HAL
open science

Designing safe and highly available distributed applications

Sreeja Sasidharan Nair

To cite this version:

Sreeja Sasidharan Nair. Designing safe and highly available distributed applications. Distributed, Parallel, and Cluster Computing [cs.DC]. Sorbonne Université, 2021. English. NNT : . tel-03339393v1

HAL Id: tel-03339393

https://theses.hal.science/tel-03339393v1

Submitted on 9 Sep 2021 (v1), last revised 27 Jan 2022 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE DE DOCTORAT DE
SORBONNE UNIVERSITÉ

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Sreeja Sasidharan NAIR

Pour obtenir le grade de

DOCTEUR de SORBONNE UNIVERSITÉ

Sujet de la thèse :

Designing safe and highly available distributed applications

soutenue le 1 juillet 2021

devant le jury composé de :

M. Marc Shapiro	Directeur de thèse
M. Carlos Baquero	Rapporteur
M. Éric Gressier-Soudan	Rapporteur
Mme. Béatrice Bérard	Examinateur
Mme. Carla Ferreira	Examinateur
M. Bradley King	Examinateur
M. Martin Kleppmann	Examinateur
M. Gustavo Petri	Examinateur

Abstract

Distributed applications replicate data to ensure high availability and low latency. To provide high availability, they support concurrent updates to the replicas. Even if replicas eventually converge, they may diverge temporarily, for instance when the network fails. The developer needs to make sure that the application state remains safe despite the temporary divergence. As described by CAP theorem[1], designing distributed applications involves fundamental trade-offs between safety and performance. We focus on the cases where safety is the top requirement.

In the first part of this thesis, for the subclass of state-based distributed systems, we propose a proof methodology for establishing that a given application maintains a given invariant. Our approach allows reasoning about individual operations separately. We demonstrate that our rules are sound, and we illustrate their use with some representative examples. We provide a mechanized proof engine for the rule using Boogie, an SMT-based tool.

The developer can choose between two forms of concurrency control - conflict resolution or coordination. The second part of the thesis presents a case study of conflict resolution in a demanding data structure: the tree.

The tree is a basic data structure present in many applications. We present a novel replicated tree data structure that supports coordination-free concurrent atomic moves, and arguably maintains the tree invariant. Our analysis identifies cases where concurrent moves are inherently safe. For the remaining cases we devise a coordination-free, rollback-free algorithm. The trade-off is that in some cases a move operation "loses" (i.e., is interpreted as skip). We prove that the data structure is convergent and maintains the tree invariant. The response time and availability of our design compares favourably with competing approaches in the literature.

The final part of the thesis develops a methodology for selecting a distributed lock configuration. Given the coordination required by an application for safety, it can be implemented in many different ways. Even restricting to locks, they can use various configurations, differing by lock granularity, type, and placement. The performance of each configuration depends on workload. We study the "coordination lattice", i.e., design space of lock configurations, and show by simulation how lock configuration impacts performance for a given workload. The lattice represents the dimensions of distributed lock configurations, and we show how to systematically navigate them.

Acknowledgements

First of all I would like to thank my advisor Marc Shapiro for his guidance and support throughout these years. This work would not have been possible without his encouragement and faith in me. Our weekly meetings and all his open-ended questions contributed a lot to this work and my personal development.

I am grateful to have my knowledgeable and supportive collaborators: Gustavo Petri, Carla Ferreira, Mário Perreira and Filipe Meirim. Thank you Gustavo, Carla, Mário and Filipe for our weekly meetings and all the ideas that came out of it.

I would like to thank all my jury members for their time and effort in reading and reviewing this work. I would also like to thank Éric Gressier-Soudan and Béatrice Bérard for being part of my comité de suivi, giving me reassurance each year.

I want to thank Lightkone European project, and RainbowFS French project for funding my thesis and providing opportunities for collaborations. The project meetings provided a venue to discuss my ideas and results. I would like to thank the collaborators of the project for their support. I also thank the jury members of the Séphora Bérrebi scholarships for granting me the award.

I wish to thank Jonathan Lejeune, Julien Sopena, Antoine Miné, Julia Lawall, Martin Kleppmann, the anonymous reviewers of ESOP 2019, PaPoC 2019, OOPSLA 2019, POPL 2020, ESOP 2020, ESOP 2021, ICDCS 2021, PODC 2021 for all the inputs and review comments which contributed to improving this work.

I am indebted to all my colleagues at LIP6, especially the members of Delys, who supported me throughout these years with their constant support and encouragement. Special thanks to Dimitrios Vasilas, Paolo Viotti, Vinh Tao, Alejandro Tomsic, Vincent Vallade, Francis Laniel, Ilyas Toumlilt, Jonathan Sid-Otmane, Saalik Hatia, Sara Hamouda, Benoît Martin, and Laurent Prosperi for making my time at the lab memorable. I want to thank Saalik, Benoît, Francis and Ilyas for all the translations throughout the years.

I am fortunate to have cheerful neighbours who helped me remain positive during the exceptional COVID-induced confinement period. My work during the confinement wouldn't have been pleasant without the nice weekend trips and picnics. My special thanks to Marianne, who took time to check grammatical errors in this work, whatever remains is mine.

Last, but not the least, I want to thank my family and friends for supporting me through this journey. Thanks to my parents, Sasidharan and Sudha, parents-in-law, Velayudhan and Prabhavathy, brother Sreehari, siblings-in-law Sandhya, Anjali, Sudheer, and niece Parvathi for their love and support. I am grateful to have a supportive husband and son,

who moved with me to France for this work. My greatest thanks to them for their love, care, and patience.

To my beloved husband Jayesh and my wonderful son Abhinav

Table of Contents

Page

List of Tables **ix**

List of Figures **xi**

1 Introduction **1**

1.1 Contributions 2

1.1.1 Proving invariant safety for highly-available distributed applications 2

1.1.2 A safe, convergent and highly-available replicated tree 2

1.1.3 Trade-offs in distributed concurrency control 3

I Verifying the design of distributed applications **4**

Introduction to Part I **5**

2 Proving invariant safety for highly-available distributed applications **8**

2.1 System Model 8

2.1.1 General Principles 8

2.1.2 Notations and Assumptions 10

2.1.3 Operational Semantics 10

2.1.4 Operational Semantics with State History 12

2.1.5 Correspondence between the semantics 13

2.2 Proving convergence: Strong Eventual Consistency 15

2.2.1 Semilattice of auction object 17

2.3 Proving Invariants 17

2.3.1 Invariance Conditions 19

2.3.2 Applying the proof rule 21

2.3.3 Coordination for Invariant Preservation 23

3 Use cases **27**

3.1 Distributed Barrier 27

3.2 Replicated lock 28

3.3 Courseware 29

3.3.1 Pseudocode of courseware 30

4 Automation **33**
4.1 Specifying a distributed application in Soteria . . . 33
4.2 Verification passes . . . 34
4.2.1 Syntax check . . . 34
4.2.2 Convergence check . . . 35
4.2.3 Safety check . . . 35
4.3 Tool evaluation . . . 37

5 Related work **38**

6 Conclusion of Part I and Future work **40**
6.1 Future work . . . 40

II Designing conflict resolution policies **42**

Introduction to Part II **43**

7 Design of a safe, convergent and coordination free replicated tree **44**
7.1 System Model . . . 46
7.2 Properties and associated proof rules . . . 47
7.2.1 Sequential safety . . . 48
7.2.2 Concurrency . . . 48
7.2.2.1 Convergence . . . 48
7.2.2.2 Precondition stability . . . 49
7.2.2.3 Independence . . . 49
7.2.3 Mechanized verification . . . 49
7.3 Sequential specification of a tree . . . 50
7.3.1 State . . . 50
7.3.2 Invariant . . . 50
7.3.3 Operations . . . 51
7.3.4 Mechanized verification of the sequential specification . . . 53
7.4 Concurrent tree specification . . . 55
7.4.1 Precondition stability . . . 55
7.4.1.1 Stability of add operation . . . 55
7.4.1.2 Stability of remove operation . . . 58
7.4.1.3 Stability of move operation . . . 59
7.5 Safety of concurrent moves . . . 61
7.5.1 Classifying moves . . . 62
7.5.2 Coordination-free conflict resolution for concurrent moves . . . 62
7.6 Convergence . . . 63
7.7 Independence . . . 63
7.7.1 Independence of add operation . . . 64
7.7.2 Independence of remove operation . . . 64

7.7.3 Independence of up-move operation 64
7.7.4 Independence of down-move operation 66
7.8 Safe specification of a replicated tree 66
7.8.1 Mechanized verification of the concurrent specification 68
7.9 Discussion 69
7.9.1 Moving from causal consistency to eventual consistency 69
7.9.2 Message overhead for conflict resolution 69
7.9.3 Computing the set of concurrent moves 70

8 Experimental study and Comparison 71

9 Related work 74

10 Conclusion of Part II and Future work 76
10.1 Future work 76

III Selecting Distributed Concurrency Control 78

Introduction to Part III 79

11 Exploring the coordination lattice 81
11.1 System Model 81
11.1.1 Application model 81
11.1.2 Network model 82
11.1.3 Workload characteristics 82
11.2 Dimensions of Concurrency control 82
11.2.1 Granularity 83
11.2.2 Mode 84
11.2.3 Lock Placement 84
11.3 The Coordination Lattice 85
11.4 Navigating the coordination lattice 87
11.4.1 Granularity selection 87
11.4.2 Mode selection 87
11.4.3 Placement selection 91

12 Experiments 92
12.1 Experimental setup 92
12.1.1 Inputs 92
12.1.2 Intermediate processes 92
12.1.3 DisLockSim - A simulation model for distributed lock 94
12.1.4 Cost of locking 94
12.2 Analysing some conflict graphs 95
12.2.1 Conflict graph involving two operations 95
12.2.2 Conflict graph involving three operations 98

13 Related work **105**

14 Conclusion of Part III and Future work **107**

14.1 Future work 107

List of Tables

Table **Page**

4.1 Time taken for analysing specification using Soteria 37

7.1 Stability analysis of sequential specification 60
7.2 Result of commutativity analysis 63
7.3 Result of dependency analysis. The cell shows the condition under which the operation in the row is independent of the operation in the column. . . 69

8.1 Latency configurations in ms 71

12.1 Average latency between replicas 94
12.2 Coordination configurations for lock l_{ab-P_x} from Figure 12.3b 96
12.3 Workloads for the conflict graph involving two operations. 96
12.4 Coordination configurations for the coordination lattice in Figure 12.5b . . . 98
12.5 Different workloads for the conflict graph involving three operations. 99

List of Figures

Figure **Page**

1.1 Evolution of state of an auction application 6

2.1 Precise Operational Semantics: Messages 11
2.2 Semantic Rules with a History of States 13
2.3 Simulation Schema 14
2.4 Monotonic semilattice conditions (implies Strong Eventual Consistency) . . 16
2.5 Semilattice of an auction object 17
2.6 Invariant Conditions 19
2.7 Evolution of state in an auction application with concurrency control 23
2.8 Safe in concurrent executions 24

7.1 Concurrent cycle causing moves 45
7.2 Move update violating tree invariant 53
7.3 Resolving conflict of concurrent remove and add 57
7.4 Critical ancestors and critical descendants 61

8.1 Experimental results. Each bar is the average of 15 runs. The error bars show standard deviation. 72

11.1 Two conflicting operations. 82
11.2 Three conflicting operations. 83
11.3 Average response time (in s) for lock acquisition requests to a 3-instance Zookeeper service. The subplots indicate different modes of locks and the groups indicate different placements. A centralised lock is placed in Paris; a clustered lock is clustered across Paris, Cape Town and New York; a distributed lock is distributed across all locations. 85
11.4 Conflict graphs 86
11.5 Conflict graph of a sample application with four conflicting operations . . . 86

12.1 The architecture of dislock experiment. The first row lists the inputs, and the second row the preprocessing stage. The dotted box represents the simulator. The yellow shaded regions represent the physical location. 93

12.2 Cost of locking for different lock placements from different replicas. The X-axis shows placement and mode configurations. The Y-axis represents response time in ms. 95
12.3 Conflict graph and coordination lattice for a single lock 95
12.4 Plots of CcRepExecTime and total execution time obtained from the experiments for different workloads for the conflict graph in Figure 12.3. . . . 97
12.5 Conflict graph and coordination lattice for two locks 98
12.6 Plots of CcRepExecTime and total execution time obtained from the experiments for different workloads for Figure 12.5. 103

Chapter 1

Introduction

Many modern applications serve users accessing shared data in different geographical regions. Examples include social networks, online multi-player games, cooperative engineering, collaborative editors, source-control repositories, or distributed file systems. One approach would be to store the application's data in a single central location, accessed remotely. However, users far from the central location would suffer long delays and outages.

Instead, the data is replicated to several locations. A user accesses the closest available replica. To ensure availability, an update must not synchronize across replicas; otherwise, when a network partition occurs, the system would block. Thus, a replica executes both queries and updates locally, and propagates its updates to other replicas asynchronously. Asynchronous replication improves availability at the expense of consistency.[1]

Updates at different locations are concurrent; this may cause replicas to diverge, at least temporarily. Replicas may diverge, but if the system ensures Strong Eventual Consistency (SEC), this ensures that replicas that have received the same set of updates have the same state [2], simplifying reasoning.

The replicated data may also require to maintain some (application-specific) invariant, i.e., an assertion about the object. We say a state is safe if the invariant is true in that state; the system is safe if every reachable state is safe. In a sequential system, this is straightforward (in principle): if the initial state is safe, and the final state of every update individually is safe, then the system is safe. However, these conditions are not sufficient in the replicated case, because concurrent updates at different replicas may interfere with one another, leading to unsafe states.

This can be fixed by coordinating between some or all types of updates. To maximize availability and latency, such coordination should be minimized.

Identifying the pair of operations that conflict is the first step. The application can be either redesigned with conflict resolution policies for the conflicting operations or it can use some form of coordination to avoid conflicting updates from executing concurrently.

Conflict resolution avoids coordination at some cost such as "losing updates". A deterministic policy decides the winning operation among the pair of concurrent conflicting operations. A client that saw a successful operation might later find that the effect of

[1] There is also synchronous replication; it is consistent but not available in the event of a network partition.

the operation has been lost, and will have to retry. Hence this approach is advisable for applications that can afford having non-definitive operations, i.e., operations that might not generate any effect.

To ensure safety for the class of applications where the operations have to be definitive, the conflicting operations must coordinate. There are multiple ways to coordinate, the most common being the use of locks. The choice of configuration for the lock will have an impact on the performance of the distributed application. The performance impact depends on the workload characteristics.

In this thesis we propose a methodology for designing correct and highly available distributed applications. We investigate the following research questions:

- *How to guarantee safety for highly available distributed applications despite concurrent updates?*
- *When and how can a distributed application execute safely without any coordination, even in the presence of conflicting operations?*
- *If coordination is unavoidable, what are the associated costs that can guide our choice?*

1.1 Contributions

1.1.1 Proving invariant safety for highly-available distributed applications

Distributed applications propagate either operations or the state to remote replicas to ensure that the data is consistent. Propagating operations require higher guarantees from the delivery layer such as causality and exactly-once delivery. Hence state-based update propagation is widely used in the industry. To answer the first question, we propose a novel proof system specialized to proving the safety of available distributed applications that converge by propagating state. This specialization supports modular reasoning, and thus it enables automation. We demonstrate that this proof system is sound. We present Soteria, the first tool supporting the verification of program invariants for state-based replicated objects. When Soteria succeeds, it ensures that every execution is safe, whether replicas are disconnected or concurrent. Otherwise, Soteria provides the list of methods that conflict. We present a number of representative case studies, which we run through Soteria.

1.1.2 A safe, convergent and highly-available replicated tree

A tree is a complex data structure that has strong properties such as being acyclic and having a unique root. Tree data structure is used in several applications, for example a file system. Concurrent moves are known to cause cycles in a tree [3]. To answer the second question, we present the design of a coordination-free, highly available and safe replicated tree data structure called Maram. We classify the move operation, into two types called up-move and down-move, prove that only concurrent down-moves causes a cycle, and design a

conflict resolution with this information. We also compute the conditions under which the conflict resolution might cause safety issues for the dependent operations and augment the conflict resolution policy to address this. The conflict resolution is coordination-free. We prove the safety of Maram with the conflict resolution. Compared to the existing solutions with locks, Maram requires no coordination and results in a lower response time.

1.1.3 Trade-offs in distributed concurrency control

To answer the third question, we study the subclass of coordination, distributed locks. In particular, we study the impact of a coordination configuration, for a given workload, on the performance of a distributed application (assuming we get the list of conflicts). We study the effects of granularity, mode and placement of locks. We introduce the *coordination lattice* which is a lattice constructed with the three dimensions of concurrency control. We present a systematic creation and navigation of the *coordination lattice* for a given application and workload. We present a simulator that can provide insight into the performance impact of a given lock configuration; this helps guide the developer in choosing an appropriate trade-off. We propose a set of rules that guide the concurrency control selection by navigating the coordination lattice and illustrate the soundness of the rules with the help of the simulator. These rules guide the developer to choose a coordination configuration for the safety of the distributed application with minimal performance impact.

Part I

Verifying the design of distributed applications

Introduction to Part I

This part of the thesis presents a proof rule for verifying the safety of distributed applications.

A distributed application often replicates its data to several locations, and accesses some available replica. Examples include social networks, online multi-player games, cooperative engineering tools, collaborative editors, source control repositories, or distributed file systems. To ensure availability, an update must not synchronize across replicas; otherwise, when a network partition occurs, the system would block. The drawback is that asynchronous updates may cause replicas to diverge or to violate the data invariants.

To address the first problem, Conflict-free Replicated Data Types (CRDTs)[2] have mathematical properties to ensure that all replicas that received the same set of updates converge to the same state [2]. To ensure availability, a CRDT replica executes both queries and updates locally and immediately, without remote coordination. It propagates its updates to the other replicas asynchronously.

There are two basic approaches to update propagation: to propagate operations, or states. In the operation-based approach, an update is first applied to some origin replica, then sent as an operation to remote replicas, which in turn apply it to update their local state. Operation-based CRDTs require the the message delivery layer to deliver messages in causal order, exactly once; the set of replicas must be known.

In the state-based approach, an update is applied to some origin replica. Occasionally, an updated replica sends its full state to some other replica, which merges the received state into its own. In turn, this replica will later send its own state to yet another replica. As long as every update eventually reaches every replica transitively, messages may be dropped, re-ordered or duplicated, and the set of replicas may be unknown. Replicas are guaranteed to converge if the set of states, as a result of updates and merge, forms a monotonic semi-lattice [2]. Due to these relaxed requirements, state-based CRDTs have better adoption [4]. They are the focus of this work.

As a running example, consider a simple auction system. For simplicity, we consider a single auction composed of the following parts:

- Its `Status`, which can move from initial state `INVALID` (under preparation) to `ACTIVE` (can receive bids) and then to `CLOSED` (no more bids accepted).
- The `Winner` of the auction, which is initially $\perp$, and eventually becomes the bid taking the highest amount. In case of ties, the bid with the lowest id wins.
- The set of `Bids` placed, which is initially empty. A bid is a tuple composed of

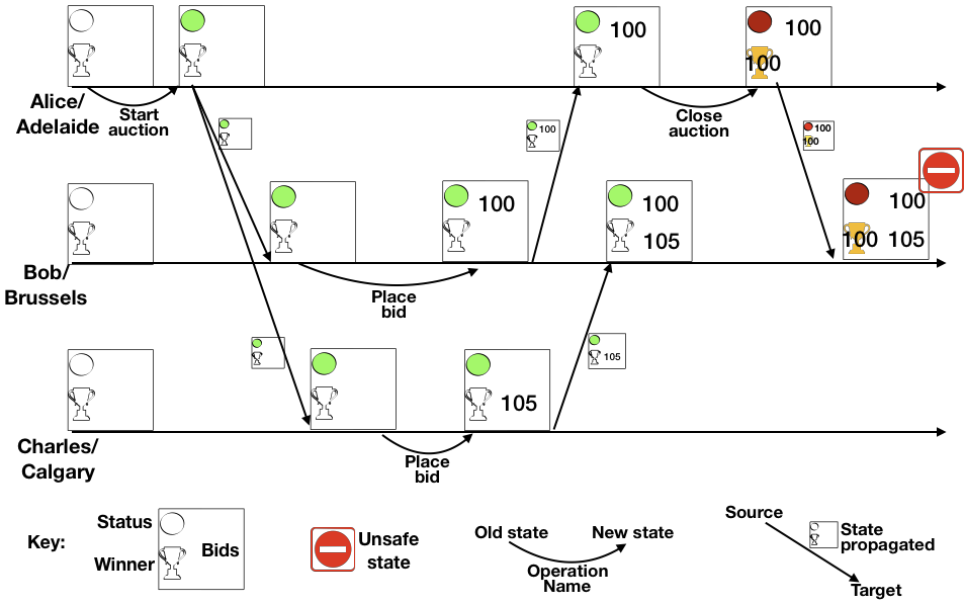


Figure 1.1: Evolution of state of an auction application

- `BidId`: A unique identifier
- `Placed`: A boolean flag to indicate whether the bid has been placed or not. Initially, it is `FALSE`. Once placed, a bid cannot be withdrawn.
- The monetary `Amount` of the bid; this cannot be modified once the bid is placed.

Figure 1.1 illustrates how the auction state evolves over time. The state of the application is geo-replicated at data centers in Adelaide, Brussels, and Calgary. Users at different locations can start an auction, place bids, close the auction, declare a winner, inspect the local replica, and observe if a winner is declared and who it is. Update is propagated asynchronously to other replicas. All replicas should eventually agree on the same auction status, the same set of bids and the same winner.

Figure 1.1 shows the state-based approach with local operations and merges. Alternatives exist where only a delta of the state —that is, the portion of the state not known to be part of the other replicas— is sent as a message [5]; since this is an optimisation, it is of no consequence to the results of this work.

Looking back to Figure 1.1, we can see that replicas diverge temporarily. This temporary divergence can lead to an unsafe state, in this case declaring a lower amount bid as the winner. This correctness problem has been addressed before; however, previous works mostly consider the operation-based propagation approach [6–9].

In contrast to previous work,[2] we consider state-based distributed applications.We find that, the specific properties of state-based propagation enable simple modular reasoning despite concurrency, thanks to the concept of *concurrency invariant*. Our proof methodology derives the concurrency invariant automatically from the sequential specification. Now, if the initial state is safe, and every update maintains both the application invariant and the concurrency invariant, then every reachable state is safe, even in concurrent

[2]Discussed in detail in Chapter 5

executions, regardless of network partitions.

We developed a tool named Soteria to automate our proof methodology. Soteria analyses the specification to detect concurrency bugs and either proves safety or provides counterexamples.

The contributions of this part are as follows:

- We propose a novel proof system, specialized in proving the safety of available applications that propagate state. This specialisation supports modular reasoning, and thus enables automation.
- We demonstrate that this proof system is sound. Moreover, we provide a simple semantics for state-propagating systems that allows us to ignore network messages altogether.
- We present Soteria, the first tool supporting the verification of program invariants for state-based distributed applications. When Soteria succeeds it ensures that every execution, whether replicas are partitioned or concurrent, is safe.
- We present a number of representative case studies, which we run through Soteria.

Gustavo Petri collaborated with us for this part of the thesis. Our contribution resulted in two publications, Nair et al. [10, 11], and an open source tool, Nair et al. [12].

Chapter 2

Proving invariant safety for highly-available distributed applications

In this chapter we present a proof methodology to ensure that a given state-based distributed application is system-safe, for a given invariant and a given amount of concurrency control.

2.1 System Model

2.1.1 General Principles

An application consists of a state, a set of operations, a merge function and an invariant. As our running example, Figure 1.1 illustrates three replicas of an auction application, at three different locations, represented by the three horizontal lines. Each line depicts the evolution of the state of the corresponding replica; time flows from left to right.

State

A distributed system consists of a number of servers, with disjoint memory and processing capabilities. The servers might be distributed over geographical regions. A set of servers at a single location stores the state of the application. This is called a single *replica*. We assume that a replica executes sequentially. The replicas are at different geographical locations, each one having a full copy of the state. In the simplest case (for instance at initialisation) the state at all replicas will be identical. The state of some replica is called its *local state*. The global view, comprising all local states, is called the *global state*.

Operations

Each replica may perform the operations defined for the application. To support availability, an operation modifies the local state at some arbitrary replica, the *origin replica* for that operation, without synchronising with other replicas (the cost of synchronisation

being significant at geo-distributed scale). An operation might consist of several changes; these are applied to the replica as a single atomic unit.

Executing an operation on its origin replica takes effect immediately. However, the state of the other replicas, called *remote replicas*, remains unaffected at this point. The remote replicas get updated when the state is eventually propagated. An immediate consequence of this execution model is that, in the presence of concurrent operations, replicas can reach different states, i.e., they diverge.

Let us illustrate this with our example in Figure 1.1. Initially, the auction is yet to start, the winner is not declared and no bids are placed. By default, a replica can execute any operation, `start_auction`, `place_bid`, and `close_auction`, locally without synchronising with other replicas. We see that the local states of replicas occasionally diverge. For example at the point where operation `close_auction` completes at the Adelaide replica, the Adelaide replica is aware of a single $100 bid, whereas the Calgary replica observes another bid for $105, and the Brussels replica observes both the bids.

State Propagation

A replica occasionally propagates its state to other replicas in the system. A replica receiving a remote state *merges* it into its own.

In Figure 1.1, the arrows between replicas represent the delivery of a message containing the state of the source replica, to be merged into the target replica. A message is labelled with the state propagated. For instance, the first message delivery at the Brussels replica represents the result of updating the local state (setting auction status to `ACTIVE`), with the state originating in the replica at Adelaide (auction started).

Similarly to operations, a merge is atomic. In Figure 1.1, Alice closes the auction at the Adelaide replica. This atomically sets the status of the auction to `CLOSED` and declares a winner from the set of bids it is aware of. The updated auction state and winner are transmitted together. Merging is performed atomically at the Brussels replica.[1]

We now specify the `merge` operation for an auction. The receiving replica's local state is denoted $\sigma = (\text{status}, \text{winner}, \text{Bids})$, the received state is denoted $\sigma' = (\text{status}', \text{winner}', \text{Bids}')$ and the result of merge is denoted as $\sigma_{new} = (\text{status}_{new}, \text{winner}_{new}, \text{Bids}_{new})$.

```
merge((status,winner,Bids),(status′,winner′,Bids′)) :
  status_new := max(status,status′)
  winner_new := winner′ ≠ ⊥ ? winner′ : winner
  Bids_new := {(id, placed ∨ placed′, max(amount, amount′)) |
        (id, placed, amount) ∈ Bids ∧ (id, placed′, amount′) ∈ Bids′}
```

Furthermore, we require the operations and merge to be defined in a way that ensures convergence. We discuss the relevant properties later, in Section 4.2.

Data Invariants

A data invariant is an assertion that must evaluate to true in every local state of every replica. Although evaluated locally at each replica, data invariant is in effect global, since

[1]Note that this leads to an unsafe state. We discuss this in detail in Subsection 2.3.2

it must be true at all replicas, and replicas eventually converge. For our running example, data invariant can be stated as follows:

- Only an active auction can receive bids, and
- the highest unique placed bid wins when the auction closes (breaking ties using bid identifiers).

This data invariant must hold true in all possible executions of the application.

