and at least the remote messages in r. The function then returns a state m that is the denotation of the set $s \cup r'$, where r' is a superset of r. This is because in between calls to get_state the CRDT might have processed additional remote operations.

[8] The notation is reminiscent of the so-called authoritative resource algebra [Jun+18].

3.2.3 CRDT Instances

The last element of Figure 3.2 we highlight is the recipe that CRDT implementers follow to use OpLib:

- First, the CRDT implementer must provide a denotation for their CRDT.
- In order to bridge the abstraction gap between the denotation, stated in terms of the sets of operations, and the effect function, which must process one operation at a time, the user provides a second specification in the form of a *labelled-transition system* (LTS). In this LTS, states are the CRDT's states and the transitions are labelled with operations. That is, a transition $s \xrightarrow{op} s'$ means that if the CRDT is in state s and an operation op is received, then it will end up in state s'. Importantly, the denotation and LTS must agree in the following sense: if h is a set of operations such that $[\![h]\!] = s$, and $s \xrightarrow{op} s'$, then we must have $[\![h \cup \{op\}]\!] = s'$.
- Finally, the user shows that their effect function is coherent with the LTS via a Hoare triple.

The first two steps are conducted outside separation logic in the meta-logic (Coq), while the last step requires proving a Hoare triple in Aneris.

We have followed the recipe above to implement 12 CRDTs, including multiple kinds of registers and sets, as well as two CRDT combinators for products and maps. Our combinators use Coq typeclasses as in Liu et al. [Liu+20] to automatically generate and prove correctness of compound CRDTs from constituent CRDTs.

Our examples come from the CRDT literature [Sha+11a; BAS14; LAB19]. Importantly, they include CRDTs where all operations naturally commute (e.g. PN-Counter) as well as others that require causality information to make operations commutative (e.g. Last-Writer-Wins Register and Add-Wins Set). This shows that our approach scales to different CRDT designs.

3.3 Reliable Causal Broadcast

The network primitives (send and receive) provided by AnerisLang are for *point-to-point* communication: that is, a process communicating with a single other process. They are also, as previously mentioned, unreliable in a number of ways: messages can get lost, duplicated, and re-ordered in transit.

A useful abstraction in distributed systems is that of *broadcast*. In broadcast, or *one-to-many* communication, a process transmits the same message to one or more other processes. There exist different broadcast algorithms providing different guarantees: one such kind is *reliable causal broadcast* (RCB). In RCB, clients are provided with two operations, broadcast(msg) and deliver() that satisfy the following properties (taken from Cachin et al. [CGR11] and classified as either liveness or safety properties):

- (RCB1, liveness) *Validity*: if a correct process p broadcasts a message m, then p eventually delivers m.
- (RCB2, **safety**) *No duplication*: no message is delivered more than once.
- (RCB3, **safety**) *No creation*: if a process delivers a message m with sender s, then m was previously broadcast by process s.

Table 3.1: Challenges and techniques employed in RCB's implementation.

Challenge	**Technique**
Messages can be dropped, re-ordered and duplicated by the network.	Stop-and-wait protocol [TS07] using sequence ids, acknowledgments, and retransmissions to handle unreliable network.
The broadcasting process can be partitioned from the network before all processes receive a broadcast.	Eager reliable broadcast (retransmissions) [CGR11].
Messages need to be delivered in causal order.	Delay delivery of messages until causal dependencies are satisfied, using a *delay queue* and *vector clocks* [BSS91].

- (RCB4, liveness) *Agreement*: if a message m is delivered by some correct process, then m is eventually delivered by every correct process.
- (RCB5, **safety**) *Causal delivery*: for any message m_1 that potentially caused a message m_2, i.e., $m_1 \rightarrow m_2$, no process delivers m_2 unless it has already delivered m_1.

In this section, we sketch our implementation of a library for RCB, RcbLib, based on Birman et al. [BSS91] and Baquero et al. [BAS14]. We proved specifications of our implementation that satisfy the three safety properties above. In fact, our RCB library implements a slightly stronger specification than regular RCB, because it exposes to its clients causality information associated to messages in the form of vector clocks. The additional information provided by this *tagged* form of RCB [BAS14] simplifies the task of building CRDTs using OpLib (see Section 3.5).

3.3.1 Implementation

Since AnerisLang's network primitives provide few guarantees, RcbLib deploys a few different techniques in order to achieve the safety properties just mentioned. Some of the challenges and their solutions are outlined in Table 3.1. Additionally, Figure 3.3 provides a high-level view of the design of the RCB algorithm. The main components are outlined below.

Receive and send threads RcbLib consists of two concurrent threads that operate on a set of shared data structures (concurrent accesses are synchronized via a lock). The *receive* thread listens for messages on a network socket and places them in a *delay queue* and a collection of *out-queues*. It also acknowledges received messages so other replicas can move on to broadcasting new messages.

The *send thread* sends the messages in the out-queues to other replicas following a *stop-and-wait* protocol [TS07]. That is, a message is repeatedly sent to another replica until it is acknowledged by the foreign replica; at which point the send thread pops the relevant out-queue and moves on to a not-yet-acknowledged message.

Library API The library has two client APIs: deliver and broadcast. The former removes a message m from the delay queue such that all of the message's causal dependencies have previously been delivered (i.e. a message that comes next according to causal order). If no such

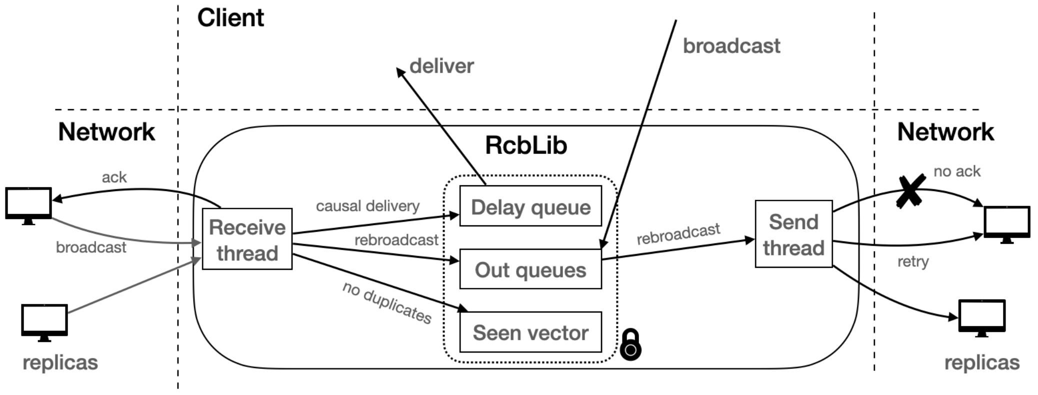


Figure 3.3: Structure of the reliable causal broadcast library.

message exists, deliver returns None; otherwise it returns Some m. The broadcast function broadcasts a message to all replicas (except to the current one). It does so by placing the message in all out-queues, so it can be later picked up by the send thread. broadcast(p) returns a new message m' containing the payload p together with the vector clock assigned to m' and the issuing replica's id. Because a replica doesn't broadcast to itself it must use the return value of broadcast if it wants to process the newly-broadcast message m'.

Vector clocks We use vector clocks to keep track of logical time [Fid87; Mat+88]. A vector clock is an array of non-negative integers; there is one array entry per replica in the system, and each entry records the number of events that originate at the corresponding replica. It is possible to merge two vector clocks by taking the maximum of their entries pointwise. We can define a partial order $\leq_{vc}$ on vector clocks by lifting $\leq$ (from $\mathbb{N}$) pointwise. The following result then holds: let a and b be events. then $a \rightarrow b$ iff $vc(a) < vc(b)$.

Replicas maintain internal state with their current vector clock. Every sent message m is also tagged with a vector clock $vc(m)$. When broadcast is called, the replica increments its entry within the internal vector clock and tags the event with it. When the receive thread receives a new message, its vector clock is not immediately merged with the replica's vector clock; instead, the merge is delayed while the message waits in the delay queue.

Delay queue In order to ensure causal delivery of messages, RCB stores messages received from other processes in a delay queue. That is, we do not deliver received messages immediately to the user. Given the internal vector clock v_i and a message m from the delay queue, we can determine whether (a) all causal dependencies of the message have been previously delivered and (b) the message has not been previously delivered. We do this using the following *delivery condition* [BSS91]:

$$\text{canDeliver}(m, v_i) \triangleq \forall k \in \{1 \dots n\} \begin{cases} vc(m)[k] = v_i[k] + 1 & \text{if } k = \mathit{origin}(m) \\ vc(m)[k] \leq v_i[k] & \text{otherwise} \end{cases}$$

Once the delivery condition for m is met, it is safe (causally consistent) to deliver m to the user in the next invocation of deliver. At that point, the internal vector clock v_i can be updated by merging it with $vc(m)$.

Out queues Consider the following scenario. There are three processes A, B and C. A broadcasts a message m to B and C. After A has sent m to B, but before it has a chance to send it to C, the network becomes partitioned into two partitions $\{A\}$ and $\{B, C\}$. Now B receives m, but C will not receive m until the partition is healed. This violates the agreement (RCB4) property of Section 3.3 because the partition might never heal, so C might never get m. Additionally, suppose that B creates a new message m', which is now causally dependent on m: $m \to m'$. Even though B and C are in the same partition, C cannot deliver m' until it delivers m first (a causal dependency). The whole system is stuck because one process is partitioned.

For this reason, RCB implements a form of *eager reliable broadcast* [CGR11]. That is, every process re-broadcasts every single message received to every other process (taking care to not enter into loops). Eager rebroadcasting is inefficient, since for every message sent there are $O(n^2)$ re-broadcasts in a system with n replicas (as opposed to $O(n)$, which is the best case for broadcast). We have chosen this mechanism for the first iteration of the RCB library due to its simplicity.

Given the need to re-broadcast messages, and because the network is unreliable, each process maintains a set of *out queues*, one per other process in the system (so n queues per node). Each queue contains the outbound messages that need to be sent to a specific process, but have not yet been acknowledged by that process. Messages are copied from the delay queue to the out queues, and are removed from the out queues when acknowledged by the intended recipient.

Seen vector A message could be received multiple times by the same process: because the network generated a duplicate or the message was re-broadcast multiple times by other processes. In either case, we need a mechanism to avoid re-delivery of the same message; in other words, we need to avoid putting the same message twice in the delay queue. To this effect, we use vector clocks as sequence identifiers. Given a message m, the pair $(origin(m), vc(m)[origin(m)])$ uniquely identifies a message in the system. We can then construct a *seen vector* where the ith entry gives us the highest sequence id of a message originating from process i that has been previously received. We only place a message originating at process i in the delay queue if its sequence id is higher (by one) than the current value of seen$[i]$.

3.3.2 *Specification*

As mentioned in Section 3.2, the specifications for deliver and broadcast (shown in Figure 3.4) use separation logic resources that keep track of the local and global states of the broadcast. The local resource $\mathsf{OwnLocal}(i, s)$ tells us that in process i RcbLib has previously delivered exactly the messages in s. Similarly, the global resource $\mathsf{OwnGlobal}(h)$ implies that h is exactly the set of messages that have been broadcast by any replica. We also maintain a *global invariant* RcbInv that ensures that global and local states are compatible. The invariant states that at all times if we combine all local states we obtain the global state, and furthermore that the local states satisfy causal delivery.

Deliver Figure 3.4 shows the specification of RCB's deliver function. The intuition is that before calling deliver we should know which messages have been previously delivered at this process (via ownership of a resource $\mathsf{OwnLocal}(i, s)$). After deliver returns, there are two possibilities:

$$
\begin{array}{l}
\textsc{DeliverSpec}\\
\langle \mathsf{OwnLocal}(i, s)\rangle\\
\quad \langle ip_i;\ \mathsf{deliver}()\rangle\\
\left\langle \begin{array}{l}
v.\ \exists s' \supseteq s.\ \mathsf{OwnLocal}(i, s') \ast\\
\quad \Big((v = \mathsf{None} \land s' = s) \lor\\
\quad (\exists w, a.\ v = \mathsf{Some}\ w \ast \mathsf{IsLocEv}(a, w) \ast\\
\quad\ s' = s \cup \{a\} \ast a \notin s \ast\\
\quad\ a \in \mathrm{Maximals}(s') \ast \mathit{origin}(a) \neq i \ast\\
\quad\ \mathsf{OwnGlobalSnapshot}(\{\lfloor a \rfloor\}))\Big)
\end{array}\right\rangle^{\mathcal{N}}
\end{array}
$$

$$
\begin{array}{l}
\textsc{BroadcastSpec}\\
\langle \mathsf{OwnLocal}(i, s) \ast \mathsf{OwnGlobal}(h)\rangle\\
\quad \langle ip_i;\ \mathsf{broadcast}(v)\rangle\\
\left\langle \begin{array}{l}
w.\ \exists a.\ \mathsf{IsLocEv}(a, w) \ast a \notin s \ast \lfloor a \rfloor \notin h \ast\\
\quad \mathit{payload}(a) = v \ast \mathit{origin}(a) = i \ast\\
\quad a \in \mathrm{Maximals}(h \cup \{\lfloor a \rfloor\}) \ast\\
\quad a \in \mathrm{Maximum}(s \cup \{a\}) \ast\\
\quad \mathsf{OwnLocal}(i, s \cup \{a\}) \ast\\
\quad \mathsf{OwnGlobal}(h \cup \{\lfloor a \rfloor\})
\end{array}\right\rangle^{\mathcal{N}}
\end{array}
$$

Figure 3.4: Logically-atomic specifications for deliver and broadcast. $\mathcal{N}$ is any namespace containing the global invariant's name.

- No messages were available for delivery, so the function returns None, and we get back our unchanged $\mathsf{OwnLocal}(i, s)$.
- There was a message a available for delivery. In this case, the function returns Some w, where w is the physical counterpart to a, reflected by the predicate $\mathsf{IsLocEv}(a, w)$. Additionally, we receive back a resource $\mathsf{OwnLocal}(i, s \cup \{a\})$. That is, we logically record the delivery of the new message. Crucially, we know that $a \notin s$, meaning that the returned message has not been previously delivered. Additionally, we get to know that a is *maximal* with respect to vector clock order in the set $s \cup \{a\}$. This means that no previously-received message could causally depend on a (but a can depend on previous messages). Finally, we obtain the resource $\mathsf{OwnGlobalSnapshot}(\{\lfloor a \rfloor\})$, [9] which serves as proof that the returned message $\lfloor a \rfloor$ did not "come out of thin air": it was properly recorded in the global state. In general, owning a *global snapshot* $\mathsf{OwnGlobalSnapshot}(r)$ gives us a lower bound r on the set of all messages sent: if we own both $\mathsf{OwnGlobal}(h)$ and $\mathsf{OwnGlobalSnapshot}(r)$ we can conclude $r \subseteq h$. The $\mathsf{OwnGlobalSnapshot}(r)$ resource

[9]The notation $\lfloor a \rfloor$ stands for the *erasure* of a. This is a technical detail we inherited from the development in Gondelman et al. [Gon+21], because we represent local and global events differently. The erasure of a local event a gives us the corresponding global event $\lfloor a \rfloor$.

is persistent (Section 3.1), meaning that we can make copies of it freely; this makes snapshots useful as certificates that a certain message was broadcast by RCB.

Broadcast Figure 3.4 also shows the specification of broadcast. Intuitively, the effect of broadcast is to generate a new message, which in our framework needs to be recorded both as part of the global state as well as of the local state of the process calling broadcast. This is why in the precondition of broadcast we need to provide both $\mathsf{OwnGlobal}(h)$ and $\mathsf{OwnLocal}(i, s)$. The function then returns a local event w and its logical representation a, as evidenced by the predicate $\mathsf{IsLocEv}(w, a)$. A few points worth pointing out:

- Unlike in traditional implementations of RCB, where broadcast returns unit, our broadcast returns the generated message (or local event) corresponding to the broadcast value. For example, if replica i broadcasts the value 2, then $\mathsf{broadcast}(2)$ returns a tuple $(2, \text{vc}, i)$ for some vector clock vc that is globally maximal. In general, the return value is of the form $(\text{payload}, \text{vc}, \text{origin})$. This is why we call our implementation *tagged* RCB, as per Baquero et al. [BAS14].

- As expected, the newly generated message has not been previously recorded. This is given by $a \notin s$ and $\lfloor a \rfloor \notin h$.

- We obtain back resources $\mathsf{OwnGlobal}(h \cup \{\lfloor a \rfloor\})$ and $\mathsf{OwnLocal}(i, s \cup \{a\})$, showing that the event has been properly recorded both locally and globally.

Logical Atomicity The observant reader might have noticed two peculiar points about the specs above.

First, the broadcast spec requires the user to provide the global state resource $\mathsf{OwnGlobal}(h)$. Separation logic is all about modular specification, so a global resource that tracks all broadcast events would seem to be antithetical to separation logic. However, we find that the global resource is useful when reasoning about closed programs, because it allows us to state invariants of the form "all messages ever sent satisfy a safety property P" (e.g. in a system with two replicas, all messages are sent by one of the replicas).

A more practical concern is how to get two processes to concurrently broadcast messages, since it would seem that the broadcast spec requires exclusive ownership of $\mathsf{OwnGlobal}(h)$; it in fact does not. The reason is that our specs do not use regular Hoare triples, but instead rely on *logically-atomic triples* [Jun+15]. Instead of the regular $\{P\}\ e\ \{Q\}$ we write $\langle P \rangle\ e\ \langle Q \rangle \mathcal{N}$. The intuition is the following: if we can prove the atomic triple above, then e is evaluated until a certain step (its *linearization point* [HW90]) at which point P holds, possibly after opening any invariant that is not in the $\mathcal{N}$ namespace. After the atomic step, Q then holds, and all opened invariants need to be closed. So Q does not necessarily hold when the function terminates, but it always holds after the linearization point. The advantage of atomic triples is that we are allowed to open invariants when proving the precondition P. This is useful in the broadcast spec, because the global resource $\mathsf{OwnGlobal}(h)$ is likely to be kept in an Iris invariant by most clients of RCB (otherwise clients will not be able to concurrently broadcast messages). Our definition of atomic triples is adapted from that in Perennial [Cha+19].

Resource lemmas As mentioned in Section 3.2.1, in addition to the specs above and the resources that track messages, we proved a number of lemmas (e.g. causality) that serve as reasoning principles for using the resources. Because our treatment of causality is an adaptation of Gondelman et al. [Gon+21], the reader can consult that paper for the full list of resource lemmas.

Safety Properties We now show how RcbLib satisfies the three the safety properties presented in Section 3.3.

(RCB2) No duplication This property follows from the deliver spec (Figure 3.4); specifically, the postcondition guarantees that the delivered message a (if one exists), was not previously delivered to the same process ($\mathsf{OwnLocal}(i, s \cup \{a\}) * a \notin s$).

(RCB3) No creation We prove this as a property of local state resources:

$$\boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}} * \mathsf{OwnLocal}(i, s_i) * \mathsf{OwnLocal}(j, s_j) * e \in s_i * \mathit{origin}(e) = j \vdash \\ \Rrightarrow_{\mathcal{E}} \exists e'. e' \in s_j \land \lfloor e' \rfloor = \lfloor e \rfloor$$

Here, you can imagine i as the process that has just received message e. If i can assert that m originated in process j, and we also have knowledge of the local state of j in the form of $\mathsf{OwnLocal}(j, s_j)$, then the lemma guarantees that e is in fact also present in s_j (or, more precisely, that one can find messages in both local histories with equal erasures). The lemma above holds in the presence of a *global invariant* $\boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}}$ that RcbLib maintains to coordinate the local state resources of different replicas.

(RCB5) Causal delivery This is the main resource lemma, which was already informally described in Section 3.2.1. The full form also holds under the global invariant, and uses global snapshots instead of the full global state:

$$\boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}} * \mathsf{OwnLocal}(i, s) * \mathsf{OwnGlobalSnapshot}(h) \vdash \Rrightarrow_{\mathcal{E}} \forall a \in s, w \in h.\ vc(w) < vc(a) \Rightarrow \\ \exists a' \in s.\ \lfloor a' \rfloor = w$$

3.3.3 *Correctness Proof and Its Relationship to Gondelman et al. [Gon+21]*

As we mentioned in Section 3.2.1, our proof that RcbLib's implementation meets the specifications in Figure 3.4, as well as our proofs of the safety lemmas that follow under the global invariant, are based on the proof recipe outlined in Gondelman et al. [Gon+21]. Gondelman et al. [Gon+21] implement and specify a causally-consistent distributed key-value store, also within separation logic using Aneris. The proof recipe they outline (which we follow) can be summarized thus:

- First, model the distributed system as a state-transition system, where each state tracks the set of events at each replica. [10] Additionally, we track the global state of the system as the union of local events.
- Next, we embed the model in separation logic by using Aneris's ghost theory to create separation logic resources that represent knowledge of the local and global states.

[10]For them, an event is a write to the key-value store; for us, an event is a delivered message.

$$
\begin{aligned}
&\Sigma = \mathbb{N} \\
&\sigma_i^0 = 0 \\
&\mathsf{prepare}_i(\mathsf{inc}, n) = \mathsf{inc} \\
&\mathsf{effect}_i(\mathsf{inc}, n) = n + 1 \\
&\mathsf{eval}_i(\mathsf{rd}, n) = n
\end{aligned}
$$

Figure 3.5: Specification of op-based counter CRDT from Baquero et al. [BAS14].

For example, Gondelman et al. [Gon+21] construct a resource $\mathsf{Seen}(i, s)$ indicating that replica i has received *at least* the writes in s. Our analogous resource is $\mathsf{OwnLocal}(i, s)$, which captures the knowledge the replica i has delivered *exactly* the messages in s.

- Construct a global invariant (another proposition) that implies that the aforementioned resources describe reachable states in the state-transition system. For example, if we own $\mathsf{OwnLocal}(i, s)$, we can then conclude (provided the global invariant holds) that s is not an arbitrary set of messages, but instead satisfies certain safety properties (e.g. s is causally-closed, the origin field of messages is in the right range, etc.). This is also the step where we prove the resource laws (e.g. causality and no-creation).
- Finally, to verify the code running in each replica, establish a *lock invariant* [BB17] that tracks the set of events that have been processed by the replica so far. In doing so, one has to carefully pick the right (combination of) resource algebras (RAs) from which to draw the separation logic resources, so that the right properties hold and invariants can be preserved.

We were also able to reuse part of Gondelman et al. [Gon+21] Coq's development in our proof of RcbLib. To be clear, we do not claim the proof recipe above as our contribution. Instead, our contribution is producing for the first time modular specifications for a general-purpose library for causal broadcast. By contrast, Gondelman et al. [Gon+21] deal with causality specifically within the context of a key-value store. In addition, our implementation includes multiple techniques to improve reliability (e.g. sequence ids, acknowledgements, eager re-broadcasts) that are not present in Gondelman et al. [Gon+21]. See Section 3.6 for additional details.

3.4 *OpLib: a Library for Implementing CRDTs*

Figure 3.5 shows a specification for a counter CRDT[11] taken from Baquero et al. [BAS14]. This is not a separation-logic specification; instead, the counter is specified by instantiating several generic components: a set of states Σ (the naturals), an initial state (0), and a function effect that given a counter state and an operation returns the resulting state (the counter has only one kind of operation: add). [12] This style of specification is used throughout the CRDT

[11]Sometimes referred to as a *grow-only* or G-Counter, because the counter can only be incremented.

[12]The spec also shows two other functions: prepare which builds an "internal" operation from an "external", user-provided operation (this can often by taken to be just the identity); and eval which queries the CRDT's state.

literature [Sha+11a; BAS14; Bur+14] and it is a useful one because it allows us to focus on the parts of a CRDT that are truly unique to the CRDT in question. By contrast, the spec leaves many details unspecified: how are messages sent from one replica to others (some kind of broadcast), what happens when the current replica tries to update its state concurrently with a remote update being processed (we need locking), how is the replica state persisted across operations (mutable state). These details are common across different CRDTs, so it would be useful to factor their implementation into a separate library that can then be instantiated by CRDT implementer. This is what we have done with our OpLib library, which reuses our RCB implementation from Section 3.3 to provide the scaffolding for implementing op-based CRDTs.

3.4.1 Implementation

OpLib's code is shown in Figure 3.6. To use the library, the user calls oplib_init and provides serialization and deserialization functions (ser and deser) for the CRDT's operations, together with the addresses of replicas (addrs), the current replica id (rid) and most importantly the logic for the specific CRDT being implemented (crdt). The crdt value has the following polymorphic type:

```
type repIdTy = int  (* replica id *)
type 'opTy msgTy = ('opTy * vector_clock) * repIdTy
type ('opTy, 'stateTy) effectFnTy = 'opTy msgTy -> 'stateTy -> 'stateTy
type ('opTy, 'stateTy) crdtTy = 'stateTy * ('opTy, 'stateTy) effectFnTy
```

That is, as in Figure 3.5, a CRDT is specified by its initial state and an effect function that knows how to transition from a state to the next. Unlike Figure 3.5, however, we now have executable OCaml code instead of a high-level specification.

Going back to oplib_init, the function uses the RcbLib library to obtain a pair of functions for delivering (receiving) and broadcasting messages to other replicas. It then allocates a reference to store the CRDT state (starting with the initial state provided by the user) and then forks an apply_thread that listens for messages sent by remote replicas, so we can apply their effects. Finally, oplib_init returns a pair of functions (get_state, update) that the user can call to query the CRDT's state and update it, respectively.

The apply_thread function runs an infinite loop that first tries to deliver the next message in causal order (using RcbLib) and then, if one exists, updates the CRDT's state using the user-provided effect function.

Finally we have the user-facing functions get_state and update. The former returns a copy of CRDT's current state; the latter uses RCB to broadcast the new operation op to other replicas. RcbLib returns the user-provided operation wrapped with causality information (so an operation becomes a message); update then uses the newly-created message and the effect function to update the CRDT state.

Three points of note: first, effect is a pure function: given a state and a message it returns the resulting state. Second, concurrent accesses to the internal state (e.g. concurrent executions of apply_thread and update) are synchronized via a lock. Finally, notice that OpLib does not directly invoke any network operations (e.g. creating a network socket, sending a message, etc.); instead, all of the networking functionality is encapsulated in RcbLib.

```
let oplib_init ser dser addrs rid crdt =
  let res = rcb_init ser dser addrs rid in
  let (del, br) = res in
  let crdt_res = crdt () in
  let (init_st, eff) = crdt_res in
  let st = ref (init_st ()) in
  let lock = newlock () in
  fork (apply_thread lock del st) eff;
  (get_state lock st, update lock br st eff)

let get_state lock st () =
  acquire lock;
  let res = !st in
  release lock;
  res

let update lock br st effect op =
  acquire lock;
  let msg = br op in
  st := effect msg !st;
  release lock

let apply_thread lock del st eff =
  loop_forever (fun () ->
      acquire lock;
      begin
        match (del ()) with
          Some msg -> st := eff msg !st
        | None -> ()
      end;
      release lock;)
```

Figure 3.6: Code of OpLib library.

$$\{\,\mathsf{gcounter}(i,k)\}\ \langle ip_i;\ \mathtt{query}()\rangle\ \{m.\ k \le m * \mathsf{gcounter}(i,m)\} \qquad \textsc{QuerySpec}$$
$$\{\,\mathsf{gcounter}(i,k)\}\ \langle ip_i;\ \mathtt{incr}()\rangle\ \{().\ \exists m.\ k < m * \mathsf{gcounter}(i,m)\} \qquad \textsc{IncrSpec}$$

Figure 3.7: G-Counter specification from Timany et al. [Tim+21].

3.4.2 *Specification*

We start by arguing why, for CRDTs, resources like $\mathsf{LocSt}(i, \bullet s, \circ h)$ that track the set of executed operations are preferable to those that track the CRDT's state. Another way to say this is that CRDTs benefit from having *intensional*[13] specifications.

From counters to replicated counters Consider a simple counter module exposing two functions: $\mathsf{incr}()$ increases the counter's value by one, and $\mathsf{read}()$ returns the counter's current value. If used in a sequential setting, one can imagine being able to prove the following specifications: $\{\mathsf{c} \mapsto \mathsf{n}\}\ incr(c)\ \{c \mapsto n+1\}$ and $\{c \mapsto n\}\ \mathsf{query}()\ \{v.v = n\}$. Now we move to a concurrent or distributed setting, where the previous specs are still provable but no longer useful, because we need to be able to increment the counter concurrently. To solve this problem, we can track a lower bound of the counter's value, instead of the counter's exact value. Then every time we increment, we can increment the lower bound by one. This is precisely how Timany et al. [Tim+21] structure their specification of a G-Counter CRDT: they have a resource $\mathsf{gcounter}(i, m)$, meaning that at replica i the counter's value is *at least* m (Figure 3.7).

This works but has at least two drawbacks. First, the incr spec is unable to distinguish between a properly-implemented counter and one that increments the state by two instead of one every time incr is called. Second, even if we are able to fix the previous issue, perhaps by tracking "contributions" as in Birkedal and Bizjak [BB17], we face an even thornier problem if we consider not an increment-only counter, but one that additionally has a decrement operation. The problem there is what to write in incr's post-condition. Since the counter's state is no longer monotonic, if we start with a $\mathsf{gcounter}(i, m)$, we can end up with a $\mathsf{gcounter}(i, k)$ where k can be greater, equal, or less than m. We have lost all knowledge about the counter's state.

Consider what happens if instead of trying to track the counter's *state* we track the *operations* that the counter has processed. First, it makes sense to split said operations into those that are generated locally and the ones that come from other replicas. This is because a replica "knows" the operations *it* has performed, but it does not know what operations have been performed remotely until query or incr are called. Figure 3.8 show these new intensional specs. Ownership of the resource $\mathsf{gcounter}(i, s, h)$ conveys knowledge that at replica i we have processed *exactly* the operations in s and *at least* the operations in h. In this case an operation is a pair (inc, i) containing the operation type (we only have one kind of operation: inc) and the replica id. Logically, calling query involves trading our knowledge of $\mathsf{gcounter(i, s, h)}$ for knowledge of $\mathsf{gcounter}(i, s, h')$, where $h \subseteq h'$. That is, after calling query we might become aware of additional remote operations, but the set s of local operations does not change. By contrast, in calling incr we trade $\mathsf{gcounter}(i, s, h)$ by $\mathsf{gcounter}(i, s \cup \{(inc, i)\}, h')$ with $h \subseteq h'$. This means that after incr returns the set of local operations has grown by exactly one element

[13]In the sense of Roscoe [Ros96], as opposed to the more common *extensional* specifications that focus on the observable effects of operations.

$$\{\mathsf{gcounter}(i,s,h)\}\ \langle ip_i;\ \mathtt{query}()\rangle\ \{m.\ \exists h' \supseteq h.\ m = |s \cup h'| * \mathsf{gcounter}(i,s,h')\} \qquad \text{QuerySpec}$$
$$\{\mathsf{gcounter}(i,s,h)\}\ \langle ip_i;\ \mathtt{incr}()\rangle\ \{().\ \exists h' \supseteq h.\ \mathsf{gcounter}(i, s \cup \{(\mathsf{inc}, i)\}, h')\} \qquad \text{IncrSpec}$$

Figure 3.8: Intensional G-Counter specifications.

(as expected), and also new remote operations might have been processed as well. This specification style solves our problems because it allows us to track what the current thread's contribution is to the counter's state. It also scales well to handling a dec operation: the incr spec would not change; we would just need to adjust query's spec so that the result m is not just the number of recorded operations $|s \cup h'|$ but instead takes into account whether each operation is an inc or a dec.

Scaling up to CRDTs via denotations The idea of tracking operations as opposed to state (Figure 3.8) can be applied to specifying additional CRDTs in addition to the G-Counter. We just need two additional ingredients: first, abstract away the function that computes the CRDT's current state from the set of received operations (so instead of returning $|s \cup h'|$ in query we want $f(s \cup h')$ for some f). Second, when operations are not naturally commutative (for example, a replicated register that stores the "last" write) CRDTs use causality information to re-introduce commutativity. This is precisely what Burckhardt et al. [Bur+14] do with their notion of *operation contexts* which "include all we need to know about a[n] [...] execution to determine the return value of a given operation" Burckhardt et al. [Bur+14]; we will use the related notion of CRDT *denotations* from Leijnse et al. [LAB19]. The definitions below are implicitly parameterized by a given CRDT; specifically by its set of operations Op and states St.

Definition (Events)**.** The set of *events* is the product $\mathsf{Event} \triangleq \mathsf{Op} \times \mathsf{VC} \times \mathbb{N}$, where VC is the type of vector clocks and the third component denotes the originating replica id for the event. We lift the partial order of vector clocks to events.

Definition (Denotations)**.** A denotation $[\![\cdot]\!] : \mathcal{P}(\mathsf{Event}) \rightharpoonup \mathsf{St}$ is a partial function from sets of events to states.

As an example, the following is the denotation for a *multi-value register* CRDT. A multi-value register stores only concurrent writes; writes that come later in causal order replace earlier ones. The set Op of operations is just $\{\mathsf{write}(z) | z \in \mathbb{Z}\}$.

$$[\![s]\!]_{\mathsf{mv\text{-}reg}} = \{(w, vc) | \exists o.(\mathsf{write}(w), vc, o) \in s \land vc \in \mathrm{Maximals}(s)\}$$

A nice feature of denotations is that they support specifying higher-order CRDT *combinators*. For example, given denotations A and B, we can form their product (another denotation) $A \times B$, defined in Section 3.5.

