Verifying Strong Eventual Consistency in δ-CRDTs

by

Taylor Blau

Supervised by Dan Grossman

A senior thesis submitted in partial fulfillment of

the requirements for the degree of

Bachelor of Science
With Departmental Honors

Computer Science & Engineering

University of Washington

June 2020

Presentation of work given on _____

Thesis and presentation approved by _____

Date _____

ABSTRACT

Conflict-free replicated data types (CRDTs) are a natural structure with which to communicate information about a shared computation in a distributed setting where coordination overhead may not be tolerated, and individual participants are allowed to temporarily diverge from the overall computation. Within this setting, there are two classical approaches: state- and operation-based CRDTs. The former define a commutative, associative, and idempotent *join* operation, and their states a *monotone join semi-lattice*. State-based CRDTs may be further distinguished into classical- and δ-state CRDTs. The former communicate their *full* state after each update, whereas the latter communicate only the *changed* state. Op-based CRDTs communicate *operations* (not state), thus making their updates non-idempotent. Whereas op-based CRDTs require little information to be exchanged, they demand relatively strong network guarantees (exactly-once message delivery), and state-based CRDTs suffer the opposite problem. Both satisfy *strong eventual consistency* (SEC).

We posit that δ-state CRDTs both (1) require less communication overhead from payload size, and (2) tolerate relatively weak network environments, making them an ideal candidate for real-world use of CRDTs. Our central intuition is a pair of reductions between state-, δ-state, and op-based CRDTs. We formalize this intuition in the Isabelle interactive theorem prover and show that state-based CRDTs achieve SEC. We present a relaxed network model in Isabelle and show that state-based CRDTs still maintain SEC. Finally, we extend our work to show that δ-state CRDTs maintain SEC when only communicating δ-state fragments, even under relatively weak network conditions.

ACKNOWLEDGEMENTS

This thesis is the product of many ideas grown out of collaboration and discussion with my advisory committee, as well as other researchers in this area.

First, Talia Ringer, my senior thesis mentor. Talia's thoughtfulness and willingness to absorb a new research area was inspiring and fostered me to look at this area from a new angle. Her patience in acquainting me with interactive theorem provers was key in making this thesis possible. Though always a source of good ideas, this thesis would not exist without Talia's unwavering support. I would be remiss if I did not mention Talia's encouragement throughout, even when the process was overwhelming.

Second, Dan Grossman, my faculty advisor. Dan has made my undergraduate experience meaningful in ways that I am not sure many others are as fortunate as I to have experienced. Dan took a skeptical pre-freshman, encouraged him to take CSE 341, and indulged him in many walks back to the Paul G. Allen building after class. Dan allowed me to T.A. for him, and was unflapped when I informed him that I had volunteered him to be my faculty advisor.[1] Of course, Dan is also a font of insight, offering new ideas and perspectives when they were needed, and always giving me something to think about after our meetings.

I would also like to thank Martin Kleppman, as well as his co-authors, for his constant correspondence throughout this work. Their work is foundational to our approach, and is the basis on which many of our ideas (and proofs) are built. Martin was always willing to discuss the state of our work, and to offer his guidance about interesting directions to pursue.

Finally, I wish to thank my family. My Mom and Dad, for their love, for always encouraging me, and for giving me the freedom to explore areas that interested me. Tracy and Richard Lippard, for their encouragement and hospitality during which significant portions of this thesis were written. Lastly, I wish to thank Maya Lippard, my partner, constant source of inspiration, and without whom this thesis would not exist.

[1] It could be said he was voluntold.

DEDICATION

To Maya, forever and ever.

CONTENTS

LIST OF FIGURES

1

INTRODUCTION

Computational systems today are larger than ever. Whereas previously one would architect their programs to run on a single system, it is now commonplace to design programs that share computation across multiple machines which communicate with each other in a coordinated fashion. Therefore, it is natural to ask why one might design from the latter perspective rather than the former. The answer is threefold:

1. *Resiliency*. Designing a computational workload to be distributed among participants tolerates the failure of any one (or more) of those participants.

2. *Scalability*. When designed from a distributed standpoint, "scaling" your workload to meet a higher demand is reduced to adding additional hardware, not designing more efficient ways to do the computation.

3. *Locality*. When a system is accessed from a broad set of geographic locations, strategic placement of hardware in locations near request-origin sites can lower latency for users.

So, it is clear that as our demand on such computations grow, that so too must our need to design these systems in a way that first considers the concerns of resiliency, scalability, and locality.

In order to design systems in this way, however, one must consider additionally the challenges imposed by not having access to shared memory among participants in the computation. If a program runs in a single-threaded fashion on a single computer, there is no need to coordinate memory accesses, since only one part of the program may read or write memory at a given time. If the program is written to be multithreaded, then the threads must coordinate among themselves by using mutexes or communication channels to avoid race conditions and other concurrency errors.

The same challenge exists when a system is distributed at the hardware and machine level, rather than among multiple threads running on a single piece of hardware. The challenge, however, is made more difficult by the fact that the communication overhead is far higher between separate pieces of hardware than between two threads.

This thesis focuses on datatypes by which computation can be coordinated across multiple machines. In particular, we formalize a set of consistency guarantees (namely, Strong Eventual Consistency, hereafter SEC) over a class of replicated datatypes, δ-state Conflict-Free Replicated Datatypes (CRDTs). We describe the preliminaries necessary to contextualize the body of this work in the following section.

1.1 PRELIMINARIES

Our discussion here focuses on CRDTs, which are designed to be both easily distributed and require relatively low coordination overhead by allowing individual participants to diverge temporarily from the state of the overall computation. That is, the computation reflects a different value depending on which participant in the computation responds to the request.

These datatypes operate in such a way so as to both avoid conflict between concurrent updates, and to avoid locking and coordination overhead [Shapiro et al., 2011]. CRDTs have seen moderate use in industry. Based on introspection of the runtime headers in iOS, Apple is believed to use CRDTs for offline synchronization of content in their note-taking app, Notes [Apple, Inc., 2018]. Redis, a popular open-source distributed cache uses CRDTs in their Enterprise offering to perform certain kinds of replication and conflict-resolution [Redis, Inc., 2020].

CRDTs are said to achieve SEC which is to say that they achieve a stronger form of *eventual consistency* (EC). We summarize the definitions of eventual- and strong eventual consistency from Shapiro et al. [2011].

Definition 1.1 (Eventual Consistency)**.** *A replicated datatype is* eventually consistent *if:*

- *Updates delivered to it are eventually delivered to all other replicas in the system.*
- *All well-behaved replicas that have received the same set of updates eventually reflect the same state.*
- *All executions on this datatype are terminating.*

Definition 1.2 (Strong Eventual Consistency)**.** *A replicated datatype is* strong eventually consistent *if:*

- *It is eventually consistent, as above.*
- *Convergence occurs immediately, that is, any two replicas that have received the same set of updates* always *reflect the same state.*

Broadly speaking, there are two classes of CRDTs, which we refer to as the op- and state-based variants. We will provide formal definitions for each of the two classes in Chapter 2. We now present brief definitions of op- and state-based CRDTs based on Baquero et al. [2014] and Shapiro et al. [2011]:

Definition 1.3 (Operation-based Conflict-Free Replicated Datatype (op-based CRDT))**.** *op-based CRDTs apply updates in two phases:*

1. *First, an operation is* prepared *locally. At this phase, the op-based CRDT combines the operation with the current state to send a representation of the update to other replicas.*

2. *Then, the represented operation is applied to other replicas using* effect, *where* effect *is commutative for concurrent operations.*

Definition 1.4 (State-based Conflict-Free Replicated Datatype (state-based CRDT))**.** *state-based CRDTs only apply updates to their local state, and periodically send serialized representations of the contents of their state to other replicas.*

Crucially, these states form a monotone join semi-lattice (i.e,. a lattice $\langle S, \sqcup \rangle$ *where for any* $s_1, s_2 \in S$ *at both* $s_1 \sqsubseteq s_1 \sqcup s_2$ *and* $s_2 \sqsubseteq s_1 \sqcup s_2$ *hold for commutative, associative, and idempotent* $\sqcup$*).*

To achieve convergence, state-based CRDTs periodically send their state to other replicas, which then replace their own state by joining the received state into their own.

1.2 OP- AND STATE-BASED TRADE-OFFS

These two classes are distinguished from one another based on their strengths and weaknesses. In one sense, op- and state-based CRDTs form a kind of a dual, where they trade off strong network guarantees for message payload size [Baquero et al., 2014].

Because the state-based CRDT needs to send a representation of its entire state, it often requires a significant amount of network bandwidth to propagate large messages [Almeida et al., 2018]. In Section 3.1 we will present an example where the payload size grows as a linear function of the number of replicas. In return for this large payload size, state-based CRDTs are able to achieve SEC even in networks that are allowed to drop, reorder, and duplicate messages.

On the other hand, op-based CRDTs require relatively little network bandwidth to send a notification of a single update (typically the representation generated in the *prepare* stage is dwarfed by the typical payload size of a state-based CRDT), but in exchange demand that the network deliver messages in-order for sequential (comparable) updates and at-most-once delivery [Shapiro et al., 2011].

Significant work in this area (Almeida et al. [2018], Cabrita and Preguiça [2017], Enes et al. [2018], van der Linde et al. [2016]) has focused on mediating these two extremes. This line of research (particularly in Almeida et al. [2018]) has identified δ-state CRDTs—a variant of the state-based CRDT which we discuss in Section 2.5—as an alternative which occupies a satisfying position between the two extremes. δ-state CRDTs behave as traditional state-based CRDTs, with the exception that their updates consist of state *fragments* instead of their entire state. These fragments (generated by δ-mutators and called δ-updates) are then applied locally at all other replicas to reassemble the full state. Because these fragments often do not need to comprise the full state, δ-state CRDTs in general have small payload size (thus requiring a similar amount of bandwidth as messages sent and received from op-based CRDTs), while still tolerating the same set of network deficiencies as state-based CRDTs. This combination of properties makes them an appealing alternative to traditional

state- and op-based CRDTs, and places interest in studying their convergence properties.

1.3 CONTRIBUTIONS

Our main contribution builds on the work in Gomes et al. [2017] and introduces a set of formally verified, machine-checked proofs in Isabelle [Nipkow et al., 2002] of the main result in Almeida et al. [2018], which we re-state below:[1]

Theorem 1.1 (Almedia, Shoker, Baquero, '18). *Consider a set of replicas of a δ-CRDT object, replica i evolving along a sequence of states $X_i^0 = \bot$, $X_i^1 = \ldots,$, each replica performing delta-mutations of the form $m_{i,k}^{\delta}(X_i^k)$ at some subset of its sequence of states, and evolving by joining the current state either with self-generated deltas or with delta-groups received from others. If each delta-mutation $m_{i,k}^{\delta}(X_i^k)$ produced at each replica is joined (directly or as part of a delta-group) at least once with every other replica, all replica states become equal.*

Here, X_i^t refers to the state of the ith replica at time t, and $m_{i,k}^{\delta}(X_i^k)$ refers to the δ-mutation applied at the ith replica at time k.

We rely on the work of Gomes et al. [2017] in order to build a handful of state- and δ-state CRDTs as in Almeida et al. [2018] to show that even under weak network guarantees[2] these δ-state CRDTs still achieve SEC.

Our verification efforts yielded a pair of CRDTs—the grow-only counter (G-Counter) and set (G-Set)—in three encodings: one state-based, and two δ-state encodings. Our key idea guiding these verification efforts is to treat op- and state-based CRDTs similarly by modeling state-based CRDTs as op-based where the operation is the join provided by the semi-lattice.[3] We show that SEC is preserved in these CRDTs, even when the underlying *network* interface has been weakened substantially from when it was introduced in the aforementioned work.

The remainder of this thesis is ordered as follows:

- In Chapter 2, we summarize existing research in the broader realm of CRDTs. We present formal definitions of op- and state-based CRDTs, and conduct a thorough discussion of their relative strengths and weaknesses. Likewise, we present a summary of some work in the area of δ-state CRDTs, and present its strengths.

- In Chapter 3, we discuss examples of two CRDTs in an op-, state-, and δ-state style. These objects will be the subject of our verification efforts in Chapter 5.

[1] The source of our proofs is available for free at: https://github.com/ttaylorr/thesis.

[2] We inherit dropping and reordering of messages from the original work of Gomes et al. [2017], but further relax the network model by also allowing messages to be duplicated.

[3] This approach is described in detail in Section 4.1.

- In Chapter 4, we outline a pair of reductions between state-, op-, and δ-state based CRDTs which guides the majority of our proof strategy.
- In Chapter 5, we discuss the outcome of our approach by presenting a pair of successfully-verified δ-state CRDTs, as well as describe our efforts in relaxing the network model in order to verify these objects over a non-trivial set of network behaviors.
- In Chapter 6, we suggest future research directions. We consider a handful of areas in which formalizing existing results may be fruitful, as well as a handful of additional approaches to the proofs we presented here.
- In Chapter 7, we conclude.

2

BACKGROUND

This chapter outlines the preliminary information necessary to contextualize the remainder of this thesis for readers unfamiliar with existing CRDT research. Here we motivate CRDTs, formalize their state- and op-based variants, and present examples of common instantiations. Finally, we conclude with a discussion of the different levels of consistency guarantees that each CRDT variant offers, and rationalize which levels of consistency are appealing in certain situations.

2.1 MOTIVATION

CRDTs are a way to store several copies of a data-structure on multiple computers which form a distributed system. Each participant in the system can make modifications to the datatype without the need for explicit coordination with other participants. CRDT implementations are designed so that coordination-free updates which may conflict with one another always have a deterministic resolution. This allows multiple participants to query and modify their *view* of the replicated datatype, without the traditional overhead and implementation burden that more stringent replication algorithms require.

Here, we'll discuss three variants of CRDTs: state-based, op-based, and δ-state based. Each of these variants achieve a consistent value by the use of different message types, and each likewise requires a different set of delivery semantics. In this chapter, we identify δ-state CRDTs as achieving an appealing set of trade-offs among each of the three variants. We restate that they are able to achieve SEC (the best reasonably-achievable consistency guarantee for most CRDT applications) while maintaining both:

- A relatively small payload size, as is the benefit of op-based CRDTs, and
- Relatively weak delivery semantics, as is the benefit of state-based CRDTs.

2.2 COORDINATED REPLICATION

In a distributed system, it is common for more than one participant to need to have a *view* of the same data. For example, multiple nodes may need to have access to the same internal data structures necessary to execute some computation. When a piece of data is shared among many participants in a system, we say that that data is *replicated*.