2.1.2 Notations and Assumptions

Let us introduce some notations and assumptions:

- We assume a fixed set of replicas, ranged over with the meta-variable $\mathtt{r} \in \mathtt{R}$ sampled from the domain of unique replica names $\mathtt{R}$.
- We denote a local state with the meta-variable $\sigma \in \Sigma$ ranged over the domain of states of the application Σ.
- The *local semantic* function $[\![]\!]$ takes an operation and a state, and returns the state after applying the operation. We write $[\![\mathtt{op}]\!](\sigma) = \sigma_{new}$ for executing operation $\mathtt{op}$ on state σ resulting in a new state σ_{new}.
- Ω denotes a partial function returning the current state of a replica. For instance $\Omega(\mathtt{r}) = \sigma$ means that in global state Ω, replica $\mathtt{r}$ is in local state σ. We will use the notation $\Omega[\mathtt{r} \leftarrow \sigma]$ to denote the global state resulting from replacing the local state of replica $\mathtt{r}$ with σ. The local state of all other replicas remains unchanged in the resulting global state.[2]
- A message propagating states between replicas is denoted $\langle\ \mathtt{r} \xrightarrow{\sigma} \mathtt{r}'\ \rangle$. This represents the fact that replica $\mathtt{r}$ has sent a message (possibly not yet received) to replica $\mathtt{r}'$, with the state σ as its payload. The meta-variable $\mathtt{M}$ denotes the messages in transit in the network.
- In Subsection 2.1.3, we will utilise a set of states to record the history of the execution. The set of past states will be ranged over with the variable $\mathtt{S} \in \mathbb{P}(\Sigma)$, where $\mathbb{P}()$ indicates the power set.
- All replicas are assumed to start in the same initial state σ_i. Formally, for each replica $\mathtt{r} \in \mathsf{dom}(\Omega_i)$ we have $\Omega_i(\mathtt{r}) = \sigma_i$.

2.1.3 Operational Semantics

In this subsection and Subsection 2.1.4 we will present two semantics for systems propagating states. Importantly, while the first semantics takes into account the effects of the network on the propagation of the states, and is hence an accurate representation of the

[2]This notation of a global state is used only pedagogically to explain our proof rule. The global state is not observable and formally, the rule is based only on the local state of each replica.

$$\textsc{Operation}\quad \frac{\Omega(\mathtt{r}) = \sigma \qquad [\![\mathsf{op}]\!](\sigma) = \sigma_{new} \qquad \Omega_{new} = \Omega[\mathtt{r} \leftarrow \sigma_{new}]}{(\Omega, \mathtt{M}) \rightarrow (\Omega_{new}, \mathtt{M})}$$

$$\textsc{Send}\quad \frac{\Omega(\mathtt{r}) = \sigma \qquad \mathtt{r}' \in \mathsf{dom}(\Omega) \setminus \{\mathtt{r}\} \qquad \mathtt{M}_{new} = \mathtt{M} \cup \{\langle\ \mathtt{r} \xrightarrow{\sigma} \mathtt{r}'\ \rangle\}}{(\Omega, \mathtt{M}) \rightarrow (\Omega, \mathtt{M}_{new})}$$

$$\textsc{Merge}\quad \frac{\begin{array}{c}\Omega(\mathtt{r}) = \sigma \qquad \langle\ \mathtt{r}' \xrightarrow{\sigma'} \mathtt{r}\ \rangle \in \mathtt{M} \\ \mathtt{M}_{new} = \mathtt{M} \setminus \{\langle\ \mathtt{r}' \xrightarrow{\sigma'} \mathtt{r}\ \rangle\} \qquad [\![\mathtt{merge}]\!](\sigma, \sigma') = \sigma_{new} \qquad \Omega_{new} = \Omega[\mathtt{r} \leftarrow \sigma_{new}]\end{array}}{(\Omega, \mathtt{M}) \rightarrow (\Omega_{new}, \mathtt{M}_{new})}$$

$$\textsc{Op \& Broadcast}\quad \frac{\begin{array}{c}\Omega(\mathtt{r}) = \sigma \qquad [\![\mathsf{op}]\!](\sigma) = \sigma_{new} \qquad \Omega_{new} = \Omega[\mathtt{r} \leftarrow \sigma_{new}] \\ \mathtt{M}_{new} = \mathtt{M} \cup \{\ \langle\ \mathtt{r} \xrightarrow{\sigma_{new}} \mathtt{r}'\ \rangle \mid \mathtt{r}' \in \mathsf{dom}(\Omega) \setminus \{\mathtt{r}\}\ \}\end{array}}{(\Omega, \mathtt{M}) \rightarrow (\Omega_{new}, \mathtt{M}_{new})}$$

$$\textsc{Merge \& Broadcast}\quad \frac{\begin{array}{c}\Omega(\mathtt{r}) = \sigma \qquad \langle\ \mathtt{r}' \xrightarrow{\sigma'} \mathtt{r}\ \rangle \in \mathtt{M} \\ \mathtt{M}_{new} = \mathtt{M} \setminus \{\langle\ \mathtt{r}' \xrightarrow{\sigma'} \mathtt{r}\ \rangle\} \qquad [\![\mathtt{merge}]\!](\sigma, \sigma') = \sigma_{new} \qquad \Omega_{new} = \Omega[\mathtt{r} \leftarrow \sigma_{new}] \\ \mathtt{M}_{new'} = \mathtt{M}_{new} \cup \{\ \langle\ \mathtt{r} \xrightarrow{\sigma_{new}} \mathtt{r}''\ \rangle \mid \mathtt{r}'' \in \mathsf{dom}(\Omega) \setminus \{\mathtt{r}\}\ \}\end{array}}{(\Omega, \mathtt{M}) \rightarrow (\Omega_{new}, \mathtt{M}_{new'})}$$

Figure 2.1: Precise Operational Semantics: Messages

execution of systems with state propagation, we will show in the next subsection that reasoning about the network is unnecessary in this kind of system. We will demonstrate this claim by presenting a much simpler semantics in which the network is abstracted away. The importance of this reduction is that the number of events to be considered, both when conducting proofs and when reasoning about applications, is greatly reduced. As informal evidence of this claim, we point at the difference in complexity between the semantic rules presented in Figure 2.1 and Figure 2.2. We postpone the equivalence argument to Theorem 2.1.6.

Figure 2.1 presents the semantic rules describing what we shall call the *precise semantics* (we will later present a more abstract version) defining the transition relations describing how the state of the application evolves.

The figure defines a semantic judgement of the form $(\Omega, \mathtt{M}) \rightarrow (\Omega_{new}, \mathtt{M}_{new})$ where $(\Omega, \mathtt{M})$ is a configuration where the replica states are given by Ω as shown above, and $\mathtt{M}$ is a set of messages that have been transmitted by different replicas and are pending to be received by their target replicas.

Rule Operation presents the state transition resulting from a replica `r` executing an operation `op`. The operation queries the state of replica `r`, evaluates the semantic function for operation `op` and updates its state with the result. The set of messages `M` does not

change. The second rule, SEND, represents the non-deterministic sending of the state of replica $\mathtt{r}$ to replica $\mathtt{r}'$. The rule has no other effect than to add a message to the set of pending messages $\mathtt{M}$. The MERGE rule picks any message, $\langle\ \mathtt{r}' \xrightarrow{\sigma'} \mathtt{r}\ \rangle$, in the set of pending messages $\mathtt{M}$, and applies the merge function to the destination replica with the state in the payload of the message, removing $\langle\ \mathtt{r}' \xrightarrow{\sigma'} \mathtt{r}\ \rangle$ from $\mathtt{M}$.

The final two rules, OP & BROADCAST and MERGE & BROADCAST represent the specific case when the states are immediately sent to all replicas. These rules are not strictly necessary since they are subsumed by the application of either OPERATION or MERGE followed by one SEND per replica. We will, however, use them to simplify a simulation argument in what follows.

We remark at this point that no assumptions are made about the duplication of messages or the order in which messages are delivered. This is in contrast to other works on the verification of properties of replicated applications [6, 9]. The reason why this assumption is not a problem in our case is that the least-upper-bound assumption of the `merge` function, as well as the inflation assumptions on the states considered in Subsection 4.2.2 (Section 4.2) mean that delayed messages have no effect when they are merged.

As customary we will denote with $(\Omega, \mathtt{M}) \xrightarrow{*} (\Omega_{new}, \mathtt{M}_{new})$ the repeated application of the semantic rules zero or more times, from the state $(\Omega, \mathtt{M})$ resulting in the state $(\Omega_{new}, \mathtt{M}_{new})$.

It is easy to see how the example in Figure 1.1 proceeds according to these rules for the auction.

For liveness, we require that an update is always broadcasted to other replicas and is eventually delivered. The following lemma, to be used later, establishes that whenever we use only the broadcast rules, for any intermediate state in the execution, and for any replica, when considering the final state of the trace, either the replica has already observed a fresher version of the state in the execution, or there is a message pending for it with that state. This is an obvious consequence of broadcasting.

Lemma 2.1.1 *If we consider a restriction to the semantics of Figure 2.1 where instead of applying the* OPERATION *rule of Figure 2.1 we apply the* OP & BROADCAST *rule always, and instead of applying the* MERGE *rule we apply* MERGE & BROADCAST *always, we can conclude that given an execution starting from an initial global state Ω_i with*

$$(\Omega_i, \emptyset) \xrightarrow{*} (\Omega, \mathtt{M}) \xrightarrow{*} (\Omega_{new}, \mathtt{M}_{new})$$

for any two replicas $\mathtt{r}$ and $\mathtt{r}'$ and a state σ such that $\Omega(\mathtt{r}) = \sigma$, then either:

- $\Omega_{new}(\mathtt{r}') \geq \sigma$, *or*
- $\langle\ \mathtt{r} \xrightarrow{\sigma} \mathtt{r}'\ \rangle \in \mathtt{M}_{new}$.

2.1.4 Operational Semantics with State History

We now turn our attention to a simpler semantics where we omit messages from configurations, but instead, we record in a separate set all the states occurring in any replica throughout the execution.

$$\frac{\Omega(\mathtt{r}) = \sigma \qquad [\![\mathsf{op}]\!](\sigma) = \sigma_{new} \qquad \Omega_{new} = \Omega[\mathtt{r} \leftarrow \sigma_{new}]}{(\Omega, \mathtt{S}) \rightarrow (\Omega_{new}, \mathtt{S} \cup \{\sigma_{new}\})} \quad \textsc{Operation}$$

$$\frac{\Omega(\mathtt{r}) = \sigma \qquad \sigma' \in \mathtt{S} \qquad [\![\mathtt{merge}]\!](\sigma, \sigma') = \sigma_{new} \qquad \Omega_{new} = \Omega[\mathtt{r} \leftarrow \sigma_{new}]}{(\Omega, \sigma) \rightarrow (\Omega_{new}, \mathtt{S} \cup \{\sigma_{new}\})} \quad \textsc{Merge}$$

Figure 2.2: Semantic Rules with a History of States

The semantics in Figure 2.2 presents a judgement of the form $(\Omega, \mathtt{S}) \rightarrow (\Omega_{new}, \mathtt{S}_{new})$ between configurations of the form $(\Omega, \mathtt{S})$ as before, but where the set of messages is replaced by a set of states denoted with the meta-variable $\mathtt{S} \in \mathbb{P}(\Sigma)$.

The rules are simple. Operation executes an operation as before, and it adds the resulting new state to the set of observed states. The rule Merge non-deterministically selects a state in the set of states and it merges a non-deterministically chosen replica with it. The resulting state is also added to the set of observed states.

Lemma 2.1.2 *Consider a state $(\Omega, \mathtt{S})$ reachable from an initial global state Ω_i with the semantics of Figure 2.2. Formally: $(\Omega_i, \{\sigma_i\}) \xrightarrow{*} (\Omega, \mathtt{S})$. We can conclude that the set of recorded states in the final configuration $\mathtt{S}$ includes all of the states present in any of the replicas*

$$\Big(\bigcup_{\mathtt{r} \in \mathsf{dom}(\Omega)} \{\Omega(\mathtt{r})\} \Big) \subseteq \mathtt{S}$$

2.1.5 Correspondence between the semantics

In this section, we show that removing the messages from the semantics, and choosing to record states instead renders the same executions. To that end, we will define the following relation between configurations of the two semantics which will be later shown to be a bisimulation.

Definition 2.1.3 (Bisimulation Relation) *We define the relation $\mathcal{R}_{\Omega_i}$ between a configuration $(\Omega, \mathtt{M})$ of the semantics of Figure 2.1 and a configuration $(\Omega, \mathtt{S})$ of the semantics of Figure 2.2 parameterized by an initial global state Ω_i and denoted by*

$$(\Omega, \mathtt{M}) \ \mathcal{R}_{\Omega_i} \ (\Omega, \mathtt{S})$$

when the following conditions are met:

1. $(\Omega_i, \emptyset) \xrightarrow{*} (\Omega, \mathtt{M})$, *and*
2. $(\Omega_i, \{\sigma_i\}) \xrightarrow{*} (\Omega, \mathtt{S})$, *and*
3. $\{ \ \sigma \mid \langle \ \mathtt{r} \xrightarrow{\sigma} \mathtt{r}' \ \rangle \in \mathtt{M} \ \} \subseteq \mathtt{S}$

$$
\begin{tikzcd}
(\Omega_i, \emptyset) \arrow[r, "*"] & (\Omega, \mathtt{M}) \arrow[r] \arrow[d, dash, "\mathcal{R}_{\Omega_i}"] & (\Omega_{new}, \mathtt{M}_{new}) \arrow[d, dashed, dash, "\mathcal{R}_{\Omega_i}"] \\
(\Omega_i, \{\sigma_i\}) \arrow[r, "*"] & (\Omega, \mathtt{S}) \arrow[r, dashed] & (\Omega_{new}, \mathtt{S}_{new})
\end{tikzcd}
$$

(a) Precise to History-preserving Simulation

$$
\begin{tikzcd}
(\Omega_i, \{\sigma_i\}) \arrow[r, "*"] & (\Omega, \mathtt{S}) \arrow[r] \arrow[d, dash, "\mathcal{R}_{\Omega_i}"] & (\Omega_{new}, \mathtt{S}_{new}) \arrow[d, dashed, dash, "\mathcal{R}_{\Omega_i}"] \\
(\Omega_i, \emptyset) \arrow[r, "*"] & (\Omega, \mathtt{M}) \arrow[r, dashed] & (\Omega_{new}, \mathtt{M}_{new})
\end{tikzcd}
$$

(b) History-preserving to Precise Simulation

Figure 2.3: Simulation Schema

In other words, two states represented in the two configurations are related if both are reachable from an initial global state and all the states transmitted by the messages ($\mathtt{M}$) is present in the history ($\mathtt{S}$).

We can now show that this relation is indeed a bisimulation. We first show that the semantics of Figure 2.2 simulates that of Figure 2.1. That is, all behaviours produced by the precise semantics with messages can also be produced by the semantics with history states. This is illustrated in the commutativity diagram of Figure 2.3a and Figure 2.3b, where the dashed arrows represent existentially quantified components that are proven to exist in the theorem.

Lemma 2.1.4 (State-semantics simulates Messages-semantics) *Consider a reachable state* $(\Omega, \mathtt{M})$ *from the initial state* Ω_i *in the semantics of Figure 2.1. Consider moreover that according to that semantics there exists a transition of the form*

$$(\Omega, \mathtt{M}) \rightarrow (\Omega_{new}, \mathtt{M}_{new})$$

and consider that there exists a state $(\Omega, \mathtt{S})$ *of the history preserving semantics of Figure 2.2 such that they are related by the simulation relation*

$$(\Omega, \mathtt{M})\ \mathcal{R}_{\Omega_i}\ (\Omega, \mathtt{S})$$

We can conclude that, as illustrated in Figure 2.3a, there exists a state $(\Omega_{new}, \mathtt{S}_{new})$ *such that*

$$(\Omega, \mathtt{S}) \rightarrow (\Omega_{new}, \mathtt{S}_{new}) \qquad \textit{and} \qquad (\Omega_{new}, \mathtt{M}_{new})\ \mathcal{R}_{\Omega_i}\ (\Omega_{new}, \mathtt{S}_{new})$$

We will now consider the lemma showing the inverse relation. To that end we will consider a special case of the semantics of Figure 2.1 where instead of applying the Operation rule, we will always apply the Op & Broadcast rule, and instead of the Merge rule, we will apply Merge & Broadcast. As we mentioned before, this is equivalent to the application of the Operation/Merge rule, followed by a sequence of applications of Send. The reason we will do this is that we are interested in showing that for any execution of the semantics in Figure 2.2 there is an equivalent (simulated) execution of the semantics of Figure 2.1. Since all states can be merged in the semantics of Figure 2.2 we have to assume that in the semantics of Figure 2.1 the states have been sent with messages. Fortunately, we can choose how to instantiate the existential send messages to apply the rules as necessary, and that justifies this choice.

Lemma 2.1.5 (Messages-semantics simulates State-semantics) *Consider a reachable state* $(\Omega, \mathtt{S})$ *from the initial state* Ω_i *in the semantics of Figure 2.2. Consider moreover that according to that semantics there exists a transition of the form*

$$(\Omega, \mathtt{S}) \rightarrow (\Omega_{new}, \mathtt{S}_{new})$$

and consider that there exists a state $(\Omega, \mathtt{M})$ *of the state-preserving semantics of Figure 2.2 such that they are related by the simulates relation*

$$(\Omega, \mathtt{M}) \ \mathcal{R}_{\Omega_i} \ (\Omega, \mathtt{S})$$

We can conclude that there exists a state $(\Omega_{new}, \mathtt{M}_{new})$ *such that*

$$(\Omega, \mathtt{M}) \rightarrow (\Omega_{new}, \mathtt{M}_{new}) \qquad \textit{and} \qquad (\Omega_{new}, \mathtt{M}_{new}) \ \mathcal{R}_{\Omega_i} \ (\Omega_{new}, \mathtt{S}_{new})$$

As before, an illustration of this lemma is presented in Figure 2.3b.

We can now conclude that the two semantics are bisimilar:

Theorem 2.1.6 (Bisimulation) *The semantics of Figure 2.1 and Figure 2.2 are bisimilar as established by the relation defined in Definition 2.1.3.*

The theorem above justifies carrying out our proofs with respect to the semantics of Figure 2.2, which has fewer rules, and better aligns with our proof methodology. This also justifies that when reasoning semantically about state-propagating application systems we can generally ignore the effects of network delays and messages.

From the standpoint of concurrency, the system model allows the execution of asynchronous concurrent operations, where each operation is executed atomically in each replica, and the aggregation of results of different operations is performed lazily as replicas exchange their state. At this point, we assume the set of states, along with the operations and merge, forms a monotonic semi-lattice. This is a sufficient condition for Strong Eventual Consistency [2, 4, 13].

We have seen that even though we achieve convergence later, there can be instances or even long periods of time during which replicas might diverge. We need to ensure that the concurrent executions are still safe. In the next section, we discuss how to ensure safety of distributed applications built on top of the system model we described.

2.2 Proving convergence: Strong Eventual Consistency

Figure 2.4 lists the sufficient conditions for a highly-available distributed application to ensure Strong Eventual Consistency (SEC). To simplify the notation, we assume that when an operation executes, the local state respects the precondition of that operation.

Let us now consider these conditions in turn:

- The first condition Poset checks that the ordering relation of the state defines a partially ordered set(poset): reflexive, transitive, and anti-symmetric.

$$(\Sigma, \sqsubseteq) \text{ is a poset} \qquad \text{(Poset)}$$

$$\forall\ \sigma, \sigma',\ \exists \sigma_{new}, [\![\texttt{merge}]\!](\sigma, \sigma') = \sigma_{new} \qquad \text{(Total)}$$

$$\forall\ \sigma, \sigma_{new},\ [\![\texttt{merge}]\!](\sigma, \sigma) = \sigma_{new} \Rightarrow \quad \sigma = \sigma_{new} \qquad \text{(Idempotent)}$$

$$\forall\ \sigma, \sigma', \sigma_{new0}, \sigma_{new1},\ \left(\begin{array}{c} [\![\texttt{merge}]\!](\sigma, \sigma') = \sigma_{new0} \\ \wedge\ [\![\texttt{merge}]\!](\sigma', \sigma) = \sigma_{new1} \end{array} \right) \Rightarrow \quad \sigma_{new0} = \sigma_{new1} \qquad \text{(Commutative)}$$

$$\forall\ \sigma, \sigma', \sigma'', \sigma_{new0}, \sigma_{new1},\ \left(\begin{array}{c} [\![\texttt{merge}]\!]([\![\texttt{merge}]\!](\sigma, \sigma'), \sigma'') = \sigma_{new0} \\ \wedge\ [\![\texttt{merge}]\!](\sigma, [\![\texttt{merge}]\!](\sigma', \sigma'')) = \sigma_{new1} \end{array} \right) \Rightarrow \quad \sigma_{new0} = \sigma_{new1} \qquad \text{(Associative)}$$

$$\forall\ \mathsf{op}, \sigma, \sigma_{new},\ [\![\mathsf{op}]\!](\sigma) = \sigma_{new} \Rightarrow \quad \sigma \sqsubseteq \sigma_{new} \qquad \text{(Inflation)}$$

$$\forall\ \sigma, \sigma', \sigma_{new},\ [\![\texttt{merge}]\!](\sigma, \sigma') = \sigma_{new} \Rightarrow \left(\begin{array}{c} \sigma \sqsubseteq \sigma_{new} \\ \wedge \sigma' \sqsubseteq \sigma_{new} \end{array} \right) \qquad \text{(UB)}$$

$$\forall\ \sigma, \sigma', \sigma_{new}, \sigma^*,\ \sigma \sqsubseteq \sigma^* \wedge \sigma' \sqsubseteq \sigma^* \wedge [\![\texttt{merge}]\!](\sigma, \sigma') = \sigma_{new} \Rightarrow \quad \sigma_{new} \sqsubseteq \sigma^* \qquad \text{(LUB)}$$

Figure 2.4: Monotonic semilattice conditions (implies Strong Eventual Consistency)

We then find a number of conditions on the `merge` function.

- The second condition, Total, says that the merge function is total.
- Conditions Idempotent, Commutative and Associative say that the merge function is idempotent, commutative and associative [4].
- Condition Inflation says that each operation `op` of the application is an inflation.
- Related to the condition above, condition UB ensures that the result of `merge` is an upper-bound of the input states. This, along with condition Inflation, is a sufficient condition for convergence, since it implies that there is a deterministic way to reconcile any two replicas that have diverged in their states through the least-upper-bound of the lattice implemented by the merge function, and also implies that the states of all replicas are progressing in the same direction (w.r.t. the ordering function) in the lattice (see Figure 2.8). It remains to see that there is a deterministic state to which all replicas will converge (assuming that no new operations arrive).
- The fourth and final condition, LUB, ensures that `merge` function is the least upper bound as per the given order. This condition guarantees that the state reached by merging multiple states is unique, making the `merge` function deterministic, and thus

guaranteeing equality at the point where all replicas have exchanged their respective states.

All these conditions ensure that the distributed application guarantees strong eventual consistency in the case where all the replicas receive copies of states incorporating all prior updates.

2.2.1 Semilattice of auction object

Let us show that our running example of an auction application converges. Subsection 2.2.1 represent the ordering relation as a semilattice following Figure 1.1. It is not hard to see that each of our operations is an inflation, and that the merge operation computes the least-upper-bound.

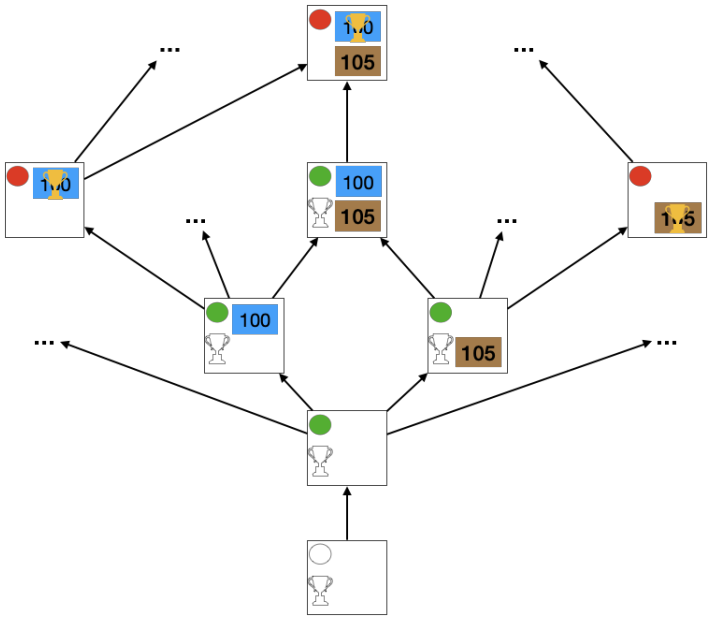


Figure 2.5: Semilattice of an auction object

2.3 Proving Invariants

In this section, we report our invariant verification strategy. Specifically, we consider the problem of verifying *data invariants* of highly-available distributed applications.

To support the verification of data invariants we will consider a syntactic-driven approach based on program logic. Bailis et al.[14] identify necessary and sufficient run-time conditions to establish the safety of application invariants for highly-available distributed databases in a criterion dubbed I-confluence. Moreover, they study the validity of a number of typical invariants and applications. Our work improves on the I-confluence criterion defined in [14] by providing a static, syntax-driven, and mostly-automatic mechanism to verify the correctness of an invariant for an application. We will address the specific differences in Chapter 5, related work.

An important consequence of our verification strategy is that while we are proving invariants about a concurrent highly-distributed system, our verification conditions are modular (on the number of API operations), and can be carried out using standard sequential Hoare-style reasoning. These verification conditions in turn entail stability of the assertions as one would have in a logic like Rely/Guarantee.

Let us start by assuming that a given initial state for the application is denoted σ_i. Initially, all replicas have σ_i as their local state. As explained earlier, each replica executes a sequence of state transitions, due either to a local update or to a merge incorporating remote updates.

Let us call *safe state* a replica state that satisfies the invariant. Assuming the current state is safe, any update (local or merge) must result in a safe state. To ensure this, every update is equipped with a precondition that disallows any unsafe execution.[3]

Formally, an update u (an operation or a merge), mutates the local state σ, to a new state $\sigma_{new} = u(\sigma)$. To preserve the data invariant, Inv_{data}, we require that the local state respects the precondition of the update, Pre_u:

$$\sigma \in \mathsf{Pre}_u \implies u(\sigma) \in \mathsf{Inv}_{data}$$

Thus, a local update executes only when, at the origin replica, the current state is safe and its precondition currently holds.

Similarly, merge must also be safe. Since merge can happen at any time, it must be the case that its precondition is always true, i.e., it constitutes an additional invariant. We call this the *concurrency invariant*. Now our global invariant consists of two parts: first, the data invariant (Inv_{data}), and second, the concurrency invariant(Inv_{conc}).

To illustrate local preconditions, consider an operation `close_auction(w:BidId)`, which sets auction status to `CLOSED` and the winner to `w` (of type `BidId`). The developer may have written a precondition such as `status` = `ACTIVE` because closing an auction doesn't make sense otherwise. In order to ensure the invariant that the winner has the highest amount, one needs to strengthen it with the clause `is_highest(Bids, w)`, defined as[4]

```
∀ b ∈ Bids, b.placed ⟹ b.Amount ≤ w.Amount
```

To illustrate the precondition of merge, let us use our running example. We wish to maintain the invariant that the highest bid is the winner. Assume a scenario where the local replica declared a winner and closed the auction. An incoming state from a remote replica contains a bid with a higher amount. When the two states are merged, we see that the resulting state is unsafe. So we must strengthen the merge operation with a precondition. We require a predicate that is at least as strong as the weakest precondition that satisfies the data invariant, Inv_{data}. The strengthened precondition looks like this:

```
status = CLOSED ⟹ ∀ Bids ∈ ℙ(𝔹), is_highest(Bids, w)
∧ status′ = CLOSED ⟹ ∀ Bids ∈ ℙ(𝔹), is_highest(Bids, w′)
```

[3] Technically, this is at least the weakest-precondition of the update for safety. It strengthens any *a priori* precondition that the developer may have set for the business logic.

