André dos Reis Martins Rijo

Licenciatura em Engenharia Informática

Building Tunable CRDTs

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

Orientadora: Carla Ferreira, Professora Auxiliar,
Universidade Nova de Lisboa

Co-orientador: Nuno Manuel Ribeiro Preguiça, Professor Associado,
Universidade Nova de Lisboa

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Setembro, 2018

Building Tunable CRDTs

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "novathesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

To my parents, friends and advisers.

Acknowledgements

I would like to thank both my advisers, Prof. Dr. Carla Ferreira and Prof. Dr. Nuno Preguiça, for giving me the opportunity to work in such an interesting theme. Their support, advising and guidance were essential for the success of this work.

I also want to thank all my colleagues, friends and professors for their continuous support and motivation. All of them, in their own way, were essential for my "survival" during this course. In special a big thank you to my best friend André Monteiro who was always available to support, motivate and put up with me every time I was more grumpy or nervous.

Finally I would like to show my most sincere gratitude to both my parents, Rosa Rijo and Carlos Rijo, who have raised, inspired, supported and helped me in every way possible during all the years that I have lived so far. Without them I would had never gotten so far and be the person who I am today.

Abstract

Nowadays multiple large-scale services are hosted on the Internet, many of them with millions of daily users. These systems need to scale efficiently, providing fast access and being always available, despite failures of the servers or of the network and high amounts of users accessing the service. As such, these services typically trade strong consistency for high availability and low latency. However, not having strong consistency implies that conflicts arising from concurrent updates will occur, which need to be solved. Conflict-Free Replicated Data Types (CRDTs) provide low latency and solve conflicts automatically, ensuring eventual state convergence.

However, one shortcoming of CRDTs is the way they deal with concurrency conflicts – usually they solve them automatically by applying a specific policy. For example, in a set, when an *add* and *remove* for the same element happen concurrently, a possible policy is to give priority to the *add*. These policies are limited and not adequate for all applications, especially since CRDTs don't allow for policies dependent on the application context, such as, give priority to *add* if it's element e, but give priority to *remove* if it's element f.

As such, in this thesis we propose a new type of CRDTs, called tunable CRDT (t-CRDT), which allows the programmer to specify for each operation what is the desired conflict solving policy, by supplying a simple boolean function. The programmer can either supply his own policy or use one of the many we provide in our t-CRDT library. T-CRDTs solve conflicts automatically by applying the policies in each operation.

This new type of CRDTs adapt more easily to each application specific needs, as it gives more control to the programmer while still having the main properties of CRDTs, i.e., eventual convergence of state and low latency. With this, it is expected that more applications can start using CRDTs as their data solution and benefit from their properties.

Keywords: CRDT, semantic model, eventual consistency, causal consistency, replication, concurrency conflicts, conflict solving policies, eventual state convergence

Resumo

Hoje em dia vários serviços de larga escala são fornecidos na Internet, com muitos destes tendo diariamente milhões de utilizadores. Estes sistemas têm que escalar eficientemente, fornecendo acesso rápido e estando sempre disponíveis, independentemente de falhas em servidores ou na rede e quantidades elevadas de utilizadores a aceder ao serviço. Para tal, normalmente estes sistemas trocam garantias de consistência forte por alta disponibilidade e latência baixa. Contudo, não ter consistência forte implica que irão ocorrer conflitos devido a actualizações concorrentes, sendo que estes conflitos têm que ser resolvidos. Os Tipos de Dados Sem Conflitos (CRDTs) providenciam baixa latência e resolvem os conflitos automaticamente, garantido convergência eventual do estado.

Contudo, uma limitação dos CRDTs é o modo como lidam com os conflitos de concorrência – tipicamente resolvem-os automaticamente através do uso de uma estratégia específica. Por exemplo, num conjunto, quando a *adição* e *remoção* de um mesmo elemento são concorrentes, uma possivel estratégia é dar prioridade à adição. Estas estratégias são limitadas e não são adequadas para todas as aplicações, principalmente porque os CRDTs não permitem estratégias dependentes do contexto da aplicação, por exemplo, dar prioridade a uma *adição* se for o elemento *e*, mas, para o elemento *f*, dar prioridade à *remoção*.

Por isso nesta tese propomos um novo tipo de CRDTs, os quais chamamos de CRDT ajustável (t-CRDT), sendo que estes permitem ao programador especificar para cada operação qual é a estratégia de resolução de conflitos desejada ao fornecer uma função booleana simples. O programador pode providenciar a sua própria função ou usar uma das muitas que fornecemos na nossa biblioteca de t-CRDTs. Os t-CRDTs resolvem os conflitos automaticamente através da aplicação das funções em cada operação.

Este novo tipo de CRDTs adapta-se mais facilmente às necessidades específicas de cada aplicação, dando mais controlo ao programador mas mantendo as propriedades principais dos CRDTs, ou seja, convergência eventual do estado e baixa latência. Com isto é esperado que mais aplicações possam começar a usar CRDTs como a sua solução de dados e beneficiar das suas propriedades.

Palavras-chave: CRDT, modelo semantico, consistência eventual, consistência causal, replicação, conflitos de concorrência, estratégia de resolução de conflitos, convergência de estado eventual

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

$\|\|$	Concurrent symbol.
$\prec$	Happens-Before symbol.
δ-CRDT	Delta-based CRDT.
CmRDT	Operation-based CRDT.
CRDT	Conflict-Free Replicated Data Type.
CvRDT	State-based CRDT.
hb	Happens-Before.
LUB	Least Upper Bound.
LWW	Last-Writter-Wins policy.
MV	Multi-Value policy.
op	operation.
op-CRDT	Operation-based CRDT.
PCS	Principle of Conflicting States.
PPE	Principle of Permutation Equivalence.
PSE	Principle of Sequential Equivalence.
SEC	Strong Eventual Consistency.
state-CRDT	State-based CRDT.
t-CRDT	Tunable CRDT.

Chapter 1

Introduction

1.1 Context

In today's society a vast part of the world population constantly uses multiple services provided on the Internet. There's many different services available, namely social networks, online shopping, entertainment services, etc. Some of those services have millions of users daily accessing them from the most diverse devices, such as computers, mobile phones, tablets, smartwatches, among others. Multiple applications have been developed surrounding those services and users expect them to be always available and have low latency, despite faults that may happen on the servers or loss of Internet connection.

For those services to scale despite the increasing amount of users, usually they are replicated across multiple machines (replicas) in multiple data centers that are distributed across the world. Having the user's data replicated allows for better fault-tolerance of the service and supports lower latency for the users. However, replicating data across the globe also implies that a good trade-off between performance, availability and data consistency needs to be found, as it is impossible for a distributed system to fully provide them simultaneously [14]. Some services, such as banking, need to have its data consistent all the time and, as such, prefer to provide partition-tolerance and strong data consistency, even if some operations take considerable time to execute. Other services, however, can cope with having weaker consistency guarantees as long as they provide low latency for the user and are highly available.

In order to provide good performance, systems with weak consistency guarantees typically allow operations to be executed in just one replica, unlike in systems with strong consistency guarantees that need to synchronize across the replicas. However, executing operations without synchronization makes the replicas' state diverge, as operations can be executed concurrently in different replicas. This poses problems on how can the

diverging replica's state be converged into one common state, despite the possibility of conflicting operations having been executed.

One possibility for solving the convergence problem is to have application programmers directly dealing with the conflicts themselves. However, not only does that make developing a distributed application more difficult, it is also error-prone [2, 27]. As such, one of the proposed solutions for that problem are the Conflict-Free Replicated Data Types (CRDTs) [27], which allows replicas to apply operations concurrently, with low latency and still guarantee that their states will eventually converge, without requiring application programmers to deal with the conflicts themselves.

1.2 Motivation

In the literature, multiple CRDTs for different data types, such as counters, sets, maps and sorted lists have been proposed [2, 12, 15, 23, 27, 31]. Most CRDTs have low requirements from the network, which allows them to be used in many different scenarios, such as a P2P network (for example, in Legion [22]) or a distributed database (for example, in AntidoteDB [10] and Riak KV [5]). Their low network requirements also make them adequate for systems which need to scale to millions of users. CRDTs forego strong consistency guarantees in exchange for low latency and high availability, allowing operations to be executed even when a replica is disconnected from the remaining replicas.

Because operations are executed without coordination between replicas, in order to guarantee state convergence CRDTs deal with concurrency conflicts by applying specific strategies or policies [1]. For example, in a set, if a concurrent addition and removal of the same element happen, priority may be given to the addition (i.e., the element stays in the set). However, some applications may prefer to give priority to the remove. For other applications, it may even be desirable for the priority to be dependent on the element or application context.

Although it is beneficial for the application programmer to not have to solve concurrency conflicts by himself, the fact that typically CRDTs only provide one policy for solving conflicts limits their usage. As such, it would be desirable for many applications if the programmer had a way to select between different concurrency conflicts policies for each operation, or a way to specify their own, while still having guarantees of state convergence. With this, the integration of CRDTs in applications would be easier, allowing more applications to use CRDTs as their data solution and benefit from their advantages.

1.3 Proposed Solution

The main goal of this work is to tackle the problem discussed previously, that is, to offer more control to the programmer in deciding how the CRDT should solve concurrency conflicts for each operation.

[1] In the remaining of this document, we'll use policy and strategy interchangeably.

One possible solution consists in introducing CRDTs which offer a different conflict solving policy or, possibly, multiple policies in one CRDT, by using different operations. For example, in a set CRDT, two removes could be offered: one of them that gives priority to concurrent adds, while the other bypasses any concurrent add for the same element. This approach is useful for some applications and, as such, we submitted a paper [24] on a set CRDT that we specified which offers those two removes. Some examples of CRDTs following this approach that we specified in early iterations of this work can be found in Appendix B. Unfortunately, this approach is still limited: the programmer has no way of specifying his own policy for solving conflicts which could be dependent on the application/data context, and the choice of policies is still small and fixed.

In this work we propose a different solution, which is both more generic and allows the user to specify his own policies. We start by defining a generic CRDT model in which the programmer can associate to each operation the policy(ies) to be applied for solving conflicts. This model is responsible for storing operations and applying the policies to calculate which operations are still relevant to the state and which ones aren't. It acts as the basis for a new type of CRDTs, which we call tunable-CRDTs (t-CRDTs). t-CRDTs ensure that all replicas eventually reach the same state, regardless of how policies are specified, as long as they are deterministic. In t-CRDTs a policy is a simple boolean function that compares two operations and decides if one of them makes the other irrelevant.

To ease the difficulty of starting to use t-CRDTs, we provide both a specification and a Java implementation of a t-CRDT library which contains common data types and multiple policies for each one. As such, a programmer can always choose to use one or more of the pre-defined policies, which already offers more variability than state-of-the-art CRDTs and should be adequate for most application scenarios. For example, in the counter t-CRDT we support an operation to reset the counter to a certain value (*set value*), along with the typical *increment* and *decrement* operations and provide different policies to control if a *set value* should consider concurrent *increments*/*decrements* or ignore them, and also which value should be kept if multiple *set values* happen concurrently. Thus, this gives the programmer a varied yet precise concurrency control even if he doesn't specify his own policies. Nevertheless we also present a methodology to guide the programmer in specifying both new policies and t-CRDTs, if needed.

Since it is of utmost importance that the mentioned t-CRDTs behave correctly, we verified the correctness of both the generic model and of the t-CRDTs provided in our library. We demonstrate mathematically correctness properties of our model, namely that it converges and what are the network and policies requirements. As for the t-CRDTs, we built a TLA+ specification for each one in the library, including all provided policies and verified using TLC that data-type specific invariants are hold true in every possible situation, as well that convergence is always ensured.

We also evaluate the performance of the generic model and of t-CRDTs by comparing them with existing operation-based CRDTs in multiple execution scenarios and analyzing metrics such as execution time, space used and message size.

1.4 Contributions

The main contributions of this work are:

- A generic model which allows for a new type of CRDTs to be specified on top of it, in which operations can have user-defined policies to express precisely the intended concurrency semantics, while ensuring state convergence;
- A new type of CRDTs built on top of the generic model, which we call tunable-CRDTs (t-CRDTs);
- A library implementing t-CRDTs for multiple data types such as counters, registers, sets and maps. Each t-CRDT also provides multiple default policies, as well as allowing the programmer to define his own policies;
- A verification of the correctness and guarantees provided by both the generic model, the implemented t-CRDTs and their policies;
- A practical evaluation of the set t-CRDT and multiple state-of-the-art operation-based set CRDTs.

1.5 Document Structure

The rest of the document is organized as follows:

1. In Chapter 2 we present the necessary background and related work. First, we introduce the CAP theorem, along with concepts related with consistency and replication. Then we formally introduce CRDTs, along with some examples of existing CRDT data types.
2. In Chapter 3 we discuss the requirements for a generic CRDT model that provides customizable concurrency control. We also provide a model meeting those requirements that acts as the basis of t-CRDTs;
3. In Chapter 4 we present a t-CRDT library for various data types with multiple policies for each one, along with a methodology to specify new t-CRDTs and policies;
4. In Chapter 5 we prove and verify correctness properties of respectively our generic model and the library's t-CRDTs;
5. In Chapter 6 we discuss a practical performance evaluation of the set t-CRDT in comparison with operation-based set CRDTs;
6. In Chapter 7 we end this thesis with our conclusions and possible future work.

Chapter 2

Related Work

In this chapter related work to CRDTs is presented, in order to be able to understand how they work and which trade-offs they present. First, the famous CAP theorem is explained in detail, followed by two sections which overview, respectively, concepts on consistency and replication. Then, two sections on CRDTs are covered: the first one describes CRDTs in detail and the second one shows multiple examples of CRDTs known in the literature.

2.1 CAP theorem

The CAP (Consistency, Availability, Partition-tolerance) theorem [14] states that, in a distributed system, at most two of the following three properties can be provided simultaneously:

- Strong Consistency: a total order on all operations executed must be defined, and each operation must appear to have been completed in a single point of time (atomicity).
- Availability: all operations that are issued to non-faulty replicas (i.e., replicas that won't fail until the operation completes) must eventually complete.
- Partition-tolerance: the system must keep on executing operations despite partitions that might happen on the network. These partitions lead to unreliability on the communication between replicas in different partitions.

Due to referred impossibility result, every distributed system is only able to fully provide two of the three mentioned properties. As such, a distributed system can be classified as CA (Consistent, Available), CP (Consistent, Partition-tolerant) or AP (Available, Partition-tolerant). However, since network partitions will happen on systems that are distributed across multiple places, in practice most distributed systems are either CP or AP.

Even though it's not possible to fully provide the three referred properties simultaneously, it should be noted that AP systems can still implement *weak* forms of consistency, which might still provide useful guarantees about the states of the system that each client can observe. CRDTs are typically used in AP systems.

2.2 Consistency

In distributed systems, consistency models specify how can the replicas' state diverge due to the execution of write operations. Consistency models also restrict which states each client is allowed to observe when a read operation is executed. Despite the existence of multiple consistency models in the literature, for the majority of them it's possible to classify each one as providing either *strong consistency* or *weak consistency*.

2.2.1 Strong and weak consistency

2.2.1.1 Strong consistency

In strong consistency, the system provides to the clients the illusion that only a single replica exists, despite operations being executed on multiple replicas. This implies that a total order for all operations executed on the system must be defined, since all replicas must evolve their state equally.

Without a total order on the operations, it would be possible for a client to request a read operation on an object in one replica and, after the read is complete, request another read on the same object on a different replica and observe a state that is different, even if no write operations happen in the meantime. This can happen because replicas need to propagate the write operations to other replicas and, as such, it is possible that one write operation has already reached some replicas but not the others. However, with a defined total order for all the operations, this situation would not be possible and, as such, strong consistency can be provided.

Even though in strong consistency the state diverges for short periods of time, this divergence is hidden from the client due to each replica contacting multiple other replicas before confirming an operation to the client, in order to guarantee that each time an operation is confirmed, at least one of the contacted replicas always has the most recent state.

Multiple models that implement strong consistency have been proposed in the literature, with the main ones being serializability and linearizability [33].

2.2.1.2 Weak consistency

In weak consistency, the system doesn't need to provide the illusion that only a single replica exists, allowing the replicas' state to diverge. This means that a client, when requesting read operations to different replicas, may observe different states even if no

write operation happened between those reads. In fact, in some weak consistency models it is not even necessary for all replicas to have the same state eventually (i.e., they can diverge in state forever).

Despite not being necessary state convergence in order to have weak consistency, some weak consistency models do provide guarantees of eventually reaching the same state in all replicas, if no write operations happen for a long enough period of time. Such a consistency model is usually known as eventual consistency. Other consistency models, such as read your writes, provide restrictions on the states that a client can observe: in this case, when a client executes a read, he will observe the effects of his last write or of a more recent write.

Multiple models that implement weak consistency exist in the literature. Some examples are eventual consistency, causal consistency, causal+ and read your writes [7].

2.2.1.3 Strong versus weak consistency

Strong consistency provides guarantees that make it simple to reason about the evolution of the state of a replicated system when compared to weak consistency. This makes applications easier to develop when using strong consistency, since the state will evolve in strict steps even when executing operations on different replicas [7].

However, in order to have strong consistency, synchronization between replicas is required, since multiple replicas need to be contacted in order for an operation to be confirmed. This limits the fault-tolerance of the system and increases the latency on the execution of each operation. It also reduces the availability of the system, due to the CAP theorem [14].

On the other hand, weak consistency doesn't need synchronization between replicas, which allows faster execution of operations, better fault-tolerance and higher availability of the system, since only a few (or possibly only one, depending on the consistency model) replicas need to be contacted in order for an operation to be completed. With weak consistency, an operation can be confirmed to the client and then propagated between replicas asynchronously [7].

2.2.2 Eventual consistency

Eventual consistency is a form of weak consistency in which the only guarantee is that, if write operations stop occurring, eventually the state of the replicas of a distributed system will converge (i.e., reach a consistent state).

The referred guarantee is a rather weak one, especially when considering that, on most systems, write operations never stop occurring, which makes it difficult to understand how the system will behave. Another problem is that multiple systems who claim to offer eventual consistency also offer more guarantees besides convergence [7]. Moreover, eventual consistency doesn't hint on how should conflicts on state between different replicas (due to a concurrent write on the same object for example) should be resolved. All

these problems (among others not mentioned here) make it difficult to specify precisely the consistency model of a distributed system which provides a weak consistency model.

2.2.3 Semantic model

As seen in the previous section, eventual consistency is a rather broad form of weak consistency, with different systems offering different "forms" of eventual consistency. This leads to the necessity of having a way of precisely specifying the consistency model of systems who claim to offer eventual consistency, i.e., what are the guarantees offered by such systems.

One possible way to specify a weak consistency model is through the use of an adequate framework, such as the one presented by Burckhardt et.al. [7]. The mentioned framework allows for precise specification of the consistency model independently of the implementation, through the use of axioms. In this framework, in order to specify the consistency model, three components need to be specified:

- Replicated data type specifications: define which replicated data types are supported (ex: counters, sets, etc.) by the system and which policies are used to solve the conflicts between the states of the same object in different replicas (i.e., how are conflicting states "merged" into one).
- Consistency specification: define the consistency model provided by the system. This definition is done by combining axioms, with those axioms restricting how the state of the system can evolve and which guarantees the system provides. These axioms can either refer to a single object or relate different multiple objects.
- Interfaces for strengthening consistency: define how can the consistency of the system be strengthened for a specific operation or a group of operations. This allows to trade performance and fault-tolerance of the system for stronger consistency guarantees only when executing certain operations.

The following subsections contain an explanation on how can replicated data types and consistency be specified in the Burckhardt et.al.'s framework [7]. However, it won't be covered how can interfaces for strengthening consistency be specified, since those are out of the scope of this document.

2.2.3.1 Replicated data type specification

Specifying a replicated data type in a system providing weak consistency can be challenging when compared to a strongly consistent system. On a strongly consistent system, the semantics of a data type can be specified by a function that, given a nonempty sequence of operations, specifies the value returned by the last operation after applying all the previous ones, by the order in the sequence.

However, on weakly consistent systems, that isn't true. On those systems, operations can happen concurrently in different replicas. Since there is no synchronization between

replicas, operations can be received in different replicas in different orders, which might wield different results when applied in each replica. However, due to the definition of eventual consistency it is necessary that, after all write operations are delivered to each replica, the states be the same.

In order to guarantee the eventual converge of the states, a strategy to resolve conflicts is necessary. The strategies identified by Burckhardt et.al [7] are:

1. Make concurrent operations commutative, i.e., make it such that any order of application of the write operations results in the same final state. This can be used, for example, for counters.
2. Order concurrent operations, i.e., order all concurrent operations in some deterministic way (for example, by using timestamps). This can be used, for example, for a last writer wins register.
3. Flag conflicts, i.e., detect the conflicts and let the user solve them. This is used, for example, by the Dynamo key-value store [11].
4. Resolve conflicts semantically, i.e., detect the conflicts and take some action that depends on the data type and the type of operations in order to solve such conflict. This can be used, for example, in a observed-remove set (a set in which when an add and a remove happen concurrently on the same element, the add wins).

A replicated data type can be specified by defining a function which, for any operation, returns what should be the result. However, since the result of an operation may depend on the previous operations and their execution order (especially for solving conflicts using one of the mentioned strategies), this function needs to receive an *operation context*, as defined by Burckhardt et. al. [7].

Definition 2.1. *An operation context for a data type t is a tuple C = (f, V, vis, ar) where:*

- *f is the operation to be applied.*
- *V is the set of operation events of the form (e, g), where e is a unique event identifier and g is an operation. The set V includes all the operations that are visible to f (i.e., all the operations applied before in the replica in which f is being applied).*
- *vis ⊆ V x V is a visibility relation, which represents which operations are visible to each operation.*
- *ar ⊆ V x V is a total and irreflexive arbitration relation, which represents a total order of the operations.*

Using the information on definition 2.1, a function for describing any replicated data type can be specified. Examples of such specifications can be found in [7, 8].

2.2.3.2 Consistency specification

Through the use of the framework and its axioms, it's possible to define multiple forms of eventual consistency. For example, it can be used to specify basic eventual consistency

(which only guarantees that eventually the states converge), specify session guarantees (e.g., that a client's read must return the same value of that client's last write or a more recent one), specify ordering guarantees (e.g., causality) or even interfaces for stronger consistency on some operations. However, only the axioms related to the guarantees provided by CRDTs will be presented here.

Before presenting the axioms, the concepts of session, action, history and execution need to be introduced.

A session represents a client identity in his multiple requests for operations issued to the replicas of a system. A session may be used to satisfy some operation ordering guarantees (e.g., that a read after a write on the same object sees the effects of that write). Each session is represented by an unique identifier (s).

Definition 2.2. *An action is a tuple (e, s, x.f, k) where e is a unique action identifier, s is the identifier of the session in which the action takes place, f is an operation performed on object x and k is the return value of the operation.*

An action represents a client operation in the history of a system. It should be noted that f must be one of the operations supported by the data type of object x, and k one of the possible return values of the referred operation.

Definition 2.3. *An history is a pair (A, so) where A is a set of actions with no duplicate action identifiers and so $\subseteq$ A x A and respects the axiom SOWf.*

An history represents the set of actions (A) issued in a session by a client, together with a session order (so) that defines the order in which those actions were issued.

Definition 2.4. *An execution is a tuple X = (A, so, vis, ar) where (A, so) is an history, and vis, ar $\subseteq$ A x A, with vis respecting the axiom VISWf and ar respecting ARWf.*

An execution represents an history with two extra components: vis and ar. These components have similar meanings to the ones used for defining a replicated data type in Definition 2.1 with the difference that they relate actions instead of operations. However, even though multiple objects may be referred in vis and ar, each relation only relates two actions on the same object, as specified by the axioms VISWf and ARWf.

For defining what Burckhardt et.al. [7] classifies as basic eventual consistency, the following axioms are needed: SOWf, VISWf, ARWf, Rval, Eventual, ThinAir.

- SOWf: so is the union of transitive, irreflexive and total orders on actions by each session. Informally, it means that so defines the order of actions in a session. Note that so can relate actions of different objects.
- VISWf: $\forall a, b$, if a is visible to b (i.e., there's a relation in vis of a to b), then a and b are actions on the same object. Informally, this means that all the visibility relations are between two actions for the same object.

- ARWf: $\forall a, b$, if a is ordered before b (i.e., there's a relation in ar of a to b), then a and b are actions on the same object. Ar is transitive and irreflexive and $ar \cap (vis^{-1}(a)$ x $vis^{-1}(a))$ is a total order for all $a \in A$. Informally, this specifies that ar represents a total order for each action in A.
- Rval: $\forall a \in A$, $a.k$ is equal to the result of applying the function that describes the data type of the object $a.x$ with the operation context that can be extracted from a. This operation context can be obtained by using the operation defined in a ($a.x.f$), the actions visible to a according to vis ($vis^{-1}(a)$), and both vis and ar projected to $vis^{-1}(a)$. In a simpler way, Rval ensures that for each action (a) in an execution X, its return value ($a.k$) corresponds to the one obtained by applying the data type's specification function (see Section 2.2.3.1) with the operation context extracted from the execution (i.e., the data type behaves as expected).
- Eventual: $\forall a \in A$, $\neg(\exists$ infinitely many $b \in A$ such that $a.x$ and $b.x$ are the same object and a is not visible to $b)$. Informally, this ensures that an action cannot be invisible to infinitely many other actions on the same object, which forces that eventually all actions become visible in every replica.
- ThinAir: $so \cup vis$ is acyclic. This prevents some counter-intuitive situations that are allowed by Rval and Eventual, but not by ThinAir. The name of ThinAir is due to the fact that the situations prevented by this rule correspond to read operations that return a value without having yet seen the write for such value (due to a speculative computation, for example), which makes it look like the value came from "thin air".

Rval and Eventual together guarantee that, eventually, every replica reaches a consistent state. ThinAir is also used in order to avoid some strange behaviours, as described above. SOWF, VISWF and ARWF are used to specify the properties of, respectively, the relations so, vis and ar.

By using more axioms it is possible to define stronger consistency models. One such model is the per-object causal consistency which, informally, guarantees that all users see the operations that are causally related by the same order.

Formally, per-object causal consistency is the combination of, at least, basic eventual consistency and two extra axioms: POCV and POCA. In order to be able to specify these axioms, an extra auxiliary relation needs to be introduced: $hbo = ((so \cap sameobj) \cup vis)^+$, where $sameobj$ means that the objects referred by two actions must be the same. This means that $so \cap sameobj$ will return all relations in so in which both actions refer to the same object. Hbo represents the per-object causality order, also known as happens-before relationship.

- POCV (Per-Object Causal Visibility): $hbo \subseteq vis$. This property ensures that an operation sees all the operations on the same object that causally affect it (for example, a read sees all the operations that causally happened before it and, as such, returns a state that reflects the effect of those operations).

- POCA (Per-Object Causal Arbitration): $hbo \subseteq ar$. This axiom guarantees that the arbitration relation ar must respect causality (i.e., the total order defined by ar must respect causality order).

A system specification can be considered as the set of histories possible for a system. Since axioms restrict the set of executions possible for a system, they effectively restrict the set of histories that the system can produce. The more axioms are chosen for a system, the less histories will be possible and, thus, the stronger the consistency model will be.

2.3 Replication

In order for a distributed system to always have all of its data accessible despite the occurrence of the failures described in the system's fault model, it is necessary to replicate the data and keep it up-to-date in multiple replicas. How the replication should be done depends on multiple factors, namely on the consistency model of the system, as described in Section 2.2, on the usage of the system (ratio of read/write operations, frequency of operations on the same object, etc.) and on the fault model of the system.

2.3.1 Synchronous and asynchronous replication

In synchronous replication, when an operation is issued by a client in a replica, multiple other replicas are first contacted before the operation is confirmed. Typically synchronous replication is used by systems that provide strong consistency since they need to provide the illusion of a single replica. Due to the possibility of concurrent requests in different replicas, normally mechanisms such as Paxos [21] or Total Order Broadcast [32] are used for totally ordering the operations.

On the other hand, asynchronous replication is normally used on systems that provide weak consistency, since on these systems an operation can be executed immediately on the source replica and confirmed to the client without needing synchronization with the remaining replicas. The operation should then be propagated in the background.

Asynchronous replication usually allows for a bigger throughput of operations per unit of time when compared to synchronous replication, since a client request doesn't need to wait for multiple replicas to reply before the client gets the result. However, with asynchronous replication, the replicas take longer to reach the same state and it's possible that their states become inconsistent for extended periods of time, which may lead to unexpected results for the clients when executing read operations on different replicas, as discussed in Section 2.2.1.2.

On some systems, before an operation is confirmed, the operation is propagated to a small group of replicas and only after those reply the operation is confirmed. The operation is propagated to the other replicas either at the same time but without waiting for a reply or after the operation is confirmed. For these type of systems, the replication can be classified as being synchronous to a small group and asynchronous to the rest of

the replicas. Such systems tend to converge sooner than fully asynchronous systems, but nevertheless both provide weak consistency only.

2.3.2 State-based and Operation-based replication

In state-based replication [27], the source replica propagates the complete state of the objects that it is replicating to the other replicas. Depending on the semantics of the consistency model, the receiving replicas may need to merge the received state with their own state, in order to guarantee that the states will converge eventually and that no operation's effects are lost.

In operation-based replication [27], a replica propagates the operations instead of the object's state. The receiving replicas will then need to execute the operations, in order to reach the same state. However, depending on the semantics of the consistency model, a total (or partial) order of the operations may be needed in order to reach a consistent state.

Generally, operations are smaller than a whole object state and, as such, operation--based replication results in smaller messages compared to state-based replication. However, if the frequency of operations is high, multiple messages may need to be sent frequently and each replica will have to execute multiple operations, which may lead to a system overload. In these cases, it may be more efficient to propagate the state.

Some optimizations can be done for both approaches. On operation-based, multiple operations can be sent in one message. On state-based, it is possible to send differences (delta) in state instead of sending the whole state, which results in smaller messages at the expense of a possibly more complex algorithm for merging states.

2.4 CRDTs

The Conflict-Free Replicated Data Types (CRDTs) [27, 28] consist in a group of data types designed to be used in large-scale distributed systems. Each CRDT represents a mutable object, which is replicated in multiple processes (replicas). In order for clients to consult or manipulate the state of a CRDT, each type of CRDT offers an interface for manipulating its object: the interface consists in a group of operations which allow to either query the object state or update it.

In the remaining of this section, multiple aspects of CRDTs are explained in detail. First, the system model and client interaction model are explained. Then, the consistency guarantees and correctness criteria of CRDTs are explained. Afterwards, the two main "flavors" of CRDTs are described: state-based CRDTs and operation-based CRDTs. Subsequently, variants of operation-based and state-based CRDTs called, respectively, pure op-based and δ-CRDTs are presented. Finally, to end this section, a small comparison between the referred approaches is given.

System model Each CRDT is considered to be replicated in a fixed set of replicas connected by an asynchronous network. The network can partition and recover and each process may crash and recover at any moment. However, if a processes recovers, the contents of its memory will be the same as before the crash. The processes don't exhibit byzantine behaviour. The replicas communicate between themselves often enough in order to guarantee that every replica eventually sees the effect of every operation that changes the state of the CRDTs that they are replicating.

Interaction model A client can request operations for a CRDT through its interface in any arbitrary replica (called the source replica [27]). A query doesn't change the object state and is executed locally: this operation doesn't even need to be propagated. On the other hand, update operations modify the object state and, as such, need to eventually be propagated to the remaining replicas. Nevertheless, an update operation is able to complete successfully in only one replica (i.e., a result can be returned by just executing it locally), as long as it is guaranteed that, eventually, the remaining replicas receive either the operation or its effects, meaning that the propagation of state changes to the other replicas can be done asynchronously.

2.4.1 Consistency guarantees

All CRDTs provide at least basic eventual consistency, as defined by Burckhardt et.al. [7] and explained in Section 2.2.3.2. With basic eventual consistency, it is ensured that two replicas who have executed the same set of updates will converge states eventually. However, all CRDTs are able to provide a stronger notion of state convergence: they are, in fact, able to ensure that any two replicas who have executed the same set of updates have equivalent states. This stronger property, which is known as strong eventual consistency (SEC) [28] is important, because it guarantees that no external mechanism for state reconciliation is needed and that the states are equivalent in two replicas as soon as their set of executed operations is equivalent.

Because CRDTs provide a type of eventual consistency and update operations can happen at any replica, it is possible for conflicting situations to happen. For example, in a set, one replica can add the element a while another replica, concurrently, removes the element a. However, one key property of CRDTs is that they automatically solve these conflicts, by applying one of the strategies explained in Section 2.2.3.1, without requiring intervention of the client.

Some CRDTs also provide, besides SEC, causality guarantees for the object they represent, known as per-object causal consistency, as defined in Section 2.2.3.2. In other words, those CRDTs provide the consistency model resulting of combining both SEC and per--object causal consistency. In the following subsections, it will be indicated which CRDTs are able to offer this combined consistency model or not. It should be noted that even

with this consistency model, the conflicting situations referred before can still happen and CRDTs will still automatically solve them.

2.4.2 Correctness criteria

CRDTs typically represent data types, such as a counter, a set, a map, etc. Intuitively, any CRDT that represents a data type should try to behave as closely as possible to the sequential specification of that data type, despite operations being executed concurrently.