We can give specifications for OpLib's get_state and update functions that are parametric on the denotations of the CRDT being implemented. These are shown in Figure 3.9.

GetStateSpec We use the get_state() function to query the CRDT's state. To verify a call to get_state(), we need to provide the local state resource $\mathsf{LocSt}(i, \bullet\, s, \circ\, h)$. When the call completes, we get back $\mathsf{LocSt}(i, \bullet\, s, \circ\, h')$ for some $h' \supseteq h$. That is, we now logically know

GetStateSpec

$$\langle \mathsf{LocSt}(i, \bullet s, \circ h) \rangle$$
$$\langle ip_i;\ \mathsf{get\_state}() \rangle$$
$$\left\langle \begin{array}{l} v.\ \exists h'\, w.\ h' \supseteq h * \mathsf{StCoh}(w, v) * \\ \quad \mathsf{LocSt}(i, \bullet s, \circ h') * [\![s \cup h']\!] = w \end{array} \right\rangle^{\mathcal{N}}$$

UpdateSpec

$$\langle \mathsf{LocSt}(i, \bullet s, \circ r) * \mathsf{GlobSt}(h) \rangle$$
$$\langle ip_i;\ \mathsf{update}(v) \rangle$$
$$\left\langle \begin{array}{l} ().\ \exists a\, r'.\ r' \supseteq r * a \notin s * a \notin h * \mathit{payload}(a) = v * \\ \quad \mathit{origin}(a) = i * a \in \mathrm{Maximals}(h \cup \{a\}) * \\ \quad a \in \mathrm{Maximum}(s \cup r' \cup \{a\}) * \\ \quad \mathsf{LocSt}(i, \bullet s \cup \{a\}, \circ r') * \mathsf{GlobSt}(h \cup \{a\}) \end{array} \right\rangle^{\mathcal{N}}$$

Figure 3.9: Logically-atomic specs for get_state and update where $\mathcal{N}$ must contain the global invariant's name.

that the CRDT has received additional remote operations (namely $h' \setminus h$), and that the local operations have not changed (because we were holding the local resource, of which there is only one copy per replica). The return value v of get_state is *coherent* with a logical representation of the state w; this is given by the predicate $\mathsf{StCoh}(w, v)$. We do this because the logical version of the state w might offer a "cleaner" representation of the state that is not polluted by the idiosyncrasies of AnerisLang's design, of which v is a value. For example, w might be a triple while AnerisLang only supports pairs, so w's encoding of v uses nested pairs. Finally, we know that the (logical version of the) return value is the denotation of the observed operations: $[\![s \cup h']\!] = w$.

UpdateSpec To update the CRDT, we call $\mathsf{update}(v)$, where v is some operation.[14] As a precondition, we must provide the local and global state resources, $\mathsf{LocSt}(i, \bullet s, \circ r)$ and $\mathsf{GlobSt}(h)$, respectively. The update function returns unit. We get back updated resources $\mathsf{LocSt}(i, \bullet s \cup \{a\}, \circ r')$ and $\mathsf{GlobSt}(h \cup \{a\})$; the latter is because around the linearization point exactly one event has been added to the entire system, namely the new event a containing the operation v. This new event originates at node i, and is maximal with respect to all other events in h, and the maximum of the (local) events in $s \cup r' \cup \{a\}$: this is just like in the broadcast spec in Figure 3.4. The new local resource $\mathsf{LocSt}(i, \bullet s \cup \{a\}, \circ r')$ indicates that we are now aware of exactly one additional local event a, as well as zero or more remote events $r' \supseteq r$. Finally, $a \notin h$, indicating that every update generates a new event.

Labelled-transition systems We have seen that denotations provide a high-level specification of a CRDT. The problem, however, is that denotations are too high-level. Specifically, the denotation has access to the entire set of operations performed on the data type, whereas in

[14]Notice when the user calls update they do not know what vector clock will be assigned to the operation; that happens internally once RCB broadcasts the message.

$$\begin{aligned}
\mathsf{VC} &= \text{representation of vector clocks} \\
\mathsf{RepID} &= \mathbb{N} \text{ (replica ids)} \\
\mathsf{St} &= \mathcal{P}(\mathbb{Z} \times \mathsf{VC}) \\
\mathsf{Event} &= \{\mathsf{write}(z) | z \in Z\} \times \mathsf{VC} \times \mathsf{RepID} \\
\mathsf{payload}(\mathsf{write}(z), \_, \_) &= z \\
\mathsf{orig}(\_, \_, r) &= r \\
\rightarrow &= \{(\mathsf{st}, \mathsf{ev}, \mathsf{st}') | \mathsf{st}' = (\mathsf{payload}(\mathsf{ev}), \mathsf{orig}(\mathsf{ev})) \cup \mathsf{filter}(\lambda e.e \geq \mathsf{ev})\mathsf{ev}\} \\
\sigma_0 &= \emptyset
\end{aligned}$$

Figure 3.10: Labelled-transition system for a multi-value register.

reality operations arrive one at a time (either from remote updates or due to local function calls). The solution is to give a second, lower-level specification for CRDTs, one that is closer to the running program. We do so using labelled-transition systems (LTS). Our LTS is a tuple (St, Event, $\rightarrow$, σ_0) containing the set St of (CRDT) states, the set Event of events which serve as labels (recall that events contain operations plus causality metadata), a (partial) transition function $\rightarrow$: St $\times$ Op $\rightharpoonup$ St, and an initial state σ_0.

Figure 3.10 shows a sample LTS for a multi-value register. A register state St is a set of pairs $\{(z, t)\}$ containing a value z written to the register together with a timestamp t (a vector clock) of when the write occurred. The transition labels Event are triples $(\mathsf{write}(z), t, r)$ containing a value z written, its timestamp t, and a replica id r of the process that issued the write. The transition relation $\mathsf{st} \xrightarrow{\mathsf{ev}} \mathsf{st}'$ is set up such that from a state st and given an event ev we can move to st′ if st′ consists of ev plus all elements of st that happened *concurrently* with ev. Finally, the initial state σ_0 is the empty set. Notice we assumed the new event ev does *not* happen before any of the writes already in st (that is, we assumed that $\forall e.e \in \mathsf{st} \implies ev \geq e$). This assumption is justified because OpLib is implemented using RcbLib, so we can assume that an operation's causal dependencies are delivered before the operation itself is, so that if $ev < e$ for some $e \in$ st, then ev $\in$ st (a contradiction).

We integrate labelled-transition systems into OpLib specs by defining *coherence* properties between (a) a denotation and the corresponding LTS and (b) the LTS and the effect function supplied by the CRDT implementer. The coherence properties are shown in Figure 3.11.

The coherence between denotation and its LTS is given by two requirements. First, the denotation of the empty set of events should be the initial LTS state σ_0. Second, if $[\![s]\!] = p$ and p steps to p' through a transition labelled e, we must have $[\![s \cup e]\!] = p'$. This last implication is weakened to hold only for new messages e that are *valid* with respect to the set of existing events s, written $\mathsf{Valid}(s, e)$.

The validity predicate encodes assumptions we can make about arriving messages because of guarantees provided by causal broadcast. That is, if $\mathsf{Valid}(s, e)$ holds, then $e \notin s$ (there are no duplicates), $e \in \mathrm{Maximals}(s \cup \{e\})$ (no already-delivered message causally depends on e), $\mathsf{EventsExt}(s \cup \{e\})$ (vector clocks uniquely identify messages) and $\mathsf{EventsTotal}(s \cup \{e\})$ (messages originating at the same replica can be totally ordered).

Finally, coherence between the LTS and the effect function is specified via a Hoare triple.

Validity of new messages

$$\begin{aligned}
\mathsf{Valid}(s,e) &\triangleq e \notin s \land e \in \mathrm{Maximals}(s \cup \{e\}) \land \mathsf{EventsExt}(s \cup \{e\}) \land \mathsf{EventsTotal}(s \cup \{e\}) \\
\mathsf{EventsExt}(s) &\triangleq \forall e\, e'. e \in s \land e' \in s \land \mathsf{vc}(e) = \mathsf{vc}(e') \implies e = e' \\
\mathsf{EventsTotal}(s) &\triangleq \forall e\, e'. e \in s \land e' \in s \land \mathit{origin}(e) = \mathit{origin}(e') \land e \neq e' \implies e < e' \lor e > e'
\end{aligned}$$

Coherence of denotation and LTS

$$\begin{aligned}
&[\![\emptyset]\!] = \sigma_0 \\
&\forall s\, p\, e\, p'. \mathsf{Valid}(s,e) \land \\
&\quad [\![s]\!] = p \land p \xrightarrow{e} p' \implies [\![s \cup e]\!] = p'
\end{aligned}$$

Coherence of LTS and effect function

EffectSpec

$$\begin{aligned}
&\left\{ \mathsf{StCoh}(s, st) \land \mathsf{EvCoh}(e, ev) \land [\![S]\!] = s \land \mathsf{Valid}(S, e) \quad \right\} \\
&\quad \langle ip_i;\ \mathsf{effect}(\mathsf{ev}, \mathsf{st}) \rangle \\
&\left\{ st'.\ \exists s'.\ \mathsf{StCoh}(s', st') \land s \xrightarrow{e} s' \quad \right\}
\end{aligned}$$

Figure 3.11: Coherence properties relating the denotation, labelled-transition system, and effect function.

[15] The spec says that if we are to execute effect(ev, st), then we must know that ev and st are coherent with their logical counterparts s and e, respectively. Additionally, there must be some set of events S such that $[\![S]\!] = s$ and e must be a valid new message with respect to S (so $\mathsf{Valid}(S, e)$). If that is the case, then if effect terminates it will return a new physical state st′ such that $s \xrightarrow{e} s'$, where s' is the logical view of st'. That is, the spec says that if we step from st to st' via ev using effect in the physical world, then we can step from s to s' via e using the LTS in the logical world.

Library interface As shown in Figure 3.6, a user of OpLib starts by calling oplib_init with a number of arguments. One of them, named crdt in Figure 3.6, is a pair (init_st, effect) consisting of the data type's initial state and its effect function, respectively. The initial state must be coherent with the LTS's initial state σ_0, so $\mathsf{StCoh}(\sigma_0, \mathsf{init\_st})$, and the effect function must satisfy EffectSpec from Figure 3.11. When oplib_init returns, it gives back a pair of functions (get_state, update) that satisfy GetStateSpec and UpdateSpec from Figure 3.9.

3.4.3 *Correctness Proof*

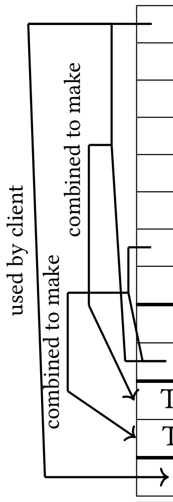
The core of OpLib's correctness proof is a *lock invariant* [BB17] asserting that the CRDT's internal state equals the denotation of the set of operations processed so far. The logical resources needed to enforce this invariant are divided across three areas of responsibility: first, a global invariant tracks the set of messages sent by all replicas, as well as the per-replica delivered messages. This global invariant also asserts that messages are sent via the RCB protocol, allowing us to inherit all resource-related lemmas from Section 3.3 (e.g. causal delivery). Next, the aforementioned lock invariant also tracks the messages delivered by a specific replica; the messages are divided in two groups: local and remote. Finally, we have the user-resources such as $\mathsf{LocSt}(i, \bullet\, s, \circ\, h)$ that are useful for verifying client programs. We use a number of resource

[15]Notice that, unlike the spec for update and get_state, the effect spec is given by a regular Hoare triple, as opposed to a logically-atomic triple. This is because effect only manipulates pure propositions and does not require any exclusive resources that need to be stored in invariants.

Table 3.2: Library metrics (lines of code).

Library	**OCaml**	**Coq Spec**	**Coq Proof**
RcbLib	196	2151	2703
OpLib	86	1352	2224
total	282	3503	4927

Table 3.3: CRDTs implemented on top of OpLib (lines of code).



CRDT	**OCaml**	**Coq Spec**	**Coq Proof**	
Positive-Negative Counter	25	88	108	Simple CRDTs
Grown-only Counter	26	88	116	
Two-Part Set	25	80	73	
Add-Wins Set	34	103	228	
Remove-Wins Set	53	99	386	
Grow-Only Set	22	74	57	
Last-Writer-Wins Register	54	136	365	
Multi-Value Register	35	93	195	
Product Combinator	30	148	187	Combinators
Map Combinator	34	153	340	
Table of Positive-Negative Counters	22	29	38	Compound CRDTs
Table of Last-Writer-Wins Registers	22	38	39	
Closed Example	17	287	99	
total	399	1416	2231	

algebras, including Timany and Birkedal [TB21]'s monotone construction, to carefully coordinate these different logical resources: for example, to prove that ownership of $\mathsf{LocSt}(i, \bullet\, s, \circ\, h)$ really does grant precise knowledge of the set of delivered local messages s, but only partial knowledge of the remotely-delivered messages h.

We refer the reader to our Coq development for full details on the proof. Table 3.2 shows the number of lines of OCaml and Coq code needed to implement and verify both RcbLib and OpLib.

3.5 Implementing CRDTs

In order to put OpLib to test we have implemented twelve CRDTs using this library. These CRDTs consist of eight simple CRDTs, two CRDT combinators, and two compound CRDTs which apply the map combinator to one of the simple CRDTs. Below, we will briefly explain these examples and discuss and summarize what is depicted in Table 3.3. The relatively low number of lines of code required to implement (in OCaml) and verify the CRDTs enumerated in Figure 3.3 shows the usefulness and success of our methodology of building CRDTs on top of the RcbLib and OpLib libraries. Moreover, as we will discuss below, the most intricate CRDTs in Table 3.3, *i.e.*, the last two rows, are those with smallest implementation and verification codes thanks to our compositional approach using CRDT combinators.

Counters We have implemented two counter variants: Grow-only Counter which can only have non-negative values and can only be incremented, and Positive-Negative Counter which can be both incremented and decremented. These two CRDTs are the simplest examples we have implemented. Part of the size of the Coq code is caused by having to show that the operations are commutative and associative; basic arithmetic facts which nonetheless need to be established formally in Coq.

Sets The only operation of the Grow-Only Set CRDT allows adding an element to the set. The Add-Wins Set and Remove-Wins Set CRDTs on the other hand support both adding elements and removing elements. They treat the removal operation differently though. The issue with the removal operation is that it causes ambiguity in case of concurrent operations which add and remove the same element. The Add-Wins Set and Remove-Wins Set, as their names indicate, resolve this ambiguity in favour of addition and removal respectively. Despite their apparent similarity these two CRDTs are conceptually different as can be seen in the difference in the number of lines of Coq code required to prove their correctness. The difference is that for the Add-Wins Set CRDT we only remember the additions in the local state. When we receive a removal operation we simply remove any element that was added *strictly* before that removal operation. This makes sense as an addition that is received after a removal can never be affected by it — in worst case, it is an addition concurrent with a removal which by definition wins. On the other hand, in the Remove-Wins Set CRDT we also need to track all remove operations in the local state of each replica as additional operations received after a removal operation can be invalidated by that removal operation. The Two-Part Set CRDT is conceptually simply obtained by gluing two Grow-Only Set CRDTs together. It tracks two sets and operations can add elements to either set. In practice, this CRDT could be obtained by combining the Grow-Only Set CRDT with the Map Combinator as a map with the domain being a fixed set of two elements (see below). However, we chose to implement this CRDT as a yet another simple example from scratch. All set CRDTs are parameterized by the collection of elements that can be stored in sets. In the OCaml code this means that the code is parameterized by a type variable for the type of elements of the set. It is only required that these elements can be serialized as we need to communicate them over the network.

Registers We have implemented two simple registers: a Multi-Valued Register and a Last-Writer-Wins Register. Just like sets these CRDTs are also parameterized by the collection of values that can be stored in these registers. The difference between these two CRDTs is the way they handle the issue of concurrent write operations. The Multi-Valued Register simply collects all possible values (time-wise maximal write operations) and presents them to the user of the register along with their corresponding time-stamp. The idea is that the user will have the authority to disambiguate the situation. The Last-Writer-Wins Register on the other hand considers the latest write in the set of maximal concurrent writes and considers that to be the valid value of the register. The concurrent nature of the events in our settings means that this method of disambiguation is not always viable. After all, concurrent events can be observed in different orders by different replicas. To obtain a complete disambiguation strategy the Last-Writer-Wins Register considers the latest write from the replica with the highest replica id to prevail.

Combinators We have implemented two CRDT combinators: the Product Combinator and the Map Combinator. The Product Combinator takes two CRDTs and constructs a CRDT where the state is the product of two states. The operations of Product CRDT are pairs of operations which take effect component-wise. The Map Combinator is a versatile combinator which takes a CRDT and constructs the CRDT of finite maps, *i.e.*, tables, of that CRDT. The state of the Map CRDT is a map with keys ranging over strings and value ranging over the states of the given CRDT. An operation is a pair of a string, the key to which the operation applies, together with an operation of the underlying CRDT. The map is initially empty. Every time an operation is received for a key that does not exist in the map it is first initialized with the initial state of the given CRDT before the operation is applied to it.

Compound CRDTS As illustrative examples we have implemented two compound CRDTs. Both of these examples use the Map Combinator. One makes a table of Last-Writer-Wins Registers while the other makes a table of Positive-Negative Counters. The fact that these relatively complicated CRDTs can be constructed and proven correct with very little effort is excellent evidence for the success of our methodology. We obtain full-functional correctness of these essentially databases, albeit with single-column tables, in under 50 lines of Coq code including the boilerplate for including necessary Coq libraries, *etc.*

A concrete closed example As a minimal smoke test for our OpLib library we prove safety (*i.e.* the program does not crash) of a simple example program. More precisely, using the so-called adequacy theorem of Aneris, we obtain that when this program is executed, as per the operational semantics of AnerisLang, it does not get stuck. This example initializes two replicas of Positive-Negative Counters with initial state 0. The first replica adds 1 to the counter and the second replica adds 2. They both proceed to read the value of the counter after adding to it. Intuitively, we expect the first replica to either read 1 or 3 while the second replica could read 2 or 3; and this is what both replicas *assert* as their last operation. This makes sense as each replica will definitely observe its own performed operation but might or might not have observed the operation performed by the other replica when it reads the counter. The assert command in AnerisLang is designed to evaluate its boolean and ignore it if it evaluates to true (it returns unit) and crash otherwise; hence showing safety of the example does indeed establish that the result each replica obtains when reading the counter is as expected. Intuitively, to establish this property, we simply need to enforce, using an invariant, that the global state (tracked using the proposition $\mathsf{GlobSt}(h)$) has at most two operations in it: an addition of 1 to the counter originating in the first replica and an addition of 2 originating from the second replica. Therefore, each replica by knowing its own local state (tracked using the proposition $\mathsf{LocSt}(i, \bullet\, s, \circ\, r)$), and using the relation between the global and local states of CRDTs, can conclude that the value read is the expected one.

3.6 Related Work

The literature on verification of CRDTs has grown over the years to produce many different approaches. In order to place our work within the mosaic of existing logics and tools, we identify several design criteria that help us build a taxonomy of the sub-field. For each criterion, we propose a concrete question that helps us classify each of the pieces of related work according to the criterion. Table 3.4 lists our proposed criteria and how to identify whether a paper

Table 3.4: Classification criteria for CRDT verification techniques.

Criterion	**Question**
Target	Does the technique target op-based or state-based CRDTs? Most verification efforts target one of the two kinds of CRDT, but not both.
Implementation	Does the paper claim to automatically produce executable code?
Convergence	Can the technique prove convergence [Sha+11b]? Convergence means that if two replicas have received the same set of events, then they are in equivalent states.
Eventual Delivery	Can the technique prove eventual delivery [Sha+11b]? Eventual delivery means that an update delivered to a correct replica eventually reaches all correct replicas. This is a liveness property.
Causality	If required, does the paper show that messages are delivered in causal order? The alternative is either that causal delivery is not required for the specific CRDT implemented, or it is required but is then assumed.
Functional Correctness	Can the technique prove functional correctness? That is, are there specifications that show how the outputs of CRDT operations depend on their inputs?
Modularity	Does the paper show an example of a client that uses the CRDT's specification to verify some property? For example, given a G-Counter CRDT can we show that if a replica calls inc twice the counter's value is at least two?
Mechanization	Are the proofs mechanized in a proof assistant?

meets each of them. Table 3.5 then looks at whether related work meets each criterion. Some works do not fit neatly in their assigned classes, nor do we argue that our choice of questions is canonical. We nevertheless think that posing concrete questions leads to a classification that is imperfect but useful.

We structure our discussion of related work around Table 3.5. Most techniques target either state-based CRDTs or op-based CRDTs, but not both. The exception is Burckhardt et al. [Bur+14], which focuses mostly on specifying both kinds of CRDTs via denotations, but not on verification.

Verification of state-based CRDTs As explained in Burckhardt et al. [Bur+14] state-based approaches guarantee causal delivery "for free". This is because communicating an entire state is (logically) equivalent to sending an operation together with all its causal dependencies.

The only modular state-based approach that we are aware of is Timany et al. [Tim+21]. This is also the only related work that proves eventual delivery. Like us, Timany et al. [Tim+21] use the Aneris separation logic; however, unlike us they only verify one example CRDT (a G-Counter), and their specification style (which tracks states as opposed to sets of operations) is less expressive than ours (see Section 3.4). To prove liveness, Timany et al. [Tim+21] develop an extension to the Iris program logic framework called Trillium, which allows them to show that the CRDT implementation refines a state-transition system. It would be interesting to restructure our development to use Trillium, since we already show that our CRDT implementations implement a labelled-transition system. Finally, Timany et al. [Tim+21] focus on (one) state-based CRDT, whereas we verify multiple op-based CRDTs (see Section 3.7 for a

Table 3.5: Comparison of different CRDT verification techniques. “Event. Del.” stands for eventual delivery, and “F.C.” for functional correctness.

Paper	Target	Implementation	Convergence	Event. Del.	Causality	F.C.	Modularity	Mechanization
Burckhardt et al. [Bur+14]	both		✓			✓		
Zeller et al. [ZBP14]	state		✓		free	✓		✓
Nair et al. [NPS20]	state		✓[16]		free			
Timany et al. [Tim+21]	state	✓	✓	✓	free	✓	✓	✓
Gomes et al. [Gom+17]	op	✓	✓					✓
Liu et al. [Liu+20]	op	✓	✓		✓	✓		✓
Liang and Feng [LF21]	op		✓			✓	✓	
Nagar and Jagannathan [NJ19]	op		✓					✓
this work	op	✓	✓		✓	✓	✓	✓

[16] In addition to convergence, Nair et al. [NPS20] prove other safety properties for specific CRDTs.

discussion of how our approach could be extended to the state-based setting).

Verification of op-based CRDTs Liu et al. [Liu+20] extend Liquid Haskell [Vaz+14] by annotating typeclass declarations with refinement types. Their system can later typecheck typeclass instances against the declarations. As a case study, they define a typeclass for op-based CRDTs and implement several instances, including a map combinator similar to ours. Instances of the CRDT class enjoy a *strong convergence* property that says that certain allowed permutations of a set of operations lead to the same final state. Additionally, they show functional correctness of their multi-set implementation by a simulation argument with respect to an abstract denotational specification (similarly to how we use denotations). They design their CRDTs so that they do not have to assume causal delivery, although in the process they do end up implementing parts of a causal broadcast algorithm (e.g. a delay queue).

The main difference between Liu et al. [Liu+20] and our work is how modular the approaches are. In Liu et al. [Liu+20] there does not seem to be a way to use strong convergence to verify a client program that uses a CRDT. [17] By contrast, as shown in Section 3.5 we can use our separation logic resources that track local and global states not only to show convergence and functional correctness, but also to verify clients. Additionally, we were able to verify causal broadcast as a general purpose library which is then re-used by our CRDT library. In their work, only the "business-logic" part of the CRDT is verified: that is their CRDTs are purely-functional data structures that are unaware of the existence of other replicas. By contrast, we verify not only the purely-functional part of a CRDT, but also all of its logic all the way through to network operations.

Liang and Feng [LF21] introduce the first technique that produces modular specifications for op-based CRDTs. Specifically, they strengthen SEC to arrive at a trace property called *Abstract Converging Consistency* (ACC), which combines SEC with functional correctness. Functional correctness is obtained by relating a concrete CRDT model Π to its abstract counterpart Γ. In proving the relation, one is allowed to re-order certain abstract (commutative) operations that satisfy an *arbitration* relation $\bowtie$. Once we prove ACC, an *abstraction* theorem gives us contextual refinement: meaning that in every program we can substitute the concrete CRDT by the abstract one (its spec) and still obtain the same results. The paper then introduces a rely-guarantee style logic to prove specification for clients using the CRDT.

Our work differs from Liang and Feng [LF21]'s in several aspects. First, their CRDTs, including the concrete variants, are closer to what we would call specifications and not executable implementations. This is because they represent CRDTs as collections of functions that go from state to state via operations (this is very similar to our LTS-based models). In contrast, our CRDT implementations are written in OCaml and so must deal with many details associated with running code: e.g. message serialization, network sockets, node-local concurrency, and mutation. Second, when proving functional correctness of a client in their system one proves a judgment of the form $\vdash \{P\}$ with $(\Gamma, \bowtie)$ do $C_1 || \ldots || C_n \{Q\}$. That is, the existence of the CRDT is baked into the top-level term that one reasons about, and the CRDT Γ is distinguished from the clients C_i. By contrast, in our setting the CRDT and client code are both written in AnerisLang, and our reasoning principles come in the form of standard separation-logic resources (e.g. $\mathsf{LocSt}(i, \bullet s, \circ h)$). We expect that our approach makes it

[17]This is backed by the fact that they do not verify the client applications (a text editor and event calendar) that use their CRDTs.

easier to re-use a verified CRDT as a component of a larger system; for example, we were able to create and use CRDT combinators.

Nagar and Jagannathan [NJ19] present a framework for automated verification of op-based CRDTs, as well as multiple examples of verified CRDTs. Importantly, their technique is parametric on the (axiomatized) consistency model afforded by the underlying communication protocol, so that the same CRDT implementation can be verified under e.g. eventual consistency and causal consistency.

There are multiple differences between this paper and our work. Their tool targets automated verification of high-level CRDT implementations (labelled-transition systems), while we do interactive verification of low-level OCaml-like code (including concurrency, mutation, higher-order functions, serialization of network messages, etc.). Additionally, the property they verify is convergence, while we verify convergence and functional correctness. Another difference is that their technique is parametric on a consistency model. By contrast, we fix causal consistency as the guarantee of our broadcast protocol. However, instead of axiomatizing the consistency guarantees, we implement and verify a general purpose library for causal broadcast. Finally, their verification engine is built specifically for (automated) verification of op-based CRDTs. By contrast, we conduct our proofs in a vanilla distributed separation logic (Aneris), using standard features like invariants and ghost state. This means that we are able to compose our proofs and specs with other verification efforts that do not target CRDTs. For example, we composed the proofs of our CRDT library with those of the causal broadcast library.

Verified causally-consistent key-value store Gondelman et al. [Gon+21] implement and verify a causally-consistent distributed key-value store, also using Aneris. Even though the term "CRDT" does not appear in their paper, they implement and model causal delivery of key-value store operations, and more generally their key-value store is very close to being a CRDT. It is not because it violates SEC: in certain execution traces, we can end up with replicas that have received the same set of writes, yet the same key is mapped to different values. This is because their tie-breaking mechanism for concurrent writes is to take the write that arrives later, which is sensitive to network delays. Additionally, their db replicas do not re-transmit dropped messages, and they do not re-broadcast messages. As mentioned in Section 3.3 we adapt Gondelman et al. [Gon+21]'s modelling of causality in separation logic so that it is applicable to a general-purpose RCB protocol. In fact, our table-of-registers example from Table 3.3 is also a key-value store where the above reliability issues are addressed. Our work can be then seen as generalizing the approach in their paper to apply to a wide range of CRDTs, as opposed to a bespoke key-value store.

Verified causal broadcast Redmond et al. [Red+22] implement and verify a library for causal broadcast in Liquid Haskell. Specifically, they define a predicate on library states called *process local causal delivery* (PLCD). Their main result is a theorem stating that PLCD is preserved by arbitrary sequences of the three library operation: receive, deliver, and broadcast. They then shows that if every process satisfies PLCD then the entire system satisfies a (global) definition of causal delivery. This is done at the model level, with the entire system modelled as an STS. The paper then uses the verified library to build an (unverified) replicated key-value store. The store code is responsible for network operations, concurrent access to library state via the

STM monad, and (de)serialization of messages. The authors evaluate the key-value store by load-testing it with multiple replicas and clients.

The main difference between Redmond et al. [Red+22] and our work is the scope of the verified components. Redmond et al. [Red+22] verify that applying a sequence of library operations starting from an initial empty state preserves PLCD. These operations are pure functions without side effects; network operations and concurrency are instead handled by an unverified library client (their key-value store). By contrast, we verify both the state manipulation functions and also their wrapper code that performs network operations, as well as concurrency control. Another key difference is that in our work we use the verified causal broadcast library as a building block over which to implement and verify our CRDT library, showing that our approach is modular. Redmond et al. [Red+22] also implement clients on top of their causal broadcast library, but their clients are unverified. Finally, Redmond et al. [Red+22] present a performance evaluation of their causal broadcast library, whereas we have not evaluated ours.

3.7 Conclusions

We have verified implementations of multiple op-based CRDTs in separation logic. We structured our development as a collection of libraries. First, we verified an RcbLib library for reliable causal broadcast. On top of RcbLib we then verified an OpLib library for building op-based CRDTs. CRDT implementers can use OpLib to specify their CRDTs as purely-functional data structures, without having to worry about low-level implementation details such as network operations and concurrency control. Finally, using OpLib we verified multiple CRDT instances: some are naturally commutative, while others use causality information and metadata to make operations commutative. That we were able to handle different kinds of CRDTs, including higher-order combinators, shows the applicability of our technique to a variety of scenarios. Our approach both can verify realistic implementations (as opposed to high-level protocols) and is modular (we can verify components in isolation and put their proofs back together to obtain verified stacks of components).

Future work: state-based CRDTs A natural question is whether our techniques could be adapted to verify state-based CRDTs. More precisely, whether local and global resources can track operations in that setting even though the entire CRDT state is sent between replicas, as opposed to sending one operation at a time. We think the answer is yes; the insight is that when two CRDT states, which are lattice elements, are merged by taking their least upper bound, the operations that generated those states can also be merged. [18] That is, logically we can also take the least upper bound in the powerset lattice of operations, so that if we are in a state $\mathsf{LocSt}(i, \bullet\, s, \circ\, h)$ and a foreign replica sends their state that resulted from operations coming from a set q, then we can update our state to $\mathsf{LocSt}(i, \bullet\, s, \circ\, h \cup q')$, where $q' = \{e \in q | origin(e) \neq i\}$. The goal of this line of work would be to produce a common specification style for both kinds of CRDTs, so that clients can use a CRDT without worrying about which implementation strategy was used.

Retrospective on our proofs The RcbLib project was sparked by the observation that we can disentangle the logical account of causality in Gondelman et al.'s work from the application

[18]This idea appeared in Burckhardt et al. [Bur+14]; the challenge would be to adapt it to separation logic so we can verify implementations modularly.

domain of their paper (a distributed key-value store). The main challenge was then to use a similar flavour of separation logic resources to verify a general-purpose reliable causal broadcast library. In particular, to provide reliability RcbLib deploys a number of techniques (sequence ids, acknowledgements, retransmissions) not present in Gondelman et al.'s work that complicate the proofs.

The challenge in verifying OpLib was in adapting the notion of CRDT denotation to a separation logic setting. In particular, it was challenging to construct the $\mathsf{LocSt}(i, \bullet\, s, \circ\, r)$ resource that tracks local events precisely and remote events loosely. From $\mathsf{LocSt}(i, \bullet\, s, \circ\, r)$ we can connect the return value of get_state to the denotation of the set of delivered local events.

What worked well: our Coq formalization makes extensive use of typeclasses to provide clean interfaces between each of the components (e.g. there are typeclasses for denotations, the LTS model, and the local and global resources). Additionally, in order to make the verification of our examples manageable we had to structure OpLib so that it would abstract away as much as possible from a CRDT implementation. This worked well; for instance, one of the authors verified most of the example CRDTs without having prior involvement in the verification of OpLib (they relied solely on OpLib's specification). Once the OpLib interface was ironed out, we were able to verify OpLib and its clients in parallel.

What did not work as well: it was more challenging than we expected to connect OpLib's logical resources to their RcbLib counterparts. This is necessary so OpLib can inherit all the resource laws that RcbLib provides (e.g. so we can rely on causality when reasoning about CRDTs). Establishing the connection was trickier than we thought because RcbLib's events have untyped payloads (we make no assumptions about the contents of broadcast messages), while OpLib uses typed payloads (for a specific CRDT we know the shape of its operations). This kind of impedance mismatch required proving many additional auxiliary lemmas, thus increasing OpLib's proof effort.