[4] Ideally, this is the weakest precondition to satisfy Inv_{data}.

$$\sigma_i \models \mathsf{Inv}_{data} \qquad \text{(SafeInit)}$$

$$\forall\ \mathsf{op}, \sigma, \sigma_{new}, \left(\begin{array}{c} \sigma \models \mathsf{Pre}_{\mathsf{op}} \wedge \\ \sigma \models \mathsf{Inv}_{data} \wedge \\ [\![\mathsf{op}]\!](\sigma) = \sigma_{new} \end{array}\right) \Rightarrow \quad \sigma_{new} \models \mathsf{Inv}_{data} \qquad \text{(SafeOp)}$$

$$\forall\ \sigma, \sigma', \sigma_{new}, \left(\begin{array}{c} (\sigma, \sigma') \models \mathsf{Pre}_{\mathtt{merge}} \wedge \\ \sigma \models \mathsf{Inv}_{data} \wedge \\ \sigma' \models \mathsf{Inv}_{data} \wedge \\ [\![\mathtt{merge}]\!](\sigma, \sigma') = \sigma_{new} \end{array}\right) \Rightarrow \quad \sigma_{new} \models \mathsf{Inv}_{data} \qquad \text{(SafeMerge)}$$

$$(\sigma_i, \sigma_i) \models \mathsf{Inv}_{conc} \qquad \text{(ConcInit)}$$

$$\forall\ \mathsf{op}, \sigma, \sigma', \sigma_{new}, \left(\begin{array}{c} \sigma \models \mathsf{Pre}_{\mathsf{op}} \wedge \\ (\sigma, \sigma') \models \mathsf{Inv}_{conc} \wedge \\ [\![\mathsf{op}]\!](\sigma) = \sigma_{new} \end{array}\right) \Rightarrow \quad (\sigma_{new}, \sigma') \models \mathsf{Inv}_{conc} \qquad \text{(ConcOp)}$$

$$\forall\ \sigma, \sigma', \sigma_{new}, \left(\begin{array}{c} (\sigma, \sigma') \models \mathsf{Pre}_{\mathtt{merge}} \wedge \\ (\sigma, \sigma') \models \mathsf{Inv}_{conc} \wedge \\ [\![\mathtt{merge}]\!](\sigma, \sigma') = \sigma_{new} \end{array}\right) \Rightarrow \quad (\sigma_{new}, \sigma') \models \mathsf{Inv}_{conc} \qquad \text{(ConcMerge)}$$

Figure 2.6: Invariant Conditions

where $\mathbb{B}$ is the set of all possible bids. This means that if the status is `CLOSED` in either of the two states, the winner should be the highest bid in any state. This condition ensures that when a winner is declared, it is the highest bid among the set of bids in any state at any replica.

2.3.1 Invariance Conditions

The verification conditions in Figure 2.6 ensure that for any reachable local state of a replica, the global invariant $\mathsf{Inv}_{data} \wedge \mathsf{Inv}_{conc}$, is a valid assertion. We assume the invariant to be a Hoare-logic style assertion over the state of the application. In a nutshell, all of these conditions check (i) the precondition of each of the operations, and that of the merge operation uphold the global invariant, and (ii) the global invariant of the application consists of the invariant and the concurrency invariant (precondition of `merge`).

We will develop this intuition in what follows. Let us now consider each of the rules:

- Clearly, the initial state of the application must satisfy the global invariant, this is checked by conditions SafeInit and ConcInit.

The rest of the rules perform a kind of inductive reasoning. Assuming that we start in a state that satisfies the global invariant, we check that any state update preserves the validity of said invariant. Importantly, this reasoning is not circular, since the initial state is known by the rule above to be safe.[5]

- Condition SafeOp checks that each of the operations, when executed starting in a state satisfying its precondition and the invariant, is safe. Notice that we require that

[5]Indeed, the proof of soundness of program logics such as Rely/Guarantee are typically inductive arguments of this nature.

the precondition of the operation be satisfied in the starting state. This is the core of the inductive argument alluded to above, all operations – which as we mentioned in Section 2.1 execute atomically w.r.t. concurrency – preserve the data invariant Inv_{data}.

Other than the execution of operations, the other source of local state changes is the execution of the merge function in a replica. It is not true in general that for any two given states of an application, the merge should compute a safe state. In particular, it could be the case that the merge function needs a precondition that is stronger than the conjunction of the invariants in the two states to be merged. The following rules deal with these cases.

- We require the merge function to be annotated with a precondition strong enough to guarantee that `merge` will result in a safe state. Generally, this precondition can be obtained by calculating the weakest precondition [15] of `merge` w.r.t. the desired invariant. Since `merge` requires two states as input, the precondition of merge has two states. We can then verify that merging two states is safe. This is the purpose of rule SafeMerge.

As per the program model of Section 2.1, any two replicas can exchange their states at any given point of time and trigger the execution of a merge operation. Thus, it must be the case that the precondition of the merge function is enabled at all times between any two replica local states. Since merge is the only point where a local replica can observe the result of concurrent operations in other replicas, we call this a *concurrency invariant* (Inv_{conc}). In other words: the *concurrency invariant is part of the global invariant* of the application. This is the main insight that allows us to reduce the proof of the distributed application to checking that both the invariant Inv_{data} and the concurrency invariant Inv_{conv} are global invariants. In particular, the latter implies the former, but for exposition purposes we shall preserve the invariant Inv_{data} in the rules.

- Just as we did with the operations above, we now need to check that whenever we have a pair of states that satisfy the concurrency invariant, if one of these states changes, the resulting pair still satisfies the concurrency invariant. This is exactly the purpose of rule ConcOp in the case where the state change originates from an operation execution in one of the replicas of the pair. This rule is similar to rule SafeOp above, where the invariant Inv has been replaced by Inv_{conc}, and consequently we have a pair of states.

- Finally, as we did with rule SafeMerge, we need to check the case where one of the states of a pair of states satisfying Inv_{conc} is updated because of yet another merge happening (w.r.t. yet another replica) in one of these states. This is the purpose of rule ConcMerge which is similar to rule SafeMerge, with Inv_{data} replaced for Inv_{conc}.

As anticipated at the beginning of this section, the reasoning about the concurrency is performed in a completely local manner, by carefully choosing the verification conditions,

and it avoids the stability blow-up commonly found in concurrent program logics. The program model, and the verification conditions allow us to effectively reduce the problem of verifying safety of an asynchronous concurrent distributed system, to the modular verification of the global invariant ($\mathsf{Inv}_{data} \wedge \mathsf{Inv}_{conc}$) as pre and post conditions of all operations and merge.

Proposition 2.3.1 (Soundness) *The proof rules in equations* SafeInit — ConcMerge *guarantee that the implementation is safe.*

To conduct an inductive proof of this lemma we need to strengthen the argument to include the set of observed states as given by the semantics of Figure 2.2.

Lemma 2.3.2 (Strengthening of Soundness) *Assuming that the equations* SafeInit — ConcMerge *hold for an implementation of a replicated application with initial state* Ω_i*. For any state* $(\Omega, \mathtt{S})$ *reachable from* $(\Omega_i, \{\sigma_i\})$*, that is* $(\Omega_i, \{\sigma_i\}) \xrightarrow{*} (\Omega, \mathtt{S})$*, we have that:*

1. *for all states* $\sigma, \sigma' \in \mathtt{S}$*,* $(\sigma, \sigma') \vDash \mathsf{Inv}_{conc}$*, and*
2. *for any state* $\sigma \in \mathtt{S}$*,* $\sigma \vDash \mathsf{Inv}_{data}$*.*

Corollary 2.3.2.1 *The soundness proposition (2.3.1) is a direct consequence of Lemma 2.3.2.*

We remark at this point that there are numerous program logic approaches to proving invariants of shared-memory concurrent programs, with Rely/Guarantee [16] and concurrent separation logic [17] underlying many of them. While these approaches could be adapted to our use case (propagating-state distributed systems), this adaptation is not evident. As an indication of this complexity: one would have to predicate about the different states of the different replicas, restate the invariant to talk about these different versions of the state, encode the non-deterministic behaviour of the propagation layer, etc. Instead, we argue that our specialized rules are much simpler, allowing for a purely sequential and modular verification that we can mechanise and automate. This reduction in complexity is the main theoretical contribution of this work.

2.3.2 Applying the proof rule

Let us apply the proof methodology to the auction application. Its data invariant, Inv_{data}, is the following conjunction:

1. Only an `ACTIVE` auction can receive bids, and
2. the highest placed bid wins when the auction is `CLOSED`.

Computing the weakest precondition of each update operation, for this invariant is obvious. For instance, as discussed earlier, `close_auction(w)` gets precondition `is_highest(Bids, w)`, because of Invariant Item 2 above. The naïve specification of the auction object that satisfies the data invariant is shown in Specification 2.1.

Specification 2.1 Naïve Auction object

State: $(status, winner, Bids)$ $\qquad$ $\texttt{INVALID} < \texttt{ACTIVE} < \texttt{CLOSED}$

Invariant:
$$\begin{aligned}
&\forall b \in Bids \,\centerdot\, b.placed \implies status \geq \texttt{ACTIVE} \wedge b.amount > 0 \\
&\wedge status \leq \texttt{ACTIVE} \implies winner = \bot \\
&\wedge status = \texttt{CLOSED} \implies winner \in Bids \wedge winner.placed \\
&\qquad\qquad \wedge is_highest(Bids, winner)
\end{aligned}$$

Comparison function:
$$\begin{aligned}
&status_\sigma \geq status_{\sigma'} \wedge (winner_\sigma \neq \bot \vee winner_{\sigma'} = \bot) \\
&\wedge (\forall b \in Bids_\sigma \cup Bids_{\sigma'} \,\centerdot\, b_\sigma.placed \ \vee \ \neg b_{\sigma'}.placed)
\end{aligned}$$

$\texttt{INVALID} < \texttt{ACTIVE} < \texttt{CLOSED}$

Merge(σ, σ'):

$$\begin{aligned}
\{Pre_{merge} \triangleq\ & (winner_\sigma = winner_{\sigma'} \vee winner_\sigma = \bot \vee winner_{\sigma'} = \bot) \\
& \wedge \forall b \in Bids_\sigma \,\centerdot\, b_\sigma.amount = b_{\sigma'}.amount \\
& \wedge status_\sigma = \texttt{CLOSED} \implies is_highest(Bids_\sigma, winner_\sigma) \\
& \qquad\qquad \wedge is_highest(Bids_{\sigma'}, winner_\sigma) \\
& \wedge status_{\sigma'} = \texttt{CLOSED} \implies is_highest(Bids_\sigma, winner_{\sigma'}) \\
& \qquad\qquad \wedge is_highest(Bids_{\sigma'}, winner_{\sigma'}) \ \}
\end{aligned}$$

$$\begin{aligned}
& status_{\sigma_{new}} = max(status_\sigma, status_{\sigma'}) \\
& winner_{\sigma_{new}} = (winner_{\sigma'} \neq \bot) \ ? \ winner_{\sigma'} \ : \ winner_\sigma \\
& \forall b \in Bids_\sigma \cup Bids_{\sigma'} \,\centerdot\, (b_{\sigma_{new}}.placed = b_\sigma.placed \vee b_{\sigma'}.placed) \\
& \qquad \wedge (b_{\sigma_{new}}.amount = (b_{\sigma'}.amount > 0) \ ? \ b_{\sigma'}.amount \ : \ b_\sigma.amount)
\end{aligned}$$

StartAuction():

$$\begin{aligned}
& \{Pre_{startauction} \triangleq status = \texttt{INVALID} \wedge winner = \bot\} \\
& \quad status = \texttt{ACTIVE}
\end{aligned}$$

PlaceBid(bid):

$$\begin{aligned}
& \{Pre_{placebid} \triangleq bid \notin Bids \wedge status = \texttt{ACTIVE} \wedge winner = \bot\} \\
& \quad Bids = Bids \cup bid
\end{aligned}$$

CloseAuction(w):

$$\begin{aligned}
& \{Pre_{closeauction} \triangleq status = \texttt{ACTIVE} \wedge \exists b \in Bids \,\centerdot\, b.placed \wedge b.amount > 0 \\
& \qquad \wedge winner = \bot \wedge is_highest(Bids, w)\} \\
& \quad status = \texttt{CLOSED} \\
& \quad winner = w
\end{aligned}$$

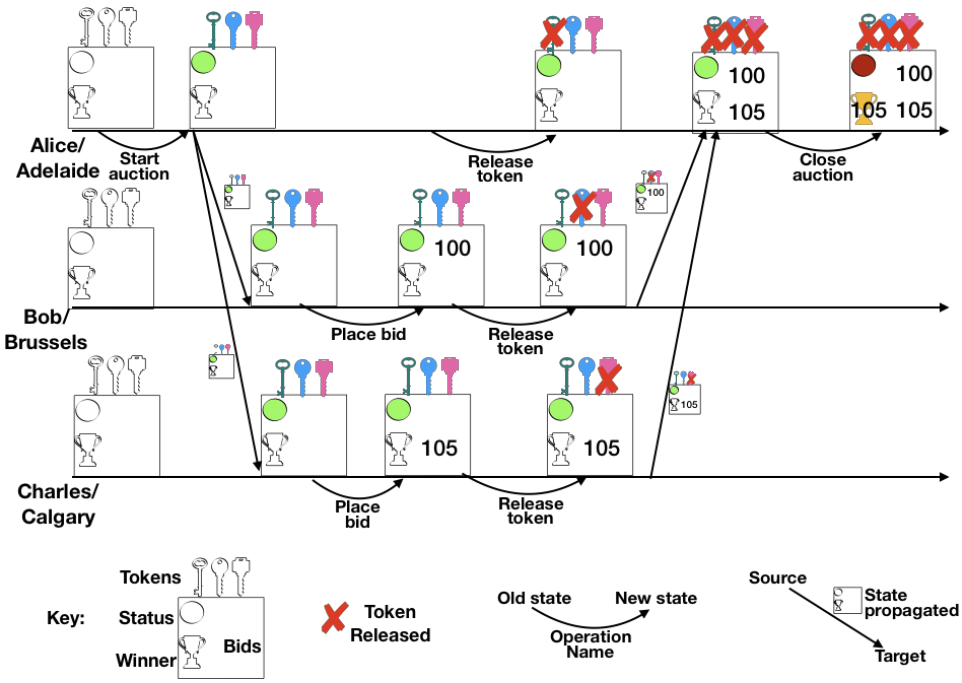


Figure 2.7: Evolution of state in an auction application with concurrency control

Despite local updates to each replica respecting the invariant Inv_{data}, Figure 1.1 showed that it is susceptible of being violated by merging. This is the case if Bob's \$100 bid in Brussels wins, even though Charles concurrently placed a \$105 bid in Calgary; this occurred because `status` became `CLOSED` in Brussels while still `ACTIVE` in Calgary. The weakest precondition of merge for safety expresses that, if `status` in either state is `CLOSED`, the winner should be the bid with the highest amount in both the states. This merge precondition, now called the concurrency invariant, strengthens the global invariant to be safe in concurrent executions. Specification 2.1 shows the concurrent invariant as the precondition of merge, Pre_{merge}.

Let us now consider how this strengthening impacts the local update operations. Since starting the auction doesn't modify any bids, the operation trivially preserves it. Placing a bid might violate Inv_{conc} if the auction is concurrently closed in some other replica; conversely, closing the auction could also violate Inv_{conc}, if a higher bid is concurrently placed in a remote replica. Thus, the auction application is safe when executed sequentially, but it is unsafe when updates are concurrent. This indicates the concurrent specification has a bug, which we now proceed to fix.

2.3.3 Coordination for Invariant Preservation

As we discussed earlier, the preconditions of operations and merge are strengthened in order to be sequentially safe. An application must also preserve the concurrency invariant in order to ensure concurrent safety. Violating this indicates the presence of a concurrency bug in the specification. In that case, the operations that fail to preserve the concurrency invariant might need to coordinate. The developer adds the required concurrency control mechanisms as part of the state in our model. The modified state is now composed of the

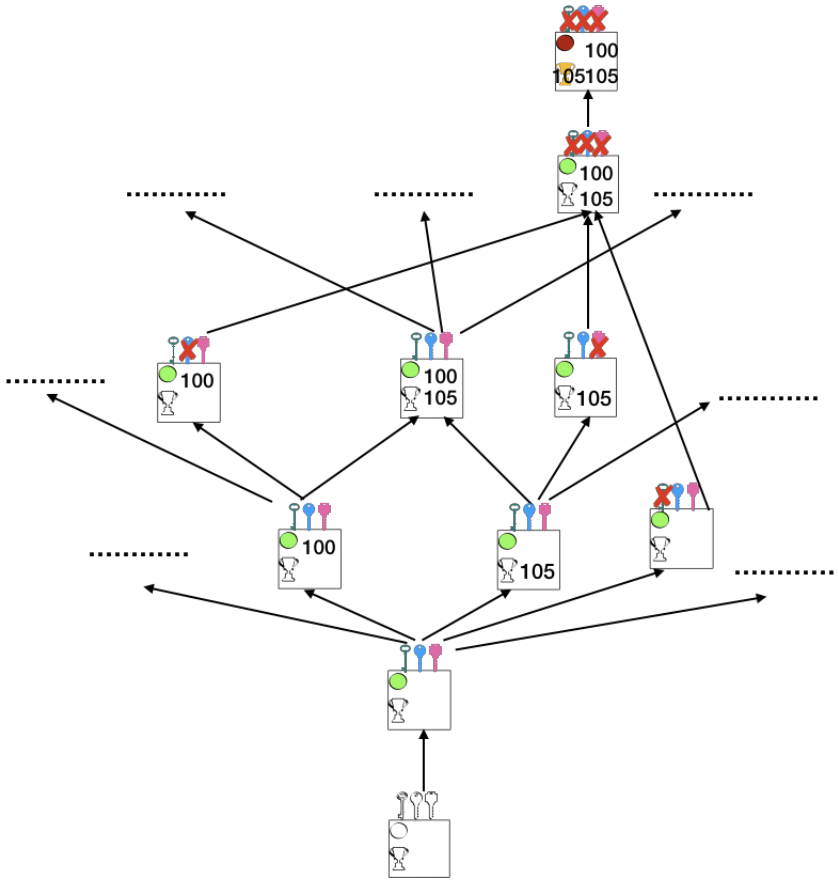


Figure 2.8: Safe in concurrent executions

state and the coordination mechanism.

Recall that in the auction example, placing bids and closing the auction did not preserve the precondition of merge. This requires strengthening the specification by adding a coordination mechanism to restrict these operations. We can enforce them to be strictly sequential, thereby avoiding any concurrency at all. But this will affect the availability of the application. In particular, it should be possible to place bids in parallel.

A concurrency control can be better designed with the workload characteristics in mind. For this particular use case, we know that placing bids is a much more frequent operation than closing an auction. Concurrent placing of bids is safe, whereas concurrency between place bid and close auction is not. This situation is similar to a readers-writer lock. We distribute tokens to each replica. As long as a replica has a token, it can place bids. Closing the auction requires recalling the tokens from all replicas. This ensures that there are no bids placed concurrently while closing auction and thus a winner can be declared, respecting the invariant. The addition of this concurrency control also updates the Inv_{conc}. Clearly, all operations must respect this modification for the specification to be considered safe.

Note that the token model described here restricts availability in order to ensure safety. Adding efficient coordination is not a problem to be solved only with application specification in hand, it rather requires the knowledge of the application dynamics such as the workload characteristics and is part of our work described in Part III.

Specification 2.2 Safe Distributed Auction

State: $(status, winner, Bids, Tokens)$ $\quad$ $\texttt{INVALID} < \texttt{ACTIVE} < \texttt{CLOSED}$

Invariant:
$$\begin{aligned}
&\forall b \in Bids \,.\, b.placed \implies status \geq \texttt{ACTIVE} \wedge b.amount > 0 \\
&\wedge status \leq \texttt{ACTIVE} \implies winner = \bot \\
&\wedge status = \texttt{CLOSED} \implies winner \in Bids \wedge winner.placed \\
&\qquad \wedge is_highest(Bids, winner) \\
&\qquad \wedge \forall r \in Reps \,.\, \neg Tokens[r]
\end{aligned}$$

Comparison function:
$$\begin{aligned}
&status_\sigma \geq status_{\sigma'} \wedge (winner_\sigma \neq \bot \vee winner_{\sigma'} = \bot) \\
&\wedge (\forall b \in Bids_\sigma \,.\, b_\sigma.placed \ \vee \ \neg b_{\sigma'}.placed) \\
&\wedge \forall r \in Reps \,.\, \neg Tokens_\sigma[r] \vee Tokens_{\sigma'}[r]
\end{aligned}$$

Merge(σ, σ'):
$$\begin{aligned}
\{Pre_{merge} \triangleq\ & (winner_\sigma = winner_{\sigma'} \vee winner_\sigma = \bot \vee winner_{\sigma'} = \bot) \\
&\wedge \forall b \in Bids_\sigma \,.\, b_\sigma.amount = b_{\sigma'}.amount \\
&\wedge status_\sigma = \texttt{CLOSED} \implies is_highest(Bids_\sigma, winner_\sigma) \\
&\qquad \wedge is_highest(Bids_{\sigma'}, winner_\sigma) \\
&\wedge status_{\sigma'} = \texttt{CLOSED} \implies is_highest(Bids_\sigma, winner_{\sigma'}) \\
&\qquad \wedge is_highest(Bids_{\sigma'}, winner_{\sigma'}) \\
&\wedge Tokens_\sigma[me] \implies Tokens_{\sigma'}[me] \\
&\wedge \forall b \in Bids_\sigma \,.\, \forall r \in Reps \,.\, \neg Tokens_\sigma[r] \\
&\qquad \wedge \neg b_\sigma.placed \implies \neg b_{\sigma'}.placed \\
&\wedge (\forall r \in Reps \,.\, \neg Tokens_\sigma[r] \wedge b \notin Bids_\sigma) \implies b \notin Bids_{\sigma'} \\
&\forall r \in Reps \,.\, \neg Tokens_\sigma[r] \implies winner_{\sigma'} = winner_\sigma \vee winner_{\sigma'} = \bot \\
&\exists r \in Reps \,.\, Tokens_\sigma[r] \implies winner_{\sigma'} = \bot \wedge winner_\sigma = \bot \ \}
\end{aligned}$$
$$\begin{aligned}
&status_\sigma = max(status_\sigma, status_{\sigma'}) \\
&winner_\sigma = (winner_{\sigma'} \neq \bot) \ ? \ winner_{\sigma'} \ : \ winner_\sigma \\
&\forall b \in Bids_\sigma \cup Bids_{\sigma'} \,.\, (b_\sigma.placed = b_\sigma.placed \vee b_{\sigma'}.placed) \\
&\qquad \wedge (b_\sigma.amount = (b_{\sigma'}.amount > 0) \ ? \ b_{\sigma'}.amount \ : \ b_\sigma.amount) \\
&\forall r \in Reps \,.\, Tokens_\sigma[r] = Tokens_\sigma[r] \wedge Tokens_{\sigma'}[r]
\end{aligned}$$

StartAuction():
$$\begin{aligned}
&\{Pre_{startauction} \triangleq status = \texttt{INVALID} \wedge winner = \bot \wedge \forall r \in Reps \,.\, Tokens[r] \ \} \\
&\quad status = \texttt{ACTIVE}
\end{aligned}$$

PlaceBid(bid):
$$\begin{aligned}
&\{Pre_{placebid} \triangleq bid \notin Bids \wedge status = \texttt{ACTIVE} \wedge winner = \bot \wedge Tokens[me] \ \} \\
&\quad Bids = Bids \cup bid
\end{aligned}$$

CloseAuction(w):
$$\begin{aligned}
&\{Pre_{closeauction} \triangleq status = \texttt{ACTIVE} \wedge winner = \bot \wedge \exists b \in Bids \,.\, b.placed \wedge b.amount > 0 \\
&\qquad \wedge is_highest(Bids, w) \wedge \forall r \in Reps \,.\, \neg Tokens[r] \ \} \\
&\quad status = \texttt{CLOSED} \\
&\quad winner = w
\end{aligned}$$

Figure 2.7 shows the evolution of the modified auction application with concurrency control. In the figure, a token is represented by the icon of a key. When a replica wants to close the auction, it can request tokens from other replicas. We indicate that a replica releases its token by a cross mark on its key. This coordination mechanism makes sure that the application is safe during concurrent executions as well. Figure 2.8 shows the semi-lattice formed by the updated specification.

The extended specification of the auction example including the concurrency control is listed in Specification 2.2 and proven correct. The shaded lines in blue indicate the effect of adding concurrency control to the state. Note that the keyword *me* indicates the current replica. We model the tokens as an array of boolean values, with one entry per replica. This addition to the state modifies the invariant and subsequently the preconditions of all operations and merge. The proof rule when applied on this modified specification verifies that the specification is safe from concurrency bugs.

To summarize, all updates (operations and merge) have to respect the global invariant ($\mathsf{Inv}_{data} \wedge \mathsf{Inv}_{conc}$). If an update violates Inv_{data}, the developer must strengthen its precondition. If an update violates Inv_{conc}, the developer must restrict concurrency through coordination mechanisms.

Chapter 3

Use cases

This chapter presents three representative examples of distributed applications with different consistency requirements. The consent object is an example of a coordination-free design, illustrating a safe object requiring only eventual consistency. The distributed lock shows a design that maintains a total order, illustrating strong consistency. The courseware example shows a mix of concurrent operations and operations with restricted concurrency. This example illustrates applications that might require some coordination to ensure safety.

For each case study, we give an overview of the operational semantics informally. We then discuss how the design preserves the safety conditions discussed in Section 2.3. We also provide pseudocode for better comprehension.

3.1 Distributed Barrier

In some distributed applications, all replicas must reach a single state for an operation to proceed. We consider the specification of a barrier object with a fixed number of replicas.

The specification of the consent object is shown in Specification 3.1. The state consists of a boolean flag, $flag$, indicating that all replicas have voted, and a boolean array, $Votes$, indicating the votes from replicas. Each replica votes by setting its dedicated entry in the boolean array. A replica cannot withdraw its vote. A replica sets $flag$ when it sees all entries of the boolean array set. The merge function is the disjunction of the individual components.

The consistency between the values of $flag$ and $Votes$ is ensured by the invariant. The invariant requires that if the flag is set, then all the replicas have voted.

We can observe that *merge* and *vote* operations maintain the invariant at all times whereas *agree* needs an extra precondition to ensure that all the replicas have voted before setting the consent flag.

Since *merge* ensures safety without any additional precondition, the object is trivially safe under concurrent executions.

Specification 3.1 Distributed Barrier

State: $(Votes, flag)$
Invariant: $flag \implies \forall r \in Reps \,.\, Votes[r]$
Comparison function: $flag_\sigma \vee (\neg flag_{\sigma'} \;\wedge\; \forall r \in Reps \,.\, Votes_\sigma[r] \vee \neg Votes_{\sigma'}[r])$
Merge(σ, σ'):
$\{Pre_{merge} \triangleq \texttt{True}\}$
$\forall r \in Reps \,.\, Votes_\sigma[r] = Votes_\sigma[r] \vee Votes_{\sigma'}[r]$
$flag_\sigma = flag_\sigma \vee flag_{\sigma'}$
Vote():
$\{Pre_{vote} \triangleq \texttt{True}\}$
$Votes[me] = \texttt{True}$
Agree():
$\{Pre_{agree} \triangleq \forall r \in Reps \,.\; Votes[r]\}$
$flag = \texttt{True}$

Specification 3.2 Replicated Lock

State: $Lock \times timestamp$
Invariant: $\exists r \in Reps \,.\; Lock[r] \wedge \;\forall r' \in Reps \,.\; (Lock[r] \wedge Lock[r']) \implies r = r'$
Comparison function: $timestamp_\sigma > timestamp_{\sigma'} \vee (timestamp_\sigma = timestamp_{\sigma'} \wedge \;\forall r \in Reps \,.\, Lock_\sigma[r] = Lock_{\sigma'}[r])$
Merge(σ, σ'):
$\{Pre_{merge} \triangleq (timestamp_\sigma = timestamp_{\sigma'} \implies \forall r \in Reps \,.\, Lock_\sigma[r] = Lock_{\sigma'}[r])$
$\wedge\; (Lock_\sigma.me \implies timestamp_\sigma \geq timestamp_{\sigma'})\}$
$timestamp_\sigma = max(timestamp_\sigma, timestamp_{\sigma'})$
$\forall r \in Reps \,.\, Lock_\sigma[r] = (timestamp_\sigma > timestamp_{\sigma'})\ ?\ Lock_\sigma[r]\ :\ Lock_{\sigma'}[r]$
Transfer(to):
$\{Pre_{transfer} \triangleq Lock[me]\}$
$timestamp = timestamp + 1$
$Lock[me] = \texttt{False}$
$Lock[to] = \texttt{True}$

3.2 Replicated lock

We now discuss a replicated lock object that ensures mutual exclusion. We use an array of boolean values, one entry per replica, to model the lock. If a replica owns the lock, the corresponding array entry is set to true. The lock is transferred to any other replica by using the $transfer$ function. The full specification is shown in Specification 3.2.