In order to catch the mentioned intuition, we introduce three concepts:

Definition 2.5 (Principle of Sequential Equivalence (PSE)). *If any two update operations are executed sequentially on a replicated data type on state* S*, then the resulting state* S' *should conform to the sequential specification.*

Definition 2.6 (Principle of Permutation Equivalence (PPE)). *If any two update operations lead to the same resulting state* S' *from state* S*, independently of the order of execution in the sequential specification, then the concurrent execution of those updates on a replicated data type must also transit from state* S *to state* S'*.*

Definition 2.7 (Principle of Conflicting States (PCS)). *If any two update operations lead to different states* S', S" *from state* S *depending on the order of execution, then the resulting state in a replicated data type should be: (i) either state S' or S"; (ii) a state with an "error mark".*

The Principle of Sequential Equivalence (PSE) represents the intuition that executing update operations sequentially in a replicated data type should have the results defined by the sequential specification of that data type. On the other hand, the Principle of Permutation Equivalence (PPE), introduced by Bieniusa et. al. [6], states that if the execution of two update operations lead to the same result *S'* in a sequential data type independently of the order of execution, then the concurrent execution of those should also lead to *S'*. In the literature, two operations that lead to the same result independently of the order of execution are said to be non-conflicting or commutative. If the operations are commutative, then intuitively their order of execution shouldn't matter.

The Principle of Conflicting States (PCS) is less intuitive when compared to the other two. However, the key idea is that when conflicting concurrent operations (for example, a remove and add of element e in a set) are executed, the result should be "something that makes sense". For the referred example, either the case in which the element e is in the set (add executed last) or e isn't in the set (remove executed last) makes sense, however, it wouldn't make sense for element g (with $e \neq g$) to be removed due to the execution of those two operations. Note that a state representing an error (for example, a boolean stating that a conflict happened) is also acceptable.

A CRDT is considered to be *sequentially conformant* if its specification follows PSE, PPE and PCS.

2.4.3 State-based CRDTs

In a state-based CRDT, also known as Convergent Replicated Data Type (CvRDT) [27], the effects of update operations are propagated by sending the local state (payload) of a replica to another replica. The receiving replica then merges the remote state with its own local state, which results in a more up-to-date state. The resulting state has the effects of all the updates that were present in both states that were merged.

Algorithm 2.1 Specification of a state-based object

```
1:  payload Payload type
2:      initial Initial value
3:  query Query (arguments) : returns
4:      pre Precondition
5:      let Evaluate at source, synchronously, no side effects
6:  update Update (arguments) : returns
7:      pre Precondition
8:      let Evaluate at source, synchronously, no side effects
9:      Side-effects at source to execute synchronously
10: compare (value1, value2) : boolean b
11:     let Is value1 ⊑ value2 in join semi-lattice?
12: merge (value1, value2) : payload mergedValue
13:     let LUB merge of value1 and value2
```

A state-based object can be specified as shown in Algorithm 2.1. *Payload* represents the current state of the object and initially has the value *initial*, which is the same in all replicas. *Query* represents a query operation, which reads the state without modifying it. *Update* represents an update operation, which modifies the state. *Compare* and *merge* are special operations that receive two states (the local state and a remote state, received through the network) and return, respectively, the result of comparing or merging both states. All of the referred operations execute locally at the source replica. Some examples of state-based CRDTs specifications can be found in Section 2.5.

Each update and query operation may, optionally, receive *arguments*, return (*returns*) values and have pre-conditions (*pre*) for execution. If an operation has a pre-condition, it is only executed if such pre-condition is enabled (i.e., true) on the source replica. Note that multiple query and update operations can be defined for the same object. A *let* statement doesn't change the state and allows to define and update local variables, by using the symbol =. Changes to state are done without a *let* statement by using the symbol :=.

In order to precisely define what a state-based CRDT is, the concepts of causal history [27], least upper bound, join semi-lattice and monotonic join semi-lattice need to be introduced.

Definition 2.8 (Causal History (state-based)). *For any replica x_i of the state-based object x, its causal history $C(x_i)$ can be defined as:*

- *Initially,* $C(x_i) = \emptyset$.
- *After executing* update *operation* f, $C(f(x_i)) = C(x_i) \cup \{f\}$.
- *After executing* merge *of states* x_i *and* x_j, $C(merge(x_i, x_j)) = C(x_i) \cup C(x_j)$.
- *After executing a* query *operation or* compare *of states* x_i *and* x_j, *the causal history remains the same* ($C(x_i)$).

Intuitively, a replica x_i's causal history represents the set of update operations that were executed in order to reach x_i's current state. In order to guarantee eventual convergence of states, it is necessary that every update operation reaches the causal history of every replica. Such propriety can be achieved if an underlying system with the following properties is assumed: (i) it transmits states between pairs of replicas at unspecified times, infinitely often; (ii) replica communication forms a connected graph.

The happens-before relation between operations f and g is defined as: $f \prec g \Leftrightarrow C(f) \subset C(g)$. I.e., if all operations in $C(f)$ are in $C(g)$, then f happened before g. If neither $f \prec g$ nor $g \prec f$, then f and g are concurrent operations ($f \parallel g$).

Definition 2.9 (Least Upper Bound (LUB)). *$m = x \sqcup y$ is the least upper bound of $\{x, y\}$ under partial order $\sqsubseteq$ iff:*

- $x \sqsubseteq m \wedge y \sqsubseteq m$.
- $\forall m'$: $x \sqsubseteq m' \wedge y \sqsubseteq m'$ *then* $m' \sqsubseteq m$.

The operator for calculating a LUB can be represented as $\sqcup$. Informally, given a partial order $\sqsubseteq$, a LUB m of $\{x, y\}$ represents the value which is partially ordered right after both x and y. For example, if we assume that the partial order is the increasing order ($\leq$) defined for natural numbers and consider $x = 5$ and $y = 8$, then $m = 8$ is the LUB. However, $m' = 9$ isn't a LUB, since $5 \leq 9$ and $8\ (y) \leq 9$ but $8\ (m) \leq 9$.

From the definition of LUB, it is possible to conclude that $\sqcup$ is commutative, idempotent and associative.

Definition 2.10 (Join Semi-lattice). *An ordered set S of object payloads and a partial order $\sqsubseteq$ that compares these states form a join semi-lattice iff, $\forall x, y \in S$, $x \sqcup y$ exists.*

Definition 2.11 (Monotonic Join Semi-lattice). *A state-based object is a monotonic join semi-lattice iff:*

1. *The ordered set S of object payloads and its partial order $\sqsubseteq$ form a join semi-lattice.*
2. *Every* update *operation applied on the object inflates upwards the state according to $\sqsubseteq$.*
3. *The* merge *operation of two states computes their LUB.*

Informally, a monotonic join semi-lattice states that, given a partial order $\sqsubseteq$, the set of possible payloads of an object and a $\sqcup$ which calculates a LUB : (i) any two payloads of that set can be compared; (ii) a LUB of any two payloads of that set can be calculated;

(iii) any update operation applied to any payload of the set inflates upwards the payload following the payload order defined by $\sqsubseteq$.

A state-based object that is a monotonic join semi-lattice is a state-based CRDT, or CvRDT. Regarding the specification of a CvRDT, it is required that *compare*(x, y) returns $x \sqsubseteq y$ and that two states are equivalent iff $x \sqsubseteq y \wedge y \sqsubseteq x$. It is also required that *merge*(x, y) calculates the LUB of x and y according to $\sqcup$ and that merge can always be executed (i.e., doesn't have any pre-conditions).

Theorem 2.1. *Assuming that the system transmits payload infinitely often between pairs of replicas over eventually-reliable point-to-point channels, and that all operations eventually terminate, then two replicas of a CvRDT eventually converge according to SEC.*

The proof for the theorem above is presented in [27, 28].

State-based CRDTs also provide, besides SEC, per-object causal consistency. Burckhardt et. al. [8] proves that state-based replicated data types provide per-object causal consistency, which implies that state-based CRDTs provide per-object causal consistency.

2.4.4 Operation-based CRDTs

In an operation-based CRDT, also known as op-based CRDT or Commutative Replicated Data Type (CmRDT) [27], a replica propagates the update operations that are invoked on it to the other replicas, so that every replica executes the same operations and reaches the same state.

Algorithm 2.2 Specification of an op-based object

```
1:  payload Payload type
2:      initial Initial value
3:  query Query (arguments) : returns
4:      pre Precondition
5:      let Evaluate at source, synchronously, no side effects
6:  update Update (arguments) : returns
7:      atSource (arguments) : returns
8:          pre Precondition at source
9:          let 1st phase: evaluate at source, synchronously, no side effects
10:     downstream (arguments passed downstream)
11:         pre Precondition against downstream state
12:         2nd phase: side-effects to downstream state to execute asynchronously
```

An op-based object can be specified as shown in Algorithm 2.2. The *payload*, *initial* and *query* have the same meanings as in state-based objects. An update operation is represented by the *update* keyword. Here, *merge* and *compare* operations are not needed, since the replicas propagate operations instead of states. As in state-based objects, multiple *query* and *update* operations can be defined and both can, optionally, receive arguments,

return values and have pre-conditions (*pre*). Some examples of op-based CRDTs specifications can be found in Section 2.5.

Unlike in state-based objects, in an op-based object an update operation has two parts. The first part, labeled *atSource*, is only executed at the source replica and if its pre-conditions are enabled. It receives the arguments directly from the update operation. The goal of this part is to prepare the necessary arguments (if any) for the second part of the operation. The first part is not allowed to change the state (i.e., no side-effects) but can, however, compute results that may be returned to the caller.

The second phase, labelled *downstream*, executes after the *atSource* phase at all replicas. At the source replica, it is executed right after *atSource*, while on the remaining replicas it is executed asynchronously when they receive the operation. This phase receives the arguments prepared by the *atSource* phase, is only executed if the pre-conditions on the downstream's state are true and updates the downstream's state. This phase is not allowed to return any results.

With the goal of precisely defining what an op-based CRDT is, the concepts of causal history (op-based) [27] and commutativity need to be introduced.

Definition 2.12 (Causal History (op-based)). *For any replica x_i of the op-based object x, its causal history $C(x_i)$ can be defined as:*

- *Initially, $C(x_i) = \emptyset$.*
- *After executing the* downstream *phase of* update *operation f, $C(f(x_i)) = C(x_i) \cup \{f\}$.*
- *After executing a* query *operation or the* atSource *phase of an* update *operation, the causal history remains the same ($C(x_i)$).*

A replica x_i's causal history represents the set of update operations that were executed in order to reach x_i's current state. In order to guarantee eventual convergence of states, it is necessary that every update operation reaches the causal history of every replica. Such propriety can be satisfied by an underlying system which implements reliable broadcast of every update to every replica following some order $<_d$ (delivery order) where the downstream precondition for each update is true.

The happens-before relation is defined in the same way as for the state-based case. A delivery order which respects the happens-before relation (defined as causal delivery, $<_{\rightarrow}$) always satisfies the downstream preconditions and makes sure that non-concurrent operations are executed by the same order in every replica and, as such, don't need to commute. Concurrent operations need to be commutative though.

Definition 2.13. *Operations f and g commute, iff for any reachable replica state S where their source pre-condition is true, the source precondition of f (respectively g) remains enabled in state $S \bullet g$ (resp $S \bullet f$), and $S \bullet f \bullet g$ and $S \bullet g \bullet f$ are equivalent states.*

Intuitively, two operations are commutative if, for any state in which the pre-conditions for both operations is enabled, applying them in different orders results in the same state (and no pre-condition is violated).

For some data types, a weaker delivery order than causal delivery $<_{\rightarrow}$ may be enough. However, in that case, more pairs of operations are concurrent (since more pairs of operations may be delivered in different orders in different replicas) and, as such, more pairs of operations need to be proven commutative.

An op-based object in which all concurrent operations commute is an op-based CRDT, or CmRDT.

Theorem 2.2. *Any two replicas of a CmRDT eventually converge according to SEC under reliable broadcast channels that deliver operations in delivery order $<_d$, assuming that all operations eventually terminate.*

The proof for the theorem above is presented in [27, 28].

It is worth noting that, if the delivery order $<_d$ provided by the underlying system is at least as strong as causal delivery $<_{\rightarrow}$, then op-based CRDTs provide both SEC and per-object causal consistency. Causal delivery ensures that operations are delivered and thus executed according to the happens-before relationship (causality) in every replica, which is enough to provide per-object causality. If the delivery order is weaker than causal delivery, only SEC is provided.

2.4.5 Pure op-based CRDTs

Pure op-based CRDTs [3] consist in a variant of op-based CRDTs in which the communication is done with "pure" operations, i.e., without being preprocessed in the source replica. As such, in pure op-based CRDTs, there is no *atSource* phase and operations are propagated directly to downstream replicas without any transformation [3].

The key insight of pure op-based CRDTs is that, in non-pure op-based CRDTs, in order to deal with concurrency conflicts (i.e., operations that don't commute naturally) additional information needs to be associated with the operation. This extra information, in practice, is used to be able to distinguish causally related operations from concurrent ones. The issue is that this information often needs to be stored as metadata in the CRDT's state, which may be a waste of unnecessary space if we assume causal delivery [3].

As such, Baqueiro et. al. suggest having the communication infrastructure expose the causality relation between operations, that is, to provide a way to know if two operations are causally related or are concurrent. They also propose a generic solution for the downstream phase, with this solution being usable in any pure op-based CRDT. In this type of CRDTs the state has two components [3]: (i) a partially ordered log of operations (POLog); (ii) a representation of the data type without metadata to represent causality.

Initially new operations are stored in the POLog. However, as more recent operations get applied, they may cancel old operations that are no longer relevant (e.g.: an *add* cancels

any *add* or *remove* that happened-before it for the same element). Canceled operations are removed from the state, which helps on keeping the state size small. The authors [3] propose to remove the causality metadata from some operations after it is ensured that no more operations concurrent to those will arrive, thus further reducing the state size.

While pure op-based CRDTs may be in certain situations more efficient than standard op-based CRDTs in terms of communication and storage overhead, they also have disadvantages. First, the programmer must supply his own functions [3] that compare operations in order to decide if one operation cancels another, which may not be easy to specify. Also the concurrency semantics are inherited from the mentioned functions, thus making solutions based on totally ordering operations such as a Last-Writter-Wins Register (see Section 2.5.2) difficult and unintuitive to specify as a pure op-based CRDT. Another concern is in terms of throughput when multiple operations exist in the state, as the authors don't include any performance analysis on the paper [3]. Finally, compared to the goals of this thesis as introduced in Section 1.2, pure op-based CRDTs aren't a possible solution as the programmer wouldn't be able to use different policies in the same CRDT - the same policy would have to be used during the lifetime of the CRDT.

Pure op-based CRDTs provide SEC, as these type of CRDTs require a system that delivers operations according to causal delivery.

2.4.6 Delta-based CRDTs

δ-CRDT, also known as Delta-State Conflict-Free Replicated Data Type [2], are a type of state-based CRDTs in which updates are propagated by sending deltas of the state which represents the effects of the most recent update operations, instead of shipping the whole state. Just as in state-based CRDTs, δ-CRDTs also use a merge operation to achieve eventual convergence, but instead of merging whole states they merge deltas into the local state of a replica.

In δ-CRDTs, update operations are replaced by delta-mutators. A delta-mutator is an update operation which modifies the state and that, instead of returning the whole state as in state-based CRDTs, or an operation as in op-based CRDTs, returns a delta-state. This delta-state represents the changes on the state originated by the delta-mutator and can be joined together in groups, called delta-groups. A delta-group is a delta-state that results from combining delta-states resulting from a group of delta-mutators.

As proven by Almeida et. al. [2], δ-CRDTs may provide only SEC or SEC and per--object causal consistency, depending on how the underlying system propagates delta--groups between replicas.

2.4.7 Comparison

As discussed previously, even though all CRDTs types provide SEC and can, potentially, also provide per-object causal consistency, there are advantages and disadvantages for each type of CRDT.

State-based CRDTs require weak assumptions of the network and allow for an unknown number of replicas. All the information of the object is present in the state, which makes them simple to reason about. However, for some data types, the state tends to grow indefinitely, which may lead to large states that need to be sent through the network, which can be inefficient [27].

Op-based CRDTs, on the other hand, only send small messages through the network. Typically, the state is smaller compared to state-based CRDTs, because some of the state is offloaded to the ordering and delivery guarantees of the network. However, op-based CRDTs require reliable broadcast which propagates the operations in a particular delivery order, which generally requires tracking group membership and, as such, may make it more difficult to add new replicas to the system. They can also be more difficult to specify due to requiring reasoning about history [27].

Pure op-based CRDTs have advantages similar to op-based CRDTs. They often require smaller messages than op-based CRDTs and on the best case scenario their state is smaller. Unfortunately pure op-based CRDTs always require causal broadcast [3], thus having similar or higher network requirements than op-based CRDTs. Another issue is that the state stores operations and their causal metadata, with operations only being removed under certain conditions. Also the authors don't present any evaluation on pure op-based CRDTs' performance, which rises concerns about their throughput.

δ-based CRDTs tackle the main disadvantages of both state-based CRDTs and op--based CRDTs. They have the advantage of not needing reliable broadcast according to a given delivery order, which means less guarantees from the network. The other main advantage is that they propagate deltas instead of whole states, which result in smaller messages compared to state-based CRDTs. However, the main problem of δ-CRDTs is that it can be challenging to specify them for non-trivial data types [2].

2.5 Examples of CRDTs

In this section, examples for both basic data types, such as counters or sets, and advanced data types, such as JSON, are presented. In either case, the main goals and challenges of specifying those data types as CRDTs is described. For the basic data types, specification for either (and sometimes, both) state or operation based CRDTs is also given.

For the specifications given in this section, the main goal is to present a version of a CRDT with a specification as simple as possible, while also being *sequentially conformant* (see Section 2.4.2) with the data type semantics. However, this means that those specifications, if implemented directly, may not be efficient and, as such, more efficient CRDTs for the same data types may exist.

In order to prove that a specification corresponds to a state-based or op-based CRDT it is needed to, respectively, prove that the possible states form a monotonic join semi--lattice and merge computes a LUB or that a delivery order which respects the downstream preconditions exist and that concurrent updates commute. Those demonstrations won't,

however, be included in this document as they are out of the scope and have been done already in related work.

2.5.1 Counter

A counter is an integer which supports three type of operations: *increment* and *decrement* which are update operations and *value*, a query operation that returns the number of increments minus the number of decrements. For simplicity reasons, it will be assumed that no overflows or underflows can happen.

Algorithm 2.3 op-based Counter CRDT

```
1:  payload integer v
2:      initial 0
3:  query value () : integer r
4:      let r = v
5:  update increment ()
6:      atSource ()                          ▷ Nothing to do at source
7:      downstream ()        ▷ No pre condition: any order for delivery is acceptable
8:          v := v + 1
9:  update decrement ()
10:     atSource ()                          ▷ Nothing to do at source
11:     downstream ()        ▷ No pre condition: any order for delivery is acceptable
12:         v := v - 1
```

An op-based CRDT counter is relatively easy to specify. Its specification can be found in Algorithm 2.3. Because increments and decrements commute, they can be applied at any order in each replica and still converge to the correct value, as long as every operation is delivered exactly once. As such, this is indeed an op-based CRDT.

Unlike the op-based CRDT counter, a state-based counter isn't as easy to specify. Intuitively, one could believe that having the payload as being one integer and the merge as a max of two states would work as expected, at least for an increment-only counter. However it doesn't. For example, if two replicas each incremented the counter and then a merge was executed, the resulting state would be 1, instead of the expected 2. If we, however, did a sum, the merge operation would no longer be idempotent (and, thus, wouldn't be a state-based CRDT) and could still result in wrong states. Both situations are represented schematically in Figure 2.1.

One possible alternative is to specify an increment-only counter in which, instead of having only one integer as the payload, the payload is a vector of integers, with one entry per replica. If each replica only increments its own entry, the *merge* operation computes the max for each integer and *value* returns the sum of all integers, then this solution works as expected. This solution is known as G-Counter [27], and its state-based specification can be found in [27].

However, the G-Counter doesn't work if it is intended to support both increments and decrements simultaneously, since decrements would be lost when doing the max on *merge*.

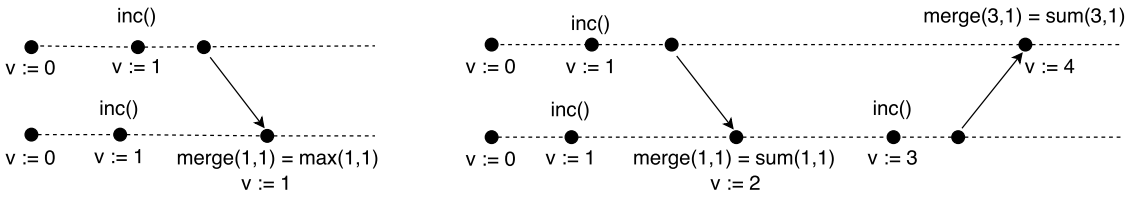

Figure 2.1: Two replicas concurrently incrementing a shared state counter. On the left, the merge is done with *max*, while on the right it's with *sum*. Both result in wrong states.

Algorithm 2.4 state-based PN-Counter CRDT

```
1:  payload integer[n] P, integer[n] N
2:      initial [0, 0, ..., 0], [0, 0, ..., 0]
3:  query value () : integer r
4:      let r = ∑_0^{n-1} P[i] - ∑_0^{n-1} N[i]
5:  update increment ()
6:      let g = myID()          ▷ myID(): returns the unique identifier of the source replica
7:      P[g] := P[g] + 1
8:  update decrement ()
9:      let g = myID()          ▷ myID(): returns the unique identifier of the source replica
10:     N[g] := N[g] + 1
11: compare (X, Y) : boolean b
12:     let b = (∀i ∈ [0, n - 1] : X.P[i] ≤ Y.P[i] ∧ ∀i ∈ [0, n - 1]: X.N[i] ≤ Y.N[i])
13: merge (X, Y) : payload Z
14:     let ∀i ∈ [0, n - 1] : Z.P[i] = max(X.P[i], Y.P[i])
15:     let ∀i ∈ [0, n - 1] : Z.N[i] = max(X.N[i], Y.N[i])
```

A simple solution is to have two integers per replica: one integer represents the amount of increments (increment integer, P) and the other the amount of decrements (decrement integer, N). The merge simply computes the max of each integer. The *value* operation returns the sum of all increment integers minus the sum of all the decrement integers. The specification for this data type is known as PN-Counter (Positive-Negative) [27] and its state-based specification is shown in Algorithm 2.4. Proof that this specification corresponds to a state-based CRDT can be found in [27].

Besides the overflow and underflow assumptions, the PN-Counter also assumes that the set of replicas is well-known. However, the same assumptions are done by the op--based CRDT counter. For the PN-Counter, it could be possible to add an extra update operation which represents the addition of a new replica (that operation would add an extra entry to the vector). The removal of a replica would be more difficult to do, however.

Both the PN-Counter and the op-based Counter CRDTs are *sequentially conformant* with the sequential counter specification. The strategy applied for solving concurrency conflicts is, in both cases, by requiring that every concurrent operation commutes (strategy one, as explained in Section 2.2.3.1). Also, for both of those CRDTs, it is possible and fairly simple to extend the interface to support additions or subtractions of values

given through the arguments of those operations (for the state-based one, it should be assumed that the values are non-negative). These extensions can be specified based on, respectively, the increment and decrement operations.

2.5.2 Register

A register is used to store an opaque object of any type. It only supports two operations: *assign*, which is an update operation that changes its value, and *value*, a query operation that returns its value.

By default, two concurrent *assigns* do not commute. The two main options for making them commute is to either define one as having precedence (and thus, one of the *assigns* is effectively lost) or to retain both values in the register. The registers that represent both approaches are known as, respectively, LWW-Register (Last-Writer-Wins) and MV-Register (Multi-Value) [27]. The MV-Register won't be discussed in this document, however a specification and description for a state-based MV-Register CRDT can be found in [27].

The key idea of the LWW-Register is to define a total order for all the *assigns*. *Assigns* done by the same replica are already causally ordered and, as such, the difficulty lies on ordering concurrent assigns. One way of achieving this is to associate to each *assign* a timestamp. Timestamps must be unique, consistent with causal order (i.e., if an *assign* causally happened before another *assign*, then the timestamp of the first must be smaller than the second's) and totally ordered. A possible implementation of those timestamps is with a vector of Lamport's logical clocks [17], with one entry per replica.

Algorithm 2.5 state-based LWW-Register CRDT

```
1:  payload X x, timestamp t                          ▷ X: any object type
2:      initial ⊥, 0
3:  query value () : X r
4:      let r = x
5:  update assign (X v)
6:      x, t := v, now()          ▷ now(): returns a timestamp consistent with causality
7:  compare (W, Y) : boolean b
8:      let b = (W.t ≤ Y.t)
9:  merge (W, Y) : payload Z
10:     if W.t ≤ Y.t then let Z.t, Z.x = Y.t, Y.x
11:     else let Z.t, Z.x = W.t, W.x
```

Algorithm 2.5 represents the specification for the state-based LWW-Register. An op-based specification can be found in Appendix A.1. Proof that both are, respectively, state-based and op-based CRDTs can be found in [27].

A register is a good example of a situation in which more than one strategy for solving conflicts might be a good option. The MV-Register solves conflicts by keeping all concurrent values and then letting the user define a new value (in a write ordered afterwards), which represents strategy three as explained in Section 2.2.3.1. On the other hand, the

LWW-Register solves conflicts by totally ordering all the operations, which corresponds to strategy two.

Both the LWW-Register and the MV-Register are *sequentially conformant*. Note that both CRDTs deal with PCS in different ways: LWW-Register chooses one of the states *(S', S")* according to the timestamps, while MV-Register puts an "error mark" by having multiple values in the register.

2.5.3 Set

Sets are collection types in which each element (or object) is present at most once (i.e., there's no duplicates). They are the building blocks of more advanced data types, such as maps or graphs, and are used by many different applications.

A set normally has, at least, the operations: (i) *add*(*e*); (ii) *remove*(*e*); (iii) *lookup*(*e*); (iv) *elements*(). Both *add*(*e*) and *remove*(*e*) are update operations which, respectively, add and remove element *e* to (resp. from) the set; *lookup*(*e*) is a query operation that returns true if the set has the element *e* and *elements*() is a query operation that returns all the elements that are in the set.

Unfortunately, in a sequential set not all pairs of update operations commute. An *add* and a *remove* commute as long as the elements they refer are different (i.e., $\forall e, f$, *add*(*e*) and *remove*(f) commute iff $e \neq f$). Two *adds* (resp. two *removes*) also commute, even if on the same element. However, an *add* and a *remove* on the same element doesn't commute, as the final result depends on their execution order. More precisely, if the *remove* is applied last, the element won't be in the set; if the *add* is applied last, the element will be in the set.

As such, the main difficulty of specifying a CRDT set is on how concurrent adds and removes on the same element are dealt with. Multiple strategies (as defined in Section 2.2.3.1) for solving these conflicts can be applied. The easiest one is to simply not support the remove operation, i.e., a Grow-Only Set (G-Set) [27]. However, if we intended to support both adds and removes, possible strategies include defining a total order (strategy two) for concurrent operations, or giving priority to one of the operations (strategy four). When priority is given to the add (resp. remove), it is said that the set is add-wins (resp. remove-wins).

Multiple variants of CRDT sets are present in the literature, however, we'll only focus on two variants: the 2P-Set and the OR-Set. The 2P-Set is described in the Appendix A.2.1 and we recommend its reading before proceeding to the OR-Set.

2.5.3.1 OR-Set

Even though for some use cases the 2P-Set may be adequate, its specification doesn't allow for an element to be added again after being removed, unlike in a sequential set. For most applications, such restriction may be unacceptable.

The key problem on letting an element to be re-added is on ensuring an order for concurrent adds and removes on the same element: if in one replica a concurrent remove and add results on the element staying in the set, then in every other replica the result must be the same, otherwise convergence is not guaranteed.

A naïve solution for the referred problem would be to associate a counter to each element. However, the resulting semantics are counter-intuitive: if two replicas concurrently remove an element, then in order for it to be re-added two adds would be needed [27].

Another alternative is to define a total order of the operations on the same element: this can be achieved by, on *add* and *remove*, associating a timestamp to the element being added/removed, as in a LWW-Register. If, for a given element, the highest timestamp is for a remove, then it isn't in the set. Otherwise, it is in the set. Such a set is called LWW-element-Set, and can be obtained by combining a 2P-Set and a LWW-Register for each added and removed element, as described by Shapiro et. al. in [27]. The LWW--element-Set is also a CRDT. Even though a LWW-element-Set allows to add and remove the same element multiple times, the results of concurrent adds and removes depends on how the timestamps are allocated [27], which might result in non intuitive results.

Another possibility is to make sure that a *remove*(e) only affects the *add*(e) operations that causally happened before the *remove*(e) (i.e., all the *add*(e) that are visible to the *remove*(e)). This can be achieved by, on *add*(e), tagging the element e with an unique identifier (id) and then, on *remove*(e), removing all the tags of e that the source replica knows about. Such a set is known as Observed-Remove Set (OR-Set) [27], because a remove only affects the adds that were "observed". As such, for concurrent *add*(e) and *remove*(e), precedence is given to the add, which implies that this set is an add-wins set.

The op-based specification for the OR-Set can be found in Algorithm 2.6. Proof that the op-based OR-Set is a CRDT can be found in [27]. The state-based specification of the OR-Set can be found in Appendix A.2.2.

In the op-based version, set A represents the pairs of (element, unique-id) of elements that are in the OR-Set. On *add*, an unique id is generated in the *atSource* phase. That unique id, along with the element, is passed to *downstream* replicas, which add the element and the unique id to A. Even if two replicas concurrently do an *add* for the same element, *lookup* only returns one element (i.e., it hides the duplicates). As for *remove*, on *atSource* phase, all the pairs of (element, unique-id) of the element being removed that are in the source's payload are collected. This set is passed to *downstream* replicas, which remove those pairs from their local state. A delivery order that delivers all *adds* for the unique ids contained in the downstream argument before the *remove* is assumed (causal order is enough), otherwise the states would not converge or an extra set for removes would be needed (as in state-based 2P-Set and state-based OR-Set).

The OR-Set has a much more intuitive behaviour compared to the sets referred before. For sequential *adds* and *removes* on the same element (and concurrent *adds* and *removes* on different elements), the result conforms to the specification of a sequential set (PSE).

Algorithm 2.6 op-based OR-Set CRDT

```
1:  payload set A                                   ▷ A: set of pairs (element e, unique-id u)
2:      initial ∅
3:  query lookup (element e) : boolean b
4:      let b = (∃u : (e, u) ∈ A)
5:  query elements () : set E
6:      let E = {e | ∃(e, u) : (e, u) ∈ A}
7:  update add (element e)
8:      atSource (element e) : unique-id u
9:          let u = unique()                        ▷ unique(): generates an unique identifier
10:     downstream (element e, unique-id u)
11:         A := A ∪ {(e, u)}
12: update remove (element e)
13:     atSource (element e) : set R
14:         pre lookup(e)                           ▷ Only removes the element if it exists
15:         let R = {(e, u) | ∃u : (e, u) ∈ A}
16:     downstream (set R)
17:         pre ∀(e, u) ∈ R : add(e, u) has been delivered      ▷ Causal order is enough
18:         A := A \ R
```

Concurrent *adds* and/or *removes* on different elements give the expected result, independently of the order of execution (PPE). As for concurrent *adds* and *removes* on the same element, the *remove* won't have observed the unique id generated by *add* and, as such, the element will stay in the set (i.e., an add-wins behaviour, which respects PCS). With this, we can conclude that the OR-Set is *sequentially conformant* with the sequential set specification.

2.5.4 Map

A map is a collection type that contains pairs of (key, value), with each key being unique and "mapping" to a value. In the same map it's possible to have the same value more than once, as long as it is associated to different keys.

Usually a map has at least the following operations: (i) *add*(k, e); (ii) *remove*(k); (iii) *contains*(k); (iv) *get*(k); (v) *keys*(); (vi) *elements*(). The *add*(k, e) associates the value e to the key k in the map; *remove*(k) removes the key (and thus, its value) from the map; *contains*(k) returns true if the key is in the map; *get*(k) returns the value associated to the key (if any); *keys*() and *elements*() return, respectively, the set of keys and values. The first two operations are updates, while the other four are queries.

A map CRDT can be obtained based on any of the set specifications referred in Section 2.5.3. The main idea is to, instead of having sets as the payload, have maps of keys to sets of values. A detail to keep in mind is that, in order for the map CRDT to behave, for sequential adds, as a sequential map, then every add to a key k should remove all the elements associated to k that the source replica knows about. However, if two

concurrent *adds* for the same key happen, then possibly both values can be kept. If a concurrent *add* and *remove* for the same key happens, then the action corresponding to the set specification used as base should be taken.

Algorithm 2.7 op-based OR-Map CRDT

```
1:  payload map A                          ▷ A: map of keys ↦ set of pairs (element, unique-id)
2:      initial ∅
3:  query contains (key k) : boolean b
4:      let b = (k ∉ A)                    ▷ Note: k isn't considered to be in A if A[k] is the empty set
5:  query get (key k) : set E              ▷ The returned set only contains elements
6:      pre contains(k)
7:      let E = {e | ∃(e, u) : (e, u) ∈ A[k]}
8:  query keys () : set K
9:      let K = {k | ∃k : k ∈ A}
10: query elements () : set E
11:     let E = {e | ∃(k, e, u) : (e, u) ∈ A[k]}
12: update add (key k, element e)
13:     atSource (key k, element e) : unique-id u, set R
14:         let u = unique()               ▷ unique(): generates an unique identifier
15:         let R = A[k]                   ▷ Collects pairs to remove from the key
16:     downstream (key k, element e, unique-id u, set R)
17:         pre ∀(e, u) ∈ R : add(k, e, u) has been delivered    ▷ Causal order is enough
18:         A[k] := A[k] ∪ {(e, u)} \ R
19: update remove (key k, element e)
20:     atSource (key k, element e) : set R
21:         pre contains(k)                ▷ Only removes the key if it exists
22:         let R = A[k]
23:     downstream (key k, set R)
24:         pre ∀(e, u) ∈ R : add(k, e, u) has been delivered    ▷ Causal order is enough
25:         A[k] := A[k] \ R
```

A concrete example of a map CRDT is the op-based Observed-Remove Map (OR--Map) present in Algorithm 2.7, which is based, as the name suggests, on the op-based OR-Set discussed in Section 2.5.3 (Algorithm 2.6). Proof that the OR-Map is an op-based CRDT can be obtained based on the OR-Set. This map CRDT is *sequentially conformant* with the sequential map specification, since operations executed sequentially on any key or concurrently on different keys give states according to the sequential map and, for concurrent *adds* on the same key but with different elements, both elements are kept ("error mark"), similarly to the MV-Register. Concurrent *adds* and *removes* on the same key behave similarly to the OR-Set.