4 *Modular Verification of State-Based CRDTs in Separation Logic*

This chapter is a copy of the ECOOP'23 paper "Modular Verification of State-Based CRDTs in Separation Logic" [Nie+23], of which I was the lead author. The text and figures have been lightly edited for formatting.

Abstract

Conflict-free Replicated Data Types (CRDTs) are a class of distributed data structures that are highly-available and weakly consistent. The CRDT taxonomy is further divided into two subclasses: state-based and operation-based (op-based). Recent prior work showed how to use separation logic to verify convergence and functional correctness of op-based CRDTs while (a) verifying implementations (as opposed to high-level protocols), (b) giving high level specifications that abstract from low-level implementation details, and (c) providing specifications that are modular (i.e. allow client code to use the CRDT like an abstract data type). We extend this separation logic approach to verification of CRDTs to handle state-based CRDTs, while respecting the desiderata (a)–(c). The key idea is to track the state of a CRDT as a function of the set of operations that produced that state. Using the observation that state-based CRDTs are automatically causally-consistent, we obtain CRDT specifications that are agnostic to whether a CRDT is state- or op-based. When taken together with prior work, our technique thus provides a unified approach to specification and verification of op- and state-based CRDTs. We have tested our approach by verifying StateLib, a library for building state-based CRDTs. Using StateLib, we have further verified convergence and functional correctness of multiple example CRDTs from the literature. Our proofs are written in the Aneris distributed separation logic and are mechanized in Coq.

Conflict-free Replicated Data Types (CRDTs) are a class of distributed data structures that trade off strong consistency in favour of high availability. That is, local updates are not blocked by inter-replica synchronization; instead, they are immediately applied locally, and then propagated to other replicas. Because of the lack of synchronization, CRDTs need a mechanism for resolving conflicting updates: a typical strategy is to make all updates commutative, so that they can be applied in any order.

There are two main implementation strategies for CRDTs, differing in how updates are propagated. *Operation-based* (op-based) CRDTs propagate local updates by reifying updates as operations, which are then transmitted to other nodes. Once these (now remote) operations are received by other replicas, they can be applied to their local states so they can "catch up". By contrast, in *state-based* CRDTs, an update is first applied to the local state, and then the *state* is propagated to other replicas. This is achieved by letting the state be an element of a join-semilattice, constraining updates to be monotonic, and combining local and remote states via the lattice's join operator. The choice of implementation style for a CRDT (op vs state-based) incurs several trade-offs. Op-based based CRDTs are conceptually simpler, but make assumptions about the underlying delivery mechanism (e.g. at most once delivery). By contrast, the state-based approach can easily cope with duplicates and messages delivered out of order, because the merge operation (modelled with joins) is idempotent, associative, and commutative. On other hand, not only must the datatype semantics be encoded via joins, but also sending the entire state across the network might be inefficient.

Figure 4.1 shows a *grow-only counter* (g-counter) CRDT implemented in both styles. A g-counter is a datatype with two operations: it can be read and it can be incremented by a non-negative number. The op-based implementation defines the counter's initial state (0), an `effect` function that adds the value we are incrementing by to the counter's current state, and a `read` function that is just the identity. An *event* is a tuple containing an operation (the value to increment by) plus metadata, including the replica id where the operation originated. The state-based g-counter is more complex. The counter's state is kept as a list of integers tracking each replica's contribution to the counter's value. The initial state is the list of all zeroes with size `numRep`, the number of replicas. A `mutator` function takes the current state and the increment value, and returns the updated list where the right entry, as determined by the operation's origin, was incremented. The `merge` function takes two states and computes their join in the underlying lattice. For the g-counter, we take the pointwise maximum of the two lists. Finally, to read the value of the g-counter we just sum all entries in the state list. These purely functional implementations capture the g-counter's core logic, but do not show how events are propagated between replicas.

The standard consistency model for CRDTs is *Strong Eventual Consistency* (SEC). SEC can, in turn, be divided into two sub-properties: *convergence* (two replicas that have processed the same set of updates must be in the same state) and *eventual delivery* (any update sent by a replica will eventually be delivered to all other replicas).

Additionally, the guarantees of SEC are sometimes strengthened to imply *causal consistency* [BSS91], meaning that the causal order of updates is respected. In other words, given updates u and w, if u *happened before* w at a replica [Lam78], then u must be processed before w at all other replicas. Causal consistency captures programmer's intuitions on how the order of operations should be preserved; for example, it implies that reads are monotonic: a read always returns data that is "fresher" than what previous reads have returned.

4.0.1 Denotational Specifications

Specifying CRDTs is tricky because of their replicated nature and relaxed consistency model. Some works adopt SEC as the correctness criteria and do not provide functional correctness specifications of CRDTs [NPS20; Gom+17; NJ19]. Eventual consistency is a key correctness property, but this approach has at least two (related) shortcomings. First, given, e.g., a g-counter implementation, proving SEC shows that different replicas eventually converge, but does not tell us what they converge to. This means we cannot rule out bugs such as subtracting instead of adding in the definition of `effect` in Figure 4.1. Another problem is that by focusing on SEC we cannot abstract away from implementation details. For example, Figure 4.1 shows two g-counter implementations that use different techniques, but we should be able to reason

```
(* op-based g-counter *)
let init = 0

let effect st e =
  let (op, _, _) = e in
  st + op

let read st = st

(* state-based g-counter *)
let init = List.init numRep (fun _ -> 0)

let mutator st e =
  let (op, src) = e in
  List.mapi (fun i c ->
    if i = src then c + op else c)

let merge st1 st2 =
  let max = fun p ->
    Int.max (fst p) (snd p) in
  List.map max (List.combine st1 st2)

let read = List.fold_left (+) 0
```

Figure 4.1: Op-based and state-based OCaml implementations of a grow-only counter. `numRep` is the number of replicas.

about a g-counter as an *abstract data type* [LZ74], without worrying about whether it is op-based or state-based.

Burckhardt et al. [Bur+14] were the first to give functional correctness specifications of CRDTs. Their key observation is that just like a sequential data type (e.g., a queue) can be specified as a partial function from a list of operations to a (final) state, a replicated data type can be specified as a partial function from a *set of events* to a state. As in Figure 4.1, an *event* contains an operation plus additional metadata, including the id of the replica that created the operation and a timestamp. The timestamps induce a partial or total order on the set of events and, furthermore, that order is consistent with causality.[1]

We call this partial function from sets of events to the CRDT's state after processing the events a *denotation*.[2] For example, the denotation for a g-counter is $[\![s]\!] = \sum_{e \in s} e.o$, where s is a set of events and $e.o$ extracts e's operation (the value to increment by). In this particular case we do not use the event metadata to specify the g-counter, but we do so for other, more complex, CRDTs where all operations do not naturally commute. Note that any CRDT specification that uses denotations trivially satisfies the convergence part of SEC, because $[\![\cdot]\!]$

[1]In Burckhardt et al. Burckhardt et al. [Bur+14] a *visibility* relation is used to order events, instead of a timestamp.

[2]The term is due to Leijnse et al. [LAB19], who recast Burckhardt et al.'s formalism in a style more amenable to specifying CRDT combinators.

is insensitive to the order in which events arrive at different replicas: if they have received the same set of events, then their states will be the same. Also notice that denotations are by nature closer to the informal op-based specification in Figure 4.1 than to the state-based one. This is because op-based CRDTs are framed in terms of individual operations.

4.0.2 Verifying with Denotations

The papers by Burckhardt et al. [Bur+14] and Leijnse et al. [LAB19] are concerned with specifying CRDTs, but there is still a need for a mechanism that ties the high-level specifications, given in terms of denotations, to executable code written using features of a modern programming language: e.g., mutation, node-local concurrency, and higher-order functions. The recent work of Nieto et al. [Nie+22] connects denotations to low-level CRDT implementations using the Aneris distributed separation logic [Kro+20]. Specifically, they show how to build separation logic propositions that track the local state of a CRDT, where, e.g., the return value of a read is then given by a denotation of the local state. Nieto et al. demonstrate their approach by verifying a library for building op-based CRDTs: the library user (the CRDT implementer) instantiates the library with a purely-functional implementation of the CRDT (similar to the op-based example in Figure 4.1), and obtain in return a replicated data type. The library handles network operations, concurrency control, and mutation of local state. Nieto et al. exclusively reason about operation-based CRDTs. As future work, the authors include a high-level sketch of how their techniques might be adapted to the state-based setting.

There is then, to the best of our knowledge, an unexplored gap in the literature for verifying functional correctness of *state*-based CRDTs using modular specifications.

Related to the last point, existing approaches to verifying CRDTs target either op-based [Gom+17; Liu+20; LF21; NJ19; Nie+22] or state-based [Tim+21; NPS20; ZBP14] CRDTs, but never both kinds. This is important because it means that it is not possible to give the *same* specification to two implementations of the same replicated data type, where each uses a different implementation strategy (as in Figure 4.1). Having specifications that hide away implementation details is something we take for granted for sequential data types (e.g. a set abstract data type can be implemented both via a linked list and a hash table, but both implementations can be given the same specification). It would be useful to have the same hold for CRDTs.

4.0.3 Contributions

We fill the gaps identified above through the following contributions:

1. We give the first modular specification of a general class of state-based CRDTs. Our specifications are given in the Aneris distributed separation logic and our proofs are mechanized in Coq.

2. Furthermore, when taken together with Nieto et al. [Nie+22], our work provides a *unified* framework for the specification and verification of *both* kinds of CRDTs. This is because our specifications of state-based CRDTs are compatible with Nieto et al.'s specifications of op-based CRDTs. We emphasize this point by re-verifying the example client program in Nieto et al. that uses a positive-negative counter CRDT, except we swap their op-based counter by a state-based equivalent. Crucially, the client's safety proof remains

virtually unchanged,[3] showing that it is possible to specify CRDTs while hiding their implementation strategy.

3. We give the first formal proof that state-based CRDTs are causally-consistent.

4. We evaluate our approach by verifying a set of example CRDTs from the literature. The evaluation shows that our techniques can handle a variety of CRDTs, including counters, sets, and higher-order combinators.

4.1 Aneris Primer

Aneris [Kro+20] is a distributed separation logic built on top of the Iris program logic framework [Jun+18]. Aneris is designed to reason about safety properties of distributed systems written in AnerisLang, which can be thought of as a subset of OCaml deeply embedded in Coq. This subset includes support for higher-order functions, mutable state, node-local concurrency (including the ability to fork new threads dynamically), as well as expressions for sending and receiving messages over UDP-style sockets. The operational semantics models an unreliable network: once sent, messages can be dropped, re-ordered, arbitrarily-delayed, and duplicated.

Figure 4.2 shows the fragment of Aneris most relevant to this paper. First, notice that the logic includes the usual connectives of a higher-order logic. The separation logic connectives include the separating conjunction $P * Q$, indicating ownership of a resource that can be split into two parts, one satisfying P and the other satisfying Q. The magic wand $P \mathrel{-\!\!*} Q$ denotes resources that, when combined with a resource satisfying P then together satisfy Q. The points-to proposition $\ell \mapsto_{ip} v$ grants exclusive ownership of memory location ℓ on the node with IP address ip alongside the knowledge that value v is stored in ℓ. In other words, only the owner of this resource is allowed to read or modify ℓ's contents. As usual, the Hoare triple $\{P\}\ \langle ip; e\rangle\ \{x.\ Q\}$ is a partial correctness assertion for expression e running on the node with IP address ip. Notice that in the postcondition we bind the return value of e to x, whose scope extends over Q.

Aneris inherits from Iris a notion of *invariant.* An invariant $\boxed{P}^{\mathcal{N}}$ ($\mathcal{N}$ being the name — see below), once established at a point in a proof, asserts that the proposition P hold throughout the execution of the program from that point on and is respected by all threads and nodes. This is enforced by the program logic and is reflected in the invariant *opening* rule. The invariant opening rule allows invariants to be accessed around atomic expressions. That is, it allows us to assume that the invariant holds before the atomic step and enforces that after the atomic step executes, the invariant needs to be *closed* again, meaning that we have to show that it holds again after the execution of the atomic step. The notation $\boxed{P}^{\mathcal{N}}$ says that P is an invariant with *namespace* $\mathcal{N}$. One can think of $\mathcal{N}$ as an identifier for the invariant that helps the logic keep track of which invariants are open at any given point in the proof: this is important because an invariant that is currently open cannot be re-opened.

The points-to proposition $\ell \mapsto_{ip} v$ is but one example of the kind of *resources* that one can assert ownership over. In fact, the user of the logic can define new kinds of resources by creating *partial-commutative monoids* (PCMs): monoids where the product is commutative

[3]Modulo some manual editing of Nieto et al.'s proof development, which could be further eliminated with additional Coq engineering work that refactors some typeclasses

$$\begin{array}{lll}
P, Q \in \mathit{iProp} ::= & \mathsf{True} \mid \mathsf{False} \mid P \wedge Q \mid P \Rightarrow Q \mid P \vee Q \mid \forall x.\ P \mid \exists x.\ P \mid \cdots & \text{higher-order logic} \\
& \mid P * Q \mid P \mathrel{-\!\!*} Q \mid \ell \mapsto_{ip} v \mid \{P\}\ \langle ip; e\rangle\ \{x.\ Q\} & \text{separation logic} \\
& \mid \boxed{P}^{\mathcal{N}} \mid \boxed{a}^{\gamma} & \text{invariants and resources} \\
& \mid \square\, P \mid {}_{\mathcal{E}_1}|\!\!\!\Rrightarrow_{\mathcal{E}_2} & \text{modalities}
\end{array}$$

Figure 4.2: Aneris fragment adapted from Nieto et al. [Nie+22].

and partial. Given a PCM $\mathbb{M}$ and $a \in \mathbb{M}$ the proposition $\boxed{a}^{\gamma}$ asserts ownership of *ghost state* a. Here, γ is the *ghost name* under which a is *stored.* Crucially, $\boxed{a}^{\gamma} * \boxed{b}^{\gamma}$ is equivalent to the monoid product $\boxed{a \cdot b}^{\gamma}$. The logic guarantees that the product of all resources stored under the same ghost name is well-defined: hence by choosing appropriate monoids we can tweak the properties of ghost state.

The proposition $\square\, P$, read *persistently* P, tells us that P holds and it does not assert ownership of any exclusive resources. We say that P is a persistent proposition if $P \vdash \square\, P$. Persistent propositions are duplicable in the sense that $\square\, P \vdash \square\, P * \square\, P$. Finally, the *update* modality ${}_{\mathcal{E}_1}|\!\!\!\Rrightarrow_{\mathcal{E}_2} P$ says that P holds after updating (allocating or modifying) resources; furthermore, we can assume that all invariants in the set $\mathcal{E}_1$ hold when establishing P but must also (re)establish all invariants in the set $\mathcal{E}_2$. The notation $|\!\!\!\Rrightarrow_{\mathcal{E}}$ is shorthand for ${}_{\mathcal{E}}|\!\!\!\Rrightarrow_{\mathcal{E}}$.

4.2 *Main Ideas*

The main idea of this paper is that, from a user's perspective, whether a CRDT is op-based or state-based is an implementation detail, and one that ought not affect the data structure's specification. To capture this idea formally we reach for two tools: separation logic propositions for tracking the global and local states of the CRDT as a function of sets of events (user operations), and high-level specifications for the CRDT based on denotations.

To track a CRDT's state we construct propositions $\mathsf{GlobSt}(s_{\mathsf{g}})$ and $\mathsf{LocSt}\,(i,\ s_{\mathsf{own}},\ s_{\mathsf{for}})$ for global and local states, respectively, as well as a global invariant $\mathsf{GlobInv}$. Ownership of the global state tells us that s_{g} is *exactly* the set of all events issued by any replica. An event is a triple $(\mathsf{op},\ \mathsf{source},\ \mathsf{time})$ where op is the user-provided operation (e.g. `inc(10)` for a g-counter), source is the id of the replica that issued the operation (ids are just natural numbers), and time is a logical timestamp that allows us to order events according to the usual *happens-before* relation. Ownership of the local state $\mathsf{LocSt}\,(i,\ s_{\mathsf{own}},\ s_{\mathsf{for}})$ tells us that replica i has issued *exactly* the events in s_{own}, and that it has received *at least* the events in s_{for}, which all originate outside of i (we only get an approximation of the events from outside of i because in between two user interactions with the CRDT at replica i, new merge operations might have taken place).

Once the above propositions have been defined, we prove a comprehensive suite of "resource lemmas" that allow a client to reason about the state of the CRDT. For example, we prove that if a user knows both $\mathsf{GlobSt}(s_{\mathsf{g}})$ and $\mathsf{LocSt}\,(i,\ s_{\mathsf{own}},\ s_{\mathsf{for}})$, and they can find events e and e' where $e \in s_{\mathsf{own}} \cup s_{\mathsf{for}}$ (e is in the local state), $e' \in s_{\mathsf{g}}$ (e' is in the global state) and e' happens before e, then e' must have been received at replica i as well: $e' \in s_{\mathsf{own}} \cup s_{\mathsf{for}}$. This is an encoding of causality in separation logic [Gon+21; Nie+22]. We do not claim this formulation as a contribution; instead, our contribution is defining the above predicates in such a way that they can track state-based CRDTs, and then proving existing lemma statements for our definitions. In effect, we have re-implemented an existing interface. This is crucial because it means that from a client's perspective it does not matter what kind of CRDT (state-based or op-based) they are working with: they all satisfy the same laws.

Technical Challenges Resource lemmas like causality hold only in the presence of the global invariant $\mathsf{GlobInv}$. This invariant guarantees that at all times the state of the system is *valid*. The system state is a tuple $\vec{s_l}$ of the local state at every replica. Defining a notion of validity suitable for state-based CRDTs is one of the tricky technical parts of our proof. Two complications arise: how to represent local states, and how to represent time.

Defining local state as an element of the lattice over which the CRDT is implemented does not work, because we lose track of the individual events. For example, we might remember that the state of the counter is `[4, 5]`, but not that it resulted from three events: two increments of 2 each at replica 0, and one increment of 5 at replica 1. Remembering events is needed to show the resource lemmas. We therefore represent local states as sets of events, but now we need a way to link these sets of events to the lattice element that is computed at runtime. We do this with denotations, which are functions from sets of events to lattice elements. In our proof, a local invariant ensures that, for every replica i, if s is the set of tracked events for i and the runtime CRDT state at i is st, we must have $[\![s]\!] = st$. The definition of denotation for a state-based CRDT must then satisfy a number of *coherence* properties with respect to the underlying lattice: for example, we require that, under certain conditions, if $[\![s]\!] = st$ and $[\![s']\!] = st'$, then $[\![s \cup s']\!] = st \sqcup st'$. In other words, our proof tracks the logical state with a free lattice of events, while the implementation computes using the CRDT-specific lattice. The denotation is the homomorphism between the two.

For lemmas like causality we also need to be able to compare events according to time. Prior work on op-based CRDTs implements a causal broadcast library that tags each event with a (physical) *vector clock* [4] that serves as a timestamp [Nie+22]. In our setting, since state-based CRDTs do not assume causal broadcast, we use a purely logical notion of time. Given an event e, its timestamp is taken to be the set of events that e causally depends on. This set is computed at the point e is created: if e was issued at replica i where the local state is an event set s, then e causally depends on every element of s. This works because we can show that local states are always *dependency-closed*: if $e' \in s$ and e_d is a dependency of e', then $e_d \in s$ as well. Our definition of logical time allows us to give the first formal proof that state-based CRDTs are causally-consistent.

Verified Examples With the above in place we turn to the verification of different state-based CRDTs. We implemented and verified five CRDTs from the literature [Sha+11a]: a grow-only counter, a positive-negative counter, an add-only set, a combinator for products of CRDTs, and a combinator for maps from strings to an underlying CRDT type. We implemented these examples in two steps: first, we implemented a StateLib library that factors out all the common elements in different examples: network calls, merging of remote states, and concurrency control. The library takes as input a purely-functional implementation of a state-based CRDT,

[4] A vector of integers, with one entry per node in the system. The ith entry tells us how many updates have been done by replica i.

in the form of a triple `(init_st, mutator, merge)`, where `init_st` is the CRDT's initial (lattice based) state, `mutator` is a function that takes a state and an operation and produces the next state, and merge implements the lattice's least upper bound operation. StateLib requires that the CRDT implementer proves the aforementioned coherence properties (e.g., that merging two states is the same as taking the denotation of the union of their corresponding event sets) about their purely-functional implementation. Given this core logic, the library returns a fully-fledged replicated data type and two functions, `get_state` and `update`, to query and update the state of the data-structure. In the second step, we wrote purely-functional implementations for each of the previously-mentioned examples and proved the relevant coherence properties so the library can be instantiated with them. The modular design of the library allows us to prove the library safe just once, and then re-use that proof to obtain safety proofs for each of the CRDTs.

Finally, we wanted to validate our claim that a client need not know whether the CRDT they are interacting with is state-based or not. We did this by using the example in Nieto et al. where they verify a client program together with an op-based positive-negative counter [Nie+22]. We were able to swap their op-based counter with our state-based positive-negative counter while leaving the proof virtually unmodified (small technical changes are required, but these could be eliminated with further Coq engineering). This shows that CRDTs can be true abstract data types, and can be specified while abstracting away implementation details.

The rest of the paper is structured as follows: Section 4.3 gives an overview of the CRDT resource lemmas in Nieto et al. [Nie+22], which we re-prove in the state-based context. Sections 4.4, 4.5, and 4.6 present StateLib's design, specification, and safety proof, respectively. Section 4.7 describes the verified example CRDTs, as well as the proof of the client program that is agnostic to the CRDT class. Section 5.7 surveys related work and Section 4.9 concludes.

4.3 Background: CRDTs in Separation Logic

We give an overview of the separation logic approach to verification of op-based CRDTs in Nieto et al. [Nie+22]. Specifically, they introduce abstract separation logic resources (abstract predicates) for tracking the local and global states of a CRDT. The abstract resources are later used in the specifications of CRDT operations. Nieto et al. reason only about op-based CRDTs, but in this paper we show how to instantiate the abstract resources so that we can verify state-based CRDT implementations as well. As we will later see, this allows clients to reason about a CRDT while remaining agnostic of the CRDT's implementation strategy.

4.3.1 Time, Events, and Denotations

We start by giving a few definitions that we will use throughout the paper.

Logical time allows us to order events in a distributed system using causal order. Time is axiomatized by a triple $(\mathsf{Time}, \leq_t, <_t)$, where Time is a set of *timestamps*, $\leq_t$ is a partial order on timestamps, and $<_t$ is the strict version of $\leq_t$. For example, for working with op-based CRDTs, Nieto et al. instantiate logical time by taking timestamps to be vector clocks and $\leq_t$ to be the associated pointwise ordering.

Logical events represent operations that are executed by the CRDT, together with metadata. A logical event is a triple $(\mathsf{op}, \mathsf{source}, \mathsf{time}) \in \mathsf{Event} \triangleq \mathsf{Op} \times \mathbb{N} \times \mathsf{Time}$. Here Op is the type of *operations* on the CRDT (e.g. $\mathsf{Op} \triangleq \{\mathsf{inc}(n) \mid n \in \mathbb{N}\}$ for a g-counter), source is the id of the replica that generated the event, and time is a timestamp. For an event e we write $e.o$, $e.s$, and $e.t$ for the operation, source, and timestamp of e, respectively.

Given two event sets s and s', we say that s is a *causally-closed subset* of s', written $s \subseteq_{\mathsf{cc}} s'$, if $s \subseteq s'$ and

$$\forall e\ e',\ e \in s' \Rightarrow e' \in s' \Rightarrow e.t \leq_t e'.t \Rightarrow e' \in s \Rightarrow e \in s$$

That is, if we start with two events from s' and the later one e' (according to timestamp ordering) is in s, then e (its causal dependency) must be in s as well.

Finally, we use *denotations* to specify CRDTs. A denotation is a tuple $(\mathsf{Op}, \mathsf{St}, s_{\mathsf{init}} : \mathsf{St}, [\![\cdot]\!] : \mathcal{P}(\mathsf{Event}) \rightharpoonup \mathsf{St})$. For example, as in Figure 4.1, for a g-counter we could have Op as previously defined, $\mathsf{St} \triangleq \mathbb{N}$, $s_{\mathsf{init}} \triangleq 0$, and $[\![s]\!] = \sum_{e \in s} \mathsf{unwrap}(e.o)$ with $\mathsf{unwrap}(\mathsf{inc}(n)) = n$. We could have also chosen St to be the set of lists of naturals of length N, where N is the number of replicas. This latter denotation would be closer to the state-based implementation.

It is useful for denotations to be partial because we can avoid giving a meaning to ill-formed sets of events. For example, suppose we have $s = \{a, b\}$ with $a.t = b.t$ but $a \neq b$. We might know that in practice events with equal timestamps must be equal, so s can never arise during an execution. We might then choose $[\![s]\!]$ to be undefined.

4.3.2 Separation Logic Resources

So far we have not shown any Aneris definitions. We do so in Figure 4.3, which shows the types of different abstract separation logic resources (predicates) that, together with associated lemmas, can be used to reason about the state of CRDTs. Specifically, these resources appear in the pre and post-conditions of functions that operate on a CRDT. The abstract resources are designed to be generic so they can be used with multiple CRDTs. Indeed, Nieto et al. [Nie+22] verified multiple op-based example CRDTs using these resources, and we have also verified multiple state-based examples. Notice that Figure 4.3 does not show the definition of the resources. The reader can think of Figure 4.3 as defining an *interface* in the software engineering sense. Nieto et al. implement this interface for op-based CRDTs, while we re-implement it for state-based CRDTs. For space reasons, we do not show the entire interface; the full interface can be found in the accompanying Coq code.

The defined resources are as follows: there is a *global invariant* $\mathsf{GlobInv}$ that holds throughout the CRDT's existence. Then we define resources for tracking *global* (GlobSt) and *local* (LocSt) states. The intuition is that if $\mathsf{GlobSt}(s_{\mathrm{g}})$ holds, then we know that s_{g} is *exactly* the set of events issued by any CRDT replica in the system. Similarly, ownership of $\mathsf{LocSt}\ (i,\ s_{\mathsf{own}},\ s_{\mathsf{for}})$ tells us that replica i has processed *exactly* the events in s_{own} and *at least* the events in s_{for}. The events in s_{own} (the "own" events) must have all originated at i, while the ones in s_{for} (the "foreign" events) must all originate outside of i. This is because a client of the CRDT knows exactly which events it has issued, but is potentially not aware of all events that have been received "in the background" since the last interaction with the CRDT.

Global and local states are *exclusive* (not shown in Figure 4.3), meaning that only one copy of the resource (or one per replica in the case of local state) can exist: e.g., $\mathsf{GlobSt}(s_{\mathrm{g}}) \twoheadrightarrow \mathsf{GlobSt}(s'_{\mathrm{g}}) \twoheadrightarrow \bot$.

By contrast, the interface also declares *global and local snapshots* ($\mathsf{GlobSnap}$ and $\mathsf{LocSnap}$), which are persistent (duplicable) and give us a *lower bound* on global and local states, respectively. Snapshots are useful as a "certificate" that an event was generated by a given replica: this is the case if one can prove, e.g., that $\mathsf{GlobSnap}(s_{\mathrm{g}})$ and $e \in s_{\mathrm{g}}$ for an event e. The use of

Resources (abstract predicates)

$$
\begin{array}{rr}
\text{(Global invariant)} & \mathsf{GlobInv} : \mathit{iProp} \\
\text{(Global state)} & \mathsf{GlobSt} : \mathcal{P}(\mathsf{Event}) \to \mathit{iProp} \\
\text{(Global snapshot)} & \mathsf{GlobSnap} : \mathcal{P}(\mathsf{Event}) \to \mathit{iProp} \\
\text{(Local state)} & \mathsf{LocSt} : \mathbb{N} \to \mathcal{P}(\mathsf{Event}) \to \mathcal{P}(\mathsf{Event}) \to \mathit{iProp} \\
\text{(Local snapshot)} & \mathsf{LocSnap} : \to\Rightarrow \mathcal{P}(\mathsf{Event}) \to \mathcal{P}(\mathsf{Event}) \to \mathit{iProp}
\end{array}
$$

Global state laws

$$
\begin{array}{rl}
\text{(GlobStTakeSnap)} & \forall E\ s, \mathcal{N}^{\uparrow} \subseteq E \Rightarrow \mathsf{GlobInv} \mathrel{-\!\!*} \mathsf{GlobSt}(s) \mathrel{-\!\!*} \\
& \quad \Rrightarrow_E \mathsf{GlobSt}(s) * \mathsf{GlobSnap}(s)
\end{array}
$$

$$
\begin{array}{rl}
\text{(GlobSnapIncl)} & \forall E\ s\ s', \mathcal{N}^{\uparrow} \subseteq E \Rightarrow \mathsf{GlobInv} \mathrel{-\!\!*} \mathsf{GlobSnap}(s) \mathrel{-\!\!*} \mathsf{GlobSt}(s') \mathrel{-\!\!*} \\
& \quad \Rrightarrow_E s \subseteq s' * \mathsf{GlobSt}(s')
\end{array}
$$

Local state laws

$$
\begin{array}{rl}
\text{(LocSnapIncl)} & \forall E\ i\ s_{\mathsf{own}}\ s_{\mathsf{for}}\ s'_{\mathsf{own}}\ s'_{\mathsf{for}}, \mathcal{N}^{\uparrow} \subseteq E \Rightarrow \mathsf{GlobInv} \mathrel{-\!\!*} \mathsf{LocSnap}(i, s_{\mathsf{own}}, s_{\mathsf{for}}) \mathrel{-\!\!*} \\
& \mathsf{LocSt}\left(i,\ s'_{\mathsf{own}},\ s'_{\mathsf{for}}\right) \mathrel{-\!\!*} \Rrightarrow_E s_{\mathsf{own}} \cup s_{\mathsf{for}} \subseteq_{\mathsf{cc}} s'_{\mathsf{own}} \cup s'_{\mathsf{for}} * \mathsf{LocSt}\left(i,\ s'_{\mathsf{own}},\ s'_{\mathsf{for}}\right)
\end{array}
$$

$$
\begin{array}{rl}
\text{(LocSnapExt)} & \forall E\ i\ i'\ s_{\mathsf{own}}\ s_{\mathsf{for}}\ s'_{\mathsf{own}}\ s'_{\mathsf{for}}, \mathsf{LocSnap}(i, s_{\mathsf{own}}, s_{\mathsf{for}}) \mathrel{-\!\!*} \mathsf{LocSnap}(i', s'_{\mathsf{own}}, s'_{\mathsf{for}}) \mathrel{-\!\!*} \\
& \Rrightarrow_E \forall e\ e', e \in s_{\mathsf{own}} \cup s_{\mathsf{for}} \Rightarrow e' \in s'_{\mathsf{own}} \cup s'_{\mathsf{for}} \Rightarrow e.t = e'.t \Rightarrow e = e'
\end{array}
$$

$$
\begin{array}{rl}
\text{(LocSnapProv)} & \forall E\ i\ s_{\mathsf{own}}\ s_{\mathsf{for}}\ e, \mathcal{N}^{\uparrow} \subseteq E \Rightarrow e \in s_{\mathsf{own}} \cup s_{\mathsf{for}} \mathrel{-\!\!*} \mathsf{GlobInv} \mathrel{-\!\!*} \\
& \mathsf{LocSnap}(i, s_{\mathsf{own}}, s_{\mathsf{for}}) \mathrel{-\!\!*} \Rrightarrow_E \exists s_g, \mathsf{GlobSnap}(s_g) * e \in s_g
\end{array}
$$

$$
\begin{array}{rl}
\text{(GlobSnapProv)} & \forall E\ i\ s_{\mathsf{own}}\ s_{\mathsf{for}}\ s_g, \mathcal{N}^{\uparrow} \subseteq E \Rightarrow \mathsf{GlobInv} \mathrel{-\!\!*} \mathsf{LocSt}\left(i,\ s_{\mathsf{own}},\ s_{\mathsf{for}}\right) \mathrel{-\!\!*} \\
& \mathsf{GlobSnap}(s_g) \mathrel{-\!\!*} \Rrightarrow_E \mathsf{LocSt}\left(i,\ s_{\mathsf{own}},\ s_{\mathsf{for}}\right) * \\
& \forall e, e \in s_g \Rightarrow \mathsf{EV\_Orig}(e) = i \Rightarrow e \in s_{\mathsf{own}}
\end{array}
$$

$$
\begin{array}{rl}
\text{(Causality)} & \forall E\ i\ s_{\mathsf{own}}\ s_{\mathsf{for}}\ s_g, \mathcal{N}^{\uparrow} \subseteq E \Rightarrow \mathsf{GlobInv} \mathrel{-\!\!*} \mathsf{LocSt}\left(i,\ s_{\mathsf{own}},\ s_{\mathsf{for}}\right) \mathrel{-\!\!*} \\
& \mathsf{GlobSnap}(s_g) \mathrel{-\!\!*} \Rrightarrow_E \mathsf{LocSt}\left(i,\ s_{\mathsf{own}},\ s_{\mathsf{for}}\right) * \\
& \forall e\ e', e \in s_g \Rightarrow e' \in s_{\mathsf{own}} \cup s_{\mathsf{for}} \Rightarrow e <_t e' \Rightarrow e \in s_{\mathsf{own}} \cup s_{\mathsf{for}}
\end{array}
$$

Figure 4.3: CRDT resources and selected lemmas, from Nieto et al. [Nie+22].

global snapshots as certificates is validated by lemma GlobSnapIncl (Figure 4.3). The lemma says that under the global invariant, if we own $\mathsf{GlobSnap}(s_g)$ and $\mathsf{GlobSt}(s'_g)$, then we must have $s_g \subseteq s'_g$. This conclusion holds under the update modality $\Rrightarrow_E$, which means that it holds possibly after opening (and closing) all invariants in the mask E. The premise $\mathcal{N}^{\uparrow} \subseteq E$ tells us that the global invariant's namespace $\mathcal{N}^{\uparrow}$ is part of E. This means that GlobInv must not be open when the lemma is called (because the proof of the lemma opens GlobInv). LocSnapIncl provides similar inclusion guarantees for local snapshots, but note that we actually get the stronger causally-closed inclusion $\subseteq_{cc}$, as opposed to just $\subseteq$.