We need to ensure that the lock is owned by exactly one replica at any given point in time, the mutual exclusion property. This is the invariant. For simplicity, we are not considering failures. In order to preserve safety, we need to enforce a precondition on the transfer operation such that the operation can only transfer the ownership of its origin replica. For state inflation, a timestamp associated with the lock is incremented during each transfer.

A merge of two states of this distributed lock will preserve the state with the highest timestamp. Assuming timestamps are unique, if the timestamps of the two states are

equal, their corresponding boolean arrays are also equal. Also, the state of the replica that owns the lock has the highest timestamp. The conjunction of these two restrictions which form the precondition of merge, $\mathsf{Pre}_{\mathtt{merge}}$, which is by definition the concurrency invariant, Inv_{conc}.

Consider the case of three replicas r_1, r_2 and r_3 sharing a distributed lock. Assume that initially replica r_1 owns the lock. Replicas r_2 and r_3 concurrently place a request for the lock. The current owner r_1, has to make a decision on the priority of the requests based on some unspecified business logic. Assume that r_1 transfers the lock to r_3. Since r_1 no longer has the lock, it cannot issue any further transfer operations. We see clearly that the transfer operation is safe due to the precondition that only the replica that owns the lock can transfer it. In the new state, r_3 is the only replica that can perform a transfer operation. We also note that this prevents any concurrent transfer operations. This guarantees mutual exclusion, and hence ensures safety in a concurrent execution environment.

Observe that due to the precondition of the $transfer$ operation, concurrent operations do not happen. The states progress through a total order, ordered by the timestamp. The transfer function increases the timestamp and the merge function preserves the highest timestamp.

3.3 Courseware

We now study an application that allows students to register and enrol in courses. The state consists of a set of students, a set of courses, and enrolments of students for different courses. Students can register and deregister, courses can be created and deleted, and a student can enrol for a course. The invariant requires enrolled students to be registered and courses to be created.

The set of students consists of a 2P-set [18] - to track registrations and deregistrations. Similarly, courses are also represented as 2P-set - tracking creations and deletions. Registration or creation monotonically adds the student or course respectively to the registered sets and deregistration or deletion monotonically adds them to the unregistered sets. The semantics currently doesn't support re-registration. Enrolment adds the (student, course) pair to the G-set [18]. Currently, we do not consider cancelling an enrolment. Merging two states takes the union of the sets.

Let us consider safety. The operations to register a student and create a course are safe without any restrictions. Therefore they do not need any precondition. The remaining three operations might violate the invariant in some cases. This leads to strengthening their preconditions. The precondition of the operation for deregistering a student and deleting a course requires no existing enrolments for them. For enrolment, both the student and the course should be registered/created and not unregistered/deleted.

Merge also requires strengthening of its precondition. It requires the set of enrolled students and courses to be registered and not unregistered in all the remote states as well. This is the concurrent invariant (Inv_{conc}) for this object.

Running this specification through our tool which we describe in Chapter 4 reveals concurrency issues for deregistering a student, deleting a course and enrolment. This

means that we need to add concurrency control to the state.

For this use case, we know that enrolling will be more frequent than deregistering a student or deleting a course. So, we model a concurrency control mechanism as in the case of the auction object discussed earlier. We assign a token to each replica for each student and course, called a student token and course token respectively. A replica will have a set of student tokens indicating the registered students and course tokens indicating the created courses. In order to deregister a student or delete a course, all replicas must have released their tokens for that particular student/course. Enrol operations can progress as long as the student token and course token are available at the local replica for the student and course for that particular enrolment.

This concurrency control mechanism now forms part of the state. The preconditions of operations and merge are recomputed and the concurrency invariant is updated. The edited specification passes all checks and is deemed safe.

3.3.1 Pseudocode of courseware

This section explains the pseudocode of the courseware application. The state consists of a set of students, *Students*, a set of courses, *Courses*, and enrolments of students for different courses, *Rolls*. Students can register and deregister, courses can be created and deleted, and a student can enrol for a course. The invariant requires enrolled students and courses to be registered and created respectively.

The set of students and courses are 2P-sets, consisting two sets - A to track registrations or creations and another R to track deregistrations or deletions. Enrolment adds the student-course pair to the set *Rolls*. Merging two states takes the union of the sets. For simplicity, we abstract the specification of a 2P-set into an ordinary set with add and remove operations.

Let us consider the safety of each operation. The operations *deregister_student*, *delete_course* and *enrol*, and *merge* have additional preconditions to be safe in sequential execution. The operations *register_student* and *create_course* are safe without any restrictions. The precondition of merge requires the set of enrolled students and courses to be registered and not unregistered in all the remote states as well. This is the concurrent invariant, Inv_{conc}, for this object.

As we discussed, our tool reports concurrency issues for *deregister_student*, *delete_course* and *enrol* operations, indicating the need of concurrency control.

For this use case, we know that enrolling will be more frequent than deregistering a student or deleting a course. So, we model a concurrency control mechanism as in the case of the auction object discussed earlier. As explained, we add student token, ST, and course token, CT, to the state. A replica will have a student token for each registered student and a course token for each created course. In order to deregister a student or delete a course, all replicas must release their tokens for that particular student/course. Enrol operations can progress as long as the student token and course token are available at the local replica for the student and course for that particular enrolment.

Specification 3.3 presents the specification of a safe courseware application. For read-

ability, we abstract the iteration over the elements of the set. For example, merging two 2P-sets of students actually has the following operations:

$$\forall s \in Students_{\sigma}[A] \cup Students_{\sigma'}[A] \bullet Students_{\sigma}[A][s] = Students_{\sigma}[s][A] \vee Students_{\sigma'}[s][A]$$
$$\wedge Students_{\sigma}[R][s] = Students_{\sigma}[s][R] \vee Students_{\sigma'}[s][R]$$

We replace it in a compact form:

$$Students_{\sigma} = Students_{\sigma} \vee Students_{\sigma'}$$

The shaded region indicate the added concurrency control. The precondition of merge is strengthened by a clause that says if there are no tokens available for a student or a course, that student and course won't be part of any new enrolment. This prevents enroling deregistered students to deleted courses.

Specification 3.3 Concurrently safe Courseware object

State: $Students \times Courses \times Rolls \times ST \times CT$

Invariant: $\forall e \in Rolls \,.\, e \implies e.student \in Students \wedge e.course \in Courses$

Comparison function: $(Students_\sigma \vee Students_{\sigma'}) \wedge (Courses_\sigma \vee Courses_{\sigma'})$
$\wedge(Rolls_\sigma \vee Rolls_{\sigma'}) \wedge(ST_\sigma \vee ST_{\sigma'}) \wedge (CT_\sigma \vee CT_{\sigma'})$

Merge(σ, σ'):

$$
\begin{aligned}
\{Pre_{merge} \triangleq\ & \forall e \in Rolls_\sigma \,.\, e.student \in Students_{\sigma'} \wedge e.course \in Students_{\sigma'} \\
& \wedge\forall e \in Rolls_{\sigma'} \,.\, e.student \in Students_\sigma \wedge e.course \in Students_\sigma \\
& \wedge\forall st \in ST_\sigma \,.\, st.student \in Students_\sigma \vee st.student \notin Students_{\sigma'} \\
& \qquad \wedge(\exists e \in Rolls_\sigma \,.\, e.student = st.student \\
& \qquad\quad \vee\nexists e \in Rolls_{\sigma'} \,.\, e.student = st.student) \\
& \wedge\forall st \in ST_{\sigma'} \,.\, st.student \in Students_{\sigma'} \vee st.student \notin Students_\sigma \\
& \qquad \wedge(\exists e \in Rolls_{\sigma'} \,.\, e.student = st.student \\
& \qquad\quad \vee\nexists e \in Rolls_\sigma \,.\, e.student = st.student) \\
& \wedge\forall ct \in CT_\sigma \,.\, ct.course \in Courses_\sigma \vee ct.course \notin Courses_{\sigma'} \\
& \qquad \wedge(\exists e \in Rolls_\sigma \,.\, e.course = ct.course \\
& \qquad\quad \vee\nexists e \in Rolls_{\sigma'} \,.\, e.course = ct.course) \\
& \wedge\forall ct \in CT_{\sigma'} \,.\, ct.course \in Courses_{\sigma'} \vee ct.course \notin Courses_\sigma \\
& \qquad \wedge(\exists e \in Rolls_{\sigma'} \,.\, e.course = ct.course \\
& \qquad\quad \vee\nexists e \in Rolls_\sigma \,.\, e.course = ct.course) \\
& \wedge\forall st \in ST_\sigma \,.\, st.replica = me \wedge st.student \notin Students_\sigma \\
& \qquad \implies st.student \notin Students_{\sigma'} \\
& \wedge\forall ct \in CT_\sigma \,.\, ct.replica = me \wedge ct.course \notin Courses_\sigma \\
& \qquad \implies ct.course \notin Courses_{\sigma'} \ \}
\end{aligned}
$$

$$
\begin{aligned}
& Students_\sigma = Students_\sigma \vee Students_{\sigma'} \\
& Courses_\sigma = Courses_\sigma \vee Courses_{\sigma'} \\
& Rolls_\sigma = Rolls_\sigma \vee Rolls_{\sigma'} \\
& ST_\sigma = ST_\sigma \wedge ST_{\sigma'} \\
& CT_\sigma = CT_\sigma \wedge CT_{\sigma'}
\end{aligned}
$$

RegisterStudent(student):

$\{Pre_{registerstudent} \triangleq \texttt{True}\}$

$Students = Students \cup \{student\}$

DeregisterStudent(student):

$$
\begin{aligned}
\{Pre_{deregstudent} \triangleq\ & \nexists e \in Rolls \,.\, e.student = student \\
& \wedge\nexists st \in ST \,.\, st.student = student \ \}
\end{aligned}
$$

$Students = Students \setminus \{student\}$

CreateCourse(course):

$\{Pre_{createcourse} \triangleq \texttt{True}\}$

$Courses = Courses \cup \{course\}$

DeleteCourse(course):

$\{Pre_{deletecourse} \triangleq \nexists e \in Rolls \,.\, e.course = course \wedge\nexists ct \in CT \,.\, ct.course = course \ \}$

$Courses = Courses \setminus \{course\}$

Enrol(student, course):

$$
\begin{aligned}
\{Pre_{enrol} \triangleq\ & student \in Students \wedge course \in Courses \\
& \wedge\exists st \in ST \,.\, st.student = student \wedge \exists ct \in CT \,.\, ct.course = course \ \}
\end{aligned}
$$

$Rolls = Rolls \cup \{(course, student)\}$

Chapter 4

Automation

In this chapter, we present a tool to automate the approach discussed in the previous chapter. Our tool, called *Soteria*, is based on the Boogie [19] verification framework. The input to Soteria is a specification of the object written as Boogie procedures, augmented with some annotations, in order to check the properties described in Section 2.3.

4.1 Specifying a distributed application in Soteria

Let us now consider how a distributed object is specified in Soteria.

State

The user of the tool provides a declaration of the local state using the global variables in Boogie. The data types can be either built-in or user defined.

Comparison function

The user provides a comparison function to determine the partial order on states. The comparison function returns true when the first state is greater than or equal to the other state. It is encoded as a function in Boogie. The tool uses this comparison function as a basis to check the inflation and lattice conditions given in Figure 2.4. The keyword `@gteq` marks the comparison function.

Operations

The user provides the implementation of the operations of the object in Boogie along with its precondition $\mathsf{Pre}_{\mathsf{op}}$. In general, operations are encoded as Boogie procedures. Alternatively, we could just require only a post-condition describing how the state transitions from the precondition to the post-condition. Notice that since in our program model operations are atomic, this is an unambiguous encoding of the operations.

A few things are important in this code. The specification declares operations that can modify the global variables using the `modifies` clause. A precondition is specified in a `requires` clause, and the postcondition in an `ensures` clause. The semantics of multiple `requires` and `ensures` clauses is conjunction.

Merge function

The tool requires the special `merge` operation to be distinguished from other operations, annotated `@merge`. As previously mentioned, the precondition of `merge` can be obtained by calculating the weakest precondition to ensure safety. The current version of Soteria does not perform this step automatically, but relies on the user to provide the preconditions. Notice that Soteria will consider this as the concurrency invariant (Inv_{conc}).

In Section 2.1 we mentioned that the `merge` procedure takes two states as arguments, in the specification input to Soteria, the procedure merge takes only one state as the argument. The state represented in the argument is the incoming state and it is merged with the local state (represented by the global variables in the specification).

Invariant

The user provides the invariant to be verified by the tool. This invariant is simply provided as a Boogie assertion over the state of the object annotated with the keyword `@invariant`.

Additional information

The components above are required for Soteria's safety checks. In addition, Boogie often requires additional annotations to help it with verification. These include:

- User-defined data types
- Constants, to declare special objects such as the origin replica "`me`", or to bound the quantifiers
- We sometimes make recourse to inductively-defined functions over aggregate data structures, for instance, to obtain the maximum in a set of values. To enable the SMT solver to use them, we axiomatize their semantics. This is particularly important for list comprehensions and array operations. In this, we follow the approach of Leino et al.[20].
- When iterating over lists, arrays or matrices, we must provide loop invariants in order to verify them by Boogie.

4.2 Verification passes

From the input specification Soteria generates a set of verification conditions, which it passes to Boogie, to be proved by leveraging SMT solvers.

This verification process comprises multiple stages, as follows:

4.2.1 Syntax check

The first step validates that the specification provided respects Boogie syntax, ignoring the Soteria-specific annotations and calls Boogie to validate that the types are correct and that the pre/post conditions provided are sound.

Then it checks that the user has provided the required annotations. Specifically, it checks the function signatures marked by `@gteq` and `@invariant` and the procedure marked by `@merge`.

4.2.2 Convergence check

The convergence stage checks the convergence of the specification. Specifically, it checks whether the specification respects Strong Eventual Consistency, i.e., that any two replicas that received the same set of updates have the same state. To guarantee SEC, objects must have the following properties[2, 4, 13]:

- The state space is equipped with an ordering operator, comparing states.
- Each individual operation is an inflation in the order. In a nutshell, the tool asks Boogie to prove the following Hoare-logic triple for every operation:

$$\begin{array}{l} \texttt{assume}\ \ \sigma \vDash \mathsf{Pre}_{\mathsf{op}} \\ \texttt{call}\ \ \sigma_{new} := \mathsf{op}(\sigma) \\ \texttt{assert}\ \ \sigma_{new} \geq \sigma \end{array}$$

- The ordering forms a join-semilattice, i.e., for any two states in the lattice, there exists a state in the lattice that is their least upper bound.
- The `merge` operation, composing states from two replicas, computes their least-upper-bound. The verification condition discharged is shown below (the primed state indicates the incoming state from a remote replica):

$$\begin{array}{l} \texttt{assume}\ \ (\sigma, \sigma') \vDash \mathsf{Pre}_{\texttt{merge}} \\ \texttt{call}\ \ \sigma_{new} := \texttt{merge}(\sigma, \sigma') \\ \texttt{assert}\ \ \sigma_{new} \geq \sigma \wedge \sigma_{new} \geq \sigma' \\ \texttt{assert}\ \ \forall \sigma*, \sigma* \geq \sigma \wedge \sigma* \geq \sigma' \implies \sigma* \geq \sigma_{new} \end{array}$$

We present the conditions formally in Section 2.2.

An alternative is to make use of the CALM theorem [21]. This allows non-monotonic operations, but requires them to coordinate. However, our aim is to provide maximum possible availability with SEC. [1]

4.2.3 Safety check

This stage verifies the safety of the specification, as discussed in Section 2.3. This is subdivided into two sub-stages:

1. *Sequential safety:* Soteria checks whether each individual operation is safe. This corresponds to the conditions (SafeOp) and (SafeMerge) in Figure 2.6. The verification condition discharged by the tool to ensure sequential safety of operations is:

[1] Convergence of our running example is discussed in Subsection 2.2.1.

$$\texttt{assume}\ \ \sigma \vDash \mathsf{Pre}_{\mathsf{op}} \wedge \mathsf{Inv}$$
$$\texttt{call}\ \ \sigma_{new} := \mathsf{op}(\sigma)$$
$$\texttt{assert}\ \ \sigma_{new} \vDash \mathsf{Inv}$$

The special case of the `merge` function is verified with the following verification condition:

$$\texttt{assume}\ \ (\sigma, \sigma') \vDash \mathsf{Pre}_{\texttt{merge}} \wedge \sigma \vDash \mathsf{Inv} \wedge \sigma' \vDash \mathsf{Inv}$$
$$\texttt{call}\ \ \sigma_{new} := \texttt{merge}(\sigma, \sigma')$$
$$\texttt{assert}\ \ \sigma_{new} \vDash \mathsf{Inv}$$

In case of failure of the sequential safety check, the designer needs to strengthen the precondition of the operation (or merge) which was unsafe.

2. *Concurrent safety:* Here we check the precondition of merge, Inv_{conc}, is an invariant for every operation. This corresponds to the conditions (ConcOp) and (ConcMerge) in Figure 2.6. As shown in Section 2.3, this ensures safety during concurrent operation.

   The verification conditions are:

$$\texttt{assume}\ \ \sigma \vDash \mathsf{Pre}_{\mathsf{op}} \wedge \mathsf{Inv} \wedge (\sigma, \sigma') \vDash \mathsf{Inv}_{conc}$$
$$\texttt{call}\ \ \sigma_{new} := \mathsf{op}(\sigma)$$
$$\texttt{assert}\ \ (\sigma_{new}, \sigma') \vDash \mathsf{Inv}_{conc}$$

   to validate each operation `op`, and

$$\texttt{assume}\ \ (\sigma, \sigma') \vDash \mathsf{Inv}_{conc} \wedge \sigma \vDash \mathsf{Inv} \wedge \sigma' \vDash \mathsf{Inv}$$
$$\texttt{call}\ \ \sigma_{new} := \texttt{merge}(\sigma, \sigma')$$
$$\texttt{assert}\ \ (\sigma_{new}, \sigma) \vDash \mathsf{Inv}_{conc}$$

   to validate `merge`. If the concurrent safety check fails, the design of the distributed object needs a replicated concurrency control mechanism embedded as part of the state.

   Notice that while this check relates to the concurrent behaviour of the distributed object, the check itself is completely sequential; it does not require reasoning about operations performed by other processes.

Soteria performs each check by generating verification conditions and using the Boogie verification engine, which in turn uses the Z3 SMT solver. There are three possible outcomes for each verification condition:

1. The verification condition is proven.
2. Unable to prove the verification condition.
3. Time out or memory overflow.

The first outcome is the desired one. For a user, the second and third outcome basically means the same - the verification condition is not proven. When all checks are validated, Soteria reports that the specification is safe. For the verification conditions that failed due

Application	Number of methods	Time taken (s)			
		Syntax check	Convergence check	Safety check	Total
Consensus	2	2.594	5.847	11.693	20.141
Distributed lock	1	2.635	4.004	8.034	14.680
Courseware	5	2.803	12.161	24.213	39.184
Auction	4	2.895	10.885	21.672	35.458

Table 4.1: Time taken for analysing specification using Soteria

to incorrect or incomplete specification, Soteria produces counterexamples with the help of Boogie and Z3. This helps the developer identify issues with the specification and fix it.

Soteria uses an SMT solver, Z3, which is fully automated. As far as the proof system is concerned, no programmer involvement is required. We present the efficient generation of synchronization control considering the workload characteristics in Chapter 11. The tool and the specifications of the case studies discussed in Chapter 3 are available at Soteria [12].

4.3 Tool evaluation

All the applications we discussed in Chapter 3 are verified using Soteria. Table 4.1 provides the analysis time taken for each example. The time reported is the average time from five runs, on a 2.5 GHz Intel Core i7 processor and 16 GB 1600 MHz DDR3 memory.

Chapter 5

Related work

This chapter discusses the literature related to the verification of available distributed applications.

Several works have concentrated on the formalisation and specification of eventually consistent systems such as Burckhardt et al. [22], Burckhardt [23], Sivaramakrishnan et al. [24], to mention but a few.

Kaki et al. [25] present a programming framework equipped with a fully automated symbolic execution engine, named Q9, that detects invariant violations. They have specified a set of consistency levels and the tool suggests the consistency level required to maintain safety. The problem they are addressing is the same as Soteria, but consider a different system model wherein operations are propagated between replicas. Moreover, symbolic execution cannot provide the same level of guarantees as a full verification.

A number of works concentrate on the specification and correct implementation of replicated data types [26, 27]. Unlike these works, we assume the data type implementation is correct and focus on proving semantic properties that hold of a distributed object i.e., invariants of interest to the application.

Gotsman et al. [6] present a proof methodology for proving invariants of distributed objects. That work is supported by a tool called the CISE tool [7]. Similar to Soteria, the CISE tool performs the safety checks using an SMT solver as a backend. A more user-friendly tool was developed by Marcelino et al. [8], named the Correct Eventual Consistency(CEC) Tool. CEC is based on Boogie verification framework and also proposes sets of tokens that the developer might use. A token represents an abstract notion of concurrency control. Nair and Shapiro [28] improved the token generation by using the counterexamples generated by Boogie.

Hamsaz[9] extends CISE by lowering the causal consistency requirements and generating concurrency control protocols. Soteria only identifies the list of conflicting operations. Part III discusses the generation of concurrency control configurations based on distributed locks, and shows the impact of application workload on the choice of concurrency control configuration. It requires reasoning about concurrent behaviours.

CISE, CEC (and more generally the work Gotsman et al. [6]) and Hamsaz focus on the safety of operation-based objects. They assume that the underlying network ensures causal consistency. Importantly, their proof methodology requires reasoning about concur-

rent behaviours, reflected as stability conditions. Soteria doesn't require reasoning about concurrent behaviour, thanks to the concurrency invariant, Inv_{conc}. This is due to a fundamental difference in operation-based and state-based update propagation model — in the former, a replica observes remote updates through a series of operations, in the latter, remote updates are observed only during a merge operation.

CISE is not well adapted to reasoning about systems that propagate state. Conversely, Soteria is not well adapted to reason about objects that propagate operations. It is future work to combine use of both CISE and Soteria to prove properties depending on the implementation of the objects at hand and also to extend them to delta-based update propagation, where a delta of the state that changed is propagated, instead of the entire state.

As mentioned in Section 2.3, Bailis et al. [14] introduced the concept of I-confluence based on a similar system model to ours. I-confluence states that for an invariant to hold in a lattice-based state-propagating distributed application, the set of *reachable* valid (i.e. invariant preserving) states must be closed under operations and merge. This condition is similar to the ones presented in Figure 2.6. However, there is a fundamental difference: while Bailis et al. [14] recognises that one needs to consider only *reachable* states when checking that the merge operation satisfies the invariant, they do not provide means to identify these reachable states.

In other words, I-confluence [14] does not provide a program logic, but rather a meta-theoretical proof about lattice-based state-propagating systems. This is indeed a hard problem. In Soteria, we instead *over-approximate the set of reachable states* by ignoring whether the states are indeed reachable, but require that their merge satisfies the invariant. Notice that this is a sound approximation since it guarantees the invariant is satisfied, and we also verify that every operation preserves this condition as shown in Corollary 2.3.2.1. It is this abstraction step that makes the analysis performed by Soteria to be syntax-driven, automated, and machine-checked. This is captured in the concurrency invariant, Inv_{conc}, which is synthesized from the user provided invariant. How to obtain this invariant is understandably not addressed in Bailis et al. [14] since no proof technique is provided. Whereas Soteria analyses a program, in contrast the I-confluence paper [14] gives no means to link the program text to the semantic model, let alone rules for verifying that the program implies invariant preservation.

A final interesting remark is that we can show how our methodology can aid in the verification of distributed objects mediated by concurrency control. Some works [24, 25, 29, 30] have considered this problem from the standpoint of synthesis, or from the point of view of which mechanisms can be used to check a certain property of the system.

Chapter 6

Conclusion of Part I and Future work

This part of the thesis presented a sound proof rule to verify invariants of state-based distributed objects. We presented the proof obligations guaranteeing that the implementation is safe in concurrent execution, by reducing the problem to checking that each operation of the object satisfies a precondition of the `merge` function.

We presented Soteria, a tool that proves concurrent correctness or identify the concurrent bugs in the design of a distributed object. We have shown several case-studies showing how to leverage Soteria to ensure correctness of distributed objects.

6.1 Future work

There are several directions for future work on both theoretical and practical aspects.

On the theoretical front, one future research direction is to leverage the modular proof rule of Soteria to develop a generic proof rule that can verify distributed objects regardless of the type of update propagation. The first step would be to support distributed objects that propagate deltas.[1] The proof rule can then be extended to include distributed objects that propagate operations. This would help verify distributed applications with transparent update propagation.

The proof rule helps in identifying the conflicting operations. The next step on this is a future research direction. The user can either opt for a coordination-free application with the help of conflict resolution policies or a lock-based solution. It would be useful to have formal guidance on the type of applications that could benefit from conflict resolution as opposed to introducing coordination.

Practically, the future work is to improve the usability of Soteria. Even though Soteria is fully automatic, the user is expected to provide a specification with pre and post conditions of operations and merge, along with the object invariants. Automating the generation of preconditions using weakest precondition calculus [31] would be a first step. The specification might also include loop invariants which are difficult to write. It would be helpful to integrate the works that focus on generating loop invariants[32] that might help writing simpler specifications.

[1] Deltas are state changes.

Currently Soteria leverages Boogie[33] and in turn Z3 SMT solver [34] to verify the proof rules. Instead of relying on a single theorem prover, Soteria can be rewritten in a different verification framework, for example Why3 [35]. Why3 has a common specification language and serves as a front-end to different theorem solvers including Z3, Alt-Ergo [36], CVC4 [37], Coq[38], etc.

Another direction of work could look at generating code and tests from the verified specification.

Part II

Designing conflict resolution policies

Introduction to Part II

In the previous part, we presented an approach for verifying the safety of a concurrent application. Our tool, Soteria, outputs the list of conflicting methods if any verification condition fails. These methods, when executed concurrently are unsafe. We say they conflict.

The next question we face is how do we use this information. There are two options: to enhance the application with a conflict resolution, or with coordination. For highly-available distributed applications, it is preferable to avoid coordination, if we could still maintain safety. Hence, the preferred solution is to design a conflict resolution algorithm.

A conflict resolution algorithm must be deterministic and insensitive to order and duplication (i.e., idempotent, associative and commutative) to ensure that all replicas observe the same eventual outcome.

In this part of the thesis, we use the study of a coordination-free replicated tree data structure to illustrate the design of a distributed application, by developing an appropriate conflict resolution algorithm. A distributed tree supports three structural operations — add, remove and move. We further classify move operations into up-moves and down-moves. We identify the conditions under which they conflict and introduce conflict resolution policies to ensure a coordination-free, safe and available replicated tree.

Carla Ferreira, Mário Pereira and Filipe Meirim collaborated with us for this part of the thesis. A research report is available [39].

Chapter 7

Design of a safe, convergent and coordination free replicated tree

Concurrent data structures are an important programming abstraction; designing concurrent data structures with non-trivial properties is complex. The tree data structure is widely used, for instance in file systems and in graphical user interfaces. Trees have particularly strong requirements: each node is unique, there is a single root, a node has a single parent and has a path to the root, and the child-parent graph is acyclic.