However, saying only that some data is "replicated" is underspecified. For example: how often is that data updated among multiple participants? How does that data behave when multiple participants are modifying it concurrently? Do all participants always have the same view of the data, or are there temporary divergences among the participants in the system?

It turns out that the answer to the last question is of paramount importance. Traditionally speaking, in a distributed system, all participants have an identical replica of any piece of shared data at all times. That is, at no moment in time will there be a replica that could atomically compare its replicated value for some data with any other replica for equality and disagree. Said otherwise, all replicated values are equal everywhere all at once. This is an appealing property to say the least, because it allows system designers to conceptually treat a distributed system as a single unit of computation. That is, if all replicas maintain the same memory, it is conceptually as if one whole machine is being replicated many times.

That being said, upholding this requirement is not a straightforward task. Some question that arise are: who coordinates when updates to a piece of data are replicated to other participants in the system? What happens when the coordinator becomes unresponsive, or otherwise misbehaves? Who is responsible for electing a new participant to take over the coordination duties of the participant which was no longer able to fulfill them?

2.3 DISTRIBUTED CONSENSUS ALGORITHMS

These questions give rise to the area of consensus algorithms. Broadly speaking, a consensus algorithm is a routine which multiple participants follow in order to agree on a shared value.

We first state briefly the properties that an algorithm must have to solve distributed consensus from Howard [2019]:

Definition 2.1 (Distributed Consensus Algorithm)**.** *An algorithm is said to solve distributed consensus if it has the following three safety requirements:*

1. Non-triviality*: The decided value must have been proposed by a participant.*
2. Safety*: Once a value has been decided, no other value will be decided.*
3. Safe learning*: If a participant learns a value, it must learn the decided value.*

In addition, it must satisfy the following two progress requirements:

1. Progress*: Under previously agreed-upon liveness conditions, if a value is proposed by a participant, then a value is eventually decided.*
2. Eventual learning*: Under the same conditions as above, if a value is decided, then that value must be eventually learned.*

The two most popular algorithms in this field are Paxos and Raft [Howard and Mortier, 2020, Lamport, 1998, Ongaro and Ousterhout, 2014]. Each implements distributed state-machine replication and can be used to implement linearizable systems. Both of these systems are notoriously difficult to understand and implement correctly in practice [Howard and Mortier, 2020]. The topics often appear in undergraduate-level courses in Distributed Systems, and have been the subject of extensive verification effort to date [Wilcox et al., 2015]. Often, these distributed systems verification efforts require an enormous amount of effort. In a companion paper Woos et al. [2016] use on the order of 45,000 lines of proof scripts to verify the complete Raft protocol in their system.

It is natural to ask what is the property of these systems that makes them difficult to implement or reason about correctly in practice. One possible answer is to look at the stringent safety requirements (that is, that once a value has been decided, no other value(s) will be decided) in these algorithms.

CRDTs are a natural response to this. By allowing participants to temporarily diverge from the state of the overall computation (cf., the second property of Definition 1.1), CRDTs allow replicas to violate the safety property of Definition 2.1. By giving up the immediacy and permanence that the safety properties of a traditional distributed consensus algorithm, CRDTs allow for a dramatically lower implementation burden in practice, and are substantially easier to reason about.

2.4 CONSISTENCY GUARANTEES

CRDTs are said to attain a weaker form of consistency known as *strong eventual consistency* [Shapiro et al., 2011]. SEC is a refinement of *eventual consistency* (EC). Informally, EC says that reads from a system eventually return the same value at all replicas, while SEC says that if any two nodes have received the same set of updates, they will be in the same state.

EC and the SEC extension are natural answers to the question we pose in Section 2.3. That is, we posit that it is the safety requirement in traditional Distributed Consensus Algorithms which make them difficult to implement correctly. EC makes only a liveness guarantee, and so on its own it is not a sufficient solution for handling distributed consensus in an environment with relaxed requirements. SEC, however, does add a safety guarantee, but the precondition (namely that only nodes which have received the same *set* of updates will be in the same state) makes it possible to relax our requirements around network delays, or particulars of a CRDT algorithm which do not send updates to all other replicas immediately.

In short, we believe that it is this relaxation–that is, that CRDTs are only required to be in the same state *eventually*, conditioned on which updates they have and have not yet received–which makes SEC an appealing consistency

property for distributed systems which more relaxed requirements than would be satisfied by a linearizable system.

We discuss each of these consistency classes in turn.

2.4.1 *Eventual Consistency*

EC captures the informal guarantee that if all clients stop submitting updates to the system, all replicas in the system eventually reach the same value [Shapiro et al., 2011]. More formally, EC requires the following three properties [Shapiro et al., 2011]:

1. *Eventual delivery*. An update delivered at some correct replica is eventually delivered at all replicas.

$$\forall r_1, r_2.\, f \in (\mathsf{delivered}\ r_1) \Rightarrow \Diamond f \in (\mathsf{delivered}\ r_2)$$

2. *Convergence*. Correct replicas which have received the same *set* of updates eventually reflect the same state.

$$\forall r_1, r_2.\ \Box\ (\mathsf{delivered}\ r_1) = (\mathsf{delivered}\ r_2) \Rightarrow \Diamond\ \Box\ q(r_1) = q(r_2)$$

3. *Termination*. All method executions terminate.

(For readers unfamiliar with modal logic notation, we use $\Diamond$ to precede a logical statement that is true at *some* time, whereas we use $\Box$ to precede a logical statement that is true at *all* times.)

EC is a relatively weak form of consistency. In Shapiro et al. [2011], it is observed that EC systems will sometimes execute an update immediately only to discover that it produces a conflict with some future update, and so frequent roll-backs may be performed. This imposes an additional constraint, which is that replicas need to form consensus on the "standard" way to resolve conflicts so that the same conflicts are resolved identically at different replicas.

We devote some additional discussion to the first property of EC. Eventual delivery requires that all updates delivered to some correct replica are eventually delivered to all other correct replicas. This property alone permits too much of the underlying network, and so it can make it difficult to reason about strong consistency guarantees over an unreliable network.

Take for an example a network which never delivers any messages. In this case, the precondition for eventual delivery is not met, and so we are relieved of the obligation to prove that updates are propagated to other replicas, since they aren't delivered anywhere in the first place. However, consider a network which delivers only the *first* message sent on it, and then drops all other messages. In this case, it *is* possible that a replica will receive some update, attempt to propagate it to other replicas, only for them to never be delivered.

To resolve this conflict in practice, one of two approaches is often taken. In the first approach, assume a fair-loss network [Cachin et al., 2011] in which each message has a non-zero probability of being delivered. To ensure that messages are delivered, each node sends each message an infinite number of times over the network, such that it will be delivered an infinite number of times.[1] This resolves the eventual delivery problem since we assumed a sufficient (but weaker) condition of the underlying network, and then showed it is possible to implement eventual delivery on top of these network semantics.

In the second approach, we first consider a set of delivery semantics P which predicates allowed and disallowed network behaviors. Typically, P is assumed to preserve causal order.[2] We then refine P to ensure that the properties of EC (and SEC) can be implemented on top of the network, resolving our problem by discarding degenerate network behaviors.

2.4.2 *Strong Eventual Consistency*

Another downside of implementing a system which only upholds EC is that EC is merely a liveness guarantee. In particular, EC does not impose any restriction on nodes which have received the same set or even sequence of messages. That is, a pair of replicas which have received the exact set of messages in the exact same order are not required to return the same value.

SEC addresses this gap by imposing a safety guarantee in addition to the previous liveness guarantees in EC. That is, a system is SEC when the following two conditions are met:

1. The system is EC, per above guidelines.
2. *Strong convergence*. Any pair of replicas which have received the same set of messages must return the same value when queried immediately.

$$\forall r_1, r_2. \, (\mathsf{delivered}\ r_1) = (\mathsf{delivered}\ r2) \Rightarrow q(r_1) = q(r_2)$$

That is, it is the strong convergence property of SEC that distinguishes it from EC. On top of EC, strong convergence is only a moderate safety restriction. In particular, it imposes no requirements on replicas which have not received the same sequence or even set of updates. So, unlike strong distributed consensus algorithms like Paxos or Raft which are fully linearizable [Lamport, 1998, Ongaro and Ousterhout, 2014], SEC allows certain replicas to be "behind." That is, a replica which hasn't yet received all relevant updates in the system is allowed to return an earlier version of the computation.

[1] This approach is due to Martin Kleppman over e-mail, but can also be found in the literature, for eg., Shapiro et al. [2011].

[2] This is a standard assumption [Gomes et al., 2017, Shapiro et al., 2011], and can be implemented by assigning a vector-clock and/or globally-unique identifier (UID) to each message at the network layer.

Informally, this means that replicas in the system are allowed to temporarily diverge from the state of the overall computation. As soon as no more updates are sent to the system, property (1) of EC requires that all replicas will *eventually* converge to a uniform view of the computation.

2.5 STATE-BASED CRDTS

Now that we have discussed EC and SEC, we will turn our attention to datatypes that implement these consistency models. CRDTs are a common way to implement the consistency requirements in SEC. So, we begin with a discussion of state-based CRDTs from their inception in Shapiro et al. [2011]. A state-based CRDT is a 5-tuple (S, s^0, q, u, m). An individual replica of a state-based CRDT is at some state $s^i \in S$ for $i \geq 0$, and is initially s^0. The value may be queried by any client or other replica by invoking q. It may be updated with u, which has a unique type per CRDT object. Finally, m merges the state of some other remote replica. Neither q nor u have pre-determined types, per se, rather they are implementation specific. We discuss a pair of examples to illustrate this point in Chapter 3.

Crucially, the states of a given state-based CRDT form a partially-ordered set $\langle S, \sqsubseteq \rangle$. This poset is used to form a join semi-lattice, where any finite subset of elements has a natural least upper-bound. Consider two elements $s^m, s^n \in S$. The least upper-bound $s = s^m \sqcup s^n$ is given as:

$$\forall s'.\ s' \sqsupseteq s^m, s^n \Rightarrow s^m \sqsubseteq s \wedge s^n \sqsubseteq s \wedge s \sqsubseteq s'$$

In other words, a $s = s^m \sqcup s^n$ is a least upper-bound of s^m and s^n if it is the smallest element that is at least as large as both s^m and s^n.

2.5.1 *Merging states*

For now, we set aside q and u, and turn our attention towards the merging function m. m resolves the states of two CRDTs into a new state, which is then assigned at the replica performing the merge. Given a suitable set of states which forms a lattice, we assume that:

$$m(s_1, s_2) = s_1 \sqcup s_2$$

for some join semi-lattice with join operation $\sqcup$, and that whenever a CRDT replica r_1 at state s_1 receives an update from another replica r_2 at state s_2, that r_1 attains a new state $s_1' = m(s_1, s_2)$. This process, in addition to each replica periodically broadcasting an update which contains its current state, is carried on continually, and m is invoked whenever a new state is received. That is, each replica is evolving over time in response to outside instruction, and in turn these updates cause internal state transitions, which themselves cause those new states to be broadcast and eventually joined at every other replica.

The $\sqcup$ operator has three mathematical properties that make it an appealing choice for joining states together as in m. These are its *commutativity*, *associativity*, and *idempotency*. That is, for any states s_1, s_2, and s_3, that:

- The operator is *commutative*, i.e., that $s_1 \sqcup s_2 = s_2 \sqcup s_1$, or that order does not matter.
- The operator is *idempotent*, i.e., that $(s_1 \sqcup s_2) \sqcup s_2 = s_1 \sqcup s_2$, or that repeated updates reach a fixed point.
- Finally, the operator is *associative*, i.e., that $s_1 \sqcup (s_2 \sqcup s_3) = (s_1 \sqcup s_2) \sqcup s_3$, or that grouping of arguments does not matter.

These mathematical properties correspond to real-world constraints that often arise naturally in the area of distributed systems. We provide examples for each of these three properties below:

COMMUTATIVITY Take, for example, that messages may occur out of order. This often happens in, for example, UDP (User Datagram Protocol) networks, where the received datagrams are not guaranteed to be in the order that they were sent. Because $\sqcup$ is commutative, replicas joining the updates of other replicas do not need to receive those updates in order, because the result of $s_1 \sqcup s_2$ is the same as $s_2 \sqcup s_1$. That is, it does not matter which of two updates from another replica arrives first, because the result is the same no matter in which order they are delivered.

For concreteness, say that we have two replicas, r_1 and r_2. r_1 initially begins at state s, and r_2 progresses through states $s_1, \ldots, s_n$ for $n > 0$. We then see that it does not matter the order in which these updates are delivered to r_1. Suppose that we have a bijection $\pi : [n] \to [n]$ which maps the true order of a state s_i to the order in which it was delivered. Then, we can see that the choice of π is arbitrary, because:

$$s \leftarrow s \sqcup \left(s_{\pi(1)} \sqcup \cdots \sqcup s_{\pi(n)}\right)$$

for any choice of π, because

$$s_{\pi(1)} \sqcup \cdots \sqcup s_{\pi(n)} = s_1 \sqcup \cdots \sqcup s_n$$

which follows from the fact that $\sqcup$ is commutative. This can be shown inductively on the number of updates, n, given the commutativity of $\sqcup$.

IDEMPOTENCY Next, it is often common for packets to be duplicated in transit over a network. That is, even though a packet may be sent from a source only once, it may be received by a recipient on the same network multiple times. For this, the idempotency of $\sqcup$ comes in handy: no matter how many times a state is broadcast from an evolving replica, any other replica on the network will tolerate that set of messages, because it only requires the message to be delivered once. Any additional duplicates are merged in without changing the state.

ASSOCIATIVITY Finally, associativity is an appealing property, too, although its applications are both less immediate and less often-used in this thesis. Suppose that several replicas of a state-based CRDT reside on a network with, say, high latency, or it is otherwise undesirable to send more messages on the network than is necessary. Because associativity implies that the grouping of updates is arbitrary, a replica can maintain a *set* of pending updates, and periodically send that set to other replicas by first folding $\sqcup$ over it and sending a single update.[3]

2.6 OP-BASED CRDTS

Operation-based (op-based) CRDTs evolve their internal states over time, but these states need not necessarily form a semi-lattice. Likewise, the communication style of op- and state-based CRDTs differ fundamentally: op-based CRDTs communicate *operations* that indicate a kind of update to be applied locally, instead of the *result* of that update (as is the case in state-based CRDTs).