GlobStTakeSnap allows us to take snapshots of global states. LocSnapExt says that if two events in a local snapshot have equal timestamps, then the events must be equal.

Finally, we have three lemmas that tie together local and global states. LocSnapProv says that if e is tracked locally, then there must exist some global snapshot $\mathsf{GlobSnap}(s_g)$ such that $e \in s_g$. That is, all local events are also tracked globally. GlobSnapProv says that if an event $e \in s_g$ has origin i and we know $\mathsf{GlobSnap}(s_g)$ (e is tracked globally), then e must also be in the local state for i. Finally, Causality is our definition of causality in separation logic. This take on causality was originally developed for reasoning about a causally-consistent key value store by Gondelman et al. [Gon+21], and later generalized by Nieto et al. [Nie+22] so it can apply to events in an arbitrary CRDT. The lemma is as follows: suppose we have two events e and e' such that e' was recorded locally at node i (that is, $e' \in s_{\mathsf{own}} \cup s_{\mathsf{for}}$ and we know $\mathsf{LocSt}\,(i,\ s_{\mathsf{own}},\ s_{\mathsf{for}})$). Next suppose that e happened before e', and e is a logically tracked event (which we can show by presenting $\mathsf{GlobSnap}(s_g)$ with $e \in s_g$). Then causal delivery requires that e be observed locally at i as well: i.e., $e \in s_{\mathsf{own}} \cup s_{\mathsf{for}}$. Gondelman et al. [Gon+21] show how this definition of causality is strong enough to prove four *session guarantees* (monotonic reads, monotonic writes, read your writes, and writes follow reads) that programmers intuitively expect when interacting with a causally-consistent datatype.

Reasoning With Resources To tie all the above together, Figure 4.4 shows how the previously-discussed resources can be used to specify the `inc` operation on a g-counter CRDT. We assume that `inc` both increments the counter and returns its current local value. The precondition for calling `inc` at replica i requires knowledge of both $\mathsf{LocSt}\,(i,\ s_{\mathsf{own}},\ s_{\mathsf{for}})$ and $\mathsf{GlobSt}(s_g)$. This is because every single increment must be tracked both locally at the replica where it is performed and globally. In the postcondition we get back $\mathsf{LocSt}\,(i,\ s_{\mathsf{own}} \cup \{e\},\ s'_{\mathsf{for}})$, where e is the event generated by the increment and $s_{\mathsf{for}} \subseteq s'_{\mathsf{for}}$. Notice that the "own events" grow by exactly one event, e, but the "foreign events" grow to some superset s'_{for} of s_{for}. This is because since the last time `inc` was called any number of new events could have been propagated from other replicas to replica i. Notice how we connect the implementation to its functional correctness specification by saying that the return value v is the denotation of the locally-observed events $s_{\mathsf{own}} \cup \{e\} \cup s'_{\mathsf{for}}$. Finally, observe that by using denotations we automatically obtain convergence (the safety part of SEC), because the return value is a function of a *set* of events, so two replicas that have seen the same set of events must return the same result.

Resources for State-Based CRDTs Two difficulties arise when instantiating the resource interface for state-based CRDTs.

1. How should we track local state? The replica state in an implementation of a state-based CRDT is a lattice element. By contrast, the resource interface logically tracks replica states as sets of events (operations). The solution is to link the two representations via a denotation: if

IncSpec

$$\{\mathsf{LocSt}\,(i,\,s_{\mathsf{own}},\,s_{\mathsf{for}}) * \mathsf{GlobSt}(s_{\mathsf{g}})\}$$
$$\langle ip_i;\ \mathsf{inc}(n)\rangle$$
$$\left\{\begin{array}{l} v.\ \exists e\ s'_{\mathsf{for}}.\ s'_{\mathsf{for}} \supseteq s_{\mathsf{for}} * e \notin s_{\mathsf{g}} * e.o = n * e.s = i \\ \quad [\![s_{\mathsf{own}} \cup \{e\} \cup s'_{\mathsf{for}}]\!] = v * \mathsf{LocSt}\,\left(i,\,s_{\mathsf{own}} \cup \{e\},\,s'_{\mathsf{for}}\right) * \mathsf{GlobSt}(s_{\mathsf{g}} \cup \{e\}) \end{array}\right\}$$

Figure 4.4: Simplified spec of an increment operation, which returns the counter's current value.

the logical state is $\mathsf{LocSt}\,(i,\,s_{\mathsf{own}},\,s_{\mathsf{for}})$, then the physical state must be $[\![s_{\mathsf{own}} \cup s_{\mathsf{for}}]\!]$, which in turn must be drawn from the appropriate lattice. The link is also needed when propagating a replica's state to other replicas. In the implementation, a replica sends a message containing its entire state e, so others can merge it. Logically, we require that the sent state be paired with a local snapshot whose denotation is e.

2. How do we track time to prove causal consistency? For their treatment of op-based CRDTs, Nieto et al. [Nie+22] implemented a causal broadcast algorithm that ensures that every message sent by a CRDT replica is delivered in causal order. This is achieved via vector clocks in the standard way. But state-based CRDTs should not rely on causal broadcast; in fact, one of the main advantages of the state-based approach is that messages can be delivered out-of-order and re-delivered without causing issues (because of the properties of least upper bound). It is not immediately clear why the state-based design satisfies causal delivery. The first key observation is that if we start with a replica state st and look at the event set s that produced it, i.e., $[\![s]\!] = st$, then s "has no holes" with respect to causality. That is, if we take an event $e \in s$ and e' is another event that happened before e, then it must be the case that $e' \in s$ as well. This is formalized via the notion of *dependency-closure* (Section 4.6.1). The second observation is that when a new operation o is applied to a local state st with $[\![s]\!] = st$, it (logically) generates a new event e' with $e'.o = o$. e''s causal dependencies are *exactly* the events in s. This is important because it means that we can track an event's dependencies *purely logically*, without the need for vector clocks. Using these ideas we are able to prove that the (Causality) lemma holds for state-based CRDTs (Lemma 4.6.6). To our knowledge, this is the first formal proof that state-based CRDTs are causally-consistent.

4.4 StateLib : a Library for Implementing State-Based CRDTs

We have structured our CRDT implementations so that common functionality is factored out into a separate library, which can then be instantiated as needed by different CRDT examples. The library, called StateLib, is responsible for maintaining the CRDT's internal state and inter-replica propagation. The library's code is shown in Figure 4.5, together with an abridged example instantiating a *grow-only set* (g-set) CRDT, a set to which we can add elements, but from which we cannot remove them [Sha+11a].

We start by describing the `init` function, which is the library's entry point. A CRDT implementer calls `init` with the following arguments: serialization and de-serialization functions, a list of all replica addresses, the current replica id, and a `crdt` parameter that describes how the specific CRDT being instantiated should behave. The serialization functions have the expected types: `type 's serT = 's -> string` and `type 's deserT = string -> 's`. The

```
let get_state lock st () =
  acquire lock;
  let res = !st in (* LP *)
  release lock;
  res

let rec loop_forever thunk =
  thunk ();
  loop_forever thunk

let apply deser lk sh st merge : =
  loop_forever (fun () ->
    let msg =
      unSOME (receiveFrom sh) in
    let st’ = deser (fst msg) in
    acquire lk;
    st := merge !st st’;
    release lk)

let update lk mut i st op =
  acquire lk;
  st := mut i !st op; (* LP *)
  release lk

let sendToAll sh dstl i msg =
  let j = ref 0 in
  let rec aux () =
    if !j < list_length dstl then
    if i = j then (j := !j +1; aux ())
    else begin
      let dst =
        unSOME (list_nth dstl !j) in
      sendTo sh msg dst;
      j := !j + 1
      aux ()
    end
    else ()
  in aux ()
```

```
let broadcast ser lk sh st dstl i =
  loop_forever(fun () ->
    Unix.sleepf 2.0;
    acquire lk;
    let s = !st in
    release lk;
    let msg = ser s in
    sendToAll sh dstl i msg)

let init ser deser addrs rid crdt =
  let ((init_st, mut), merge) = crdt in
  let st = ref (init_st ()) in
  let lk = newlock () in
  let sh = socket () in
  let addr = unSOME (list_nth addrs rid) in
  socketBind sh addr;
  fork (apply deser lk sh st merge);
  fork (broadcast ser lk sh st addrs rid);
  let get = get_state lk st in
  let upd = update lk mut rid st in
  (get, upd)

(* G-Set instantiation *)

let mutator i st op = set_add op st
let merge st1 st2 = set_union st1 st2
let init_st = set_empty
let gset_crdt = ((init_st, mutator), merge)

(* Instantiate via *)
let (get, upd) = init ... gset_crdt
```

Figure 4.5: StateLib implementation and a G-Set example. Linearization points are marked with an LP comment.

`crdt` argument has type `('o,'s) crdT`, parameterized on operations and states:

```
type 's mergeT = 's -> 's -> 's
type ('o,'s) mutT = int -> 's -> 'o -> 's
type ('o,'s) crdtT = (('s * ('o,'s) mutT) * 's mergeT)
```

That is, a value of type `('o, 's) crdtT` is a triple `(init_st, mutator, merge)` containing an initial state, a mutator function, and a merge function. The mutator takes a replica id, the current state, and a new operation, and returns the state that results after applying the operation locally. The merge function takes two states and returns their least upper bound.

Back to the body of `init`, we see that it unpacks the `crdt` argument. It then allocates a local reference to hold the current state of the CRDT, a lock to control updates to the state, and a socket over which it can communicate with other replicas. The function then spawns two concurrent threads, `apply` and `broadcast`, in charge of receiving updates from other replicas and propagating local updates, respectively. Finally, `init` returns a pair of functions `get_state` and `update` allowing the library user to query and update the CRDT state.

Both `get_state` and `update` have simple implementations. The former just dereferences the local state, while the latter uses the user-provided mutator to compute the CRDT's next state. Both operations are guarded by a lock.[5]

The `apply` function, which is spawned off as a separate thread from `init`, is responsible for receiving updates from other replicas and then merging them with the local state, using the user-provided `merge` function. The call to `receiveFrom` blocks until a message is available at the given socket handle. The function `unSOME : 'a option -> 'a` unwraps a value of an option type, crashing if the argument is `None`. Notice that the received message needs to be deserialized via the user-provided `deser` function. Also notice that `apply` does not terminate, but mutates the CRDT's state.

The dual of `apply` is `broadcast`, which is tasked with propagating the local state to other replicas. This function also runs on a separate thread, and loops forever, working solely via side effects. The `broadcast` function retrieves a copy of the local state, serializes it, and then calls a helper function `sendToAll`. This helper takes a list of IP addresses `dstl`, the current replica id `i` and the message (state) to be sent. It then loops over the elements of `dstl` and sends the message to each of them (taking care to not send a message to itself).

Finally, Figure 4.5 sketches how one might implement a g-set via the library. We represent the state with a sequential set. The mutator is just set insertion, `merge` is implemented via set union, and the initial state is the empty set. We can then package these three components into a tuple `gset_crdt`, and provide the latter as an argument to `init` (together with the serialization functions and replica IP addresses). From `init` we get back a pair functions for querying the current value of the set and updating it.

4.5 *Specifying* StateLib

The StateLib library has two interfaces: an internal interface used by the CRDT *implementer*, consisting of the `init` function, and an external interface used by the CRDT *client*, consisting of get_state and update.

[5]In the case of `get_state`, loads in AnerisLang are atomic, so the lock is not strictly needed; however, dereferences are not atomic in OCaml [DSM18], which we use to run our code, so we use a lock.

InitStSpec: $[\![\emptyset]\!] = \mathsf{initSt}$

MutatorSpec

$$\left\{\begin{array}{l}[\![s]\!] = st \land e.o = op \land e.s = i \land e \notin s \\ \land\ \mathsf{CohVal}(s \cup \{e\}) \land \mathrm{Maximum}(e) s \cup \{e\}\end{array}\right\}$$
$$\langle ip_i;\ \mathsf{mut}(st, op)\rangle$$
$$\{st'.\ [\![s \cup \{e\}]\!] = st' \land st \leq_L st'\}$$

MergeSpec

$$\left\{\begin{array}{l}[\![s_1]\!] = st_1 \land [\![s_2]\!] = st_2 \land \mathsf{SectIncl}(s_1, s_2) \\ \land\ \mathsf{CohVal}(s_1) \land \mathsf{CohVal}(s_2) \land \mathsf{CohVal}(s_1 \cup s_2)\end{array}\right\}$$
$$\langle ip_i;\ \mathsf{merge}(st_1, st_2)\rangle$$
$$\{st'.\ st_1 \sqcup st_2 = st' \land [\![s_1 \cup s_2]\!] = st'\}$$

$$\mathsf{sect}(s, i) \triangleq \{e \in s \mid e.s = i\}$$
$$\mathsf{SectIncl}(s, s') \triangleq \forall i.\ \mathsf{sect}(s, i) \subseteq \mathsf{sect}(s', i) \lor \mathsf{sect}(s', i) \subseteq \mathsf{sect}(s, i)$$

$\mathsf{CohVal}(s) \triangleq$ a version of "local state validity" (Section 4.6.1) that does not imply $\mathsf{depclosed}(s)$.

$$\textsc{CrdtSpec}((\mathsf{initSt}, \mathsf{mut}, \mathsf{merge})) \triangleq \textsc{InitStSpec}(\mathsf{initSt}) * \textsc{MutatorSpec}(\mathsf{mut}) * \textsc{MergeSpec}(\mathsf{merge})$$

InitSpec

$$\{\ldots * \textsc{CrdtSpec}(\mathsf{crdt})\}$$
$$\langle ip_i;\ \mathsf{init}(\mathsf{addrs}, \mathsf{repId}, \mathsf{crdt})\rangle$$
$$\{(\mathsf{get\_state}, \mathsf{update}).\ \mathsf{LocSt}\,(i,\ \emptyset,\ \emptyset) * \textsc{GetStateSpec}(\mathsf{get\_state}) * \textsc{UpdateSpec}(\mathsf{update})\}$$

Figure 4.6: Internal specifications. GetStateSpec and UpdateSpec are defined in Figure 4.7.

4.5.1 *Internal Interface*

Recall that StateLib is initialized by the CRDT implementer through an `init` function, taking in (de-) serialization functions for the CRDT state, a list of replica IP addresses, and finally a `crdt` argument describing the lattice being implemented (Figure 4.5). This last parameter is a triple $\mathsf{crdt} = (\mathsf{initSt}, \mathsf{mut}, \mathsf{merge})$ consisting of the CRDT's initial state $\mathsf{initSt} \in \mathsf{LatSt}$, a *mutator* function $\mathsf{mut} : \mathsf{LatSt} \to \mathsf{Event} \to \mathsf{LatSt}$, and a *merge* function $\mathsf{merge} : \mathsf{LatSt} \to \mathsf{LatSt} \to \mathsf{LatSt}$.

The CRDT implementer first defines a poset $(\mathsf{LatSt}, \leq_L)$ and then proves that it is a lattice. Because our tracking of replica states is defined in terms of event sets (Figure 4.3) we need a way to connect said events to the physical CRDT state, which is a lattice element. Intuitively, we would like to guarantee that the CRDT's physical state is the denotation of the set of events received so far. For this to be true, we need certain coherence properties between event sets, their denotations, and lattice elements. These are shown in Figure 4.6 and consist of specifications for initSt, mut and merge.

InitStSpec says that the denotation of the empty set of events must be the initial state.

MutatorSpec says that if we start in a state $st = [\![s]\!]$ and through a mutation get to a state $st' = \mathsf{mut}(st, op)$, then we can also arrive at st' by taking the denotation $[\![s \cup \{e\}]\!]$, where e is the event containing op. We also need to show that the mutator is monotonic: we must have $st \leq_L st'$ in the lattice order. In proving these goals we get to make additional assumptions about s and e. Specifically, we can assume that s is a set of events that is *valid with respect to coherence* (we explain validity in Section 4.6.1); additionally, we know that e is the maximum element with respect to timestamp ordering in the set $s \cup \{e\}$. Intuitively, this is because e is a new event being added, so it has every event in s as a causal dependency.

MergeSpec shows coherence of merges. Here we start with two states st and st' that we

GetStateSpec

$$\left\langle \mathsf{LocSt}\,(i,\ s_{\mathsf{own}},\ s_{\mathsf{for}})\right\rangle$$
$$\langle ip_i;\ \mathsf{get\_state}()\rangle$$
$$\left\langle \begin{array}{l} v.\ \exists s'_{\mathsf{for}}\ w.\ s'_{\mathsf{for}} \supseteq s_{\mathsf{for}} * \mathsf{StCoh}(w, v) * \\ \quad \mathsf{LocSt}\left(i,\ s_{\mathsf{own}},\ s'_{\mathsf{for}}\right) * [\![s_{\mathsf{own}} \cup s'_{\mathsf{for}}]\!] = w \end{array}\right\rangle^{\mathcal{N}}$$

UpdateSpec

$$\langle \mathsf{LocSt}\,(i,\ s_{\mathsf{own}},\ s_{\mathsf{for}}) * \mathsf{GlobSt}(s_{\mathsf{g}})\rangle$$
$$\langle ip_i;\ \mathsf{update}(v)\rangle$$
$$\left\langle \begin{array}{l} ().\ \exists e\ s'_{\mathsf{for}}.\ s'_{\mathsf{for}} \supseteq s_{\mathsf{for}} * e \notin s_{\mathsf{own}} * e \notin s_{\mathsf{g}} * e.o = v * \\ \quad e.s = i * e \in \mathrm{Maximals}(s_{\mathsf{g}} \cup \{e\}) * \\ \quad \mathrm{Maximum}(e) s_{\mathsf{own}} \cup s'_{\mathsf{for}} \cup \{e\} * \\ \quad \mathsf{LocSt}\left(i,\ s_{\mathsf{own}} \cup \{e\},\ s'_{\mathsf{for}}\right) * \mathsf{GlobSt}(s_{\mathsf{g}} \cup \{e\}) \end{array}\right\rangle^{\mathcal{N}}$$

Figure 4.7: External specifications. $\mathcal{N}$ is any invariant namespace that includes GlobInv's name. Adapted from Nieto et al. [Nie+22].

want to merge to get a third state st''. We know that $[\![s]\!] = st$ and $[\![s']\!] = st'$, and we would like to conclude that $[\![s \cup s']\!] = st''$ and also that merge is in fact computing the least upper bound, so $st \sqcup st' = st''$. Once again we get to assume validity of the relevant event sets, and now additionally we know an inclusion property of sections (a section is a subset of events that originates at a specific replica). The proposition SectIncl(s, s') tells us that if we look at any particular section, say section i, then either sect(s, i) is a subset of sect(s', i), or the other way around. Intuitively, this is because sections are always "complete": if a replica has received event $(6, 5)$, it must have also received all events in the range $(6, 1), \ldots, (6, 4)$.

We package the three specifications in the assertion CrdtSpec(crdt), which asserts that each of the components of the crdt tuple satisfies the corresponding spec above. Finally we have specification for the init function. In the precondition of InitSpec, we assert that the crdt argument satisfies CrdtSpec. In the postcondition, we learn that init returns a pair of functions get_state and update that satisfy the same-named specifications (described in the next section). We also gain ownership of the resource LocSt $(i,\ \emptyset,\ \emptyset)$, indicating that the local replica has yet to receive any events (because it has just been initialized).

4.5.2 External Interface

StateLib's external interface consists of two functions: get_state, which takes no arguments and returns the local state of the CRDT, and update, which takes an operation, updates the local state, and returns a Unit. Figure 4.7 shows specifications for both functions; these specifications are identical to the ones for the same-named functions in Nieto et al.'s library for op-based CRDTs [Nie+22]. This is by design: by proving that our library meets the same specification as the equivalent library for op-based CRDTs we then make it possible for client programs to use (and reason about) a replicated data type without knowledge of whether the data type is op-based or state-based. That is, we hide implementation details to turn CRDTs into true abstract data types.

Looking at Figure 4.7, the reader will notice that the specifications use angle brackets instead of braces: i.e. we write $\langle P\rangle e\langle Q\rangle^{\mathcal{N}}$, instead of the usual Hoare triple $\{P\}e\{Q\}$. The former is a *logically-atomic* triple [Jun+18]. The motivation for these triples is that Aneris invariants can only be opened around *atomic* steps (otherwise concurrent threads might observe invariant violations); this means we cannot use a specification $\{P\}e\{Q\}$ if we need to open an invariant to prove P, provided e is not atomic, as is the case for both get_state and update. In particular, the precondition of update requires the global state $\mathsf{GlobSt}(s_{\mathsf{g}})$, which a client is likely to keep in an invariant because it is shared by all (concurrent) replicas. Logically-atomic triples solve this problem by allowing us to open invariants when proving the pre-condition. Their informal semantics are as follows: if we know $\langle P\rangle e\langle Q\rangle^{\mathcal{N}}$, then we know that e executes without crashing (although it might not terminate) provided that P holds. When proving P we are allowed to use any invariants that are not in namespace $\mathcal{N}$. [6] We also need to show that if Q holds we can close any invariants that were open when proving P. Another point of view is that P and Q hold around a *linearization point* in e; the linearization points for `get_state` and `update` are marked with an `LP` comment in Figure 4.5.

The specification of get_state can be read as follows. Before calling get_state we should know the local state at replica i: $\mathsf{LocSt}\ (i,\ s_{\mathsf{own}},\ s_{\mathsf{for}})$; afterwards, the function returns a physical state v which is *coherent* with a logical state w, [7] the local state is now $\mathsf{LocSt}\ (i,\ s_{\mathsf{own}},\ s'_{\mathsf{for}})$ where $s'_{\mathsf{for}} \supseteq s_{\mathsf{for}}$ (reflecting the fact the set of local events is unchanged, but additional remote events might have been received), and the returned value w is the denotation of $s_{\mathsf{own}} \cup s'_{\mathsf{for}}$. Notice that the fact that the returned value is a function of the set of received events automatically gives us the safety part of eventual consistency (convergence): if two replicas have received the same set of events, then they are in the same state.

The specification of update says that we need to know both the local state $\mathsf{LocSt}\ (i,\ s_{\mathsf{own}},\ s_{\mathsf{for}})$ at replica i and the global state $\mathsf{GlobSt}(s_{\mathsf{g}})$. This is because every event needs to be tracked both locally and globally. The function does not return any meaningful value, but we do get (logical) knowledge that the set of events has expanded: locally we now know $\mathsf{LocSt}\ (i,\ s_{\mathsf{own}} \cup \{e\}\ ,\ s'_{\mathsf{for}})$ with $s_{\mathsf{for}} \subseteq s'_{\mathsf{for}}$, and globally we have $\mathsf{GlobSt}(s_{\mathsf{g}} \cup \{e\})$. The new event e ($e \notin s_{\mathsf{g}}$) has the value we updating by as its payload ($e.o = v$) and it originates at replica i. Finally, we know that this new event is more recent than any other locally-received event ($\mathrm{Maximum}(e)s_{\mathsf{own}} \cup s'_{\mathsf{for}} \cup \{e\}$), and we also know that no other event (even globally) has the new event as its dependency: $e \in \mathrm{Maximals}(s_{\mathsf{g}} \cup \{e\})$.

4.6 *Verifying StateLib*

To prove that StateLib meets its external specification we follow a proof methodology inspired by previous Aneris developments [Gon+21; Nie+22]:

1. We first model the CRDT as a state-transition system (STS), where the STS states are tuples detailing the state of the CRDT both globally and at each replica (Section 4.6.1). Transitions correspond to mutations and merges. Crucially, we show that transitions preserve *state validity*, a safety invariant from which we can derive properties of interest (e.g., causality). This is done in the meta-logic (i.e., Coq) and outside of separation logic.

[6]This is to prevent a user of StateLib from opening the global invariant, which is needed by StateLib. Invariants cannot be reopened, to preserve soundness of the logic.

[7]The *state coherence* predicate $\mathsf{StCoh}(w, v)$ links the physical and logical states. This is useful when the physical state has a more involved representation due to limitations of AnerisLang: for example, v might be a pair of pairs while w is a 3-tuple, because AnerisLang only supports pairs.

2. We then embed the STS model inside Aneris via a combination of invariants and ghost state (defined via PCMs, see Section 4.6.3). We use state validity and the properties of the relevant PCMs to show that the resource interface from Section 4.3 holds.

3. Finally, using the separation logic resources defined in the previous step, we prove that STATELIB's implementation meets its external specification (Section 4.6.4).

The rest of Section 4.6 is technical in nature, so the reader interested in an overview of our work can skip to Section 4.7. We highlight Lemmas 4.6.5 and 4.6.6, which, to our knowledge, are the first formal proofs that state-based CRDTs are causally-consistent.

4.6.1 State-Transition System Model

We model the execution of a CRDT via an STS that keeps track of the per-replica state, as well as the global state. We then show a number of safety invariants that hold for any execution of the STS. In later sections we show how the AnerisLang implementation simulates the STS, therefore inheriting its safety properties. We use the simulation to prove the resource laws in Figure 4.3.

The reader might wonder why we develop this STS model when state-based CRDTs already have a well-understood model that is lattice-based. We do this because our goal is to prove that STATELIB satisfies general functional correctness specifications that apply to both state-based CRTDs and op-based CRDTs. To this end we write our high-level specifications in terms of denotations, which talk about sets of events instead of lattice elements. This is why we need the STS model below. At an operational level, the STS model is needed to define the ghost state in Section 4.6.3 and as such is not directly exposed to the user.

We start by defining a purely logical notion of time that allows us to reason about causality in the absence of vector clocks. *Logical time* for state-based CRDTs is a triple $\mathsf{LogTime}_{\mathsf{st}} \triangleq (\mathcal{P}(\mathsf{EvId}), \subseteq, \subset)$, where $\mathsf{EvId} \triangleq \mathbb{N} \times \mathbb{N}$. Here EvId is the set of *event ids*, which are pairs (r, n) of a *replica id* and a *sequence number*, respectively. We show that $\mathsf{LogTime}_{\mathsf{st}}$ satisfies the requirements on logical time from Section 4.3.1.

Given a set $d \in \mathcal{P}(\mathsf{EvId})$ of event ids we can extract the subset that originates at a given replica id via a *section*: $\mathsf{sect}(d, i) \triangleq \{(i, n) \mid (i, n) \in d\}$. We can also compare event ids, but only if they are in the same section: $(s, n) \leq (s, n') \triangleq n \leq n'$.

We define logical events as triples $(\mathsf{op}, \mathsf{src}, \mathsf{time}) \in \mathsf{Op} \times \mathbb{N} \times \mathcal{P}(\mathsf{EvId})$. We tag each event e with the set of event ids of its *causal dependencies* $e.t$ and are then able to sort events according to causal order. For example, if $e.t = \{(1, 1)\}$ and $e'.t = \{(1, 1), (2, 1)\}$, then $e.t <_t e'.t$. Let $s \in \mathcal{P}(\mathsf{Event})$. We lift causal dependencies to sets of events: $\mathsf{deps}(s) \triangleq \bigcup_{e \in s} e.t$.

The *id* of an event e can be computed by counting the number of dependencies that originate at e's origin: $\mathsf{id}(e) = (e.s, |\mathsf{sect}(e.t, e.s)|)$. This way of computing event ids makes sense only if we assume that sequence numbers (a) start at 1 and (b) there are no "holes" in the ids stored in $e.t$. In our proof we maintain an invariant that implies these two properties.

To ensure causal consistency, we care about event sets that are closed with respect to causal dependencies. Let s be a set of events, then s is *dep-closed*, written $\mathsf{depclosed}(s)$, if $\forall e \in s, id \in e.t, \exists e' \in s, \mathsf{id}(e') = id$. For example, if $e \in s$ and $(1, 2) \in e.t$, then we must have $e' \in s$ with $\mathsf{id}(e') = (1, 2)$. Dep-closure is preserved by set union, which is key because it will allow us to link logical and physical states when we take least upper bounds of the latter.

We now define *local states*, which track the state of the CRDT at a given replica. Unlike in the implementation the replica state will not be a lattice element, but a set of events: $\mathsf{Lst} = \mathcal{P}(\mathsf{Event})$. We lift sections to local states: if $s \in \mathsf{Lst}$ then $\mathsf{sect}(s, i) \triangleq \{e \mid e.s = i\}$.

Recall that $\mathsf{numRep} \in \mathbb{N}$ denotes the number of replicas. We define *global states* $\mathsf{Gst} \triangleq \mathcal{P}(\mathsf{Event}) \times \mathsf{Lst}^{\mathsf{numRep}}$. The intuition for a global state $(s_{\mathrm{g}}, \vec{s_l}) \in \mathsf{Gst}$ is that the first component s_{g} gives us a *global view* of the system (we will ensure that s_{g} equals the union of all $s_{l,i}$). The second component $\vec{s_l}$ is a vector of length numRep containing the local state at each replica.

Definition 4.6.1 (State-transition system model)**.** The state-transition system model is $\mathcal{S} = (\mathsf{Gst}, \mathsf{init}_{\mathcal{S}}, \rightarrow_{\mathcal{S}})$. STS states are elements of Gst and $\mathsf{init}_{\mathcal{S}} \triangleq (\emptyset, \vec{\emptyset})$ is the initial state. The transition relation $\rightarrow_{\mathcal{S}} \in \mathsf{Gst} \times \mathsf{Gst}$ is defined by the following two inference rules:[8]

$$\frac{s_{l,i} = s \qquad d = \mathsf{deps}(s) \qquad n = |\mathsf{sect}(d, i)| + 1 \qquad t = \{(i, n)\} \cup d \qquad e = (\mathsf{op}, i, t)}{(s_{\mathrm{g}}, \vec{s_l}) \rightarrow_{\mathcal{S}} (s_{\mathrm{g}} \cup \{e\}, \vec{s_l}[i \mapsto s \cup \{e\}])}\ \mathsf{TUpdate}$$

$$\frac{s \subseteq s_{l,j} \qquad \mathsf{depclosed}(s)}{(s_{\mathrm{g}}, \vec{l}) \rightarrow_{\mathcal{S}} (s_{\mathrm{g}}, \vec{l}[i \mapsto s_{l,i} \cup s])}\ \mathsf{TMerge}$$

The TUpdate rule models the execution of a new operation at a particular replica. The premises say that the local state at replica i is s and d is the set of dependencies of all events in s. Then we compute the sequence number for the new event: since it originates in i this needs to be exactly one larger than the number of dependencies in d that come from i, hence $n = |\mathsf{sect}(d, i)| + 1$. Then we build a timestamp for the new event: every (old) event in s should be a causal dependency of the new event, plus the new event's id is also a dependency, so $t = \{(i, n)\} \cup d$. Finally we build the new event $e = (\mathsf{op}, i, t)$. Given all the above, we can take a step in the STS from a state $(s_{\mathrm{g}}, \vec{s_l})$ to a state that includes the new event e. Because e is new, it should be added both to the global state and the local state for i. The notation $\vec{s_l}[i \mapsto s']$ stands for the vector that is like $\vec{s_l}$ except that the ith entry is now s'.

The TMerge rule models merge operations where a replica updates its state by receiving and merging a (potentially old) state that was sent by another replica. In the rule, we start with some subset s of the local state at replica j. It is crucial that said subset be dep-closed so that we can preserve causality. In the rule's conclusion, we merge s with the state at replica i. That is, we can think of this rule as saying that replica j transmitted its state to replica i, which subsequently merged it. Finally, notice that the fact that s is a subset of $s_{l,j}$ and not exactly $s_{l,j}$ allows us to model the delay imposed by the network on message transmission — the fact that s is a *dep-closed* subset of $s_{l,j}$ means that s is a version of the state of the j^{th} replica from the past.

4.6.2 Safety Invariants

The goal of the STS model is to allow us to show a number of safety invariants about the execution of the system. We do this through the notions of *local* and *global state validity.* Let $s \in \mathsf{Lst}$. Then s is a *valid local state*, written $\mathsf{LocStValid}(s)$, if all the following hold:

[8]As usual $\rightarrow_{\mathcal{S}}^{*}$ denotes the reflexive transitive closure of $\rightarrow_{\mathcal{S}}$.