Much current work in concurrent data structure design focuses on lock-free or wait-free coordination, using primitives such as compare-and-swap (CAS). However, in a distributed and replicated setting, even CAS is too strong. Consider for instance a file system replicated to several locations over the globe, or through a mobile network. Network round-trip-time between continents can be anywhere between 0.1 and 1.0 seconds; the mobile network may disconnect completely. To ensure availability, a user of the file system must be able to update a replica locally, and update *without coordinating at all* with the other replicas. Replicas will converge eventually, by exchanging updates asynchronously.

It is a major challenge to maintain safety in this context; specifically, in this case, to maintain the tree structure. This is a widespread issue; indeed, many replicated file systems have serious anomalies, including incorrect or diverged states [40, Section 6 for some examples], violating the tree invariant [41], non-atomic moves [40], re-introducing coordination [3], or requiring roll-backs [42].

Concurrent atomic move operations are a crucial problem [41]. Consider for instance a tree composed of the root and children a and b as shown in Figure 7.1. One replica moves a underneath b, while concurrently (without coordination) the other replica moves b under a. Naïvely replaying one replica's updates at the other produces an $a - b$ cycle disconnected from the root. There can be no coordination-free solution to this problem that is not somehow anomalous [3].

Supporting low latency, high-availability and safety, this part introduces a new coordination-free, safe, replicated CRDT [2] tree data structure, called *Maram*. Maram supports the usual operations to query the state, to add or to remove a node, and also supports an atomic *move* operation. The price to pay is that some move operations "lose", i.e., have no effect.

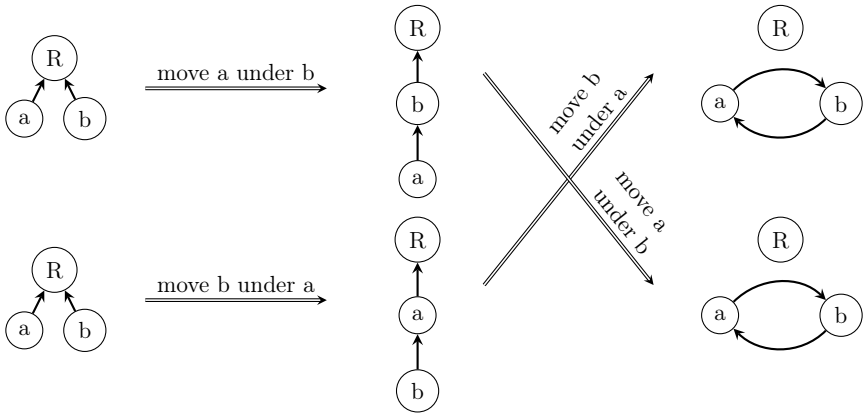


Figure 7.1: Concurrent cycle causing moves

Query and add are safe since the former does not change the state and the latter inflates the state monotonically [21]. Remove marks the corresponding node as a "tombstone," but leaves it in the data structure, as is common in replicated data structures [43]. Moves can be divided into two cases: *up-moves*, where a node moves near to the root of the tree, and *down-moves*, where it moves farther away from the root. We devise a deterministic arbitration rule for the conflicts of both the moves:

- For *up-move*: against a concurrent up-move of the same node, we assign a total order between operations, whereby the up-move with the highest priority wins, and the other loses.
- For *down-move*: against a concurrent up-move, the up-move wins, and the down-move loses; against a concurrent down-move, we use the same strategy as for two concurrent up-moves on the same node.

Further, we examine the effect of "losing" a move operation. We identify the conditions under which another causally dependent move would be unsafe under a move that loses. For each move, we preserve a causal history of the dependent operations and the move will lose if it was dependent on a causally preceding conflicting move that loses.

We provide arguments for the safety violations of a replicated tree, in the presence of concurrent updates (including moves), being coordination-free. To this effect, we apply the CISE proof methodology [6]. It follows that every state reachable from the initial state —whether sequentially or concurrently— satisfies the tree invariant.[1] We also use an additional analysis to check the causal dependencies between move operations.

Maram satisfies an additional desirable property, *monotonic reads* [44]. This requires that a replica that has delivered some update will not roll it back.

Effect of a move remains tentative until all concurrent moves have been delivered. A concurrent move may "win" over it, i.e., the effect of the winning move supersedes the losing one, whose effect is skipped. We show the effect of skipping on causally dependent move

[1] We furthermore claim (without proof) that Maram is live, in the sense that, if every message sent is eventually delivered to all replicas, then, given some update originating at some replica, its postcondition eventually takes effect at every replica.

operation; the skipping is safe with respect to the other operations since it is independent of any other operation.

We present the principles of Maram, provide arguments for its correctness, and compare the performance of Maram to competing solutions in a simulated geo-replicated environment. The response time of Maram is 1.35 times of the safe rollback-based design, and 1.36 times of the unsafe uncoordinated design (both due to overhead of computing the metadata required for conflict resolution), and up to 11 times faster than (safe) lock-based designs. Furthermore, Maram stabilizes (i.e., its updates become definitive) three times faster than a safe rollback-based design when almost one-third of the workload consist of move operations.

7.1 System Model

A distributed system is modelled as a set of processes, distributed over a (high-latency, failure-prone) communication network. The processes have disjoint memory and processing capabilities, and they communicate through message passing. A process does not fail. Every message is eventually delivered to its destination. Message delivery is consistent with happens-before (causal consistency).

State and invariant:

The data structure (in this case, a tree) is *replicated* at a number of processes, called its *replicas*. The information managed by a replica on behalf of the data structure is called its *local state*. The union of local states is called the *global state*.[2]

A data structure is associated with an *invariant*, a predicate that must always be satisfied in every local state of a replica. Although evaluated locally, an invariant describes a global property, in the sense that it must be true at all replicas.

Operations:

An unspecified client application submits an operation at some replica of its choice, which we call the *origin* replica of that operation. For availability, the origin replica should carry out the operation without waiting to coordinate with other replicas.

An update operation has a *postcondition* that specifies the state after the operation executes, and a *precondition* that indicates the domain of the operation. As discussed in more detail later, when the operation executes with no concurrency, its precondition guarantees that the operation terminates with the postcondition satisfied.

Updates:

When a client submits an operation, the origin replica generates an *effector* (a side-effecting lambda), atomically applies the effector to the origin state, and sends the effector to all

[2] Note that this global view cannot be observed by any single replica and is merely an explanatory device.

the other replicas. Every replica eventually receives and delivers the effector, atomically applying it to its own local state.[3] The effector eventually executes at every replica.

We assume that effectors are delivered in causal order. This means that, if some replica that observed an effector u later generates an effector v, then any replica that observes v has previously observed u.[4]

In what follows, we ignore queries, and identify an update operation with executing its effector at all replicas.

7.2 Properties and associated proof rules

Consider some data structure (in this case a tree) characterized by a safety *invariant* (in this case, the tree invariant). We say that a state is *local-safe* if it satisfies the data structure's invariant. An update is *op-safe* if, starting from a local-safe state, it leaves it a local-safe state. The distributed data structure is *safe* if every update is op-safe. According to the CISE logic [6], a distributed data structure is safe if the following properties hold:

1. *Sequential safety*: Consider an environment restricted to sequential execution (operations execute one after another; there is no concurrency). If the initial state is local-safe at every replica, and each update is op-safe, it follows that the data structure is safe under sequential execution. Classically, sequential op-safety implies that each operation's precondition satisfies the weakest-precondition of the invariant with respect to the operation [31].
2. *Convergence:* Strong Eventual Consistency (SEC) [2] states that two replicas that have delivered the same set of operations must be in the same state, i.e., the system converges. If operations commute (as defined later), then SEC is guaranteed [2].
3. *Precondition stability:* In addition to sequential safety, updates must remain op-safe in the presence of concurrent (uncoordinated) updates. To ensure this, we apply the CISE precondition stability rule [6]: consider two updates u and v; if the execution of u does not make the precondition of v false, nor vice-versa (*precondition stability*), then executing u and v concurrently is op-safe. This must be true for all concurrent pairs of operations.

CISE logic helps us identify the conditions under which concurrent operations conflict. When conflicting, CISE requires the operations to acquire tokens, that bring in a global synchronization point. Hence all updates in CISE are assumed to be definitive.

In order to augment the CISE analysis for handling tentative updates, we add a condition for *independence* to check whether skipping a move affects a move that already observed the effect of the skipped one. The independency analysis is inspired from Houshmand and Lesani [9], even though they also, like CISE, do not consider tentative updates.

Independence analysis: Consider two updates u and v that are safe, u executed before v. If moving v before u still maintains the safety of v, v is said to be independent of u.

[3] Note that, at this point, the system is committed to this operation, and the operation's precondition must be true at the remote replica.

[4] In Section 7.9 we consider relaxing this requirement to eventual consistency, which states only that all updates are eventually delivered at all replicas.

Otherwise, if v is unsafe before executing u, v is dependent on the effect of u.

7.2.1 Sequential safety

Let us refine the proof obligations of the first step, sequential safety, i.e., local-safety under sequential execution.

The set of reachable states comprises the initial state, and all states transitively reachable as a result of executing updates sequentially. The set of reachable states is a subset of the set of all possible states. Formally, we note the set of states Σ, a state σ, the initial state σ_{init}, an update u, its precondition Pre_u, and the set of updates U. When execution is sequential:

$$\sigma_{init} \in \Sigma \tag{7.1}$$

and

$$\forall u \in U, \sigma \in \Sigma \,.\, \sigma \models \mathsf{Pre}_u \implies u(\sigma) \in \Sigma \tag{7.2}$$

Σ is the smallest set satisfying (7.1) and (7.2) through a sequence of legal updates from the initial state.

The data structure must satisfy its invariant in every sequentially reachable state: this property is called *sequential safety*. Formally, if Inv denotes the invariant, then

$$\forall \sigma \in \Sigma \,.\, \sigma \models \mathsf{Inv} \tag{7.3}$$

If the initial state is safe and all sequential updates preserve the invariant, by induction, the data structure is sequentially safe. Formally, if the initial state, σ_{init}, satisfies the invariant, Inv,

$$\sigma_{init} \models \mathsf{Inv} \tag{7.4}$$

and each update u executing on a state σ preserves the invariant,

$$\forall u \in U, \sigma, \sigma' \in \Sigma \,.\, \sigma \models (\mathsf{Inv} \wedge \mathsf{Pre}_u) \wedge u(\sigma) = \sigma' \implies \sigma' \models \mathsf{Inv} \tag{7.5}$$

then the invariant holds true for all reachable states. Pre_u is the weakest precondition required to maintain the safety of update u. Weakest precondition for an update can be calculated by predicate transformer semantics as described by Dijkstra [31].

7.2.2 Concurrency

Let us now turn to concurrent execution, and consider the proof obligations for convergence and safety.

7.2.2.1 Convergence

If a replica initiates an update u, while concurrently another replica initiates v, the first replica executes their effectors in the order $u; v$ and the second one in the order $v; u$.

Without precaution, it is likely that their states diverge.

To prevent this, the Strong Eventual Consistency (SEC) property [2] requires that any two replicas that delivered the same updates are in equivalent states. To satisfy SEC, effector functions are designed to commute, i.e., both orders above leave the data in the same state. We define commutativity as follows:

$$\forall u_1, u_2 \in U, \sigma, \sigma_1, \sigma_2 \in \Sigma \,.\, u_1(\sigma) = \sigma_1 \wedge u_2(\sigma) = \sigma_2 \implies u_2(\sigma_1) = u_1(\sigma_2) \tag{7.6}$$

7.2.2.2 Precondition stability

The main proof obligation for concurrent execution is that the precondition of any effector is stable against (i.e., not negated by) an effector that may execute concurrently [6]. This CISE rule is a variant of rely-guarantee reasoning, adapted to a replicated system where effectors execute atomically and definitively. The precondition stability condition can be formally specified as follows:

$$\forall u_1, u_2 \in U, \sigma, \sigma' \in \Sigma \,.\, \sigma \models (\mathsf{Inv} \wedge \mathsf{Pre}_{u_1} \wedge \mathsf{Pre}_{u_2}) \wedge u_1(\sigma) = \sigma' \implies \sigma' \models \mathsf{Pre}_{u_2} \tag{7.7}$$

Gotsman et al. [6] uses *Tokens* to formalize concurrency control. Two operations that share the same token do not execute concurrently. Since we are designing a coordination-free data structure, we consider the set of tokens to be an empty set, and hence absent from the formalisation.

7.2.2.3 Independence

In order to ensure that the safety of an operation is not impacted by skipping any previous operations, we augment the precondition stability analysis with an independence analysis as presented by Houshmand and Lesani [9]. An operation u_2 is said to be independent of operation u_1 if the precondition of u_2, Pre_{u_2}, is enabled even without executing u_1. The condition for independency can be formally specified as follows:

$$\begin{aligned}\forall u_1, u_2 \in U, \sigma, \sigma', \sigma'', \sigma''' \in \Sigma \,.\, \sigma \models (\mathsf{Inv} \wedge \mathsf{Pre}_{u_1}) \wedge \sigma' \models (\mathsf{Inv} \wedge \mathsf{Pre}_{u_2}) \\ \wedge \sigma'' \models \mathsf{Inv} \wedge u_1(\sigma) = \sigma' \wedge u_2(\sigma') = \sigma'' \wedge u_2(\sigma) = \sigma''' \implies \sigma \models \mathsf{Pre}_{u_2} \wedge \sigma''' \models \mathsf{Inv}\end{aligned} \tag{7.8}$$

In short, u_2 is independent of u_1 if, irrespective of whether u_1 executed before u_2, the execution of u_2 is safe. This condition is required for safety only if the effect of u_1 is tentative, i.e., if u_1 has conflict resolution policies while applying the update on the state.

7.2.3 Mechanized verification

In order to mechanically discharge the proof obligations listed above, we the use Why3 system [45], augmented with the CISE3 plug-in [46]. Why3 is a framework used for the deductive verification of programs, *i.e.*, "the process of turning the correctness of a program into a mathematical statement and then proving it" [47]. The CISE3 plug-in automates the CISE proof rules described above, and generates the required sequential-safety, com-

mutativity and stability checks. Why3 then computes a set of proof obligations, that are discharged via external theorem provers.

7.3 Sequential specification of a tree

The specification of a data structure consists of its state, a set of operations, and an invariant. In this section, we will develop a sequentially-safe specification of a tree.

7.3.1 State

The state of a tree data structure consists of a set of nodes, Nodes, and a relation on nodes, mapping a child node to its parent. The parent relation is indicated by $\rightarrow$. The ancestor relation, $\rightarrow^*$ is defined as

$$\forall a, n \in \mathsf{Nodes} \,.\, n \rightarrow^* a \triangleq n \rightarrow a \;\; \vee \exists p \in \mathsf{Nodes} \,.\, n \rightarrow p \wedge p \rightarrow^* a \tag{7.9}$$

At initialization, the set of nodes consists of a single *root* node. The parent of the root is the root itself. The initial state of the tree is thus $\mathsf{Nodes} = \{\mathsf{root}\}$ where $\mathsf{root} \rightarrow \mathsf{root}$.

A crucial aspect of the abstract representation of the tree is how to express the relation between nodes. Three choices are possible, either maintain a child-to-parent mapping, a parent-to-child mapping, or both. In particular, when implementing a tree, traversal efficiency depends on keeping both up and down pointers [48]. Considering that child-to-parent and parent-to-child mappings describe dual views (i.e., node p is the parent of node n iff node n is a descendant of node p) we selected the one that leads to a simpler specification. An advantage of using a child-to-parent mapping is that it can be maintained as a function, ensuring that each node has a unique parent. The alternative parent-to-child mapping would require a more complex representation, e.g., a function that maps each node to its set of direct descendants, which would negatively impact both the simplicity of the specification and the proof effort.

7.3.2 Invariant

Formally, the invariant of the tree data structure is as follows:

$$\mathsf{root} \in \mathsf{Nodes} \wedge \mathsf{root} \rightarrow \mathsf{root} \wedge \forall n \in \mathsf{Nodes} \,.\, n \neq \mathsf{root} \implies \mathsf{root} \not\rightarrow n \tag{Root}$$

$$\wedge\; \forall n \in \mathsf{Nodes} \,.\, \exists p \in \mathsf{Nodes} \,.\, n \rightarrow p \tag{Parent}$$

$$\wedge\; \forall n, p, p' \in \mathsf{Nodes} \,.\, n \rightarrow p \wedge n \rightarrow p' \implies p = p' \tag{Unique}$$

$$\wedge\; \forall n \in \mathsf{Nodes} \,.\, n \rightarrow^* \mathsf{root} \tag{Reachable}$$

Clause *Root* lists the properties of the root node; present in Nodes, and is the only node to be its own parent. Clause *Parent* asserts that every node in the tree has a parent present in the tree. Clause *Unique* requires the parent for each node to be unique. Clause *Reachable* imposes that the root is an ancestor of all nodes.

We call this conjunction the *tree invariant*.

$$\mathsf{Inv} \triangleq \mathit{Root} \wedge \mathit{Parent} \wedge \mathit{Unique} \wedge \mathit{Reachable} \tag{7.10}$$

A further invariant:

$$\forall n \in \mathsf{Nodes} \,.\, n \neq \mathsf{root} \implies n \not\rightarrow^* n \tag{Acyclic}$$

which forbids cycles (no node is ancestor of itself, except root), can be derived from the previous invariants. Since the parent relation inductively defines the ancestor relation, by *Unique* there is a unique path to a given ancestor of a node. By *Reachable*, the root node is an ancestor of every node in the tree. In this scenario, a cycle would require a node to have multiple parents, which is prevented by *Unique*.

7.3.3 Operations

We consider the following three structural operations: add, remove and move.

Add

An add operation has two arguments: the node to be added, n, and its prospective parent, p. The add effector adds node n to Nodes and the mapping $n \rightarrow p$ to the parent relation. The postcondition of the add effector indicates this:[5]

$$\mathsf{Post}_{add(n,p)} \triangleq n \in \mathsf{Nodes} \wedge n \rightarrow p \tag{7.11}$$

To ensure that the tree invariant is preserved, we derive, through the weakest precondition calculus, the precondition that n is a new node and p is already in the tree, i.e.,

$$\mathsf{Pre}_{add(n,p)} \triangleq n \notin \mathsf{Nodes} \wedge p \in \mathsf{Nodes} \tag{7.12}$$

Let us decompose the derivation of this precondition. If the add operation is updating a safe state, i.e., the starting state respects the invariant, and if the precondition is satisfied, then the update should maintain the invariant. Hereafter, we highlight the precondition clauses needed to ensure each part of the invariant.[6]

$$\frac{\mathsf{Inv} \wedge n \notin \mathsf{Nodes} \qquad [\![add(n,p)]\!]}{\mathsf{Post}_{add(n,p)} \wedge \mathit{Root}} \qquad \frac{\mathsf{Inv} \wedge p \in \mathsf{Nodes} \qquad [\![add(n,p)]\!]}{\mathsf{Post}_{add(n,p)} \wedge \mathit{Parent}}$$

$$\frac{\mathsf{Inv} \wedge n \notin \mathsf{Nodes} \qquad [\![add(n,p)]\!]}{\mathsf{Post}_{add(n,p)} \wedge \mathit{Unique}} \qquad \frac{\mathsf{Inv} \wedge p \in \mathsf{Nodes} \qquad [\![add(n,p)]\!]}{\mathsf{Post}_{add(n,p)} \wedge \mathit{Reachable}}$$

With the derived preconditions, the add operation can be specified as follows:

(Add-Operation)

$$\frac{\mathsf{Inv} \wedge n \notin \mathsf{Nodes} \wedge p \in \mathsf{Nodes} \qquad [\![add(n,p)]\!]}{\mathsf{Inv} \wedge n \in \mathsf{Nodes} \wedge n \rightarrow p}$$

[5] For readability, we simplify the postcondition to express only the changes caused by the operation. The part of the state not mentioned remains unaffected.

[6] Denoted in inference style, as in Kaki et al. [25]. An update event is noted $[\![.]\!]$.

If the add operation is issued on a state that does not contain n, and p is in the tree, then n is added to the tree with a pointer to p. If the operation is issued in a state that does not respect the precondition, it is skipped.

Remove operation:

Remove receives as argument a node n to be deleted. Its effector removes node n from the set of nodes. The postcondition of the remove operation indicates this effect:

$$\mathsf{Post}_{remove(n)} \triangleq n \notin \mathsf{Nodes} \tag{7.13}$$

Similar to add, we list the predicates needed to preserve each clause of the invariant. In this case, we must ensure that n is not the root, and n is a leaf node, i.e., there are no child nodes for n.

$$\frac{\mathsf{Inv} \wedge n \neq \mathsf{root} \qquad [\![remove(n)]\!]}{\mathsf{Post}_{remove(n)} \wedge \mathit{Root}} \qquad \frac{\mathsf{Inv} \wedge \forall n' \in \mathsf{Nodes} \,.\, n' \not\rightarrow n \qquad [\![remove(n)]\!]}{\mathsf{Post}_{remove(n)} \wedge \mathit{Parent}}$$

$$\frac{\mathsf{Inv} \wedge \mathtt{true} \qquad [\![remove(n)]\!]}{\mathsf{Post}_{remove(n)} \wedge \mathit{Unique}} \qquad \frac{\mathsf{Inv} \wedge \forall n' \in \mathsf{Nodes} \,.\, n' \not\rightarrow n \qquad [\![remove(n)]\!]}{\mathsf{Post}_{remove(n)} \wedge \mathit{Reachable}}$$

In summary, the remove operation can be specified as follows:

$$\frac{\begin{array}{l}(\textsc{Remove-Operation}) \\ \mathsf{Inv} \wedge n \neq \mathsf{root} \wedge \forall\, n' \in \mathsf{Nodes} \,.\, n' \not\rightarrow n \qquad [\![remove(n)]\!]\end{array}}{\mathsf{Inv} \wedge n \notin \mathsf{Nodes}}$$

If a remove operation is issued on a state where n is not root and has no children, then n is removed from the tree; otherwise it is skipped.

Move operation:

The move operation takes two arguments: the node to be moved n, and the new parent p'. Its effector changes the parent of node n to p', with the following postcondition:

$$\mathsf{Post}_{move(n,p')} \triangleq n \rightarrow p' \tag{7.14}$$

Note that the postcondition does not state that the previous parent is no longer a parent of node n, i.e., $n \not\rightarrow p$, because of the uniqueness of the child-to-parent relationship as discussed in Subsection 7.3.1.

To preserve the expected behaviour of the move operation we require the node to be moved to be present in the tree. Together with this precondition, we derive the additional

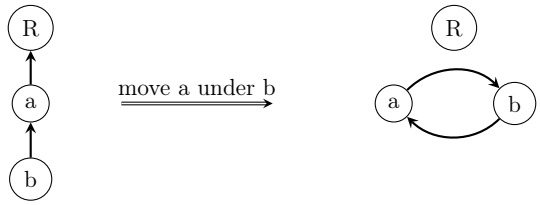


Figure 7.2: Move update violating tree invariant

clauses required for the safety of move.

$$\frac{\mathsf{Inv} \wedge n \neq \mathsf{root} \qquad [\![move(n,p')]\!]}{\mathsf{Post}_{move(n,p')} \wedge Root} \qquad \frac{\mathsf{Inv} \wedge p' \in \mathsf{Nodes} \qquad [\![move(n,p')]\!]}{\mathsf{Post}_{move(n,p')} \wedge Parent}$$

$$\frac{\mathsf{Inv} \wedge \mathtt{true} \qquad [\![move(n,p')]\!]}{\mathsf{Post}_{move(n,p')} \wedge Unique} \qquad \frac{\mathsf{Inv} \wedge \begin{array}{c} p' \in \mathsf{Nodes} \wedge p' \not\rightarrow^* n \wedge \\ (n \neq root \implies p' \neq n) \end{array} \qquad [\![move(n,p')]\!]}{\mathsf{Post}_{move(n,p')} \wedge Reachable}$$

These last condition for ensuring *Reachable* is to prevent move from creating a cycle, rendering some nodes unreachable, as we show with the following counterexample.

Consider nodes a and b in Figure 7.2. Root node R is the parent of node a, i.e., $a \rightarrow R$ and node a is the parent of node b, $b \rightarrow a$, and hence R is the ancestor of b, $b \rightarrow^* R$. Moving node a under node b will make both a and b unreachable from the root, and also form a cycle. This violates the invariant by invalidating the tree structure. To avoid this scenario, a precondition is needed that prevents moving a node underneath itself. When moving node n from its current parent to the new parent p', p' should not be n (except when $n = p = p' = root$) or a descendant of n, $p' \neq n \wedge p' \not\rightarrow^* n$.

Combining all these conditions, the move operation can be specified as follows:

(Move-Operation)

$$\frac{\mathsf{Inv} \wedge n \in \mathsf{Nodes} \wedge n \neq \mathsf{root} \wedge p' \in \mathsf{Nodes} \wedge p' \neq n \wedge p' \not\rightarrow^* n \qquad [\![move(n,p')]\!]}{\mathsf{Inv} \wedge n \rightarrow p'}$$

For the move operation to be safe, n is not the root, p' must be in the tree, n and p' are different, and p' is not a descendant of n.

7.3.4 Mechanized verification of the sequential specification

Following the formalization of the tree data structure above, we use Why3 to mechanically prove its sequential safety. The mechanical proof requires some extra definitions and axioms.

We need a predicate for reachability. For this, we first define a path, a sequence of nodes related by the parent relation. We use $s[n]$ to indicate the nth element in the sequence s. We denote the set of possible sequences of nodes by S. The path predicate determines the validity conditions for a path s between nodes x and y in state σ. If $x = y$, the path has length zero. Otherwise, the length of the path is greater than zero, where the first path element must be x, all contiguous path elements are related by the parent relation, and node y is the parent of the last path element. We say y is reachable from x if there exists

a path from x to y. Formally,

$$path(\sigma, x, y, s) \triangleq length(s) = 0 \wedge x = y \tag{7.15}$$
$$\vee (length(s) > 0 \wedge s[0] = x \wedge s[length(s) - 1] \rightarrow y \wedge$$
$$\forall\, 0 \leq i < length(s) - 1 \,.\, s[i] \rightarrow s[i+1])$$
$$reachability(\sigma, x, y) \triangleq \exists s \in S \,.\, path(\sigma, x, y, s) \tag{7.16}$$

To formalize the properties of the *path* predicate, we define a set of axioms as follows:

$$path_to_parent \triangleq \forall \sigma \in \Sigma \,.\, \forall x, y \in \mathsf{Nodes} \,.\, x \rightarrow y \implies \exists s \in S \,.\, path(\sigma, x, y, s) \wedge s = [x] \tag{7.17}$$
$$path_composition \triangleq \forall \sigma \in \Sigma \,.\, \forall x, y, z \in \mathsf{Nodes} \,.\, \exists s_1 \in S \,.\, path(\sigma, x, y, s_1) \tag{7.18}$$
$$\wedge\, y \rightarrow z \implies \exists s_2 \in S \,.\, path(\sigma, x, z, s_2) \wedge s_2 = s_1 + [y]$$
$$path_transitivity \triangleq \forall \sigma \in \Sigma \,.\, \forall x, y, z \in \mathsf{Nodes}, s_1, s_2 \in S \,.\, path(\sigma, x, y, s_1) \tag{7.19}$$
$$\wedge\, path(\sigma, y, z, s_2) \implies \exists s_3 \in S \,.\, path(\sigma, x, z, s_3) \wedge s_3 = s_1 + s_2$$
$$path_uniqueness \triangleq \forall \sigma \in \Sigma \,.\, \forall x, y \in \mathsf{Nodes}, s_1, s_2 \in S \,.\, path(\sigma, x, y, s_1) \tag{7.20}$$
$$\wedge\, path(\sigma, x, y, s_2) \implies s_1 = s_2$$
$$path_exclusion \triangleq \forall \sigma \in \Sigma \,.\, \forall x, y, z \in \mathsf{Nodes}, s \in S \,.\, x \not\rightarrow^* y \wedge path(\sigma, z, y, s) \implies x \notin s \tag{7.21}$$
$$path_separation \triangleq \forall \sigma \in \Sigma \,.\, \forall x, y, z \in \mathsf{Nodes}, s_1, s_2 \in S \,.\, path(\sigma, x, y, s_1) \tag{7.22}$$
$$\wedge\, path(\sigma, y, z, s_2) \wedge x \neq y \wedge x \neq z \wedge y \neq z \implies s_1 \cap s_2 = \emptyset$$

Axiom *path_to_parent* defines the singleton path of a node to its parent. The recursive composition of paths is axiomatized in *path_composition*. The transitivity property is defined in *path_transitivity*. Axiom *path_uniqueness* asserts there is a single path between two nodes. The *path_exclusion* expresses the conditions for excluding nodes from a path. Lastly, *path_separation* defines a convergence criterion essential for Why3's SMT solvers, asserting that the direction of the path is converging towards the root.