An op-based CRDT is a 6-tuple (S, s^0, q, t, u, P). As in Section 2.5, S, s^0, and q, retain their original meaning (that is, the state set, an initial state, and a query function). In op-based CRDTs, the pair (t, u) takes the place of the m merging function from state-based CRDTs. t and u correspond to *prepare-update* and *effect-update*, respectively. When an update is made by a caller (say, for example, incrementing the value of an op-based CRDT counter), it is done in two phases [Shapiro et al., 2011]:

1. First, the *prepare-update* implementation t is applied at the replica receiving the update. t is side-effect free, and prepares a representation of the operation about to take place.

2. Then, the *effect-update* implementation u is applied at the local and remote replicas if and only if the delivery precondition P is met, causing the desired update to take effect. P is interpreted temporally [Shapiro et al., 2011], and is a precondition on whether or not operations necessary to process the *current* operation have already been incorporated into the CRDT's state. P is traditionally assumed to be disabled until all messages which happened before the current message have been delivered, preserving causality.

This is the critical distinction between op- and state-based CRDTS: state-based CRDTs propagate their state by applying a local update and taking advantage of the lattice structure of their state-space in order to define a convenient merge function. On the other hand, op-based CRDTs propagate

[3] "Periodically" is arbitrary and is left up to the implementation, but it would be easy to imagine that this could be interpreted as whenever the set reaches a certain size, and/or after a certain amount of time has passed since flushing the set of pending updates.

their state by sending the *representation* of an update to other replicas as an instruction. This critical juncture translates into a corresponding relaxation in the operation (t, u), which is that unlike the state-based CRDTs whose m must be commutative, associative, and idempotent, and op-based CRDT implementation of (t, u) need only be commutative.

To explain why, we briefly restate the definition of a causal history for op-based CRDTs:

Definition 2.2 (op-based Causal History [Shapiro et al., 2011]). *An object's casual history $C = \{c_1, \ldots, c_n\}$ is defined as follows. Initially, $c_i^0 = \emptyset$ for all $i \in \mathcal{I}$. If the kth method execution is idempotent (that is, it is either q or t), then the causal history remains unchanged in the kth step, i.e., that $c_i^k = c_i^{k-1}$. If the execution at k is non-idempotent (i.e., it is u), then $c_i^k = c_i^{k-1} \cup \{u_i^k(\cdot)\}$.*

Causal history of an op-based CRDT is defined based on the *happens-before* relation $\rightarrow$ as follows. An update (t, u) happens before (t', u') (i.e., that $(t, u) \rightarrow (t', u')$) iff $u \in c_j^{K_j(t')}$ if K_j is the injective mapping from operation to execution time. Shapiro and his co-authors go on to describe a sufficient definition for the commutativity of (t, u) in op-based CRDTs. In effect, they say that two pairs (t, u) and (t', u') commute if and only if for any reachable state $s \in S$ the effect of applying them in either order is the same. That is, $s \circ u \circ u' \equiv s \circ u' \circ u$.

They claim that having commutativity for concurrent operations as well as an in-order delivery relation P for comparable updates is sufficient to prove that op-based CRDTs achieve SEC.

2.7 δ-STATE CRDTS

In this section, we describe the refinement of CRDTs that is the interest and focus of the body of this thesis. That is the δ-state CRDT, as described in Almeida et al. [2018]. In their original work, Almeida and his co-authors describe δ-state CRDTs as:

> ...ship[ping] a *representation of the effect* of recent update operations on the state, rather than the whole state, while preserving the idempotent nature of *join*.

We will present an example of the δ-state CRDT in a below section. For now, we focus on the background material necessary to contextualize δ-state CRDTs. This refinement can be thought of as taking ideas from both state- and op-based CRDTs to mediate some of the trade-offs described above. Like a state-based CRDT, δ-state based CRDTs have both internal states and message payloads that form a join semi-lattice. This endows the δ-state CRDT with a commutative, associative, and idempotent *join* operator, as before. Likewise,

this means that the δ-state CRDT supports relaxed delivery semantics, such as delayed, dropped,[4] reordered, and duplicated message delivery.

Unlike a state-based CRDT, however, δ-state CRDTs do not send their internal state s^k after an update at time $k-1$. We require that these states have natural representations of their *updates* which do not require sending the full state to all other replicas. In many circumstances, these updates can often be represented as "smaller" items within the set of all possible reachable states. For example, in a CRDT which supports adding to a set of items, a δ-mutation may be the singleton set containing the newly-added item, whereas a traditional state-based CRDT may include the full set.

This means that:

- δ-state CRDTs support the same weak requirements from the network as ordinary state-based CRDTs. That is, they support dropping, duplicating, reordering, and delaying of messages.
- δ-state CRDTs have similarly low-overhead of message size as op-based CRDTs.

On the converse, δ-state CRDTs do not:

- ...have potentially large payload size, as state-based CRDTs are prone to have.
- ...require a strong delivery semantics P that ensures ordered, at-most-once delivery as op-based CRDTs do.

Said otherwise, δ-state CRDTs have the relative strengths of both state- and op-based CRDTs without their respective drawbacks. This makes them an area of interest, and they are the subject to which we dedicate the remainder of this thesis.

[4] In this thesis, we consider dropped messages as having been delayed for an infinite amount of time, allowing us to reason about a smaller set of delivery semantics.

3

ELEMENTARY CRDT INSTANTIATIONS

In this chapter, we provide the specification of two common CRDT instantiations in an op-, state-, and δ-state based style. We discuss the Grow-Only Counter (G-Counter) and Grow-Only Set (G-Set). Both of these will be the subject of our verification efforts in Chapter 5.

In each of the below, we assume that $\mathcal{I}$ refers to the set of node identifiers corresponding to the active replicas. In this thesis, we consider $\mathcal{I}$ to be fixed during execution; that is, we do not support addition or deletion of replicas. In practice, CRDTs do support a dynamic set of replicas, but we make this assumption for the simplicity of our formalism.

3.1 EXAMPLE: GROW-ONLY COUNTER

3.1.1 *State-based G-Counter*

The G-Counter supports two very simple operations: inc (increment), and query. When inc is invoked, the counter updates its internal state to increment the queried value by one. When query is invoked, the counter returns a number which represents the number of increment operations that have occurred globally in the system, for which the replica processing the query knows about. Note that this number is always at least as large as the number of times that inc has been invoked *at that replica*, and never larger than the true value of times inc has been invoked globally.

This is our first example of SEC, where replicas that are "behind," i.e., that have not received all updates from all other replicas, are not guaranteed to reflect the same value upon being queried.[1] Concretely, suppose that an inc has occurred at at least one other replica which has not yet broadcast its updated state. The replica being queried will have therefore not yet merged the updated state from the replica(s) receiving inc,[2] and so those update(s) will not be reflected in the value returned by querying.

We present a state-based G-Counter CRDT for concreteness, and then discuss its definition:

[1] Perhaps these messages were delayed or dropped in transit, or otherwise the other replicas have not broadcast their updates yet. The latter is uncommon in traditional state-based CRDTs, but is an often-used operation in variants of state-based CRDTs (including δ-state CRDTs) where updates are bundled into *intervals* which are sent in a way that preserves causality of updates.

[2] Because we cannot merge updates we do not know about.

$$\mathsf{G\text{-}Counter}_s = \begin{cases} S : \mathbb{N}_0^{|\mathcal{I}|} \\ s^0 : [0, 0, \cdots, 0] \\ q : \lambda s.\ \sum_{i \in \mathcal{I}} s(i) \\ u : \lambda s, i.\, s\,\{i \mapsto s(i) + 1\} \\ m : \lambda s_1, s_2.\ [\max\{s_1(i), s_2(i)\} : i \in \mathsf{dom}(s_1) \cup \mathsf{dom}(s_2)] \end{cases}$$

Figure 1: Specification of a state-based G-Counter CRDT.

Notice that the state space $\mathbb{N}_0^{|\mathcal{I}|}$ does not match the return type of the query function, q, which is simply $\mathbb{N}_0$. In Figures 1 and 2, we utilize a *vector* counter, which should be familiar to readers acquainted with *vector clocks* [Lamport, 1978].[3]

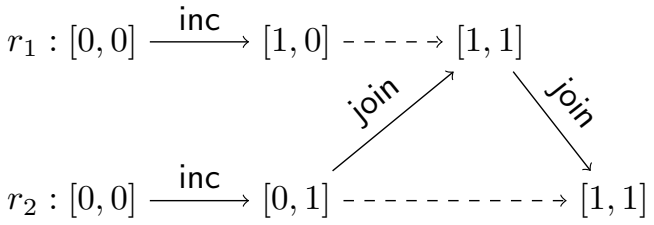

Figure 2: A correct execution of vector-based state G-Counters exchanging updates.

When an inc is invoked at the ith replica, it updates its own state to increment by one the vector element associated with the ith replica, here denoted $s\{i \mapsto s(i) + 1\}$. Finally, upon receiving an update from another replica, the pairwise maximum is taken on each of the vector elements. Note that this is a commutative, associative, and idempotent operation, and so it forms the least upper-bound of a lattice of vectors of natural numbers.

3.1.2 *op-based G-Counter*

In the op-based variant of the G-Counter, we can rely on a delivery semantics P which guarantees at-most-once message delivery.[4] From this, we say that replicas which are "behind" have not yet received the set of all inc operations performed at other replicas. Replicas which are "behind" may "catch up" when they receive the set of undelivered messages. However, these replicas never are "ahead" of any other replica, i.e., they never receive a message which doesn't

[3] Unlike traditional vector clocks, the vector *counter* only stores in each replica's slot the number of inc operations performed *at that replica*.

[4] That is, the network is allowed to drop, reorder, and delay messages, but a single message will never be delivered more than once.

correspond to a single inc operation at some other replica, thus they need not be idempotent.

We present now the full definition of the op-based G-Counter:

$$\mathsf{G\text{-}Counter}_o = \begin{cases} S : \mathbb{N}_0 \\ s^0 : 0 \\ q : \lambda s.\, s \\ t : \mathsf{inc} \\ u : \lambda s, p.\, s + 1 \end{cases}$$

Figure 3: Specification of an op-based G-Counter CRDT.

Because replicas are sometimes behind but never ahead, we know that the number of messages received at any given replica is no greater than the sum of the number of inc operations performed at other replicas, and the number of inc operations performed locally. So, the op-based G-Counter needs only to keep track of the number of inc operations it knows about globally, and this can be done using a single natural number. Hence, $S = \mathbb{N}_0$, and the bottom state is 0.

The query operation q is as straightforward as returning the current state. The *prepare-update* function t always produces the sentinel inc, indicating that an increment operation should be performed at the receiving replica. Finally, u takes a state and an arbitrary payload[5] and returns the successor.

Another approach to specifying the op-based G-Counter CRDT would be to more closely mirror the state-space of its state-based counterpart, as follows:

$$\mathsf{G\text{-}Counter}'_o = \begin{cases} S : \mathbb{N}_0^{|\mathcal{I}|} \\ s^0 : [0, 0, \cdots, 0] \\ q : \lambda s.\, \sum_{i \in \mathcal{I}} s(i) \\ t : (\mathsf{inc}, i)) \\ u : \lambda s, p.\, s\{i \mapsto s(i) + 1\} \end{cases}$$

Figure 4: Alternative specification of an op-based G-Counter CRDT.

where i represents the local node's identifier. Note that, while correct, restrictive delivery semantics P do not require such a verbose specification, since the at-most-once delivery guarantees allow us to simply increment our local count each time we receive an update, since no updates are duplicated over the network.

[5] Unused in the implementation here, since the only operation is inc.

3.1.3 *δ-state based G-Counter*

We conclude this subsection by turning our attention to the δ-state based G-Counter. We begin first by presenting its full definition:

$$\mathsf{G\text{-}Counter}_\delta = \begin{cases} S : \mathbb{N}_0^{|\mathcal{I}|} \\ s^0 : [0, 0, \cdots, 0] \\ q^\delta : \lambda s.\ \sum_{i \in \mathcal{I}} s(i) \\ u^\delta : \lambda s, i.\ \{i \mapsto s(i) + 1\} \\ m^\delta : \lambda s_1, s_2.\ \{\max\{s_1(i), s_2(i)\} : i \in \mathsf{dom}(s_1) \cup \mathsf{dom}(s_2)\} \end{cases}$$

Figure 5: Specification of a δ-state based G-Counter CRDT.

It is worth mentioning the extreme levels of similarity it shares with its state-based counterpart. Like the state-based G-Counter, the δ-state based G-Counter uses the state-space $\mathbb{N}_0^{|\mathcal{I}|}$, and has $s^0 = [0, 0, \cdots, 0]$. Its query operation and merge are defined identically.

However, unlike the state-based G-Counter, the δ-state based G-Counter implements the update function as $\lambda s, i.\{i \mapsto s(i) + 1\}$. That is, instead of returning the amended map (recall: $s\{\cdots\}$), the δ-state based G-Counter returns the *singleton map* containing *only* the updated index. Because of the definition of m (namely, that it does a pairwise maximum over the *union* of the domains of the two states), sending the singleton map is equivalent to sending the full map with all other entries being equal.

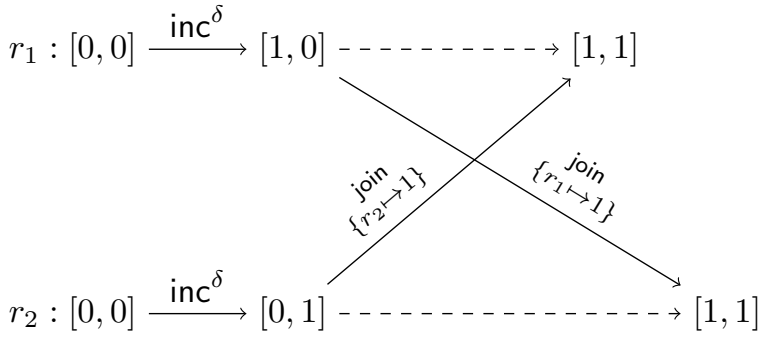

Figure 6: A pair of vector-based δ-state G-Counters replicas exchanging updates with each other.

This follows from the facts that: (1) the entry being updated has the same pairwise maximum independent of all other entries in the map, and (2) the pairwise maximum of all *other* entries does not depend on the updated entry. So,

taking the pairwise maximum of any state with the singleton map containing one updated value is equivalent to taking the pairwise maximum with our own state modulo one updated value. m is therefore referred to as a *δ-mutator*, and the value it returns is an *δ mutation* [Almeida et al., 2018].

This principle of sending *smaller* states (the δ mutations) which communicate only the *changed* information is a general principle which we will return to in the remaining example.