Note that it would be possible to define a map CRDT in which each key always maps to, at most, one value. However, that would involve solving conflicts on concurrent adds which could, possibly, be done by defining a total order for the adds (as in LWW-Register) or a way to compare all the possible values and, for example, choose the bigger value on

concurrent adds for the same key.

2.5.5 RGA

The Replicated Growable Array (RGA) is a replicated ordered list (or sequence) data type introduced by Roh et. al. [25]. It supports four operations: (i) *insert*; (ii) *delete*; (iii) *update*; (iv) *read*; with all of these receiving as an argument the "position" to be accessed/modified by the operation (*insert* and *update* also receive the value to insert/update).

However, unlike in a sequential ordered list, an index isn't enough to precisely define where an element should be inserted in. For example, consider the sequence *abc* and two concurrent insertions of, respectively, *d* and *e* on position 2 (after *c*): if the operations are executed in different orders in different replicas, then the states will diverge, as some replicas will end up with *abcde* and others with *abced*. As such, Roh et. al. propose the s4vector [25] structure in order to uniquely identify each position in the sequence and totally order concurrent conflicting operations, guaranteeing eventual convergence of all replicas.

One important detail is that the *insert* operation, unlike in a sequential ordered list, doesn't specify the index where the value should be inserted, but rather the position (s4vector) of the element *after which* the value should be inserted to. Considering again the example given above, multiple inserts at the start of the sequence could happen concurrently with another insert that was intended to place a value after *b*: if an absolute position was given instead of the position of *b*, then the insert could end up in a non--intended position due to the referred concurrent inserts. Another detail is that removed positions are kept as tombstones [25] (similarly to the *R* set in the state-based OR-Set specification in Appendix A), which allows for an insert to still happen successfully on a replica even if on that replica the position in the argument had already been removed.

An op-based CRDT specification for the RGA data type can be found in [27].

2.5.6 Treedoc

Treedoc is an op-based CRDT proposed by Preguiça et. al. [23, 26] designed to be used for collaborative text editing. Treedoc is based on a binary tree and each node stores an element (for example, a character or an image) and is identified by a node id (*posID*). Three operations are supported: (i) *insert*(*posID*, *element*); (ii) *delete*(*posID*) and (iii) *read*(*posID*).

Similarly to the RGA data type discussed previously, one of the main problems is how to define *posID*. In order to solve concurrency conflicts in Treedoc, it is required that *posID* are unique and totally ordered (consistent with causality). Since identifiers are unique, then any two concurrent *inserts* commute. Two *deletes* on the same *posID* do not pose a problem, because the *delete* operation is idempotent. Finally, if an *insert* and a *delete* refer the same *posID*, then the *insert* has causally happened before the *delete* and, as such, are not concurrent [23].

Because it is intended that an element can be inserted between any two nodes, the *posID* domain needs to be dense. Even though real numbers represent a dense domain, it would theoretically require infinite precision. As such, Preguiça et. al. [23, 26] propose a different solution, which consists in using the path to a node as the node's *posID*. The total order is obtained by traversing the tree, from the root, in infix order. However, by using just the path, it is possible for two concurrent *inserts* with the same path to happen: this is solved by allowing nodes (called major nodes) to contain mini-nodes. Each mini node contains a disambiguator, which allows to distinguish and totally order the mini-nodes inside a major node.

Treedoc also uses tombstones for removed nodes, however, in some situations, they can be safely deleted (for example, when a deleted node has no children). A major problem with Treedoc is that, as *inserts* and *deletes* are executed, the tree may become unbalanced and thus, performance may be affected. Two optimizations are proposed to deal with tombstones and tree unbalance, which the authors name of *explode* and *flatten* [23]. However, *flatten* requires a distributed commitment algorithm, which has limited scalability.

2.5.7 Logoot

Logoot is an op-based CRDT proposed by Weiss et. al. [30, 31] that, similarly to the Treedoc, is designed to be used for collaborative text editing. However, besides the differences between both in terms of data structure and identifiers, Logoot has an extension called Logoot-Undo [31] which supports, not only the *insert*, *delete* and *read* operations, but also the *undo* operation (and the *redo*, i.e., an *undo* of an *undo*).

In Logoot-Undo, in order to provide commutativity for concurrent operations, each element in a document is identified by an unique, totally ordered identifier. An identifier is composed by a list of positions, with each position having three values: a number, a replica identifier and a clock value. Each time an element is inserted into a document, a new identifier is generated, with different strategies for generating them being available [31]. Because identifiers are totally ordered and consist in a list of positions, it is always possible to generate an identifier between any two identifiers, even if the digit on both identifiers is the same (in that case, the new identifier is the composition of the first identifier and a new generated position). This implies that the size of an identifier can grow unbounded, however, experimental evaluation done shows that identifiers tend to stay small [31].

In order to support the *undo* operation, and because multiple concurrent *undos* may be issued for the same identifier, each element has associated to it a "visibility degree", which is a counter. If the counter's value is higher than 0, then the element is visible, otherwise it isn't. After an *insert*, the counter's value is 1. A *delete* decrements the counter. *Undo* and *redo* of insert operations, respectively, decrement and increment the counter (and the opposite for delete). As such, the element "dissapears" if the counter goes from

1 to 0 and "reappears" if it goes from 0 to 1. This solution is similar to the PN-Counter proposed by Shapiro et.al [27].

A deleted element (i.e., one with a counter of 0 or lower) is not kept as a tombstone in the document data structure. However, concurrently deleted elements (i.e., ones in which the counter goes below 0) are kept in a separate data structure known as the cemetery [31]. The size of the cemetery tends to be small compared to the document size, since only concurrently deleted elements go there (and, if an element gets re-added due to an undo, it leaves the cemetery).

2.5.8 JSON

The JavaScript Object Notation (JSON) is a tree data format whose nodes can be one of the following types: (i) map (or object); (ii) list (or array) or (iii) basic data type (number, boolean, string, null). A map can contain one or more pairs of (key, value), while the list contains an ordered collection of values. On both cases, the values can either be another map, list, or one of the basic data types. Each value is considered a node.

Kleppmann et. al. present in [15] a CRDT for the JSON data type. The key challenge of defining this CRDT is supporting *inserts*, *removes* and *updates* of the inner objects (map, lists, basic data types) of a JSON document, without treating them as opaque. In other words, this means directly supporting, for example, adding a value to a map that is inside a list in a JSON document. Most of the CRDTs in the literature [2, 12, 23, 27, 31] consider the inner values as opaque. This means that, in those CRDTs, if we, for example, have a set as a value of the map, in order to add an element to the set it would be necessary to delete and then insert the whole set with the new element. Due to concurrency problems, in these types of CRDTs the effects of some update operations on the same value may be lost, since normally a last-writer wins approach is used for these elements.

Directly supporting operations at any node of the tree poses concurrency problems that don't occur on data types such as the OR-Set. These problems happen because conflicting concurrent operations can happen at different levels of the tree. For example, a replica may issue an operation that deletes a map A, while another replica concurrently issues an operation that adds the element c to the array B that is in the map A. Semantics for dealing with these conflicts and others are defined in [15].

The JSON CRDT is, thus, a good example of how multiple CRDT data types can be composed (it has maps, lists and registers, with the registers representing a basic data type). An advantage of having a JSON CRDT is that not only is JSON a data format used to exchange information between many applications on the web, it is also used to store the state of some applications, which can allow them to switch into using CRDTs to represent their state more easily [15].

Chapter 3

Generic CRDT Model

In this chapter we propose a CRDT model which acts as a basis for specifying tunable-CRDTs (t-CRDTs). This model defines the necessary metadata along with what's required from the user to specify. The model we propose lets the user precisely define what happens for each operation when it is compared to other operations, allowing the user to combine multiple different policies in just one CRDT.

For this model we assume the system model described in Section 2.4. We also assume that every replica eventually receives every operation, but we make no assumptions on the delivery order nor on how many times each operation is delivered.

Our model is generic enough to be the basis for many different t-CRDTs. As will be proven in Chapter 5, convergence is always ensured by the model as long as the policies defined by the user are deterministic and the mentioned network requirements are met. Our model does not, however, ensure any of the three correctness principles introduced in Section 2.4.2, as that would require our model to be less generic and more restrictive. As such, each t-CRDT must ensure those principles by itself. We'll detail this further in Chapter 5.

3.1 Model Requirements

The key property of CRDTs is that they solve concurrency conflicts automatically by applying a specific policy. Usually in CRDTs the visible state (i.e., the state returned by query operations) gets changed after an operation is applied. However, for most CRDTs, not all operations are relevant to the final visible state. In fact, some operations lose their effect on state after a concurrent or a more recent operation happens. For example, consider a set CRDT. If an $add(e)$ is executed after (i.e., related by happens-before [17]) a

remove(*e*), then the *remove*(*e*) won't have any effect on the visible state. I.e., it is *obsolete* [1].

The concept of an *obsolete* operation is essential to specify our model. In a way, policies can be interpreted as functions that, given a set of operations, decide which operations still have an effect on the visible state and those who don't, i.e., are *obsolete*. As such, one essential requirement of our model is to separate *obsolete* operations from *active* ones. With this, it is possible to calculate at any moment which operations are still relevant to the final state.

First requirement: Concept of *obsolete* and *active* operations and a way to classify them.

As with any other CRDT it is also needed to have a way to read the state. However, since it is intended for the model to be as generic as possible and work for any type of CRDT, the read functions aren't incorporated in the model but are, instead, supplied by either a "wrapper" CRDT (ie, a CRDT which uses the generic model as a basis) or by the user. Either way, the key idea is that those functions consult the set of *active* operations in order to compute a state to return to the user.

Second requirement: User-supplied read functions that read the set of *active* operations in order to compute a state to return to the user.

It was mentioned that a way to separate *active* from *obsolete* operations is needed. Even though it is a requirement from the model to have an algorithm to classify those operations, the decision of what's obsolete and what's not shouldn't be done by the model. In fact, that decision depends both on the data type semantics and on the policy used to solve conflicts.

As such, since the objective is to allow the user to define his own policies for each operation, the generic model shouldn't have any policy functions. The policies to be applied should be supplied by the user in each operation, in the form of a function that must decide at least one of those: (i) when the operation should be obsoleted; (ii) when the operation should make obsolete other operations. The model's algorithm will then apply the functions supplied in each operation to calculate the set of active operations.

Third requirement: User-supplied policy functions for each operation. These policies must be able to determinate when the operation they're associated to should be obsoleted or, alternatively, when it should obsolete others.

These three requirements form what we identify as the bare minimum to be able to specify a generic model which allows the user to define his own policies.

[1] On the remaining of this document we'll use obsolete and canceled interchangeably.

3.2 Specification

The op-based specification for our generic CRDT satisfying the previously discussed requirements can be found in Algorithm 3.1, while the state-based can be found in Appendix D. In short, the algorithm calculates the set of active operations whenever a query function is executed. This set is obtained by applying the policies defined by each operation to every other operation in the state.

Algorithm 3.1 Generic op-based data type

1: **payload** set O ▷ O: set of received operations.
2: **initial** $\emptyset$
3: **update** *addOp* (operation *op*)
4: **atSource** (operation *op*)
5: **downstream** (operation *op*)
6: $O := O \cup \{op\}$

▷ Auxiliary procedure used by query operations.
▷ Calculates the set of active operations and removes unnecessary operations.
7: **procedure** *calculateState* () : set *nonObs*
8: **let** $obsByHB = \{op : op \in O \wedge \exists otherOp \in O : op \prec otherOp \wedge otherOp.hb(otherOp, op)\}$
▷ Operations obsoleted by happens-before aren't needed anymore.
9: $O := O \setminus obsByHB$
▷ Collects operations obsoleted by concurrency.
10: **let** $obsByConcurrency = \{op : op \in O \wedge \exists otherOp \in O : op \parallel otherOp \wedge (op.selfObsoletePolicy(otherOp, op) \vee otherOp.otherObsoletePolicy(otherOp, op))\}$
11: **let** $nonObs = O \setminus obsByConcurrency$

▷ readFunction: function supplied by the user that consults the set of active operations (calculateState) and returns some kind of result.
12: **query** *query* (function *readFunction*, arguments *otherArguments*) : result *r*
13: **let** $r = readFunction(calculateState(), otherArguments)$

Note: A possible optimization is to cache *nonObs* and use it for query instead of *calculateState*() until there's a change to the state.

The state is simply the set of received operations (set O). The model has only one update operation – $add(op)$. This operation is fairly simple – it just adds the received operation to O. No order for operation delivery is assumed – we only require that each operation is delivered at least once. The set of active operations is calculated by the auxiliary procedure *calculateState*(), which is called whenever a query function is applied.

3.2.1 Operations

Each operation can have as many fields as the user finds necessary to correctly represent the intended data type and to apply the policies. Besides the policies, each operation has one mandatory field: an history field, which must provide enough information to decide whenever it happened-before, is concurrent or happened-after any other operation. A

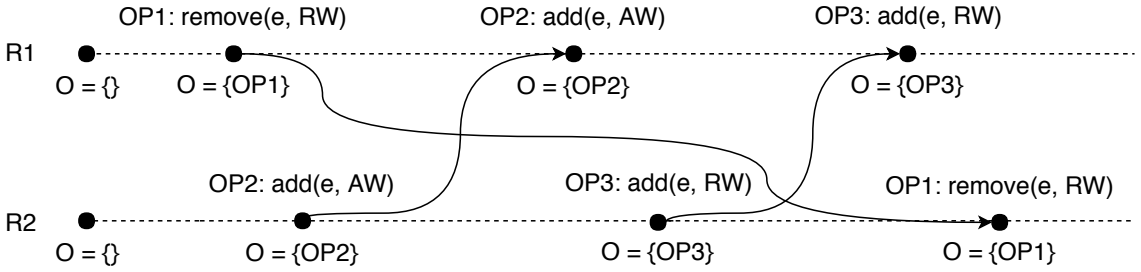

Figure 3.1: Example of a situation in which replicas would diverge if they could delete concurrent obsolete operations. Here RW and AW stand for, respectively, remove-wins and add-wins policies. Assume that add(e, AW) and add(e, RW), respectively, win and lose versus concurrent remove(e, RW).

simple (yet inefficient) solution is to have in each operation a set with every operation (or an id that identifies them) that happened-before it. A more efficient solution is to have logical clocks consistent with causal order, as described in [13].

If we consider a set CRDT, both *add* and *remove* would have the policies fields, the history and a field for storing the element which is being added or removed.

3.2.2 Calculating active operations

The procedure *calculateState*() is responsible for calculating the set of active operations. For that goal, it applies the policies associated to each operation in set *O*. This procedure satisfies the **First requirement** previously mentioned.

In detail, the steps executed by the procedure are:

1. Calculate and remove from the state the set of operations that are canceled by happens-before. These operations will never be relevant to the state again and can, thus, be safely removed.
2. Calculate the set of operations obsoleted by concurrency. This set is calculated by applying, for each operation (op), its *selfObsoletePolicy* and *otherObsoletePolicy* functions (these will be explained later) to itself and every other operation. If either of those return true at least once, then op is considered obsoleted. Note that we can't delete operations obsoleted by concurrency, as they are still relevant to determine wherever other concurrent operations should be turned obsolete or not. Otherwise we would break convergence, as shown in Figure 3.1.
3. Calculate the set of active operations, which consists in the set of operations minus the set of canceled operations.

3.2.3 Policy functions

Each operation (op) has policy functions associated to it. More precisely, it has 3 policy functions, which are:

1. *happens-before policy (hbPolicy)*: user-defined policy that receives op and another operation (otherOp) as arguments, where otherOp $\prec$ op (i.e., otherOp happened-before op). The function should return true if op makes otherOp obsolete. This function is used to determinate when op should obsolete other operations that happened before it;
2. *selfObsoletePolicy*: user-defined policy that receives another operation (otherOp) and op as arguments, where op $\parallel$ otherOp (i.e., otherOp and op are concurrent). The function is used to determinate when op should be turned obsolete by other concurrent operations. As such, it should return true if otherOp makes op obsolete;
3. *otherObsoletePolicy*: user-defined policy that receives op and another operation (otherOp) as arguments, where op $\parallel$ otherOp. The function is used to determinate which concurrent operations op can obsolete and, as such, should return true if op makes otherOp obsolete.

It is required for all of these predicates that: (i) the result is deterministic and (ii) they eventually terminate. The happens-before policy has two extra requirements: (i) the predicate must be transitive and (ii) the result of the predicate must only depend on the two operations in the arguments, namely it must not consult the set of operations (set O).

To exemplify, consider a set t-CRDT for which we want an add-wins policy. For this data type a *hbPolicy* only needs to check if the elements are the same, independently of the set t-CRDT being add-wins or remove-wins – an *add* of element e should never cancel a *remove* of element f (with $e \neq f$) that happened-before, but it must cancel it if that remove was to element e. As for *selfObsoletePolicy* and *otherObsoletePolicy* it is necessary to specify one or more predicates that define the concurrent behavior of *add* and *remove*, that is, that a *remove* should lose to a concurrent *add* of the same element.

Note that nothing prevents the user from defining a single predicate to express the three policies. However, that would make specifying the policies more difficult, as the user would need to differentiate between concurrent, happened-before and happened-after and write a predicate dealing with all these cases individually. Separating operations by their relation allows the user to only think about one relation per predicate. In fact, the user never needs to consider operations related by happened-after, as it is reasonable to assume that an operation will never obsolete operations that happen after it.

The discussed predicates fulfill the previously discussed **Third requirement**. As will be seen in Chapter 4, it is often enough and easier to specify a predicate used by both *selfObsoletePolicy* and *otherObsoletePolicy*. Note however that this is only possible if the conflict resolution policies to be applied for both cases are the same.

3.2.4 Queries

The *query* operation is used whenever the user intends to consult the state. It acts as an intermediary between the user and the model by filtering the active operations. As such, the *query* operation receives as arguments a function that reads the set of active operations,

along with other arguments necessary for that function, and computes a result. Thus, this satisfies the **Second requirement**, by allowing users to define their own functions to read the state.

Usually a read function consists in a function that searches for the existence of a certain type of operation. For example, the *lookup*(*e*) in a set CRDT would consist in searching for the existence of an *add*(*e*) operation in the state, assuming that a *remove*(*e*) and an *add*(*e*) are never simultaneously present on the active state. If that isn't true, *lookup* would be different – it would also have to take in consideration *removes* and apply a policy by itself. This implies that the way policies are specified may affect how read operations are specified. However, as long as policies are correctly and carefully specified, then read functions should be fairly simple, as will be seen in the next chapter.

Chapter 4

Specifying tunable CRDTs

In this chapter we demonstrate how the generic model discussed on Chapter 3 can be used as a basis to specify useful t-CRDTs with well-defined policies that guarantee the adequate behavior.

We start by presenting a t-CRDT library[1] which contains some basic data types. For each t-CRDT we present both its API and the associated pre-defined policy functions. All of the t-CRDTs we present respect the three principles we introduced in Section 2.4.2 (PSE, PPE and PCS) when used with the provided policies. For each t-CRDT we built a formal specification and verified both these principles and other data-type specific properties, as will be detailed in Chapter 5.

To finalize, we propose a methodology that can be used to correctly specify t-CRDTs using the generic model and illustrate the process by using the set t-CRDT as an example.

4.1 Library

To demonstrate the usability of our generic model we present a group of t-CRDTs, each one representing a different data type that was built on top of the generic model, following the methodology that will be discussed in Section 4.2. Along with each data type we also include some of its most common policies. We present two basic data types (registers and counters) and two collection data types (sets and maps).

The discussion of each data type is organized as follows. First, we shortly describe what the data type is and what can be done with it. Afterwards we present a specification of the t-CRDT for that data type, which includes both the operations and the queries. Then we present some policies that can be used for that data type, followed by remarks

[1] Our Java implementation of the complete t-CRDT library is available at: https://github.com/AndreRijo/T-CRDTs

in which we explain how the policies should be used and what is their priority order (i.e., what happens when multiple policies are used concurrently). Finally we end by showing an application example in which the t-CRDT and multiple of its policies can be used.

4.1.1 Register

A register is a data type that stores a single element of some type. Its API usually contains one operation to update the register's value and one query that returns its current value. A common usage of registers is, for example, in file systems, in which the elements are files or blocks of a file.

Specification Algorithm 4.1 contains the specification of an op-based register t-CRDT. *Assign* updates the value of the register with the one received as argument. It also receives a clock as argument, which will be used by some policies. *Value* returns the current value(s) of the register. *AddOp* and *calculateState* are abstractions provided by the generic model.

Algorithm 4.1 op-based register t-CRDT

```
1: update assign (policy hbP, policy selfP, policy otherP, X v, clock clk)
2:     atSource (policy hbP, policy selfP, policy otherP, X v, clock clk) : operation op
3:         let op = [hbPolicy → hbP, selfObsoletePolicy → selfP, otherObsoletePolicy → otherP, value → v, clock → clk]
4:     downstream (operation op)
5:         addOp(op)
6: query value () : set v
7:     let v = {v : ∃op ∈ calculateState() : op.value = v}
```

Policies Figure 4.1 contains the function for the register's hb-policy. An *assign* obsoletes all other *assigns* that happened-before it (note that the model guarantees that $otherOp \prec op$ when calling the function).

$$registerHB(op, otherOp) \triangleq true$$

Figure 4.1: Happens-before (hb) policy function for the register t-CRDT.

We define in our library three concurrency policies for the register data type: (i) *MV*; (ii) *weakLWW*; (iii) *strongLWW*. The last two apply last-writter-wins semantics, using the *clock* value associated to each operation in order to only keep one *assign* when multiple concurrent *assigns* happen. The difference between them is that *strongLWW* also obsoletes concurrent *assigns* with *MV* policies, while *weakLWW* doesn't. The multi-value policy (*MV*) allows for all concurrent *assigns* to be kept in the state. The three mentioned policies are defined in Figure 4.2.

$isLWW(op)$	$\triangleq$	$op.selfObsoletePolicy = weakLWW$
		$\vee\ op.selfObsoletePolicy = strongLWW$
$weakLWW(op, otherOp)$	$\triangleq$	$isLWW(op)\ \wedge\ isLWW(otherOp)\ \wedge\ otherOp.clock < op.clock$
$strongLWW(op, otherOp)$	$\triangleq$	$isLWW(op)\ \wedge\ (otherOp.selfObsoletePolicy = MV$
		$\vee\ otherOp.clock < op.clock)$
$MV(op, otherOp)$	$\triangleq$	$false$

Figure 4.2: Concurrency policies functions for the register t-CRDT. For readability reasons, we define the *isLWW* auxiliary function which checks if the policy received as argument is a LWW variant.

API Figure 4.3 summarizes the API for the register t-CRDT, which contains the supported operations, queries and policies.

ops	=	$assign(value, clock)$
queries	=	$value()$
happens-before function	=	*registerHB*
policy functions	=	*weakLWW*, *strongLWW*, *MV*

Figure 4.3: API for the register t-CRDT

Remarks The *LWW* policies require a logical clock with total order guarantees (consistent with causality). Multiple different implementations are possible [13, 17]. One possibility is to have a counter for each replica associated with an unique identifier (e.g., the replica's MAC address), that is, a vector clock [13, 27]. One advantage of this implementation is that it can also be used as an implementation for the history field of the generic model (i.e., to detect if two operations are concurrent, happens-before or happens-after).

The defined concurrency policies form the following priority order: *weakLWW* $\rightarrow$ *MV* $\rightarrow$ *strongLWW*. Considering the order and how the policies are defined, this means that when one *assign* is executed with *strongLWW*, no other concurrent *assign* will be in the visible state (i.e., the register only has one value). On the other hand, when only *MV* and *weakLWW* are used, it is possible for the register to have multiple values associated to it due to concurrency.

Depending on the application scenarios, different combinations of policies can be used. We define the following scenarios:

1. only *LWW* semantics are needed: either use *weakLWW* or *strongLWW* (no need to use both);
2. only *MV* semantics are needed: only use *MV*;
3. both *LWW* and *MV* semantics are needed but wherever an *assign* with *MV* is executed, the latest concurrent *assign* with *LWW* should be kept: use *weakLWW* and *MV*;

4. both *LWW* and *MV* semantics are needed but all concurrent *MV assigns* should be obsoleted in the presence of at least one *assign* with *LWW*: use *strongLWW* and *MV*;
5. both (3) and (4), with each one being chosen for a different situation: use *weakLWW* for (3), *strongLWW* for (4) and *MV* for when keeping multiple values is desired.

Application example As an example of a situation in which having a register t-CRDT has advantages over a register CRDT with only one policy, consider a video hosting service similar to Youtube. In this service each video is available in multiple replicas and a user can connect to any of the replicas to watch a video.

Consider that the administrators of this service want to collect multiple statistics on how users use their service. For our example, assume there is a very popular video for which the administrators want to know, on average, how much of the video's length is watched by the users. Assume that each replica maintains enough information to calculate the actual average time watched for that popular video in that replica, but may not have the latest information about the other replicas. This implies that to know the true average across all replicas, multiple replicas may need to be consulted.

Let's assume that the administrators want an estimate as soon as possible, even if it isn't 100% accurate. As such, we can replicate the average of all replicas in a register t-CRDT and have the replicas update it as they gather information from other replicas. To speed up the process, multiple replicas will try to gather enough information to calculate the true average and, as they do so, they calculate intermediate results and update the register t-CRDT with those results. Since intermediate results aren't 100% precise, it is reasonable to keep all of the latest concurrent *assigns*, which can be achieved by using the *MV* policy. However, as soon as one replica calculates the true average, this *assign* should obsolete all other intermediate results, which can be achieved by *strongLWW*. Also, as soon as a replica sees one *assign* with *strongLWW*, that replica can stop calculating the average. Note that having two or more replicas concurrently setting the true average is not a problem – *strongLWW* will ensure only one survives.

To summarize, situations in which we may have multiple intermediate or imprecise results concurrent with one or more one final or precise values are examples of scenarios for which our register t-CRDT is more adequate than using a register CRDT with only LWW or MV policy.

4.1.2 Counter

A counter is a data type that stores an integer, supporting operations to increment and decrement its value by one. Optionally, it may also support operations to increase/decrease the value by a given amount, or an operation to reset the counter's integer to a certain value. A counter should also support a query to return the counter's current value.

Counters are widely used in distributed systems, with some usages examples being to count the number of visits of a webpage or the number of likes in a social media post.

Specification Algorithm 4.2 contains the specification of an op-based counter t-CRDT. It supports the basic *increment* and *decrement* operations (which can be easily extended to support incrementing/decrementing values received as argument), along with an operation to reset the value, *setValue*. The counter's current value can be obtained with the *value* query. Finally, *addOp* and *calculateState* are abstractions provided by the generic model, while *getIncs*, *getDecs* and *getSetValue* are auxiliary procedures used by *value*.

Algorithm 4.2 op-based counter t-CRDT

```
1:  update increment (policy hbP, policy selfP, policy otherP)
2:      atSource (policy hbP, policy selfP, policy otherP) : operation op
3:          let op = [hbPolicy → hbP, selfObsoletePolicy → selfP, otherObsoletePolicy → otherP, type → inc]
4:      downstream (operation op)
5:          addOp(op)
6:  update decrement (policy hbP, policy selfP, policy otherP)
7:      atSource (policy hbP, policy selfP, policy otherP) : operation op
8:          let op = [hbPolicy → hbP, selfObsoletePolicy → selfP, otherObsoletePolicy → otherP, type → dec]
9:      downstream (operation op)
10:         addOp(op)
11: update setValue (policy hbP, policy selfP, policy otherP, integer v)
12:     atSource (policy hbP, policy selfP, policy otherP) : operation op
13:         let op = [hbPolicy → hbP, selfObsoletePolicy → selfP, otherObsoletePolicy → otherP, type → set, value → v]
14:     downstream (operation op)
15:         addOp(op)
16: query value () : integer v
17:     let nonObs = calculateState()
18:     let v = #(getIncs(nonObs)) − #(getDecs(nonObs)) + getSetValue(nonObs)
19: procedure getIncs (set nonObs) : set incs
20:     let incs = {op ∈ nonObs : op.type = inc}
21: procedure getDecs (set nonObs) : set decs
22:     let decs = {op ∈ nonObs : op.type = dec}
23: procedure getSetValue (set nonObs) : integer v ▷ Assumes that no two setValues with different values are simultaneously active
24:     let v = if (∄op ∈ nonObs : op.type = set) then 0 else op.value
```

Policies Figure 4.4 contains two hb-policy functions for the counter. *Increments* and *decrements* use the *incDecHB* function, while *setValue* uses the *setValueHB* function. Those two functions are needed because *increments* and *decrements* don't cancel previous operations, but *setValue* does as it must discard any previous value.

Our library supports four concurrency policies for the counter data type: (i) *maxWrite*; (ii) *maxMerge*; (iii) *minWrite* and (iv) *minMerge*. "Min/max" refer to which value is kept

$incDecHB(op, otherOp)$	$\triangleq$	$false$
$setValueHB(op, otherOp)$	$\triangleq$	$true$

Figure 4.4: Happens-before (hb) policy functions for the counter t-CRDT. *IncDecHB* and *setValueHB* should be used by, respectively, *increments*/*decrements* and *setValue*.

when two or more concurrent *setValues* occur: respectively, keep the max or min of all. As for "merge/write", they respectively keep or ignore all *increments* and *decrements* that are concurrent to the *setValue* applying the policy. Note that all these policies are to be used by *setValue*, as the other two operations are naturally commutative between themselves and as such don't need concurrency policies. The specification for the referred policies can be found in Figure 4.5.

$maxWrite(op, otherOp)$	$\triangleq$	$max(op, otherOp) \lor write(op, otherOp)$
$maxMerge(op, otherOp)$	$\triangleq$	$max(op, otherOp) \lor merge(op, otherOp)$
$minWrite(op, otherOp)$	$\triangleq$	$min(op, otherOp) \lor write(op, otherOp)$
$minMerge(op, otherOp)$	$\triangleq$	$min(op, otherOp) \lor merge(op, otherOp)$
$max(op, otherOp)$	$\triangleq$	$op.type = set \land otherOp.type = set$ $\land$ ($otherOp.selfObsoletePolicy = minWrite$ $\lor otherOp.selfObsoletePolicy = minMerge$ $\lor$ (($op.selfObsoletePolicy = maxWrite$ $\lor op.selfObsoletePolicy = maxMerge$) $\land op.value > otherOp.value$))
$min(op, otherOp)$	$\triangleq$	$op.type = set \land otherOp.type = set$ $\land otherOp.selfObsoletePolicy \neq maxWrite$ $\land otherOp.selfObsoletePolicy \neq maxMerge \land op.value < otherOp.value$
$write(op, otherOp)$	$\triangleq$	$otherOp.type = inc \lor otherOp.type = dec$
$merge(op, otherOp)$	$\triangleq$	$false$

Figure 4.5: Concurrency policies for the counter t-CRDT. *MaxWrite*, *maxMerge*, *minWrite* and *minMerge* correspond to the policy functions, while the other four are just auxiliary functions.

API Figure 4.6 summarizes the API for the counter t-CRDT, which contains the supported operations, queries and policies.

ops	=	*increment*(), *decrement*(), *setValue*(*value*)
queries	=	*value*()
happens-before function	=	*incDecHB*, *setValueHB*
policy functions	=	*maxWrite*, *maxMerge*, *minWrite*, *minMerge*

Figure 4.6: API for the counter t-CRDT

Remarks The *value* query assumes that at most one *setValue* is active or, alternatively, that all active *setValues* have the same value. The previously defined policies ensure the

latter. If this wasn't guaranteed, an implementation of the query could return different results depending on how the operations are stored, which would be incorrect. Writing a different function for the query isn't an alternative, as that would imply filtering *setValues* by applying a certain policy in the query, which goes against the principle of having the update operations define the policies.

When two or more *setValues* are executed concurrently with conflicting policies, priority is deterministically given to one of the policies. Specifically, for *max* and *min*, priority is given to *max*, independently of the values associated to *min* (e.g., a *max* with value 2 would win over a concurrent *min* with value 5). The max policy needs to check both *op*'s and *otherOp*'s *selfObsoletePolicy*, in order to deal correctly with the referred priority. As for *merge* and *write*, due to the way the policies are specified and how the generic model works, concurrent *increments* and *decrements* are always obsoleted even if the *setValue* with *merge* wins [1]. We believe that this is not a problem, as these policies naturally conflict and, as such, any deterministic result is acceptable. As such, we have not investigated on how to get around this fact, but we conjure that it would be possible to get around it by specifying policies for *increment* and *decrement* to deal with this case.

Choosing which policies are adequate to an application depends on the scenarios in which the counter will be used. The answer to the following two questions can be used as a guideline for choosing the policies:

1. Is the intention when executing a *setValue* to keep concurrent *increments* and *decrements*?
2. When two or more *setValues* are executed, which value should be kept?

The answer to both questions gives the policy that should be used. For example, for question 1, if the answer is to keep *increments* and *decrements*, then use one of the *merge* policies. If for question 2 the answer is *max*, then use one of the *max* policies. As such, if we combine both answers, we get that we should use the *maxMerge* policy.

Finally we note that it would be possible to specify a *LWW* policy to deal with concurrent *setValues*, similarly to what was done for the register. That would require, however, a logical clock, which may be an undesired overhead for a counter.