We also require extra axioms to express the properties of the unaffected nodes in the case of add and move operations. They are as follows:

$$\sigma_{add} = add(n, p)(\sigma)$$
$$\sigma_{move} = move(n, p)(\sigma)$$
$$remaining_nodes_add \triangleq \forall \sigma \in \Sigma \,.\, \forall n' \in \mathsf{Nodes}, s_1, s_2 \in seq(\mathsf{Nodes}) \,.\, n' \neq n \tag{7.23}$$
$$\wedge\, path(\sigma, n', \mathsf{root}, s_1) \wedge path(\sigma_{add}, n', \mathsf{root}, s_2) \implies s_1 = s_2$$
$$descendants_move \triangleq \forall \sigma \in \Sigma \,.\, \forall n' \in \mathsf{Nodes}, s_1, s_2 \,.\, path(\sigma, n', c, s_1) \tag{7.24}$$
$$\wedge\, path(\sigma_{move}, n', c, s_2) \implies s_1 = s_2$$
$$remaining_nodes_move \triangleq \sigma \in \Sigma \,.\, \forall n' \in \mathsf{Nodes}, s_1, s_2 \,.\, n' \not\rightarrow^* n \tag{7.25}$$
$$\wedge\, path(\sigma, n', \mathsf{root}, s_1) \wedge path(\sigma_{move}, n', \mathsf{root}, s_2) \implies s_1 = s_2$$

The state σ_{add} is obtained by applying $add(n, p)$ operation to σ. The axiom *remaining_nodes_add* asserts that the paths already present in the tree remain in the tree after executing the add operation. Given that the move operation updates σ to σ_{move}, axiom *descendants_move* asserts that the descendants of the node being moved continue to be its descendants, and *remaining_nodes_move* asserts that other paths are not affected. These axioms are de-

fined to ensure that the paths to the root, from nodes unaffected by move or add operations, remain unchanged. The specification proven using Why3 is available in Meirim et al. [49].

7.4 Concurrent tree specification

In this section, we discuss the convergence and concurrent safety of the tree. In a sequential execution environment, as seen in Section 7.3, if the initial state and each individual update are safe, then all reachable states are safe. This is not true when executing concurrently on multiple replicas. In this case, there are three extra proof obligations (Subsection 7.2.2.1, Subsection 7.2.2.2, Subsection 7.2.2.3):

- Ensuring that different replicas converge, despite effectors being executed concurrently in different orders.
- Ensuring that safety of an update is not violated by a concurrent update.
- Ensuring that a tentative update does not effect the safety of the dependent update.

For ease of exposition, first we discuss concurrent safety; convergence is deferred to Section 7.6, since the conflicts occurring in the latter can be addressed using the policies discussed in the former, and independence is discussed in Section 7.7.

7.4.1 Precondition stability

We use the precondition stability rule of CISE logic (Subsection 7.2.2.2) to analyze the concurrent safety of our tree data structure. For each operation, we analyze whether it violates the precondition of any other concurrent operation. Formally, operation op_1 is stable under operation op_2 if,

$$\frac{\mathsf{Inv} \wedge \mathsf{Pre}_{op_1} \wedge \mathsf{Pre}_{op_2} \qquad [\![op_2]\!]}{\mathsf{Inv} \wedge \mathsf{Post}_{op_2} \wedge \mathsf{Pre}_{op_1}} \tag{7.26}$$

We check the sequential specification for stability. If this fails, then it will be necessary to modify the specification, so that it does satisfy stability.

7.4.1.1 Stability of add operation

Concurrent adds: First we check the stability of the precondition of add against itself. Let us consider two operations $add(n_1, p_1)$ and $add(n_2, p_2)$. Using Equation (7.26), we get

$$\begin{aligned}
\mathsf{Pre}_{add(n_1,p_1)} &\triangleq n_1 \notin \mathsf{Nodes} \wedge p_1 \in \mathsf{Nodes} \\
\mathsf{Pre}_{add(n_2,p_2)} &\triangleq n_2 \notin \mathsf{Nodes} \wedge p_2 \in \mathsf{Nodes} \\
\mathsf{Post}_{add(n_2,p_2)} &\triangleq n_2 \in \mathsf{Nodes} \wedge n_2 \rightarrow p_2
\end{aligned}$$

$$\frac{\mathsf{Inv} \wedge \mathsf{Pre}_{add(n_1,p_1)} \wedge \mathsf{Pre}_{add(n_2,p_2)} \wedge n_1 \neq n_2 \qquad [\![add(n_2, p_2)]\!]}{\mathsf{Inv} \wedge \mathsf{Post}_{add(n_2,p_2)} \wedge \mathsf{Pre}_{add(n_1,p_1)}} \tag{7.27}$$

The highlighted clause $n_1 \neq n_2$ is required for the stability condition. Indeed, the sequential specification does not disallow adding the same node at different replicas, and the clause $n \notin \mathsf{Nodes}$ is unstable therein. Thus the analysis highlights a subtlety.

Concurrent remove: Let us check the stability of the precondition of $add(n_1, p_1)$ against a concurrent $remove(n_2)$. Using (7.26), we get:

$$\begin{aligned}
\mathsf{Pre}_{add(n_1,p_1)} &\triangleq n_1 \notin \mathsf{Nodes} \wedge p_1 \in \mathsf{Nodes} \\
\mathsf{Pre}_{remove(n_2)} &\triangleq n_2 \neq \mathsf{root} \wedge \forall\, n' \in \mathsf{Nodes} \,.\, n' \not\rightarrow n_2 \\
\mathsf{Post}_{remove(n_2)} &\triangleq n_2 \notin \mathsf{Nodes}
\end{aligned}$$

$$\frac{\mathsf{Inv} \wedge \mathsf{Pre}_{add(n_1,p_1)} \wedge \mathsf{Pre}_{remove(n_2)} \wedge \boxed{n_2 \neq p_1} \qquad [\![remove(n_2)]\!]}{\mathsf{Inv} \wedge \mathsf{Post}_{remove(n_2)} \wedge \mathsf{Pre}_{add(n_1,p_1)}} \tag{7.28}$$

In the sequential specification, clause $p_1 \in \mathsf{Nodes}$ in the precondition of add is unstable against a remove of its parent; performing those operations concurrently would be unsafe.

To fix this, we see two possible approaches. The classical way is to strengthen the precondition with coordination, for instance locking to avoid concurrency. We reject this, as it conflicts with our objective of availability under partition. Our alternative is to weaken the specification thanks to coordination-free conflict resolution. We apply a common approach, to mark a node as deleted, as a so-called *tombstone*, without actually removing it from the data structure.[7]

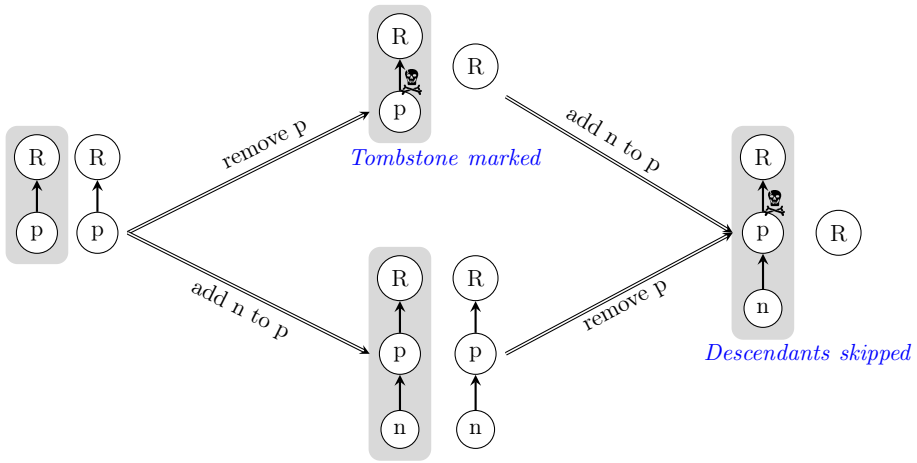
We now distinguish a *concrete* state and its *abstract* view. We modify the specification to include a set of tombstones, TS (initially empty), in the concrete state. The abstract state is the resolved state as seen by some application using Maram. An *abstraction function* maps the concrete state to the abstract state.

The concrete and abstract states of a tree are the same if either there are no nodes in the set of tombstones or for each node in the set of tombstones, all its descendants are also present in the set of tombstones. In other cases, the abstraction function need to provide guidance on the presence of the descendants of a node that appears in the set of tombstones.

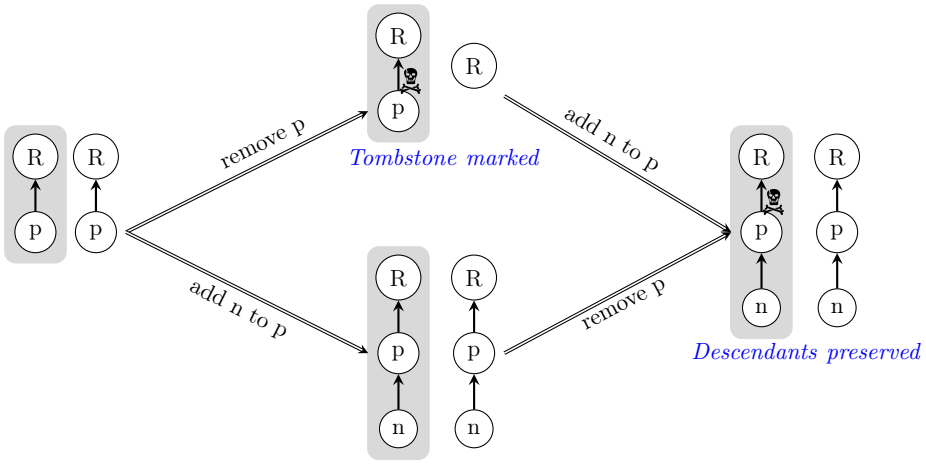
We present two *abstraction functions* - *skipping_ abstraction* and *keeping_ abstraction*. The *skipping_ abstraction* skips the descendants of the node that is marked as a tombstone. The *keeping_ abstraction*, on the other hand, preserves the tombstoned node if it observes the node has a descendant not in the set of tombstones. Both the abstraction functions satisfy the required safety properties since they only change the view of the tree for an application. Therefore the choice is application-specific.

Formally, if Nodes_{con} and Nodes_{abs} denote the set of nodes in the concrete and abstract

[7] Ideally, one will remove the tombstone at some safe time in the future; this is non-trivial [50] and out of the scope of this paper.



(a) Skipping abstraction



(b) Keeping abstraction

Figure 7.3: Resolving conflict of concurrent remove and add

state respectively,

$$\begin{aligned} \mathit{skipping_abstraction} \triangleq & \forall n \in \mathsf{Nodes}_{con} \bullet n \notin \mathsf{TS} \wedge \nexists n' \in \mathsf{Nodes}_{con} \bullet \\ & n' \in \mathsf{TS} \wedge n \rightarrow^* n' \iff n \in \mathsf{Nodes}_{abs} \end{aligned} \tag{7.29}$$

$$\begin{aligned} \mathit{keeping_abstraction} \triangleq & \forall n \in \mathsf{Nodes}_{con} \bullet n \notin \mathsf{TS} \vee \exists n' \in \mathsf{Nodes}_{con} \bullet \\ & n' \notin \mathsf{TS} \wedge n' \rightarrow^* n \iff n \in \mathsf{Nodes}_{abs} \end{aligned} \tag{7.30}$$

To illustrate the difference, consider the tree consisting of the root and a single child, as shown in Figure 7.3. One replica performs a remove of node p, while concurrently another replica adds n under p. In the first replica, node p is marked as a tombstone in the concrete state (the shaded box). Thus, the abstract state shows node p removed. When the replicas exchange their updates, they converge to the concrete state (the state in the shaded box). Figure 7.3a and Figure 7.3b show the result of a *skipping_ abstraction* and *keeping_ abstraction* respectively. In both the cases, node p is marked as a tombstone. In the case of the *skipping_ abstraction*, node p and the descendants are "skipped". Meanwhile for *keeping_ abstraction*, since its descendant n is not a tombstone, p is "revived" in the abstract view.

With tombstones, let us update the postcondition for remove:

$$\mathsf{Post}_{remove(n)} \triangleq n \in \mathsf{TS} \tag{7.31}$$

Let us now derive the predicates needed to preserve each clause of the invariant in this refined case.

$$\frac{\mathsf{Inv} \wedge n \neq \mathsf{root} \qquad [\![remove(n)]\!]}{\mathsf{Post}_{remove(n)} \wedge \mathit{Root}} \qquad \frac{\mathsf{Inv} \wedge \mathtt{true} \qquad [\![remove(n)]\!]}{\mathsf{Post}_{remove(n)} \wedge \mathit{Parent}}$$

$$\frac{\mathsf{Inv} \wedge \mathtt{true} \qquad [\![remove(n)]\!]}{\mathsf{Post}_{remove(n)} \wedge \mathit{Unique}} \qquad \frac{\mathsf{Inv} \wedge \mathtt{true} \qquad [\![remove(n)]\!]}{\mathsf{Post}_{remove(n)} \wedge \mathit{Reachable}}$$

To maintain sequential safety in the modified remove specification, the precondition forbids only removing the root node. As the remove operation does not alter the tree structure, reachability is not impacted. The refined specification of the remove operation is as follows:

(Remove-Operation)

$$\frac{\mathsf{Inv} \wedge n \neq \mathsf{root} \qquad [\![remove(n)]\!]}{\mathsf{Inv} \wedge n \in \mathsf{TS}}$$

The application could strengthen this precondition with an added clause to delete only the leaf nodes visible in the abstract view. This helps prevent accident loss of a sub-tree. Since this is not necessary for safety, we are not considering that condition.

Concurrent move: Next we check the stability of the precondition of add under a concurrent move operation. Let us consider two operations $add(n_1, p_1)$ and $move(n_2, p_2')$. Using (7.26), we get

$$\begin{aligned}
\mathsf{Pre}_{add(n_1,p_1)} &\triangleq n_1 \notin \mathsf{Nodes} \wedge p_1 \in \mathsf{Nodes} \\
\mathsf{Pre}_{move(n_2,p_2')} &\triangleq n_2 \in \mathsf{Nodes} \wedge n_2 \neq \mathsf{root} \wedge p_2' \in \mathsf{Nodes} \wedge p_2' \neq n_2 \wedge p_2' \not\rightarrow^* n_2 \\
\mathsf{Post}_{move(n_2,p_2')} &\triangleq n_2 \rightarrow p_2'
\end{aligned}$$

$$\frac{\mathsf{Inv} \wedge \mathsf{Pre}_{add(n_1,p_1)} \wedge \mathsf{Pre}_{move(n_2,p_2')} \wedge \mathtt{true} \qquad [\![move(n_2, p_2')]\!]}{\mathsf{Inv} \wedge \mathsf{Post}_{move(n_2,p_2')} \wedge \mathsf{Pre}_{add(n_1,p_1)}} \tag{7.32}$$

We see that the precondition of add is stable against a concurrent move operation.

7.4.1.2 Stability of remove operation

Concurrent add: Consider the sequential specification of two operations $remove(n_1)$ and $add(n_2, p_2)$. Using (7.26), we get

$$\begin{aligned}
\mathsf{Pre}_{remove(n_1)} &\triangleq n_1 \neq \mathsf{root} \wedge \forall\, n' \in \mathsf{Nodes} \,.\, n' \not\rightarrow n_1 \\
\mathsf{Pre}_{add(n_2,p_2)} &\triangleq n_2 \notin \mathsf{Nodes} \wedge p_2 \in \mathsf{Nodes} \\
\mathsf{Post}_{add(n_2,p_2)} &\triangleq n_2 \in \mathsf{Nodes} \wedge n_2 \rightarrow p_2
\end{aligned}$$

$$\frac{\mathsf{Inv} \wedge \mathsf{Pre}_{remove(n_1)} \wedge \mathsf{Pre}_{add(n_2,p_2)} \wedge \boxed{n_1 \neq p_2} \qquad [\![add(n_2,p_2)]\!]}{\mathsf{Inv} \wedge \mathsf{Post}_{add(n_2,p_2)} \wedge \mathsf{Pre}_{remove(n_1)}} \tag{7.33}$$

We see that the clause that node n_1 has to be a leaf node is not satisfied if $n_1 = p_2$ since add operation introduces a child node under p_2. However, the refined specification of tombstones as described above does not require the node n_1 to be a leaf node. So that solution fixes this conflict as well.

Concurrent remove: Consider the sequential specification of two remove operations $remove(n_1)$ and $remove(n_2)$. Using (7.26), we get

$$\begin{aligned}
\mathsf{Pre}_{remove(n_1)} &\triangleq n_1 \neq \mathsf{root} \wedge \forall\, n' \in \mathsf{Nodes} \,.\, n' \not\rightarrow n_1 \\
\mathsf{Pre}_{remove(n_2)} &\triangleq n_2 \neq \mathsf{root} \wedge \forall\, n' \in \mathsf{Nodes} \,.\, n' \not\rightarrow n_2 \\
\mathsf{Post}_{remove(n_2)} &\triangleq n_2 \notin \mathsf{Nodes}
\end{aligned}$$

$$\frac{\mathsf{Inv} \wedge \mathsf{Pre}_{remove(n_1)} \wedge \mathsf{Pre}_{remove(n_2)} \wedge \boxed{\texttt{true}} \qquad [\![remove(n_2)]\!]}{\mathsf{Inv} \wedge \mathsf{Post}_{remove(n_2)} \wedge \mathsf{Pre}_{remove(n_1)}} \tag{7.34}$$

We see that the remove operation is stable under a concurrent remove. Furthermore, the refined specification is also stable since it adds n_1 and n_2 to TS.

Concurrent move: Consider the sequential specification of two operations $remove(n_1)$ and $move(n_2, p_2')$. Using (7.26), we get

$$\begin{aligned}
\mathsf{Pre}_{remove(n_1)} &\triangleq n_1 \neq \mathsf{root} \wedge \forall\, n' \in \mathsf{Nodes} \,.\, n' \not\rightarrow n_1 \\
\mathsf{Pre}_{move(n_2,p_2')} &\triangleq n_2 \in \mathsf{Nodes} \wedge n_2 \neq \mathsf{root} \wedge p_2' \in \mathsf{Nodes} \wedge p_2' \neq n_2 \wedge p_2' \not\rightarrow^* n_2 \\
\mathsf{Post}_{move(n_2,p_2')} &\triangleq n_2 \rightarrow p_2'
\end{aligned}$$

$$\frac{\mathsf{Inv} \wedge \mathsf{Pre}_{remove(n_1)} \wedge \mathsf{Pre}_{move(n_2,p_2')} \wedge \boxed{n_1 \neq p_2'} \qquad [\![move(n_2,p_2')]\!]}{\mathsf{Inv} \wedge \mathsf{Post}_{move(n_2,p_2')} \wedge \mathsf{Pre}_{remove(n_1)}} \tag{7.35}$$

We see that the clause for the remove operation that n_1 should be a leaf node is violated if a node is moved under it. Again, we can observe that the refined specification of remove eliminates this issue due to the absence of the violation-causing clause.

7.4.1.3 Stability of move operation

Concurrent add: Consider the sequential specification of two operations $move(n_1, p_1')$ and $add(n_2, p_2)$. Using (7.26), we get

$$\begin{aligned}
\mathsf{Pre}_{move(n_1,p_1')} &\triangleq n_1 \in \mathsf{Nodes} \wedge n_1 \neq \mathsf{root} \wedge p_1' \in \mathsf{Nodes} \wedge p_1' \neq n_1 \wedge p_1' \not\rightarrow^* n_1 \\
\mathsf{Pre}_{add(n_2,p_2)} &\triangleq n_2 \notin \mathsf{Nodes} \wedge p_2 \in \mathsf{Nodes} \\
\mathsf{Post}_{add(n_2,p_2)} &\triangleq n_2 \in \mathsf{Nodes} \wedge n_2 \rightarrow p_2
\end{aligned}$$

$$\frac{\mathsf{Inv} \wedge \mathsf{Pre}_{move(n_1,p_1')} \wedge \mathsf{Pre}_{add(n_2,p_2)} \wedge \boxed{\texttt{true}} \qquad [\![add(n_2,p_2)]\!]}{\mathsf{Inv} \wedge \mathsf{Post}_{add(n_2,p_2)} \wedge \mathsf{Pre}_{move(n_1,p_1')}} \tag{7.36}$$

Stability		Stable against concurrent operation		
		$add(n_2, p_2)$	$remove(n_2)$	$move(n_2, p_2')$
Operations	$add(n_1, p_1)$	$n_1 \neq n_2$	$p_1 \neq n_2$	$true$
	$remove(n_1)$	$n_1 \neq p_2$	$true$	$n_1 \neq p_2'$
	$move(n_1, p_1')$	$true$	$p_1' \neq n_2$	$p_1' \not\rightarrow^* n_2$

Table 7.1: Stability analysis of sequential specification

The precondition of move is stable against a concurrent add operation.

Concurrent remove: Consider the sequential specification of two remove operations $move(n_1, p_1')$ and $remove(n_2)$. Using (7.26), we get

$$\begin{aligned}
\mathsf{Pre}_{move(n_1,p_1')} &\triangleq n_1 \in \mathsf{Nodes} \wedge n_1 \neq \mathsf{root} \wedge p_1' \in \mathsf{Nodes} \wedge p_1' \neq n_1 \wedge p_1' \not\rightarrow^* n_1 \\
\mathsf{Pre}_{remove(n_2)} &\triangleq n_2 \neq \mathsf{root} \wedge \forall\, n' \in \mathsf{Nodes} \,\boldsymbol{.}\, n' \not\rightarrow n_2 \\
\mathsf{Post}_{remove(n_2)} &\triangleq n_2 \notin \mathsf{Nodes}
\end{aligned}$$

$$\frac{\mathsf{Inv} \wedge \mathsf{Pre}_{move(n_1,p_1')} \wedge \mathsf{Pre}_{remove(n_2)} \wedge n_2 \neq p_1' \qquad [\![remove(n_2)]\!]}{\mathsf{Inv} \wedge \mathsf{Post}_{remove(n_2)} \qquad \wedge \mathsf{Pre}_{move(n_1,p_1')}} \tag{7.37}$$

Observe here that removing n_2 violates the clause $p_1' \in \mathsf{Nodes}$ if n_2 and p_1' are the same. However, in our refined specification, the postcondition of remove is $n_2 \in \mathsf{TS}$, keeping the clause $p_1' \in \mathsf{Nodes}$ stable.

Concurrent move: Consider the sequential specification of two operations $move(n_1, p_1')$ and $move(n_2, p_2')$. Using (7.26), we get

$$\begin{aligned}
\mathsf{Pre}_{move(n_1,p_1')} &\triangleq n_1 \in \mathsf{Nodes} \wedge n_1 \neq \mathsf{root} \wedge p_1' \in \mathsf{Nodes} \wedge p_1' \neq n_1 \wedge p_1' \not\rightarrow^* n_1 \\
\mathsf{Pre}_{move(n_2,p_2')} &\triangleq n_2 \in \mathsf{Nodes} \wedge n_2 \neq \mathsf{root} \wedge p_2' \in \mathsf{Nodes} \wedge p_2' \neq n_2 \wedge p_2' \not\rightarrow^* n_2 \\
\mathsf{Post}_{move(n_2,p_2')} &\triangleq n_2 \rightarrow p_2'
\end{aligned}$$

$$\frac{\mathsf{Inv} \wedge \mathsf{Pre}_{move(n_1,p_1')} \wedge \mathsf{Pre}_{move(n_2,p_2')} \wedge p_1' \not\rightarrow^* n_2 \qquad [\![move(n_2, p_2')]\!]}{\mathsf{Inv} \wedge \mathsf{Post}_{move(n_2,p_2')} \wedge \mathsf{Pre}_{move(n_1,p_1')}} \tag{7.38}$$

We see here that a concurrent move of p_1 or an ancestor of p_1 invalidates the precondition clause $p_1' \not\rightarrow^* n_1$ that prevents a cycle from forming. This is a subtle condition missed in many previous works [3, 40, 42]; hence it highlights the value of a formal analysis. We discuss this condition in more detail in Section 7.5 and explain how we refine the specification for stability.

Table 7.1 shows the summary of the stability analysis on the sequential specification discussed in Section 7.3. A condition indicates that the precondition of the operation in that row is stable under the operation in the column under the condition.

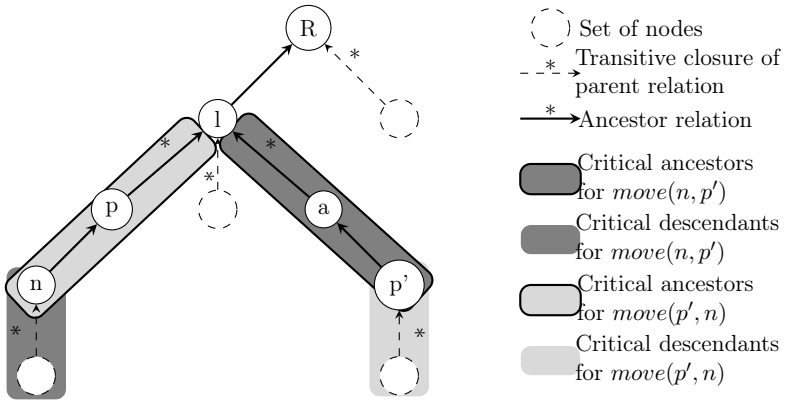


Figure 7.4: Critical ancestors and critical descendants

7.5 Safety of concurrent moves

We return to concurrent moves and study in more detail how a move operation on a remote replica might affect the precondition of a concurrent move in the local replica. Consider a local operation $move(n, p')$. In a sequential execution, precondition clause $p' \not\rightarrow^* n$ forbids moving n under itself (which would cause a cycle). However a concurrent remote move of p' or an ancestor of p' under n will not preserve the precondition of the operation, $p' \not\rightarrow^* n$, resulting in a cycle. We call this move as a *cycle-causing-concurrent-move*.

Observe that the precondition prevents an ancestor of n moving under itself in sequential execution, whereas a concurrent move of the ancestors of p' may result in a cycle. Therefore, only the concurrent move of the ancestors of p' that are not ancestors of n would lead to a cycle. We call this set of ancestors *critical ancestors*, and the set of n and its descendants *critical descendants*. Formally,

$$critical_ancestors(n, p') \triangleq \{a \in \mathsf{Nodes} \,.\, p' \rightarrow^* a \wedge n \not\rightarrow^* a\} \tag{7.39}$$

$$critical_descendants(n, p') \triangleq \{d \in \mathsf{Nodes} \,.\, d \rightarrow^* n\} \tag{7.40}$$

To illustrate critical ancestors and critical descendants, consider two concurrent move operations $move(n, p')$ and $move(p', n)$. Figure 7.4 shows the critical ancestors and critical descendants of both move operations. The node l is the *Least Common Ancestor (LCA)* of n and p', i.e., their common ancestor farthest from the root. The dark gray region with a border represents the critical ancestors of $move(n, p')$; the borderless dark gray region represents their critical descendants. The lighter gray region with and without borders represents the critical ancestors and critical descendants of $move(p', n)$.

A concurrent move operation moving a node from the set of critical ancestors under any critical descendant of a local move would result in a cycle. Note that the set of critical descendants of the local move overlaps with the set of critical ancestors of the remote cycle-causing-concurrent-move. Hence, we consider only the critical ancestors of move operations.