3.2 EXAMPLE: G-SET

The G-Set is the other primitive CRDT that we study in this thesis. In essence, the G-Set is a *monotonic set*. In other words, the G-Set supports the insertion and query operations, but does not support item removal. This is a natural consequence of the state needing to form a monotone semi-lattice, where set deletion would destroy the lattice structure.[6]

3.2.1 *State-based G-Set*

We begin our discussion with the state-based G-Set CRDT, the definition of which we present below. This is our first example of a *parametric* CRDT instance, where the type of the CRDT is defined in terms of the underlying set of items that it supports.

$$\mathsf{G\text{-}Set}_s(\mathcal{X}) = \begin{cases} S : \mathcal{P}(\mathcal{X}) \\ s^0 : \{\} \\ q : \lambda x.\, x \in s \\ u : \lambda x.\, s \cup \{x\} \\ m : \lambda s_1, s_2.\, s_1 \cup s_2 \end{cases}$$

Figure 7: state-based G-Set CRDT

For some set $\mathcal{X}$, we can consider the state-based G-Set CRDT instantiated over it, $\mathsf{G\text{-}Set}_s(\mathcal{X})$. The state-space of this CRDT is the power set of $\mathcal{X}$, which we denote $\mathcal{P}(\mathcal{X})$. Initially, the G-Set begins as the empty set, here denoted $\{\}$. The three operations are defined as follows:

- The query function q is an unary relation, i.e., it determines which elements are contained in the G-Set.

[6] To support removal from a CRDT-backed set, the 2P-Set is often used. Verifying this object is left to future work, which we discuss in Section 6.1.

- The update function u produces the updated set formed by taking the union of the existing set, and the singleton set containing the item to-be-added.
- Finally, the merge function m takes the union of two sets.

Note crucially that the merge function $\cup$ defines the least upper-bound of two sets, and thus endows our CRDT with a lattice structure. In this lattice of sets, we say that for some set $\mathcal{X}$, the lattice formed is $\langle \mathcal{P}(\mathcal{X}), \subseteq \rangle$.

3.2.2 *op-based G-Set*

In the op-based variant of the G-Set CRDT, we replace the state-based CRDT's update function u with the op-based pair (t, u). The state space, initial state, as well as the query and merge functions (q and m, respectively) are defined identically. We present the full definition as follows:

$$\mathsf{G\text{-}Set}_o(\mathcal{X}) = \begin{cases} S : \mathcal{P}(\mathcal{X}) \\ s^0 : \{\} \\ q : \lambda x.\, x \in s \\ t : \lambda x.\, (\mathsf{ins}, x) \\ u : \lambda p.\, s \cup \{(\mathsf{snd}\ p)\} \\ m : \lambda s_1, s_2.\, s_1 \cup s_2 \end{cases}$$

Figure 8: op-based G-Set CRDT

The only difference between this CRDT instantiation and the state-based one is in the definition of (t, u).[7] In the state-based CRDT, we sent the updated state, i.e., $s \cup \{x\}$. In the op-based variant, we send a *representation* of the effect, which we take to be the pair (ins, x), where ins is a sentinel marker indicating that the second element in the pair should be inserted.

Upon receipt of the message (ins, x), our op-based G-Set CRDT computes the new state $s \cup \{(\mathsf{snd}\ p)\}$, where p is the message payload.

3.2.3 *δ-state based G-Set*

Finally, we turn our attention to the δ-state based G-Set CRDT. As was the case with the δ-state based G-Counter CRDT, this object is defined identically as to the state-based counter, with the notable exception of its update function, u.[8]

[7] This is a pattern that will become familiar during Chapter 5.
[8] This again will be another familiar pattern in Chapter 5.

For full formality, we present its definition below:

$$\mathsf{G\text{-}Set}_\delta(\mathcal{X}) = \begin{cases} S : \mathcal{P}(\mathcal{X}) \\ s^0 : \{\} \\ q^\delta : \lambda x.\, x \in s \\ u^\delta : \lambda x.\, \{x\} \\ m^\delta : \lambda s_1, s_2.\, s_1 \cup s_2 \end{cases}$$

Figure 9: δ-state based G-Set CRDT

Here, the only difference is between the state- and δ-state based CRDT's definition of the update method, u. In the state-based G-Set, update was defined as $u : \lambda x.\, s \cup \{x\}$. But in the δ-state based G-Set, the update is defined as $u : \lambda x.\, \{x\}$. Note crucially that these two kinds of updates are equal when applied to the same local state. Consider a state- and δ-state based G-Set, both starting at the same state s^t. For the state-based G-Set, we have:

$$\begin{aligned} m(s^t, u(x)) &= s^t \cup (s^t \cup \{x\}) \\ &= (s^t \cup s^t) \cup \{x\} \\ &= s^t \cup \{x\} \end{aligned}$$

whereas for the δ-based G-Set, we have directly:

$$m^\delta(s^t, u^\delta(x)) = s^t \cup \{x\}$$

4

CRDT REDUCTIONS

This chapter outlines the key component of our proof strategy. We begin with a reduction allowing us to convert from state- to op-based CRDTs. This reduction is used in Chapter 5 to show a preliminary encoding of two state-based CRDTs. We conclude with a reduction from δ-state to op-based CRDTs, which is used extensively in the latter part of Chapter 5 to show that δ-state CRDTs achieve SEC.

Specifically, we will discuss the following:

- In Section 4.1, we will describe a mapping $\phi_{\text{state}\to\text{op}}$ to reduce state-based CRDTs to op-based CRDTs.
- In Section 4.2, we will describe a mapping $\phi_{\delta\to\text{op}}$ to reduce δ-state CRDTs to op-based CRDTs.

We state these reductions as "maxims". They are stated here in brief, but we will return to them in Sections 4.1 and 4.2.

Maxim 4.1. *A state-based CRDT is an op-based CRDT where the* prepare-update *phase returns the updated state, and the* effect-update *is a join of two states.*

Maxim 4.2. *A δ-state based CRDT is an op-based CRDT whose messages are δ-fragments, and whose operation is a pseudo-join between the current state and the δ fragment.*

4.1 STATE-BASED CRDTS AS OP-BASED

This section describes a reduction from state-based CRDTs to op-based CRDTs. We describe this reduction to exemplify how to reduce between CRDT classes, and use this in Chapter 5 to show that two state-based CRDTs achieve SEC.

Consider some state-based CRDT $C = (S, s^0, q, u, m)$. This object C has a set of states S, an initial state s^0, along with functions for querying the state (q), updating its state (u), and merging its state with the state of some other object (m). Our question is to define a mapping ϕ as follows:

$$\phi_{\text{state}\to\text{op}} : \underbrace{(S, s^0, q, u, m)}_{\text{state-based CRDTs}} \longrightarrow \underbrace{(S, s^0, q, t, u, P)}_{\text{op-based CRDTs}}$$

For our purposes, we view $\phi_{\text{state}\to\text{op}}$ as a homomorphism between state- and op-based CRDTs.

Note that P (the delivery precondition on the right-hand side) is the only element which does not have a natural analog on the left-hand side. Traditionally it is common to have a P which preserves causality, but this is not necessary for our proofs (since we map states identically as in the following section). Therefore, we assume that P is always met, in which case delivery can always occur immediately on the right-hand side.

We'll now turn to describing the details of $\phi_{\text{state}\to\text{op}}$, which for convenience in this section, we'll abbreviate as simply ϕ.[1] To understand ϕ, we'll consider how it maps the state S (along with s^0 and q) separately from how it maps the update procedure u.

4.1.1 *Mapping states under ϕ*

Let us begin our discussion with a consideration to how ϕ maps the state S from a state-based CRDT to an op-based one. In practice, it would be unrealistic to treat the state space of a state-based CRDT as equal to that of its op-based counterpart. Doing so would discard one of the key benefits of op-based CRDTs over state-based ones, which is that they are often able to represent the same set of query-able states using simpler structures. For example, state-based counters (such as the G-Counter_s and PN-Counter_s) often use a vector representation to represent the number of "increment" operations at each node, but op-based counters often instead use scalars (cf., the examples in Section 3.1).

In order to make ϕ a simple reduction, we allow the state spaces of the CRDT before and after the reduction to be identical. Though CRDT designers can often be more clever than this in practice, this makes reasoning about the transformation much simpler for the purposes of our proofs. Likewise, since the query function q is defined in terms of the state-space, S, we let ϕ preserve the implementation of q under the mapping, too.

4.1.2 *Mapping updates under ϕ*

Now that we have described the process by which ϕ maps S, s^0, and q, we still need to address the implementation of u and m under mapping. Our guiding principle is the following theorem (which we state and discuss here, but have not mechanized):

[1] In the following section, we'll define a new homomorphism between op- and δ-state based CRDTs, at which point we will distinguish between the two mappings when it is unclear which is being referred to.

Theorem 4.1. *Let C_s be a state-based CRDT with $C_s = (S, s^0, q, u, m)$. Define an op-based CRDT C_o as follows:*

$$C_0 = \begin{cases} S_o : S \\ s_o^0 : s^0 \\ q_o : q \\ t_o : \lambda p.\, u(p...) \\ u_o : \lambda s_2.\, m(s^t, s_2) \end{cases}$$

then, C_o and C_s reach equivalent states given equivalent updates and delivery semantics.

Proof sketch. By simulation. Since $s_o^0 = s^0$, both objects begin in the same state. Since $q_o = q$, if the state of C_o and C_s are equal, then q_o will reflect as much. Finally, an update is prepared locally by computing the updated state-based representation. That update is applied both locally and at all replicas by merging the prepared state into C_o's own state, preserving the equality. □

In other words, we decompose the *update* function of a state-based CRDT into the *prepare-update* and *effect-update* functions of an op-based CRDT. Let p be the set of parameters used to invoke the update function u of a state-based CRDT, i.e., that $u(p...)$ produces the desired updated state. Then the *prepare-update* returns a serialized representation of $u(p...)$, which is to say that it returns the updated state. The *effect-update* implementation then takes that representation and applies it by invoking the merge function m with the effect representation and its own state to produce the new state.

This introduces Maxim 4.1, which unifies state- and op-based CRDTs as behaving identically when the op-based CRDT performs a join of two states. We restate this Maxim for clarity:

Maxim 4.1. *A state-based CRDT is an op-based CRDT where the* prepare-update *phase returns the updated state, and the* effect-update *is a join of two states.*

4.2 δ-STATE BASED CRDTS AS OP-BASED

In the previous section, we described a general procedure for converting state-based CRDTs into op-based CRDTs. In this section, we treat the insight from the previous section as guidance for how to design a similar reduction to convert δ-state CRDTs into op-based CRDTs. We will use this reduction to encode δ-state CRDTs into the library presented in Gomes et al. [2017] in order to verify that δ-state CRDTs are SEC.

Similarly as in the previous section, we describe a (new) mapping $\phi_{\delta \to \text{op}}$ of type:

$$\phi_{\delta \to \text{op}} : \underbrace{(S, s^0, q, u^\delta, m^\delta)}_{\delta\text{-based CRDTs}} \longrightarrow \underbrace{(S, s^0, q, t, u, P)}_{\text{op-based CRDTs}}$$

For the same reasons as in Section 4.1.2, we let ϕ preserve the state space, initial state, and query function. Again, we let P be the delivery precondition which is always met (since messages exchanged are idempotent, and so there is no need to preserve either causality or at-most-once delivery as is traditional).

In Section 4.1, we treated a state-based CRDT's state as the representation of the effect for an operation-based CRDT. In this section, we do the same for the *δ-state fragment*, which we naturally think as a difference of two states.

Concretely, let $t : S \to S \to T$ for some type T not necessarily equal to S which represents the type of all δ-fragments. We define two examples as follows:

- For the G-Set CRDT, the δ mutator, m^δ produces the singleton set containing the element added in the last operation. Since only one item can be added at a time, computing the following with the before- and after-states is sufficient to generate the representation:

  $$t = \lambda s_1, s_2.\, s_2 \setminus s_1$$

  where $S = T = \mathcal{P}(\mathcal{X})$. This is an example where the CRDT has a type where both the state- and δ-state fragments are members of S.

- For the G-Counter CRDT, the δ mutator produces a pair type containing the identifier of a node with a changed value, and the new value which is assigned to that identifier. t is defined as:

  $$t = \lambda s_1, s_2. \quad \min_{\substack{i \in \mathcal{I} \\ s_1[i] \neq s_2[i]}} \quad (i, s_2[i])$$

  (Observe that this function is not defined for two states $s_1 = s_2$, nor does it need to be, since the before- and after states are guaranteed to be different after invoking u^δ).

  Here we have an example of $T \neq S$, where T instead equals $'id \times \mathbb{N}$.

t is now capable of generating the δ-state fragment corresponding to any pair of states from before and after and invocation of u^δ. Now we need to define the op-based CRDT's implementation of u to recover a new state given a value of type T. Here, let $u : S \to T \to S$, which takes in a current state as well as a δ-fragment and produces a new state.

Intuitively, u is a sort of inverse over the last argument and return value of t. That is, where t was taking the difference of two states, u recovers that difference into a new state. We define two example implementations of u as follows:

- For the G-Set CRDT, the new state is recovered by taking the union of the current state, along with the state carrying the new item. That is:

  $$u = \lambda s, t.\, s \cup t$$

- For the G-Counter CRDT, the new state is recovered by taking the old state, and replacing the entry whose index is equal to the first part of an update with the value described by the second part of that update.

Importantly, u and t needs to satisfy three important properties:

1. u and t can never work together to produce a state which is not by either the current state, or the δ-fragment. That is, for any state s', we must have that:
$$\forall s \sqsubseteq s'.\, u(s, t(s, s')) \sqsubseteq s'$$
Or in other words, if our starting state is lower in the lattice than s', taking the δ-fragment between s and s' and then re-applying that to s cannot produce a new state which is $\sqsupseteq s'$.
2. At all times, all replicas must reflect all updates performed at that replica.
3. All replicas which have received the same set of messages have the same state.

Together, these properties are sufficient to re-introduce Maxim 4.2, which we restate here for clarity:

Maxim 4.2. *A δ-state based CRDT is an op-based CRDT whose messages are δ-fragments, and whose operation is a pseudo-join between the current state, and the δ fragment.*

Therefore, we have a straightforward procedure for reasoning about δ-state CRDTs in terms of op-based CRDTs, which is to convert any δ-state CRDT into an op-based CRDT, and then use the existing framework of Gomes et al. [2017] to mechanize that that CRDT achieves SEC.