Application example To show the usefulness of the policies defined for the counter, consider a distributed clicking game (also known as incremental game) [34, 35] in which multiple players can share a save and cooperate to progress in the game. Any participating player can at any time: (i) click in certain objects to increase the current, shared in-game cash; (ii) buy a building to generate automatic income (i.e., extra cash every second); (iii) buy an upgrade to the amount of cash generated every time a player does a click.

To simplify the problem, we'll only consider how the current amount of cash is stored.

[1] This happens due to the fact that obsolete operations still have effect over other concurrent operations, as otherwise different results would be obtained depending on the order of execution.

For this we propose to use the counter t-CRDT, in which clicks are represented by *increments* and the change to the current money due to buying a building or an upgrade is represented by a *setValue*. Using a counter t-CRDT for this is adequate as this allows any player to take actions without having to wait for synchronization to happen and solve conflicting situations automatically, as clicks and buys of buildings and/or upgrades can happen concurrently.

To start, consider the case in which multiple players buy buildings concurrently. Since it is possible that there is not enough cash to buy all of those buildings, a solution is to keep only the buy which represents the highest acquisition (and, as such, the lowest *setValue* of the current cash). We want, however, to keep all concurrent *increments* generated by other players' clicks and, as such, we should use the *minMerge* policy.

Consider now that at least one player decides to buy an upgrade to the cash generated by each click. This unfortunately conflicts with concurrent *increments*, as it is not clear if those should increment the value by the amount defined before the upgrade or after the upgrade. As such, the safest action is to ignore concurrent *increments* – this can be achieved by using the *minKeep* policy. Another example in which this policy is useful is for the *reset* option that some clicking games have. Usually this option exchanges all the cash, building and upgrades for some exclusive, otherwise unobtainable bonuses. In this situation it is also desired to ignore both concurrent *increments* and other *setValues*.

To summarize, a distributed cooperative clicking game is an example in which a simple counter CRDT is just not enough and, as such, precise control of the policies is needed, which makes our counter t-CRDT adequate.

4.1.3 Set

A set is a collection data type which stores elements of any type without repetition. Elements can be added or removed from the set and it's possible to execute queries to check if a given element is in the set or to return all stored elements. Sets are also the basis of more advanced data types such as maps and graphs, which makes them frequently used. Some usage examples include: (i) storing the IDs of received messages in a reliable broadcast primitive; (ii) checking if a given user exists/is currently online in a system.

Specification Algorithm 4.3 contains the specification of an op-based set. Adding and removing an element given as argument is supported by, respectively, *add* and *remove*. These two operations also receive a clock as argument, which is used by the *LWW* policy. *Lookup* returns true if the element received as argument is in the set, while *elements* returns all elements in the set. *AddOp* and *calculateState* are abstractions provided by the generic model.

Policies Figure 4.7 contains the hb-policy function for the set. Intuitively, an *add* or *remove* should only cancel a previous operation if it's for the same element, as it would

Algorithm 4.3 op-based set t-CRDT

```
1: update add (policy hbP, policy selfP, policy otherP, element e, clock clk)
2:     atSource (policy hbP, policy selfP, policy otherP, element e, clock clk) : operation op
3:         let op = [hbPolicy → hbP, selfObsoletePolicy → selfP, otherObsoletePolicy → otherP, type → add, element → e, clock → clk]
4:     downstream (operation op)
5:         addOp(op)
6: update remove (policy hbP, policy selfP, policy otherP, element e, clock clk)
7:     atSource (policy hbP, policy selfP, policy otherP, element e, clock clk) : operation op
8:         let op = [hbPolicy → hbP, selfObsoletePolicy → selfP, otherObsoletePolicy → otherP, type → remove, element → e, clock → clk]
9:     downstream (operation op)
10:        addOp(op)
11: query lookup (element e) : boolean b        ▷ Assumes that for the same element an add and remove can't be simultaneously active
12:    let b = (∃op ∈ calculateState() : op.type = add ∧ op.element = e)
13: query elements () : set E                   ▷ Same assumption as lookup
14:    let E = {e : ∃op ∈ calculateState() : op.type = add ∧ op.element = e}
```

make no sense to have, for example, an $add(f)$ cancel an $add(e)$ (with $e \neq f$).

$$setHB(op, otherOp) \triangleq op.element = otherOp.element$$

Figure 4.7: Happens-before (hb) policy function for the set t-CRDT.

We provide five concurrency functions for the set data type: (i) *normalRem*; (ii) *normalAdd*; (iii) *priorityRem*; (iv) *priorityAdd* and (v) *LWW*. The combination of these five functions allows us to provide four different policies: (i) *add-wins*; (ii) *rem-wins*; (iii) *priorityAdd-wins* and (iv) *LWW*. The specification of the mentioned functions can be found in Figure 4.8, while the details on how to use these policies will be described in **Remarks**.

$$
\begin{array}{lcl}
normalRem(op, otherOp) & \triangleq & false \\
normalAdd(op, otherOp) & \triangleq & op.element = otherOp.element \ \wedge \ op.type = add \\
 & & \wedge \ otherOp.type = remove \ \wedge \ otherOp.selfObsoletePolicy = normalRem \\
priorityRem(op, otherOp) & \triangleq & op.element = otherOp.element \ \wedge \ op.type = remove \\
 & & \wedge \ otherOp.type = add \ \wedge \ otherOp.selfObsoletePolicy = normalAdd \\
priorityAdd(op, otherOp) & \triangleq & op.element = otherOp.element \ \wedge \ op.type = add \\
 & & \wedge \ otherOp.type = remove \ \wedge \ otherOp.selfObsoletePolicy \neq LWW \\
LWW(op, otherOp) & \triangleq & op.element = otherOp.element \ \wedge \ op.selfObsoletePolicy = LWW \\
 & & \wedge \ (otherOp.selfObsoletePolicy \neq LWW \ \vee \ op.clk > otherOp.clk)
\end{array}
$$

Figure 4.8: Concurrency policies for the set t-CRDT.

API Figure 4.9 summarizes the API for the set t-CRDT, which contains the supported operations, queries and policies.

ops	=	*add*(*element*,*clock*), *remove*(*element*,*clock*)
queries	=	*lookup*(*element*), *elements*()
happens-before function	=	*setHB*
policy functions	=	*normalRem*, *normalAdd*, *priorityRem*, *priorityAdd*, *LWW*

Figure 4.9: API for the set t-CRDT

Remarks Unlike in the register and counter t-CRDTs, here each policy is intended to be used by only one type of operation, except for *LWW*. More precisely, *normalRem* and *priorityRem* should only be used by *removes*, while *normalAdd* and *priorityAdd* should only be used by *adds* and *LWW* can be used by both. The idea behind these policies is that, in each operation, the user can select the intended priority for that operation – e.g., to execute a remove with priority use *priorityRem*; to execute an add without priority use *normalAdd*. On the other hand, if the intention is to rely on a total ordering defined by logical clocks, use *LWW*.

NormalAdd, *priorityRemove*, *priorityAdd* and *LWW* define, respectively, the *add-wins*, *rem-wins*, *priorityAdd-wins* and *LWW* policies. *NormalRem* represents the minimum priority and ensures that the *remove* operation doesn't obsolete any concurrent operation, thus letting the other operations define the concurrency semantics (this function can be seen as a dummy policy which lets others decide).

The mentioned functions define the following priority order to ensure that conflicting policies (e.g., *add-wins* and *rem-wins*) don't, in fact, conflict: *normalRem* → *normalAdd* → *priorityRem* → *priorityAdd* → *LWW*. To exemplify, if we execute concurrently an *add* and a *remove* for the same element with, respectively, *normalAdd* and *priorityRem* functions, the *remove* will win.

As with other data types, choosing the policies which are adequate for an application depends on the scenarios and needs of the application. However, the following topics can be used as a guideline:

1. for *add-wins*, use *normalRem* and *normalAdd*;
2. for *rem-wins*, use *normalAdd* and *priorityRem*;
3. to represent actions with different priorities, (e.g. admin's operations with priority over normal user's), use *normalAdd* and *normalRem* for normal users, *priorityAdd* and *priorityRem* for priority users.
4. to rely in a total order of operations instead of priorities, use *LWW*.

The clock received as argument for *add* and *remove* is only used by *LWW* and, as such, can have a dummy value when other policies are used. For *LWW*, the clock should meet the requirements described in the register's remarks in Section 4.1.1.

Both *lookup* and *elements* queries assume that at any moment either only *adds* or only *removes* are active for each element, but never both. Our policies ensure this property, as otherwise these queries would be applying a policy to solve the conflict, which violates the t-CRDT principle of all policies being defined in update operations.

Application example To demonstrate the usefulness of the set t-CRDT, consider a distributed chat service in which users can send messages directly to other users or to a chat room. Assume that despite the service having multiple replicas, a client only needs to have an active connection with one replica to be considered online.

For this example, we'll focus on how can the set of currently online users be maintained as accurate as possible. Assume that this set is replicated in all replicas and that a set t-CRDT is used to represent it. As such, logging into the system, establishing new connections and re-connections are represented by *adds*, while logging out, connection drops and interruptions are represented by *removes*.

Assume that we have two replicas, R1 and R2, and that the user Bob has an active connection with replica R1, thus being online. Consider that Bob's connection to R1 is interrupted (e.g., due to a network partition) and that R1 detects this failure and executes a *remove*(Bob) in the set of online users. Now consider that before R1 and R2 communicate, Bob establishes a new connection with R2, represented by *add*(Bob). This *add* and *remove* are concurrent and we intend that *add* wins, since Bob has one active connection and thus should appear as online. For this situation, we can use for *add* and *remove*, respectively, the *normalAdd* and *normalRem* policies, which gives us the intended result.

Consider now a follow-up to the previous scenario in which Bob's connection to R1 is still interrupted and the connection to R2 is active. Assume that Bob sends a logout request to replica R2 and that, while the application Bob is using is closing, the connection to replica R1 is healed. In this case, we once again have a concurrent *remove* and *add* which correspond, respectively, to the logout on replica R2 and the re-connection on replica R1. In this case we can't use *normalRem* for *remove*, as in that case the user would appear as online until replica R1 detected that Bob's connection to R1 isn't active, which could take long. As such, it would be more adequate for the *remove* to use *priorityRem* policy (*add* would still use *normalAdd*), which ensures Bob is removed from the set of online users as soon as he logouts.

To sum it up, any service for which ensuring the set of online users is as accurate as possible is important in spite of any concurrent connection losses, new connections, logouts and logins [2], is an example of an application in which using a set t-CRDT is advantageous over other set CRDTs with only one policy.

[2] A login can be represented by an *add* with *priorityAdd* policy, which ensures it has priority over concurrent logouts for cases such as a user logging in a different replica shortly after doing a logout.

4.1.4 Map

A map is a collection data type formed by pairs of (key, value), with each key being unique (similarly to a set) and "mapping" to a value. Pairs can be added and removed at any time. It is possible to execute queries to check the existence of a certain key, return the value associated to it or the set of all keys or elements present in the map. Maps are widely used in distributed systems, mainly in situations in which objects have some property that uniquely identifies it (e.g., a message ID or the username/mail of a user) and we want to obtain the object's data just by providing that property's value. They are also the basis of key-value store databases.

Specification Algorithm 4.4 contains the specification of an op-based map t-CRDT. Associating an element to a key and storing it is done by *add*, while removing a key and its element is done by *remove*. *Add* receives as arguments a key, an element and a clock (used by *LWW* policy), while *remove* receives a key and a clock. *Contains*, *get*, *keys* and *elements* are the query operations which, respectively, check if a key is in the map, get the element(s) associated to the key and return the set of keys or elements in the map. Finally, *addOp* and *calculateState* are abstractions provided by the generic model.

Algorithm 4.4 op-based map t-CRDT

```
1:  update add (policy hbP, policy selfP, policy otherP, key k, element e, clock clk)
2:      atSource (policy hbP, policy selfP, policy otherP, key k, element e, clock clk) : operation op
3:          let op = [hbPolicy → hbP, selfObsoletePolicy → selfP, otherObsoletePolicy → otherP, type → add, key → k, element → e, clock → clk]
4:      downstream (operation op)
5:          addOp(op)
6:  update remove (policy hbP, policy selfP, policy otherP, key k, clock clk)
7:      atSource (policy hbP, policy selfP, policy otherP, key k, clock clk) : operation op
8:          let op = [hbPolicy → hbP, selfObsoletePolicy → selfP, otherObsoletePolicy → otherP, type → remove, key → k, clock → clk]
9:      downstream (operation op)
10:         addOp(op)
11: query contains (key k) : boolean b    ▷ Assumes that for each key an add and a remove can't be simultaneously active
12:     let b = (∃op ∈ calculateState() : op.type = add ∧ op.key = k)
13: query get (key k) : set v    ▷ Same assumption as contains. Note that depending on the policies multiple elements may be associated to the same key due to concurrency.
14:     let v = {e : ∃op ∈ calculateState() : op.type = add ∧ op.key = k ∧ op.element = e}
15: query keys () : set V                                    ▷ Same assumption as contains
16:     let V = {k : ∃op ∈ calculateState() : op.type = add ∧ op.key = k}
17: query elements () : set E                                ▷ Same assumption as contains
18:     let E = {e : ∃op ∈ calculateState() : op.type = add ∧ op.element = e}
```

Policies Figure 4.10 contains the hb-policy function for the map. Similarly to the set, an operation should only cancel previous operations if the key is the same.

$$mapHB(op, otherOp) \triangleq op.key = otherOp.key$$

Figure 4.10: Happens-before (hb) policy function for the map t-CRDT.

Our library provides four concurrency policies for the map data type, with two of them dealing with concurrent *adds* and *removes* while the other two deal with multiple concurrent *adds*. These policies are, respectively: (i) *addWins* and *remWins*; (ii) *LWW* and *MV*. The specification for the four policies can be found in Figure 4.11.

$$\begin{array}{lcl}
LWW(op, otherOp) & \triangleq & op.key = otherOp.key \ \wedge \ op.selfObsoletePolicy = LWW \\
 & & \wedge \ (otherOp.selfObsoletePolicy = MV \\
 & & \quad \vee \ (otherOp.selfObsoletePolicy = LWW \wedge \ otherOp.clk < op.clk)) \\
MV(op, otherOp) & \triangleq & false \\
remWins(op, otherOp) & \triangleq & op.key = otherOp.key \ \wedge \ op.type = remove \ \wedge \ otherOp.type = add \\
addWins(op, otherOp) & \triangleq & op.key = otherOp.key \ \wedge \ op.type = add \ \wedge \ otherOp.type = remove
\end{array}$$

Figure 4.11: Concurrency policies for the map t-CRDT. The first two are for conflicts between multiple *adds*, while the last two are for conflicts between *adds* and *removes*.

API Figure 4.12 summarizes the API for the map t-CRDT, which contains the supported operations, queries and policies.

ops	=	$add(key, element, clock)$, $remove(key, clock)$
queries	=	$contains(key)$, $get(key)$, $keys()$, $elements()$
happens-before function	=	*mapHB*
policy functions	=	*LWW*, *MV*, *remWins*, *addWins*

Figure 4.12: API for the map t-CRDT

Remarks In the map t-CRDT there are two types of conflicts to solve: (i) which element(s) to keep when two or more concurrent *adds* for the same key happen and (ii) if the key should be kept when a concurrent *add* and *remove* happen for the same key. To simplify the problem but without loss of generality, we separate those two conflicts in different policies and relegate the responsibility of solving the first conflict to *adds* and the second conflict to *removes*.

To solve the first conflict, we provide *MV* and *LWW* policies to be used with *adds* which, respectively, state that all concurrent *adds* or the latest *add* should be kept. In case one or more *adds* with *LWW* and one or more *adds* with *MV* policies are concurrent, only the latest *add* is kept. Note that a dummy value can be used for the clock in *adds* and *removes* when using a policy other than *LWW*, even if some other *adds* use *LWW*.

To solve the second conflict, we provide *addWins* and *remWins* to be used with *removes*. Note that because these policies are used only by *removes*, they don't conflict – a *remove* will never obsolete another *remove*. However, if at least one *remove* with *remWins* is executed, then all concurrent *adds* for the same key will be obsoleted, even if another concurrent *remove* with *addWins* is executed.

To choose which policies to use in an application, the answer to the following questions can be used as a guideline:

1. When multiple concurrent *adds* for the same key are executed, should all elements be kept or only the latest one?
2. When concurrent *adds* and *removes* occur, should the key and the elements specified by those *adds* be kept or removed?

To exemplify, consider that the answer to those questions is, respectively, latest element and remove. As such, for this scenario, the policies to be used are, respectively, *LWW* for *adds* and *remWins* for *removes*. As another example, if the answers were keep the latest element and keep adds, the policies to be used would be, respectively, *LWW* for *adds* and *addWins* for *removes*.

Similarly to the set and register t-CRDTs, the logical clock received as argument is only used by the *LWW* policy and should meet the requirements specified in the register's remarks in Section 4.1.1. For operations using policies other than *LWW*, the clock can have a dummy value.

The queries *contains*, *get*, *keys* and *elements* assume that for the same key there is no situation in which both an *add* and a *remove* are active simultaneously, as it was assumed for *lookup* in the set t-CRDT. The provided policies ensure this property. Note that, however, it is acceptable to have multiple *adds* for the same key but with different elements, which happens when using the *MV* policy. In this case, *get* will return the set of elements associated to that key.

Application example As an example to demonstrate the utility of a map t-CRDT, consider a P2P network of processes executing multiple tests of a distributed algorithm. Assume that the algorithm takes long to execute and the user supervising the system (supervisor) wants to consult statistics from time to time about the progress of the algorithm. Assume also that there is a central server which is responsible for collecting the results of each process and, after a certain percentage of processes finish, the test can be stopped in every process still running it and the system moves on to the next test.

For this situation, we can represent the statistics with multiple map t-CRDTs, with each map being for one type of statistic and each key corresponding to one test (assume that the amount of tests is higher than the amount of different statistics). To simplify, let's focus on only one statistic – the average CPU usage of the test.

While the algorithm is being executed, each process periodically updates the CPU usage map with the latest measured CPU usage since the start of the test in that process.

As these values are intermediate values, it is reasonable to keep all concurrent updates, which allows the supervisor to see, for example, an estimated average of the CPU usage across the replicas. These updates are represented by *adds* with the MV policy.

On the other hand, after a certain amount of processes have finished the ongoing test and the central server is ready to set the final value of the average CPU usage, all concurrent *adds* should be ignored and thus the *LWW* policy should be used with the *add*. Each process that is still executing the test aborts it after seeing the central server's *add*.

After the supervisor receives the test results from the central server, the supervisor can execute a *remove* on the key corresponding to the test. This *remove* should use the *remWins* policy, in order to win versus a replica that may be lagging behind and still be executing the test.

To conclude, situations in which it is necessary to alternate between policies due to users or processes according to the process' or client' priority for *adds* and/or *removes* form examples of situations in which a map t-CRDT is more adequate than other map CRDTs. Another simple example of situations in which our map t-CRDT is adequate is for cases in which different keys should have different policies (e.g., a map with normal and premium users, in which different policies are used to solve conflicts depending if the user is normal or premium).

4.2 Methodology

Our methodology separates the process of specifying t-CRDTs in 3 steps, each one focused on a different problem, thus easing the process of specifying a t-CRDT. The steps are:

1. Specifying update operations, i.e., what will change the state;
2. Specifying query operations, i.e., what and how will the state be read;
3. Specifying the set of policies functions that we intend to use.

The first two parts form the interface of our t-CRDT and need to be specified before the policies, as the operations impact how the policies are specified. A general rule of thumb is to specify the operations and queries without thinking too much about the policies. Not only this keeps the process simpler, but it also allows to add or remove policies at any time if we desire to, without usually requiring changes to the interface.

In the next subsections we'll detail how do to each step. As an example, we'll specify a set t-CRDT with the usual *add*/*remove* operations and three concurrency policies.

4.2.1 Update operations

The first step in our methodology is to define the operations that change the state. Unlike most CRDTs, in t-CRDTs we don't need to worry about how operations change the state, because the state is the set of all operations. Thus, updating the state consists in adding

an operation to the set. As such, what will give a representation to our data type is the arguments of each operation and the query functions.

In t-CRDTs a key aspect is to define the arguments necessary to represent the data type. In practice, the arguments usually consist in:

- the three policy functions as defined in Section 3.2.3 to be be used by this operation;
- a "name" (type) for the operation - this will be useful for specifying the policies, as we need to distinguish between different types of operations, e.g., distinguish an *add* from a *remove*;
- the arguments that are necessary to give a meaning to the operation. In other words, the arguments that are used for that operation for the data type we are trying to represent, in a non-replicated context.
- a clock, if we intend to use *LWW* policies or similar. This can be defined after the policies, as only operations which use *LWW* need to have a clock.

If we consider the set t-CRDT, we have two update operations:

- *add*, which adds an element to the set;
- *remove*, which removes an element from the set.

Both operations need the following arguments:

- the three policy functions that this operation will use (this will be detailed in Section 4.2.3);
- the string "add" (resp. "remove") as a name/type for the operation. Any value can be used, as long as the one used for *add* is different from the one for *remove*;
- the element we want to add (resp. remove).

4.2.2 Queries

The second step in our methodology consists in defining the necessary queries for the data type. Queries are functions that are used to read the state and based on it calculate some kind of result to be returned to the user. In t-CRDTs, queries are also what gives a "meaning" to the data type along with operations, as the state is simply a set of operations.

When specifying a query, assume that only the set of active operations, i.e., the relevant state, is read. This can be achieved by calling the procedure *calculateState* provided by the generic model.

Usually in a t-CRDT query functions consist in searching for the existence of one or more operations that meet a certain criteria. For example, in a set we have at least two query functions: *lookup* and *elements* which, respectively, return if an element is in the set and the set of elements in the set. In a set t-CRDT, these queries can be specified as in Figure 4.13.

Intuitively the *lookup*(*e*) query should consist in checking if an *add* operation whose element is *e* exists. This is in fact enough, as long as we assume that our policies guarantee

$$
\begin{array}{lcl}
\mathit{lookup}(e) & \triangleq & \exists op \in \mathit{calculateState}() : op.\mathit{type} = \mathit{add} \wedge op.\mathit{element} = e \\
\mathit{elements}() & \triangleq & \{e : \exists op \in \mathit{calculateState}() : op.\mathit{type} = \mathit{add} \wedge op.\mathit{element} = e\}
\end{array}
$$

Figure 4.13: Specification of *lookup* and *element* queries for the set t-CRDT.

that in no situation an *add* and *remove* for the same element are active simultaneously. A good rule of thumb when writing query functions is to assume that all conflicts such as this one are solved by the policies and, as such, aren't present in the active state returned by *calculateState*. Not following this rule implies having to apply policies for conflicts in both queries and update operations, which complicates the problem and breaks the principle of having only the update operations define the policies. We'll see later how policies can ensure such properties.

As for *elements*, intuitively it consists in the set of elements for which *lookup* returns true. However, in order to avoid having to call *calculateState* multiple times, we instead define *elements* as the set of elements for which there is an *add* present in the active state, instead of calling *lookup* for every element in the domain.

4.2.3 Policy functions

The final step in our methodology is to define the policies that are intended to be used in the operations of the t-CRDT that is being specified. Policy functions are what defines which operations are still relevant or not and, as such, they are what ensure that the t-CRDT behaves as expected and according to the three principles introduced in Section 2.4.2: PSE, PPE and PCS.

As explained in Section 3.2.3, three types of policy functions need to be defined:

- happens-before policy (hbPolicy);
- selfObsoletePolicy;
- otherObsoletePolicy.

All of these functions receive two operations as arguments and should return true if the first operation makes the second one obsolete.

4.2.3.1 HbPolicy

The *hbPolicy*(*op*, *otherOp*) receives any two operations such that *otherOp* happened-before *op*, where *op* is the operation to which this function is associated to. The goal of this function is thus, for two operations related by happens-before, decide if *otherOp* is made obsolete by *op*.

The *hbPolicy* is the responsible for ensuring the PSE correctness principle, as this property states that sequential updates (i.e., related by happens-before) executed on a t-CRDT should behave as it would in the respective sequential data type. For instance,

in a set t-CRDT, we must ensure that its *hbPolicy* does not obsolete an *add(e)* due to an *remove(f)* that happened-after it, as that would not happen in a sequential set.

In most cases a single *hbPolicy* is enough for all operations in a given t-CRDT, as obsoleting operations by happens-before depends on the data type rather than on the concurrency policies that we want to use. As such, the key aspect for defining a *hbPolicy* function is to find which operations of the data type cancel others that happened-before it and in which conditions that happens.

For the set t-CRDT, the *hbPolicy*(*op*, *otherOp*) should return true only if the element referred by both is the same. A *hbPolicy* for the set which respects the sequential set specification (and thus, PSE) can be found in Figure 4.7.

4.2.3.2 Concurrency policies

Together *selfObsoletePolicy*(*op*, *otherOp*) and *otherObsoletePolicy*(*op*, *otherOp*) specify the intended concurrency semantics for a given operation (*op*). More precisely, when applied to *op*, *selfObsoletePolicy*(*otherOp*, *op*) states when *op* is turned obsolete by other concurrent operations, while *otherObsoletePolicy*(*op*, *otherOp*) states when *op* obsoletes other operations.

Since both functions together control the t-CRDT's concurrency behavior, they are responsible for ensuring both PPE and PCS. For PPE, operations that are naturally commutative in the sequential data type must still be commutative in the t-CRDT (i.e., they should not obsolete each other). As for PCS, we just need to make sure that policies are deterministic and, in conflicting scenarios (e.g: concurrent *remove(e)* and *add(e)*) nothing abnormal happens (e.g: *add(f)* getting obsoleted due to the *remove(e)* and *add(e)* conflict).

We finally note that the user can define the functions for *selfObsoletePolicy* and *otherObsoletePolicy* individually or use a common one for both. Either option is fine, unless the user wants two different semantics for the same operation (*op*) – one for obsoleting other operations and another for when *op* should be obsoleted. For completeness we'll discuss both options but we'll start with the later, as it was the option chosen to define the policies in our t-CRDT library.

Before defining the policy functions it is necessary to decide, in an abstract way, which policies we want to support in the t-CRDT. This is required as we need to specify the policies in a way that ensures they don't conflict when applied to conflicting or non-commutative operations (e.g., an *add* and *remove* of the same element in a set). The solution for this is to define a priority order for the policies, which in practice states which policy wins in a conflicting scenario. Without this, we may have unexpected/undesired results in some situations. To exemplify, consider a set t-CRDT with *add-wins* and *rem-wins* policies and that *add* and *remove* are executed concurrently for the same element. The result depends on how the policies are specified and which ones are associated to each operation, but regardless the result will be one of those:

1. *add* gets obsoleted but *remove* doesn't;

2. *remove* gets obsoleted but *add* doesn't;
3. both operations get obsoleted;
4. both operations don't get obsoleted.

If the policies are specified according to a priority order for conflicting operations, situations (3) and (4) can be avoided. Even though (3) could be acceptable for the set (it would look as if *remove* had won, albeit these semantics aren't clear and thus should be avoided), for other data types such as the register it is usually unacceptable, as if concurrent *assigns* are executed in a register at least one should survive. For the set (4) is unacceptable, as it would imply having the queries apply a policy by themselves.

For specifying most concurrent policies usually it is needed to compare the type of both operations along with other deterministic factors, in order to decide if the second operation should get obsoleted by the first one. For example, in a set t-CRDT, usually both the elements and the operation types are compared, as an *add* only obsoletes a *remove* (and vice-versa) if the elements are the same. One notable exception to this are last-writter-wins policies, which have a global clock abstraction. Defining lww-policies is usually simple, as one just needs to compare the "clock" property (and, in the set case, the element also), which is provided by the system as an argument for the operation.

To exemplify how concurrency policies can be specified, consider that we want two policies for the set t-CRDT: (i) add-wins and (ii) rem-wins. Obviously if we don't take care in specifying these policies, they will conflict, as it is not obvious what should be the result when, for example, an *add* and *remove* for the same element are executed concurrently with, respectively, *add-wins* and *rem-wins* policies. As previously explained, a solution is to define a priority order – in this case, we'll define that we want *removes* with *rem-wins* to win over *adds* with *add-wins*. This can be ensured by having the *add-wins* policy check if the other *remove* has a *rem-wins* policy – if it has, then *add-wins* always returns false.

The *add-wins* and *rem-wins* policies can be specified with three functions, as defined in Figure 4.14. The idea here is that whenever we execute a *remove*, we decide if we want it to have priority (*rem-wins*) or not (*add-wins*). For that effect we use in *removes*, respectively, the *priorityRem* and *normalRem* functions. As for *add*, we should always use *normalAdd*, as it is the *remove* who decides which priority it has.

$normalRem(op, otherOp)$	$\triangleq$	$false$
$normalAdd(op, otherOp)$	$\triangleq$	$op.element = otherOp.element \wedge\ op.type = add$
		$\wedge\ otherOp.type = remove \wedge\ otherOp.selfObsoletePolicy = normalRem$
$priorityRem(op, otherOp)$	$\triangleq$	$op.element = otherOp.element \wedge\ op.type = remove$
		$\wedge\ otherOp.type = add$

Figure 4.14: Concurrency functions for *add-wins* and *remove-wins* policies.

To exemplify how can we add a new policy after having some already defined, consider that we now also want an *add* with top-most priority that wins over any concurrent *removes* for the same element, despite they having priority or not. We'll call this policy

of *priorityAdd-wins*. For this policy we need to define a new function, which we'll call *priorityAdd*. This function must obsolete any concurrent *remove* that has the same element as the *add* with priority. We also need to modify *priorityRem* to make sure it doesn't obsolete *adds* with priority. Both functions can be found in Figure 4.15. Note that no changes are needed for *normalAdd* and *normalRem*, nor to the interface of *add* or *remove*.

$$
\begin{array}{lcl}
priorityRem(op, otherOp) & \triangleq & op.element = otherOp.element \wedge\ op.type = remove \\
 & & \wedge\ otherOp.type = add \wedge\ otherOp.selfObsoletePolicy = normalAdd \\
priorityAdd(op, otherOp) & \triangleq & op.element = otherOp.element \wedge\ op.type = add \\
 & & \wedge\ otherOp.type = remove
\end{array}
$$

Figure 4.15: New *priorityRem* and *priorityAdd* functions which ensure the *priorityAdd-wins* policy.

Specifying selfObsoletePolicy and otherObsoletePolicy separately. As mentioned before, the user might specify these policies with different functions for *selfObsoletePolicy* and *otherObsoletePolicy*. This has the advantage of allowing different policies to be used for the same operation depending if it's checking whenever itself should get obsolete or others. For example, one could use *normalAdd* for *selfObsoletePolicy* and *priorityAdd* for *otherObsoletePolicy*. Another possible advantage is that, for some users, it may be more intuitive to specify policies this way.

The approach to individually specify *selfObsoletePolicy* and *otherObsoletePolicy* is similar to specifying both in just one function, apart from one detail – the user doesn't need to worry about the fact that the function can be applied simultaneously to *selfObsoletePolicy* and *otherObsoletePolicy*. This implies that the user doesn't need to verify, respectively, the type of *otherOp* or *op*, as these will always correspond to the operation these policies are associated to. The user does, however, need to specify two functions for each "policy" – one for *selfObsoletePolicy* and another for *otherObsoletePolicy*. For example, the *priorityRem* previously defined would become the two functions defined in Figure 4.16.

$$
\begin{array}{lcl}
selfPriorityRem(op, otherOp) & \triangleq & op.element = otherOp.element\ \wedge\ op.type = add \\
 & & \wedge\ op.otherObsoletePolicy = otherPriorityAdd \\
otherPriorityRem(op, otherOp) & \triangleq & op.element = otherOp.element\ \wedge\ otherOp.type = add \\
 & & \wedge\ otherOp.selfObsoletePolicy = selfNormalAdd
\end{array}
$$

Figure 4.16: *PriorityRem* functions for *selfObsoletePolicy* and *otherObsoletePolicy*.

To sum it up, if the user intends to use different policies for *selfObsoletePolicy* and *otherObsoletePolicy*, then an individual function for each one is needed. Otherwise, it is entirely up to user preference to specify a common function for both or individual ones.

Chapter 5

Correctness

In this chapter we prove some key properties of our generic model that have been assumed in previous chapters. Namely, we prove that our model is able to provide SEC by assuming every operation is delivered at least once and the policies are deterministic. We even prove that, in fact, it's possible to have some operations that are never delivered to some replicas and still have the same observable state in every replica, similarly to a non-uniform replicated system [9]. We also discuss why can't the three correctness principles introduced in Section 2.4.2 (PSE, PPE and PCS) be ensured by the generic model and, thus, must be guaranteed by each t-CRDT that we specify.

We also verify other relevant properties of our generic model with the aid of TLA+, PlusCal and TLC [18, 19, 36]. We use TLA+ and PlusCal (which translates to TLA+) to write the specification of our generic model and properties that we want to verify, and then we use TLC to check if those properties hold true for every possible state that can be generated by a limited amount of processes and operations (we'll detail this in the relevant section). Verifying these properties is essential, as besides guaranteeing that the model works as expected, it also means that there are less concerns that the user needs to worry about when specifying policies. For example, ensuring convergence means that the user doesn't need to worry if their policies ensure convergence or not, as that is guaranteed by the model itself.

Finally, to conclude the chapter, we'll show how can TLA+ and TLC be used to verify the correctness of a specific t-CRDT. Namely, we'll explain how can operations, policies and queries be specified and which properties should be verified. Verifying correctness properties of a t-CRDT is important to ensure that the policies behave as expected, as otherwise unexpected results may occur. For instance, queries may return incorrect results if those properties don't hold. One example of such a property that has been assumed frequently for multiple t-CRDTs in Section 4.1 is "if two or more conflicting

operations occur concurrently, at most one survives". We'll show how can this be verified by specifying the set t-CRDT (including the policies) and some of its properties in TLA+.