7.5.1 Classifying moves

Let us take a step back and analyze the types of move operations. Some move operations result in a node moving farther away from the root, called *down-move*s, in contrast to those moving it nearer to the root, or to remain at the same distance, called *up-move*s. We define rank as the distance of a node from the root node, as follows:

$$\mathsf{rank}(\mathsf{root}) = 0 \tag{7.41}$$

$$n \to p \wedge n \neq \mathsf{root} \implies \mathsf{rank}(n) = \mathsf{rank}(p) + 1 \tag{7.42}$$

$$\textit{up-move}(n, p') \triangleq \textit{rank}(n) > \textit{rank}(p') \tag{7.43}$$

$$\textit{down-move}(n, p') \triangleq \textit{rank}(n) \leq \textit{rank}(p') \tag{7.44}$$

Consider an up-move operation, $\textit{move}(n, p')$, i.e., $\mathsf{rank}(n) > \mathsf{rank}(p')$. Since critical descendants are descendants of n or n, their rank will be at least the rank of n. Similarly, since critical ancestors are p' or ancestors of p', their rank will be at most the rank of p'. Hence, in the case of an up-move, the rank of a critical descendant will be always greater than the rank of a critical ancestor, i.e.,

$$\begin{aligned}\forall n, p, p', d, a \in \mathsf{Nodes} \,\boldsymbol{.}\, n \to p \wedge \mathsf{rank}(n) > \mathsf{rank}(p') \wedge d \twoheadrightarrow n \wedge p' \twoheadrightarrow a \\ \implies \mathsf{rank}(d) > \mathsf{rank}(a)\end{aligned} \tag{7.45}$$

This implies that a cycle-causing-concurrent-move can only be a down-move. Hence, we have that concurrent up-moves are safe; stability issues can occur only between two concurrent down-moves, or between an up-move and a down-move.

7.5.2 Coordination-free conflict resolution for concurrent moves

Let us now design a coordination-free conflict resolution policy for the moves that conflict. We have two possibilities, either avoid concurrent moves by avoiding concurrency using some coordination technique such as locking, or weaken the specification using a conflict resolution policy. In this work we prefer the later since our aim is to design a coordination-free data structure.

A conflict resolution policy is required if both the concurrent move operations move a node in the set of critical ancestors of the other. The choice of conflict resolution policy is always somewhat arbitrary. We propose the following:

- If the concurrent moves are up-moves, apply both their effects.
- If there is a concurrent up-move and down-move, up-move wins and the down-move is skipped.
- If the concurrent moves are down-moves, deterministically resolve conflict, eg., the operation with the *highest priority* wins. The priority of a move operation can be set by the application, with a condition - priorities are unique [51].

Contrast our approach with the alternative that uses shared-exclusive locks for concurrent moves [3]. Consider concurrent operations $\textit{move}(n, p')$, moving node n under p', and $\textit{move}(p', n)$, moving node p' under n. These operations compete for a lock. The one that

Commutativity		Operations		
		$add(n_2, p_2)$	$remove(n_2)$	$move(n_2, p'_2)$
Operations	$add(n_1, p_1)$	*true*	*true*	*true*
	$remove(n_1)$	*true*	*true*	*true*
	$move(n_1, p'_1)$	*true*	*true*	$\neg(n_1 = n_2 \wedge p'_1 \neq p'_2)$

Table 7.2: Result of commutativity analysis

succeeds first will apply its move, blocking the other. When it releases the lock, this releases the second one, but its precondition is no longer valid and it cannot execute. Thereby, safety is preserved, at the cost of aborting the second move. The present work essentially achieves the same end result, but without the overhead of locking. Our experiments in Chapter 8 show the performance difference.

7.6 Convergence

As discussed in Section 7.2, to ensure convergence concurrent updates should commute [2]. The results of the commutativity analysis is show in Table 7.2.

Add and remove operations result in adding the added and removed node to Nodes and TS respectively. Since set union is commutative, each of these two operations commutes with itself and with the other.

The move operation changes the parent pointer of a node. It commutes with add and remove, since it does not have an effect on set membership.

However, observe that in the sequential specification, two moves do not commute, if the same node is moved to two different places. In Subsection 7.5.2, we have explained a winning operation between a concurrent up-move and down-move and concurrent down-moves. Concurrent up-moves on the same node can be non-commutative though. This issue is fixed by the deciding the priority between the concurrent up-moves (like concurrent down-moves).

7.7 Independence

We use the independence conditions from Subsection 7.2.2.3 to check for safety violations due to tentative moves. We check whether each operation is independent of up-move and down-move since they are the only operations that have tentative effects. For the dependent operations, we compute the condition under which it is dependent and use it to devise dependency resolution policies. In order to compute dependency conditions, we use the dependency analysis proposed by Houshmand and Lesani [9]. An operation op_2 is dependent on op_1 if the execution of op_1 enabled Pre_{op_2} that was not enabled before its execution, i.e.,

$$\frac{\mathsf{Inv} \wedge \mathsf{Pre}_{op_1} \qquad [\![op_1]\!]}{\mathsf{Inv} \wedge \mathsf{Post}_{op_1} \wedge \mathsf{Pre}_{op_2}} \tag{7.46}$$

If the dependency condition evaluates to `true`, then op_2 is independent of op_1. We use this analysis to check the independence of add, remove, up-move and down-move with respect to a historical tentative operation, i.e., an up-move or down-move performed before the operation under observation.

7.7.1 Independence of add operation

Historical up-move: We use Equation 7.7 as follows:

$$\begin{aligned}
\mathsf{Pre}_{up\text{-}move(n_1,p'_1)} &\triangleq n_1 \in \mathsf{Nodes} \wedge n_1 \neq \mathsf{root} \wedge p'_1 \in \mathsf{Nodes} \wedge\ n_1 \neq p'_1 \wedge p'_1 \not\rightarrow^* n_1 \wedge \mathsf{rank}(n_1) > \mathsf{rank}(p'_1) \\
\mathsf{Post}_{up\text{-}move(n_1,p'_1)} &\triangleq \texttt{skip} \vee n_1 \rightarrow p'_1 \\
\mathsf{Pre}_{add(n_2,p_2)} &\triangleq p_2 \in \mathsf{Nodes} \wedge n_2 \notin \mathsf{Nodes}
\end{aligned}$$

Since the historical up-move doesn't change the membership of Nodes, we can see that add is independent of up-move.

Historical down-move: An add operation is independent of a historical down-move in the same manner because it does not change the membership of Nodes either.

7.7.2 Independence of remove operation

Historical up-move: For checking the independence of remove, Equation 7.7 becomes:

$$\begin{aligned}
\mathsf{Pre}_{up\text{-}move(n_1,p'_1)} &\triangleq n_1 \in \mathsf{Nodes} \wedge n_1 \neq \mathsf{root} \wedge p'_1 \in \mathsf{Nodes} \wedge\ n_1 \neq p'_1 \wedge p'_1 \not\rightarrow^* n_1 \wedge \mathsf{rank}(n_1) > \mathsf{rank}(p'_1) \\
\mathsf{Post}_{up\text{-}move(n_1,p'_1)} &\triangleq \texttt{skip} \vee n_1 \rightarrow p'_1 \\
\mathsf{Pre}_{remove(n_2)} &\triangleq n_2 \neq \mathsf{root}
\end{aligned}$$

Since $n_2 \neq \mathsf{root}$ is unaffected by a historical up-move, remove is independent of up-move.

Historical down-move: Similarly to historical up-move, a historical down-move also has no impact of the precondition of a remove operation. Hence remove is independent of a historical down-move.

7.7.3 Independence of up-move operation

Historical up-move: Now we analyse whether an up-move is independent of a historical up-move.

$$\begin{aligned}
\mathsf{Pre}_{up\text{-}move(n_1,p'_1)} &\triangleq n_1 \in \mathsf{Nodes} \wedge n_1 \neq \mathsf{root} \wedge p'_1 \in \mathsf{Nodes} \wedge\ n_1 \neq p'_1 \wedge p'_1 \not\rightarrow^* n_1 \wedge \mathsf{rank}(n_1) > \mathsf{rank}(p'_1) \\
\mathsf{Post}_{up\text{-}move(n_1,p'_1)} &\triangleq \texttt{skip} \vee n_1 \rightarrow p'_1 \\
\mathsf{Pre}_{up\text{-}move(n_2,p'_2)} &\triangleq n_2 \in \mathsf{Nodes} \wedge n_2 \neq \mathsf{root} \wedge p'_2 \in \mathsf{Nodes} \wedge\ n_2 \neq p'_2 \wedge p'_2 \not\rightarrow^* n_2 \wedge \mathsf{rank}(n_2) > \mathsf{rank}(p'_2)
\end{aligned}$$

We first divide the postcondition of the historical up-move into two parts: on the one hand, `skip`, which leaves the state as it was; and on the other hand, $n_1 \rightarrow p'_1$, which changes the parent relation. Then we divide the precondition of the second up-move into two parts, $n_2 \in \mathsf{Nodes} \wedge n_2 \neq \mathsf{root} \wedge p'_2 \in \mathsf{Nodes} \wedge\ n_2 \neq p'_2$, which is unaffected by the

historical up-move, and $p_2' \not\rightarrow^* n_2 \wedge \mathsf{rank}(n_2) > \mathsf{rank}(p_2')$, which is potentially effected by the second part of the postcondition of the historical up-move.

Note that $\mathsf{Pre}_{up\text{-}move(n_2,p_2')}$ was not enabled before the execution of op_1, i.e., the execution of op_1 enabled at least one predicate $p_2' \not\rightarrow^* n_2$ or $\mathsf{rank}(n_2) > \mathsf{rank}(p_2')$. Let us consider them one at a time.

Let us derive the conditions under which moving a node to a different parent introduces an ancestor relation that enables the condition $p_2' \not\rightarrow^* n_2$ (it was previously disabled). This means that the historical up-move operation caused a disconnection between p_2' and n_2. This will happen only if the node being moved by the historical up-move was either n_2 or a descendant of n_2 and the new parent of the current move was either n_1 or a descendant of n_1 (the node moved by the historical up-move). Hence we have $(n_1 = n_2 \vee n_1 \rightarrow^* n_2) \wedge (p_2' = n_1 \vee p_2' \rightarrow^* n_1)$.

The condition $\mathsf{rank}(n_2) > \mathsf{rank}(p_2')$ will be enabled after an up-move only if $\mathsf{rank}(p_2')$ decreased.[8] This will happen only if p_2' was the node moved or its descendant. Hence we have that $p_2' = n_1 \vee p_2' \rightarrow^* n_1$.

The historical up-move either enabled one or both of the conditions. Combining them gives $p_2' = n_1 \vee p_2' \rightarrow^* n_1$, the condition under which an up-move, $up\text{-}move(n_2, p_2')$, is dependent on a historial up-move, $up\text{-}move(n_1, p_1')$.

Historical down-move: To check for independence of an up-move with a historical down-move, we have the following condition:

$$\mathsf{Pre}_{down\text{-}move(n_1,p_1')} \triangleq n_1 \in \mathsf{Nodes} \wedge n_1 \neq \mathsf{root} \wedge p_1' \in \mathsf{Nodes} \wedge\ n_1 \neq p_1' \wedge p_1' \not\rightarrow^* n_1 \wedge \mathsf{rank}(n_1) \leq \mathsf{rank}(p_1')$$
$$\mathsf{Post}_{down\text{-}move(n_1,p_1')} \triangleq \texttt{skip} \vee n_1 \rightarrow p_1'$$
$$\mathsf{Pre}_{up\text{-}move(n_2,p_2')} \triangleq n_2 \in \mathsf{Nodes} \wedge n_2 \neq \mathsf{root} \wedge p_2' \in \mathsf{Nodes} \wedge\ n_2 \neq p_2' \wedge p_2' \not\rightarrow^* n_2 \wedge \mathsf{rank}(n_2) > \mathsf{rank}(p_2')$$

We apply the same reasoning as in the previous case for the condition $p_2' \not\rightarrow^* n_2$, obtaining $(n_1 = n_2 \vee n_1 \rightarrow^* n_2) \wedge (p_2' = n_1 \vee p_2' \rightarrow^* n_1)$ as the condition under which an up-move is dependent under a historical down-move.

There is a difference in the second part though; the condition $\mathsf{rank}(n_2) > \mathsf{rank}(p_2')$ will be enabled after a down-move only if $\mathsf{rank}(n_2)$ increases (not possible for a down-move to decrease the rank). This will happen only if n_2 was the node moved or its descendant, i.e., $n_2 = n_1 \vee n_2 \rightarrow^* n_1$.

Combining both the conditions, we have $((n_1 = n_2 \vee n_1 \rightarrow^* n_2) \wedge (p_2' = n_1 \vee p_2' \rightarrow^* n_1)) \vee (n_2 = n_1 \vee n_2 \rightarrow^* n_1)$ as the condition under which an up-move, $up - move(n_2, p_2')$, is dependent on a historial down-move, $down\text{-}move(n_1, p_1')$.

[8] Note that $\mathsf{rank}(n_2)$ cannot increase since an up-move does not cause the rank of any move to increase.

7.7.4 Independence of down-move operation

Historical up-move: The pre and postconditions required to analyse the dependence of a down-move operation under a historical up-move is as follows:

$$\mathsf{Pre}_{up\text{-}move(n_1,p'_1)} \triangleq n_1 \in \mathsf{Nodes} \wedge n_1 \neq \mathsf{root} \wedge p'_1 \in \mathsf{Nodes} \wedge\ n_1 \neq p'_1 \wedge p'_1 \not\rightarrow^* n_1 \wedge \mathsf{rank}(n_1) > \mathsf{rank}(p'_1)$$
$$\mathsf{Post}_{up\text{-}move(n_1,p'_1)} \triangleq \mathtt{skip} \vee n_1 \rightarrow p'_1$$
$$\mathsf{Pre}_{down\text{-}move(n_2,p'_2)} \triangleq n_2 \in \mathsf{Nodes} \wedge n_2 \neq \mathsf{root} \wedge p'_2 \in \mathsf{Nodes} \wedge\ n_2 \neq p'_2 \wedge p'_2 \not\rightarrow^* n_2 \wedge \mathsf{rank}(n_2) \leq \mathsf{rank}(p'_2)$$

Note that the reasoning for the up-move operation also remains valid here since the effect of both moves are the same, only their preconditions differ, only the clause comparing the ranks of the node and the new parent differs. The first part of the dependency condition remains, $(n_1 = n_2 \vee n_1 \rightarrow^* n_2) \wedge (p'_2 = n_1 \vee p'_2 \rightarrow^* n_1)$.

The condition $\mathsf{rank}(n_2) \leq \mathsf{rank}(p'_2)$ will be effected only if the historical up-move decreased the rank of n_2. Hence we have the condition $n_2 = n_1 \vee n_2 \rightarrow^* n_1$.

Combining the clauses, we have $((n_1 = n_2 \vee n_1 \rightarrow^* n_2) \wedge (p'_2 = n_1 \vee p'_2 \rightarrow^* n_1)) \vee (n_2 = n_1 \vee n_2 \rightarrow^* n_1)$, the condition under which a down-move is dependent on a historical up-move.

Historical down-move: We consider the following pre and postconditions:

$$\mathsf{Pre}_{down\text{-}move(n_1,p'_1)} \triangleq n_1 \in \mathsf{Nodes} \wedge n_1 \neq \mathsf{root} \wedge p'_1 \in \mathsf{Nodes} \wedge\ n_1 \neq p'_1 \wedge p'_1 \not\rightarrow^* n_1 \wedge \mathsf{rank}(n_1) \leq \mathsf{rank}(p'_1)$$
$$\mathsf{Post}_{down\text{-}move(n_1,p'_1)} \triangleq \mathtt{skip} \vee n_1 \rightarrow p'_1$$
$$\mathsf{Pre}_{down\text{-}move(n_2,p'_2)} \triangleq n_2 \in \mathsf{Nodes} \wedge n_2 \neq \mathsf{root} \wedge p'_2 \in \mathsf{Nodes} \wedge\ n_2 \neq p'_2 \wedge p'_2 \not\rightarrow^* n_2 \wedge \mathsf{rank}(n_2) \leq \mathsf{rank}(p'_2)$$

We use the reasoning as in the previous cases on these ang get $p'_2 = n_1 \vee p'_2 \rightarrow^* n_1$, the condition under which a down-move, $down\text{-}move(n_2, p'_2)$, is dependent on a historial down-move, $down\text{-}move(n_1, p'_1)$.

We see that up-move and down-move operations are dependent on each other and add and remove are independent of up-move and down-move. We also derived the conditions under which up-moves and down-moves are dependent on each other. We use this information to design dependence resolution policies.

7.8 Safe specification of a replicated tree

We incorporate the stability, commutativity, and independence analysis results and the design refinements, resulting in the coordination-free, safe and convergent replicated tree data structure specified in Specification 7.1. The state now consists of a set of nodes, Nodes, and tombstones, TS. Since the tombstones also form part of the tree, they also have to maintain the tree structure. The invariants refer to the set of nodes which includes tombstones.

We also introduce some definitions to help define the coordination-free and conflict-free up-move and down-move operations. We define an operation as a tuple consisting of its type (add, remove, up-move or down-move), its parameters, and its priority. The priority is arbitrary (e.g. supplied by the application); the only condition being that priorities

Specification 7.1 Concurrent specification of Maram

State: $\mathsf{Nodes} \times \mathsf{TS}$

Invariant:

$$\mathsf{root} \to \mathsf{root} \wedge \forall n \in \mathsf{Nodes}.\ \mathsf{root} \not\to n \wedge \mathsf{root} \notin \mathsf{TS} \qquad (Root)$$

$$\wedge\ \forall n \in \mathsf{Nodes}.\ n \neq \mathsf{root} \wedge \exists p \in \mathsf{Nodes}\,.\, n \to p \qquad (Parent)$$

$$\wedge\ \forall n, p, p' \in \mathsf{Nodes}\,.\, n \to p \wedge n \to p' \implies p = p' \qquad (Unique)$$

$$\wedge\ \forall n \in \mathsf{Nodes}\,.\, n \neq \mathsf{root} \implies n \to^* \mathsf{root} \qquad (Reachable)$$

Add operation:

$$\frac{\mathsf{Inv} \wedge p \in \mathsf{Nodes} \wedge n \notin \mathsf{Nodes} \qquad [\![add(n,p)]\!]}{\mathsf{Inv} \wedge n \in \mathsf{Nodes} \wedge n \to p} \quad (\textsc{Add-Operation})$$

Remove operation:

$$\frac{\mathsf{Inv} \wedge n \neq \mathsf{root} \qquad [\![remove(n)]\!]}{\mathsf{Inv} \wedge n \in \mathsf{TS}} \quad (\textsc{Remove-Operation})$$

Definitions:

$$\begin{aligned}
operation &\triangleq (type, params, priority)\\
\mathtt{C} &\triangleq set\ of\ concurrent\ operations\\
\mathtt{H} &\triangleq history\ of\ operations\ available\ at\ the\ origin\ replica\\
crit\text{-}anc\text{-}overlap(op_1, op_2) &\triangleq op_1.params.n \in critical_ancestor(op_2) \wedge\\
&\quad op_2.params.n \in critical_ancestor(op_1)\\
self\text{-}or\text{-}under(n) &\triangleq \{n' | n' = n \vee (n' \in \mathsf{Nodes} \wedge n' \to^* n)\}
\end{aligned}$$

Move operation:

$$\frac{\begin{array}{c}\mathsf{Inv} \wedge n \in \mathsf{Nodes} \wedge n \neq \mathsf{root} \wedge p' \in \mathsf{Nodes} \wedge\ n \neq p' \wedge p' \not\to^* n \wedge \mathsf{rank}(n) > \mathsf{rank}(p')\\ [\![up\text{-}move(n,p')]\!]\end{array}}{\begin{array}{l}
\nexists op \in \mathtt{C}\,.\, op.type = up\text{-}move \wedge\ op.params.n = n \wedge\ op.priority > priority\\
\nexists op \in \mathtt{H}\,.\, (op.type = up\text{-}move \wedge\ \ p' \in self\text{-}or\text{-}under(op.params.n))\\
\qquad \vee (op.type = down\text{-}move \wedge\ \ (n \in self\text{-}or\text{-}under(op.params.n)\\
\qquad\qquad \vee\ (op.params.n \in self\text{-}or\text{-}under(n)\\
\qquad\qquad\qquad \wedge\ p' \in self\text{-}or\text{-}under(op.params.n)))) \qquad \implies \mathsf{Inv} \wedge n \to p'
\end{array}} \quad (\textsc{Up-move-Operation})$$

$$\frac{\begin{array}{c}\mathsf{Inv} \wedge n \in \mathsf{Nodes} \wedge n \neq \mathsf{root} \wedge p' \in \mathsf{Nodes} \wedge\ n \neq p' \wedge p' \not\to^* n \wedge \mathsf{rank}(n) \leq \mathsf{rank}(p')\\ [\![down\text{-}move(n,p')]\!]\end{array}}{\begin{array}{l}
\nexists op \in \mathtt{C}\,.\, op.type = up\text{-}move\\
\qquad \wedge (crit\text{-}anc\text{-}overlap(down\text{-}move(n,p'), op) \vee\ op.params.n = n)\\
\wedge \nexists op \in \mathtt{C}\,.\, op.type = down\text{-}move\\
\qquad \wedge (crit\text{-}anc\text{-}overlap(down\text{-}move(n,p'), op) \vee\ op.params.n = n)\\
\qquad \wedge op.priority > priority\\
\nexists op \in \mathtt{H}\,.\, (op.type = up\text{-}move \wedge\ \ (n \in self\text{-}or\text{-}under(op.params.n)\\
\qquad\qquad \vee\ (op.params.n \in self\text{-}or\text{-}under(n)\\
\qquad\qquad\qquad \wedge\ p' \in self\text{-}or\text{-}under(op.params.n))))\\
\qquad \vee (op.type = down\text{-}move \wedge\ \ p' \in self\text{-}or\text{-}under(op.params.n)) \qquad \implies \mathsf{Inv} \wedge n \to p'
\end{array}} \quad (\textsc{Down-move-Operation})$$

are totally ordered. We define `C` as the set of operations concurrent with the operation under consideration. `H` is the set of operations seen by the current operation. We also define operations on critical ancestors as *crit-anc-overlap*, where the node being moved is a member of the set of critical ancestors of the other operation. *self-or-under* indicates the node itself and its descendants.

With the help of these definitions, we define the up-move and down-move operations in three parts: the actual precondition needed to ensure sequential safety, the conflict resolution condition (highlighted in light blue), the dependency condition (highlighted in dark blue), and the update on the state. Note that the conflict resolution and dependency checks are performed while applying the effect of the operation on the local and remote replicas, while the precondition is checked only at the local replica.

7.8.1 Mechanized verification of the concurrent specification

We use the CISE3 plug-in, presented in Subsection 7.3.4, to identify conflicts as shown in Table 7.1 and Table 7.2. Given the sequential specification from Section 7.3, CISE3 generates proof obligations to check stability and commutativity of executing pairs of operations.

Provable concurrent execution

We update the sequential specification of Section 7.3 by adding the conflict resolution policies from Section 7.8. For example, we place the additional precondition that added nodes are unique onto the add operation:

```
assume { ... ∧ n1 ≠ n2 }
```

We refine the definition of type `state` to include tombstones, as follows:

```
type state = { mutable nodes: fset elt; ...;
      mutable tombstones: fset elt; }
```

We update the specification of `rem` operation accordingly:

```
val rem (n : elt) (s : state) : unit
  ensures  { s.tombstones = add n (old s).tombstones }
```

where `add` stands for the logical adding operation on sets.

Verifying the stability of `move` operation with itself need extra information on a set of concurrent operations. We also update the `state` type definition to include ranking and critical ancestors information.

Finally, 55 verification conditions are generated for the implementation and given specification of `move_refined`. All of these are automatically verified, using a combination of SMT solvers. The specification and the proof results are available at Meirim et al. [49].

Independent		Under		
		$add(n_2, p_2)$	$remove(n_2)$	$move(n_2, p'_2)$
Operation	$add(n_1, p_1)$	$p_1 \neq n_2$	$true$	$true$
	$remove(n_1)$	$n_1 \neq n_2$	$true$	$true$
	$move(n_1, p'_1)$	$n_1 \neq n_2 \vee p'_1 \neq n_2$	$true$	$n_2 \notin self\text{-}or\text{-}under(n_1)$ $\vee p'_1 \notin self\text{-}or\text{-}under(n_2)$

Table 7.3: Result of dependency analysis. The cell shows the condition under which the operation in the row is independent of the operation in the column.

7.9 Discussion

7.9.1 Moving from causal consistency to eventual consistency

Houshmand and Lesani [9] propose dependency analysis to help relax the requirement of causal delivery. We run this analysis for all operations, irrespective of whether the update is tentative or definitive.

Table 7.3 shows the results of the dependency analysis of Maram. We can observe that no operations are dependent on remove, and add and remove are not dependent on move. As there is no fully independent operation, relaxing causal delivery is not helpful to Maram.

7.9.2 Message overhead for conflict resolution

In order to use Maram in a real-world application, we need to understand the overhead of conflict resolution. Conflict resolution requires some meta information that is sent along with the update message from the origin replica. This may have an impact on the bandwidth lost, hence understanding the components is important.

The conflict resolution policy of Maram needs information to compute a set of concurrent operations. Assuming a replica works as a single threaded process, we use vector clocks. The size of vector clocks is linear with the number of replicas. This poses an additional overhead.

Conflict resolution also takes as input the set of critical ancestors, descendants, and the priority.The size of the set of critical ancestors depends on the depth of the subtree comprising the least common ancestor of the node being moved and the destination parent. The size of the set of critical ancestors is linear to the difference in the rank of the new parent and the least common ancestor. The size of the set of descendants might be large for the nodes nearer to the root. This poses an overhead on the metadata. The priority can be a single number or a string and is independent of other factors. Hence using the conflict resolution of Maram will cause a considerable overhead on message delivery.

The time taken to compute this metadata is the difference between the response time of a naïve unsafe replicated tree and Maram in Figure 8.1c.

7.9.3 Computing the set of concurrent moves

Maram requires a set of concurrent operations to apply the conflict resolution. For this, the Maram system layer does not busy-wait. Every replica makes progress locally, without waiting to receive remote logs (availability under partition). Conflict resolution applies only after a replica receives a concurrent conflicting operation.

To conclude, Maram is a safe, coordination-free replicated tree, designed using conflict resolution policies.

Chapter 8

Experimental study and Comparison

In this section, we conduct an evaluation to showcase the high availability of Maram.

We measure availability in two parts *response time* and *stabilization time*. The first metric, *response time*, is the time taken to log and acknowledge a client request. Recall that the effect of a move operation in our specification consists of either updating the state, or a skip. The update remains tentative until any conflict is resolved, i.e., until all concurrent updates have been delivered. In order to measure this, we introduce a metric called *stabilization time*. Stabilization time measures the duration for which an update is in a transient state.

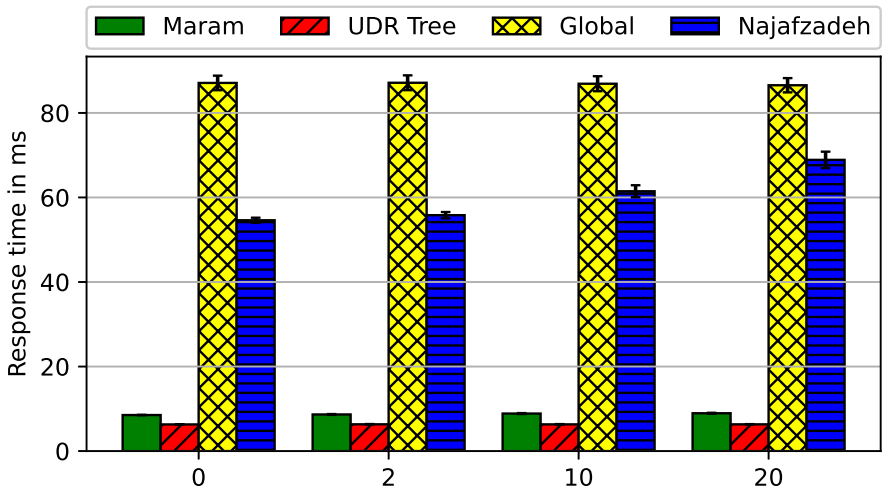
We run the experiments with three replicas connected in a mesh with a FIFO connection. We simulate different network latencies, as shown in Table 8.1. We run the experiments on DELL PowerEdge R410 machine with 64 GB RAM, 24 cores @ 2.40GHz Intel Xeon E5645 processor and Linux kernel version 3.16.0-4-amd64.