5

EXAMPLE CRDTS UNDER RELAXED NETWORK MODEL

We have mechanized proofs that two state- and δ-state based CRDTs achieve SEC. We relax the underlying network model to support non-unique messages (Section 5.1), and then showed that both the state- and δ-state based G-Counter and G-Set inhabit SEC (Sections 5.2 and 5.3). Finally, we present an alternative encoding of the reduction in Chapter 4 for δ-state CRDTs (Section 5.4).

5.1 NETWORK RELAXATIONS

In Gomes et al. [2017], Gomes and his co-authors provided a network model which makes the following set of assumptions:

1. All messages received by some node were broadcast by some other node.
2. All messages broadcast by some node were received by that node (i.e., all messages are delivered locally in a reliable fashion).
3. All messages are unique.

These assumptions allow the network to drop, reorder, and delay messages in transit.

Because op-based CRDTs only deliver updates once, it is traditional to assume a delivery relation P which predicates the set of network executions that we are allowed to reason about. For example, a network execution which drops all messages in transit, or does not preserve causality cannot be shown to exhibit SEC, and so it is not a member of the relation P. Such an assumption is standard in the literature and goes back to the original work in Shapiro et al. [2011].

In Gomes et al. [2017], the authors make extensive use of Isabelle's *locale* feature [Nipkow et al., 2002], which for our purposes we can consider as Isabelle's implementation of parametric proofs. Specifically, Gomes et al. [2017] define a locale for SEC, which they call *strong-eventual-consistency*. To instantiate this locale, CRDT replicas must meet the following preconditions:

- Messages which have a causal dependence are delivered in-order; concurrent messages may be delivered in any order (i.e., the $\prec$ relation is preserved during delivery).
- The set of messages delivered at each node is distinct.[1]

[1] Note that the messages transited by the network may be non-distinct. This is another standard assumption which can be implemented by tagging each message with a vector clock or assigning a globally unique identifier, and having each receiving node discard duplicates.

- That concurrent operations commute.
- That correct nodes do not fail, i.e., that they remain responsive during the execution.

While we consider the above to be a reasonable delivery semantics, we wish to relax the network model in order to support duplicated messages. This behavior is not permitted by the original network model in Gomes et al. [2017], which assumes that each message in transit on the network has a unique identifier.

To see this, consider the following example:

Example 5.1. *Consider two systems which have multiple replicas of CRDT counters. System A uses op-based counters, and system B uses state-based counters. Consider two replicas in each system, call these r_1 and r_2. Suppose the following happens in each system:*

- *A inc operation is performed at replica r_1, which causes a message to be sent to all other replicas. In system A, this message is $[1, 0, \cdots, 0]$, and in system B this messages is inc.*
- *While in route to replica r_2, this message is duplicated, and both copies are received at replica r_2.*

Notice that $q(r_2)$ results in a different value based on whether or not you queried the replica belonging to system A or system B. In system A, the duplicate message is "ignored," since merging the same message twice is idempotent due to $\sqcup$, and $q(r_2) = 1$ as expected. In system B, the additional update is *applied, meaning that $q(r_2) = 2$, which is a safety violation.*

So, while it is often a safety violation for an op-based CRDT to receive the same message twice,[2] state- and δ-state based CRDTs can and should tolerate this class of degenerate behaviors.

The general principle is as follows:

Theorem 5.1. *State-based CRDTs exhibit SEC even when operating in a network environment permitting non-unique messages.*

Proof. By induction on the number of times i a message m' is received. When $i = 1$, the goal is trivially established. When $i > 1$, the idempotency of $\sqcup$ shows that:

$$m \sqcup \underbrace{m' \sqcup \cdots \sqcup m'}_{i-1 \text{ times}} \sqcup m' = m \sqcup m' \sqcup m' = m \sqcup m'$$

where the second equality follows from the inductive hypothesis, and the third from the fact that $m' \sqcup m' = m'$ by the idempotency of $\sqcup$. □

[2] This is the primary reason why it is a standard assumption of op-based network models to disallow non-unique messages

This result guides our approach as follows: to show a stronger result that uses Theorem 5.1 (i.e., that state- and δ-state based CRDTs achieve SEC no matter how many times), the network model originally presented in Gomes et al. [2017] should be extended to remove the assumption that message identifiers are unique.

5.1.1 *Delivery Semantics*

In their original network model, the authors of Gomes et al. [2017] use an Isabelle *locale* in order to parameterize varying instantiations of the network based on certain assumptions. They provide the following definition for the Network locale [Gomes et al., 2017]:

locale *network* = *node-histories history*
  for *history* :: *nat* ⇒ *'msg event list* +
  fixes *msg-id* :: *'msg* ⇒ *'msgid*
  assumes *delivery-has-a-cause*:
    ⟦ *Deliver m* ∈ *set* (*history i*) ⟧ ⟹ ∃ *j*. *Broadcast m* ∈ *set* (*history j*)
  and *deliver-locally*: ⟦ *Broadcast m* ∈ *set* (*history i*) ⟧ ⟹ *Broadcast m* $\sqsubset^{i}$ *Deliver m*
  and *msg-id-unique*: ⟦ *Broadcast m1* ∈ *set* (*history i*);
    Broadcast m2 ∈ *set* (*history j*);
    msg-id m1 = *msg-id m2* ⟧ ⟹ *i* = *j* ∧ *m1* = *m2*

Figure 10: Isabelle specification of the Network locale as given in Gomes et al. [2017].

In order to extend the network model of Gomes et. al. to support duplicated messages, we need to remove the assumption *msg-id-unique*, which allows the enclosed proofs to assume that messages have unique identifiers. While this assumption is part of the locale, proofs are allowed to assume that if two messages m_1 and m_2 with the same identifier (i.e., that $msg\text{-}id m_1 = msg\text{-}id m_2$) exists in the history of two nodes, that either the two nodes or two messages are identical.

Although our proofs are still instantiated after fulfilling the qualifier P, we still wish to reason about an expanded set of network executions which includes message dropping.[3]

For our purposes, we begin by specifying a relaxed *network* locale as follows:

[3] Since op-based CRDTs require causality-preserving semantics P, we cannot remove the dependence on P without substantial alternation to the library. We leave this to future work, and discuss it in greater detail in Chapter 6.

locale *network* = *node-histories history*
  for *history* :: *nat* ⇒ '*msg event list* +
  fixes *msg-id* :: '*msg* ⇒ '*msgid*
  assumes *delivery-has-a-cause*:
    ⟦ *Deliver m* ∈ *set* (*history i*) ⟧ ⟹ ∃ *j*. *Broadcast m* ∈ *set* (*history j*)
  and *deliver-locally*: ⟦ *Broadcast m* ∈ *set* (*history i*) ⟧ ⟹ *Broadcast m* ⊏^{i} *Deliver m*

Removing this assumption immediately invalidates many of the proofs contained within the *network* locale. These proofs are broken due to a variety of reasons, ranging from something as simple as referencing a now-missing assumption, to more complex issues, e.g., a proof which relies on the uniqueness of delivered messages.

We now describe our strategy for repairing these proofs:

1. First, remove the assumption *msg-id-unique* from the *network-with-ops* locale, as above.
2. Identify the set of broken proofs. In each broken proof, do the following:
   (a) Identify the earliest broken proof step.
   (b) Delete it and all proof steps following it.
   (c) Replace the proof body with the term **sorry**.
3. In any order, consider a proof which ends with **sorry**, and repair the proof.

In total, there were four (4) key lemmas which needed repair. These were: *hb-antisym*, *hb-has-a-reason*, *hb-cross-node-delivery*, and *hb-broadcast-broadcast-order*. After removing the *msg-id-unique* assumption, each of the above four proofs were able to be repaired automatically by Isabelle's proof search procedure **sledgehammer** [Nipkow et al., 2002].

In each of the CRDTs that we do verify, we are required to instantiate a lemma stating:

$$\textit{apply-operations xs} = \textit{apply-operations ys}$$

where *xs* and *ys* are lists of messages delivered to a pair of replicas by the network. In other words, no matter what messages are delivered in what order, the two replicas attain the same state. Following the original proofs provided for op-based CRDTs in Gomes et al. [2017], our proofs of this lemma make the standard assumption that:

$$\textbf{set}\ (\textit{node-deliver-messages xs}) = \textbf{set}\ (\textit{node-deliver-messages ys})$$

Note that although we require that the set of operations delivered at two nodes is identical in order for those two nodes to attain the same value, we are able to reason over an expanded set of network behaviors. For example, if some

message *m* appears in either of the two sets above, we know that it only appears in that node's history once, by the *msg-id-unique* assumption. But without that assumption, we know instead that it appears *at least* once in each of the node's log of history.

This is a key distinction, since not knowing how many times a message was delivered to either of the two replicas means that we are able to conclude that they reach the same state if the same set of messages is delivered *at least once* to each of the replicas. Said otherwise, it does not matter how many times a message was delivered at each of two replicas, so long as it was delivered at least once at both. This allows us to exercise the latter case of Example 5.1 using the relaxed network model.

5.2 STATE-BASED CRDTS

Equipped with a relaxed network model, we are now ready to verify two examples of state-based CRDTs.

5.2.1 *State-based G-Counter*

We begin first with the G-Counter, the formal definition of which can be found in Section 3.1. Following our intuition in Maxim 4.1, we define a type to represent the *state* and *operation* of a state-based G-Counter, presented below:

type-synonym $('id)\ state = 'id \Rightarrow int\ option$
type-synonym $('id)\ operation = 'id\ state$

Figure 11: Isabelle definitions for *state* and *operation* for a state-based G-Counter CRDT.

Here, we let the state be a partial mapping from a transparent *'id* type (the value of which uniquely identifies a replica in the system) to an *int* which specifies the number of increment operations performed at that replica. Like in Section 3.1, this defines a vector-like object, where each slot in the vector corresponds to the number of increment operations performed at some unique replica in the system. We define this mapping to be partial, where the *None* value signals that no increments have been performed at a given node.[4]

Next we define the operation to be a type-level synonym for the *'id state* type. This encodes that operations *are* states. We interpret that upon receipt of an operation that we replace our current state with the join of it and the state encoded by the operation, which is an implementation of Maxim 4.1.

Before introducing the interpretation of *gcounter-op* (which will be responsible for performing this join operation as described), we look at a few other functions which are defined to act over this type:

[4] This choice is arbitrary, and could have easily have been implemented as mapping to 0 instead.

```
fun option-max :: int option ⇒ int option ⇒ int option  where
option-max (Some a) (Some b) = Some (max a b) |
option-max x None = x |
option-max None y = y

fun inc :: 'id ⇒ ('id state) ⇒ ('id operation) where
inc who st = (case (st who) of
   None ⇒ st(who := Some 0)
 | Some c ⇒ st(who := Some (c + 1)))
```

Figure 12: Isabelle definitions for state-based G-Counter-related functions.

The function *inc* specifies (for demonstration purposes) how to increment the value in a vector for some node. That is, *inc* specifies the procedure to execute when an increment operation is performed at some replica. Since our proofs reason purely about transitions of states, and not the external forces that drive them, this function is never called by our proofs, but merely left for the reader as a demonstration of how to drive the system.

The other function *option-max* specifies the pair-wise maximum of two *int option* values. Note that these are the right-hand side of the mapping in *'id state*, and so this function is used to merge the state received from some other replica. We will prove some additional facts about this function shortly, but for now we interpret it as taking the maximum of two optional integers, where a present integer is always preferred over an absent one,[5] and the maximum of two absent integers is *None*.

Now that we have a way to interpret the pair-wise maximum of two states which constitute a join, we can specify our definition of the "operation" for a state-based G-Counter CRDT. Recall that as in Maxim 4.1, we need to specify an operation which is the join of two states. We present now the definition as used in our proofs:

```
fun gcounter-op :: ('id operation) ⇒ ('id state) ⇀ ('id state) where
gcounter-op theirs ours = Some (λ x. option-max (theirs x) (ours x))
```

Figure 13: Isabelle definition for the "operation" of a state-based G-Counter CRDT.

Here, we specify a function that produces a partial mapping from an operation and state to a new state. The function is not total (that is, it *can* return *None* for some input) to indicate a crash. For our purposes, we do not specify such a case, and so the function always returns *Some* for any input.[6] Here, the state on the left-hand side indicates the state that our replica currently has. The

[5] That is, the maximum of *Some x* and *None* is *Some x*.

[6] Note that this *None* is different from the partial mapping of the *'id state* type, which specifies that the *count* of increment operations at some replicas may zero. Returning *None* from *gcounter-op* indicates that there is no state at all, i.e., a crash has occurred.

"operation" so-to-speak is the state at some *other* replica. By encoding the state from a remote replica in this fashion, we are implicitly saying that this state should be joined with our current state, and that the result of this join should replace our current state. So, we return a new state, which is a function which maps node identifiers to the maximum of the associated value between our previous state, and the state at some other replica.

For example, if our state in a four-replica system is:

$$\{r_1 : 1, r_2 : \bot, r_3 : 2, r_4 : \bot\}$$

and the state of some replica is:

$$\{r_1 : \bot, r_2 : 1, r_3 : 3, r_4 : \bot\}$$

the resulting state is:

$$\{r_1 : 1, r_2 : 1, r_3 : 3, r_4 : \bot\}$$

In Isabelle, we encode this as a function which forms a closure over the local and remote states, and then computes the maximum some given node identifier x. In practice, this is the lazy equivalent to computing all of the values up front upon merging.

Now that we have an instantiation of how to modify and merge states (the equivalent of the u and m), it remains to show that this is a suitable instantiation of the *strong-eventual-consistency* locale.[7]

A first-try instantiation shows that it is not possible to do so without additional proofs. Upon inspecting the unmet goals, we can deduce that Isabelle wants a proof of the commutativity and associativity of *option-max*, the key function used to implement the merge of two states. We leave the full definitions of these proofs to Section A.1; most are able to be completed with induction and term simplification only, and so are not of great interest to this section.

Once we have a proof of commutativity and associativity (Isabelle can infer that *option-max* is idempotent automatically), we then state an important lemma and corollary, which are as follows:

lemma (**in** *gcounter*) *concurrent-operations-commute*:
assumes *xs prefix of i*
shows *hb.concurrent-ops-commute* (*node-deliver-messages xs*)
using *assms*
apply(*clarsimp simp*: *hb.concurrent-ops-commute-def*)
apply(*unfold interp-msg-def*, *simp*)
done

Figure 14: Isabelle proofs that concurrent operations commute in the state-based G-Counter.