5.1 Proofs of convergence and network requirements

In the previous chapters we assumed that if all operations are eventually delivered at least once to every replica and the policies are deterministic, then the generic model guarantees eventual convergence of the state or, more precisely, Strong Eventual Consistency (SEC) [28]. In this section we'll provide a proof for that claim, using the pseudo-code in Algorithm 3.1 as a basis.

We consider to have state convergence if, for any two replicas replicating a t-CRDT who received the same operations, the set of operations O (i.e., the state of the generic model) after executing *calculateState*() is the same. We also consider that two replicas have equivalent observable state if the result returned by *calculateState*() is the same.

Theorem 5.1 (exactly once delivery guarantees SEC). *If the happens-before policies of every operation are deterministic, transitive and all policy functions eventually terminate, then exactly once delivery ensures SEC.*

Proof. To start, assume that even with exactly once delivery it would be possible to have different states in replicas $R1$, $R2$ if $op1$ and $op2$ are delivered by different orders. Assume that $op1 \prec op2$ and that $hbOp1$ and $hbOp2$ are the set of operations that are obsoleted through happens-before by, respectively, $op1$ and $op2$. If $op1$ isn't obsoleted through happens-before by $op2$, then the final result of O will be its initial value excluding operations in $(hbOp1 \cup hbOp2)$, as the final result is the same independently of whichever is eliminated first. Otherwise, independently of whichever is delivered first, $op1$ will be removed from O due to $op2$. Even if $op2$ gets removed due to another operation, then due to transitivity $op1$ will also be removed by that operation. Similar conclusions can be made for $op2 \prec op1$. If $op1 \parallel op2$, then neither of them will remove each other from O and, thus, the state is the same. However this is a contradiction, as for any delivery order we got the same result. Finally, we note that *calculateState* and *addOp* always terminate, as long as the policy functions also terminate. Thus, exactly once delivery guarantees SEC.□

Using the previous proof as a basis, we prove now that at least once delivery (a weaker delivery guarantee) is enough to have SEC.

Theorem 5.2 (at least once delivery guarantees SEC). *If the happens-before policies of every operation are deterministic, transitive and all policy functions eventually terminate, then at least once delivery ensures SEC.*

Proof. Now assume that delivering an operation $op1$ more than once could lead to a different state from just delivering once. Adding an existing element to a set doesn't change its state, so the result of executing *calculateState*() (which is deterministic if all

policies also are) after *op1* is delivered one or more times is always the same, assuming that it wasn't ever deleted from O. If we assume otherwise, then that means that there exists at least one *op2* in O such that *op2* obsoletes *op1* by happens-before. In that case, considering the previous proof and that happens-before policies are deterministic and transitive, then we conclude that there will always be an *op3* in O, possibly different from *op2*, such that $op1 \prec op3$ and *op3* obsoletes *op1* by happens-before. Thus, we conclude that delivering an operation one or more times always leads to the same final result. □

At the start of this chapter we made a bold statement – we guarantee the same observable state in every replica even if certain operations aren't ever delivered to all replicas. Obviously this doesn't apply to all operations, but rather only to some. In fact, it only applies to operations that were already obsoleted by happens-before by some other operation. This is due to the fact that *calculateState*() deletes those operations from the state as they are no longer relevant for the observable state.

Theorem 5.3 (operations obsoleted by happens-before don't need to be delivered). *For every operation* op1 *such that there exists* op2 *where* $\text{op1} \prec \text{op2}$ *and* op2 *obsoletes* op1 *by happens-before, then every replica will have the same state as long as* op2 *is delivered to every replica.*

Proof. Assume that there are two operations such that $op1 \prec op2$ and *op2* obsoletes *op1* by happens-before. Assuming that all replicas *R1*, ..., *Rn* have the same state, if we deliver *op1* and *op2* to all replicas, then after each one executes *calculateState()* at least once they will all have the same operations in O and none will have *op1* in O. As such, if we never deliver *op1* to some replicas in *R1*, ..., *Rn* but deliver *op2* to all, then they will still have the same O after executing *calculateState()*. Thus, we conclude that delivering operations obsoleted by happens-before isn't necessary to have the same observable state. □

An important property to prove is that if all replicas have the same state, then the state observed by the user (i.e., the one returned by *calculateState*()) is the same in every replica.

Theorem 5.4 (equivalent state implies equivalent observable state). *If any two replicas have equivalent state, then they also have equivalent observable state, as long as all policy functions are deterministic.*

Proof. Assume that all replicas have equivalent states (i.e., the set O is equal). It was already proven that for equivalent O the result of applying the happens-before policies is the same, as long as they are deterministic and transitive. As such, if concurrency policies are also deterministic, then the result of *calculateState*() is the same in any two replicas with equivalent state. This is due to the fact that the order for which policies are applied doesn't matter (all policies of operations that weren't removed by happens-before are applied to every operation) and the set of operations for which the policies will be

applied is the same. Since the result of *calculateState*() is the same, then we have the same observable state as long as all policies are deterministic. □

5.2 TLA+, PlusCal and TLC

TLA+ [18] is a specification language developed by Leslie Lamport which allows to precisely define the behavior of a system by using simple mathematics. Since it is a specification language, it is more expressive than traditional programming languages, allowing to specify complex systems and properties with a relatively short amount of code. For example, the *write* policy for the counter defined in Section 4.1.2 can be specified in TLA+ as: *write*(op, otherOp) $\triangleq$ otherOp.type = INC $\vee$ otherOp.type = DEC. In TLA+ this is known as the definition of an operator, which is similar to defining a function in a traditional programming language.

PlusCal [19] is an algorithm language that let's the user write algorithms which are then translated to TLA+ specifications. Since any TLA+ expression is valid in PlusCal, it is as expressive as TLA+ yet simpler to start using due to its syntax similarities with other programming languages. Algorithm 5.1 includes a simple PlusCal algorithm that increments a variable. This small example is enough to show that, similarly to programming languages, it is possible to declare variables (keyword "variables") and alter their value by using ":=". Between "begin" and "end algorithm" is where the algorithm's main code is specified. Auxiliary "methods" can be defined by the keywords "macro" and "procedure". Procedures have the advantage of allowing local variables to be defined but force the usage of a "label" after each call to a procedure, which implies that more steps in an algorithm can be concurrent and thus, more states are generated, slowing down the verification process when compared to macros.

Algorithm 5.1 Simple PlusCal algorithm

```
1: EXTENDS Naturals
2: (*–algorithm increment
3: variables a = 0;
4: begin
5: a := a + 1
6: end algorithm)
```

TLC [36] is a model checker for TLA+ specifications. The basis of a model checker is to generate, for a given specification/algorithm, all possible states that can be reached and verify for them all if properties/invariants of the system are respected. This implies that TLC can only check a subset of TLA+ specifications, as it is possible to write correct TLA+ that generates infinite states. In order to verify a specification with TLC a model needs to be defined, which consists in defining which invariants should be checked and the value of the constants defined in the specification.

In this document we won't detail on how to write TLA+ specifications nor PlusCal algorithms, as it is out of scope of this document. We will, instead, focus on showing how can both languages be used to verify properties of both our generic CRDT model and specific t-CRDTs. Tutorials for both TLA+ and PlusCal can be found on [18, 20, 29]

5.3 Correctness principles - PSE, PPE and PCS

As we previously mentioned, our generic model can't ensure by itself PSE, PPE or PCS. The reason for this is simple – these principles detail how a replicated data type should behave both in sequential (PSE) and concurrent scenarios (PPE and PCS). However, the sequential and concurrent behavior are entirely defined by, respectively, the *hbPolicy* and the two concurrent policies (*selfObsoletePolicy* and *otherObsoletePolicy*). As such, it must be the policies to ensure the referred properties.

As an example, consider that we have a set t-CRDT and we define that all the policies return true for any pair of operations. This theoretically meets our model's policy requirements, as *hbPolicy* is transitive and both *hbPolicy*, *selfObsoletePolicy* and *otherObsoletePolicy* are deterministic and eventually terminate. However, this goes against PSE, PPE and PCS, as the set t-CRDT with those policies would not behave as a sequential set for sequential updates, would not respect commutativity and executing an operation for element e obsoletes any previous or concurrent operation for any element.

As such, we believe it is impossible to specify a model as generic as ours and that ensures these properties. Both PSE, PPE and PCS require data type specific behavior, which limits generality. Thus, we relegate the responsibility of ensuring those properties for the policies of each t-CRDT.

5.4 TLA+ specification of the generic model

In order to verify invariants for the generic model (or any t-CRDT) using TLC we first need to write a TLA+ specification that describes the behavior of the generic model. Albeit the pseudo-code for the model is relatively short, there are still multiple parts that need to be specified in TLA+ in order to test its behavior:

1. a representation of the state and how are operations and their arguments stored;
2. the ability to identify when two operations are concurrent or one happened-before another;
3. a way to apply policy functions and associate them to operations;
4. two operators which collect all operations that, respectively, are obsoleted by happens-before or concurrency;
5. an operator that returns the state observed by query operations (i.e., the set of active operations);

6. a representation of multiple processes executing operations and communicating between themselves to share their updates.

An essential part of specifying our generic model is to specify the behavior of the procedure *calculateState*(), which corresponds to steps 4 and 5. In the rest of this section we'll discuss how can each part be specified in order to get a specification that accurately represents the model and, after the operations, policies and invariants of a specific t-CRDT are specified, is verifiable by TLC.

1: State and operations representation. The state of the generic model in one replica is simply a set which is then used to store the operations that are generated by that replica and received from others. This can be represented as: **variables** ops = {}.

Adding an operation *op* is relatively simple if we use PlusCal, since it simply consists in defining *ops* as being *ops* with the union of *op*. We can define a macro that receives an operation and stores it as in Figure 5.1.

```
macro addOp(op)
begin
  ops := ops ∪ {op};
end macro;
```

Figure 5.1: PlusCal macro to store an operation.

To represent an operation we can use the record data type that TLA+ provides, which is similar to structs or records of common programming languages. Thus, we can associate to each value we want to store in the operation a key and then consult the value by using the respective key. Since the arguments of an operation depend on the data type that the t-CRDT is representing, we can't define all arguments yet, but we can at least define the keys for arguments that are common to most t-CRDTs, which are:

- "type", which stores the op's name;
- "hist", which stores the necessary information for determining wherever op is concurrent, happens-before or happens-after any other operation;
- "hb", "selfP" and "otherP" which store, respectively, op's happens-before, selfObsoletePolicy and otherObsoletePolicy functions;
- "id", which stores the identifier of the replica that generated this operation. This id is used to distinguish between operations generated by different replicas.

Each t-CRDT will have to define one macro for each type of update operation, similarly to what was done in Section 4.1. It is then that macro's responsibility to both generate and store the operation with all necessary arguments.

2: Identify if two operations are concurrent/happens-before. As mentioned before, each operation stores the necessary information to check for concurrency or happens-before in the "hist" field. One way of checking that is to have each operation store in that field all operations that happened-before it. Remember that a specification isn't the same as an implementation, so space efficiency is not as important as readability of the specification. Since each operation knows what happened before it, then for any two given operations *op*, *otherOp*, checking if $otherOp \prec op$ consists in checking if *otherOp* is in *op*'s hist. On the other hand, checking for concurrency is straightforward – one just needs to check that neither $otherOp \prec op$ nor $op \prec otherOp$ return true, and that the operations are different. Figure 5.2 shows how can this be specified in TLA+ by using two operators.

$$
\begin{array}{lcl}
\mathit{happenedBefore}(\text{op}, \text{otherOp}) & \triangleq & \text{otherOp} \in \text{op.hist} \\
\mathit{concurrent}(\text{op}, \text{otherOp}) & \triangleq & \text{op} \neq \text{otherOp} \\
 & & \land \neg\, \mathit{happenedBefore}(\text{op}, \text{otherOp}) \\
 & & \land \neg\, \mathit{happenedBefore}(\text{otherOp}, \text{op})
\end{array}
$$

Figure 5.2: TLA+ operators for identifying if two operations are related by happens-before or are concurrent.

3: Applying and associating policy functions. Each operation must have associated to it the policies to apply, or some identifier for them. Doing the later is actually preferred – it makes it easier to read, since it's better to read a constant name (e.g., PRIORITY_ADD) than a function and its arguments. If we use identifiers, we can then use an auxiliary operator that, given a policy name and the operations to apply the policy, chooses the right function to apply. This operator needs to be refined for each t-CRDT, since each one will define different policies depending on the data type. Generally, the operator looks similar to the example in Figure 5.3.

$$
\begin{array}{lcl}
\mathit{applyPolicy}(\text{policy}, \text{op}, \text{otherOp}) \triangleq & & \\
\quad \textbf{CASE } \text{policy} = \text{POLICY\_ONE} & \rightarrow & \mathit{policyOne}(\text{op}, \text{otherOp}) \\
\quad \square\ \text{policy} = \text{POLICY\_TWO} & \rightarrow & \mathit{policyTwo}(\text{op}, \text{otherOp}) \\
\quad \ldots & & \\
\quad \square\ \text{policy} = \text{LAST\_POLICY} & \rightarrow & \mathit{lastPolicy}(\text{op}, \text{otherOp})
\end{array}
$$

Figure 5.3: Example of a TLA+ operator for choosing a policy to apply.

For simplicity, we also define a similar operator named *applyHB*. As such, we refer to happens-before policies in *applyHB*, while concurrent policies are referred in *applyPolicy*.

4: Collecting operations obsoleted by happens-before or concurrency. The first part of specifying the procedure *calculateState*() is to determine the set of operations that are obsoleted by happens-before policies. Figure 5.4 contains an operator which corresponds to the translation of line 8 in Algorithm 3.1 (Section 3.2) to TLA+. *HappenedBefore* is the

previously defined operator that specifies $op \prec otherOp$ and *applyHB* applies *otherOp*'s happens-before policy. *Ops* corresponds to the replica's *ops* variable (this is needed due to each replica having its own *ops* variable).

```
getHB(ops) ≜ {op ∈ ops :
                ∃ otherOp ∈ ops :
                    happenedBefore(otherOp, op)
                    ∧ applyHB(otherOp.hb, otherOp, op)}
```

Figure 5.4: TLA+ operator for calculating set of operations obsoleted by happens-before.

On the other hand, Figure 5.5 contains an operator for collecting operations obsoleted by concurrency, which corresponds to the second part of *calculateState*() and is the translation to TLA+ of line 10 in Algorithm 3.1. Note that in this case *ops* corresponds to the variable *ops* after removing the operations obsoleted by happens-before, that is, ops \ *getHB*(ops). *Concurrent* and *applyPolicy* are the previously defined operators that, respectively, specify $op \parallel otherOp$ and apply a concurrency policy.

```
getConcurrentObs(ops) ≜ {op ∈ ops :
                           ∃ otherOp ∈ ops :
                               concurrent(otherOp, op)
                               ∧ (applyPolicy(op.selfP, otherOp, op)
                                   ∨ applyPolicy(otherOp.otherP, otherOp, op))}
```

Figure 5.5: TLA+ operator for calculating set of operations obsoleted by concurrency.

5: Returning the set of active operations. The final part of specifying the procedure *calculateState*() is to use the two previously defined operators to calculate the set of active operations. This can be achieved with the operator in Figure 5.6.

```
getActive(ops) ≜ LET nonHB ≜ ops \ getHB(ops)
                     concurrent ≜ getConcurrentObs(nonHB)
                 IN nonHB \ concurrent
```

Figure 5.6: TLA+ operator for the procedure *calculateState*().

GetActive in Figure 5.6 corresponds to the whole *calculateState*() procedure, except for the side-effect of updating *ops*, which this operator doesn't do. Note that updating O (*ops*) when executing *calculateState*() is optional as it is, in fact, just an optimization. We have already proven before in Section 5.1 that this optimization doesn't affect the correctness of the generic model.

The TLA+ constructor **LET IN** is used to create two temporary variables which store, respectively, the set of all operations except the ones obsoleted by happens-before and

the set of operations obsoleted by concurrency. These temporary variables are defined between **LET** and **IN**, while the final result returned by the operator is defined after **IN**.

With the first five steps we have specified the complete behavior of the generic model.

6: Multiple processes executing operations. Since we already defined the generic model in TLA+, what is left to have a complete specification is to define an algorithm that simulates multiple processes executing operations and communicating with each other. To define this, we'll use PlusCal.

The algorithm executed by each replica/process corresponds to a *while* cycle which executes operations until a certain number of operations (per replica) are executed. Each time a process wants to do an action, it can either choose to execute one of the possible operations provided by the t-CRDT or choose to communicate with another replica to receive an operation that it doesn't yet know and execute it. If that replica already executed the max number of operations, then it asks for the remaining operations of the other replicas. Unfortunately defining the set of possible operations depends on the type of t-CRDT that is being specified, which means we can only specify part of the algorithm (the rest must be changed for each t-CRDT).

Algorithm 5.2 contains the specification of the algorithm executed by each replica. First we define some constants whose values will be defined by the models we run in TLC. Of the variables that we define, we highlight two of them – *finished* and *msgs*. *Finished* is a structure that signals for each replica if it already finished the algorithm or not, which is useful for verifying properties that are only valid at the end, e.g., convergence. As for *msgs*, it is used to represent the communication between replicas, where each key is associated to one replica and stores all operations that replica doesn't yet know. *Self* represents the ID of the process who is executing the code.

The main cycle consists in each process executing actions until it knows all operations of all replicas. An action can be either creating an operation or receiving another's process operation that wasn't yet known in that replica. Each replica creates as many operations as defined by *MAX*. The final **else** ensures that independently of the choices made when advancing states, eventually all replicas have every operation.

Operator to save operations to the shared buffer. Albeit unused in the generic model, we define an extra operator in Figure 5.7 which is used to add a new operation to the operation buffer *msgs*, whenever a new operation is created. This operator represents the propagation of a local operation to all other replicas by adding the operation to every entry of *msgs*, except to the process who created the operation, since that one already knows it. This should be used when specifying the operations of a t-CRDT, as each operation must be saved both on the replica's *ops* variable and on the *msgs* buffer.

Invariants. Even though we can't run the generic model specification (as it lacks types of operations) on its own, we can, however, specify invariants that will be verified when

Algorithm 5.2 Main algorithm

```
1:  CONSTANTS MAX_OP, REPLICAS

2:  variables msgs = [x ∈ REPLICAS ↦ ⟨⟩], finished = [x ∈ REPLICAS ↦ false]
                                                    ▷ Variables shared between processes

3:  process replica ∈ REPLICAS
4:  variables ops = {}, actions = 0, headOp;        ▷ Variables specific to each process
5:  begin
6:      While:
7:          while (Cardinality(ops) < (Cardinality(REPLICAS) * MAX_OP) do
8:              if (actions < MAX_OP) then
9:                  either
10:                     ▷ t-CRDT specific code that chooses one of the possible operations and
    increments actions
11:                 or
12:                     if (Len(msgs[self]) > 0) then
13:                         headOp := Head(msgs[self]);
14:                         msgs[self] := Tail(msgs[self]);
15:                         addOp(headOp);
16:                     end if;
17:                 end either;
18:             else
19:                 await (Len(msgs[self]) > 0);
20:                 headOp := Head(msgs[self]);
21:                 msgs[self] := Tail(msgs[self]);
22:                 addOp(headOp);
23:             end if;
24:         end while;
25:     End:
26:         finished[self] := TRUE;
27: end process
```

$$
\begin{array}{rcl}
saveOp(\text{op, me}) & \triangleq & [r \in REPLICAS \rightarrow \\
 & & \textbf{IF } \text{r} = \text{me} \\
 & & \textbf{THEN } \text{msgs[r]} \\
 & & \textbf{ELSE } \text{Append(msgs[r], op)}]
\end{array}
$$

Figure 5.7: TLA+ operator to save an operation to the shared buffer *msgs*.

the specification is completed with a specific t-CRDT. Specifying an invariant is just like specifying any other operator in TLA+, except that its result must be a boolean. One obvious invariant to verify for every t-CRDT is convergence, that is, if after all operations are executed the states are equal in every replica. Figure 5.8 contains a rigorous verification of convergence.

The invariant only verifies if there's convergence when all replicas have already executed all operations (which is represented by finished[p] = TRUE). Here convergence is

$$
\begin{array}{lcl}
\mathit{convergence} & \triangleq & (\forall\ \mathrm{r} \in \mathrm{REPLICAS} : \mathrm{finished}[\mathrm{r}] = \mathrm{TRUE}) \implies \\
 & & \quad (\forall\ \mathrm{r1} \in \mathrm{REPLICAS},\ \mathrm{r2} \in \mathrm{REPLICAS} : \\
 & & \quad\quad \mathit{sameState}(\mathrm{r1}, \mathrm{r2})) \\
\mathit{sameState}(\mathrm{r1}, \mathrm{r2}) & \triangleq & \mathit{sameOPs}(\mathrm{r1}, \mathrm{r2}) \land \mathit{sameHB}(\mathrm{r1}, \mathrm{r2}) \land \mathit{sameObs}(\mathrm{r1}, \mathrm{r2}) \\
\mathit{sameOPs}(\mathrm{r1}, \mathrm{r2}) & \triangleq & \mathrm{ops}[\mathrm{r1}] = \mathrm{ops}[\mathrm{r2}] \\
\mathit{sameHB}(\mathrm{r1}, \mathrm{r2}) & \triangleq & \mathit{getHB}(\mathrm{ops}[\mathrm{r1}]) = \mathit{getHB}(\mathrm{ops}[\mathrm{r2}]) \\
\mathit{sameObs}(\mathrm{r1}, \mathrm{r2}) & \triangleq & \mathit{getConcurrentObs}(\mathrm{ops}[\mathrm{r1}]) = \mathit{getConcurrentObs}(\mathrm{ops}[\mathrm{r2}])
\end{array}
$$

Figure 5.8: TLA+ invariant for convergence

defined as all replicas having the same operations in *ops* and the same result of *getHB* and *getConcurrentObs*. Note that we don't need to verify if *getActive* returns the same result, as this is implied by the others. Another important aspect is that it is safe to verify if *ops* is equal in every replica, as this specification doesn't include the optimization of removing operations obsoleted by happens-before from the state – if it did, then we would need to first call *getActive* and only afterwards verify, as it would be correct for *ops* to diverge between replicas until *getActive* is executed.

A curious reader might notice that here *ops* was accessed as if it was an array. This is because when PlusCal is translated to TLA+, it converts the local variables of each process to a structure, with the keys being the identifiers of each process. As such, if we want to access those variables outside of PlusCal, we need to index them by process.

5.5 TLA+ specification of a t-CRDT

Specifying t-CRDTs in TLA+ and verifying if certain invariants hold is of utmost importance as it ensures the CRDT behaves as expected and that what was assumed as true when specifying the policies in, in fact, true. Fortunately specifying t-CRDTs is relatively simple when compared to specifying the generic model, as not much needs to be done besides defining the queries and operations. Unfortunately, as it is for verifying any system's correctness, the main difficulty is on specifying the right invariants.

To specify a t-CRDT, the following steps need to be done:

1. specify the supported operations;
2. specify the supported queries;
3. specify the supported policies;
4. update the algorithm that chooses operations (step 6 of the generic model specification) in order to choose one of the t-CRDT's operations;
5. specify invariants that are data-type specific;
6. specify invariants that verify the correctness properties introduced in Section 2.4.2 (PSE, PPE and PCS);
7. specify one or more models to be ran on TLC.

We have written and verified TLA+ specifications for all the t-CRDTs that are present

in Section 4.1. However, for compactness reasons, we won't present them all – we'll only include the set t-CRDT, as an illustration for the process of specifying a t-CRDT in TLA+.

1: Operations. Specifying an operation consists in defining a macro that creates an operation with the necessary information and then executes two operators: *saveOp* and *addOp* to, respectively, store the op in the shared buffer (*msgs*) and on the local replica state (*ops*). It is also necessary to define the constants that represent the possible types of operations. Unfortunately, PlusCal doesn't allow us to define temporary variables in macros, which implies that the code for creating the operation needs to be duplicated. An alternative would be to define a procedure, but they are slower.

For the set t-CRDT, we have two operations: *add* and *remove*. Algorithm 5.3 contains a PlusCal representation of both.

Algorithm 5.3 PlusCal code for *add* and *remove* operations

```
1:  CONSTANTS ADD, REMOVE
2:  macro addSource (elem, selfObsoletePolicy, otherObsoletePolicy, happensBefore, ts)
3:  begin
4:      msgs := saveOp([type ↦ ADD, e ↦ elem, selfP ↦ selfObsoletePolicy,
5:  otherP ↦ otherObsoletePolicy, hb ↦ happensBefore, hist ↦ {h ∈ ops: TRUE},
6:  id ↦ self, clk ↦ ts], self)
7:      addOp([type ↦ ADD, e ↦ elem, selfP ↦ selfObsoletePolicy, otherP ↦ otherObso-
    letePolicy, hb ↦ happensBefore, hist ↦ {h ∈ ops: TRUE}, id ↦ self, clk ↦ ts])
8:  end macro;
9:  macro removeSource (elem, selfObsoletePolicy, otherObsoletePolicy, happensBefore,
    ts)
10: begin
11:     msgs := saveOp([type ↦ REMOVE, e ↦ elem, selfP ↦ selfObsoletePolicy,
12: otherP ↦ otherObsoletePolicy, hb ↦ happensBefore, hist ↦ {h ∈ ops: TRUE},
13: id ↦ self, clk ↦ ts], self)
14:     addOp([type ↦ REMOVE, e ↦ elem, selfP ↦ selfObsoletePolicy, otherP ↦ other-
    ObsoletePolicy, hb ↦ happensBefore, hist ↦ {h ∈ ops: TRUE}, id ↦ self, clk ↦ ts])
15: end macro;
```

Since we support *LWW* policies, we also need to implement a logical clock that provides a total order consistent with causality. We'll see later how can this be achieved. Note that when running TLC on this specification, if the policies to be tested don't need a clock, then the user should either remove the field *clk* or associate to it a default value (e.g., 0), in order to prevent TLC from generating unnecessary states in which the only difference is the clk's value.

2: Queries. Supporting the queries of a t-CRDT in TLA+ consists in translating the queries in the formal specification of the t-CRDT such as the ones given in Section 4.1 to TLA+. In most cases the translation is straightforward (e.g., *lookup*), but in others it may require some adaptations (e.g., *elements*).

For the set t-CRDT whose specification is on Section 4.1.3, the two queries *lookup* and *elements* can be specified in TLA+ as in Figure 5.9.

```
lookup(ops, elem)  ≜  ∃ op ∈ getActive(ops) : op.type = ADD ∧ op.e = elem
elements(ops)      ≜  LET active ≜ getActive(ops)
                      IN          {e ∈ ALL_ELEMENTS : ∃ op ∈ active :
                                      op.type = ADD ∧ op.e = e}
```

Figure 5.9: TLA+ specification of the *lookup* and *element* queries.

Lookup corresponds to a direct translation to TLA+ and should be fairly simple to understand. As for *elements*, the translation is not direct as we need to specify the domain of possible elements (ALL_ELEMENTS) before we actually specify which elements are in the set. This is necessary due to the fact that the construct " : " in TLA+ is actually a set filter. Initially, TLA+ creates a set with all elements specified before ":" and afterwards it filters them, in order to keep only the elements for which the part after ":" is true.

3: Policies. Supporting policies in TLA+ usually requires three steps:

- defining the constants that represent the identifier/name of each policy;
- translating the policies' formal specification to TLA+;
- updating *applyPolicy* and *applyHB* operators to include the policies.

However, if the policies use functionalities that are not directly supported by the generic model (such as a logical clock for *LWW*), we also need to specify those functionalities. We'll see later how can that be done for the logical clock case.

For set t-CRDT's case, the TLA+ operators for the policies specified in Section 4.1.3 can be found in Figure 5.10.

As for *applyPolicy* and *applyHB*, their TLA+ specification can be found on Figure 5.11.

4: Updating the algorithm. In most cases, updating the algorithm consists in specifying which operations and arguments can each process choose in each action (i.e., completing line 10 of Algorithm 5.2). In this case, however, extra changes are required, since we also want to support a logical clock with total order guarantees (and consistent with causality) for *LWW* policy. One simple way in TLA+ to specify this is to define a *globalClk* variable, which consists in a shared counter – each time a replica does an operation, it increments the counter and associates its value to the operation. Since each cycle iteration is atomic, it's impossible to have two different operations with the same clock value. Note that while it would be possible to use a vector clock per replica instead of a shared clock, it would add complexity that is irrelevant for the algorithm.

Algorithm 5.4 contains the specification of a modified main algorithm, in order to incorporate the necessary changes to support the set t-CRDT. Line 1 contains all previously defined constants, while line 2 contains both the old variables and the new *globalClk* variable. Other than that, the rest of the code is unmodified, apart from the addition of lines

Algorithm 5.4 Set t-CRDT main algorithm

```
1:  CONSTANTS MAX_OP, REPLICAS, ALL_ELEMENTS, ADD, REMOVE, NORMAL_-
    ADD, NORMAL_REMOVE, PRIORITY_ADD, PRIORITY_REMOVE, LWW, SET_HB

2:  variables msgs = [x ∈ REPLICAS ↦ ⟨⟩], finished = [x ∈ REPLICAS ↦ false], globalClk
    = 0                                          ▷ Variables shared between processes

3:  process replica ∈ REPLICAS
4:  variables ops = {}, actions = 0, headOp;     ▷ Variables specific to each process
5:  begin
6:      While:
7:          while (Cardinality(ops) < (Cardinality(REPLICAS) * MAX_OP) do
8:              if (actions < MAX_OP) then
9:                  either
10:                     globalClk := globalClk + 1;
11:                     actions := actions + 1;
12:                     with e ∈ ALL_ELEMENTS do
13:                         with p ∈ ADD_POLICIES do
14:                             addSource(e, p, p, SET_HB, globalClk);
15:                         end with;
16:                     end with;
17:                 or
18:                     globalClk := globalClk + 1;
19:                     actions := actions + 1;
20:                     with e ∈ ALL_ELEMENTS do
21:                         with p ∈ REMOVE_POLICIES do
22:                             removeSource(e, p, p, SET_HB, globalClk);
23:                         end with;
24:                     end with;
25:                 or
26:                     if (Len(msgs[self]) > 0) then
27:                         headOp := Head(msgs[self]);
28:                         msgs[self] := Tail(msgs[self]);
29:                         addOp(headOp);
30:                     end if;
31:                 end either;
32:             else
33:                 await (Len(msgs[self]) > 0);
34:                 headOp := Head(msgs[self]);
35:                 msgs[self] := Tail(msgs[self]);
36:                 addOp(headOp);
37:             end if;
38:         end while;
39:     End:
40:         finished[self] := TRUE;
41: end process
```

CONSTANTS NORMAL_ADD, NORMAL_REMOVE, PRIORITY_ADD, PRIORITY_REMOVE, LWW, SET_HB

$$
\begin{array}{lcl}
\mathit{hbSet}(\text{op, otherOp}) & \triangleq & \text{op.e} = \text{otherOp.e} \\
\mathit{normalRem}(\text{op, otherOp}) & \triangleq & \text{FALSE} \\
\mathit{normalAdd}(\text{op, otherOp}) & \triangleq & \text{op.e} = \text{otherOp.e} \\
 & & \quad \wedge\ \text{op.type} = \text{ADD} \\
 & & \quad \wedge\ \text{otherOp.type} = \text{REMOVE} \\
 & & \quad \wedge\ \text{otherOp.selfP} = \text{NORMAL\_REM} \\
\mathit{priorityRem}(\text{op, otherOp}) & \triangleq & \text{op.e} = \text{otherOp.e} \\
 & & \quad \wedge\ \text{op.type} = \text{REMOVE} \\
 & & \quad \wedge\ \text{otherOp.type} = \text{ADD} \\
 & & \quad \wedge\ \text{otherOp.selfP} = \text{NORMAL\_ADD} \\
\mathit{priorityAdd}(\text{op, otherOp}) & \triangleq & \text{op.e} = \text{otherOp.e} \\
 & & \quad \wedge\ \text{op.type} = \text{ADD} \\
 & & \quad \wedge\ \text{otherOp.type} = \text{REMOVE} \\
 & & \quad \wedge\ \text{otherOp.selfP} \neq \text{LWW} \\
\mathit{lww}(\text{op, otherOp}) & \triangleq & \text{op.e} = \text{otherOp.e} \\
 & & \quad \wedge\ \text{op.selfP} = \text{LWW} \\
 & & \quad \wedge\ (\text{otherOp.selfP} \neq \text{LWW} \\
 & & \qquad \vee\ \text{op.clk} > \text{otherOp.clk})
\end{array}
$$

Figure 5.10: TLA+ specification of the set policies defined in Section 4.1.3.

$$
\begin{array}{lcll}
\mathit{applyPolicy}(\text{policy, op, otherOp}) \triangleq & & & \\
\quad \textbf{CASE}\ \text{policy} = \text{NORMAL\_REM} & \rightarrow & \mathit{normalRem}(\text{op, otherOp}) & \\
\quad \square\ \text{policy} = \text{NORMAL\_ADD} & \rightarrow & \mathit{normalAdd}(\text{op, otherOp}) & \\
\quad \square\ \text{policy} = \text{PRIORITY\_REM} & \rightarrow & \mathit{priorityRem}(\text{op, otherOp}) & \\
\quad \square\ \text{policy} = \text{PRIORITY\_ADD} & \rightarrow & \mathit{priorityAdd}(\text{op, otherOp}) & \\
\quad \square\ \text{policy} = \text{LWW} & \rightarrow & \mathit{lww}(\text{op, otherOp}) & \\
\mathit{applyHB}(\text{policy, op, otherOp}) & \triangleq & \mathit{hbSet}(\text{op, otherOp}) &
\end{array}
$$

Figure 5.11: TLA+ code of *applyPolicy* and *applyHB* operators adapted to the set.

10-24, which contain the logic for choosing operations. Since we want to generate every possible state, we need to instruct TLC to create states for all combinations of possible policies, elements and operations, which is achieved by the **either** and **with** constructs.