(DepClosed)	$\mathsf{depclosed}(s)$
(SameOrigComp)	$\forall i,\ \forall e\ e' \in \mathsf{sect}(s,i), e.t <_t e'.t \vee e.t = e'.t \vee e'.t <_t e.t$
(ExtId)	$\forall e\ e' \in s, \mathsf{id}(e) = \mathsf{id}(e') \Rightarrow e = e'$
(ExtTime)	$\forall e\ e' \in s, e.t = e'.t \Rightarrow e = e'$
(OrigRange)	$\forall e \in s, e.s < \mathtt{numRep}$
(SeqIdComplete)	$\forall e \in s, n \in \mathbb{N}, 0 < (e.s, n) \leq \mathsf{id}(e) \Rightarrow (e.s, n) \in e.t$
(SeqIdNon0)	$\forall e \in s, r\ n \in \mathbb{N}, \mathsf{id}(e) = (r, n) \Rightarrow 0 < n$
(EvIdMon)	$\forall e\ e' \in s, e.s = e'.s \Rightarrow e.t \leq_t e'.t \Rightarrow \mathsf{id}(e) \leq \mathsf{id}(e')$
(EvIdIncl)	$\forall e \in s, \mathsf{id}(e) \in e.t$
(EvIdTime)	$\forall e\ e' \in s, \mathsf{id}(e) \in e'.t \Rightarrow e.t \leq_t e'.t$

The different requirements on valid local states are as follows. (DepClosed) requires that a valid state s be also dep-closed. (SameOrigComp) says that events with the same origin can be totally ordered by timestamp ordering. (ExtId) and (ExtTime) say that events with equal id, resp. timestamp, must be equal. (OrigRange) ensures that replica ids are in the expected range. (SeqIdComplete) says that if $e \in s$ and e's id is e.g. $(4, 10)$, then all timestamps in the range $(4, 1) \ldots (4, 9)$ must also be in e's dependencies. (SeqIdNon0) says that all event ids have a sequence number that starts at 1. (EvIdMon) requires that timestamps ordering and event id ordering agree. (EvIdIncl) says that an event id must be included in the event's dependencies. Finally, (EvIdTime) requires that if e's id is in the dependencies of e' then e must have in fact happened before e' according to timestamp ordering. The definition of local state validity has been simplified with respect to prior developments [Gon+21; Nie+22]. This is because these works use vector clocks as their notion of logical and physical time, whereas we only track time logically via sets of dependencies.

From local state validity we obtain a number of derived lemmas. For example, we can relate dep-closed and causally-closed subsets:

Lemma 4.6.2. *Let* $s \subseteq s_g$ *where* $\mathsf{depclosed}(s)$ *and* $\mathsf{LocStValid}(s_g)$*. Then* $s \subseteq_{\mathsf{cc}} s_g$*.*

Proof. We have to show that if $e \in s$, $e' \in g$ and $e'.t \leq_t e.t$ we also have $e' \in s$. From $\mathsf{LocStValid}(g)$ we get $\mathsf{depclosed}(g)$. From Lemma **??** we know that $\mathsf{id}(e') \in e.t$. Since $e \in s$, which is dep-closed, then $\exists e'' \in s$ such that $\mathsf{id}(e'') = \mathsf{id}(e')$. Since both e' and e'' are in g, and $\mathsf{LocStValid}(g)$, then we can apply (ExtId) to conclude $e' = e''$, which gives us $e' \in s$, as needed. □

Let s be a set of events such that $\mathsf{LocStValid}(s)$. The following lemma says we can always generate a "fresh" event id by incrementing the sequence number.

Lemma 4.6.3. *Let* $d = \mathsf{deps}(s)$ *and* $n = |\mathsf{sect}(d, i)| + 1$*. Then* $(i, n) \notin d$*.*

Proof. Since s is valid, we know $\mathsf{depclosed}(s)$. Suppose that $(i, n) \in \mathsf{deps}(s)$. Then we must have $e \in s$ with $\mathsf{id}(e) = (i, n)$. Unfolding the definition of event id, we get $|\mathsf{sect}(e.t, i)| = n$. But we have $e.t \subseteq d$, so $\mathsf{sect}(e.t, i) \subseteq \mathsf{sect}(d, i)$. This means that $n = |\mathsf{sect}(e.t, i)| \leq |\mathsf{sect}(d, i)| < n$, a contradiction. □

Now we define validity of global states. Let $q = (s_g, \vec{s_l}) \in \mathsf{Gst}$. Then q is a *valid global state*, written $\mathsf{GlobStValid}(q)$, if all the following hold:

(InclLocal) $\quad s_g = \bigcup_{1 \le i \le \mathtt{numRep}} s_{l,\,i}$

(InclOrig) $\quad \forall e \in s_g, e \in s_{l,\,e.s}$

(GlobValid) $\quad \mathsf{LocStValid}(s_g)$

(LocValid) $\quad \forall 1 \le i \le \mathtt{numRep}, \mathsf{LocStValid}(s_{l,\,i})$

Given a global state $(s_g, \vec{s_l})$, global state validity amounts to requiring that s_g be the union of the $s_{l,\,i}$ (InclLocal), that events be present in the local state from which they supposedly originate (InclOrig), that the global state s_g itself be valid if treated as a local state (GlobValid) and that each local state be valid (LocValid). The definition of global state validity is essentially unchanged from prior work [Gon+21; Nie+22].

The definitions of local and global state validity are motivated by two desiderata: they must be *invariants*, i.e. hold for all reachable states (including the initial state); and they must imply the CRDT resource lemmas from Figure 4.3 "at the model level".

Theorem 4.6.4 (Validity invariant). *Let* $q \in \mathsf{Gst}$ *such that* $\mathsf{init}_{\mathcal{S}} \rightarrow^*_{\mathcal{S}} q$. *Then* $\mathsf{GlobStValid}(q)$.

Proof. The proof is by induction on the derivation of $\mathsf{init}_{\mathcal{S}} \rightarrow^*_{\mathcal{S}} s$. First note that $\mathsf{GlobStValid}(\mathsf{init}_{\mathcal{S}})$, which holds trivially because all local states are empty. In general, we have $g \rightarrow_{\mathcal{S}} g'$ with $\mathsf{GlobStValid}(g)$, and have to show that both TUpdate and TMerge preserve global validity. We do this by unfolding definitions. We only show preservation of (DepClosed) below: the other predicates are handled similarly and the reader can consult our mechanized proof for details.

Case (TUpdate): we need to prove $\mathsf{LocStValid}(g \cup \{e\})$ and $\mathsf{LocStValid}(s \cup \{e\})$, where g, s and e are as in Definition 4.6.1. Since we are only showing dep-closedness, this amounts to showing $\mathsf{depclosed}(g \cup \{e\})$ and $\mathsf{depclosed}(s \cup \{e\})$. For the latter, notice that $\mathsf{GlobStValid}(g)$ implies $\mathsf{depclosed}(s)$, so it suffices to show that $\forall r \in e.t, \exists e' \in s \cup \{e\}$ such that $\mathsf{id}(e') = r$. Let r be one of e's dependencies, so by construction $r \in \{(i, n) \cup \mathsf{deps}(s)\}$. If $r = (i, n)$ then we can choose $e' = e$, because $\mathsf{id}(e') = (i, |\mathsf{sect}(e'.t, i)|) = (i, |\mathsf{sect}(\mathsf{deps}(s), i) \cup \{(i, n)\}|) = (i, |\mathsf{sect}(\mathsf{deps}(s), i)| + 1) = (i, n)$, where the next-to-last equality is justified by Lemma 4.6.3. If $r \in \mathsf{deps}(s)$, then the result follows because s is dep-closed. Finally, to show $\mathsf{depclosed}(g \cup \{e\})$ we proceed as above and note that $s \subseteq g$.

Case (TMerge): here we only need to show $\mathsf{depclosed}(\vec{l_i} \cup s)$, where $s \subseteq \vec{l_j}$ and $\mathsf{depclosed}(s)$. This follows because the union of two dep-closed sets is also dep-closed. □

We use validity to show model-level counterparts of the lemmas in Figure 4.3. Intuitively, ownership of $\mathsf{GlobSt}(s_g)$ tells us that the global state is $(s_g, \vec{s_l})$, whereas if we know $\mathsf{GlobSnap}(s'_g)$ all we can say is that $s'_g \subseteq s_g$. Similarly, ownership of $\mathsf{LocSt}\,(i,\, s_{\mathsf{own}},\, s_{\mathsf{for}})$ corresponds to knowing that the local state $s_{l,\,i}$ (which we can assume to be valid) equals $s_{\mathsf{own}} \uplus s'_{\mathsf{for}}$ (disjoint union), with $s_{\mathsf{for}} \subseteq s'_{\mathsf{for}}$. The local snapshot $\mathsf{LocSnap}(i, s_{\mathsf{own}}, s_{\mathsf{for}})$ give us $s_{\mathsf{own}} \uplus s_{\mathsf{for}} \subseteq s_{l,\,i}$. With this analogy in mind, here is the model-level version of causality.

Lemma 4.6.5 (Model-level causality). *Let* $q = (s_g, \vec{s_l})$ *and* $\mathsf{GlobStValid}(q)$. *Also let* $e \in s \subseteq_{\mathsf{cc}} s_{l,\,i}$ *and* $e' \in s_g$, *with* $e'.t <_t e.t$. *Then* $e' \in s$.

Proof. From $\mathsf{GlobStValid}(q)$ we can conclude $\mathsf{LocStValid}(s_{l,i})$, which in turn gives us $\mathsf{depclosed}(s_{l,i})$. Since $s_{l,i} \subseteq s_{\mathrm{g}}$, by Lemma 4.6.2 we get $s_{l,i} \subseteq_{\mathsf{cc}} s_{\mathrm{g}}$. Since we assumed $s \subseteq_{\mathsf{cc}} s_{l,i}$, we can use transitivity of $\subseteq_{\mathsf{cc}}$ to conclude $s \subseteq_{\mathsf{cc}} s_{\mathrm{g}}$. This implies the conclusion. □

4.6.3 Separation Logic Encoding

The next step in the proof is to encode the validity invariant using separation logic. We do this using a combination of Iris invariants and resources [Jun+18]. Recall that Iris invariants are propositions that hold throughout the execution of the operational semantics and resources are elements of partial commutative monoids (PCMs).

Here we use resources whose ownership reveals what state the system is partially in (e.g. the set of events received by a specific replica). In particular, we use three main PCM constructions, which we review below: [9]

1. The *authoritative* PCM *Auth*($\mathbb{M}$), where $\mathbb{M}$ is itself a PCM. Given a PCM $\mathbb{X}$, we can define the *extension order* on elements of the carrier as follows: $x \leq_{\mathbb{X}} y \triangleq \exists z, x \cdot z = y$. The authoritative construction gives us two kinds of resources: a *full part* $\bullet_{\mathbb{M}}\, g$ and one or more *fragmental parts* $\circ_{\mathbb{M}}\, s$, where $g, s \in \mathbb{M}$. Ownership of the full part is exclusive, while ownership of a fragment $\circ_{\mathbb{M}}\, s$ is exclusive or persistent depending on whether ownership of s is exclusive or persistent in $\mathbb{M}$. The fragmental parts are guaranteed to be smaller than the full part according to extension order, so that if we own $\boxed{\bullet_{\mathbb{M}}\, g}^{\gamma} * \boxed{\circ_{\mathbb{M}}\, s}^{\gamma}$ we can conclude $s \leq_{\mathbb{M}} g$. We also have that $\circ_{\mathbb{M}}\, g \cdot \circ_{\mathbb{M}}\, s = \circ_{\mathbb{M}}\, (g \cdot s)$.

2. The *fractional* PCM *Frac*(X), where X is a carrier set. Elements of this monoid are of the form s^p, where $s \in X$ and $p \in \mathbb{Q}_{(0,1]}$. This PCM allows us to split and re-combine fractions of a resource: $s^{p+q} = s^p \cdot s^q$. We also know that if we own multiple fractions then they must add to less than 1. This is useful to e.g. make a proposition exclusive (non-duplicable) by defining it as a fraction greater than $\frac{1}{2}$ as no two copies of such a resource can be owned separately. We also know that all fractions agree on the underlying element: $\boxed{s^p}^{\gamma} * \boxed{r^q}^{\gamma}$ implies $s = r$.

3. The *monotone* PCM $\textsc{Mono}(R)$, where $R \subseteq X \times X$ is a pre-order on a carrier set X [TB21]. This PCM allows us to lift R to the extension order of $\textsc{Mono}(R)$: any $x \in X$ can be injected into $\textsc{Mono}(R)$ via a $\mathsf{principal}_R$ function such that $xRy \iff \mathsf{principal}_R(x) \leq_{\textsc{Mono}(R)} \mathsf{principal}_R(y)$. Combining this with the authoritative PCM gives us a monoid *Auth*($\textsc{Mono}(R)$) where if we know $\boxed{\bullet_{\mathbb{M}}\, \mathsf{principal}_R(g)}^{\gamma} * \boxed{\circ_{\mathbb{M}}\, \mathsf{principal}_R(s)}^{\gamma}$ we can conclude sRg. We instantiate this construction with $R = \subseteq_{\mathsf{cc}}$.

We use the defined invariants and resources to prove the interface described in Figure 4.3. In this section, we sketch out proofs for some of the interface lemmas.

The global invariant uses the predicate **GI** below. The predicate states that there exists a (model-level) global state h which is valid. Furthermore, it asserts ownership of global and local resources defined by the predicates $\mathbf{GR}(s_{\mathrm{g}})$ and $\mathbf{LR}(i, s_{l,i})$, respectively.

$$\mathbf{GI} \triangleq \exists h \in \mathsf{Gst}. h = (s_{\mathrm{g}}, \vec{s_l}) * \mathsf{GlobStValid}(h) * \mathbf{GR}(s_{\mathrm{g}}) * \mathop{\Large\ast}_{i=1}^{\mathsf{numRep}} \mathbf{LR}(i, s_{l,i})$$

Given the above definition and an invariant name ι, we can instantiate the $\mathsf{GlobInv}$ predicate from Figure 4.3 by allocating an Aneris invariant stating that **GI** holds after every execution step: $\mathsf{GlobInv} \triangleq \boxed{\mathbf{GI}}^{\iota}$.

[9]We use a few additional PCMs in the Coq formalization but elide those additional structures here for the sake of brevity.

The *global resource* predicate $\mathbf{GR}(s_{\mathsf{g}})$ asserts ownership of two pieces of ghost state, both of which precisely track the value of s_{g}: $\mathbf{GR}(s_{\mathsf{g}}) \triangleq \boxed{s_{\mathsf{g}}^{\frac{1}{3}}}^{\gamma_{\mathsf{gst}}} * \boxed{\bullet_{\mathbb{S}}\, s_{\mathsf{g}}}^{\gamma_{\mathsf{gsnap}}}$.

The ghost state $s_{\mathsf{g}}^{\frac{1}{3}}$ is drawn from the *Frac*(Gst) PCM. Its purpose is to track the global set of events. The remaining $\frac{2}{3}$ fraction is kept *outside* of the invariant as the user-facing resource GlobSt from Figure 4.3: $\mathsf{GlobSt}(s_{\mathsf{g}}) \triangleq \boxed{s_{\mathsf{g}}^{\frac{2}{3}}}^{\gamma_{\mathsf{gst}}}$. Because the fraction in GlobSt is greater than a half, we can prove that $\mathsf{GlobSt}(s_{\mathsf{g}})$ is exclusive (GlobStExcl, Figure 4.3).

The second part of $\mathbf{GR}(s_{\mathsf{g}})$ asserts ownership of $\bullet_{\mathbb{S}}\, s_{\mathsf{g}}$. Here, $\mathbb{S}$ is the PCM of finite sets of events, with set union as composition. This means that $p \leq_{\mathbb{S}} q$ iff $p \subseteq q$. Consequently, $\boxed{\bullet_{\mathbb{S}}\, s_{\mathsf{g}}}^{\gamma_{\mathsf{gsnap}}} * \boxed{\circ_{\mathbb{S}}\, s'_{\mathsf{g}}}^{\gamma_{\mathsf{gsnap}}}$ implies $s'_{\mathsf{g}} \subseteq s_{\mathsf{g}}$. We keep the full part in the invariant and use the fragmental part to define global snapshots: $\mathsf{GlobSnap}(s) \triangleq \boxed{\circ_{\mathbb{S}}\, s'_{\mathsf{g}}}^{\gamma_{\mathsf{gsnap}}}$. Note that these fragmental parts are persistent (duplicable) as the set union operation is idempotent.

The next step is to define the local resources predicate $\mathbf{LR}(i, s)$ which tracks in the invariant the local resources for replica i:

$$\begin{aligned}
\mathbf{LR}(i, s) \triangleq\ & \exists s_{\mathsf{own}}\ s_{\mathsf{for}}\ s_{\mathsf{sub}},\ s = s_{\mathsf{own}} \cup s_{\mathsf{for}} * s_{\mathsf{own}} \cup s_{\mathsf{sub}} \subseteq_{\mathsf{cc}} s_{\mathsf{own}} \cup s_{\mathsf{for}} \\
& * \mathsf{LocEv}(i, s_{\mathsf{own}}) * \mathsf{ForEv}(i, s_{\mathsf{for}}) * \mathsf{ForEv}(i, s_{\mathsf{sub}}) * \boxed{s_{\mathsf{own}}^{\frac{1}{3}}}^{\gamma_{\mathsf{own}_i}} * \boxed{s_{\mathsf{for}}^{\frac{1}{2}}}^{\gamma_{\mathsf{for}_i}} * \boxed{s_{\mathsf{sub}}^{\frac{1}{3}}}^{\gamma_{\mathsf{sub}_i}} \\
& * \boxed{\bullet_{\mathbb{M}}\, \mathsf{principal}_{\subseteq_{\mathsf{cc}}}(s_{\mathsf{own}} \cup s_{\mathsf{for}})}^{\gamma_{\mathsf{ccfor}_i}} * \boxed{\bullet_{\mathbb{M}}\, \mathsf{principal}_{\subseteq_{\mathsf{cc}}}(s_{\mathsf{own}} \cup s_{\mathsf{sub}})}^{\gamma_{\mathsf{ccsub}_i}}
\end{aligned}$$

The predicate $\mathbf{LR}(i, s)$ says that s can be broken up into two (disjoint) sets s_{own} and s_{for}. The sets are disjoint because every event in s_{own} originates at replica i, whereas all events in s_{for} originate outside of replica i. This is expressed by the predicates $\mathsf{LocEv}(i, p) = \forall e \in p. e.s = i$ and $\mathsf{ForEv}(i, p) = \forall e \in p.\ e.s \neq i$, respectively. Additionally, there is a third set s_{sub} which is a subset of s_{for} (this is implied by $s_{\mathsf{own}} \cup s_{\mathsf{sub}} \subseteq_{\mathsf{cc}} s_{\mathsf{own}} \cup s_{\mathsf{for}}$). The intuition for s_{own} and s_{for} is that they precisely track the set of events that have been delivered at replica i and originate at i or outside of i, respectively. However, the user at replica i is not aware of all those events: specifically, while the user is aware of (the effects of) all its local events, it might not have observed all events that originate outside of i. The set s_{sub} precisely tracks the set of remote events, $\mathsf{ForEv}(i, s_{\mathsf{sub}})$, that replica i has observed and is aware of. The tracking of all these event sets is precise because of the ownership of the fractions $\boxed{s_{\mathsf{own}}^{\frac{1}{3}}}^{\gamma_{\mathsf{own}_i}} * \boxed{s_{\mathsf{for}}^{\frac{1}{2}}}^{\gamma_{\mathsf{for}_i}} * \boxed{s_{\mathsf{sub}}^{\frac{1}{3}}}^{\gamma_{\mathsf{sub}_i}}$. Additionally, the invariant has "read-only" access to the three pieces of ghost state because it does not possess the full fractions. Before pressing on with the definition of $\mathbf{LR}(i, s)$, let us look at the definition of local state and snapshot from Figure 4.3:

$$\begin{aligned}
\mathsf{LocSt}\,(i,\ s_{\mathsf{own}},\ s_{\mathsf{sub}}) &\triangleq \boxed{s_{\mathsf{own}}^{\frac{1}{3}}}^{\gamma_{\mathsf{own}_i}} * \boxed{s_{\mathsf{sub}}^{\frac{2}{3}}}^{\gamma_{\mathsf{sub}_i}} * \mathsf{LocSnap}(i, s_{\mathsf{own}}, s_{\mathsf{sub}}) \\
\mathsf{LocSnap}(i, s_{\mathsf{own}}, s_{\mathsf{sub}}) &\triangleq \mathsf{LocEv}(i, s_{\mathsf{own}}) * \mathsf{ForEv}(i, s_{\mathsf{sub}}) * \boxed{\circ_{\mathbb{M}}\, \mathsf{principal}_{\subseteq_{\mathsf{cc}}}(s_{\mathsf{own}} \cup s_{\mathsf{sub}})}^{\gamma_{\mathsf{ccsub}_i}}
\end{aligned}$$

Notice that ownership of the local state resource gives us knowledge of some but not all of the missing fractions for s_{own} and s_{sub}: $\boxed{s_{\mathsf{own}}^{\frac{1}{3}}}^{\gamma_{\mathsf{own}_i}} * \boxed{s_{\mathsf{sub}}^{\frac{2}{3}}}^{\gamma_{\mathsf{sub}_i}}$. This corresponds to our intuition that s_{sub} tracks the set of events the user is aware of. Notice that local state does not contain a fraction of s_{for}, because otherwise the library could not accept remote updates in a background thread.

We next explain the use of the PCM $\mathbb{M} \triangleq \textsc{Mono}(\subseteq_{\mathsf{cc}})$, where $\subseteq_{\mathsf{cc}}$: Event $\times$ Event. We use two instances of this PCM, each under a different family of ghost names: $\gamma_{\mathsf{ccsub}_i}$ and $\gamma_{\mathsf{ccfor}_i}$. The local state contains just $\gamma_{\mathsf{ccsub}_i}$ and the invariant contains both $\gamma_{\mathsf{ccsub}_i}$ and $\gamma_{\mathsf{ccfor}_i}$. The intuition for holding the fragmental part $\circ_{\mathbb{M}}\, \mathsf{principal}_{\subseteq_{\mathsf{cc}}}(s_{\mathsf{own}} \cup s_{\mathsf{sub}})$ is to allow us to prove inclusion of local snapshots (LocSnapIncl, Figure 4.3).

We can now sketch the proof of the causality lemma. This proof is representative of our methodology: use the properties of the different PCMs to identify parts of a (valid) global state we are currently in, and then rely on a model-level lemma to get the result we want.

Lemma 4.6.6 (Causality, Figure 4.3)**.** $\mathsf{GlobInv} \mathrel{-\!\!*} \mathsf{LocSt}\,(i,\, o,\, s) \mathrel{-\!\!*} \mathsf{GlobSnap}(h) \mathrel{-\!\!*} \Rrightarrow_{\iota} \mathsf{LocSt}\,(i,\, o,\, s) \,*$ $\forall e\; e', e \in h \Rightarrow e' \in o \cup s \Rightarrow e <_t e' \Rightarrow e \in o \cup s.$

Proof. We open the global invariant and learn that the current global state $(s_{\mathsf{g}}, \vec{s_l})$ is valid. We also obtain global resources $\boxed{s_{\mathsf{g}}^{\frac{1}{3}}}^{\gamma_{\mathsf{gst}}} * \boxed{\bullet_{\mathbb{S}}\, s_{\mathsf{g}}}^{\gamma_{\mathsf{gsnap}}}$ and local resources $\boxed{s_{\mathsf{own}}^{\frac{1}{3}}}^{\gamma_{\mathsf{own}_i}} * \boxed{s_{\mathsf{sub}}^{\frac{1}{3}}}^{\gamma_{\mathsf{sub}_i}}$, where $s_{\mathsf{own}} \cup s_{\mathsf{sub}} \subseteq_{\mathsf{cc}} s_{\mathsf{own}} \cup s_{\mathsf{for}} = s_{l,\,i}$. From $\mathsf{LocSt}\,(i,\, o,\, s)$ we get $\boxed{o^{\frac{1}{3}}}^{\gamma_{\mathsf{own}_i}} * \boxed{s^{\frac{2}{3}}}^{\gamma_{\mathsf{sub}_i}}$, which tells us that $o = s_{\mathsf{own}}$ and $s = s_{\mathsf{sub}}$. From $\mathsf{GlobSnap}(h)$ we get $\boxed{\circ_{\mathbb{S}}\, h}^{\gamma_{\mathsf{gsnap}}}$, which tells us that $h \subseteq s_{\mathsf{g}}$. This means we have $e \in s_{\mathsf{g}}$ and e' is in a causally-closed subset of $s_{l,\,i}$, so we can finish by applying Lemma 4.6.5. □

There are two additional resources in the definition of $\mathbf{LR}(i, s)$: $\boxed{s_{\mathsf{for}}^{\frac{1}{2}}}^{\gamma_{\mathsf{for}_i}}$ and $\boxed{\bullet_{\mathbb{M}}\, \mathsf{principal}_{\subseteq_{\mathsf{cc}}}(s_{\mathsf{own}} \cup s_{\mathsf{for}})}^{\gamma_{\mathsf{ccfor}_i}}$. These are used in the definition of the lock invariant and socket protocol, respectively. We explain them in the next section.

In addition to proving the resource lemmas from Figure 4.3, we also need to show that the PCMs we have chosen can make frame-preserving updates that are compatible with the two transitions (TUpdate and TMerge) from Definition 4.6.1. We refer to reader to our Coq formalization for details.

4.6.4 *Safety Proof*

StateLib's `init` function allocates a reference with the CRDT's initial state and spawns two concurrent threads: `apply` receives states from other replicas and merges them with the current state, and `broadcast` regularly sends the current state to all other replicas. Access to the (shared) local state is coordinated via a spinlock. The associated *lock invariant* [BB17], defined by the predicate $\mathbf{LI}(i, \ell)$ below, is the key ingredient of the library's safety proof (ℓ is the memory location holding the CRDT's state). When a thread acquires the lock, it gets to assume $\mathbf{LI}(i, \ell)$; unlike a regular invariant, which needs to be restored after a single atomic step, a lock invariant need not be restored until the thread releases the lock.

$$\begin{aligned}\mathbf{LI}(i, \ell) \triangleq \exists st\; s_{\mathsf{own}}\; s_{\mathsf{for}}.\; & \mathsf{LocEv}(i, s_{\mathsf{own}}) * \mathsf{ForEv}(i, s_{\mathsf{for}}) * \ell \mapsto_{\mathsf{ip}} st \\ & * [\![s_{\mathsf{own}} \cup s_{\mathsf{for}}]\!] = st * \boxed{s_{\mathsf{own}}^{\frac{1}{3}}}^{\gamma_{\mathsf{own}_i}} * \boxed{s_{\mathsf{for}}^{\frac{1}{2}}}^{\gamma_{\mathsf{for}_i}}\end{aligned}$$

The lock invariant says that the CRDT's state is always the denotation of some set of events $s_{\mathsf{own}} \cup s_{\mathsf{for}}$ (ip is the IP address of replica i). Additionally, the invariant holds resources $\boxed{s_{\mathsf{own}}^{\frac{1}{3}}}^{\gamma_{\mathsf{own}_i}} *$ $\boxed{s_{\mathsf{for}}^{\frac{1}{2}}}^{\gamma_{\mathsf{for}_i}}$ which guarantee that the we are "in sync" with the logical state recorded for replica i in the global invariant GlobInv. Notice the lock invariant keeps γ_{for_i} and not γ_{sub_i} because the library knows exactly the set of foreign events that have been processed so far. The table below summarizes where the different fractions of γ_{own_i}, γ_{for_i} and γ_{sub_i} are kept:

	γ_{own_i}	γ_{for_i}	γ_{sub_i}
Global invariant	$\frac{1}{3}$	$\frac{1}{2}$	$\frac{1}{3}$
Lock invariant	$\frac{1}{3}$	$\frac{1}{2}$	
Local state	$\frac{1}{3}$		$\frac{2}{3}$

As part of the proof we also need to define StateLib's socket protocol $\mathbf{SP}(i, st)$; i.e. a predicate that holds for all states st received by a replica (which dually creates a proof obligation whenever a replica messages others). The abridged version below assumes that st is already deserialized and that i is the replica id of the message's sender:

$$\begin{aligned}\mathbf{SP}(i, st) \triangleq \exists s'_{\mathsf{own}}\ s'_{\mathsf{for}}.\ &\mathsf{LocEv}(i, s'_{\mathsf{own}}) * \mathsf{ForEv}(i, s'_{\mathsf{for}}) * [\![s'_{\mathsf{own}} \cup s'_{\mathsf{for}}]\!] = st \\ &* \mathsf{LocStValid}(s'_{\mathsf{own}} \cup s'_{\mathsf{for}}) * \boxed{\circ_{\mathbb{M}}\ \mathsf{principal}_{\subseteq_{\mathsf{cc}}}(s'_{\mathsf{own}} \cup s'_{\mathsf{for}})}^{\gamma_{\mathsf{ccfor}_i}}\end{aligned}$$

The socket protocol assumes that the received state st is the denotation of the union $s'_{\mathsf{own}} \cup s'_{\mathsf{for}}$, where s'_{own} and s'_{for} are event sets that are local and foreign, respectively, relative to the message's sender (not relative to the receiver). Additionally, we know that $s'_{\mathsf{own}} \cup s'_{\mathsf{for}}$ is a causally closed subset of the events recorded at replica i (in particular, we know the sender is not accidentally including events that have not been previously recorded).

Since both `apply` and `broadcast` recurse forever, their specifications are not very interesting: in particular, we do not care about their post-conditions. We do care about preserving the lock and global invariants as they execute. We briefly sketch the proof of `apply`. Before `apply` updates the CRDT state via `st := merge !st st'`, we know that all the following hold: the global invariant $\mathsf{GlobInv}$, the lock invariant $\mathbf{LI}(i, \mathtt{st})$, and the socket protocol $\mathbf{SP}(j, st')$, where i and j are the ids of the local and sender replicas, respectively, and $i \neq j$. We open the lock invariant and get $\mathtt{!st} = [\![s_{\mathsf{own}} \cup s_{\mathsf{for}}]\!] * \boxed{s_{\mathsf{own}}^{\frac{1}{3}}}^{\gamma_{\mathsf{own}_i}} * \boxed{s_{\mathsf{for}}^{\frac{1}{2}}}^{\gamma_{\mathsf{for}_i}}$. Similarly, from the socket protocol we know that $\mathtt{st'} = [\![s'_{\mathsf{own}} \cup s'_{\mathsf{for}}]\!] * \mathsf{LocStValid}(s'_{\mathsf{own}} \cup s'_{\mathsf{for}}) * \boxed{\circ_{\mathbb{M}}\ \mathsf{principal}_{\subseteq_{\mathsf{cc}}}(s'_{\mathsf{own}} \cup s'_{\mathsf{for}})}^{\gamma_{\mathsf{ccfor}_i}}$. We would like to apply the coherence lemma ($\mathsf{MergeCoh}$), which tells us that merging two states is the same as (1) merging the corresponding events that generated those states, and (2) then taking the denotation of the union of the event sets. The premises of ($\mathsf{MergeCoh}$) all follow from local state validity after opening the global invariant, and from the fact that $\boxed{\circ_{\mathbb{M}}\ \mathsf{principal}_{\subseteq_{\mathsf{cc}}}(s'_{\mathsf{own}} \cup s'_{\mathsf{for}})}^{\gamma_{\mathsf{ccfor}_i}}$ proves that the events we are merging have been previously recorded. The resulting logical state is $\boxed{s_{\mathsf{own}}^{\frac{1}{3}}}^{\gamma_{\mathsf{own}_i}} * \boxed{(s_{\mathsf{for}} \cup s'_{\mathsf{own}} \cup \{e \in s'_{\mathsf{for}} | e.s \neq i\})^{\frac{1}{2}}}^{\gamma_{\mathsf{for}_i}}$.

The proof of `get_state` uses the lock invariant to conclude that the returned state is the denotation of the set of events received so far. It then uses the global invariant: specifically the relation $s_{\mathsf{own}} \cup s_{\mathsf{sub}} \subseteq_{\mathsf{cc}} s_{\mathsf{own}} \cup s_{\mathsf{for}}$ in the definition of $\mathbf{LR}(i, s)$ to update the local state to $\mathsf{LocSt}\,(i,\ o,\ f')$. The proof of `update` uses the preservation of global validity under a $\mathsf{TUpdate}$ transition (Theorem 4.6.4) to show that $\mathsf{GlobInv}$ is preserved. We refer the reader to our formalization for additional details.

4.7 *Verified CRDTs*

To test StateLib we verified five example CRDTs from the literature: grow-only counter (g-counter), grow-only set, product combinator, map combinator, and positive-negative counter (pn-counter). We also verified a closed program consisting of client code that uses the pn-counter. This closed program appears in Nieto et al. [Nie+22], and we were able to swap out their op-based pn-counter for our state-based version without modifying their safety proof.[10] We used the closed example as a case study in giving our state-based CRDT the same specification as its op-based counterpart, hiding implementation details in the process.

In this section we focus on describing the pn-counter and the closed program. A pn-counter is a data structure that supports two operations: `add(z)` adds the integer `z`, which may be negative, to the counter, and `get_value` returns the counter's current value. The initial value is 0. The denotation for pn-counter used in Nieto et al. [Nie+22] is $[\![s]\!] = \sum_{e \in s} e.o$. We implemented the counter as a wrapper over the product `prod(g-counter, g-counter)`. We now explain how `g-counter` and `prod` work.