[7] Recall that instating this locale is equivalent to a proof that the object it is being instantiated with has SEC.

```
corollary (in gcounter) counter-convergence:
  assumes set (node-deliver-messages xs) = set (node-deliver-messages ys)
    and xs prefix of i
    and ys prefix of j
  shows apply-operations xs = apply-operations ys
using assms by(auto simp add: apply-operations-def intro: hb.convergence-ext
concurrent-operations-commute
          node-deliver-messages-distinct hb-consistent-prefix)
```

Figure 15: Isabelle proofs that the state-based G-Counter is convergent.

This and the above proof establish the following two lemmas:

- Operations that have been delivered at some node can be applied in any order up to causality and still achieve the same state (there is a more general result, since *all* operations on the G-Counter are concurrent, but we specialize to showing a more specific case to guide Isabelle's reuse of the proof).
- Having the same set of operations delivered at any two replicas ensures that those replicas are in the same state.

The first property is a helpful lemma which is used in internal proofs, but the second lemma should be familiar to the reader: this is the safety property of SEC! Note also that this is the first time that we are seeing our efforts in relaxing the network model bear fruit. That is, even though the two *sets* must be equal, we do not make a restriction on the number of times that a particular message is delivered at either node. This allows us to prove a stronger result that this CRDT achieves a consistent result despite the number of times that a message was (or was not) duplicated.

Finally, once we have shown these two properties, we can instantiate the *strong-eventual-consistency* locale, which is witness to the fact that this CRDT object achieves SEC. We present the instantiation now, and leave the proof to Section A.1:

```
sublocale sec: strong-eventual-consistency weak-hb hb interp-msg
  λops. ∃ xs i. xs prefix of i ∧ node-deliver-messages xs = ops λ x. None
```

Figure 16: Isabelle proof that the state-based G-Counter CRDT is SEC.

Incidentally by this point, the proof that SEC is inhabited by our encoding of the G-Counter SEC is mostly automatic, up to giving Isabelle some hints about rewrite and simplification rules that it should apply.

5.2.2 *State-based G-Set*

Now that we have verified a state-based G-Counter CRDT, we turn our attention to the other CRDT object for study in this thesis. This will be the state-based

G-Set, which is described in detail in Section 3.2. Readers may notice that the remaining sections in this chapter are shorter and shorter as we build up and reuse techniques from earlier proofs in later ones.

For now, we begin with an instantiation of the state-based G-Set CRDT, as follows:

type-synonym $('a)$ *state* $=$ *'a set*
type-synonym $('a)$ *operation* $=$ *'a state*

Figure 17: Isabelle types for the state and operations of a state-based G-Set.

Like in Figure 7, we parameterize the CRDT on the type of element in the set, which we denote in Isabelle as *'a*. Similar to our offers in the previous sub-section, we define the *operation* type to be a type-level synonym for the *state* type, which we interpret in the same way (that is, that receiving an "operation" from some other replica is equivalent to being told to merge our state with the received one, and replace our current state with the result).

Next, we define a simple insertion operation:

fun *insert* :: *'a* $\Rightarrow$ $($*'a state*$)$ $\Rightarrow$ $($*'a operation*$)$ **where**
insert a as $=$ $\{$ *a* $\}$

fun *gset-op* :: $($*'a operation*$)$ $\Rightarrow$ $($*'a state*$)$ $\rightharpoonup$ $($*'a state*$)$ **where**
gset-op a as $=$ *Some* $($ *as* $\cup$ *a* $)$

Figure 18: Isabelle definition of the insertion operation for a state-based G-Set.

Again, we define an operation *insert* for demonstration purposes.[8] Now that we have a convenience function for generating states that could be used to drive state transitions within the system, we can instate the interpretation of an operation at a state-based G-Set CRDT. This is the second function in the above Isabelle snippet. Like the state-based G-Counter CRDT, we map a pair of *'a operation* and *'a state* to a new state of the same type, or *None*.[9]

Faithful to the original specification in Figure 7, we interpret the join of two states (that is, two sets of items, one per replica) as the merge operation.

Because we are using Isabelle's *set* library and its built-in function $\cup$, we can leverage proofs about built-in Isabelle types, including the fact that $\cup$ is commutative, associative, and idempotent. Therefore, unlike our experience in the previous sub-section when specifying the state-based G-Counter CRDT, we do not need to prove these facts ourselves.[10]

[8] Again, our proofs reason about state *transitions*.

[9] The existing library in Gomes et al. [2017] requires that this function be a partial mapping, but we do not specify any behaviors which would cause our node to crash in ordinary execution here.

[10] Recall that in this instance, we were using a user-defined function *option-max*, and had an expanded obligation to prove that this function was commutative and associative; Isabelle inferred idempotence automatically.

Aside from some additional proofs which are standard to all of our instantiations of CRDTs using the library from Gomes et al. [2017], we can immediately instantiate the *strong-eventual-consistency* locale without additional proof. We present the statement of this locale below, and leave it and the additional proofs about the state-based G-Set to Section A.2.

sublocale *sec*: *strong-eventual-consistency weak-hb hb interp-msg*
$\lambda ops.\ \exists xs\ i.\ xs$ *prefix of* $i \wedge$ *node-deliver-messages* $xs = ops\ \{\}$

Figure 19: Isabelle instantiation of the *strong-eventual-consistency* locale for the state-based G-Set.

5.3 δ-STATE BASED CRDTS

We have reached the climax of this chapter in which we now set out to verify that δ-state based CRDT equivalents of the G-Counter and G-Set are also inhabitants of the SEC locale.

This follows simply from construction and our reductions. Recall the sentiment of our claim in Maxim 4.2 that all δ-state CRDTs are themselves like the op-based equivalent of state-based CRDTs, only with additional restriction on what states are sent to other replicas. All that suffices to show is that restricted executions of the datatype–that is, ones in which only δ-state fragments are sent, and not full state–still inhabit the SEC locale.

Recall that, since our proofs reason about state transitions inductively, we have implicitly covered the case in which only δ-fragments of state are exchanged between replicas. This is a consequence of our encoding of δ-state CRDTs as op-based CRDTs, and the fact that all δ-based CRDT messages are also state-based CRDT messages.

Since we have verified our CRDTs as inhabiting the SEC locale over all possible operations, we produced proofs for δ-state CRDTs as a side-effect of our strategy in Maxims 4.1 and 4.2.

We devote the remainder of this section to stating the types of the operation-producing functions for the δ-based CRDT equivalents of the G-Counter and G-Set.

5.3.1 *δ-state based G-Counter*

We begin first with our full definition of the δ-state based G-Counter CRDT. Like the state-based variant, we treat the state as a partial mapping between a transparent node identifier type and an optional value, referring to the number of increment operations performed locally at that node. Following Maxim 4.1, we treat the operation again as a type-level synonym for the state.

Similar to our treatment of the state-based G-Counter CRDT, we encode the state as a partial mapping from the set of node identifiers to an integer number

of times that an increment operation was performed at the replica belonging to that node identifier. Likewise, we treat the operation as a type-level synonym for this definition of the state.

The only difference (besides renaming *gcounter-op* to *delta-gcounter-op*) is that: the function *update* does not ever return a value from the underlying state which does not belong to the replica being updated. That is, we return a state which is *only* defined for the single replica being updated.

The full definition of the updated operation function in Isabelle is as follows:

```
fun inc :: 'id ⇒ ('id state) ⇒ ('id operation) where
inc who st = (λj. if who = j
              then Some (1 + (case (st who) of None ⇒ 0 | Some (x) ⇒ x))
              else None)
```

Figure 20: Isabelle definition of the δ-state G-Counter CRDT.

Here, we return a δ-state which is only defined for the single replica identifier being incremented. That is, we only return a value which is not *None* for the occurrence when the parameter *j* is bound to a value which equals *who*. When this is met, we increment the value in the state by one, and return the sum.

Since the body of the δ-based G-Counter CRDT is the same, and only the convenience function changed, all other proofs are the same. In Section 5.4, we discuss an alternative encoding which limits the kind of messages being sent at the type-level to be restricted only to δ-fragments.

5.3.2 *δ-state based G-Set*

Finally, we turn our attention to the remaining CRDT instance: the G-Set. Similar to our experience verifying the δ-based G-Counter, the specification of the CRDT itself is identical to the original encoding in Section 5.2.2, following our intuition in Maxim 4.2.

For completeness, we present the full instantiation of this type (again leaving the additional proofs to the Appendix in Section A.4):

type-synonym $('a)$ *state* $=$ *'a set*
type-synonym $('a)$ *operation* $=$ *'a state*

fun *insert* :: *'a* $\Rightarrow$ $('a$ *state*$)$ $\Rightarrow$ $('a$ *operation*$)$ **where**
insert a as $=$ $\{$ *a* $\}$

fun *delta-gset-op* :: $('a$ *operation*$)$ $\Rightarrow$ $('a$ *state*$)$ $\rightharpoonup$ $('a$ *state*$)$ **where**
delta-gset-op a as $=$ *Some* $($ *as* $\cup$ *a* $)$

locale *delta-gset* $=$ *network-with-ops - delta-gset-op* $\{\}$

Figure 21: Isabelle definition of the δ-state G-Set CRDT.

Notice that our encoding is identical as in the state-based G-Set example, but the definition of *insert* has changed. Instead of constructing and sending the union of the current set and the singleton set containing the item we wish to add, we construct only the singleton set.

This guides our understanding that if this CRDT only sends messages that are able to be generated from the modified *insert* function, that it will achieve SEC, and indeed we are able to instantiate the *strong-eventual-consistency* locale over this type. Because our proofs are inductive over state *transitions*, we have implicitly proved the case where only δ-fragments are sent as well.

In the following section, we discuss an alternate encoding which permits a more direct proof of this fact.

5.4 ALTERNATIVE ENCODING OF THE δ-STATE REDUCTION

In this Section, we discuss an alternative encoding in Isabelle of δ-state CRDTs. Our key insight following Maxim 4.2 is that in a system where the proofs are done inductively over state transitions, all executions which only exchange δ-fragments are implicitly verified. That is, since these messages comprise a subset of the set of messages which are sent by state-based CRDTs, our inductive hypothesis still holds, and the result is preserved for δ-state CRDTs.

But the key restriction in Maxim 4.2 is that δ-state CRDTs are ordinarily allowed to send only *fragments* of their state, not the entire state.[11] For our purposes, we devote the remainder of this section to exploring how this restriction is encoded at the type level in our proofs in Isabelle.

The approach that we take here is to let the operation type be a type-level synonym for a sort of refinement type of the state. Consider for a brief example the G-Set CRDT. Here, the full state is *'a set*, but the δ-fragments are singleton sets. Ordinarily we would make a type-level alias from *'a operation* to be the

[11] This restriction does not hold for certain anti-entropy algorithms which are implemented on top of δ-based CRDTs [Almeida et al., 2018]. This left to future work and discussed briefly in Section 6.2.

same as *'a state*, but this is too permissive. Recall that the operation–for our purposes–is analogous to the kind of the update message sent between replicas. We want to encode that this can *only* be the singleton set, not any arbitrary set. To do this, we let the *operation* type be a single element of *'a* type, which we interpret as the singleton set.

In the following sections, we will consider two examples of this restriction. Note that we are proving the same thing, so the underlying proof statement is unchanged. That is, in Section 5.3 we were reasoning about an inductive hypothesis over *all* possible state transitions. In this section, we are reasoning about smaller single transitions (e.g., in the case of a G-Set, adding at most one element in each step), but this is still sufficient to reason about all possible state transitions.

5.4.1 *Refined δ-state based G-Counter*

We begin first with the δ-state based G-Counter, and specify it using our alternate encoding. Recall that in the original specification in Section 5.3.1, we let the state type be a (partial) mapping from a transparent node identifier type *'a* to *nat*.

Aliasing the operation type to be a type-level synonym for the state allowed our δ-state CRDT instantiation to send *any* message, which is too permissive. Recall that in a δ-state G-Counter, we typically send a single update, e.g., $\{r_1 \mapsto n\}$ for a single replica r_1 incrementing the number of operations performed up to n. We specify this single-update in Isabelle as follows:

type-synonym ('*id*) *state* = *'id* ⇒ *int option*
type-synonym ('*id*) *operation* = *'id* × *int*

Figure 22: Isabelle definitions for the *state* and *operation* types for the restricted δ-based G-Counter.

Here, we encode the restriction that a δ-state G-Counter can only send an update about a single replica by encoding that its operation type is a pair of a transparent node identifier value and the number of increments performed at that node.

In the following figure, we present the remainder of the altered definitions to work around this more restricted *operation* type.

fun *inc* :: *'id* ⇒ (*'id state*) ⇒ (*'id operation*) **where**
inc who st = (*who*, (*1* + (*case* (*st who*) *of None* ⇒ *0* | *Some* (*x*) ⇒ *x*)))

fun *op-to-state* :: (*'id operation*) ⇒ (*'id state*) **where**
op-to-state (*who*, *count*) = (λ*x*. *if x* = *who then Some count else None*)

fun *delta-gcounter-op* :: (*'id operation*) ⇒ (*'id state*) ⇀ (*'id state*) **where**
delta-gcounter-op theirs ours = *Some* (λ *x*. *option-max* ((*op-to-state theirs*) *x*) (*ours x*))

Figure 23: Isabelle definitions of the remainder of functions for the restricted δ-state G-Counter.

In the above, we omit the definition of *option-max*, which is identical to Figure 12. First, we reimplement *inc* to return a value of the correct type by constructing a pair of the node being incremented, and the value that it is being incremented to. Now that this is done, we update our implementation of *delta-gcounter-op* to match the new type. Again, we return a function which takes the pairwise maximum between the old and new values corresponding to a given node. However, we can no longer pass the given operation as input to this function, since it does not have the same type as the state of our CRDT in this encoding.

To address this, we *convert* the operation into a state by constructing a state which is only defined for the single node being updated, and returns *None* for all other values. Once we have this, we can then call it with an arbitrary node to take its pairwise maximum to generate an updated state.

After specifying the CRDT using this alternate encoding, we did not have to update any of our existing proofs developed in Section 5.3.1, since Isabelle was able to infer the remainder of facts it needed to recheck our existing proofs.

5.4.2 *Refined δ-state based G-Set*

In this section, we apply the same techniques to show that an alternate encoding of the δ-based G-Set still achieves SEC. As illustrated in 5.4 we replace the definition of the *operation* type to only allow for restricted, single-element messages as follows:

type-synonym (*'a*) *state* = *'a set*
type-synonym (*'a*) *operation* = *'a*

Figure 24: Isabelle definitions for the *state* and *operation* types for the restricted δ-based G-Set.