ADD_POLICIES and REMOVE_POLICIES can be defined by either the model or in the specification. Their usage is to represent the set of the policies that TLC can choose from for each operation. If the intention is to define in the model, then they need to be added to the constants. Otherwise, they can be defined as shown in Figure 5.12.

$$
\begin{array}{lcl}
\text{ADD\_POLICIES} & \triangleq & \{\text{NORMAL\_ADD, PRIORITY\_ADD, LWW}\} \\
\text{REMOVE\_POLICIES} & \triangleq & \{\text{NORMAL\_REMOVE, PRIORITY\_REMOVE, LWW}\}
\end{array}
$$

Figure 5.12: Set of possible policies to be used in *add* and *remove* operations.

5: Data-type specific invariants. The difficult part of any system specification is to define the invariants, i.e., conditions that are always true in every possible state. Nevertheless, defining strong invariants is essential as it ensures that key components of the system work as expected.

Depending on the t-CRDT different invariants should be verified. Usually the invariants check certain properties that are associated to the t-CRDT data type and, as such, must hold true for every process. Generally, besides the convergence invariant (which we previously defined), we want to define and verify at least the following types of invariants:

1. one or more invariants that check that after the replicas have converged, the observable state in all replicas is the same (i.e., queries return the same results).
2. one or more invariants that check the nonexistence of two conflicting operations, e.g., an *add* and a *remove* of the same element in a set, on the observable (active) state.
3. an invariant that checks if at least one operation is active, assuming that at least one was ever executed.

To exemplify, let's consider the set t-CRDT.

$$
\begin{array}{lcl}
\mathit{observableConvergence} & \triangleq & (\forall\ \mathrm{r} \in \mathrm{REPLICAS} :\ \mathrm{finished}[\mathrm{r}] = \mathrm{TRUE}) \implies \\
 & & \quad \forall\ \mathrm{r1} \in \mathrm{REPLICAS}, \mathrm{r2} \in \mathrm{REPLICAS} : \\
 & & \quad\quad \mathit{sameElements}(\mathrm{r1}, \mathrm{r2}) \\
 & & \quad\quad \land\ \mathit{sameLookup}(\mathrm{r1}, \mathrm{r2}) \\
\mathit{sameElements}(\mathrm{r1}, \mathrm{r2}) & \triangleq & \mathit{elements}(\mathrm{ops}[\mathrm{r1}]) = \mathit{elements}(\mathrm{ops}[\mathrm{r2}]) \\
\mathit{sameLookup}(\mathrm{r1}, \mathrm{r2}) & \triangleq & \forall\ \mathrm{e} \in\ \mathrm{ALL\_ELEMENTS} : \\
 & & \quad \mathit{lookup}(\mathrm{ops}[\mathrm{r1}], \mathrm{e}) = \mathit{lookup}(\mathrm{ops}[\mathrm{r2}], \mathrm{e})
\end{array}
$$

Figure 5.13: TLA+ invariant for observable convergence.

Figure 5.13 contains a TLA+ specification for the observable state invariant. Verifying if for every process finished[p] = TRUE implies that all replicas have already converged, as they only finish when they have all operations. Note that, as explained previously, when PlusCal is translated to TLA+, all variables that are local to a process are converted to a structure, where each key is a process identifier that indexes the local variable of that process. This only affects the code that is written in TLA+, bearing no effect in PlusCal code.

In the case of the set t-CRDT, there's only one situation on which we have incompatible operations: a concurrent *remove* and an *add* for the same element. The invariant in Figure 5.14 verifies that in no situation those two operations for the same element are simultaneously active.

A third relevant invariant to check is that, for each element, at least one operation is active in any state, assuming that one operation was already executed for it. Figure 5.15 contains a possible TLA+ specification of this invariant.

Albeit the concept of this invariant is simple, specifying it isn't. First, we prepare two

```
noSimultaneousAddRemove ≜ ∀ r ∈ REPLICAS :
                            LET active ≜ getActive(ops[r])
                            IN         ∄ op1 ∈ active, op2 ∈ active :
                                          op1.e = op2.e
                                          ∧ op1.type = ADD
                                          ∧ op2.type = REMOVE
```

Figure 5.14: TLA+ invariant for checking if in no situation two conflicting operations (*add* and *remove* for the same element) are active simultaneously.

```
elementsInObs(r) ≜ {e ∈ ALL_ELEMENTS : ∃ op ∈ getConcurrentObs(ops[r]) : op.e = e}
atLeastOneOp     ≜ ∀ r ∈ REPLICAS :
                     LET active      ≜ getActive(ops[r])
                         obsElements ≜ elementsInObs(r)
                     IN              ∀ e ∈ obsElements : ∃ op ∈ active : op.e = e
```

Figure 5.15: TLA+ invariant for checking if, for every element that ever was in the set, there's a least one operation that isn't obsoleted.

auxiliary variables which hold, respectively, the set of active operations (*active*) and the set of elements that have at least one operation that refers to it as obsoleted (*obsElements*). Then we check that for each element that is in *obsElements*, at least one operation referring to that same element isn't obsolete. Note that if an operation was executed for an element and that element isn't on *obsElements*, then it means that operation isn't obsoleted, which respects the invariant. As in other invariants, this is verified for each process.

Depending on the t-CRDT other types of invariants can be specified, but the ones explained here are generally enough to ensure most common assumptions that are done when specifying policies.

6: Correctness invariants As previously mentioned, since the three correctness properties are related to data type specific behavior, then to make sure that a given t-CRDT behaves correctly we need to specify invariants that are sufficient conditions for each property. For compactness, we'll only include here the invariant for PSE in the set t-CRDT, while the remaining ones for the set t-CRDT can be found in Appendix E.

To ensure that the set t-CRDT respects the PSE principle, the PSE invariant must check the following three conditions, which together are sufficient for PSE:

1. any *add* or *remove* on a given element *e* has no effect on other elements, i.e., queries return the same result except for possibly *e*;
2. an *add* or *remove* on a given element *e* cancels all *adds* and *removes* on element *e* that happened-before;
3. if the latest operation according to happens-before was an *add* (resp. *remove*) on element *e*, and assuming there are no operations concurrent to the latest one for element *e*, then *lookup*(*e*) returns true (resp. false).

```
noOtherElementsAffected(op, prevState, newState) ≜
    elements(prevState) \ {op.e} = elements(newState) \ {op.e}
    ∧ {otherOp ∈ prevState : otherOp.e ≠ op.e} = {otherOp ∈ newState : otherOp.e ≠ op.e}
verifyPSE(op, prevHBState, newHBState, prevState, newState) ≜
(1)  noOtherElementsAffected(prevState, newState)
(2)  ∧ ((op ∉ newHBState ∧ ∃ otherOp ∈ prevState : otherOp.e = op.e
(2)      ∧ happenedBefore(otherOp, op) ∧ lookup(prevState, op.e) = lookup(newState, op.e))
(2)    ∨ (op ∈ newHBState ∧ (∄ otherOp ∈ newHBState : otherOp.e = op.e
(2)      ∧ (happenedBefore(otherOp, op) ∨ happenedBefore(op, otherOp)))
(3)      ∧ (op ∉ newState
(3)        ∨ (op.type = REMOVE ∧ ¬lookup(newState, op.e))
(3)        ∨ (op.type = ADD ∧ lookup(newState, op.e)))))
```

Figure 5.16: TLA+ invariant for PSE property. The numbers between parenthesis match with the conditions each line is verifying.

Figure 5.16 contains the TLA+ specification of the PSE invariant. The arguments *prevHBState* and *newHBState* correspond to the states obtained by removing the operations canceled by happens-before (*getHB*) from the set of operations that we had, respectively, before and after adding *op*. As for *prevState* and *newState*, they correspond to the results of executing *getActive*. The idea here is to verify that, for every possible state transition, the PSE invariant holds true.

NoOtherElementsAffected verifies that applying an operation only affects the state concerning the element referred by the operation (*op.e*), leaving the rest of the state unaffected. This ensures the first condition required for PSE and is, in fact, also necessary for PPE and PCS. As for the lines marked with (2), they verify that there can't be two operations for the same element present in the state unless they are concurrent and that the ones in the state are the latest according to happens-before, thus ensuring the second condition. We also verify that adding an operation for an element when there is already a more recent one in the state has no effect in the visible state, i.e., *lookup*(*op.e*) returns the same result. Finally, for the third condition, we verify that one of these situations must be true: (i) *op* isn't in the active state due to a concurrent operation and thus, PSE doesn't apply; (ii) *op* is in the active state and is a remove, thus *lookup*(*op.e*) returns false; (iii) *op* is in the active state and is an add, thus *lookup*(*op.e*) returns true. As long as at least one of these is true, the third condition is ensured. Since *verifyPSE* verifies the three sufficient conditions for PSE, then if this invariant holds true for every possible state we ensure that our set t-CRDT respects PSE.

7: Models. Specifying a model can be done in two steps: (i) define a value for each constant and (ii) specify which invariants should be checked.

The TLA+ toolbox [16] contains a graphical interface that helps on creating a model to be ran on TLC. Figure 5.17 includes a possible model for the set t-CRDT. For the constants that represent a name (e.g., ADD), we can attribute a model value. A model value is a value that is different from any other value (i.e., an identifier), which is what we

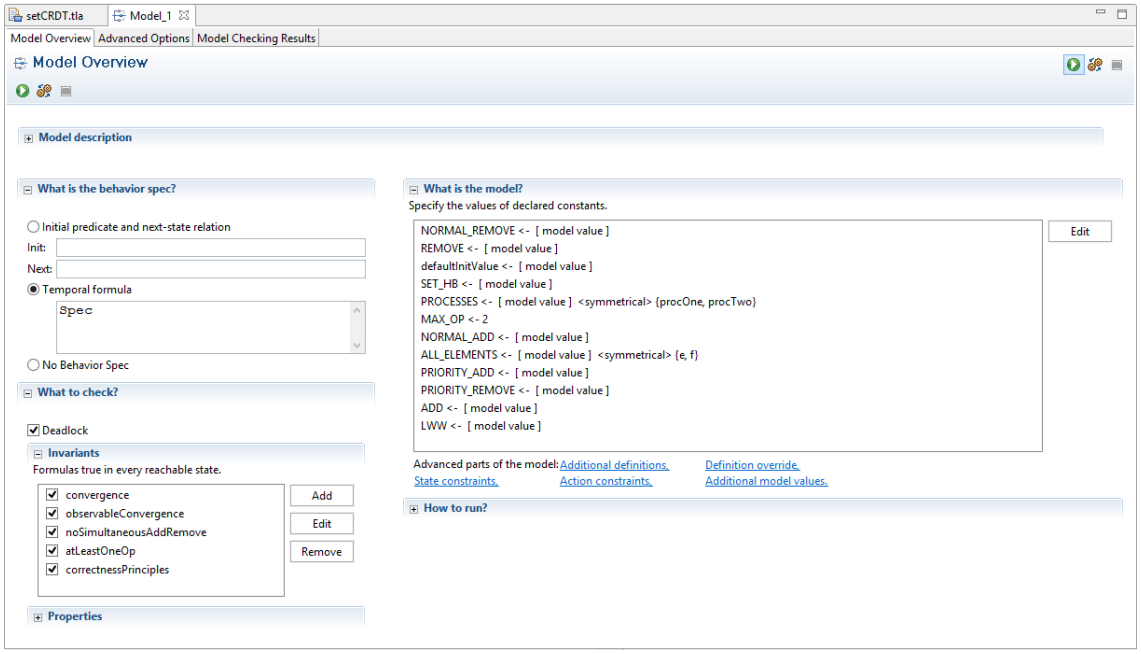

Figure 5.17: Example of a model definition in TLA+ toolbox. Note: *correctnessPrinciples* corresponds to the invariant discussed in Appendix E which checks PSE, PPE and PCS.

number of different elements	Number of clients and ops per client		
	2ops/2replicas	3ops/2replicas	2ops/3replicas
1 element	00:00:04h	00:03:32h	00:58:26h
2 elements	00:00:12h	02:43:00h	02:10:00:00d
3 elements	00:00:25h	15:19:00h	N.A.

Table 5.1: Execution times of different models for set t-CRDT.

need for constants that identify operations/policies. Usually the MAX constant should be at least 2 and REPLICAS should have at least 2 members, in order to test the data type in a distributed system and test all concurrency and happens-before scenarios.

For completeness, we show the execution time of multiple models in TLC for the set t-CRDT in Table 5.1. All of those models were executed on a single node with 2 Intel Xeon E5-2620 v2 CPUs and 64 GB of RAM.

For all of the tested models except one, TLC was able to generate every possible state and successfully verify that all invariants hold true in every state. Unfortunately, for the model with 3 replicas, 2 ops per replica and 3 different elements, TLC wasn't able to generate every possible state as the node ran out of available disk space. This model generated approximately 9 billion states and used around 750GB. We predict that more than 1.5TB of disk space would be needed to fully run this model. We note that even in this model all invariants did hold true for every generated state.

Finally, we highlight that without TLA+ and TLC (or equivalent languages and tools) verifying the correctness of t-CRDTs and of the generic model would be much more difficult. In fact, TLA+ played a crucial role in detecting situations in which early iterations of both the generic model and certain policies didn't behave as expected.

Chapter 6

Evaluation

In this chapter we provide a performance comparison of a t-CRDT with equivalent op-CRDTs. To this end we implemented a distributed system in which multiple clients can execute operations on their local CRDT replicas and send operations they executed to each other. The system is also able to collect multiple statistics as operations are being executed, which gives us the necessary information to evaluate CRDT's performance.

Despite having all of the t-CRDTs described in Section 4.1 implemented in our system[1], in the remaining of this chapter we'll only compare the performance of set CRDTs. The following set op-CRDTs have been implemented and will be compared with the set t-CRDT: (i) OR-Set [27], (ii) optimized OR-Set [6], (iii) R&AW-Set [24], (iv) optimized R&AW-Set [24]. The first two provide an *add-wins* policy, while the other two provide both *add-wins* and *rem-wins* simultaneously. Both the optimized and non-optimized R&AW-Set CRDTs result from a different approach we took previously in order to provide multiple policies in one CRDT [24].

The main goal of our evaluation is to compare how our set t-CRDT performs in multiple scenarios when compared to state-of-the-art set op-CRDTs in terms of: (i) operations per second; (ii) state size and (iii) average message size.

6.1 Implementation details

In this section we discuss multiple implementation details that we consider necessary to understand the practical evaluation's results.

Implementation language. The previously mentioned system was implemented in Scala and Java. In detail, the t-CRDT library in Section 4.1 was fully implemented in Java while

[1] Our Java implementation of the complete t-CRDT library is available at: https://github.com/AndreRijo/T-CRDTs

everything else (set op-CRDTs, clients, tests, etc.) was implemented in Scala, along with the use of Akka's library to help implement the clients and their communication. Note that Scala compiles to Java bytecode and is completely compatible with Java.

Communication and op-CRDTs implementation. Our communication system can group multiple operations in one message, thus avoiding the overhead of sending operations individually. A client sends a message to every other client after reaching a target number of new operations, which is configurable. Each message contains the update operations generated since the last message that was sent (i.e., every operation is sent only once to each client unless a message is lost). Each client is represented by an Akka actor and, as such, clients always communicate by messages independently of being all executed in the same node or in a distributed environment.

For each message our system provides at least once delivery guarantees, as required by t-CRDTs. We guarantee this by associating to each message a message ID and then having the receiving client send an ACK with the message ID to the sender to confirm the reception. If the sender doesn't receive the ACK after a certain (configurable) amount of time, it resends the message to that client.

Since the original specifications of op-based OR-Set and R&AW-Set require casual delivery, we had to adapt our implementation of them in order to work under at least once delivery, which has implications on their performance. In terms of used space by each set op-CRDT our implementation uses the same space as a standard implementation (i.e., one which assumes causal delivery), but the execution time is possibly different due to the adaptations needed for the CRDT's algorithms.

Concurrency detection (history field implementation). The history field [2] in each t-CRDT operation is implemented with a vector clock [13]. As it is possible to obtain a total order from a vector clock plus an unique ID for each replica, the same vector clock that is used for history is also used as a logical clock for operations whose policy requires one (e.g: lww), with the replica ID being used to order concurrent operations.

Operations implementation. All CRDT operations have one byte for determining the type of operation (*add*, *remove*, *lookup* and *elements*) and, except for *elements*, the element the operation refers to (string, in which we assume each char uses 1 byte).

Some operations of the non-optimized versions of set op-CRDT need to carry one or more unique identifiers (8 bytes integer per unique). On the other hand the optimized versions carry instead in their operations a vector clock (4 bytes integer per client).

Operations of the set t-CRDT need to carry a vector clock for history, which uses a 4 bytes integer per client. We represent policies with 8 bytes function pointers. Since

[2] The history field (described in Section 3.2.1) is responsible for providing enough information in order to be able to compare the operation it is associated to with any other in order to decide which one happened-before or if they are concurrent.

each operation needs 3 policies (*hbPolicy*, *selfObsoletePolicy* and *otherObsoletePolicy*) then each operation carries 24 bytes for the policies. Using function pointers assumes that every replica knows all necessary policies and that no new policy is added during the life time of the t-CRDT instance, which should be the case for most real systems. Note that it would be possible for each operation to carry the code for the policy instead of pointers, which would allow to add a new policy at any time. That would, however, be less space efficient.

Generic model implementation. We implemented three different versions of the op-based generic model described in Algorithm 3.1. The first version corresponds almost directly to the specification, while the other two are optimized models for data types such as sets, maps and arrays. The op-based specification for both optimized models can be found in Appendix C, while a description of the state-based versions can be found in Appendix D.

In all of the three versions queries are the bottleneck of t-CRDTs and as such each one tries to somewhat optimize them. Also in all three versions we provide the option to cache the active state when it is calculated – this can be useful for scenarios in which the frequency of query operations is considerably higher than update operations frequency. During our experimental evaluation we had, however, this option disabled.

First version: original. Our implementation of the first version is basically a translation of the specification to Java. It does have, however, a small optimization – in *calculateState*, when deciding if one operation should be obsoleted, we stop going through the set of all operations as soon as we find one that obsoletes it. Depending on the t-CRDT this might save some considerable time, and it is never slower than going through the whole set.

We note that this version, just like the specification in Algorithm 3.1, has a time complexity of $O(n^2)$ in both worst-case, expected and best-case scenario, where n is number of operations in the state. As such, it is expected for queries in large data sets using this model to take long to execute. The other two versions address this issue by effectively reducing the time complexity for the expected and best-case scenarios.

Second version: opt-t-CRDT. Our second implementation, which we call optimized t-CRDT (or opt-t-CRDT), takes advantage of a property that certain data types have in their common usage scenario. To exemplify this property, consider a set t-CRDT. If we analyze the policies defined for the set in Section 4.1.3, we observe that all of the policies never return true when comparing two operations with different elements, that is, two operations with different elements never obsolete each other. Even if we intend to define policies that are dependent on the element (e.g: add loses to remove if both have element a but add wins if both are b), usually those policies will still only affect operations for the same element. This same property can be observed in other data types, such as maps (keys) and arrays (positions).

The advantage of having this property is that we can effectively separate the state in multiple parts, with each part corresponding to one element/key/position. As such, our state becomes a map of parts to set of operations. This comes with two big advantages for the execution of queries, as it allows to reduce the time complexity in the expected and best-case scenario:

1. for queries which only consult one part, e.g: *lookup*(*e*) in a set, we can provide an *individualCalculateState* procedure, which is similar to *calculateState* of the first version but that only computes the active state for the part received as argument. For the *lookup*(*e*) example, *individualCalculateState* would only compute the active state for element *e*, with all operations concerning *e* stored in *e*'s entry in the map. This procedure has a smaller time complexity for the expected and best-case scenario – $O(m^2)$, where m is the number of operations in the part being calculated. For instance, considering a set with 20000 elements and 100000 uniformly distributed operations (5 operations per element), executing *individualCalculateState* costs $5^2 = 25$, instead of $100000^2 = 10^{10}$ as in the first version.
2. for queries which consult the whole state, e.g: *elements* in a set, *calculateState* is still optimized – for each operation we only compare with other operations of the same element, instead of all operations. The temporal complexity becomes $O(o * m^2)$, where o is the number of different elements and m the average number of operations in each part. To exemplify, consider again a set with 20000 elements and 100000 uniformly distributed operations. Executing *calculateState* here costs $20000 * 5^2 = 5 * 10^5$, which is quite a decrease compared to 10^{10}.

We note that, unfortunately, this version doesn't provide any improvement for the worst case scenario. This scenario is, however, very unlikely for instances with multiple elements, as it corresponds to the case in which *all* operations are for the same element.

Third version: inc-t-CRDT. Our final implementation of the generic model, which we call incremental t-CRDT (or inc-t-CRDT), is similar to the second version – it also has the state represented as a map of parts to set of operations and has both *individualCalculateState* and *calculateState*. It also only works for data types with the parts concept previously introduced.

The difference in this version is that, when an operation for a given part is added to the state (i.e. at the execution of *addOp*), the set of operations obsoleted by happens-before for that part is calculated right away and removed from the state. This has two advantages – first it prevents the state from growing too much even when queries are rarely executed, as in both the first and second versions the state is only "cleaned" when *calculateState* or *individualCalculateState* are executed. Secondly, it speeds up the execution of queries as those no longer need to calculate which operations are obsoleted by happens-before. This change also comes with a disadvantage – adding an operation now takes longer as more computation is done compared to the other versions which only add the operation

to the state. In terms of time complexity now *addOp* is O(m^2), while *calculateState* and *individualCalculateState* stay the same as in the second version.

In short, this version trades some time efficiency of update operations for faster queries and an average smaller state.

6.2 Tests characteristics

All tests were executed with three clients in a localhost environment, i.e., the three of them in the same node. This means that both network transfer time and latency weren't considered, but the time required to create a message was. Each client generated and executed 1.5 millions of update operations locally in its CRDT replica (for a total of 4.5 millions of update operations), with the distribution of *adds* and *removes* depending on the test scenario. Each client also executed some *lookup* and *elements* queries, with the amount of each depending on the scenario. In all scenarios the elements to *add*, *remove* and *lookup* were randomly chosen from a set of 20000 different uniformly distributed strings. In all tests, the target message size was 37500 operations, which leads to 40 messages sent by each client and a total of 240 messages exchanged, assuming no loss. The retransmission timeout was configured to 2000ms, albeit it was never needed as this was executed in localhost environment. Whenever a replica received a message it applied the received operations as soon as possible, which implies that in practice each client applied 4.5 millions of update operations.

Since OR-Set offers *add-wins* semantics and R&AW-Set offers simultaneously *add-wins* and *remove-wins*, in all tests our set t-CRDT could use the policies that provide either of those semantics. Precisely, from the policies defined in Section 4.1.3, *add* always used *normalAdd* and for remove it was randomly chosen between *normalRem* and *priorityRem* with a 50% chance for each in order to get, respectively, *add-wins* and *remove-wins*.

In order to test effectively how our set t-CRDT performs compared to OR-Set and R&AW-Set op-CRDTs, we varied the following parameters across the tests:

- percentage of *adds* and *removes*: we tested both 50%-50% and 90%-10% of adds-removes. When choosing to execute a remove, both R&AW-Set and set t-CRDT have a 50%-50% chance to choose, respectively, add-wins or remove-wins semantics. The main goal is to evaluate if our set t-CRDT behaves differently depending on the type of operation as other set CRDTs do [24];
- percentage of *lookups* and *elements*: we tested executing 1500, 15000, 150000 and 750000 (resp. 0.1%, 1%, 10% and 50% of total update operations generated by the client) *lookups* in each client along with the 1.5 millions update operations. This means that, for example, with 10% lookups each client generated 1.5 million updates and 150000 *lookups*. We also tested not executing any *elements*, 0.01% and 0.1% *elements*. This allow us to test the overhead of executing *calculateState* and the benefit of having *individualCalculateState*, which should both be the performance

bottleneck of t-CRDTs.

The combination of different configurations shown above originate multiple scenarios, from which we'll discuss and present in the following section the scenarios that have, in our opinion, the most relevant results. In order to have precise results, each tested scenario was executed 13 times, with the first three always being discarded as warmup. The results present in the next section are thus the mean of the 3 clients across the 10 remaining executions, that is, each value presented is an average of 30 values. For example the execution time of a test is the average of the execution time of each client (3) in each execution (10).

Besides the mentioned scenarios we also test if varying which policies are offered by the t-CRDT has any significant impact on the performance. We expect that to have little to no impact in the t-CRDT's performance.

All tests were executed in a node which has 2 Intel Xeon E5-2620 v2 CPUs with 64 GB of RAM.

6.3 Experimental results

In this section we present the results obtained from executing some of the different scenarios explained in the previous section. These results are organized in multiple charts, with each one focusing on one metric (e.g: execution time). Except for the last two graphs, each graph includes the results for each set op-CRDTs previously mentioned and our set t-CRDT both in the optimized and incremental models. The last two graphs show the results of varying the policies offered by both set t-CRDTs and thus doesn't include op-CRDTs which always offer the same policies.

6.3.1 Impact of the ratio of add versus removes

Figures 6.1 and 6.2 contain the results of executing the different set CRDTs with, respectively 50%-50% and 90%-10% add-remove distribution, with both having 0.1% *lookup* queries. The graphs on the left and right represent, respectively, the growth of execution time and metadata size as operations were being executed.

Analysis of 50%-50% add-remove

In terms of execution time the optimized set t-CRDT executes faster than all set op-CRDTs, while being very close to the optimized OR-Set. Since very few queries are executed, this means that the opt-t-CRDT rarely calculates the set of active operations for a given element and, thus, most of the computation is just storing the generated/received operation, which is fast. On the other hand some of the op-CRDTs take longer as the non-optimized versions have to calculate the set of unique identifiers to remove when executing a *remove* and, for the R&AW-Sets, they need to access more than one data structure for some operations. The incremental set t-CRDT is around 50% slower than

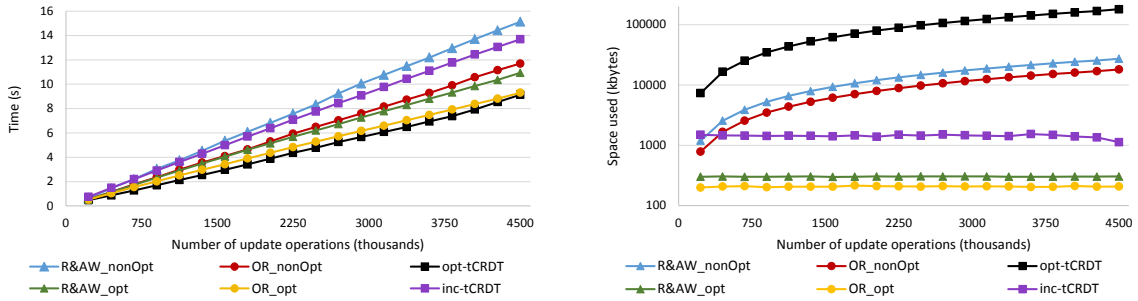

Figure 6.1: Average time and size of metadata (y-axis) of each set CRDT in the test with 50%-50% add-remove distribution and 0.1% *lookups*. Note that the x-axis represents the total amount of update operations generated by all clients instead of just one client, as each client needs to execute all updates operations. The y-axis in the metadata graphic is in binary logarithmic scale.

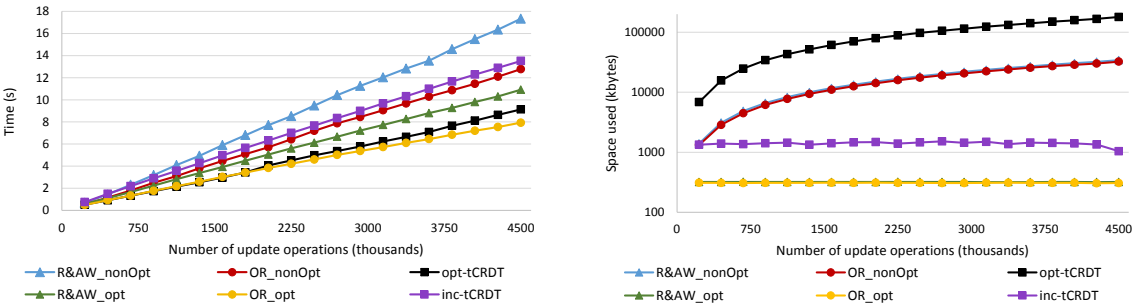

Figure 6.2: Average time and size of metadata of each set CRDT in the test with 90%-10% add-remove distribution and 0.1% *lookups*.

the optimized set t-CRDT due to the extra cost of calculating the operations obsoleted by happens-before for a given element every time an operation is added. It is still faster than the non-optimized R&AW-Set though.

As for the metadata size, as expected the op-based optimized versions are quite efficient – they only need to store a vector clock for each added element (R&AW-Set also needs to store a vector clock for each element removed with remove-wins policy), which means the data size is relatively constant as the amount of elements is capped at 20000. The size of non-optimized versions grows as time goes on, since they keep generating and storing new unique identifiers. As for the t-CRDTs, unfortunately the optimized one uses a lot of space – approximately 6.7 and 623 times more than, respectively, the non-optimized and optimized R&AW-Set. On the other hand, the incremental version is more space efficient – it constantly uses less than 1.5MBytes, being around 19 times better but 4.65 times worse than, respectively, the non-optimized and optimized R&AW-Set.

For both t-CRDTs the extra space is due to having to store all information of each operation instead of just the element and/or unique identifier/clock. The optimized t-CRDT uses even more space due to rarely removing the operations obsoleted by happens-before from the state, as *lookup*(*e*) is rarely executed and even then only the non-relevant operations relative to *e* are removed.

As a side note, the reason for the drop in size by the end of execution for the incremental version is due to the fact that not all clients end at exactly the same time, which implies that one client will execute some operations after all others, thus cleaning even more operations by happens-before. This was observed in every scenario we tested.

Analysis of 90%-10% add-remove
Our set t-CRDTs take approximately the same execution time and used space for both distributions, which leads us to conclude that, as expected, the type of operation doesn't have any effect for set t-CRDTs. For set op-CRDTs the opposite is true, as the difference between 90%-10% and 50%-50% is not negligible – all use more space (albeit the difference between R&AW-Set and OR-Set is very small in this case), the non-optimized are slower but the optimized OR-Set is faster. This is due to the fact that in common set op-CRDTs, unlike in our set t-CRDT, different computations and data structures are accessed for different operations, which originates different execution times and storage overhead depending on the operation distribution.

Conclusion
Unlike in the evaluated set op-CRDTs, in our set t-CRDTs both execution time and used space isn't affected by the distribution of update operations, as both do the same actions. Our optimized set t-CRDT execution time is close to the optimized OR-Set's and faster than any of the other analyzed CRDTs. As for our incremental set t-CRDT, it trades some time efficiency (while still being faster than non-optimized R&AW-Set) for space efficiency. In the two analyzed scenarios our optimized set t-CRDT has a considerable space overhead compared to set op-CRDTs, mainly due to two reasons: (i) necessity of storing all operation data instead of using data type specific structures; (ii) rare execution of queries (*lookup*) implies that the optimized set t-CRDT's state is rarely clean and, when it is, it's only cleaned for the element referred by *lookup*.

6.3.2 Impact of partial state queries - lookup

Figures 6.3 and 6.4 contain the results of executing the different set CRDTs with, respectively, 10% and 50% *lookups*, with both cases having an add-remove distribution of 50%-50%. The percentage of *lookups* represents how many queries are executed relative to the amount of update operations.

Analysis of 10% lookups
Compared to the results observed before for 0.1% *lookup* with 50%-50% add-remove it's noticeable that our optimized set t-CRDTs executes on average 30% slower. This slowdown is due to the complexity of queries, which have to calculate the active state for the element referred by *lookup*. As such, in this scenario the execution time for it is closer to the optimized R&AW-Set and non-optimized OR-Set. All of the set op-CRDTs and our

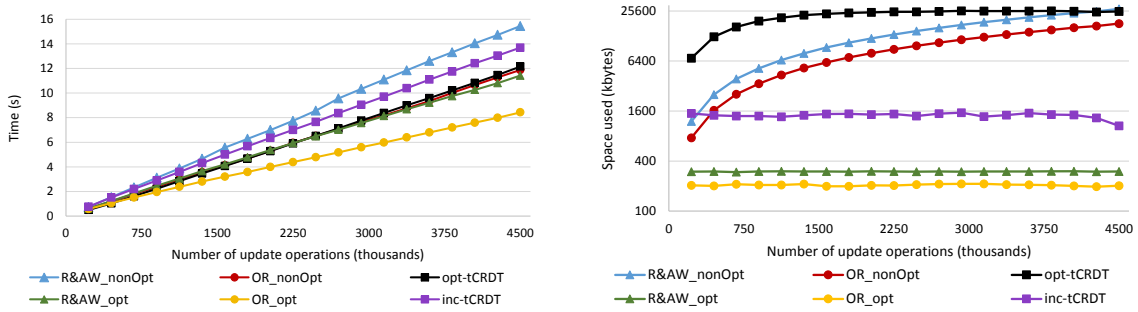

Figure 6.3: Average time and size of metadata (y-axis) of each set CRDT in the test with 10% *lookups* and 50%-50% add-remove distribution.