The g-counter is a simpler version of pn-counter where we can only add non-negative numbers. Operations are of the form `add(n)` with $n \in \mathbb{N}$. If N is the number of CRDT replicas, the lattice state is a vector $\vec{c}$ with N entries. The ith entry tracks the contribution of replica i to the counter's state. The initial value is the 0-vector of length N. The mutator is defined as $\mathsf{mut}(\vec{c}, e) = s[e.s \mapsto \vec{c}_{e.s} + e.o]$, and merges are done by taking the maximum of two vectors pointwise. The denotation $[\![s]\!]$ is the pointwise sum of all the vectors in s.

Given two CRDTs $C_A = (\mathsf{Op}_A, \mathsf{St}_A, \mathsf{init}_A, \mathsf{mut}_A, \mathsf{merge}_A)$ and $C_B = (\mathsf{Op}_B, \mathsf{St}_B, \mathsf{init}_B, \mathsf{mut}_B, \mathsf{merge}_B)$, their product is $C_{A\times B} = (\mathsf{Op}_A \times \mathsf{Op}_B, \mathsf{St}_A \times \mathsf{St}_B, \mathsf{init}_A \times \mathsf{init}_B, \mathsf{mut}_{A\times B}, \mathsf{merge}_{A\times B})$. Both the mutator and merge function operate in a component-wise fashion. The denotation $[\![s]\!]$ splits s into two sets s_A and s_B of A-events and B-events, respectively. It then computes the pair $([\![s_A]\!]_A, [\![s_B]\!]_B)$. Additionally, our product combinator is parameterized by a predicate $P \subseteq \mathsf{Op}_A \times \mathsf{Op}_B$ that all product operations must satisfy, so to be more precise the set of operations of a product is $\{q \in \mathsf{Op}_A \times \mathsf{Op}_B \mid P(q)\}$. This predicate is useful for defining the pn-counter because we will restrict product operations to alter at most one of the two components.

We then implement the pn-counter as a wrapper over the product `prod(g-counter, g-counter)`. Specifically, we wrap the product's `get_state` and `update` as follows (`sum_entries` is the function that sums all the entries of a vector):

```
let pncounter_add z =
  if z >= 0 then prod_update((z, 0))
            else prod_update((0, -z))
```

```
let pncounter_get_value () =
  let (v1, v2) = prod_get_state () in
  (sum_entries v1) - (sum_entries v2)
```

In the instantiation of the product, we choose $P(o_1, o_2) \triangleq o_1 = 0 \vee o_2 = 0$, giving us a bijection between pn-counter operations (integers) and product operations (pairs of naturals, at least one of which is zero). The bijection is key to the proof because at different points we need to switch between the external view of the state $\mathsf{LocSt}\,(i,\, s_{\mathsf{own}},\, s_{\mathsf{for}})$ (where s_{own} and s_{for} are sets of pn-counter operations) and the internal one (where they are sets of product operations) without loss of information. We refer the reader to our Coq formalization for details on the proof, but do want to mention that the proof of the pn-counter wrapper ended up being more challenging than we had anticipated, in part due to technical subtleties introduced by the shift in viewpoint alluded to before.

[10] We have added one additional property to the common CRDT resource interface (an excerpt of which was shown in Figure 4.3), and thus we extended the proof of Nieto et al. [Nie+22] such that this additional property is also proved for their op-based library.

Finally, we describe the closed program we verified. This example shows client code interacting with a pn-counter. The relevant snippet is shown below: we have two replicas, A and B, each of which increments the counter, reads it, and then asserts that the read returns one of two possible values. In A the possible values are 1 and 3, depending on whether the remote operation `add 2` has been propagated from B to A by the time the read happens. In either case, the `add 1` must be visible by the subsequent read because the latter happened later according to program order: this is the so-called *reads-follow-writes* session guarantee [Gon+21]. An analogous situation happens for replica B.

```
(* replica A *)
add 1;
let r = get_value () in
assert (r = 1 || r = 3)
```

```
(* replica B *)
add 2;
let r = get_value () in
assert (r = 2 || r = 3)
```

The reader can consult Nieto et al. [Nie+22] for more details on the proof that the assertions in the example above do not fail, but the important part is that we were able to swap out the op-based counter implementation by our state-based counter, while (almost) keeping the proof intact. The two implementations are quite different: the op-based implementation relies on a causal broadcast algorithm, while ours does not and instead relies on lattice properties, as well as on applications of the product combinator. The fact that both implementations meet the same specification is good evidence that the denotation-based separation logic specifications are general, flexible, and can hide implementation details.

4.8 Related Work

As previously mentioned, the idea of specifying CRDTs as a partial function from event histories (including causality metadata) to the data type's state comes from Burckhardt et al. [Bur+14]. We also learned from their paper that state-based CRDTs can be thought of as transitively delivering (the effects of) events when states are merged, which in turn makes it possible to prove, as we have done, that state-based CRDTs are causally-consistent. [11]

Leijnse et al. [LAB19] reformulate the above specification style so it can be better applied to higher-order combinators, coining the term CRDT denotation. They work solely with specifications, and do not implement these combinators nor verify an implementation.

The idea of tracking the state of a concurrent data structure via a logical set of operations that is divided into contributions by the current thread and those originating from other threads, as in our $\mathsf{LocSt}\,(i,\ o,\ f)$ resource, has previously appeared in the context of the FCSL logic (where they are termed "self" and "other" contributions, respectively) [Nan; Del+16].

The Compass separation logic framework [Dan+22] (also Iris-based) can be used to specify and verify functional correctness of concurrent data structures in a relaxed memory model. There are a number of commonalities with our work: their specs are also given as logically-atomic triples that track the state of a data structure as a function of the set of writes that are visible by a given thread. They develop a notion of logical *views* that is similar to our local snapshots $\mathsf{LocSnap}(i, o, f)$ (without the distinction between own and foreign events): ownership of a view provides a lower bound on the set of observed events, and the views contain

[11] They do mention that state-based CRDTs are causally-consistent, but there is no formal proof, or even a precise lemma statement.

logical metadata that tracks the happens-before relation between writes. The main difference with our work is that we operate at the intersection of weak consistency and message passing, whereas their work is in the context of shared memory.

Zeller et al. [ZBP14] implement and formally verify multiple state-based CRDTs in Isabelle/HOL. To our knowledge, they are the first to explicitly link denotation-style specifications to their lattice-based implementations. Like us, they prove both convergence and functional correctness. There are two main differences with our work: their system model is an STS where the states map each replica id to the replica state (this is very similar to our STS model from Section 4.6.1). By contrast, in our work the system model is the operational semantics of AnerisLang, with support for mutation, node-local concurrency, higher-order functions, etc. This means that while their technique can only be used to reason about a CRDT in isolation, ours can verify a system where the CRDT and a client (or a larger distributed system of which the CRDT is a small part) are executing together, and where both of these are implemented in a realistic programming language. The second difference is that in Zeller et al.'s work one needs to come up with a different invariant that implies coherence between the denotation and the lattice for each implemented CRDT, as well as proving for each example that the invariant is preserved by the different transitions. In our work, we prove just one invariant, global state validity, and the CRDT implementer then needs to prove coherence given that global validity holds.

Nair et al. [NPS20] verify several state-based CRDTs. These are not "pure" CRDTs in the sense that some of data structure's operations are at times disabled. For example, they show how to verify a distributed lock implemented as a CRDT (the release-the-lock operation can only be enabled if the local replica owns the lock). Proof wise, this means that sometimes it is useful to enforce example-specific invariants on the CRDT being verified: it would be interesting to modify StateLib so it can support these user-defined invariants and so can tackle the examples presented in their paper.

Gondelman et al. [Gon+21] verify functional correctness of a causally-consistent key-value store using Aneris. Their work introduces the encoding of causality in separation logic that we use. However, their key-value store is not a CRDT because it violates convergence. Additionally, their work is closer to op-based CRDTs because writes are propagated individually.

Timany et al. [Tim+21] develop an extension of Aneris called Trillium where one simultaneously proves both safety and also that the program being verified refines an STS model. Unlike in our work, where the STS is used just to prove safety, their refinement is *history-preserving*, which allows them to prove liveness properties as well. As one case study, they show that a state-based G-Counter CRDT is eventually consistent. They do not tackle any other (more complex) CRDTs, and their specification of the G-Counter relies on the fact that the G-Counter is monotonic. It would be interesting to recast our work using the Trillium methodology with the goal of showing that any CRDT implemented via StateLib is eventually-consistency (that is, additionally showing eventual delivery).

The closest related work is Nieto et al. [Nie+22], from which we inherit the CRDT resource interface from Figure 4.3, the modular structure of the implementation (a core library that can be instantiated for different CRDT examples), as well as the general structure of the proof: a state-transition system model that is embedded in the logic, as well as a lock invariant that ties the denotation-style specification to the lattice-based state. The main difference is that their paper deals exclusively with op-based CRDTs: as mentioned in Section 4.3.2 adapting their technique to the state-based setting leads to a number of technical challenges we had to solve in our approach.

4.9 Conclusions

We have shown how to give modular specifications to realistic state-based CRDT implementations using the Aneris separation logic. Our specifications show both convergence and functional correctness relative to an abstract denotational model of the CRDT.

We have explored our approach by implementing and verifying a library, StateLib, for building state-based CRDTs. Our library takes as input a purely-functional implementation of a state-based CRDT's core logic, together with coherence proofs between the CRDT's lattice-based and denotation-based models. The library then produces as output a fully-fledged CRDT that is replicated over multiple nodes. Using the library we implemented and verified multiple example CRDTs from the literature.

When taken together with Nieto et al. [Nie+22], our work presents a unified framework for the specification and verification of op- and state-based CRDTs. To show that we can abstract away from the fact that a CRDT is state-based, we re-prove Nieto et al.'s interface for resources tracking CRDT state using a new definition of resources that is compatible with state-based CRDTs. We test this approach by showing that one can start with a client program that uses an op-based counter CRDT, swap out the counter's implementation by our state-based implementation, and recover the safety proof for the entire closed program while making minimal changes to the original proof.

5 *Towards Session Guarantees for Client-Centric Consistency*

This chapter is an extended, self-contained, version of Section 5 of the POPL'21 paper "Distributed Causal Memory: Modular Specification and Verification in Higher-Order Distributed Separation Logic" [Gon+21], which I co-authored. Parts of the chapter's introduction, the entirety of Section 5.1, and some figures are copied and edited from the above paper, as well as the accompanying technical appendix [Gon+20]. The rest is original material.

Abstract

Client-centric consistency models describe what assumptions a client program can make when interacting with a replicated data type. Specifically, the four *session guarantees*: *monotonic reads*, *read your writes*, *monotonic writes*, and *writes follow reads* specify how client operations relate to prior local observations, without regard for concurrent updates performed by other clients. We show how these session guarantees follow from the encoding of causality in separation logic developed by Gondelman et al. [Gon+21]. We do this by implementing and verifying a *session manager* library on top of Gondelman et al. [Gon+21]'s verified causally-consistent database. The session manager allows interaction with the database in a client-server setting. Even though the session-manager's specifications are weaker than those of the key-value store and rely exclusively on persistent (non-ephemeral) resources, they are strong enough to imply the four session guarantees. To our knowledge, our work is the first formally-verified treatment of session guarantees. We have formalized all the presented results in the Coq proof assistant.

The distributed systems literature distinguishes between two kinds of consistency models: *data centric* and *client centric* [TS07, Chapter 7].

Data centric consistency models constrain the order of operations across multiple replicas and account for concurrent updates. Causal consistency [Aha+95] is a data centric consistency model because it restricts the order of operations to respect causal order. In Gondelman et al. [Gon+21] my co-authors and I presented the first specification and verification of an implementation of a causally-consistent distributed database that supports modular verification of full functional correctness of clients and servers. This chapter is an extended version of a case study included in that paper. The case study investigates the use of the verified distributed database from the perspective of client-centric consistency.

Client-centric consistency models constrain how operations by a *single client* relate to prior operations performed by the same client, *without regard for concurrent updates.* [1] There are four

[1]Clients are assumed to execute operations sequentially, without intra-node concurrency.

client-centric consistency models, usually called *session guarantees* [Ter+94]; they are *read your writes* (RYW), *monotonic reads* (MR), *writes follow reads* (WFR), and *monotonic writes* (MW). Each of these guarantees is a principle for reasoning about operations that happen within a *session*, a sequence of writes performed by a client against *one or more* database replicas.

The session guarantees are attractive as reasoning principles because they offer an intuitive perspective for client application programmers. For example, consider the following scenario, lifted from Tanenbaum and Steen [TS07, Chapter 7]. A mail application is geo-replicated and weakly-consistent. A client connects to replica A and retrieves a list of unread emails. Within the same session, the client then connects to a different replica B and again retrieves a list of unread emails. Because of weak consistency, it is possible that a specific unread email is present in replica A but not replica B. This could manifest as an email "disappearing" from the list of unread emails displayed by the mail client, which might be unintuitive to the user. The monotonic reads guarantee rules out this scenario by ensuring that

> If a process reads the value of a data item x, any successive read operation on x by that process will always return that same value or a more recent value [TS07].

In our example, if replica B is missing some of the previously-retrieved emails then either the client is re-directed to a more up-to-date replica, or replica B "catches up" before replying to the client's request.

This chapter develops a separation-logic based account of session guarantees for our causally-consistent database. We show that our definition of causality is strong enough to imply all four session guarantees. Specifically, we implement a *session manager* library that allows clients to interact with the distributed database in a client-server setting (through message passing). We give specifications for the session manager that use only *persistent* (duplicable) resources, simplifying client reasoning. Even though these specifications are weaker than the original database read and write specifications from Gondelman et al. [Gon+21], they are expressive enough to imply the session guarantees. We then prove the session guarantees as four client programs that interact with the database through the session manager.

The minor hedging in the section's title ("*Towards* Session Guarantees") comes from a simplifying assumption we make in our development. We assume that a client connects to the *same* replica throughout a session. These *tethered* sessions are in fact the common case in a client-server interaction, because switching between replicas is inefficient (but sometimes necessary because of e.g. server failure). We explicitly leave the verification of a more-realistic session manager that can multiplex over different replicas as future work.

5.1 Background: a Causally-Consistent Distributed Database

We now review the specification of the causally-consistent distributed database in Gondelman et al. [Gon+21].

Two simple, illustrative examples of causal dependency are depicted in Figure 5.1; programs executed on different nodes are separated using double vertical bars. Notice that in our setting all database keys are uninitialized at the beginning and the read operation returns an optional value indicating whether or not the key is initialized. In both examples, the read(x) command returns the value 37, as indicated by the comment in the code, as the preceding wait command waits for the effects of write$(y, 1)$ to be propagated. In the topmost example (illustrating direct causal dependence) the write$(x, 37)$ command immediately precedes the

```
write(x, 37) || wait(y = 1)
write(y, 1)  || read(x) // reads Some 37
```

```
write(x, 0)  || wait(x = 37) || wait(y = 1)
write(x, 37) || write(y, 1)  || read(x) // reads Some 37
```

Figure 5.1: Two examples of causal dependency: direct (left) and indirect (right, [Llo+11]). Copied from Gondelman et al. [Gon+21].

write(y, 1) command on the same node; hence any node that observes 1 for key y should also observe 37 for key x. However, in the bottommost example, the write(y, 1) command is executed on the middle node *only after* the value of 37 is observed for key x on that node; hence, in this example too, any node that observes 1 for key y should also observe 37 for key x.

5.1.1 Overview

Gondelman et al. [Gon+21]'s specifications are based on a mathematical model tracking the abstract state of the local key-value store at every replica, *i.e.*, the history of updates. The specification represents this model using Iris's ghost theory to track auxiliary state (state that is *not* physically present at runtime and only tracked logically for verification purposes). The history of updates consists of the following information:

1. For each replica, we track a *local history* of all memory updates that the replica has observed since its initialization. It includes both local write operations (which are observed immediately) and updates due to synchronization with other replicas.

2. We also track an *abstract global memory* that, for each key, keeps track of all write events to that key (by any replica in the system).

We refer to the elements of local histories as *apply events* and to the elements of the abstract global memory as *write events*. Both apply and write events carry all the necessary information about the original update, including the logical time of the corresponding apply or write operation. We implement logical time using vector clocks, but all that matters is that it is a partial order. The ordering is defined such that it reflects causal order: if the time of event e is strictly less than the time of e', then e' *causally depends* on e, and if the time of e and e' are incomparable, then e and e' are *causally independent*. This allows us to formulate the causal consistency of the distributed database as follows:

If a node observes an apply event a, it must have already observed all write events of the abstract global memory that happened before (according to logical time) the write event corresponding to a. (Causal Consistency)

The specifications we give to the read and write operations reflect the effects of these

EVENTS

$$
\begin{array}{lll}
(\mathrm{K}, v, t, o) & \in & \mathit{WriteEvent} \triangleq \mathit{Keys} \times \mathsf{Value} \times \mathsf{Time} \times \mathbb{N} \\
(\mathrm{K}, v, t, o, m) & \in & \mathit{ApplyEvent} \triangleq \mathit{Keys} \times \mathsf{Value} \times \mathsf{Time} \times \mathbb{N} \times \mathbb{N} \\
\mathit{Maximals(X)} & \triangleq & \{x \mid x \in X \land \forall y \in X.\, \neg(x.t < y.t)\} \\
\mathit{Maximum(X)} & \triangleq & \begin{cases} \mathsf{Some}\ x & \mathit{if}\ x \in X \land \forall y \in X.\, x \neq y \implies y.t < x.t \\ \mathsf{None} & \mathit{otherwise} \end{cases} \\
\mathit{Observe} & : & \wp^{\mathsf{fin}}(\mathit{ApplyEvent}) \xrightarrow{\mathsf{fin}} \mathit{ApplyEvent} \\
\lfloor \cdot \rfloor & : & \mathit{ApplyEvent} \xrightarrow{\mathsf{fin}} \mathit{WriteEvent}
\end{array}
$$

MEMORY

$$
\begin{array}{lll}
s_i & \in & \mathit{LocalHistory} \triangleq \wp^{\mathsf{fin}}(\mathit{ApplyEvent}) \\
M & \in & \mathit{GlobalMemory} \triangleq \mathit{Keys} \xrightarrow{\mathsf{fin}} \wp^{\mathsf{fin}}(\mathit{WriteEvent})
\end{array}
$$

Figure 5.2: Mathematical model of distributed causal memory with abstract notion of validity. Edited from from Gondelman et al. [Gon+21].

operations on the tracked histories. The intuitive reading of the specifications is as follows:

- *Either,* read(K) *returns nothing, in which case we know that the local history contains no observed events for key K.*
- *Or,* read(K) *returns some value v, in which case we know that there is an apply event a in the local history with value v; a has a corresponding write event in the global memory; and a is a maximal element (w.r.t. time and hence causality) in the local history.* (Read Spec)

After the write operation write(K, v) *there is a new write event w added to the global memory and a new apply event a corresponding to it in the local history, and a is the* maximum *element (w.r.t. time and hence causality) of the local history, i.e., the event a causally depends on all other events in the local history.* (Write Spec)

5.1.2 Mathematical Model

In this section we formalize the mathematical model of causality. Figure 5.2 shows the model definitions and properties needed to reason about clients.

In the model, a write event is represented as a four-tuple (K, v, t, o) consisting of a key, a value, the time, and the index of the replica on which the write event happened. In the concrete implementation time is represented using vector clocks, but to reason about client code, all we need is an abstract notion of time, and therefore our model uses a notion of logical time represented by a partial order $\leq$ (we write $<$ for the strict version of it). We can decide whether two write events w_1 and w_2 are causally related by comparing their times: if $w_1.t < w_2.t$, then

Table 5.1: Propositions to track the state of the key-value store. Copied from Gondelman et al. [Gon+21].

Proposition	**Intuitive meaning**
$\mathsf{Seen}(i, s)$	The set s is *a causally closed subset of* the local history of replica i
$\mathsf{Snap}(\mathrm{K}, h)$	The set h is *a subset of* the global memory for key K
$\mathrm{K} \rightharpoonup_u h$	The global memory for key K is *exactly* h

We say s is a *causally closed* subset of s' if: $s \subseteq s' \wedge \forall a_1, a_2 \in s'.\ a_1.t < a_2.t \wedge a_2 \in s \Rightarrow a_1 \in s$.

w_1 must have *happened before* w_2, and w_2 is *causally dependent* on w_1. When $w_1.t$ and $w_2.t$ are incomparable, then the events w_1 and w_2 are *causally independent* or *concurrent.*

To account for how write events are applied locally on each replica we use the notion of an *apply event.* Thus an apply event only makes sense in the context of a particular replica. Formally, given a replica i, an *apply event* is represented by a five-tuple $a = (\mathrm{K}, v, t, o, m)$, where m is the number of write events applied on replica i. We refer to m as the *sequence identifier* of a. When $i = o$, the apply event corresponds to a write operation invoked on the replica itself, whereas if $i \neq o$, then the apply event corresponds to a write event received from replica i. Given an apply event $a = (\mathrm{K}, v, t, o, m)$, we denote by $\lfloor a \rfloor$ the write event (K, v, t, o), which we refer to as the *erasure of* a.

The *local history* of replica i, written s_i, is the set of all apply events observed by the replica since its initialization. The *abstract global memory,* written M, is a finite map from keys to finite sets of write events. We model the local key-value store for a replica i simply as a finite map from keys to values.

Given a set X of write or apply events, $\mathrm{Maximals}(X)$ (resp. $\mathrm{Maximum}(X)$) denotes the set of maximal events (resp. the maximum event) w.r.t. the time ordering. Note that, for any events $e, e' \in \mathrm{Maximals}(X)$, the time of e and e' are incomparable and hence e and e' are causally unrelated. Given a non-empty set of apply events A, the event $\mathsf{Observe}(A)$ is the maximum element of A w.r.t. the ordering of sequence identifiers. (If A is empty, we let $\mathsf{Observe}(A)$ be some default apply event).

5.1.3 Specification

Iris Predicates to Represent the State of the Key-Value Store. Recall the intuitive specifications that we gave for the read and write operations on our distributed database in Section 5.1.1. These specs only assert that certain write/apply events are added to the global memory/local history. Hence, it suffices to have a persistent proposition in the logic that asserts the partial information that certain events are indeed part of the local history or global memory. For this purpose, we introduce the persistent abstract predicates Seen and Snap which intuitively assert knowledge of a subset of the local history, and global memory, respectively. These abstract predicates and their intuitive meaning are presented in Table 5.1. Notice that the Seen predicates assert knowledge of a subset of the local history that is *causally closed* as defined in the figure. In addition to the partial knowledge about the global memory represented using the Snap predicate, it is also useful to track the precise contents of the global memory for each key. We do this using the ephemeral abstract proposition $\mathrm{K} \rightharpoonup_u h$ which, intuitively, asserts that the set of *all* write events for the key K is h. We can track the precise contents of the global memory because all write events in the global memory can only originate from a write

Properties of global memory, *i.e.*, Snap and $\rightharpoonup_u$ predicates:

$$\mathsf{Snap}(\mathrm{K}, h) * \mathsf{Snap}(\mathrm{K}, h') \vdash \mathsf{Snap}(\mathrm{K}, h \cup h') \qquad \text{(Snap union)}$$

$$\mathrm{K} \rightharpoonup_u h \vdash \mathrm{K} \rightharpoonup_u h * \mathsf{Snap}(\mathrm{K}, h) \qquad \text{(Take Snap)}$$

$$\boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}} * \mathrm{K} \rightharpoonup_u h * \mathsf{Snap}(\mathrm{K}, h') \vdash \Rrightarrow_{\mathcal{E}} \mathrm{K} \rightharpoonup_u h * h' \subseteq h \qquad \text{(Snap inclusion)}$$

$$\boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}} * \mathsf{Snap}(\mathrm{K}, h) * \mathsf{Snap}(\mathrm{K}, h') \vdash \Rrightarrow_{\mathcal{E}} \forall w \in h, w' \in h'.\ w.t = w'.t \Rightarrow w = w' \qquad \text{(Snap extensionality)}$$

Properties of local histories, *i.e.*, the Seen predicate:

$$\boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}} * \mathsf{Seen}(i, s) * \mathsf{Seen}(i, s') \vdash \Rrightarrow_{\mathcal{E}} \mathsf{Seen}(i, s \cup s') \qquad \text{(Seen union)}$$

$$\boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}} * \mathsf{Seen}(i, s) * \mathsf{Seen}(i', s') \vdash \Rrightarrow_{\mathcal{E}} \forall a \in s, a' \in s'.\ a.t = a'.t \Rightarrow a.k = a'.k \land a.v = a'.v \qquad \text{(Seen global extensionality)}$$

$$\boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}} * \mathsf{Seen}(i, s) * \mathsf{Seen}(i, s') \vdash \Rrightarrow_{\mathcal{E}} \forall a \in s, a' \in s'.\ a.t = a'.t \Rightarrow a = a' \qquad \text{(Seen local extensionality)}$$

$$\boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}} * \mathsf{Seen}(i, s) * a \in s \vdash \Rrightarrow_{\mathcal{E}} \exists h.\ \mathsf{Snap}(a.k, h) * \lfloor a \rfloor \in h \qquad \text{(Seen provenance)}$$

Causality in terms of resources and predicates:

$$\boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}} * \mathsf{Seen}(i, s) * \mathsf{Snap}(\mathrm{K}, h) \vdash \Rrightarrow_{\mathcal{E}} \forall a \in s, w \in h.\ w.t < a.t \Rightarrow \exists a' \in s_{|\mathrm{K}}.\ \lfloor a' \rfloor = w \qquad \text{(Causality)}$$

The set $s_{|\mathrm{K}}$ is the set of apply events in s with key K: $s_{|\mathrm{K}} \triangleq \{a \in s \mid a.k = \mathrm{K}\}$.

Figure 5.3: Laws governing database resources. The mask $\mathcal{E}$ is any arbitrary mask that includes $\mathcal{N}_{\mathsf{GI}}$. Copied from Gondelman et al. [Gon+21].

operation on the distributed database. On the other hand, we cannot have precise knowledge about local histories because at any point in time, due to concurrency, a replica may observe new events.

In addition to the abstract predicates just discussed, the client will also get access to a global invariant $\boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}}$ which, intuitively, asserts that there is a valid global state, and that the predicates Seen, Snap, and $\rightharpoonup_u$ track this global state.[2] Clients need not know the definition of this invariant and can just treat it as an abstract predicate.

5.1.4 Laws Governing Database Resources

The laws governing the predicates Seen, Snap, and $\rightharpoonup_u$, are presented in Figure 5.3. Notice that most of these laws only hold under the assumption that the global invariant holds. This can also be seen in the fact that they are expressed in terms of an update modality with a mask

[2] $\mathcal{N}_{GI}$ is a fixed name of the global invariant.

that enables access to the global invariant. All of these laws make intuitive sense based on the intuitive understanding of the predicates Seen, Snap, and $\rightharpoonup_u$. For instance, the law (Snap union) asserts that if we know that the sets h and h' are both subsets of the global memory for a key K, then so must the set $h \cup h'$. The extensionality laws essentially state that events are uniquely identifiable with their time: if two events have the same time, then they are the same event. Note that the law (Seen union), as opposed to the law (Snap union), requires access to the invariant. This is because, we need to establish causal closure (see Figure 5.1) for $s \cup s'$ in the conclusion of the law with respect to the local history tracked in the global invariant. The most important law in Figure 5.3 is the law (Causality). This law allows us to reason about causality: if a replica i has observed an event a that has a time greater than a write event w, i.e. a causally depends on w, then replica i must have also observed w (it must have a corresponding apply event a'). Notice that the *causal closure* property of local histories s for which we have $\mathsf{Seen}(i, s)$ in Figure 5.1 is crucial for the (Causality) law to hold.

5.1.5 Specs for the Read and Write Operations

Figure 5.4 shows the specification for reading from and writing to the distributed database locally on a replica i.

Read Specification. The post condition of the read operation asserts that the client gets back a set of apply events s', $\mathsf{Seen}(i, s')$, observed by replica i performing the read operation such that $s' \supseteq s$. The reason for the $s' \supseteq s$ relation is that during the time since performing the last operation by replica i, *i.e.*, when we had observed the set s, some write events from other replicas may have been applied locally.

When read(K) is executed on a replica i, it either returns None or Some v for a value v. If it returns None, then the local memory does not contain any values for key K. Hence the local history s' restricted to key K, $s'_{|K}$ should be empty *cf.* the definition of $s'_{|K}$ in Figure 5.3.

Otherwise, if it returns Some v, then the local memory contains the value v for key K. This can happen only if the local memory of the replica at the key K has been updated, and the latest update for that key has written the value v. Consequently, the local history s' must have recorded this update as *the latest apply event* a for the key K, *i.e.*, $\mathsf{Observe}(s'_{|K}) = a$. Hence $a \in \mathrm{Maximals}(s'_{|K})$. The reader may wonder why a is not the maximum element, but only in the set of maximal elements. To see why, suppose that just before the read operation was executed, two external causally-unrelated writes have been applied locally on replica i, so that the local history recorded them as two distinct apply events whose times are incomparable. Naturally, one of two writes must have been applied before the other and the latest observed apply event *must* correspond to the subsequent second write event. However, as the apply operation is hidden from the client, there is no way for the client to observe which of the two writes is the latest. Consequently, all the client can know is that the latest observed event a is *one of the most recent* local updates for key K, *i.e.*, among the maximal elements. Naturally, the write event $\lfloor a \rfloor$ should be in the abstract global memory. This is expressed logically by the proposition $\mathsf{Snap}(K, \{\lfloor a \rfloor\})$.

Write Specification. The postcondition of the write specification ensures that after the execution of write(K, v), the client gets back the resources $K \rightharpoonup_u h \uplus \{\lfloor a \rfloor\}$ and $\mathsf{Seen}(i, s' \uplus \{a\})$, where a and $\lfloor a \rfloor$ are respectively the apply and write events that model the effect of the write

$$\textsc{ReadSpec}$$
$$\{\mathsf{Seen}(i, s)\}$$
$$\langle ip_i;\ \mathsf{read}(\mathrm{K})\rangle$$
$$\left\{\begin{array}{l} v.\ \exists s' \supseteq s.\, \mathsf{Seen}(i, s') \ast \\ \quad \Big((v = \mathsf{None} \land {s'}_{|\mathrm{K}} = \emptyset) \lor \\ \quad (\exists a \in {s'}_{|\mathrm{K}}.\ v = \mathsf{Some}\ a.v \ast \mathsf{Snap}(\mathrm{K}, \{\lfloor a \rfloor\}) \ast a \in \mathrm{Maximals}({s'}_{|\mathrm{K}}) \ast \mathsf{Observe}({s'}_{|\mathrm{K}}) = a)\Big) \end{array}\right\}$$

$$\textsc{WriteSpec}$$
$$\{\mathsf{Seen}(i, s) \ast \mathrm{K} \rightharpoonup_u h\}$$
$$\langle ip_i;\ \mathsf{write}(\mathrm{K}, v)\rangle$$
$$\left\{\begin{array}{l} ().\ \exists s' \supseteq s.\, \exists a.\, \mathrm{K} = a.k \ast v = a.v \ast \mathsf{Seen}(i, s' \uplus \{a\}) \ast \mathrm{K} \rightharpoonup_u h \uplus \{\lfloor a \rfloor\} \ast \\ \quad a = \mathrm{Maximum}(s' \uplus a) \ast \lfloor a \rfloor \in \mathrm{Maximals}(h \uplus \{\lfloor a \rfloor\}) \end{array}\right\}$$

Figure 5.4: Read and write specifications. Copied from Gondelman et al. [Gon+21].

operation. The mathematical operation $\uplus$ is the disjoint union operation on sets; $A \uplus B$ is undefined if $A \cap B \neq \emptyset$.

As for read, the new set of apply events s' can be a superset of s. Contrary to read, the postcondition for write states that $a = \mathrm{Maximum}(s' \uplus a)$, *i.e.*, that a is actually *the most recent* apply event. This matches the intuition that the update $\mathsf{write}(\mathrm{K}, v)$ causally depends on any other apply event previously observed at this replica.

While a is the maximum apply event locally, its erasure $\lfloor a \rfloor$ is only guaranteed to be among the maximal write events, *i.e.*, $\lfloor a \rfloor \in \mathrm{Maximals}(h \uplus \{\lfloor a \rfloor\})$. Intuitively, this is because there can be other write events in h, performed by other replicas, that we have not yet locally observed. As those events are not observed on our replica, the newly added write event $\lfloor a \rfloor$ does not causally depend on them and hence does not have a strictly greater time—in practice those write events have times that are incomparable to that of $\lfloor a \rfloor$ as neither depend on the other.

5.2 *Session Guarantees and Causal Consistency*

Having reviewed the treatment of causal consistency in Gondelman et al. [Gon+21], we now move on to present the session guarantees as originally defined by Terry et al. [Ter+94]. We will use the following (informal) auxiliary definitions:

$\mathsf{DB}(R, t)$	The set of writes received by replica R at or before time t.
$\mathsf{WriteOrder}(\mathsf{w_1}, \mathsf{w_2})$	A predicate that is true if write w_1 should be ordered before w_2.
$\mathsf{RelevantWrites}(S, t, r)$	The smallest set of writes that is *complete* for read operation r and $\mathsf{DB}(S, t)$. A set of writes W is complete with respect to a read r and replica state $\mathsf{DB}(S, t)$ if for any subset D of $\mathsf{DB}(S, t)$ that also contains W (i.e. $W \subseteq D \subseteq \mathsf{DB}(S, t)$), the result of executing r against W is the same as the result of executing r against D.