First observe that the underlying type for *'a state* is unchanged, but that the new type for *'a operation* only allows a single value to be communicated in messages between two nodes.

Faced with this additional restriction, we update our proofs accordingly. Following the example in the previous section, we can imagine that our proofs will need to be updated in two locations:

1. The definition of *insert* will become simplified, since we will no longer have to refer to the current state when generating the message signaling an item has been inserted.

2. The interpretation of the operation will become slightly more complex, since we will have to treat the incoming item *encoded* in the operation as a singleton set, and will thus have to do that conversion.

We include the updated definitions of these two functions in Isabelle below:

fun *insert* :: $'a \Rightarrow ('a\ state) \Rightarrow ('a\ operation)$ **where**
insert a - = a

fun *delta-gset-op* :: $('a\ operation) \Rightarrow ('a\ state) \rightharpoonup ('a\ state)$ **where**
delta-gset-op a as = *Some* (*as* $\cup$ { *a* })

Figure 25: Isabelle definitions of the remainder of functions for the restricted δ-state G-Set.

Notice that our insertion operation became dramatically simpler. In fact, the function is so simple, that it is the identity on its first parameter. We could have dropped the second parameter from the function entirely,[12] but we leave it there to illustrate the fact that it *can* be ignored.

This simplification is balanced with a small amount of complexity added in the *delta-gset-op* function, which now constructs a singleton set from the incoming operation–referred to as *a*–into { *a* }.

As before, Isabelle is able to infer the remaining set of facts given our definitions above in order to check the unmodified proofs from the original encoding.

5.5 CONCLUSION

In this Chapter, we motivated our rationale behind relaxing the network model on top of which we verify our CRDTs. We described our proof strategy for relaxing the network model, and presented two example CRDTs which we verified on top of this network model. We began each example by showing the state-based object, and a proof that each CRDT inhabits the SEC locale, even on the relaxed network model.

We then reasoned that our state-based example CRDTs in fact establish the same goal for δ-state CRDTs without additional modification, according to our

[12] Making its type signature $'a \Rightarrow ('a\ operation)$.

result in Maxim 4.2. Finally, we presented an alternate encoding which restricts the set of messages nodes are allowed to send together, which more closely approximates the set of messages that δ-CRDTs are allowed to send, and again proved that this encoding is an inhabitant of SEC.

6

FUTURE WORK

This chapter outlines potential future research directions based on interesting and under-explored areas in this work. Here, we will outline three directions in the area of verifying δ-state CRDTs, as well as some insight that might be gained by exploring each of these directions. It is our hope that future researchers in this area may choose to conduct further investigation into these areas.

6.1 VERIFYING ADDITIONAL δ-STATE CRDTS

In our work, we presented examples of two δ-state CRDTs: the δ G-Counter, and the δ G-Set. An immediate future direction is to investigate and verify more instances of δ-state CRDTs.

One area of particular interest is in the *composition* of multiple δ-state CRDTs. We have begun investigating the instantiation of a *pair* locale, which takes as arguments two independent δ-state CRDTs, known as "left" and "right." Our hope is that provided existing instantiations of both of the sub-CRDTs, that a *pair* locale given two already-verified CRDTs could be used without additional proof burden to create another instance of the *network-with-ops* locale. That is: can two already-verified δ-state CRDTs be used to compose a new δ-state CRDT which is their product without additional proof burden?

If this were possible, two new CRDTs would be verified without effort: the PN-Counter and the 2P-Set. These two CRDTs are the most straightforward composition of other known CRDTs. Namely, the PN-Counter supports both an inc and dec operation by maintaining *two* counters (each of which is treated as a single G-Counter, so the overall state is still monotone and thus forms a join semi-lattice).

The PN-Counter has two δ-state based G-Counter, which we refer to as $(\mathsf{fst}\, s)$ and $(\mathsf{snd}\, s)$, where $s \in S$ refers to the state of the PN-Counter. One possible specification for a δ-based PN-Counter follows:

$$\mathsf{PN\text{-}Counter}_\delta = \begin{cases} S : \mathbb{N}_0^{|\mathcal{I}|} \times \mathbb{N}_+^{|\mathcal{I}|} \\ s^0 : [0, 0, \cdots, 0] \times [0, 0, \cdots, 0] \\ q^\delta : \lambda s. \sum_{i \in \mathcal{I}} (\mathsf{fst}\ s)(i) - \sum_{i \in \mathcal{I}} (\mathsf{snd}\ s)(i) \\ u^\delta : \lambda s, (i, op). \begin{cases} \{i \mapsto 1\} \times \emptyset & \text{if } op = + \\ \emptyset \times \{i \mapsto 1\} & \text{if } op = - \end{cases} \\ m^\delta : \lambda s_1, s_2. \begin{array}{l} \{\max\{i_1, i_2\} : i \in \mathsf{dom}((\mathsf{fst}\ s_1) \cup (\mathsf{fst}\ s_1))\} \\ \times \{\max\{i_1, i_2\} : i \in \mathsf{dom}((\mathsf{snd}\ s_1) \cup (\mathsf{snd}\ s_1))\} \end{array} \end{cases}$$

Figure 26: δ-state based PN-Counter CRDT

Minor additional consideration is given to the updating function, u^δ, which returns an empty-set on the counter *not* being updated. Finally, the merging function m^δ merges the left- and right-hand sides of the counter separately, and returns a pair. The 2P-Set is similar in function to the above, substituting a δ-state based G-Set in place of the G-Counter.

If such a *pair* locale exists, we believe it would be as straightforward as instantiating this locale over two copies of the G-Counter and G-Set to obtain the PN-Counter and 2P-Set immediately.

6.2 DIRECT δ-STATE CRDT PROOFS

To explore this idea, we drew significant inspiration from the work of Almedia and his co-authors in Almeida et al. [2018] to restate δ-state CRDTs in terms of op-based CRDTs in an effort to reuse as much of their library as possible.

A significant drawback of this approach is that we are bound to the same restrictions as op-based CRDTs, which are inherently more restricted than state-based CRDTs. Much of this restriction comes from the *eventual delivery* property of EC, which states that [Shapiro et al., 2011]:

$$\forall i, j.\, f \in c_i \Rightarrow \Diamond f \in c_j$$

or that for any pair of correct replicas i, j with histories c_i and c_j, respectively, an update received at one of those replicas is eventually received at all other replicas.

Of course, under relaxed delivery semantics (i.e., in the case that the network may delay messages for an infinite amount of time), op-based CRDTs do not achieve this property [Shapiro et al., 2011]. Namely, if an operation is performed at some replica, and that message is dropped while in transit to another replica, that replica will never receive the message.

State-based CRDTs do not suffer from this problem, since *every* update they send encapsulates the history of all previous updates, since each update is

either reflected in the state, or subsumed by some later update which is itself reflected in the state [Shapiro et al., 2011]. Since the entirety of the state is shared with each replica during an update, state-based CRDTs do not need to impose an additional delivery relation in order to prove that they achieve SEC.

op-based CRDTs, on the other hand, *do* need to specify an additional delivery relation on top of their definition. That is, the delivery relation P is a *predicate* over network behaviors in which the eventual delivery property can hold. In other words, for op-based CRDTs:

$$P \Rightarrow \forall i, j.\, f \in c_i \Rightarrow \Diamond f \in c_j$$

where it is a standard assumption that P preserves (1) message order up to concurrent messages and (2) at least once delivery [Almeida et al., 2018, Shapiro et al., 2011].[1]

However, specifying δ-based CRDTs as a refinement of state-based CRDTs directly would not be sufficient for a constructive proof that δ-state based CRDTs achieve SEC. This is due to the fact that δ-state CRDTs send state *fragments*, which makes them the state-based analogue of op-based CRDTs. Without an additional delivery relation, δ-state CRDT replicas which do not receive some update will never catch up without additional updates.

Consider the figure below. In this example, we have three δ-state CRDT replicas of a δ-based GCounter, and an inc is performed at r_1. Immediately, r_1 generates the state fragment $\{r_1 \mapsto 1\}$, and sends it to the other replicas, r_2 and r_3. For the sake of example, say that the network drops the update in route to r_3 such that it is never received by r_3:

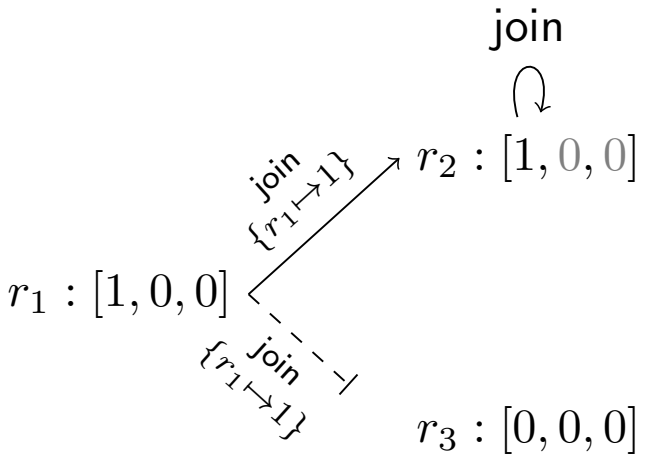

Figure 27: δ-state CRDTs violating SEC without the causal merging condition.

[1] In practice, vector timestamps or globally unique identifiers are associated with each message at the network layer, and messages are reordered upon delivery to ensure that messages are delivered in the correct order. Since all messages are eventually delivered under the precondition P, this is a standard assumption.

Without any future updates, neither of the replicas that *have* received the update will ever have reason to update r_3 again. This is a demonstration of a SEC violation, since:

$$\{r_1 \mapsto 1\} \in r_i \not\Rightarrow \Diamond\{r_1 \mapsto 1\} \in r_3, \quad i \in \{1, 2\}$$

That is, though the update $\{r_1 \mapsto 1\}$ is in the node histories of r_1 and r_2 (both of which are behaving correctly), that update is *never* in the history of node r_3, which is also behaving correctly.

A critical issue in the above example is that r_2 merges the update from r_1 immediately–thus placing the update in that node's history–without knowing whether or not it has been received by r_3. An anti-entropy algorithm like in Almeida et al. [2018] addresses these problems. The goal of an anti-entropy algorithm for δ-state CRDTs is to do the following:

1. On an operation, generate the δ-mutation, and apply it to both the local state, and a temporary *δ-group*.
2. Periodically, randomly choose between the current state and current δ-group, and send its entire contents to all other replicas, and flush the δ-group.

This ensures that–even without outside interaction–the system as in Figure 27 will eventually recover. This follows since either one of r_1 or r_2 will at some point send their full state to all other replicas, including r_3, at which point r_3 will have caught up.

We believe that it would be a worthwhile research goal to encode this anti-entropy algorithm into a proof assistant, and specify that δ-state CRDTs achieve SEC *without* a correspondence to traditional op-based CRDTs. Similarly to our work, in which we found a correspondence between δ- and op-based CRDTs, we believe that specifying δ-state CRDTs on their own would highlight the ways in which δ-state CRDTs are *different* from op-based CRDTs.

Likewise, specifying the goal in this fashion would allow the proof to reason about more network behaviors without a delivery predicate P, since the proof would be aided by the periodic behavior of the anti-entropy algorithm above.

6.3 CAUSALLY CONSISTENT δ-CRDTS

Another difference between op- and state-based CRDTs is that state-based CRDTs require a *causal merging condition* in order to ensure causal consistency (that is, that updates are applied in a fashion that preserves their causality), whereas in op-based CRDTs this is traditionally an assumption placed on P [Shapiro et al., 2011].

The authors of Almeida et al. [2018] define a *δ-interval* $\Delta_i^{a,b}$ as:

$$\Delta_i^{a,b} = \bigsqcup \left\{ d_i^k : k \in [a, b) \right\}$$

that is, $\Delta_i^{a,b}$ contains the deltas that occurred at replica i beginning at time a and up until time b. They go on to use this δ-interval to define the *causal merging condition*, which is that replica i only joins a δ-interval $\Delta_j^{a,b}$ into its own state X_i if:

$$X_i \sqsupseteq X_j^a$$

That is, updates are only applied locally if they occurred *before* the latest-known update at replica i.

Algorithms which uphold the causal merging condition on δ-intervals have been proven on paper to satisfy Causal Consistency (CC) in addition to SEC. To our knowledge, this result has not been mechanized, and so we believe it would be a worthwhile direction of future research to specify the causal merging condition and associated anti-entropy algorithms which preserve it into an interactive theorem prover and mechanize the results of Almeida et al. [2018].

If the above is the subject of further exploration, we believe that it would be additionally possible to prove that δ-state CRDTs achieve SEC by a *simulation proof* that establishes their correspondence with state-based CRDTs. This is mentioned as Proposition 3 in Almeida et al. [2018], but we believe that this is another fruitful area for formal verification.

7 CONCLUSION

In this thesis, we extended the work in Gomes et al. [2017] to provide a mechanized proofs that δ-CRDTs [Almeida et al., 2018] achieve SEC [Shapiro et al., 2011].

Our central intuition (cf., Sections 4.1 and 4.2) was to treat δ- and state-based CRDTs as refinements of op-based CRDTs. This allowed us to successfully verify that two CRDTs–the G-Counter, and G-Set–achieve SEC when specified both in the state- and δ-state based style.

In addition, we relaxed the network model by removing an assumption that all messages are unique. While our main result is still predicated on a set of nice delivery semantics P, this allowed us to quantify over an expanded set of all possible network executions.

Together, this allowed us to restate the main result of Almeida et al. [2018] in a mechanized fashion. We believe that δ-state CRDTs satisfy an appealing "best-of-both-worlds" property. δ-state CRDTs require relatively little of the network (like op-based CRDTs), yet still maintain a relatively small payload size (like state-based CRDTs). This places great interest on formal verification of their convergence properties.

In the future, we hope to see our result extended by specifying δ-state CRDTs in terms of their state-based counterparts, as well as mechanizing well-known anti-entropy algorithms and causality constraints on applying updates from other replicas [Almeida et al., 2018]. We believe that this would be sufficient to remove the precondition on a set of delivery semantics P from our result.

A

ADDITIONAL PROOFS

In this appendix, we provide the full proof scripts used in this work. The source is available for free at: https://github.com/ttaylorr/thesis.