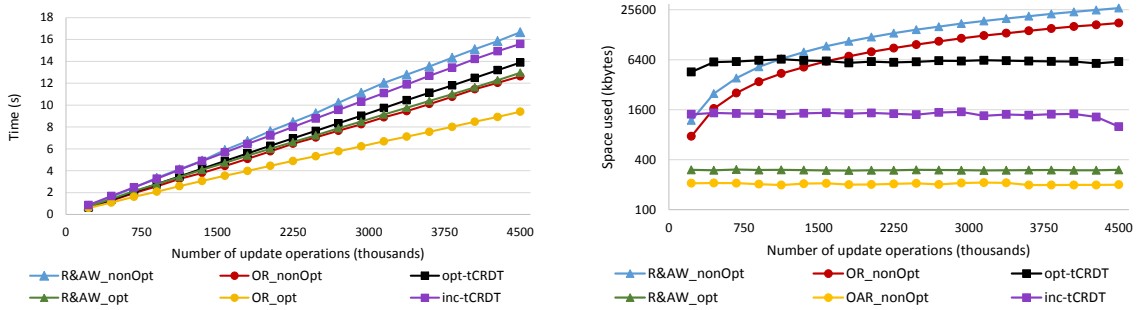

Figure 6.4: Average time and size of metadata (y-axis) of each set CRDT in the test with 50% *lookups* and 50%-50% add-remove distribution.

incremental set t-CRDT execute at approximately the same speed when comparing 0.1% with 10% *lookup*.

In terms of space, once again the results for set op-CRDTs are similar between 0.1% and 10% *lookup*. On the other hand, our optimized set t-CRDT uses considerably less space for 10% *lookup* – in fact, the space used seems to cap at approximately 25 MBytes, which is around the same as non-optimized R&AW-Set has by the end. This is because in optimized and non-optimized t-CRDTs executing queries more frequently implies that the state is also cleaned more frequently, thus getting rid of more operations that are no longer relevant. On the other hand for our incremental set t-CRDT the used space is independent of the queries frequency, as irrelevant operations are removed when adding more recent operations.

Analysis of 50% lookups

Considering the previously analyzed graphs and the one on Figure 6.4, a trend can be seen relative to the optimized set t-CRDT – as we increase the frequency of queries, the execution time takes longer but the size of data caps earlier. The used space for the incremental set t-CRDT remains unchanged, but the execution time increased.

For the 50% *lookup* case, the total execution time for the optimized t-CRDT is around 14s, which is a bit more than the optimized R&AW-Set but considerably less than the non-optimized R&AW-Set. As for the metadata size, initially it is bigger than any of the

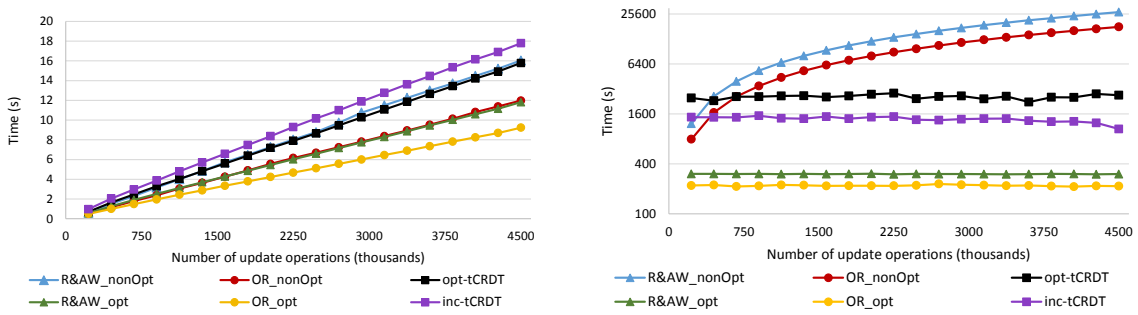

Figure 6.5: Average time and size of metadata (y-axis) of each set CRDT in the test with 0.01% *elements*, 0% *lookups* and 50%-50% add-remove distribution.

other set CRDTs but rapidly it reaches a cap of approximately 6MBytes and when the total update operations executed reaches the 1.5 millions mark the state starts becoming smaller than both non-optimized versions. By the end of the test the metadata size of the optimized set t-CRDT is around 3 and 4.5 times smaller than the non-optimized versions of, respectively, OR and R&AW sets. It is still bigger than the state of optimized versions or of the incremental set t-CRDT though.

Conclusion

In the optimized t-CRDTs, as we increase the frequency of query operations (even the ones that only consult part of the state), the size of the metadata decreases, reaching a cap earlier. This cap depends both on the max number of different elements that the set can have and the frequency of queries, as obviously if the max number of elements is not limited the size of the state can grow infinitely, both in the optimized set t-CRDT and all analyzed set op-CRDTs. The decrease of state size can be quite significant: from 0.1% to 50% *lookups* it decreased by about 2970%! On the opposite side, the state size of the incremental set t-CRDT is independent of queries frequency.

Unfortunately as we increase the frequency of queries the execution time also increases: from 0.1% to 50% *lookup* it increased by around 52% and 13.7% for respectively the optimized and incremental versions. Nevertheless the increase in execution time is quite smaller than the savings in space for the optimized version. Both versions are still faster than the non-optimized R&AW-Set, while providing at least the same semantics and with the possibility for more rich, user-defined policies.

6.3.3 Impact of full state queries - elements

The *elements* query requires for the complete active state to be computed for every element. As such, it is expected for this operation to slow down considerably the execution of the test for both set t-CRDTs. Taking this in consideration we tested low percentages of *elements* execution – 0.01% and 0.1%, which are shown respectively in Figures 6.5 and 6.6. For both cases we show the results for the two set t-CRDTs and the four set op-CRDTs, with a 50%-50% adds-removes distribution and 0% *lookups*.

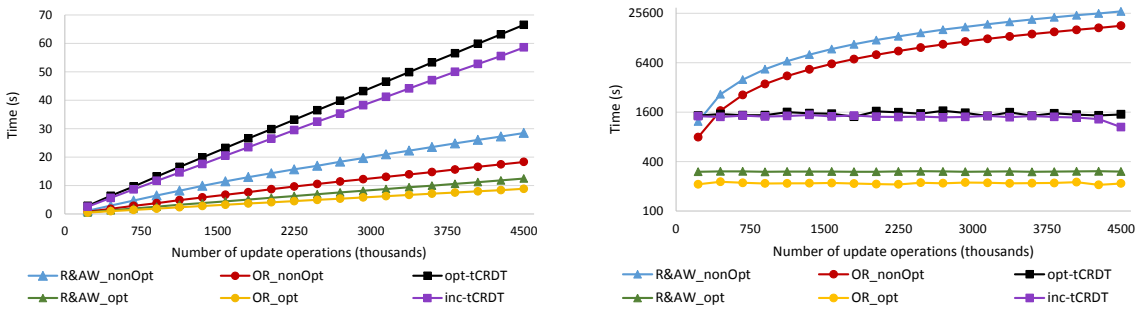

Figure 6.6: Average time and size of metadata (y-axis) of each set CRDT in the test with 0.1% *elements*, 0% *lookups* and 50%-50% add-remove distribution.

Analysis of 0.01% elements

The *elements* query demonstrate that, as expected, queries are the bottleneck of t-CRDTs. Executing only 150 (0.01%) *elements* queries per client as in Figure 6.5 during the execution of 4.5 million update operations is enough to increase considerably the execution time. For the optimized set t-CRDT it almost doubles when compared to the scenario with no *elements* and 0.1% *lookups*, being just barely faster than non-optimized R&AW-Set. Fortunately for the incremental t-CRDT the difference is smaller, but it is now slower than non-optimized R&AW-Set. The reason for *elements* slowing down more than *lookup* for set t-CRDTs is that *elements* requires computing the complete active state – in this case, the algorithm is applied to 20000 elements instead of 1, which justifies the slowdown. Op-based set CRDTs don't need to execute this complex algorithm, thus they are barely slowed.

As for metadata size, since *elements* implies cleaning the whole state, the optimized set t-CRDT uses less space than in previously analyzed scenarios, capping at 2.7MBytes.

Analysis of 0.1% elements

For the 0.1% *elements* scenario the execution time for both optimized and improved t-CRDT is considerably longer than for any of the analyzed set op-CRDTs. This reinforces the idea that calculating the whole active state is a complex operation, especially for wide datasets (in this case, 20000 elements). This scenario is also the only one in which the incremental set t-CRDT is faster than the optimized, ending the test 8s earlier.

The *elements* query is executed often enough to keep the state size of the optimized set t-CRDT as low as the incremental one – on average they have 1.6 (resp. 1.45) MBytes.

Conclusion

Queries which require calculating the whole active state are slow in t-CRDTs, even when executed with low frequency. They are, however, an effective way of keeping the data size of the optimized set t-CRDT small – executing just one once in a while (or directly executing *calculateState*) when a replica has few operations being executed on the t-CRDT

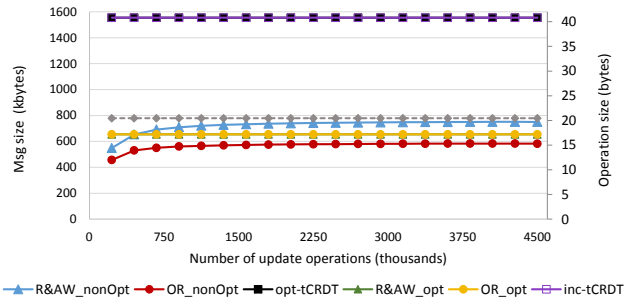

Figure 6.7: Average message and operation size (left and right y-axis) of each set CRDT in the test with 50%-50% add-remove distribution and 0.1% *lookups*. The dashed line corresponds to both set t-CRDTs' message and operation size after the optimization described in the text.

is enough to get rid of all irrelevant operations and thus bring the data size down considerably, without affecting too much the execution time.

6.3.4 Message size overhead

We now analyze the overhead in message size that the set t-CRDTs incur over set op-CRDTs. For this we include in Figure 6.7 the average message and operation size for the referred CRDTs, in the 50%-50% adds-removes and 0.1% *lookup* scenario.

As expected there is a considerable overhead in message size for both set t-CRDTs when compared to the others. This is due to the fact that each t-CRDT operation has to carry both a vector clock (12 bytes) for history information and 24 bytes in policies pointers, along with the element (5 bytes on average). If instead of function pointers we used an one byte identifier for each policy, we could had saved 21 bytes (a bit more than half of the operation size), which would lead to the result shown as a dashed line in Figure 6.7. Comparatively to others, the size of an operation would be less than half a byte bigger than the non-optimized R&AW set (the bigger one) and just 30% more than the non-optimized OR set (the smaller one). This space optimization could, however, have potentially negative effects on execution time, as it would require accessing an extra structure that maps identifiers to functions. It would also make the t-CRDT library less intuitive to use.

Regardless, we believe that for both set t-CRDTs and set op-CRDTs, the size of each operation is relatively small and doesn't increase much as the number of operations increases. In fact, for set t-CRDTs, it even stays constant. Finally we note that our communication system doesn't take advantage of the fact that operations deleted from the t-CRDT state don't need to be delivered, which could originate savings in transferred data.

6.3.5 Impact of varying the offered policies

Figure 6.8 contains the results of varying which policies are used by both set t-CRDTs for the scenario with an add-remove distribution of 50%-50% and 10% *lookups*. For the cases

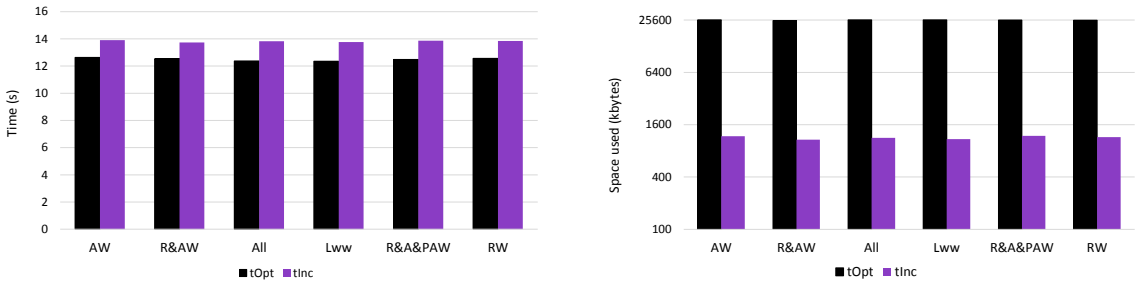

Figure 6.8: Average time and size of metadata (y-axis) for both set t-CRDTs by the end of each test for different policies in t-CRDTs. The add-remove distribution is 50%-50% with 10% *lookups*. The abbreviations AW, RW, PAW and LWW correspond to, respectively, add-wins, remove-wins, priorityAdd-wins and last-writer-wins.

in which more than one policy is available, the probability of choosing each one follows an uniform distribution.

As we previously conjured, varying which policies are offered by our t-CRDT doesn't seem to affect neither the execution time nor the metadata size considerably. This is due to the fact that all policies except *LWW* do similar computations and thus take approximately the same time to execute. *LWW* needs to compare the clocks, but that comparison is fast as the vector clocks only have 3 positions.

We believe that the slight variation (around 100ms on average) is not correlated to which policies are offered. To justify this, consider the R&AW and the "all" scenarios – for these scenarios, the optimized (respectively incremental) set t-CRDT is around 150ms faster (respectively around 100ms slower) for R&AW when compared to the "all", even though both versions execute the same algorithm for deciding which operations are obsoleted by concurrency. As such, we expect that as we increase the amount of executions of each test, the difference tends to zero.

Similar results are expected for user-defined policies, unless the user defines policies that execute considerably more computation than our policies. In that case, depending on the extra computation and the query frequency, the execution time might increase.

6.3.6 Overall evaluation

In general, both the optimized and incremental versions of set t-CRDT execute operations at a rate similar to other set op-CRDTs in the scenarios we analyzed that didn't use *elements* queries. The results show that in those situations the incremental version was always faster than non-optimized R&AW-Set, while the optimized set t-CRDT in some situations performed a bit slower than optimized R&AW-Set and in others its performance was close to the fastest set op-CRDT (optimized OR-Set), with one of scenarios even being the fastest. Both t-CRDTs perform slowly when executing *elements* over big data sets. Also varying which policies are offered by the t-CRDTs doesn't seem to have any considerable impact.

Message sizes in t-CRDTs are bigger than for op-CRDTs, due to the history and policies information that they have to carry. This also implies bigger state size, as for each stored operation the whole information needs to be kept, unlike in op-CRDTs in which data-type specific structures reduce the necessary data amount. For the incremental t-CRDT the state's size is independent of update and query distribution rates, depending only on how much concurrency and different elements there is. In the analyzed scenarios, on average its data size is 5 times bigger than optimized R&AW-Set but up to 20 times less than non-optimized R&AW-Set. As for the optimized t-CRDT, if queries are rarely executed the size can grow bigger than non-optimized set op-CRDTs, but we show that 10% *lookups* is enough to have a smaller size than non-optimized R&AW-Set by the end of the test. Also both of our t-CRDT versions eventually cap their size, unlike non-optimized op-CRDTs.

In comparative terms between the optimized and incremental t-CRDT versions, the choice resides in which is more important – time or space efficiency. If sacrificing between 15% and 50% time efficiency justifies having between 4 and 100 times smaller state, then incremental is preferred. If time is more important, then optimized should be used. Note that it's possible in the optimized t-CRDT to execute periodically a call to *calculateState* (or to *elements* query) to clean up the state without affecting too much the performance.

A concerned reader may wonder if the fact that t-CRDTs were implemented in Java while op-CRDTs were implemented in Scala may have an effect on the results. We unfortunately didn't had time to also implement the t-CRDTs in Scala to confirm this, which is a limitation of our experimental evaluation. We, however, believe that the difference should be negligible, for the following reasons:

- Both Scala and Java compile to Java bytecode, thus the instruction set is the same. Also, they both run on the same JVM;
- Scala is 100% compatible with Java, namely Java structures can be directly accessed in Scala code without needing any conversion. As such, mixing Scala and Java code should have no considerable overhead;
- For implementing both op-CRDTs and t-CRDTs we used HashSets, HashMaps and arrays. The first two are implemented with the same principle (hash table) in both Java and Scala and, thus, are expected to have similar efficiency. Scala arrays correspond directly to Java arrays, thus the performance should be the same. We do not, unfortunately, have any benchmarks that confirm these expectations though.

Chapter 7

Conclusion

CRDTs are used widely in distributed systems for which performance and fault tolerance are top-tier priorities. Some examples of such systems include Legion [22], AntidoteDB [10], Riak KV [5] and Akka Distributed Data [1], among others. Their usage is justified not only by their performance and low network requirements, but also because they solve any concurrency conflict automatically without requiring external intervention by applying a pre-defined policy. State-of-the-art CRDTs only provide one policy per CRDT instance, which complicates their adaptation to applications that require different conflict resolution policies than the one provided by default.

In this thesis we proposed a solution which gives the programmer total control on how conflicts are solved. For this we introduced the design of a new base CRDT model, the generic CRDT model. This model allows t-CRDTs that can represent data types such as sets and counters, among others, to be defined on top of it, while maintaining key properties of state-of-the-art CRDTs and letting the programmer specify for each operation the policy for solving conflicts. The model is responsible for storing operations, applying their policies and deciding which operations are relevant for the state.

We also presented the specification of a t-CRDT library for common data types (counters, registers, sets and maps). For each data type we provide multiple different policies that the programmer can choose to use, which we believe are adequate to most applications. Nevertheless the programmer can define his own policies and t-CRDTs with the help of the discussed methodology. We also implemented our t-CRDT library in Java, making it open source and available at `https://github.com/AndreRijo/T-CRDTs`.

We proved that our generic CRDT model has lower network requirements than most op-based CRDTs, as at least once delivery of each operation is enough for t-CRDTs. We have also shown that if policies are deterministic replicas eventually converge to the same state. Also for each t-CRDT and its policies in our library we wrote TLA+ specifications

and ran them on TLC, successfully verifying that key properties of each t-CRDT are hold true in every possible scenario.

Finally we evaluated the performance of two set t-CRDTs, each one using a different generic model implementation as basis. We compared both with state-of-the-art set op-CRDTs and verified that they can execute operations with comparable speed. The generality of our model and more powerful concurrency control given to the programmer implies extra storage overhead and slower execution of certain (but not all) queries, but nevertheless one of our models implementation was able to consistently have much lower metadata size than non-optimized set op-CRDTs.

7.1 Publications

In parallel with the development of this thesis we explored a different approach for the problem of providing multiple concurrency policies in one CRDT. This approach consists in specifying non-generic CRDTs that have some operations duplicated in order to provide multiple policies. We have published a description of this approach, along with a set state-CRDT and op-CRDT which provides simultaneously *add-wins* and *remove-wins* by providing two different *remove* operations. This paper can be found at [24].

7.2 Future work

Albeit we have showed that our set t-CRDT has performance comparable to set op-CRDTs, there is still room for improvement, namely in terms of used space. An interesting feature that could be explored would be the ability to detect operations that weren't yet replaced by more recent operations but will never be relevant again due to other concurrent operations. If such thing is possible, more operations could be deleted permanently from the state, thus reducing the average storage overhead.

Another approach would be to investigate a different way to apply the user-defined policies, without requiring an algorithm that has a temporal complexity of $O(n^2)$ on the number of operations when executing queries. We have already introduced an efficient optimization that can be used for data types such as maps, sets and arrays, but a different optimization or algorithm that works for all data types is still required. Specifying a different algorithm for this is difficult though, as the model must ensure convergence for any data type that the user specifies on top of it as long as all policies are deterministic, while ensuring the policies' intention is kept as much as possible. Assuming stronger delivery guarantees such as causal delivery may help in specifying a more efficient algorithm.

While we provide a Java library with t-CRDTs that are ready to use, it would however be interesting to expand this library for different data types, such as arrays, lists, graphs, JSON and others. Incorporating the library in existing databases such as Riak or AntidoteDB would also be relevant.

Bibliography

[1] Akka. *Akka Documentation - Distributed Data*. URL: https://doc.akka.io/docs/akka/2.5/distributed-data.html (visited on 09/14/2018).

[2] P. S. Almeida, A. Shoker, and C. Baquero. "Efficient state-based crdts by delta-mutation." In: *International Conference on Networked Systems*. Springer. 2015, pp. 62–76.

[3] C. Baquero, P. S. Almeida, and A. Shoker. "Making operation-based CRDTs operation-based." In: *IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer. 2014, pp. 126–140.

[4] C. Baquero, P. S. Almeida, and A. Shoker. "Pure Operation-Based Replicated Data Types." In: *arXiv preprint arXiv:1710.04469* (2017).

[5] Basho. *Riak KV: Riak Distributed Data Types*. URL: http://basho.com/products/riak-kv/riak-distributed-data-types/ (visited on 01/24/2018).

[6] A. Bieniusa, M. Zawirski, N. Preguiça, M. Shapiro, C. Baquero, V. Balegas, and S. Duarte. "An optimized conflict-free replicated set." In: *arXiv preprint arXiv:1210.3368* (2012).

[7] S. Burckhardt, A. Gotsman, and H. Yang. "Understanding eventual consistency." In: *Microsoft Research Report MSR-TR-2013-39* (2013).

[8] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. "Replicated data types: specification, verification, optimality." In: *ACM SIGPLAN Notices*. Vol. 49. 1. ACM. 2014, pp. 271–284.

[9] G. Cabrita and N. Preguiça. "Non-uniform Replication." In: *arXiv preprint arXiv:1711.07733* (2017).

[10] T. S. Consortium. *AntidoteDB: A planet-scale, available, transactional database with strong semantics*. URL: http://syncfree.github.io/antidote/ (visited on 01/24/2018).

[11] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. "Dynamo: amazon's highly available key-value store." In: *ACM SIGOPS operating systems review* 41.6 (2007), pp. 205–220.

[12] A. Deftu and J. Griebsch. "A scalable conflict-free replicated set data type." In: *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*. IEEE. 2013, pp. 186–195.

[13] C. J. Fidge. "Timestamps in message-passing systems that preserve the partial ordering." In: *Australian Computer Science Conference*. Australian National University. Department of Computer Science, 1988.

[14] S. Gilbert and N. Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services." In: *Acm Sigact News* 33.2 (2002), pp. 51–59.

[15] M. Kleppmann and A. Beresford. "A Conflict-Free Replicated JSON Datatype." In: *IEEE Transactions on Parallel and Distributed Systems* (2017).

[16] L. Lamport. *The TLA toolbox*. URL: https://lamport.azurewebsites.net/tla/toolbox.html (visited on 08/17/2018).

[17] L. Lamport. "Time, clocks, and the ordering of events in a distributed system." In: *Communications of the ACM* 21.7 (1978), pp. 558–565.

[18] L. Lamport. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[19] L. Lamport. "The PlusCal algorithm language." In: *International Colloquium on Theoretical Aspects of Computing*. Springer. 2009, pp. 36–60.

[20] L. Lamport. *A PlusCAL User's Manual–P-Syntax Version 1.8*. Tech. rep. 2018.

[21] L. Lamport et al. "Paxos made simple." In: *ACM Sigact News* 32.4 (2001), pp. 18–25.

[22] A. van der Linde, P. Fouto, J. Leitão, N. Preguiça, S. Castiñeira, and A. Bieniusa. "Legion: Enriching Internet Services with Peer-to-Peer Interactions." In: *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee. 2017, pp. 283–292.

[23] N. Preguica, J. M. Marques, M. Shapiro, and M. Letia. "A commutative replicated data type for cooperative editing." In: *Distributed Computing Systems, 2009. ICDCS'09. 29th IEEE International Conference on*. IEEE. 2009, pp. 395–403.

[24] A. Rijo, C. Ferreira, and N. Preguiça. "Set CRDT com Múltiplas Políticas de Resolução de Conflitos." In: *INForum 2018 - Actas do 10º Simpósio de Informática*. Universidade de Coimbra, 2018.

[25] H.-G. Roh, M. Jeon, J.-S. Kim, and J. Lee. "Replicated abstract data types: Building blocks for collaborative applications." In: *Journal of Parallel and Distributed Computing* 71.3 (2011), pp. 354–368.

[26] M. Shapiro and N. Preguiça. "Designing a commutative replicated data type." In: *arXiv preprint arXiv:0710.1784* (2007).

[27] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. "A comprehensive study of convergent and commutative replicated data types." Doctoral dissertation. Inria–Centre Paris-Rocquencourt; INRIA, 2011.

[28] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. "Conflict-free replicated data types." In: *Symposium on Self-Stabilizing Systems*. Springer. 2011, pp. 386–400.

[29] H. Wayne. *Learn TLA+ > Introduction*. URL: https://learntla.com/introduction/ (visited on 08/17/2018).

[30] S. Weiss, P. Urso, and P. Molli. "Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks." In: *Distributed Computing Systems, 2009. ICDCS'09. 29th IEEE International Conference on*. IEEE. 2009, pp. 404–412.

[31] S. Weiss, P. Urso, and P. Molli. "Logoot-undo: Distributed collaborative editing system on p2p networks." In: *IEEE Transactions on Parallel and Distributed Systems* 21.8 (2010), pp. 1162–1174.

[32] M. Wiesmann and A. Schiper. "Comparison of database replication techniques based on total order broadcast." In: *IEEE Transactions on Knowledge and Data Engineering* 17.4 (2005), pp. 551–566.

[33] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. "Understanding replication in databases and distributed systems." In: *Distributed Computing Systems, 2000. Proceedings. 20th International Conference on*. IEEE. 2000, pp. 464–474.

[34] Wikipedia. *Increlmental game*. URL: https://en.wikipedia.org/wiki/Incremental_game (visited on 09/06/2018).

[35] C. Wilk. *Clicker games - how parody became an addictive gaming genre*. URL: https://www.gamepressure.com/e.asp?ID=1730 (visited on 09/06/2018).

[36] Y. Yu, P. Manolios, and L. Lamport. "Model checking TLA+ specifications." In: *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. Springer. 1999, pp. 54–66.

Appendix A

Examples of CRDTs

A.1 Operation based LWW-Register

Algorithm A.1 op-based LWW-Register CRDT

```
1: payload X x, timestamp t                                   ▷ X: any object type
2:     initial ⊥, 0
3: query value () : X r
4:     let r = x
5: update assign (X v)
6:     atSource () : timestamp lt
7:         let lt = now()          ▷ now(): returns a timestamp consistent with causality
8:     downstream (v, lt)          ▷ No pre condition: any order for delivery is acceptable
9:         if t ≤ lt then x, t := v, lt
```

Algorithm A.1 represents the specification for the op-based LWW-Register discussed in Section 2.5.2. The specification is similar to the state-based one (see Section 2.5.2), with the payload and *value* operation being exactly equal. As for *assign*, the timestamp is generated in the *atSource* phase, otherwise each replica would generate different timestamps for the same *assign* and concurrent operations would be ordered differently, resulting in divergent states. On the *downstream* phase, the timestamp received is compared with the one in the payload, similarly to what is done on *merge* on the state-based version. This comparison is needed because of possible concurrent assigns and to cope with any delivery order.

A.2 Set

A.2.1 2P-Set

A possible solution for solving the conflicts of concurrent $add(e)$ and $remove(e)$ is to specify a set in which, after an element is removed, it can never be added again. Such a set is known as Two-Phase Set (2P-Set) [27].

Algorithm A.2 state-based 2P-Set CRDT

```
1:  payload set A, set R                    ▷ A: added elements; R: removed elements
2:      initial ∅, ∅
3:  query lookup (element e) : boolean b
4:      let b = (e ∈ A ∧ e ∉ R)
5:  query elements () : set E
6:      let E = {e | ∃e : e ∈ A ∧ e ∉ R}
7:  update add (element e)
8:      A := A ∪ {e}
9:  update remove (element e)
10:     pre lookup(e)                       ▷ Only removes the element if it exists
11:     R := R ∪ {e}
12: compare (X, Y) : boolean b
13:     let b = (X.A ⊆ Y.A ∧ X.R ⊆ Y.R)
14: merge (X, Y) : payload Z
15:     let Z.A = X.A ∪ Y.A
16:     let Z.R = X.R ∪ Y.R
```

The state-based specification of a 2P-Set CRDT is shown in Algorithm A.2. 2P-Set's payload is formed by two local sets: set A stores the elements that were added by some *add* operation; set R stores the elements that were removed by some *remove* operation. The *remove* operation is only executed if the element is in the set, as otherwise it would be possible to prevent an element from ever being able to be in the set. An element is considered to be in the 2P-Set if it is in A but not in R. Proof that the 2P-Set corresponds to a state-based CRDT can be found in [27].

With this strategy, all $add(e)$ (and also $remove(e)$) that are ordered after a $remove(e)$ have no effect and, as such, the 2P-Set represents a remove-wins set. Intuitively, what this set does is to order all $add(e)$ operations as happening before all $remove(e)$, which is enough to solve the conflicting situations. This ordering, however, doesn't respect causality, as even if an $add(e)$ is executed sequentially after a $remove(e)$ the element e won't be inserted into the set, unlike in a sequential set. This also implies that this CRDT isn't *sequentially conformant*, since it breaks PSE.

A.2.2 OR-Set

Algorithm A.3 contains the specification for the state-based version of the OR-Set discussed in Section 2.5.3.1. Proof that this state-based specification is a CRDT could be

Algorithm A.3 state-based OR-Set CRDT

```
1:  payload set A, set R          ▷ A: pairs (element e, unique-id u); R: removed unique ids
2:      initial ∅, ∅
3:  query lookup (element e) : boolean b
4:      let b = (∃u : (e, u) ∈ A ∧ u ∉ R)
5:  query elements () : set E
6:      let E = {e | ∃(e, u) : (e, u) ∈ A ∧ u ∉ R}
7:  update add (element e)
8:      let u = unique()                  ▷ unique(): generates an unique identifier
9:      A := A ∪ {(e, u)}
10: update remove (element e)
11:     pre lookup(e)                     ▷ Only removes the element if it exists
12:     let TR = {u | ∃u : (e, u) ∈ A}
13:     R := R ∪ TR
14: compare (X, Y) : boolean b
15:     let b = (X.A ⊆ Y.A ∧ X.R ⊆ Y.R)
16: merge (X, Y) : payload Z
17:     let Z.A = X.A ∪ Y.A
18:     let Z.R = X.R ∪ Y.R
```

obtained based on the proof for the state-based G-Set, which is present in [27].

The state-based version is based on a combination of the state-based 2P-Set and op-based OR-Set. As in 2P-Set, two sets are used: A, for *adds* and R, for *removes*. The idea of the *add* and *remove* are the same as in op-based OR-Set, however on *remove* the removed unique ids are kept in R. This is needed because *merge* should compute a LUB and no order for delivery of states is assumed, unlike in the op-based OR-Set which assumes that every *add* whose unique id is in a *remove* is delivered before that *remove*. Because the elements are never removed from A, the *lookup* operation needs to check both A and R.

Appendix B

Custom CRDTs

In the last few years multiple CRDTs for both basic and advanced data types have been proposed in the literature [2, 12, 15, 23, 27, 31]. However, most of these CRDTs offer very few operations, unlike data types offered by a programming language such as Java.

Another problem is that often CRDTs deal with concurrency conflicts in a way that may be undesirable for some applications. For example, an OR-Set gives precedence to an add when a concurrent add and remove happen for the same element (add-wins policy), that is, the element stays in the set. However, for some applications, it may be more adequate to have a remove-wins policy or, perhaps, support both policies and allow the programmer to choose the one more adequate for each situation, through the use of different operations.

Due to the mentioned problems, creating CRDTs that support more operations and offer more options for solving conflicts is thus desirable, in order to allow more distributed systems to more easily start using CRDTs. As such, in this chapter, multiple "Custom CRDTs" are defined. These CRDTs can be either an extension of an already existing CRDT but with more operations (for example, a counter CRDT which allows both adding or subtracting arbitrary values and also resetting the counter's value) or a CRDT with a different conflict solving policy, possibly even combining the policies of more than one CRDT (for example, a set CRDT with both add-wins and remove-wins policies).

B.1 Set

Set CRDTs deal with concurrent adds and removes on the same element through the use of very specific semantics or policies for solving concurrency conflicts. Some set CRDTs, such as the G-Set and 2P-Set, don't even respect the sequential semantics of a set. Others, such as the OR-Set, follow the expected sequential semantics, but have to give priority to

one of the operations when a concurrent add and remove happen for the same element.

In this section, three set CRDTs are presented. The first one is a remove-wins set CRDT which, unlike the 2P-Set, allows for an element to be re-added. The remaining two are set CRDTs which allow for both add-wins and remove-wins policies. In both CRDTs, the different policies are supported through two different remove operations: one for add-wins (i.e., add has precedence) and the other for remove-wins (i.e., the remove has precedence). The key difference between those two CRDTs is that the first one doesn't support an element to be re-added after being removed with the remove operation for remove-wins, but the second one does.

B.1.1 Remove-wins: OA-Set

A set with a remove-wins policy is one in which, when a concurrent *add* and *remove* happen for the same element, the expected result is for the element to not be in the set. The 2P-Set provides a remove-wins policy, but with a twist: after an element is removed, it can never be added again to the set. This behaviour, not only breaks the sequential set semantics but is also undesirable for many applications.

Algorithm B.1 state-based OA-Set CRDT

1: **payload** set A, set R ▷ A, R: sets of pairs (element e, unique-id u).
▷ R: unique-ids generated by *remove*. A: unique-ids removed by *add*
2: **initial** $\emptyset$, $\emptyset$
3: **query** *lookup* (element e) : boolean b
4: **let** $b = (\exists u : (e,u) \in A \wedge (\nexists u' : (e,u') \in R \wedge (e,u') \notin A))$ ▷ All *removes* were overwritten by *add*.
5: **query** *elements* () : set E
6: **let** $E = \{e \mid \exists (e,u) : (e,u) \in A \wedge (\nexists u' : (e,u') \in R \wedge (e,u') \notin A))$
7: **update** *add* (element e)
8: **let** $TR = \{(e,u) \mid \exists u : (e,u) \in R\}$
9: **if** $TR = \emptyset$ **then let** $TR = \{(e, unique())\}$ ▷ *unique*(): generates an unique identifier
10: $A := A \cup TR$
11: **update** *remove* (element e)
12: **pre** *lookup*(e) ▷ Only removes the element if it exists
13: **let** $u = unique()$ ▷ *unique*(): generates an unique identifier
14: $R := R \cup \{(e,u)\}$
15: **compare** (X, Y) : boolean b
16: **let** $b = (X.A \subseteq Y.A \wedge X.R \subseteq Y.R)$
17: **merge** (X, Y) : payload Z
18: **let** $Z.A = X.A \cup Y.A$
19: **let** $Z.R = X.R \cup Y.R$

A state-based specification of a set CRDT which provides a remove-wins policy and allows re-adding elements is shown in Algorithm B.1. We call this CRDT of Observed--Add Set (OA-Set).