The four guarantees, copied from Terry et al. [Ter+94], are:

- *Read Your Writes:* If read r follows write w in a session and r is performed at replica S at time t then w is included in $\mathsf{DB}(S, t)$.
- *Monotonic Reads:* If read r_1 occurs before r_2 in a session, and r_1 accesses server S_1 at time t_1 and r_2 accesses server S_2 at time t_2, then $\mathsf{RelevantWrites}(S_1, t_1, r_1) \subseteq \mathsf{DB}(S_2, t_2)$.
- *Writes Follow Reads:* If read r_1 precedes write w_2 in a session and r_1 is performed at server S_1 at time t_1, then for any server S_2, if w_2 is in $\mathsf{DB}(S_2, t_2)$ then for any $w_1 \in \mathsf{RelevantWrites}(S_1, t_1, r_1)$ we have $w_1 \in \mathsf{DB}(S_2, t_2)$ and $\mathsf{WriteOrder}(\mathsf{w_1}, \mathsf{w_2})$.
- *Monotonic Writes:* If write w_1 precedes write w_2 in a session, then for any server S_2, if $w_2 \in \mathsf{DB}(S_2, t)$ then $w_1 \in \mathsf{DB}(S_2, t)$ and $\mathsf{WriteOrder}(\mathsf{w_1}, \mathsf{w_2})$.

Or, more informally:

- RYW: reads observe writes not older than preceding writes.
- MR: reads observe writes not older than writes observed by preceding reads.
- WFR: writes and writes observed through preceding reads propagate to all replicas in program order.
- MW: writes propagate to all replicas in program order.

Notice that RYW and MR talk about what writes are observed within a *single* replica, while WFR and MW talk about the order of writes in *all* replicas.

Connection to Causal Consistency Because events that happen in the same session are causally-ordered (in a total way), and session guarantees talk about enforcing a propagation order that respects session order, it is not surprising that session guarantees have some connection to causal consistency. In fact, Brzezinski et al. [BSW04] formalize the trace semantics of both causal consistency and session guarantees and prove that causal consistency implies all four session guarantees.

This motivates our work in this section. We would also like to obtain a version of the session guarantees from our embedding of causality in separation logic, both because session guarantees are useful as reasoning principles, and also because it would be evidence that our definition of causality is in the right track.

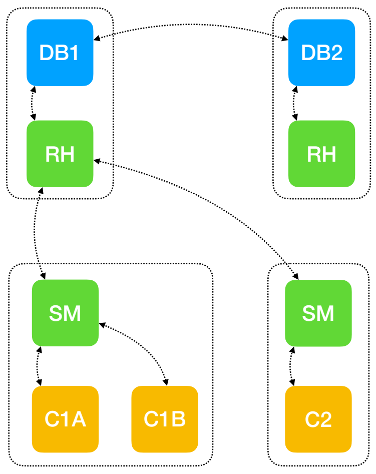


Figure 5.5: Clients using the distributed database via the session manager library. Copied from Gondelman et al. [Gon+21].

5.3 *Session Manager Library*

In the examples in Gondelman et al. [Gon+21, Section 4.5] each client program is co-located on the *same* node as the database replica that it reads from and writes to. By contrast, in a *client-server* architecture, a client might interact with *multiple* replicas (servers), and clients and replicas are located on *different* nodes.

Because the session guarantees assume a client-server setting, we implement a *session manager* library that allows clients located in a different node to read and write from the distributed database. The library consists of two components: a client *stub* that proxies requests to the server, and a *request handler* that handles the requests server-side. Figure 5.5 illustrates how clients (C1A, C1B, and C2) running on different nodes communicate with multiple database replicas (DB1 and DB2) via the session manager stub (SM) and request handler (RH).

5.3.1 *Session Manager Stub*

Figure 5.6 shows the code of the client stub. The client stub is responsible for proxying client requests (mostly reads or writes) to a remote replica, so it implements a simple form of remote procedure call.

The types `sm_req` and `sm_res` define the set of requests a client issues and the server's responses, respectively. An `InitReq` initializes the session. A `ReadReq k` reads the current value of `k` at the contacted replica. A `WriteReq (k, v)` writes the value `v` under key `k`.

The helper function `listen_wait_for_seqid` takes a socket and a *sequence id*, a natural number that uniquely identifies each request. The function blocks until a message with the given sequence id is available at the socket. The sequence id is then incremented and the function returns the received message.

The `session_exec` function executes a request `req` at the database replica with address `server_addr`. Executing a request consists of serializing the request object, sending a message to the server, and then waiting for a response with the right sequence id number using `listen_wait_for_seqid`. Because we want requests within a session to be totally-ordered, execution of a request is serialized with a lock.

```
type db_key
type db_val

type sm_req =
  InitReq
| ReadReq of db_key
| WriteReq of db_key * db_val

type sm_res =
  InitRes
| ReadRes of db_val
| WriteRes

let rec listen_wait_for_seqid skt seq_id =
  let res_raw = listen_wait skt in
  let res = deser_res (fst res_raw) in
  let tag = fst res in
  let vl = snd res in
  if (tag = !seq_id) then
    (seq_id := !seq_id + 1;
    vl)
  else
    listen_wait_for_seqid skt seq_id

let session_exec skt seq_id lock server_addr req =
  acquire lock;
  let msg = ser_req req in
  sendTo skt msg server_addr;
  let res = listen_wait_for_seqid skt seq_id in
  release lock;
  res

let sm_setup client_addr =
  let skt = socket () in
  socketbind skt client_addr;
  let seq_id = ref 0 in
  let lock = newlock () in
  let connect_fn server_addr =
    session_exec skt seq_id lock server_addr InitReq;
    ()
  in
  let read_fn server_addr key =
    session_exec skt seq_id lock server_addr (ReadReq key)
  in
  let write_fn server_addr key vl =
    session_exec skt seq_id lock server_addr (WriteReq (key, vl));
    ()
  in
  (connect_fn, read_fn, write_fn)
```

Figure 5.6: Session manager stub. Runs client-side. Copied from Gondelman et al. [Gon+20].

```
let rec request_handler skt rd_fn wr_fn =
  let req_raw = listen_wait skt in
  let sender = snd req_raw in
  let req = deser_req (fst req_raw) in
  let seq_id = fst req in
  let res =
    match (snd req) with
    | Some InitReq -> InitRes
    | Some ReadReq k -> ReadRes (rd_fn k)
    | Some WriteReq (k, v) -> wr_fn k v; WriteRes
    | None -> assert false
  in
  sendTo skt (ser_res (seq_id, res)) sender;
  request_handler skt rd_fn wr_fn

let server dbs db_id req_addr =
  let fns = init dbs db_id in
  let rd_fn = fst fns in
  let wr_fn = snd fns in
  let skt = socket () in
  socketbind skt req_addr;
  request_handler skt rd_fn wr_fn
```

Figure 5.7: Request handler. Runs server-side. Copied from Gondelman et al. [Gon+21].

Finally, the function `sm_setup` is the entry point the client calls to initialize the session manager stub. The function takes a local socket address from which to connect to a server. It then sets up some internal state, including the sequence id counter which is initialized to 0. `sm_setup` returns a 3-tuple with three closures: `connect_fn`, `read_fn`, and `write_fn` for initializing a session, reading a key, and writing to a key. All these closures take as argument a `server_addr` parameter with the address of the server to contact. The client is responsible for selecting the right server: in the examples we show later the client always contacts the same replica.

5.3.2 Request Handler

Figure 5.7 shows the code of the request handler, which runs server-side.

The function `request_handler` takes a socket on which to listen for requests and read and write functions to access the database. It then loops waiting for a request. When one is received, it is deconstructed and pattern matched against. For `InitReq`, the handler just acknowledges receipt of the message. For `ReadReq` and `WriteReq`, the handler calls the read or write functions on the database, and then sends a response possibly containing the return value (for reads). The response is then sent to the client.

The `server` function initializes the database by calling the database's `init` function [Gon+21]. It then obtains read and write functions to access the database. After setting up a socket to listen on, the function calls `request_handler`.

SM-CONNECT

$$\{sa_i \Mapsto \Phi_{\mathsf{DB}_i}\}$$
$$\langle ip_{client};\ \mathsf{sconnect}(ip_i)\rangle$$
$$\left\{\exists s.\mathsf{Seen}(i,s) * \mathop{\Large\ast}_{k\in\mathsf{Keys}} \exists h_k.\mathsf{Snap}(k,h_k) * \boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}}\right\}$$

SM-READ

$$\{sa_i \Mapsto \Phi_{\mathsf{DB}_i} * \mathsf{Seen}(i,s) * k \in Keys * \mathsf{Snap}(k,h)\}$$
$$\langle ip_{client};\ \mathsf{sread}(ip_i,k)\rangle$$
$$\left\{\begin{array}{l} v.\exists s' \supseteq s, h' \supseteq h.\ * \mathsf{Seen}(i,s') * \mathsf{Snap}(k,h') * \\ \quad \big((v = \mathsf{None} * s'_{|k} = \emptyset) \\ \quad \lor (\exists a,w.\ v = \mathsf{Some}\ w * a.v = w * a.k = k * a \in \mathrm{Maximals}(s'_{|k}) * \lfloor a \rfloor \in h')\big)\end{array}\right\}$$

SM-WRITE

$$\{sa_i \Mapsto \Phi_{\mathsf{DB}_i} * \mathsf{Seen}(i,s) * k \in Keys * \mathsf{Snap}(k,h)\}$$
$$\langle ip_{client};\ \mathsf{swrite}(ip_i,k,v)\rangle$$
$$\left\{\begin{array}{l} \exists a, s' \subseteq s, h' \subseteq h.\ a.k = k * a.v = v * \mathsf{Seen}(i,s') * \mathsf{Snap}(k,h') \\ * a \notin s * a' \in s' * \lfloor a \rfloor \notin h * \lfloor a' \rfloor \in h' * \lfloor a \rfloor \in \mathrm{Maximals}(h') * \mathrm{Maximum}(s') = \mathsf{Some}\ a\end{array}\right\}$$

Figure 5.8: Session manager specifications. Copied from Gondelman et al. [Gon+21].

In summary, the session manager library provides a thin wrapper over the database so that the latter can be used in a client-server setting.

5.4 *Session Manager Specifications*

The public session manager API consists of three functions: `sconnect`, `sread`, and `swrite`. [3] Within a session, the client is supposed to call `connect` once and then follow up with a series of reads and writes. The intuition for the specifications of these three functions is a combination of two observations: (a) we would like to use only the persistent resources $\mathsf{Seen}(i, s)$ and $\mathsf{Snap}(k, h_k,)$ and (b) in every interaction with the database (whether a read or a write) we "update" our understanding of the database's current state. Within the logic, this update happens by providing our "old" copy of $\mathsf{Seen}(i, s)$ and $\mathsf{Snap}(k, h_k)$ and obtaining back fresher copies $\mathsf{Seen}(i, s')$ and $\mathsf{Snap}(k, h'_k)$, where fresher means $s \subseteq s'$ and $h_k \subseteq h'_k$. In other words, local and global snapshots are monotonically updated.

With this in mind, Figure 5.8 shows specifications for the session manager functions.

The specification for `sconnect` has one precondition: the proposition $sa_i \Mapsto \Phi_{\mathsf{DB}_i}$, meaning that the socket address sa_i we are connecting to corresponds to replica running the ith request handler (where request handler order is just the DB replica order). We will later explain the socket protocol Φ_{DB_i}. There is no physical return value, but there is a (purely logical) postcondition granting knowledge of various facts. We learn that the database's internal state

[3]We use `sread` instead of `read` to differentiate between the session manager API and the database API.

is at least $\mathsf{Seen}(i, s)$ for some s. Additionally, we learn that for every key k there is a history of updates that includes at least h_k. We also learn of the existence of the (opaque to the client) global invariant $\boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}}$. Knowledge of the $\boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}}$ is necessary to use the resource lemmas (e.g. the causality lemma in Figure 5.3).

The specification for `sread` takes in its precondition four propositions: the socket interpretation $sa_i \Mapsto \Phi_{\mathsf{DB}_i}$, local and global snapshots $\mathsf{Seen}(i, s)$ and $\mathsf{Snap}(k, h)$ (for the key k we are reading), as well as knowledge that the key is valid. `sread` returns a value v and propositions $\mathsf{Seen}(i, s')$ and $\mathsf{Snap}(k, h')$, with $s \subseteq s'$ and $h \subseteq h'$. As explained, this represents the fact that subsequent reads return values that are more up-to-date than what a client has previously observed. Just like in the postcondition of the database's read function, there are two possibilities. Possibly the key has not been set, in which case $v = \mathsf{None}$ and the section $s'_{|k}$ is empty. Alternatively, the returned value is $\mathsf{Some}\ w$ and the value w of the key comes from an apply event a that is maximal. The erasure of a is present in the global snapshot.

Finally we describe the specification of `swrite`. The preconditions of `swrite` are the same as those of `sread`. Similarly, in the postcondition we obtain updated resources $\mathsf{Seen}(i, s')$ and $\mathsf{Snap}(k, h')$. We also know that the written pair (k, v) corresponds to an apply event a that is *new* (was not previously present in h), but is now present in both s' and h'. The usual maximality conditions hold for a.

5.5 Session Guarantees

The session guarantees describe the relationship between reads and writes happening within the same session. Our version of the guarantees are separation logic specifications for four programs. Each program consists of a client connecting to the database and performing two reads or writes (hence the four combinations):

Guarantee	**Program**		
Read Your Writes	sconnect(ip);	swrite(ip,k,v);	sread(ip,k)
Monotonic Reads	sconnect(ip);	sread(ip,k);	sread(ip,k)
Monotonic Writes	sconnect(ip);	swrite(ip,k1,v1);	swrite(ip,k2,v2)
Writes Follow Reads	sconnect(ip);	sread(ip,k1);	swrite(ip,k2,v)

Figure 5.9 shows specifications for each of the four guarantees. All four specifications take as a precondition knowledge that we are connecting to a database replica, expressed via the socket interpretation $sa_i \Mapsto \Phi_{\mathsf{DB}_i}$. The specifications have been simplified by omitting some administrative network-related resources.

Read Your Writes The specification for RYW says that there are apply (local) events a_w and a_r corresponding to the client's write and read operations. These two events are recorded in the local history $\mathsf{Seen}(i, s)$ (s is also existentially quantified). Crucially, the value v_r returned by the read (notice that the read cannot return None, since we have just written to the same key) is not less recent than the written value: $\neg(a_r.t < a_w.t)$. The event containing v_r could be the same, more recent, or concurrent compared to the write event.

Monotonic Reads The MR specification has a postcondition with three disjunctions, depending on which reads returned an empty result. In particular, notice the "missing" case because

SM-READ-YOUR-WRITES

$$\{sa_i \Mapsto \Phi_{\mathrm{DB}_i}\}$$

$$\langle ip_{client};\ \mathsf{sconnect}(ip_i); \mathsf{swrite}(ip_i, k, v_w); \mathsf{sread}(ip_i, k)\rangle$$

$$\left\{ v_o.\ \begin{array}{l} \exists s, a_w, a_r, v_r.\ v_o = \mathsf{Some}\ v_r * a_w.k = k * a_w.v = v_w * a_r.k = k * a_r.v = v_r \\ * \mathsf{Seen}(i, s) * a_w, a_r \in s * \neg(a_r.t < a_w.t) \end{array} \right\}$$

SM-MONOTONIC-READS

$$\{sa_i \Mapsto \Phi_{\mathrm{DB}_i}\}$$

$$\langle ip_{client};\ \mathsf{sconnect}(ip_i); \mathsf{let}\ v_1 = \mathsf{sread}(ip_i, k)\ \mathsf{in}\ \mathsf{let}\ v_2 = \mathsf{sread}(ip_i, k)\ \mathsf{in}\ (v_1, v_2)\rangle$$

$$\left\{ v_o.\ \begin{array}{l} \exists v_{o1}, v_{o2}.\ v_o = (v_{o1}, v_{o2}) \\ * \left(\begin{array}{l} (\exists s. v_{o1} = \mathsf{None} * v_{o2} = \mathsf{None} * \mathsf{Seen}(i, s) * s_{|k} = \emptyset) \\ \vee \\ (\exists s, v_2, a_2. v_{o1} = \mathsf{None} * v_{o2} = \mathsf{Some}\ v_2 * \mathsf{Seen}(i, s) \\ * a_2.k = k * a_2.v = v_2 * a_2 \in \mathrm{Maximals}(s_{|k})) \\ \vee \\ (\exists s, v_1, v_2, a_1, a_2. v_{o1} = \mathsf{Some}\ v_1 * v_{o2} = \mathsf{Some}\ v_2 * \mathsf{Seen}(i, s) \\ * a_1.k = k * a_1.v = v_1 * a_2.k = k * a_2.v = v_2 * a_1 \in s * a_2 \in \mathrm{Maximals}(s_{|k}) \\ * \neg(a_2.t < a_1.t)) \end{array} \right) \end{array} \right\}$$

SM-MONOTONIC-WRITES

$$\{sa_i \Mapsto \Phi_{\mathrm{DB}_i}\}$$

$$\langle ip_{client};\ \mathsf{sconnect}(ip_i); \mathsf{swrite}(ip_i, k_1, v_1); \mathsf{swrite}(ip_i, k_2, v_2)\rangle$$

$$\left\{ \begin{array}{l} \exists s_1, a_1, a_2.\ a_1.k = k_1 * a_1.v = v_1 * a_2.k = k_2 * a_2.v = v_2 \\ * \mathsf{Seen}(i, s_1) * a_1, a_2 \in s_1 * a_1.t < a_2.t \\ * (\forall j, a, s_2, a_2. \mathsf{Seen}(j, s_2) * a \in s_2 * a_2.t \le a.t \\ \qquad \Rrightarrow\!\!*_\top \exists a_1', a_2'.\ \lfloor a_1' \rfloor = \lfloor a_1 \rfloor * \lfloor a_2' \rfloor = \lfloor a_2 \rfloor * a_1', a_2' \in s_2 * a_1'.t < a_2'.t) \end{array} \right\}$$

SM-WRITES-FOLLOW-READS

$$\{sa_i \Mapsto \Phi_{\mathrm{DB}_i}\}$$

$$\langle ip_{client};\ \mathsf{sconnect}(ip_i); \mathsf{let}\ v = \mathsf{sread}(ip_i, k_r)\ \mathsf{in}\ \mathsf{swrite}(ip_i, k_w, v_w); v\rangle$$

$$\left\{ v_o.\ \begin{array}{l} \exists s_1, a_w.\ a_w.k = k_w * a_w.v = v_w * \mathsf{Seen}(i, s_1) * a_w \in s_1 \\ * \left(\begin{array}{l} v_o = \mathsf{None} \\ \vee \\ \exists a_r, v_r.\ v_o = \mathsf{Some}\ v_r * a_r.k = k_r * a_r.v = v_r * a_r \in s_1 * a_r.t < a_w.t \\ * (\forall j, a, s_2, a_2. \mathsf{Seen}(j, s_2) * a \in s_2 * a_w.t \le a.t \\ \qquad \Rrightarrow\!\!*_\top \exists a_r', a_w'.\ \lfloor a_r' \rfloor = \lfloor a_r \rfloor * \lfloor a_w' \rfloor = \lfloor a_w \rfloor * a_r', a_w' \in s_2 * a_r'.t < a_w'.t) \end{array} \right) \end{array} \right\}$$

Figure 5.9: Session guarantees. Copied from Gondelman et al. [Gon+21].

if the first read returns a result then the key is populated, so the second read cannot return None. The options are thus

- If both reads return empty, then the section $s_{|k}$ from the local snapshot $\mathsf{Seen}(i, s)$ must be empty.
- If the first read returns something but the second one returns nothing, then all we know is that the returned value is maximal.
- In the third case, let v_1 and v_2 be the values returned by the first and second read, respectively. Let a_1 and a_2 be the corresponding apply events, both present in $\mathsf{Seen}(i, s)$. Then again we can conclude that the second read corresponds to an event that is not less recent than the event of the first read.

Monotonic Writes The specification for MW says that there are two events a_1 and a_2 corresponding to the first and second write, respectively. Because both writes happened in the same session, we can conclude that the second one is strictly more recent than the first one: $a_1.t < a_2.t$ (in particular, we know they are not concurrent). We also know that both writes are recorded at replica i because a_1 and a_2 are in $\mathsf{Seen}(i, s_1)$.

Recall that the MW guarantee says that not only are the writes in the expected order at the replica where they were executed, but that they are propagated in that same order by all other replicas. This is reflected in the specification through the implication

$$\begin{aligned}&\forall j, a, s_2, a_2. \mathsf{Seen}(j, s_2) * a \in s_2 * a_2.t \leq a.t\\ &\qquad\qquad\Rrightarrow_{\top} \exists a_1', a_2'.\ \lfloor a_1' \rfloor = \lfloor a_1 \rfloor * \lfloor a_2' \rfloor = \lfloor a_2 \rfloor * a_1', a_2' \in s_2 * a_1'.t < a_2'.t\end{aligned}$$

What we are saying here is that for *any* replica j, if we can prove that its state includes the snapshot $\mathsf{Seen}(j, s_2)$ and there is an event a in s_2 that is at least as recent as a_2 (the event corresponding to the second write), then events like a_1 and a_2 are also present in s_2 and have the right order. By "an event like a_1" we mean an event a_1' whose erasure equals the erasure of a_1 (in particular, they have the same key and value). Finally, notice that this is not a regular implication: instead it is a view shift $\Rrightarrow_{\top}$. This is because in order to prove the implication we need to use the causality lemma, which requires opening the global invariant $\boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}}$. In other words, the MW specification says that if we wait long enough the two write events will appear in any replica in the right order.

Writes Follow Reads The specification of WFR has a postcondition containing the write event $a_w \in s_1$, with $\mathsf{Seen}(i, s_1)$. Then follows a disjunction:

- If the read returns None, then we do not learn anything else because there is no initial read event.
- If the read returns a value v_r, then we learn of the existence of a read event $a_r \in s_1$. We also know that the read and write events are properly sequenced: $a_r.t < a_w.t$. Then follows an implication identical to the monotonic writes example: this implication says that the read event is executed before the write event in all nodes that are sufficiently up-to-date.

5.5.1 Proving the Guarantees

Given the session manager's specifications, the proofs of the session guarantees follow straightforwardly by (1) obtaining initial local and global snapshots from the postcondition of sconnect and (2) threading the local and global snapshots through calls to sread and swrite. We show the proof outline for MW:

$$
\begin{array}{l}
\{sa_i \Mapsto \Phi_{\mathsf{DB}_i}\} \\
\quad \langle ip_{client};\ \mathsf{sconnect}(ip_i)\rangle \\
\left\{\mathsf{Seen}(i, s) * \bigstar_{k \in \mathsf{Keys}} \exists h_k. \mathsf{Snap}(k, h_k) * \boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}}\right\} \\
\{\mathsf{Snap}(k_1, h_{k_1})\} \\
\quad \langle ip_{client};\ \mathsf{swrite}(ip_i, k_1, v_1)\rangle \\
\left\{\begin{array}{l}\mathsf{Seen}(i, s') * \mathsf{Snap}(k_1, h'_{k_1}) * s \subseteq s' * h_{k_1} \subseteq h'_{k_1} \\ a_1 \in s' * \lfloor a_1 \rfloor \in h'_{k_1} * a_1.k = k_1 * a_1.v = v_1\end{array}\right\} \\
\{\mathsf{Snap}(k_2, h_{k_2})\} \\
\quad \langle ip_{client};\ \mathsf{swrite}(ip_i, k_2, v_2)\rangle \\
\left\{\begin{array}{l}\mathsf{Seen}(i, s'') * \mathsf{Snap}(k_2, h'_{k_2}) * s' \subseteq s'' * h_{k_2} \subseteq h'_{k_2} \\ a_2 \in s'' * a_2 \in h'_{k_2} * a_2.k = k_2 * a_2.v = v_2 * \mathrm{Maximum}(s'') = \mathsf{Some}\ a_2 \\ a_1 \in s'' * a_1.t < a_2.t\end{array}\right\}
\end{array}
$$

At this point, we are left with proving the implication

$$
\begin{array}{l}
\forall j, a, s_2, a_2. \mathsf{Seen}(j, s_2) * a \in s_2 * a_2.t \leq a.t \\
\qquad\qquad \Rrightarrow_\top \exists a'_1, a'_2.\ \lfloor a'_1 \rfloor = \lfloor a_1 \rfloor * \lfloor a'_2 \rfloor = \lfloor a_2 \rfloor * a'_1, a'_2 \in s_2 * a'_1.t < a'_2.t
\end{array}
$$

This follows by two applications of the causality lemma (Figure 5.3) and the properties of erasure. To apply causality we need the global invariant, but we know of it from the postcondition of sconnect.

5.6 Proving the Session Manager

The session manager's safety proof can be conceptually divided into three parts: the proofs of the request handler and client stub, and the definition of the socket protocol that acts as "glue" between the previous two components (and allows their modular verification).

5.6.1 Ghost State

Before we can tackle the socket protocol we need to define some auxiliary *ghost state*, i.e. logical state used solely to reason about the program. The intuition is that we use the ghost state to track the correspondence between a sequence ID and the associated client request.

Figure 5.10 shows the definition of logical requests and associated ghost state. A logical request has type *SMReq*, which is a tagged union, and is either a connection request, a read request, or a write request. A connection request *ConnReq*(i) models the start of a session tethered to the ith replica. A read request *ReadReq*(i, k, s, h) reads key k from the ith replica. It also includes the previously observed local and global snapshots, so that the server

Logical requests

$$\begin{array}{rll} \mathit{SMReq} \triangleq & \mathit{ConnReq}(i) & \in \mathbb{N} \\ \uplus & \mathit{ReadReq}(i,k,s,h) & \in \mathbb{N} \times \mathit{Keys} \times \mathit{LocalHistory} \times \mathit{GlobalMemory} \\ \uplus & \mathit{WriteReq}(i,k,v,s,h) & \in \mathbb{N} \times \mathit{Keys} \times \mathsf{Value} \times \mathit{LocalHistory} \times \mathit{GlobalMemory} \end{array}$$

Mapping of sequence ids to requests

$$\mathit{SeqId} \triangleq \mathbb{N}$$

$$\mathit{ReqMap} \triangleq \mathit{Auth}(\mathit{Map}(\mathit{SeqId}, \mathit{Agree}(\mathit{SMReq})))$$

$$x \in \mathit{ReqMap} \implies x = \begin{cases} \bullet\, m & \text{mapping is } \textit{exactly } m \\ \circ\, m & \text{mapping is } \textit{at least} \text{ (includes) } m \end{cases}$$

$$\mathsf{IsReq}(\gamma, n, r) \triangleq \boxed{\circ_{\mathit{ReqMap}}\, \{n \mapsto \mathsf{ag}(r)\}}^{\gamma}$$

Properties of request resources

$$\mathsf{persistent}(\mathsf{IsReq}(\gamma,n,r)) \qquad \text{(ReqPers)}$$

$$\top \vdash \mathrel{|\!\!\Rrightarrow} \exists \gamma. \boxed{\bullet_{\mathit{ReqMap}}\, \emptyset}^{\gamma} \qquad \text{(ReqInit)}$$

$$\mathsf{IsReq}(\gamma,n,r) * \mathsf{IsReq}(\gamma,n,r') \vdash r = r' \qquad \text{(ReqAgree)}$$

$$n \notin \mathsf{keys}(m) * \boxed{\bullet_{\mathit{ReqMap}}\, m}^{\gamma} \vdash \mathrel{|\!\!\Rrightarrow} \boxed{\bullet_{\mathit{ReqMap}}\, (m \cup \{n \mapsto \mathsf{ag}(r)\})}^{\gamma} * \mathsf{IsReq}(\gamma,n,r) \qquad \text{(ReqAlloc)}$$

$$\boxed{\bullet_{\mathit{ReqMap}}\, m}^{\gamma} * \mathsf{IsReq}(\gamma,n,r) \vdash n \in \mathsf{keys}(m) \qquad \text{(ReqElem)}$$

Figure 5.10: Requests and associated resources.

can reply with a proof that the new snapshots are a superset of the old ones. A write request $\mathit{WriteReq}(i,k,v,s,h)$ writes value v to key k at the ith replica. It too includes snapshot information.

Next we define the ghost state for tracking sequence ids, natural numbers, to the corresponding requests. We build our ghost state using standard Iris combinators [BB17]. The resource algebra *ReqMap* of request maps is

$$\mathit{Auth}(\mathit{Map}(\mathit{SeqId}, \mathit{Agree}(\mathit{SMReq})))$$

The elements of this resource algebra are either *full parts* of the form $\bullet\, m$ or *fragmental parts* of the form $\circ\, m$. The extension order $\preceq$ of the underlying maps is given by $m_1 \preceq m_2$ if $\mathsf{keys}(m_1) \subseteq \mathsf{keys}(m_2)$ and both maps agree on the mapped values for the set of common keys. The agreement is enforced by the *agreement* construction $\mathit{Agree}(\mathit{SMReq})$: $\mathsf{ag}(r)$ is the constructor that injects a request into this resource algebra.

We then define the request resource $\mathsf{IsReq}(\gamma,n,r)$. Intuitively, this proposition says that r is the nth request issued by a client. The resource is defined as the fragmental part containing the singleton map $\boxed{\circ_{\mathit{ReqMap}}\, \{n \mapsto \mathsf{ag}(r)\}}^{\gamma}$.

Finally, we prove a number of lemmas about the request resource. (ReqPers) says that $\mathsf{IsReq}(\gamma,n,r)$ is persistent, so it can duplicated. This is useful because in the proof the client will "send" a *copy* of $\mathsf{IsReq}(\gamma,n,r)$ as a certificate that r really is the nth request; the certificate will then be returned by the server. (ReqInit) shows we can initialize the full part of the resource with the empty map of requests. (ReqAgree) shows that there can only be one request with sequence number n: this is also crucial for matching up the server's responses with the client's requests. (ReqAlloc) shows that we can allocate a new request by picking a new sequence ID. Finally, (ReqElem) shows that a certified request has necessarily been seen before.

5.6.2 Socket Protocol

Recall that a socket protocol is a predicate on messages that must be satisfied by all messages received at a socket address (modulo duplicated messages). One can think of the socket protocol in a dual way: from the sender's perspective, the protocol is a *proof obligation*; from the receiver's perspective, it is an *assumption*. Figure 5.11 shows the definition of the socket protocols for the request handler and client stub.

We first define the set of logical responses as a tagged union. Notice that only reads include extra information in a response: the value v that was read. The responses for starting a session and writing to the database are just acknowledgements that the operation has completed (but they do communicate additional logical information, such as the knowledge of local and global snapshots).

The request handler uses as family of socket protocols Φ_{DB_i}, parameterized on the replica ID i. The protocol says that we can assume that the sender of the message uses a socket protocol ψ. We also require a certificate $\mathsf{IsReq}(\gamma, n, r)$ showing that the request was made by the client stub (who as we will see produces these request resources). Based on the kind of request we receive, we then compute a pair (P, Q) of pre- and post-conditions for the request. The request handler can assume P and will need to prove $Q(q)$ in order to reply to the client, where q is the response to the request. This is expressed via the proposition

$$\square\begin{pmatrix}\forall m_{res}, q.\ \mathsf{IsValidRes}(m_{res}, n, q) * \mathsf{IsReq}(\gamma, n, r) * \mathsf{ReqResMatch}(r, q) * \\ \qquad Q(q) \mathrel{-\!\!*} \psi(m_{res})\end{pmatrix}$$

This proposition says that for all responses m_{res}, provided that $Q(q)$ (the postcondition), together with some other facts, holds then we know the client's socket protocol $\psi(m_{res})$ holds of the response. In other words, we can respond to the client provided we know $Q(q)$. Notice also that we need to "send back" a copy of $\mathsf{IsReq}(\gamma, n, r)$: this is so the client can match their request to the response. Additionally, notice that the entire proposition is under the persistently modality. Intuitively, this means when replying to the client we need to prove $Q(q)$ using only persistent propositions. The reason for using the persistently modality is that it makes the verification of the request handler easier: specifically, we can "remember" (by duplicating the above proposition) what form an answer to a request should have, even if the request is duplicated.

Finally, to compute the pre- and post-conditions for a request, we pattern match on the request type. The preconditions include the local and global snapshots, so we can prove that the response is more up-to-date than the request. The postconditions are exactly the postconditions used by the session manager specifications in Figure 5.8.

The client's socket protocol $\Phi_{\mathsf{client}_\gamma}$ is parameterized on the ghost name γ (created by the client when allocating the ghost state for tracking the mapping of sequence IDs to requests). This socket protocol requires the certificate $\mathsf{IsReq}(\gamma, n, r)$ and then connects the request r to a response q (corresponding to m_{res}), and then in turn connects q to the appropriate session manager postcondition from Figure 5.8.