A.1 STATE-BASED G-COUNTER CRDT

locale *gcounter* = *network-with-ops* - *gcounter-op* λ *x*. *None*

lemma (**in** *gcounter*) *option-max-assoc*:
option-max a (*option-max b c*) = *option-max* (*option-max a b*) *c*
apply (*induction a*; *induction b*; *induction c*)
apply (*auto*)
done

lemma (**in** *gcounter*) *option-max-commut*: *option-max a b* = *option-max b a*
apply (*induction a*; *induction b*)
apply (*auto*)
done

lemma (**in** *gcounter*) [*simp*] : *gcounter-op x* ▷ *gcounter-op y* = *gcounter-op y* ▷ *gcounter-op x*
apply (*auto simp add*: *kleisli-def option-max-assoc*)
apply (*simp add*: *option-max-commut*)
done

lemma (**in** *gcounter*) *concurrent-operations-commute*:
assumes *xs prefix of i*
shows *hb.concurrent-ops-commute* (*node-deliver-messages xs*)
using *assms*
apply(*clarsimp simp*: *hb.concurrent-ops-commute-def*)
apply(*unfold interp-msg-def*, *simp*)
done

corollary (**in** *gcounter*) *counter-convergence*:
assumes *set* (*node-deliver-messages xs*) = *set* (*node-deliver-messages ys*)
and *xs prefix of i*
and *ys prefix of j*
shows *apply-operations xs* = *apply-operations ys*
using *assms* **by**(*auto simp add*: *apply-operations-def*
intro: *hb.convergence-ext concurrent-operations-commute*

node-deliver-messages-distinct hb-consistent-prefix)

context *gcounter* **begin**

sublocale *sec*: *strong-eventual-consistency weak-hb hb interp-msg*
λops. ∃ *xs i*. *xs prefix of i* ∧ *node-deliver-messages xs* = *ops λ x*. *None*
apply(*standard*; *clarsimp*)
apply(*auto simp add*: *hb-consistent-prefix drop-last-message node-deliver-messages-distinct concurrent-operations-commute*)
apply(*metis* (*full-types*) *interp-msg-def gcounter-op.elims*)
using *drop-last-message* **apply** *blast*
done
end

end

A.2 STATE-BASED G-SET CRDT

locale *gset* = *network-with-ops* - *gset-op* {}

lemma (**in** *gset*) [*simp*] : *gset-op x* ▷ *gset-op y* = *gset-op y* ▷ *gset-op x*
apply (*auto simp add*: *kleisli-def*)
done

lemma (**in** *gset*) *concurrent-operations-commute*:
assumes *xs prefix of i*
shows *hb.concurrent-ops-commute* (*node-deliver-messages xs*)
using *assms*
apply(*clarsimp simp*: *hb.concurrent-ops-commute-def*)
apply(*unfold interp-msg-def*, *simp*)
done

corollary (**in** *gset*) *set-convergence*:
assumes *set* (*node-deliver-messages xs*) = *set* (*node-deliver-messages ys*)
and *xs prefix of i*
and *ys prefix of j*
shows *apply-operations xs* = *apply-operations ys*
using *assms* **by**(*auto simp add*: *apply-operations-def intro*: *hb.convergence-ext concurrent-operations-commute node-deliver-messages-distinct hb-consistent-prefix*)

context *gset* **begin**

sublocale *sec*: *strong-eventual-consistency weak-hb hb interp-msg*
λops. ∃ *xs i*. *xs prefix of i* ∧ *node-deliver-messages xs* = *ops* {}
apply(*standard*; *clarsimp*)

apply(*auto simp add*: *hb-consistent-prefix drop-last-message node-deliver-messages-distinct concurrent-operations-commute*)
apply(*metis* (*full-types*) *interp-msg-def gset-op.elims*)
using *drop-last-message* **apply** *blast*
done
end

end

A.3 δ-STATE G-COUNTER CRDT

locale *delta-gcounter* = *network-with-ops* - *delta-gcounter-op* λ *x*. *None*

lemma (**in** *delta-gcounter*) *option-max-assoc*:
option-max a (*option-max b c*) = *option-max* (*option-max a b*) *c*
apply (*induction a*; *induction b*; *induction c*)
apply (*auto*)
done

lemma (**in** *delta-gcounter*) *option-max-commut*: *option-max a b* = *option-max b a*
apply (*induction a*; *induction b*)
apply (*auto*)
done

lemma (**in** *delta-gcounter*) [*simp*] : *delta-gcounter-op x* ▷ *delta-gcounter-op y* = *delta-gcounter-op y* ▷ *delta-gcounter-op x*
apply (*auto simp add*: *kleisli-def option-max-assoc*)
apply (*simp add*: *option-max-commut*)
done

lemma (**in** *delta-gcounter*) *concurrent-operations-commute*:
assumes *xs prefix of i*
shows *hb.concurrent-ops-commute* (*node-deliver-messages xs*)
using *assms*
apply(*clarsimp simp*: *hb.concurrent-ops-commute-def*)
apply(*unfold interp-msg-def*, *simp*)
done

corollary (**in** *delta-gcounter*) *counter-convergence*:
assumes *set* (*node-deliver-messages xs*) = *set* (*node-deliver-messages ys*)
and *xs prefix of i*
and *ys prefix of j*
shows *apply-operations xs* = *apply-operations ys*
using *assms* **by**(*auto simp add*: *apply-operations-def intro*: *hb.convergence-ext concurrent-operations-commute node-deliver-messages-distinct hb-consistent-prefix*)

context *delta-gcounter* **begin**

sublocale *sec*: *strong-eventual-consistency weak-hb hb interp-msg*
λops. ∃ *xs i*. *xs prefix of i* ∧ *node-deliver-messages xs* = *ops λ x*. *None*
apply(*standard*; *clarsimp*)
apply(*auto simp add*: *hb-consistent-prefix drop-last-message node-deliver-messages-distinct concurrent-operations-commute*)
apply(*metis* (*full-types*) *interp-msg-def delta-gcounter-op.elims*)
using *drop-last-message* **apply** *blast*
done
end

end

A.4 δ-STATE G-SET CRDT

locale *delta-gset* = *network-with-ops* - *delta-gset-op* {}

lemma (**in** *delta-gset*) [*simp*] : *delta-gset-op x* ▷ *delta-gset-op y* = *delta-gset-op y* ▷ *delta-gset-op x*
apply (*auto simp add*: *kleisli-def*)
done

lemma (**in** *delta-gset*) *concurrent-operations-commute*:
assumes *xs prefix of i*
shows *hb.concurrent-ops-commute* (*node-deliver-messages xs*)
using *assms*
apply(*clarsimp simp*: *hb.concurrent-ops-commute-def*)
apply(*unfold interp-msg-def*, *simp*)
done

corollary (**in** *delta-gset*) *set-convergence*:
assumes *set* (*node-deliver-messages xs*) = *set* (*node-deliver-messages ys*)
and *xs prefix of i*
and *ys prefix of j*
shows *apply-operations xs* = *apply-operations ys*
using *assms* **by**(*auto simp add*: *apply-operations-def intro*: *hb.convergence-ext concurrent-operations-commute node-deliver-messages-distinct hb-consistent-prefix*)

context *delta-gset* **begin**

sublocale *sec*: *strong-eventual-consistency weak-hb hb interp-msg*
λops. ∃ *xs i*. *xs prefix of i* ∧ *node-deliver-messages xs* = *ops* {}
apply(*standard*; *clarsimp*)
apply(*auto simp add*: *hb-consistent-prefix drop-last-message node-deliver-messages-distinct concurrent-operations-commute*)

```
  apply(metis (full-types) interp-msg-def delta-gset-op.elims)
 using drop-last-message apply blast
 done
end

end
```

A.5 RESTRICTED δ-STATE G-COUNTER CRDT

```
locale delta-gcounter = network-with-ops - delta-gcounter-op λ x. None

lemma (in delta-gcounter) option-max-assoc:
 option-max a (option-max b c) = option-max (option-max a b) c
 apply (induction a; induction b; induction c)
 apply (auto)
 done

lemma (in delta-gcounter) option-max-commut: option-max a b = option-max b a
 apply (induction a; induction b)
 apply (auto)
 done

lemma (in delta-gcounter) [simp] : delta-gcounter-op x ▷ delta-gcounter-op y = delta-gcounter-op
y ▷ delta-gcounter-op x
 apply (auto simp add: kleisli-def option-max-assoc)
 apply (simp add: option-max-commut)
 done

lemma (in delta-gcounter) concurrent-operations-commute:
 assumes xs prefix of i
 shows hb.concurrent-ops-commute (node-deliver-messages xs)
 using assms
 apply(clarsimp simp: hb.concurrent-ops-commute-def)
 apply(unfold interp-msg-def, simp)
 done

corollary (in delta-gcounter) counter-convergence:
 assumes set (node-deliver-messages xs) = set (node-deliver-messages ys)
    and xs prefix of i
    and ys prefix of j
  shows apply-operations xs = apply-operations ys
using assms by(auto simp add: apply-operations-def intro: hb.convergence-ext concurrent-operations-commute
          node-deliver-messages-distinct hb-consistent-prefix)

context delta-gcounter begin
```

sublocale *sec*: *strong-eventual-consistency weak-hb hb interp-msg*
λops. ∃ *xs i*. *xs prefix of i* ∧ *node-deliver-messages xs* = *ops λ x*. *None*
apply(*standard*; *clarsimp*)
apply(*auto simp add*: *hb-consistent-prefix drop-last-message node-deliver-messages-distinct concurrent-operations-commute*)
apply(*metis* (*full-types*) *interp-msg-def delta-gcounter-op.elims*)
using *drop-last-message* **apply** *blast*
done
end

end

A.6 RESTRICTED δ-STATE G-SET CRDT

locale *delta-gset* = *network-with-ops* - *delta-gset-op* {}

lemma (**in** *delta-gset*) [*simp*] : *delta-gset-op x* ▷ *delta-gset-op y* = *delta-gset-op y* ▷ *delta-gset-op x*
apply (*auto simp add*: *kleisli-def*)
done

lemma (**in** *delta-gset*) *concurrent-operations-commute*:
assumes *xs prefix of i*
shows *hb.concurrent-ops-commute* (*node-deliver-messages xs*)
using *assms*
apply(*clarsimp simp*: *hb.concurrent-ops-commute-def*)
apply(*unfold interp-msg-def*, *simp*)
done

corollary (**in** *delta-gset*) *set-convergence*:
assumes *set* (*node-deliver-messages xs*) = *set* (*node-deliver-messages ys*)
and *xs prefix of i*
and *ys prefix of j*
shows *apply-operations xs* = *apply-operations ys*
using *assms* **by**(*auto simp add*: *apply-operations-def intro*: *hb.convergence-ext concurrent-operations-commute node-deliver-messages-distinct hb-consistent-prefix*)

context *delta-gset* **begin**

sublocale *sec*: *strong-eventual-consistency weak-hb hb interp-msg*
λops. ∃ *xs i*. *xs prefix of i* ∧ *node-deliver-messages xs* = *ops* {}
apply(*standard*; *clarsimp*)
apply(*auto simp add*: *hb-consistent-prefix drop-last-message node-deliver-messages-distinct concurrent-operations-commute*)
apply(*metis* (*full-types*) *interp-msg-def delta-gset-op.elims*)
using *drop-last-message* **apply** *blast*

```
  done
end

end
```

BIBLIOGRAPHY

P. S. Almeida, A. Shoker, and C. Baquero. Delta state replicated data types. *Journal of Parallel and Distributed Computing*, 111:162–173, Jan 2018. ISSN 0743-7315. doi: 10.1016/j.jpdc.2017.08.003. URL http://dx.doi.org/10.1016/j.jpdc.2017.08.003.

Apple, Inc. ios runtime headers. https://github.com/nst/iOS-Runtime-Headers, 2018.

C. Baquero, P. S. Almeida, and A. Shoker. Making Operation-Based CRDTs Operation-Based. In D. Hutchison, T. Kanade, B. Steffen, D. Terzopoulos, D. Tygar, G. Weikum, K. Magoutis, P. Pietzuch, J. Kittler, J. M. Kleinberg, A. Kobsa, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, and C. P. Rangan, editors, *4th International Conference on Distributed Applications and Interoperable Systems (DAIS)*, volume LNCS-8460 of *Distributed Applications and Interoperable Systems*, pages 126–140, Berlin, Germany, June 2014. Springer. doi: 10.1007/978-3-662-43352-2_11. URL https://hal.inria.fr/hal-01287738.

G. Cabrita and N. Preguiça. Non-uniform replication, 2017.

C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer Publishing Company, Incorporated, 2nd edition, 2011. ISBN 3642152597.

V. Enes, P. S. Almeida, C. Baquero, and J. Leitão. Efficient synchronization of state-based crdts, 2018.

V. B. F. Gomes, M. Kleppmann, D. P. Mulligan, and A. R. Beresford. Verifying strong eventual consistency in distributed systems. *CoRR*, abs/1707.01747, 2017. URL http://arxiv.org/abs/1707.01747.

H. Howard. Distributed consensus revised. Technical Report UCAM-CL-TR-935, University of Cambridge, Computer Laboratory, Apr. 2019. URL https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-935.pdf.

H. Howard and R. Mortier. Paxos vs raft. *Proceedings of the 7thWorkshop on Principles and Practice of Consistency for Distributed Data*, Apr 2020. doi: 10.1145/3380787.3393681. URL http://dx.doi.org/10.1145/3380787.3393681.

L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978. ISSN 0001-0782. doi: 10.1145/359545.359563. URL https://doi.org/10.1145/359545.359563.

L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998. ISSN 0734-2071. doi: 10.1145/279227.279229. URL https://doi.org/10.1145/279227.279229.

T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association. ISBN 978-1-931971-10-2. URL https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

Redis, Inc. Under the hood: Redis crdts. https://redislabs.com/docs/under-the-hood/, 2020.

M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free Replicated Data Types. Research Report RR-7687, July 2011. URL https://hal.inria.fr/inria-00609399.

A. van der Linde, J. a. Leitão, and N. Preguiça. δ-crdts: Making δ-crdts delta-based. In *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data*, PaPoC '16, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342964. doi: 10.1145/2911151.2911163. URL https://doi.org/10.1145/2911151.2911163.

J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *PLDI 2015: Proceedings of the ACM SIGPLAN 2015 Conference on Programming Language Design and Implementation*, pages 357–368, Portland, OR, USA, June 2015.

D. Woos, J. R. Wilcox, S. Anton, Z. Tatlock, M. D. Ernst, and T. Anderson. Planning for change in a formal verification of the raft consensus protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2016, page 154–165, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341271. doi: 10.1145/2854065.2854081. URL https://doi.org/10.1145/2854065.2854081.