The intuition behind the OA-Set is similar to the OR-Set, that is, to tag elements with an unique identifier. However, instead of generating an unique identifier on *add*, it is generated on *remove*. This allows for *adds* to only affect *removes* that happened before (i.e., are observable). As such, for concurrent *add*(e) and *remove*(e), precedence is given to *remove*(e), which guarantees the intended remove-wins policy.

The payload is composed by two sets of pairs (element, unique-id): A for adds and R for removes. On *remove*, an unique id is generated, associated to element and stored in R. On *add*, all unique ids associated to the element being added that are in the source's R set are collected and stored in A. The practical effect of this is that the *add* overrides *removes* that happened before, but doesn't affect any concurrent removes (remove-wins). If the element isn't in R, an unique id is generated and stored in A, in order to mark the element as being in the set. An element is considered to be in the OA-Set (i.e., *lookup* returns true) if the element is in A and all the unique ids for that element in R are also in A, that is, all observable *removes* happened before the latest *add* for that element. The *merge* computes the union of both states for sets A and R.

The OA-Set is *sequentially conformant* with the sequential set specification. In fact, sequential operations on any element and concurrent operations on different elements have the same behaviour as in OR-Set, with the only difference being on how concurrent *adds* and *removes* on the same element are handled: in this set, precedence is given to the remove, which conforms with PCS.

An op-based specification for the OA-Set can be obtained based on the state-based one. On *remove*, the source replica generates an unique id and passes it downstream. The downstream replicas then verify if that unique id wasn't already removed by an *add*, that is, if the id isn't in A. If it isn't, then they associate the id to the element and store them in R. On *add*, the source replica collects all unique ids associated to the element in R and passes it downstream. The downstream replicas then remove all those unique ids from R and store them in A. The *lookup* operation only needs to check if the element isn't in R and is in A. Any delivery order for *add* and *remove* operations is acceptable.

B.1.2 Add-wins with permanent remove: ORPR-Set

As mentioned before, for some applications it may be useful to have a set which provides simultaneously add-wins and remove-wins policies, through the use of different operations. For example, consider a set which represents the users who are currently in a chat room. Usually, an add-wins behaviour may be preferable, since it means that if the connection for a certain user is lost for a brief moment, he will stay in the set (two replicas may concurrently execute a *remove* and an *add* representing, respectively, the loss of connection and re-connection of the user). However, when the user does a *logout*, it makes sense that the user is removed from the set even if concurrent *adds* happen. As such, this is an example of a situation where providing both an add-wins and remove-wins behaviour is beneficial.

Algorithm B.2 state-based ORPR-Set CRDT

1: **payload** set A, set R, set PR ▷ A: pairs (element e, unique-id u); R: removed unique ids (*remove*); PR: removed elements (*permRemove*)
2: **initial** $\emptyset$, $\emptyset$, $\emptyset$
3: **query** *lookup* (element e) : boolean b
4: **let** $b = (\exists u : (e,u) \in A \wedge e \notin PR \wedge u \notin R)$
5: **query** *elements* () : set E
6: **let** $E = \{e \mid \exists (e,u)\text{: } (e,u) \in A \wedge e \notin PR \wedge u \notin R\}$
7: **update** *add* (element e)
8: **let** $u = unique()$ ▷ *unique*(): generates an unique identifier
9: $A := A \cup \{(e,u)\}$
10: **update** *remove* (element e)
11: **pre** *lookup*(e) ▷ Only removes the element if it exists
12: **let** TR $= \{u \mid \exists u : (e,u) \in A\}$
13: $R := R \cup TR$
14: **update** *permRemove* (element e)
15: **pre** *lookup*(e) ▷ Only removes the element if it exists
16: $PR := PR \cup \{e\}$
17: **compare** (X, Y) : boolean b
18: **let** $b = (X.A \subseteq Y.A \wedge X.R \subseteq Y.R \wedge X.PR \subseteq Y.PR)$
19: **merge** (X, Y) : payload Z
20: **let** $Z.A = X.A \cup Y.A$
21: **let** $Z.R = X.R \cup Y.R$
22: **let** $Z.PR = X.PR \cup Y.PR$

A way of providing both add-wins and remove-wins policies is to combine the 2P-Set with the OR-Set. Algorithm B.2 represents the CRDT that results from combining both, which we name of Observed-Remove-Permanent-Remove Set (ORPR-Set). *Remove* is the remove operation for when the add-wins policy is desired, while *permRemove* is used for remove-wins. The main idea is to use a different set for each type of remove: *remove* uses R, while *permRemove* uses PR. If an element is in PR, then it will never be in the set: this is represented by $e \notin PR$ in *lookup*. This effectively grants the remove-wins policy, but doesn't allow an element to be re-added after using *permRemove* (an element can always be re-added it if it only removed by the *remove* operation). *Remove* and *add* work just like in OR-Set. As for *merge* and *compare*, the only difference is that we also need to do, respectively, the union or compare of the PR sets.

Because an element can't be re-added after being removed with *permRemove*, this CRDT isn't *sequentially conformant*, just like the 2P-Set.

B.1.3 Add-wins with remove-wins: OAR-Set

Even though the ORPR-Set provides both add-wins and remove-wins policies, it has the same limitation as 2P-Set: after an element is removed with *permRemove*, it can no longer be re-added. This breaks the sequential semantics of a set and is undesirable for most

applications.

Algorithm B.3 state-based OAR-Set CRDT

1: **payload** set A, set RW, set R
▷ A: *add* pairs (element e, unique-id u)
▷ RW: *removeWins* pairs (element e, unique-id u)
▷ R: removed unique-ids
2: **initial** $\emptyset, \emptyset, \emptyset$
3: **query** *lookup* (element e) : boolean b
4: **let** $b = (\exists u : (e,u) \in A \land u \notin R \land (\nexists u' : (e,u') \in RW \land u' \notin R))$ ▷ There's at least one unique in A not yet *removed* and all *removeWins* were overwritten by *add*.
5: **query** *elements* () : set E
6: **let** $E = \{e \mid \exists (e,u) : (e,u) \in A \land u \notin R \land (\nexists u' : (e,u') \in RW \land u' \notin R)\}$
7: **update** *add* (element e)
8: **let** $u = unique()$ ▷ *unique*(): generates an unique identifier
9: **let** $TR = \{u \mid \exists u : (e,u) \in RW\}$
10: $A := A \cup \{(e,u)\}$
11: $R := R \cup TR$
12: **update** *remove* (element e)
13: **pre** *lookup*(e) ▷ Only removes the element if it exists
14: **let** $TR = \{u \mid \exists u : (e,u) \in A\}$
15: $R := R \cup TR$
16: **update** *removeWins* (element e)
17: **pre** *lookup*(e) ▷ Only removes the element if it exists
18: **let** $u = unique()$ ▷ *unique*(): generates an unique identifier
19: $RW := RW \cup \{(e,u)\}$
20: **compare** (X, Y) : boolean b
21: **let** $b = (X.A \subseteq Y.A \land X.R \subseteq Y.R \land X.RW \subseteq Y.RW)$
22: **merge** (X, Y) : payload Z
23: **let** $Z.A = X.A \cup Y.A$
24: **let** $Z.RW = X.RW \cup Y.RW$
25: **let** $Z.R = X.R \cup Y.R$

As such, we present here the Observed-Add-Remove Set (OAR-Set), which combines the ideas of the OR-Set and OA-Set, providing two removes: *remove* for add-wins and *removeWins* for remove-wins. The state-based and op-based specifications of the OAR-Set are, respectively, in Algorithms B.3 and B.4. We have also submitted a paper [24] covering the OAR-Set, which includes both an unoptimized (equivalent to the one in Algorithm B.3) and an optimized (not covered in this document) version of the state-based OAR-Set.

On the state-based version the payload is formed by three sets: (i) A, which stores the pairs of (element, unique-id) generated by *add*; (ii) RW, which stores the pairs of (element, unique-id) generated by *removeWins*; (iii) R, which stores the unique ids that were removed by either *add* or *remove*. On *removeWins*, an unique id is generated, associated to the element and stored in RW. On *add*, all known unique ids in RW for that element are marked as removed, i.e., stored in R. An *add* only affects *removeWins* that happened

Algorithm B.4 op-based OAR-Set CRDT

1: **payload** set A, set RW, set R
   ▷ A: *add* pairs (element e, unique-id u)
   ▷ RW: *removeWins* pairs (element e, unique-id u)
   ▷ R: removed unique-ids
2: **initial** $\emptyset$, $\emptyset$, $\emptyset$
3: **query** *lookup* (element e) : boolean b
4: **let** $b = (\exists u : (e,u) \in A \wedge \nexists u' : (e,u') \in RW)$ ▷ All *removeWins* were overwritten by add
5: **query** *elements* () : set E
6: **let** $E = \{e \mid \exists (e,u) : (e,u) \in A \wedge \nexists u' : (e,u') \in RW\}$
7: **update** *add* (element e)
8: **atSource** (element e) : unique-id u, set TR
9: **let** $u = unique()$ ▷ *unique*(): generates an unique identifier
10: **let** $TR = \{u \mid \exists u : (e,u) \in RW\}$
11: **downstream** (element e, unique-id u, set TR)
12: **if** $u \notin R$ **then** $A := A \cup \{(e,u)\}$ ▷ No delivery order assumed
13: $RW := RW \setminus \{(e,u) \mid \exists (e,u) : (e,u) \in RW \wedge u \in TR\}$
14: $R := R \cup TR$
15: **update** *remove* (element e)
16: **atSource** (element e) : set TR
17: **pre** *lookup*(e) ▷ Only removes the element if it exists
18: **let** TR $= \{u \mid \exists (e,u) : (e,u) \in A\}$
19: **downstream** (set TR)
20: $A := A \setminus \{(e,u) \mid \exists (e,u) : (e,u) \in A \wedge u \in TR\}$
21: $R := R \cup TR$
22: **update** *removeWins* (element e)
23: **atSource** (element e) : unique u
24: **pre** *lookup*(e) ▷ Only removes the element if it exists
25: **let** $u = unique()$ ▷ *unique*(): generates an unique identifier
26: **downstream** (element e, unique u)
27: **if** $u \notin R$ **then** $RW := RW \cup \{(e,u)\}$ ▷ No delivery order assumed

before, which allows concurrent *removeWins* to win over that *add* (remove-wins).

The *add* operation also generates an unique id and stores it in A along with the element, as in the OR-Set. On *remove*, all known unique ids that are associated to the element in A are marked as removed by storing them in R. Because a *remove* only affects *adds* that happened before, concurrent *adds* win over that *remove* (add-wins).

The *lookup* operation checks if an element is in the OAR-Set, taking into consideration both the add-wins and remove-wins policies. As such, an element e is considered to be in the OAR-Set if: (i) e is in A with at least one unique id not yet removed, i.e., is not in R; (ii) all unique ids for that e in RW have been removed by an *add*, i.e., are in R.

The op-based version is similar to the state-based one, except that on the *downstream* part, care is taken in order to deal with any order for delivery of the operations, allowing

any delivery order to satisfy SEC for this CRDT (causal delivery is needed if causal consistency is also pretended, as explained in Section 2.4.4). If causal delivery is assumed, the op-based version can be optimized: the *R* set will no longer be necessary (as in OR-Set).

The OAR-Set is *sequentially conformant*. As in OR-Set and OA-Set, sequential operations respect the set specification (PSE). Concurrent *add*, *remove* and *removeWins* on different elements naturally commute (PPE), with both *removes* behaving as the sequential set *remove*. As for concurrent *add*, *remove* and *removeWins* on the same element, depending on the combination they can result in either the element being in the set or not: if the three or just *add* and *removeWins* are present, then the element will be removed (remove-wins), while for *add* and *remove* the element will be in the set (add-wins). Any of those cases conforms with PCS, which proves that the OAR-Set is, indeed, *sequentially conformant*.

Appendix

Op-based specification of optimized and incremental versions

Algorithm C.1 Optimized op-based data type for sets, maps, arrays and similar

1: **payload** map O ▷ O: map of parts to sets of received operations.
2: **initial** $\emptyset$
3: **update** *addOp* (operation *op*)
4: **atSource** (operation *op*)
5: **downstream** (operation *op*)
6: $O[op.part].add(op)$

▷ Auxiliary procedure used by query operations.
▷ Calculates the set of active operations and removes irrelevant operations for part p.
7: **procedure** *individualCalculateState* (part p) : set *nonObs*
8: **let** $obsByHB := \{op : op \in O[p] \wedge \exists otherOp \in O[p] : op \prec otherOp \wedge otherOp.hb(otherOp, op)\}$
▷ Operations obsoleted by happens-before aren't needed anymore.
9: $O[p].removeAll(obsByHB)$
▷ Collects operations obsoleted by concurrency.
10: **let** $obsByConcurrency = \{op : op \in O[p] \wedge \exists otherOp \in O[p] : op \parallel otherOp \wedge (op.selfObsoletePolicy(otherOp, op) \vee otherOp.otherObsoletePolicy(otherOp, op))\}$
11: **let** $nonObs = O[p] \setminus obsByConcurrency$

▷ Calculates the active state for every part in O.
12: **procedure** *calculateState* () : set *nonObs*
13: **let** $nonObs = \{op : \exists p \in O : op \in individualCalculateState(p)\}$

▷ readFunction: function supplied by the user that consults the set of active operations for part p (*individualCalculateState*) and returns some kind of result.
14: **query** *individualQuery* (function *readFunction*, part p, arguments *otherArguments*) : result r
15: **let** $r = readFunction(individualCalculateState(p), p, otherArguments)$

▷ Same as *individualQuery* but in this case *readFunction* consults the whole active state.
16: **query** *query* (function *readFunction*, arguments *otherArguments*) : result r
17: **let** $r = readFunction(calculateState(), otherArguments)$

Note: A possible optimization is to cache *nonObs* and use it for query instead of *calculateState*() or *individualCalculateState*(p) until there's a change to, respectively, any part or p.
Note2: A part can be, for example, an element/key/position in a set/map/array.

Algorithm C.2 Incremental op-based data type for sets, maps, arrays and similar

1: **payload** map O ▷ O: map of parts to sets of received operations.
2: **initial** $\emptyset$
3: **update** *addOp* (operation *op*)
4: **atSource** (operation *op*)
5: **downstream** (operation *op*)
6: $O[op.part].add(op)$
7: **let** $obsByHB := \{op : op \in O[op.part] \wedge \exists otherOp \in O[op.part] : op \prec otherOp \wedge otherOp.hb(otherOp, op)\}$
▷ Operations obsoleted by happens-before aren't needed anymore.
8: $O[op.part].removeAll(obsByHB)$

▷ Auxiliary procedure used by query operations.
▷ Calculates the set of active operations for part p.
9: **procedure** *individualCalculateState* (part p) : set *nonObs*
▷ Collects operations obsoleted by concurrency.
10: **let** $obsByConcurrency = \{op : op \in O[p] \wedge \exists otherOp \in O[p] : op \parallel otherOp \wedge (op.selfObsoletePolicy(otherOp, op) \vee otherOp.otherObsoletePolicy(otherOp, op))\}$
11: **let** $nonObs = O[p] \setminus obsByConcurrency$

▷ Calculates the active state for every part in O.
12: **procedure** *calculateState* () : set *nonObs*
13: **let** $nonObs = \{op : \exists p \in O : op \in individualCalculateState(p)\}$

▷ readFunction: function supplied by the user that consults the set of active operations for part p (*individualCalculateState*) and returns some kind of result.
14: **query** *individualQuery* (function *readFunction*, part p, arguments *otherArguments*) : result r
15: **let** $r = readFunction(individualCalculateState(p), p, otherArguments)$

▷ Same as *individualQuery* but in this case *readFunction* consults the whole active state.
16: **query** *query* (function *readFunction*, arguments *otherArguments*) : result r
17: **let** $r = readFunction(calculateState(), otherArguments)$

Note: A possible optimization is to cache *nonObs* and use it for query instead of *calculateState*() or *individualCalculateState*(p) until there's a change to, respectively, any part or p.
Note2: A part can be, for example, an element/key/position in a set/map/array.

Appendix D

State-based Generic CRDT Model

For completeness, we present in this appendix a specification for the state-based version of the generic CRDT model introduced in Chapter 3. We also explain the (few) necessary adaptations for the t-CRDTs that we provide in our t-CRDT library (check Section 4.1) in order for them to work with the state-based version of the model. Finally we discuss how can the optimized and incremental models described in Section 6.1 be adapted for the state-based version.

D.1 Generic model specification

The state-based version of the generic model can be found in Algorithm D.1. It is similar to the op-based version, with both *calculateState* and *query* being completely unchanged. As for *addOp*, the only change is that the state is updated right away on the local replica instead of returning an operation and then doing the changes on downstream.

As for any state-based CRDT, a *merge* operation is needed. In the case of the generic model, the *merge* corresponds to the union of sets O present in both states. Doing the union implies that operations that were previously deleted due to happens-before may be re-added, as one of the replicas might have not yet deleted those operations. This does not pose a problem though, as if one operation (*op*) is deleted from the state then it implies that there's another operation (*otherOp*) in that state, with $op \prec otherOp$ and for which *otherOp.hb(otherOp, op)* returns true. As such, *otherOp* will also be in the union of both states, which implies that *op* will be deleted again the next time *calculateState* is executed.

Optionally, if we want to prevent the merged state from having unnecessary operations (and, as such, have a smaller state), we can calculate the set of operations obsoleted by happens-before (*obsByHB*) as we do the merge and remove those from the merged state.

Algorithm D.1 Generic state-based data type

1: **payload** set O ▷ O: set of received operations.
2: **initial** $\emptyset$
3: **update** *addOp* (operation *op*)
4: $O := O \cup \{op\}$

▷ Auxiliary procedure used by query operations. Calculates the set of active operations and removes unnecessary operations
5: **procedure** *calculateState* () : set *nonObs*
6: **let** $obsByHB = \{op : op \in O \wedge \exists otherOp \in O : op \prec otherOp \wedge otherOp.hb(otherOp, op)\}$
▷ Operations obsoleted by happens-before aren't needed anymore.
7: $O := O \setminus obsByHB$
▷ Collects operations obsoleted by concurrency.
8: **let** $obsByConcurrency = \{op : op \in O \wedge \exists otherOp \in O : op \parallel otherOp \wedge (op.selfPolicy(otherOp, op) \vee otherOp.otherObsoletePolicy(otherOp, op))\}$
9: **let** $nonObs = O \setminus obsByConcurrency$

▷ readFunction: function supplied by the user that consults the set of active operations (calculateState) and returns some kind of result.
10: **query** *query* (function *readFunction*, arguments *otherArguments*) : result *r*
11: **let** $r = readFunction(calculateState(), otherArguments)$
12: **compare** (X, Y) : boolean b
13: **let** $b = (X.O \subseteq Y.O)$
14: **merge** (X, Y) : payload Z
15: **let** $Z.O = X.O \cup Y.O$

Note: A possible optimization is to cache *nonObs* and use it for query instead of *calculateState*() until there's a change to the state

This modified version of merge can be found in Figure D.1 and, in practice, it exchanges time efficiency for a smaller state, as calculating *obsByHB* has a time complexity of $O(n^2)$, with n being the number of operations in the union of both states.

merge (X, Y) : payload Z
let $allOps = X.O \cup Y.O$
let $obsByHB = \{op : op \in allOps \wedge \exists otherOp \in allOps : op \prec otherOp \wedge otherOp.hb(otherOp, op)\}$
let $Z.O = allOps \setminus obsByHB$

Figure D.1: Modified merge for the state-based generic model which deletes unnecessary operations.

D.2 Adapting t-CRDTs for the state-based version

Adapting existing t-CRDTs to work with the state-based generic model is actually quite simple, as it only requires two small changes:

1. in every update operation, get rid of the **atSource** and **downstream** parts and call *addOp* right away;

2. add a compare and merge operations which call, respectively, the compare and merge operations of the generic model.

As an example, we show in Algorithm D.2 the set t-CRDT that was presented in Section 4.1.3 adapted to the state-based model. Note that no changes are necessary for the queries or the policies.

Algorithm D.2 state-based set t-CRDT

```
1: update add (policy hbP, policy selfP, policy otherP, element e, clock clk)
2:     let op = [hbPolicy → hbP, selfObsoletePolicy → selfP, otherObsoletePolicy → otherP,
   type → add, element → e, clock → clk]
3:     addOp(op)
4: update remove (policy hbP, policy selfP, policy otherP, element e, clock clk)
5:     let op = [hbPolicy → hbP, selfObsoletePolicy → selfP, otherObsoletePolicy → otherP,
   type → remove, element → e, clock → clk]
6:     addOp(op)
7: query lookup (element e) : boolean b      ▷ Assumes that for the same element an add
   and remove can't be simultaneously active
8:     let b = (∃op ∈ calculateState() : op.type = add ∧ op.element = e)
9: query elements () : set E                                  ▷ Same assumption as lookup
10:    let E = {e : ∃op ∈ calculateState() : op.type = add ∧ op.element = e}
11: compare (X, Y) : boolean b
12:    let b = compare(X, Y)                    ▷ Call to the generic's model compare.
13: merge (X, Y) : payload Z
14:    let Z = merge(X, Y)                      ▷ Call to the generic's model merge.
```

D.3 Optimized model

The optimization described in Section 6.1 consists in changing the state from being a set of operations to being a map, which allows to represent the state of data types such as sets, maps and arrays more efficiently. Considering the set, the key idea behind this optimization is that two different elements are usually independent from each other, that is, the usual policies for the set always return false if the involved operations are for different elements. This allows us to separate the state in multiple "parts", more precisely, we can represent the state as a map of elements to set of operations. This makes *calculateState* more efficient and even allows to compute the active state only for one "part" (*individualCalculateState*) for some queries such as *lookup*.

The changes required relatively to the generic model are the same as described in Section 6.1, with one extra detail – *merge* and *compare* also need to be modified as follows:

- *merge(X, Y)* – for each key present in both sets, do the union of the sets associated to that key. For keys only present in one of the states, keep the key and its set;
- *compare(X, Y)* – returns true if all keys in X are in Y and, for each key in X, the set associated that key in X is contained in Y for the same key.

D.4 Incremental model

The incremental version described in Section 6.1 can also be adapted to a state-based version. The changes to *addOp*, *calculateState* and *individualCalculateState* are the same as for the op-based version. *Compare* is equal to the optimized version, but additional changes are required for *merge*, as it now must calculate the set of operations obsoleted by happens-before and remove them from the merged state (as the queries no longer do that). This can be achieved by doing similar changes to what was done to *merge* in Figure D.1 when compared with Algorithm D.1.

Appendix E

TLA+ correctness invariants for the set t-CRDT

In this appendix we present the complete process of verifying that our set t-CRDT respects the three correctness principles introduced in Section 2.4.2. Besides introducing the invariants for PPE and PCS for the set t-CRDT (the invariant for PSE was already introduced in Section 5.5), we also explain how we guarantee that these invariants are verified for every possible execution history.

We note that this process was also done for the remaining t-CRDTs included in our library and discussed in Chapter 4. We ran TLC on the specifications of the other t-CRDTs and verified that they all ensure PSE, PPE and PCS, along with other data-type specific invariants. In this appendix we will only, however, include the TLA+ specification of these principles for the set t-CRDT.

E.1 Replaying the history

To verify the correctness principles for each t-CRDT, we took the approach of specifying operators which, based on the execution history of each replica, replayed every state transition and verified that for each transition all of the correctness principles hold true. As discussed in Chapter 5, since TLC will generate every possible system state, this implies that TLC will also generate every possible history. As such, if we specify an operator which based on an history replays every state transition that occurred in that history, then we can verify the correctness principles for every possible state transition.

To have access to the history of a given replica, a small change is needed in the generic model specification, which for simplicity reasons was omitted in Chapter 5. This change consists in adding an extra variable *history*, which is a structure that maps replicas to their histories. We define an history as being the order for which operations were executed in

the replica the history belongs to. Besides adding the variable an extra change is required – we also need to modify the *addOp* operator to, besides storing the operation, also add it to the replica's history. The modified *addOp* can be found in Figure E.1.

```
macro addOp(op)
begin
  history[self] := Append(history[self], op);
  ops := ops ∪ {op};
end macro;
```

Figure E.1: Modified *addOp* macro that also stores an operation in the history.

Given the complete history of a replica, we can replay every state transition of that replica. To achieve this, intuitively what we need to do is to start with an empty state and iterate the operations in the history by insertion order and, in each step, add the current operation to the previous state. As we do each step (or state transition), we need to verify that the three principles PPE, PSE and PCS hold true when passing from the "previous" state to the "new" state due to the addition of the current operation.

```
correctnessPrinciples ≜ (∀p ∈ PROCESSES : finished[p] = TRUE) ⟹
                            ∀p1 ∈ PROCESSES :
                                prepareStates(p1, {}, hist[p1])

prepareStates(p, prevState, hist) ≜
    IF (Len(hist) > 0) THEN
        LET op        ≜ Head(hist)
            newState  ≜ prevState ∪ {op}
            prevNonHB ≜ prevState \ getHB(prevState)
            prevNonObs ≜ prevNonHB \ getConcurrentObs(prevNonHB)
            newNonHB  ≜ newState \ getHB(newState)
            newNonObs ≜ newNonHB \ getConcurrentObs(newNonHB)
        IN  noOtherElementsAffected(op, prevNonObs, newNonObs)
            ∧ verifyPSE(op, prevNonHB, newNonHB, prevNonObs,
                  newNonObs)
            ∧ verifyPPE(op, prevNonObs, newNonObs)
            ∧ verifyPCS(op, prevNonObs, newNonObs)
            ∧ prepareStates(p, newState, Tail(hist))
    ELSE TRUE
```

Figure E.2: TLA+ operators to verify the correctness principles.

Figure E.2 contains two operators, *correctnessPrinciples* and *prepareStates*. The first one is fairly simple, as the only thing it does is call the *prepareStates* operator for every replica after all replicas have already converged. Since the whole history is replayed, it is sufficient to only verify the principles after all replicas converged. *CorrectnessPrinciples* can be seen as an aggregation invariant for every state transition and every correctness principle and is, thus, the invariant that is added to the models to be verified by TLC.

As for *prepareStates*, it is a recursive operator that, in short, builds a state transition and verifies it. It receives as arguments the process to verify, the state with the operations that

were verified so far and the remaining of the history. Each call to *prepareStates* calculates the new state after adding the operation on the start of the history (*op*) and the resulting states of removing the operations obsoleted by happens-before and by concurrency from both the previous and the new state, which is represented by **LET**. Afterwards, the invariant for each principle is called, passing as arguments the necessary states and the operation that was added and, finally, the next step of *prepareStates* is called with the new state and with the latest operation applied removed from the history, thus ensuring that the whole history will be verified.

E.2 Verifying PSE

The *noOtherElementsAffected* and *verifyPSE* were already explained in Section 5.5 and their specification can be found in Figure 5.16. To recapitulate, *noOtherElementsAffected* verifies a necessary condition for every principle which consists in checking that an operation only affects the state relative to the element referred by that operation, leaving the remaining of the state intact. As for *verifyPSE*, as the name suggests it verifies that the sufficient conditions for PSE are not violated in the state transition. A curious reader might notice that the *verifyPSE* invariant in Figure 5.16 also calls *noOtherElementsAffected*, which is unnecessary as *prepareStates* already does so. In the original specification that we wrote, in fact *verifyPSE* doesn't call *noOtherElementsAffected*, but we had to adapt the specification due to not having introduced *prepareStates* and *correctnessPrinciples* in Section 5.5.

E.3 Verifying PPE

The PPE property requires that the execution of commutative operations leads to the same final state independently of their execution order. To ensure this, we define the following conditions as sufficient to ensure PPE:

1. any *add* or *remove* on a given element *e* has no effect on other elements, i.e., queries return the same result except for possibly *e*. This implies that operations for different elements are commutative, as each operation will only affect the part of the state relative to its own element;
2. assuming that no concurrent *remove* was executed for a given element *e*, two or more *adds* executed concurrently for *e* lead to the same visible state, i.e., *lookup*(*e*) returns true, independently of their execution order;
3. assuming that no concurrent *add* was executed for a given element *e*, two or more *removes* executed concurrently for *e* lead to the same visible state, i.e., *lookup*(*e*) returns false, independently of their execution order;

The last two conditions might seem odd, as they respectively assume that no concurrent *removes* or *adds* were executed concurrently. We note that if this isn't the case, then we have a conflicting scenario – one or more *adds* and one or more *removes* being

executed concurrently for the same element. In this case the operations aren't naturally commutative and thus it should be verified by PCS and not PPE.

```
                verifyPPE(op, prevState, newState)  ≜
(2)                 ((op.type = ADD ∧ ∃ otherOp ∈ prevState :
(2)                     otherOp.e = op.e ∧ otherOp.type = ADD) ⟹
(2)                         lookup(prevState, op.e) ∧ lookup(newState, op.e))
(3)                 ∨ ((op.type = REMOVE ∧ ∃ otherOp ∈ prevState :
(3)                     otherOp.e = op.e ∧ otherOp.type = REMOVE) ⟹
(3)                         ¬lookup(prevState, op.e) ∧ ¬lookup(newState, op.e)
```

Figure E.3: TLA+ invariant for PPE property. The numbers between parenthesis match with the conditions each line is verifying.

Figure E.3 contains the TLA+ specification of the PPE invariant. The arguments *prevState* and *newState* correspond to the set of active operations that we have, respectively, before and after adding *op*. The first condition for PPE is ensured by *noOtherElementsAffected*, which is called by *prepareStates* before calling *verifyPPE*. As for the second condition, it corresponds to the lines marked with (2), which verify that if element *op.e* was already in the state due to an *add* operation and *op* is an *add*, then after executing *op* the element *op.e* will still be in the state, which implies that *adds* for the same element are commutative. The third condition is verified in the lines marked with (3) and the idea is similar to the *add* verification. Precisely, it verifies that if element *op.e* wasn't in the state due to an *remove* operation and *op* is a *remove*, then after executing *op* the element *op.e* still won't be in the state, which implies that *removes* for the same element are commutative. As such, *verifyPPE* verifies the three sufficient conditions for PPE, thus ensuring that our set t-CRDT respects PPE.

E.4 Verifying PCS

The PCS property is a bit vague, as it only requires that the state resulting from executing two conflicting operations is either a state with an error mark or the state obtained by executing the operations in one of the possible execution orders. For the set, the only situation in which there are conflicting operations is when one or more *adds* and one or more *removes* for the same element are executed concurrently. Intuitively, we want to make sure that having a conflict for one element doesn't affect any other element, as that would be odd. However, the policies we defined in Section 4.1.3 ensure an interesting property (which was verified by the previously introduced *noSimulatenousAddRemove* invariant) – in no situation there is both an *add* and a *remove* for the same element simultaneously in the active state. This allows us to verify for PCS that, independently of conflicting operations having been executed or not, if only *adds* (resp. *removes*) for element *e* are in the state, then *lookup*(*e*) must return true (resp. *false*).

Considering what was discussed above, we formally define that the following conditions are sufficient to ensure PCS for the set t-CRDT:

1. any *add* or *remove* on a given element *e* has no effect on other elements, i.e., queries return the same result except for possibly *e*. This is required to ensure operations for different elements don't conflict when executed concurrently;
2. if only *adds* for a given element *e* are in the state (equivalently, no *removes* for *e* are in the state and at least one operation for *e* was already executed), then *lookup*(*e*) must return true;
3. if only *removes* for a given element *e* are in the state (equivalently, no *adds* for *e* are in the state and at least one operation for *e* was already executed), then *lookup*(*e*) must return false;

Note that for the last two conditions we don't need to verify if, respectively, an *add* or *remove* is in the state, as this is already implied by the previously introduced invariant *atLeastOneOp*. This invariant ensures that if at least one operation was already executed for a given element *e*, then at least one operation for *e* is in the active state. Thus, if for example no *adds* are in the state, then at least one *remove* is.

```
    verifyPCS(op, prevState, newState)  ≜
(2)     ((∄ otherOp ∈ newState :
(2)         otherOp.e = op.e ∧ otherOp.type = REMOVE) ∧ lookup(newState, op.e))
(3)     ∨ ((∄ otherOp ∈ newState :
(2)         otherOp.e = op.e ∧ otherOp.type = ADD) ∧ ¬lookup(newState, op.e))
```

Figure E.4: TLA+ invariant for PCS property. The numbers between parenthesis match with the conditions each line is verifying.

Figure E.4 contains the TLA+ specification of the PCS invariant. The arguments *prevState* and *newState* have the same meanings as before. Similarly to the PPE invariant, the first condition is ensured by *noOtherElementsAffected* which is called before *verifyPCS* for the same state transition. The second condition corresponds to the lines marked with (2), which verify that if no *remove* for element *op.e* is in the state then *lookup*(*op.e*) returns true. Similarly, the third condition corresponds to the lines marked with (3) and verify that if no *add* for element *op.e* is in the state then *lookup*(*op.e*) returns false. Thus, *verifyPCS* verifies the three sufficient conditions for PCS, which ensures that our set t-CRDT respects PCS.

E.5 Conclusion

In this appendix we discussed the process of verifying the correctness principles for a t-CRDT, which consists in identifying the sufficient conditions for each principle and then writing invariants that verify if those conditions hold true for every possible state

transition. To exemplify, we showed the TLA+ specification of the invariants for each principle and their conditions for the set t-CRDT.

We remind the reader that this process was done for every t-CRDT included in our library that was introduced in Chapter 4, thus ensuring that our t-CRDTs behave correctly when used with the provided policies.