Logical responses

$$\mathit{SMRes} \triangleq \mathit{ConnRes}() \uplus \mathit{ReadRes}(v) \uplus \mathit{WriteRes}()$$

Request handler's (server) socket protocol

ψ	client's socket protocol
n	request sequence id
γ	client-supplied ghost name
r, q	logical request and response
m, m_{res}	physical request and response
P, Q	pre and post conditions obtained from the request

$$\begin{aligned}\Phi_{\mathsf{DB}_i}(m) \triangleq\ & \exists \psi, n, \gamma, r, P, Q.\\ & m.\mathsf{orig} \Mapsto \psi * \mathsf{IsValidReq}(m.\mathsf{body}, n, r) * \mathsf{IsReq}(\gamma, n, r) * \mathsf{ReqDb}(r) = i *\\ & (P, Q) = \mathsf{SelectPrePost}(r) *\\ & P *\\ & \square\begin{pmatrix}\forall m_{res}, q.\ \mathsf{IsValidRes}(m_{res}, n, q) * \mathsf{IsReq}(\gamma, n, r) * \mathsf{ReqResMatch}(r, q) *\\ Q(q) \twoheadrightarrow \psi(m_{res})\end{pmatrix}\end{aligned}$$

$$\begin{aligned}\mathsf{SelectPrePost}(\mathit{ConnReq}(i)) &= (\top, \lambda q.\mathsf{ConnectPost}(i))\\ \mathsf{SelectPrePost}(\mathit{ReadReq}(i,k,s,h)) &= \begin{pmatrix}k \in \mathit{Keys} * \mathsf{Seen}(i,s) * & , \lambda q. & \exists v.\ q = \mathit{ReadRes}(v) *\\ \mathsf{Snap}(k,h) & & \mathsf{ReadPost}(i,k,s,h,v)\end{pmatrix}\\ \mathsf{SelectPrePost}(\mathit{WriteReq}(i,k,v,s,h)) &= \begin{pmatrix}k \in \mathit{Keys} * \mathsf{Seen}(i,s) * & , \lambda\_.\mathsf{WritePost}(i,k,v,s,h)\\ \mathsf{Snap}(k,h) & \end{pmatrix}\end{aligned}$$

$\mathsf{IsValidReq}(s, n, r) \triangleq$ Does the message body s contain the logical request r with sequence number n?

$\mathsf{ReqResMatch}(r, q) \triangleq$ Does the request r match the response q?

$\mathsf{ReqDb}(r) \triangleq$ The replica ID for request r.

$\mathsf{ConnectPost}, \mathsf{ReadPost}, \mathsf{WritePost} \triangleq$ postconditions in Figure 5.8

Client stub's socket protocol

$$\begin{aligned}\Phi_{\mathsf{client}_\gamma}(m) \triangleq\ & \exists n, r, q, v.\\ & \mathsf{IsValidRes}(m.\mathsf{body}, n, q) * \mathsf{IsReq}(\gamma, n, r) * \mathsf{ReqResMatch}(r, q) *\\ & \mathsf{SelectResPost}(r, q, v)\end{aligned}$$

$$\begin{aligned}\mathsf{SelectResPost}(\mathit{ConnReq}(i), q, v) &= \mathsf{ConnectPost}(i)\\ \mathsf{SelectResPost}(\mathit{ReadReq}(i,k,a,h), q, v) &= (q = \mathit{ReadRes}(v) * \mathsf{ReadPost}(i,k,a,h,v))\\ \mathsf{SelectResPost}(\mathit{WriteReq}(i,k,v,a,h), q, v) &= \mathsf{WritePost}(i,k,v,s,h)\end{aligned}$$

$\mathsf{IsValidRes}(s, n, q) \triangleq$ Does the message body s contain the logical response q with sequence number n?

Figure 5.11: Client and server socket protocols

5.6.3 *Request Handler Proof*

Recall that the request handler (Figure 5.7) consists of set up code that initializes the database replica, obtains accessors for reading and writing to the database, and then enters an infinite "loop" (implemented via recursion) waiting for requests and then replying to them.

We do not show the full proof of the request handler, but instead highlight a few key ideas:

- Just like at the implementation level the request handler is a thin layer over the distributed database, its correctness proof "just" re-arranges the pre- and post-conditions of the database specifications to meet the handler's specifications. For example, the session manager functions require that we "grow" the local and global snapshots $\mathsf{Seen}(i, a)$ and $\mathsf{Snap}(k, h)$ into snapshots $\mathsf{Seen}(i, a')$ and $\mathsf{Snap}(k, h')$ with $a \subseteq a'$ and $h \subseteq h'$. Examining the database's read and write specifications in Figure 5.4, notice that they already take a local snapshot in the precondition and return an updated copy. To grow the global snapshot we use the (Take Snap) lemma (Figure 5.3) to turn a user points-to $k \rightharpoonup_u h'' \uplus \{w\}$ into the persistent predicate $\mathsf{Snap}(k, h'' \uplus \{w\})$, and then use the (Snap Union) lemma to obtain $\mathsf{Snap}(k, h) * \mathsf{Snap}(k, h'' \uplus \{w\}) \vdash \mathsf{Snap}(k, h \cup h'' \uplus \{w\})$. We can then let $h' = h \cup h'' \uplus \{w\}$ as needed.

- Related to the previous point, notice that in order to execute a write operations against key k of the database we need the exclusive resource $k \rightharpoonup_u h$. This resource is not provided by the session manager, because the latter uses online persistent resources. To that effect we maintain a *memory invariant* $\boxed{\mathsf{MemInv}}^{\mathcal{N}_{\mathsf{MI}}}$:

  $$\mathsf{MemInv} \triangleq \mathop{\Large\ast}_{k \in \mathsf{Keys}} \exists h_k . k \rightharpoonup_u h_k$$

  That is, we keep *all* the points-to resources for the database in an invariant: the request handler can then open the invariant and retrieve the right points-to when writing to the local replica.

- Finally, in order to verify the infinite loop in the request handler we use a loop invariant:

  $$\mathbf{LI} \triangleq \exists R, T.\ sa \rightsquigarrow (R, T) * \mathop{\Large\ast}_{m \in R} \Phi_{\mathsf{DB}_i}(m)$$

  The invariant says that the set of messages received at socket address sa (on which the request handler is communicating) is R. Furthermore, for every received message m we know that $\Phi_{\mathsf{DB}_i}(m)$ holds. In particular, we know that we can respond to the sender of the request by satisfying the post-condition in Φ_{DB_i}. This means we can respond just by forwarding the operation to the database and returning whatever the database returned. Recall that the post-condition of Φ_{DB_i} is under the persistently modality, so we can reuse the post-condition as needed to reply to duplicated messages.

5.6.4 *Client Stub Proof*

The main challenge in verifying the client stub is in building a logical notion of session. Specifically, once we send a request to the server, we need a way to correlate that sent request to the response provided by the server. We build the logical session using the authoritative resource $\bullet_{\mathit{ReqMap}}\, m$. Here m is a map from sequence IDs (natural numbers) to associated logical requests. In particular, the entry in the map with the highest sequence ID corresponds to the last (in-flight) request, which is the one for which we need an answer (by the time we are sending request n all requests numbered less than n have already been answered). We link the map m to the client's physical state via a *lock invariant* $\text{isLock}(ip, \ell_{\mathsf{sid}}, \mathsf{LockInv}_{ip}(\ell_{\mathsf{sid}}, \gamma, z, a))$. The isLock predicate is standard from Iris [BB17], and should be read as saying that at node ip the memory location ℓ_{sid} is protected with a lock that guarantees the proposition $\mathsf{LockInv}_{ip}(\ell_{\mathsf{sid}}, \gamma, z, a)$. When a thread acquires the lock, it gains access to the guarded proposition. Dually, to release the lock we have to prove that the guarded proposition continues to hold. The LockInv is defined thus:

$$
\begin{aligned}
\mathsf{LockInv}_{ip}(\ell_{\mathsf{sid}}, \gamma, z, sa) \triangleq \exists n, m, R, S.& \\
&\ell_{\mathsf{sid}} \mapsto_{ip} n * \boxed{\bullet_{\mathit{ReqMap}}\, m}^{\gamma} * (\forall m \geq n, m \notin \mathsf{dom}(m)) * \\
&z \overset{ip}{\hookrightarrow} \texttt{socket}\ sa * sa \rightsquigarrow (R, T) * \\
&\mathop{\LARGE *}_{msg \in R} \exists n', q. \mathsf{IsValidRes}(msg.\mathsf{body}, n', q) * n' < n
\end{aligned}
$$

The lock invariant says that there is a current sequence number n, a request map m, and received and transmitted message sets R and T such that the memory location ℓ_{sid} points to n and the map m does not include any requests with sequence number greater than or equal to n. We also assert ownership of the full part $\bullet_{\mathit{ReqMap}}\, m$. Intuitively, this means the client know *exactly* the set of previously-issued requests (this makes sense because it is the client that issued said requests). The invariant says that the socket handle z points to the socket with address sa and that on address sa we have sent and received the sets of messages R and T, respectively. Finally, we assert that all received messages correspond to valid responses to some request, but crucially all the already-received responses have sequence number strictly less than n.

`listen_wait_for_seqid` The helper `listen_wait_for_seqid s n` loops until a response with sequence ID `n` is received. To verify it we require in the precondition of the function the certificate $\mathsf{IsReq}(\gamma, n, r)$ indicating that the current request is r. We also get to assume the lock invariant, since by the time we enter this function `session_exec` has acquired the lock. The postcondition of the function is the preserved lock invariant together with knowledge that the response satisfies the postcondition of the appropriate session manager API (i.e. connect, read, or write, depending on the type of response). After receiving a message, we consider two cases:

- If the message's sequence ID is not n, then we can proceed by Löb induction.
- If the sequence ID is n, then again we consider two cases:
  - If the received message msg is new, then we can assume the socket protocol $\Phi_{\mathsf{client}_i}(msg)$, which already implies the postcondition of the session manager API. However, we still need to know that the message we received is a response to the *same* request that we issued. We do so issuing the (ReqAgree) property from Figure 5.10 and we are done.
  - If the received message is not new, then we use the part of the lock invariant that says that all previously-received messages have sequence number $< n$, arriving at a contradiction.

`session_exec` The `session_exec` function acquires the lock, sends appropriate request to the server, waits for a response using `listen_wait_for_seqid` then releases the lock. To verify this function, we take as a precondition all propositions needed to send a message (e.g. $\mathsf{Seen}(i, s)$ and $\mathsf{Snap}(k, h)$). Before sending a message, we use the (ReqAlloc) lemma Figure 5.10 to insert a new key in the map m corresponding to the new message being created. (ReqAlloc) gives us the certificate $\mathsf{IsReq}(\gamma, n, r)$ needed for the server's socket protocol. After calling `listen_wait_for_seqid` we can assume the latter's postcondition, which implies the postconditions of the session manager functions. In order to release the lock we need to prove that the lock invariant continues to hold. This is the case because, among others, not only did we allocate a new logical request, but we also incremented the sequence ID pointer inside the session manager.

`sm_setup` The verification of `sm_setup` is straightforward since all we need to do is allocate a few logical resources (e.g. for the lock) and call `session_exec`, which does the heavy lifting.

5.7 Related Work

The four session guarantees were introduced in Terry et al. [Ter+94] as a way for clients to selectively-strengthen the weak-consistency guarantees of their Bayou replicated database [Pet+96]. In addition to defining the guarantees, they also show how to implement them, both naively and more efficiently using version vectors.

A number of papers have formally defined session guarantees in terms of event histories [BSW04; ZW10], as well explored the relation between the guarantees and PRAM [BSW03] and causal consistency [BSW04].

Client-centric consistency has experience somewhat of a renaissance with the adoption of cloud storage and NoSQL solutions [Vog09]. These systems usually implement some form of eventual consistency, which might or might not imply the session guarantees. Some of these systems do not explicitly advertise support for specific session guarantees, but nevertheless support them *in practice*, at least for a percentage of requests or in "typical" scenarios (e.g. absent server failures) [BK13].

Because the session guarantees are designed so they can be observed by clients, they are amenable to benchmarking by client code [BT11; Wad+11]. Viewed from this perspective, support for a consistency guarantee is not a binary question, but instead a quantifiable measure that is more akin to system availability. For example, Bermbach and Tai [BT11] show that, as of 2011, 12% of reads to Amazon S3 violated monotonic reads.

The related work section of the paper that this chapter is based on [Gon+21] lists multiple formal method techniques that target verification of weakly-consistent data stores [CBG15; Got+16; Kak+18; LBC16]. None of these specify or verify session guarantees.

5.8 Conclusions

We have shown four session guarantees that apply to Gondelman et al. [Gon+21]'s causally consistent database. We did so by implementing and verifying a session manager library that uses Gondelman et al. [Gon+21]'s distributed database in a client-server setting. The session guarantees then follow as four example clients that use the session manager. Ours is the first formal treatment of session guarantees that we are aware of, and in line with prior work our results confirm that causal consistency implies all four session guarantees. This serves as evidence that Gondelman et al. [Gon+21]'s definition of causality in separation logic accurately captures the essence of causal consistency.

Future Work Our session manager specifications use only persistent resources. This opens up the window for automatic verification of clients of the session manager, since ephemeral resources are harder to work with, and so some clients of the session manager might not need the full power of separation logic.

As previously mentioned, our sessions make the simplifying assumption that a client only connects to one server. It would be interesting to relax this assumption; in order to do so we would need to change the implementation of Gondelman et al. [Gon+21]'s causally consistent database so that database operations return the associated vector clocks. Exposing vector clock would in turn allow us to extend the session manager with replica selection logic.

Bibliography

[Aha+95] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. “Causal Memory: Definitions, Implementation, and Programming”. In: *Distributed Comput.* 9.1 (1995), pp. 37–49. DOI: 10.1007/BF01784241 (cit. on pp. 4, 31, 91).

[ASB18] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. “Delta state replicated data types”. In: *J. Parallel Distributed Comput.* 111 (2018), pp. 162–173. DOI: 10.1016/j.jpdc.2017.08.003 (cit. on p. 11).

[App01] Andrew W Appel. “Foundational proof-carrying code”. In: *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*. IEEE. 2001, pp. 247–256 (cit. on p. 22).

[Att+16] Hagit Attiya, Sebastian Burckhardt, Alexey Gotsman, Adam Morrison, Hongseok Yang, and Marek Zawirski. “Specification and Complexity of Collaborative Text Editing”. In: *ACM Symposium on Principles of Distributed Computing*. PODC 2016. ACM, July 2016, pp. 259–268. DOI: 10.1145/2933057.2933090 (cit. on p. 19).

[Bai+13] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. “Bolt-on causal consistency”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*. 2013, pp. 761–772. DOI: 10.1145/2463676.2465279 (cit. on p. 31).

[BAS14] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. “Making operation-based CRDTs operation-based”. In: *Proceedings of the First Workshop on the Principles and Practice of Eventual Consistency, PaPEC@EuroSys 2014, April 13, 2014, Amsterdam, The Netherlands*. Ed. by Marc Shapiro. ACM, 2014, 7:1–7:2. DOI: 10.1145/2596631.2596632 (cit. on pp. 12, 23, 32, 37–39, 43, 45, 46).

[BK13] David Bermbach and Jörn Kuhlenkamp. “Consistency in distributed storage systems: An overview of models, metrics and measurement approaches”. In: *Networked Systems: First International Conference, NETYS 2013, Marrakech, Morocco, May 2-4, 2013, Revised Selected Papers*. Springer. 2013, pp. 175–189 (cit. on p. 113).

[BT11] David Bermbach and Stefan Tai. “Eventual consistency: How soon is eventual? An evaluation of Amazon S3’s consistency behavior”. In: *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing*. 2011, pp. 1–6 (cit. on p. 113).

[BB17] Lars Birkedal and Ales Bizjak. “Lecture Notes on Iris: Higher-Order Concurrent Separation Log”. In: (2017) (cit. on pp. 18, 23, 45, 48, 52, 85, 108, 112).

[BSS91] Kenneth Birman, Andre Schiper, and Pat Stephenson. “Lightweight Causal and Atomic Group Multicast”. In: *ACM Transactions on Computer Systems (TOCS)* 9.3 (1991), pp. 272–314. DOI: 10.1145/128738.128742 (cit. on pp. 3, 22, 23, 39, 40, 63).

[Bir86] Kenneth P. Birman. *ISIS: A System for Fault-Tolerant Distributed Computing.* Tech. rep. USA, 1986 (cit. on p. 3).

[BSW03] Jerzy Brzezinski, Cezary Sobaniec, and Dariusz Wawrzyniak. “Session guarantees to achieve PRAM consistency of replicated shared objects”. In: *International Conference on Parallel Processing and Applied Mathematics.* Springer. 2003, pp. 1–8 (cit. on p. 113).

[BSW04] Jerzy Brzezinski, Cezary Sobaniec, and Dariusz Wawrzyniak. “From Session Causality to Causal Consistency”. In: *PDP.* IEEE Computer Society, 2004, pp. 152–158 (cit. on pp. 25, 99, 113).

[Bur+14] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. “Replicated Data Types: Specification, Verification, Optimality”. In: *41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL 2014. ACM, Jan. 2014, pp. 271–284. DOI: 10.1145/2535838.2535848 (cit. on pp. 19–23, 32, 37, 46, 49, 56, 57, 60, 64, 65, 88).

[CGR11] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. “Introduction to Reliable and Secure Distributed Programming”. In: Springer Science & Business Media, 2011. Chap. 3 (cit. on pp. 32, 35, 38, 39, 41).

[CBG15] Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. “A Framework for Transactional Consistency Models with Atomic Visibility”. In: *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015.* 2015, pp. 58–71. DOI: 10.4230/LIPIcs.CONCUR.2015.58 (cit. on p. 113).

[Cha+19] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. “Verifying concurrent, crash-safe systems with Perennial”. In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019.* Ed. by Tim Brecht and Carey Williamson. ACM, 2019, pp. 243–258. DOI: 10.1145/3341301.3359632 (cit. on p. 43).

[Cha+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. “Bigtable: A Distributed Storage System for Structured Data”. In: *ACM Trans. Comput. Syst.* 26.2 (2008), 4:1–4:26. DOI: 10.1145/1365815.1365816 (cit. on p. 31).

[CD10] Kristina Chodorow and Michael Dirolf. *MongoDB - The Definitive Guide: Powerful and Scalable Data Storage.* O’Reilly, 2010 (cit. on p. 31).

[Dan+22] Hoang-Hai Dang, Jaehwang Jung, Jaemin Choi, Duc-Than Nguyen, William Mansky, Jeehoon Kang, and Derek Dreyer. “Compass: strong and compositional library specifications in relaxed memory separation logic”. In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation.* 2022, pp. 792–808 (cit. on p. 88).

[DFG22] Kevin De Porre, Carla Ferreira, and Elisa Gonzalez Boix. “VeriFx: Correct Replicated Data Types for the Masses”. In: (July 2022). eprint: 2207.02502 (cit. on pp. 19, 20).

[Del+16] Germán Andrés Delbianco, Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. "Concurrent data structures linked in time". In: *arXiv preprint arXiv:1604.08080* (2016) (cit. on p. 88).

[DSM18] Stephen Dolan, K. C. Sivaramakrishnan, and Anil Madhavapeddy. "Bounding data races in space and time". In: *PLDI*. ACM, 2018, pp. 242–255 (cit. on p. 75).

[Fid87] Colin J Fidge. "Timestamps in Message-Passing Systems That Preserve the Partial Ordering". In: (1987) (cit. on p. 40).

[GL02] Seth Gilbert and Nancy A. Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". In: *SIGACT News* 33.2 (2002), pp. 51–59. DOI: 10.1145/564585.564601 (cit. on pp. 5, 30).

[Gom+17] Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. "Verifying Strong Eventual Consistency in Distributed Systems". In: *Proc. ACM Program. Lang.* 1.OOPSLA (2017), 109:1–109:28. DOI: 10.1145/3133933 (cit. on pp. 19, 20, 22, 57, 63, 65).

[Gon+20] Léon Gondelman, Simon Oddershede Gregersen, Abel Nieto, Amin Timany, and Lars Birkedal. *Distributed Causal Memory: Modular Specification and Verification in Higher-Order Distributed Separation Logic - Technical Appendix*. 2020 (cit. on pp. 27, 91, 101).

[Gon+21] Léon Gondelman, Simon Oddershede Gregersen, Abel Nieto, Amin Timany, and Lars Birkedal. "Distributed Causal Memory: Modular Specification and Verification in Higher-Order Distributed Separation Logic". In: *Proc. ACM Program. Lang.* 5.POPL (2021), pp. 1–29. DOI: 10.1145/3434323 (cit. on pp. 13, 21–27, 36, 42, 44, 45, 59, 68, 72, 78, 81, 82, 88, 89, 91–96, 98, 100, 102, 103, 105, 113, 114).

[Got+16] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. "'Cause I'm strong enough: reasoning about consistency choices in distributed systems". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. Ed. by Rastislav Bodík and Rupak Majumdar. ACM, 2016, pp. 371–384. DOI: 10.1145/2837614.2837625 (cit. on pp. 19, 20, 113).

[HW90] Maurice Herlihy and Jeannette M. Wing. "Linearizability: A Correctness Condition for Concurrent Objects". In: *ACM Trans. Program. Lang. Syst.* 12.3 (1990), pp. 463–492. DOI: 10.1145/78969.78972 (cit. on pp. 30, 43).

[Jun+18] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. "Iris from the ground up: A modular foundation for higher-order concurrent separation logic". In: *J. Funct. Program.* 28 (2018), e20. DOI: 10.1017/S0956796818000151 (cit. on pp. 13, 18, 34, 37, 66, 78, 83).

[Jun+15] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. "Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning". In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 2015, pp. 637–650. DOI: 10.1145/2676726.2676980 (cit. on pp. 23, 43).

[Kak+18] Gowtham Kaki, Kartik Nagar, Mahsa Najafzadeh, and Suresh Jagannathan. "Alone together: compositional reasoning and inference for weak isolation". In: *Proc. ACM Program. Lang.* 2.POPL (2018), 27:1–27:34. DOI: 10.1145/3158115 (cit. on p. 113).

[Kle+18] Martin Kleppmann, Victor B F Gomes, Dominic P Mulligan, and Alastair R Beresford. "OpSets: Sequential Specifications for Replicated Datatypes (Extended Version)". In: (May 2018). eprint: 1805.04263 (cit. on p. 20).

[Kre+18a] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. "MoSeL: a general, extensible modal framework for interactive proofs in separation logic". In: *PACMPL* 2.ICFP (2018), 77:1–77:30. DOI: 10.1145/3236772 (cit. on pp. 13, 28).

[Kre+18b] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. "MoSeL: a general, extensible modal framework for interactive proofs in separation logic". In: *Proc. ACM Program. Lang.* 2.ICFP (2018), 77:1–77:30. DOI: 10.1145/3236772 (cit. on p. 18).

[KTB17] Robbert Krebbers, Amin Timany, and Lars Birkedal. "Interactive proofs in higher-order concurrent separation logic". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017.* 2017, pp. 205–217 (cit. on p. 18).

[Kro+20] Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. "Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems". In: *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings.* 2020, pp. 336–365. DOI: 10.1007/978-3-030-44914-8_13 (cit. on pp. 3, 13, 18, 22, 33, 34, 65, 66).

[Lad+22] Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, and Joseph M. Hellerstein. "Katara: Synthesizing CRDTs with Verified Lifting". In: *Proceedings of the ACM on Programming Languages* 6.OOPSLA2 (Oct. 2022), pp. 1349–1377. DOI: 10.1145/3563336. eprint: 2205.12425 (cit. on pp. 19, 20).

[Lam78] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Commun. ACM* 21.7 (1978), pp. 558–565. DOI: 10.1145/359545.359563 (cit. on pp. 3, 11, 32, 63).

[Lam79] Leslie Lamport. "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs". In: *IEEE Trans. Computers* 28.9 (1979), pp. 690–691 (cit. on p. 4).

[LAB19] Adriaan Leijnse, Paulo Sérgio Almeida, and Carlos Baquero. "Higher-Order Patterns in Replicated Data Types". In: *PaPoC@EuroSys.* ACM, 2019, 5:1–5:6. DOI: 10.1145/3301419.3323971 (cit. on pp. 20, 37, 38, 49, 64, 65, 88).

[LBC16] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. “Chapar: certified causally consistent distributed key-value stores”. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016.* 2016, pp. 357–370. DOI: 10.1145/2837614.2837622 (cit. on p. 113).

[Li+14] Cheng Li, João Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. “Automating the Choice of Consistency Levels in Replicated Systems”. In: *USENIX Annual Technical Conference.* ATC 2014. USENIX, June 2014, pp. 281–292 (cit. on pp. 19, 20).

[LF21] Hongjin Liang and Xinyu Feng. “Abstraction for Conflict-Free Replicated Data Types”. In: *PLDI ’21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211.* Ed. by Stephen N. Freund and Eran Yahav. ACM, 2021, pp. 636–650. DOI: 10.1145/3453483.3454067 (cit. on pp. 19–22, 33, 57, 58, 65).

[LZ74] Barbara H. Liskov and Stephen N. Zilles. “Programming with Abstract Data Types”. In: *SIGPLAN Symposium on Very High Level Languages.* ACM, 1974, pp. 50–59 (cit. on pp. 12, 64).

[Liu+20] Yiyun Liu, James Parker, Patrick Redmond, Lindsey Kuper, Michael Hicks, and Niki Vazou. “Verifying Replicated Data Types with Typeclass Refinements in Liquid Haskell”. In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 216:1–216:30. DOI: 10.1145/3428284 (cit. on pp. 19, 20, 38, 57, 58, 65).

[Llo+11] 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: *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011.* 2011, pp. 401–416. DOI: 10.1145/2043556.2043593 (cit. on pp. 31, 93).

[Lyn96] Nancy A Lynch. *Distributed algorithms.* Elsevier, 1996 (cit. on p. 3).

[Mat+88] Friedemann Mattern et al. *Virtual Time and Global States of Distributed Systems.* Univ., Department of Computer Science, 1988 (cit. on p. 40).

[NJ19] Kartik Nagar and Suresh Jagannathan. “Automated Parameterized Verification of CRDTs”. In: *CAV (2).* Vol. 11562. Lecture Notes in Computer Science. Springer, 2019, pp. 459–477 (cit. on pp. 57, 59, 63, 65).

[NPS20] Sreeja S. Nair, Gustavo Petri, and Marc Shapiro. “Proving the Safety of Highly-Available Distributed Objects”. In: *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings.* Ed. by Peter Müller. Vol. 12075. Lecture Notes in Computer Science. Springer, 2020, pp. 544–571. DOI: 10.1007/978-3-030-44914-8_20 (cit. on pp. 19, 20, 57, 63, 65, 89).

[Nan] Aleksandar Nanevski. *Separation logic and concurrency. Oregon programming languages summer school, 2016* (cit. on p. 88).

[Nie+23] Abel Nieto, Arnaud Daby-Seesaram, Léon Gondelman, Amin Timany, and Lars Birkedal. “Modular Verification of State-Based CRDTs in Separation Logic”. In: *37th European Conference on Object-Oriented Programming*. ECOOP 2023. Schloss Dagstuhl, July 2023. DOI: `10.4230/LIPIcs.ECOOP.2023.12` (cit. on pp. 13, 27, 62).

[Nie+22] Abel Nieto, Léon Gondelman, Alban Reynaud, Amin Timany, and Lars Birkedal. “Modular verification of op-based CRDTs in separation logic”. In: *Proc. ACM Program. Lang.* 6.OOPSLA2 (2022), pp. 1788–1816 (cit. on pp. 13, 27, 30, 65, 67–73, 77, 78, 81, 82, 87–90).

[Pet+96] Karin Petersen, Mike Spreitzer, Douglas Terry, and Marvin Theimer. “Bayou: replicated database services for world-wide applications”. In: *Proceedings of the 7th workshop on ACM SIGOPS European workshop: Systems support for worldwide applications*. 1996, pp. 275–280 (cit. on p. 113).

[Red+22] Patrick Redmond, Gan Shen, Niki Vazou, and Lindsey Kuper. “Verified Causal Broadcast with Liquid Haskell”. In: *arXiv preprint arXiv:2206.14767* (2022). DOI: `10.48550/arXiv.2206.14767` (cit. on pp. 59, 60).

[Rey02] John C Reynolds. “Separation logic: A logic for shared mutable data structures”. In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE. 2002, pp. 55–74 (cit. on p. 13).

[Ros96] A. W. Roscoe. “Intensional Specifications of Security Protocols”. In: *CSFW*. IEEE Computer Society, 1996, pp. 28–38 (cit. on p. 48).

[Sha+11a] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Research Report 7506. INRIA, Jan. 2011 (cit. on pp. 5, 11, 19, 31, 32, 38, 46, 68, 73).

[Sha+11b] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. “Conflict-Free Replicated Data Types”. In: *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*. Ed. by Xavier Défago, Franck Petit, and Vincent Villain. Vol. 6976. Lecture Notes in Computer Science. Springer, 2011, pp. 386–400. DOI: `10.1007/978-3-642-24550-3\_29` (cit. on pp. 5, 31, 32, 56).

[Siv12] Swaminathan Sivasubramanian. “Amazon dynamoDB: a seamlessly scalable non-relational database service”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*. 2012, pp. 729–730. DOI: `10.1145/2213836.2213945` (cit. on p. 31).

[Sou+22] Vimala Soundarapandian, Adharsh Kamath, Kartik Nagar, and KC Sivaramakrishnan. “Certified Mergeable Replicated Data Types”. In: *43rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2022. ACM, June 2022, pp. 332–347. DOI: `10.1145/3519939.3523735`. eprint: `2203.14518` (cit. on p. 19).

[TS07] Andrew S. Tanenbaum and Maarten van Steen. *Distributed systems - principles and paradigms, 2nd Edition*. Pearson Education, 2007. ISBN: 978-0-13-239227-3 (cit. on pp. 24, 39, 91, 92).

[Tea23] The Aneris Team. *Aneris Documentation.* https://github.com/logsem/aneris/blob/master/documentation.pdf. Last updated: March 15, 2023. Accessed: June 15, 2023. 2023 (cit. on pp. 14, 16, 18).

[Teaa] The Aneris Team. *Aneris Coq Formalization.* https://github.com/logsem/aneris. Accessed June 20, 2023 (cit. on p. 18).

[Teab] The Aneris Team. *Ping Pong Example.* https://github.com/logsem/aneris/tree/master/aneris/examples/ping_pong_done. Accessed June 15, 2023 (cit. on p. 15).

[Ter+94] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent B. Welch. "Session Guarantees for Weakly Consistent Replicated Data". In: *Proceedings of the Third International Conference on Parallel and Distributed Information Systems (PDIS 94), Austin, Texas, USA, September 28-30, 1994.* 1994, pp. 140–149. DOI: 10.1109/PDIS.1994.331722 (cit. on pp. 4, 24, 92, 98, 99, 113).

[TB21] Amin Timany and Lars Birkedal. "Reasoning about Monotonicity in Separation Logic". In: *CPP.* ACM, 2021, pp. 91–104. DOI: 10.1145/3437992.3439931 (cit. on pp. 53, 83).

[Tim+21] Amin Timany, Simon Oddershede Gregersen, Léo Stefanesco, Léon Gondelman, Abel Nieto, and Lars Birkedal. "Trillium: Unifying refinement and higher-order distributed separation logic". In: *arXiv preprint arXiv:2109.07863* (2021). DOI: 10.48550/arXiv.2109.07863 (cit. on pp. 13, 19, 48, 56, 57, 65, 89).

[Tyu+19] Misha Tyulenev, Andy Schwerin, Asya Kamsky, Randolph Tan, Alyson Cabral, and Jack Mulrow. "Implementation of Cluster-wide Logical Clock and Causal Consistency in MongoDB". In: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019.* 2019, pp. 636–650. DOI: 10.1145/3299869.3314049 (cit. on p. 31).

[Vaz+14] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. "Refinement Types for Haskell". In: *ICFP.* ACM, 2014, pp. 269–282. DOI: 10.1145/2628136.2628161 (cit. on p. 58).

[Vog09] Werner Vogels. "Eventually consistent". In: *Communications of the ACM* 52.1 (2009), pp. 40–44 (cit. on p. 113).

[Wad+11] Hiroshi Wada, Alan D Fekete, Liang Zhao, Kevin Lee, and Anna Liu. "Data Consistency Properties and the Trade-offs in Commercial Cloud Storage: the Consumers' Perspective." In: *CIDR.* Vol. 11. 2011, pp. 134–143 (cit. on p. 113).

[ZWS23] George Zakhour, Pascal Weisenburger, and Guido Salvaneschi. "Type-Checking CRDT Convergence". In: *Proceedings of the ACM on Programming Languages* 7.PLDI (June 2023). DOI: 10.1145/3591276 (cit. on pp. 19, 20).

[ZBP14] Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. "Formal Specification and Verification of CRDTs". In: *FORTE.* Vol. 8461. Lecture Notes in Computer Science. Springer, 2014, pp. 33–48. DOI: 10.1007/978-3-662-43613-4_3 (cit. on pp. 19, 20, 22, 57, 65, 89).

[ZW10] Yuqing Zhu and Jianmin Wang. “Client-centric consistency formalization and verification for system with large-scale distributed data storage”. In: *Future Generation Computer Systems* 26.8 (2010), pp. 1180–1188 (cit. on p. 113).