We use a warm-up workload to initialize the tree before each experiment. It consists of 1056 operations (1012 add, 15 remove, and 30 move operations) to create a tree with 997 nodes, including the root. We then run concurrent workloads on the three replicas, varying conflict rates at 0%, 2%, 10%, and 20%. The conflicting methods are a percentage of the total concurrent workload. For each run, we submit 250 concurrent operations to each replica (750 operations in total), broken into 60% add, 12% remove, and 28% move operations. Half of the moves are up-moves and the remaining half down-moves.

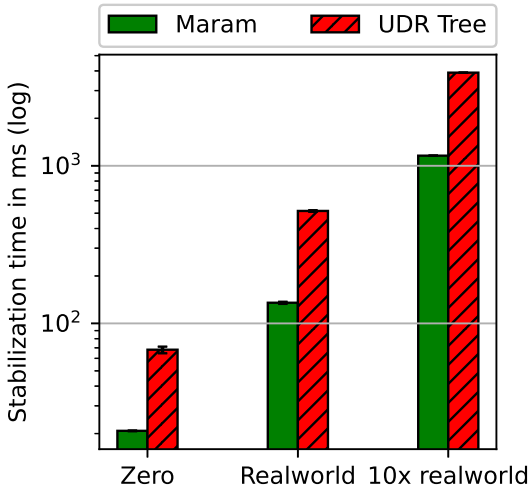
We compare Maram with three solutions from the literature: (i) UDR tree (short for Undo-Do-Redo tree) [42]; (ii) all move operations acquiring a global lock (Global); and, (iii) move operations acquiring read locks on critical ancestors and write lock on the moving

	Latency	Replicas								
		Configuration 1			Configuration 2			Configuration 3		
		P	B	NY	P	B	NY	P	B	NY
Replicas	Paris (P)	0	0	0	0	144	75	0	1440	750
	Bangalore (B)	0	0	0	144	0	215	1440	0	2150
	New York (NY)	0	0	0	75	215	0	750	2150	0

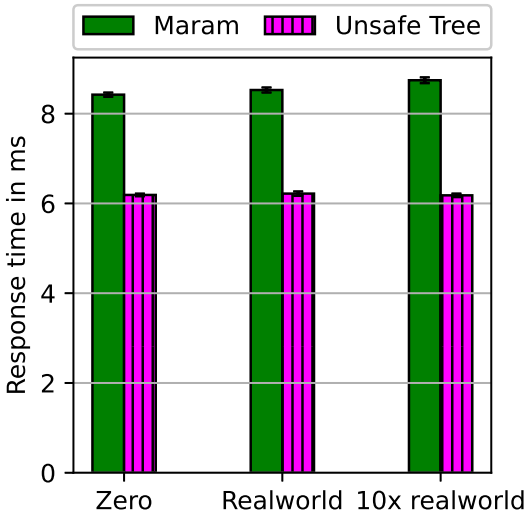
Table 8.1: Latency configurations in ms



(a) Response time for different conflict rates



(b) Stabilization time for different latencies (logarithmic scale)



(c) Overhead introduced by conflict resolution policy

Figure 8.1: Experimental results. Each bar is the average of 15 runs. The error bars show standard deviation.

node (Najafzadeh) [3].

We have implemented the operations of Maram as generator-effector pairs. The client request is received by the replica. The replica computes the local state from the known list of logs.[1] If the local state satisfies the precondition, the generator function generates an effector function. The effector function is essentially the predicate under the line for each operation in our specification. The replica adds the effector function to the list of logs in the local replica, broadcasts the effector function asynchronously to other replicas and acknowledges the client.

The average response time for each design for different conflict rates with latency configuration 2 (Table 8.1) are shown in Figure 8.1a. Observe that Maram and UDR tree have lower response times than the lock-based designs, owing to the coordination-free design. The slight increase in the response time of Maram compared to UDR tree is due to the calculation time for the meta data required for conflict resolution. The response time for Najafzadeh [3] increases with conflict rate, due to lock contention, whereas that of Global is the same across all conflict rates since the proportion of lock-acquiring-moves remains the same.

Figure 8.1b shows the average stabilization time for our design and the UDR tree design [42] on a logarithmic scale for different latency configurations. Our solution gives lower stabilization time, since only moves have transient state, whereas for a UDR tree [42] all operations are in transient state until a local replica asserts that there are no more concurrent operations. Note here that Maram's stabilisation time does not depend on conflict rate, but only on the proportion of down-moves in the workload. As this proportion tends towards 100%, the stabilisation time of Maram tends to be the same as that of UDR.

Next, we run an experiment to measure the overhead introduced by the conflict resolution policy. As a lower bound, we compare the response time of Maram with a naïve unsafe implementation, that uses a simple eventual consistency approach, and thus is not safe. Figure 8.1c shows the response time of both the designs. Maram has a slight overhead in exchange for safety.

[1] The state of the replica is generated dynamically when a client issues a request.

Chapter 9

Related work

Several authors addressed the problem of designing a replicated tree minimizing coordination. Martin et al. [52] compare several possible designs. They use set CRDTs to construct replicated trees with different conflict semantics. Their design supports add and remove operations. However they do not consider move operations.

Kumar and Satyanarayanan [53] discuss the issues with move operations in Coda file system. They have omitted transparent resolution of move operations; instead they mark them as conflicting operations. In contrast to our approach where we automatically resolve conflicts using conflict resolution, they require the user to intervene. Moreover, they haven't discussed about cycle-causing-concurrent-moves.

Bjørner [41] discusses the development of Distributed File System Replication (DFS-R). The author identifies the issues with concurrent move using a model checker. Bjørner [41] identifies several possible solutions including, moving the conflicting nodes to the root or the least common ancestor, or reverting the conflicting nodes parent back to its previous parent. The author noted that the model checker had state space explosion when trying to analyse the algorithm.

Dropbox[54] identifies the issue with concurrent moves. They introduce a synchronisation service between client and server with strong consistency. This design allows atomic safe moves, at the expense of coordination.

Najafzadeh et al. [3] provide a design of a replicated tree for a file system. Our concept of critical ancestors and least common ancestor are inspired by Najafzadeh et al. [3]. The solution requires acquiring read locks on the critical ancestors and a write lock on the node being moved. They have verified their design using CISE principle [6] similar to us. Chapter 8 shows the performance impact on the response time for each client request.

Tao et al. [40] propose a replicated tree with a coordination-free move operation. They achieve it by implementing each move as a non-atomic copy and delete. An atomic update provides all or no guarantee, i.e., either the update is applied or it is not. Ensuring atomicity avoids partial execution of updates. Being non-atomic, it might lead to having multiple copies of the same node.

Compared to the above solutions, our design supports an atomic move operation without coordination.

Kleppmann et al. [42] propose the UDR tree that supports atomic move operations,

based on the concept of opsets. Opsets eventually totally order all operations (in a log), this is more expensive than our solution based on partial order, taking more time for all operations to stabilize. When a replica receives an operation that has to be inserted into a log, the operations until the point of insertion from the end of the log are undone, the inserted operation is performed and all the undone operations are redone. For each concurrent operation received, this might be performed, resulting in undoing and redoing the same set of operations multiple times. They require a delivery layer that is only eventually consistent, whereas our solution requires causal delivery.

The main difference between the work of Kleppmann et al. [42] with Maram is that we formalize the conditions under which a move might skip - both due to a concurrent move or due to a historic move.

Kaki et al. [55] define the concept of Mergeable Replicated Data Types (MRDTs), inspired from three-way-merge. The safety of an MRDT binary tree depends on the labeling of the child-parent relations (whether it belongs to the right or left of the ancestor). It also requires keeping track of all the ancestor relations apart from the parent-child relations. A generic MRDT tree can be considered as an extension to the MRDT binary tree, but requires tracking all ancestor relations and a complex lexicographical ordering when concretizing the merged result. This would increase the metadata overhead. Moreover, identifying the unsafe move operation would be helpful for the user to perform any compensation. This is not possible with MRDTs. In contrast to MRDTs, our approach is based on two-way merge. This saves us from preserving the last common history of two replicas that are broadcasting their updates.

Chapter 10

Conclusion of Part II and Future work

This part of the thesis presents the design of a coordination-free, safe, convergent and highly available replicated tree data structure, Maram. We study the cycle-causing-concurrent-moves, classify them into up-moves and down-moves, and present a conflict resolution for conflicting moves. We study the effect of the conflict resolution on the dependant operations. We provide arguments for safety and convergence of Maram, and experimentally demonstrate the efficiency of the design, by comparing with the existing solutions.

This illustrates the methodology of designing a distributed application without any coordination, thanks to conflict resolution policies and dependency conditions.

10.1 Future work

The future work of this part of the thesis has two main directions. The first direction concerns the design of a replicated tree. The second direction is more general, concerning the design of similar data structures or distributed applications.

Design of a replicated tree: There are different venues to improve the work in this direction. The current design is based on operation-based update propagation model. A useful next step would be a design that provides the same result as Maram supporting state-based and delta-based update propagation.

The next step is reducing the metadata due to the tombstones and keeping the historical move operations. Ideally, one will remove the tombstone (also truncate the set of historical moves) at some safe time in the future; this is non-trivial [50] and is future work of this thesis.

Further future work is to study the effect of combining moves with conflict resolution policies with lock-based moves. This might be required to ensure that a move is definitive. For example, if a program expects a particular directory structure during execution, a concurrent move might crash the program. To avoid this, we should be able to specify some move operations as definitive, thus acquiring locks.

Design of coordination-free distributed applications: As we saw in Section 7.7, tentative operations might effect future operations that are dependent on it. In this part we formulated the notion of independence analysis, but we haven't provided soundness proof for the check. The proof rule should be sound for applications having tentative updates.

Once we have the soundness of the proof rule, we can automate the reasoning with the help of Why3 and prove the dependency conditions mechanically.

Furthermore, the independence analysis discussed in Section 7.7 is valid only when the effect of an update is either apply or skip, i.e., it either changes the state of the replica or leaves the replica state untouched. This proof rule is not adequate for the case where an effector might have two different state changes depending upon the state where the effect is applied. This is another possible line of research.

Part III

Selecting Distributed Concurrency Control

Introduction to Part III

Part I studied a proof technique for verifying the safety of distributed applications. The proof tool returns a list of conflicting methods, which would result in an unsafe state if executed concurrently.

Accordingly, the developer has two choices to ensure concurrent safety — the optimistic approach, designing conflict resolution algorithms (as illustrated in Part II), and the pessimistic approach. The former implies that a user might see intermediate results that might be conflicting with another concurrent operation, and would require conflict reconciliation. In short, a user could observe stale data and it would incur some costs.

In some applications, seeing stale data might be acceptable, whereas in others it might incur extremely high costs. When the cost of stale data is prohibitive, the developer needs to synchronize.

To illustrate divergence and compensation, consider for instance an online auction. The auction platform guarantees that the highest bid wins. Alice from Adelaide puts up a painting for auction. Bob from Brussels and Charles from Calgary quote $100 and $105 respectively. Alice observes the bid from Bob, but not Charles's due to a communication delay. She closes the auction and the application declares Bob as the winner. When the connection is restored, Alice observes that the lower bid won. The optimistic approach corrects itself, apologizes and compensates the participants. The pessimistic approach prevents this from happening by coordinating bid placement and closing an auction.

Having identified the conflicts, e.g., by static analysis [6, 56–59], the required coordination can be implemented in many ways, trading overhead against parallelism. The design space is multi-dimensional: locks can have various levels of granularity; different types of lock can be used (for example, mutex vs. shared/exclusive locks); the placement of the lock object has a significant impact. [1]

Furthermore, the performance of an option depends on the workload and specifically on the frequency distribution and geolocation of the conflicting operations. For instance, in the auction example, placing bids is a frequent operation and can be concurrent, whereas closing an auction is infrequent; therefore the protocol should optimise for the former, even at the expense of the latter. An appropriate protocol would allow place-bid in parallel under a lock in shared mode, whereas close-auction uses exclusive mode. Coordinating multiple auctions with a single lock would disallow concurrent close-auctions, but may decrease overhead, depending on workload.

[1] We focus on locking as the coordination primitive. We believe that our reasoning can be extended with more dimensions to alternative primitives, such as leases, consensus, broadcast or multicast.

To systematize the dimensions of coordination, we construct a *coordination lattice*. It allows us to systematically navigate the concurrency control dimensions of granularity, mode, and placement. The lattice structure is derived from the granularity dimension. Each element of the lattice consists of one or more two-dimensional planes — mode and placement. The choice of granularity affects the cost of both lock acquisition and of lock contention (the coarser the lock, the lesser the cost of acquisition, but the higher the contention). Placement affects only the lock acquisition costs, and mode affects lock contention. Therefore the major dimension of the coordination lattice is granularity with mode and placement as secondary dimensions. Navigating any dimension has an impact on the overhead of locking.

We propose a systematic approach to the design of correct coordination protocols, enabling the designer to select one according to performance metrics, with the following contributions:

- A formalisation of the design space of correct coordination configurations, as a *coordination lattice*. Based on the application's conflict graph, the lattice develops along three design dimensions: granularity, mode and placement.
- A set of metrics to navigate the cost of different coordination configurations with respect to a given workload.
- A tool for evaluating and comparing the performance of each point in the lattice with respect to the workload model.
- Illustration of the above using representative examples.

Chapter 11 studies the design space with the *coordination lattice* with a set of metrics. Chapter 12 provides a set of experiments that illustrates the coordination lattice and the associated metrics, along with the experimental setup.

Chapter 11

Exploring the coordination lattice

In this chapter we explain the construction and navigation of a *coordination lattice* for selecting an optimal distributed lock.

11.1 System Model

In this section we discuss the semantics of a distributed application, the underlying system on which the application runs, and the characteristics of the expected workload. We then see how these factors effect the choice of concurrency control configurations.

11.1.1 Application model

A distributed application consists of a set of operations operating on shared data *replicated* over geo-distributed *locations*. The application is considered *safe* when it respects some given invariant on the shared data in all executions. We assume that every operation is safe when run in isolation, i.e., transitions from a safe state (one where the replica satisfies the invariant) to another safe state. A pair of operations *conflicts* if their concurrent execution might violate the invariant. For instance, in our auction example, operations include create-auction, place-bid or close-auction. Each such operation satisfies the invariant that an open auction has no winner, and that the winner of a closed auction is the highest bidder. Since close-auction designates the winner, and place-bid may change the highest bid, they conflict.

Verification methods are available to analyze the specification of a distributed application [6, 11]. The tools that implement these proof rules gives us the pairs of conflicting methods [7, 8, 12, 60].

Inspired by the work of Houshmand and Lesani [9], we represent the list of conflicts as a *conflict graph*. A vertex of the graph represents an operation and an edge represents a conflict between the corresponding operations. A conflict graph might contain disjoint subgraphs. All the operations that conflict with some operation are included in the same subgraph as that operation.

Whether two operations conflict or not may depend on a predicate, called the *conflict condition*. In our auction example, two updates conflict only if they involve the same auc-

tion; furthermore, close-auction conflicts only with a concurrent place-bid with a higher bidder. The tool by Nair and Shapiro [60] generates conflict conditions from the counterexamples provided by the underlying verification tool. Under these conditions, the operations should coordinate with each other. In this part of the thesis, we realise this coordination using distributed locks. We generate a lock for each conflict based on the parameters involved in the conflict condition. A lock is subscripted by the operations acquiring it and the corresponding coordination condition (after hyphen). This gives us a set of fine-grained locks for the conflict graph.

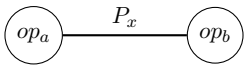


Figure 11.1: Two conflicting operations.

For example, consider two operations op_a and op_b conflicting when the parameter x of both the operations are equal as shown in Figure 11.1. This means that the two operations need to coordinate when operating on the same value of parameter x. If op_a is called with signature $op_a(x_1)$ we create the lock l_{ab-x_1} that is shared with $op_b(x_1)$.

We call *service time* as the time between lock acquire and release, i.e., the actual time taken by the operation to serve a client request if it didn't conflict with any other operation. Service time is usually negligible compared to geo-scale network latencies, but there also exist long-running transactions. In our application model, we consider an operation's service time, measured as the time between lock acquire and release.

11.1.2 Network model

The application is deployed over a network of geographically-distant locations. Each location contains a copy of the data, called a *replica*. A user accesses an arbitrary replica; a single replica. We consider that a replica is sequential and that the network latency between the client and the replica is zero (i.e., the client is at the same location).

The main factor of interest is the transmission latency between locations, which typically runs in tens or hundreds of milliseconds. This influences in particular the delay to acquire a lock that is placed at a remote location.

11.1.3 Workload characteristics

We model an application workload by the frequency distribution by operation by location. For instance, bids for a piece of furniture offered in France has more place bid operations than close auction and has high geo-locality in France.

11.2 Dimensions of Concurrency control

In this section, we take a closer look at the three dimensions and how they affect the overhead introduced due to locking.

11.2.1 Granularity

For a given conflict graph, we start with fine-grained locks, per operation per conflict, based on the coordination condition. Coarsening replaces several (fine-grained) locks with a single (coarse-grained) one; it also replaces the coordination condition of the coarse-grained lock to be the disjunction of the fine-grained locks. The granularity of a lock affects both components of the lock overhead - acquisition cost and contention. An operation that previously acquired and released multiple fine-grained locks, now uses the single coarse-grained replacement. Similarly, instead of coordinating on the smaller coordination conditions, the coarsened lock needs to coordinate on a stronger coordination condition. This decreases acquire/release overhead, at the expense of parallelism (higher contention).

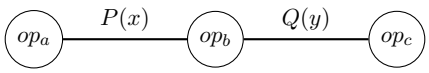


Figure 11.2: Three conflicting operations.

For instance, consider the conflict graph in Figure 11.2. Operations op_a and op_b conflict on the condition $P(x)$, acquiring the lock l_{ab-P_x}. Operations op_b and op_c need to coordinate on condition $Q(y)$ by acquiring lock l_{bc-Q_y}.

Let us consider how the dimension of coarsening is applicable here. The locks l_{ab-P_x} and l_{bc-Q_y} can be coarsened into a single lock $l_{abc-P_xQ_y}$. The coordination condition of the resulting lock is the disjunction of the coordination conditions of the fine-grained locks.

For example, assume $P(x) \triangleq x < 100$ and $Q(y) \triangleq y > 20$. The coarsened precondition for the joined combined lock, $l_{abc-P_xQ_y}$, will be $P_xQ_y \triangleq x < 100 \vee y > 20$. Now consider an operation call with signature $op_a(10)$. This means for this call, $x = 10$ and hence op_a needs to acquire $l_{abc-P_xQ_y}$

Coarsening reduces lock acquisition costs for op_b, since it only needs to acquire one lock instead of two. At the same time, it increases lock contention for op_a and op_c since they have to now wait for not only op_b as in the previous case, but also for each other.

We can call this way of coarsening the *operation coarsening* since it decreases the number of locks to be acquired per operation. Another dimension of coarsening is to coarsen the coordination condition, called *condition coarsening.* Condition coarsening doesn't reduce the number of lock acquisitions, and instead increases the lock contention since multiple calls to the same operation with different parameter values synchronize unnecessarily. [1] Hence in this work, we do not consider coarsening on this dimension.

As we discussed in Subsection 11.1.1, the conflict graph of an application contains one or more disjoint subgraphs. An operation in one subgraph can safely execute concurrently with any other operation outside of its subgraph. Hence we only coarsen until we reach a single lock for a subgraph; further coarsening will not reduce lock acquisition costs, but will only increase contention.

[1] Coarser conditions may be easier to evaluate, hence widely used in practice; for example, coarsening the conflict condition of Figure 11.2 $P(x) =$ `true` or $Q(y) =$ `true`. This work currently focusses on the performance impact of coordination configurations, not the ease of use.

11.2.2 Mode

Among the many known locking techniques [61], we focus on the difference between mutex locks and shared/exclusive locks, which we call lock mode. Mutex locks totally order the conflicting operations whereas shared/exclusive locks maintain a partial order where the operations with shared mode run concurrently, and those with exclusive mode blocks all other operations that share the lock.

When operations conflict, they acquire lock either in shared mode or exclusive mode. If an operation conflicts with itself, it has to acquire lock in exclusive mode.

Consider for example, operations op_a and op_b that share $lock_{ab-<x>}$ either in mutex mode or shared/exclusive mode. Acquiring the lock in mutex mode will totally order op_a and op_b, whereas acquiring the lock in shared mode by op_a and in exclusive mode by op_b, will maintain a total order for op_b and between op_a and op_b, but allows concurrent executions of op_a. This is safe if op_a doesn't conflict with itself.

When coarsening, multiple operations might be sharing a single coarse-grained lock. Allocating shared/exclusive mode for coarse-grained lock might violate safety if conflicting operations acquire the lock in shared mode. In order to maintain correctness, we require that at least one operation involved in each conflict of the conflict graph acquire exclusive mode. In terms of graph theory, we require at least the minimum vertex cover to acquire the exclusive mode.

A distributed lock can use several algorithms to provide shared/exclusive and mutual exclusive guarantees. Redlock [62], FencedLock [63], the distributed transactional lock from NuoDB [64], DynamoDB lock client [65], the lock service of etcd [66], and Zookeeper Lock recipes [67] are some commonly used distributed locks.

The underlying algorithms for mutual exclusive locks and shared exclusive locks differ between these popular lock providers. This reflects in the cost of lock acquisition for different lock providers. For example, consider the algorithm behind shared/exclusive lock. The main difference in the different modes is the parallelism allowed, shared mode allows parallelism with other shared modes, whereas exclusive mode allows zero parallelism.

In NuoDB, acquiring a shared lock incurs zero overhead, whereas the exclusive mode pays a price of three times the network latency. In Zookeeper, the lock acquisitions for both modes have the same cost. We ran an experiment with 3 instances of Zookeeper to test the response times of different modes of locking from different replicas placed at geographically distant locations. The average response times are shown in Figure 11.3. Each bar represents the average of 100 calls.

Observe that the response time is the same at a given replica irrespective of the mode. As a reasonable simplification, we assume that the cost of lock acquire/release does not depend on the mode. Hence mode selection only affects lock contention, not lock acquisition costs.

11.2.3 Lock Placement

Popular distributed lock services such as Zookeeper [67] and etcd [66] are based on consensus across their members. The number of members can vary and can be placed at

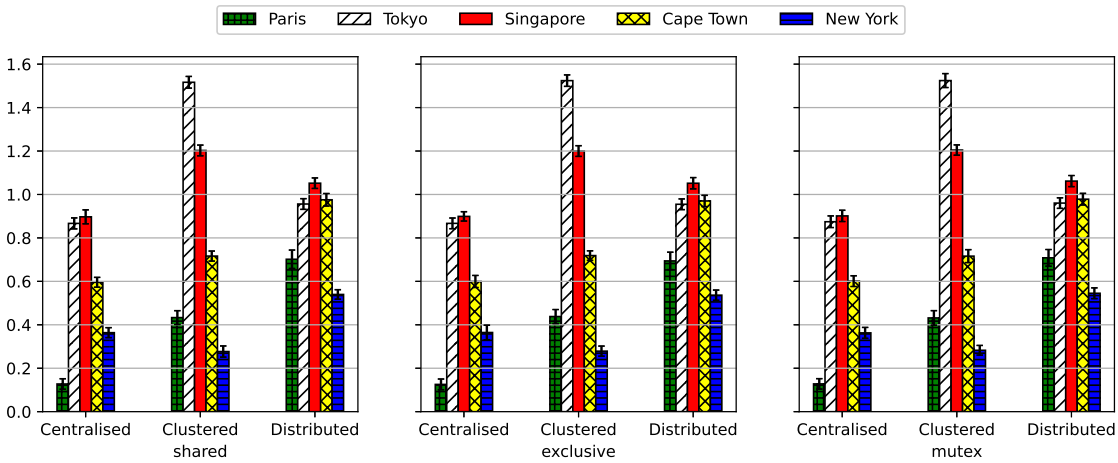


Figure 11.3: Average response time (in s) for lock acquisition requests to a 3-instance Zookeeper service. The subplots indicate different modes of locks and the groups indicate different placements. A centralised lock is placed in Paris; a clustered lock is clustered across Paris, Cape Town and New York; a distributed lock is distributed across all locations.

various geographical regions. Following this model, lock placement involves both a placement type and the actual lock location. Types include centralised (the lock service is at a single location), clustered (the lock service is distributed across a subset of locations), or fully distributed (it is present in all the locations of the system). When the lock service is clustered or fully distributed, this increases response time since the replicas have to execute a distributed agreement protocol. Figure 11.3 shows the increase in response time for clustered and distributed lock placements. Henceforth, we consider only the centralised placement.

The centrally placed lock can be co-located with any one of the replicas, or be at some other location. We consider the case where it is co-located with a replica. A client placed in a remote location will suffer additional network latency to access it. Thus, placement affects lock response time. This in turn affects contention, since the longer the delay, the higher the probability that a conflicting operation will have to wait.

11.3 The Coordination Lattice

Given a conflict graph and associated conditions, there always exists a safe coordination protocol: assign a fully distributed mutex lock to each conflict edge; an associated operation, for which the coordination condition is true,[2] acquires the lock on call, and releases on return. This fine-grained approach serves as our initial solution, but it may be expensive.

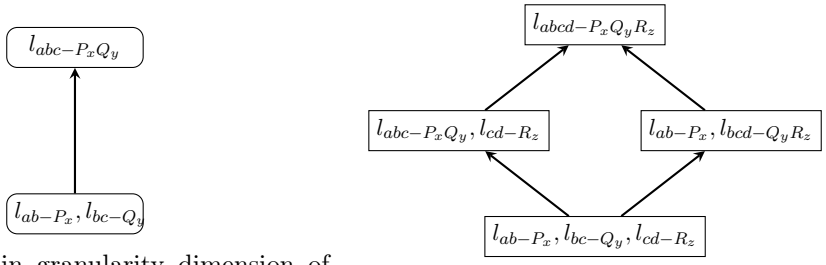
We present a lattice representation of the three dimensions of concurrency control, called the *coordination lattice*. This representation helps a systematic traversal of the lock configuration dimensions. The coordination lattice is constructed from the static characteristics of the application namely, the conflict graph, along with the coordination conditions.

[2] A coordination condition is evaluated locally at the calling replica, and is not required to be protected by the lock [6].

(a) Conflict involving three operations

(b) Conflict involving four operations

Figure 11.4: Conflict graphs



(a) Projection in granularity dimension of the coordination lattice of Figure 11.4a

(b) Projection in granularity dimension of the coordination lattice of Figure 11.4b

Figure 11.5: Conflict graph of a sample application with four conflicting operations

Of the three lattice dimensions (granularity, mode and placement) we consider granularity to be the main "plane". This is because the mode and placement decisions depend on first knowing which locks to consider, i.e., on granularity. Therefore, it makes sense to construct the lattice by first varying granularity, and then varying mode and placement for a given granularity.

The structure of the coordination lattice is derived from the dimension of granularity. The bottom element of the lattice is the finest grained locks and the top element is the coarsest grained locks. Each level of the lattice is formed by coarsening the locks in the previous level.

Each lattice element is a set of locks that are two-dimensional planes with mode and placement axes.

Let us examine the construction of a coordination lattice for the two conflict graphs shown in Figure 11.4. The conflict graph shown in Figure 11.4a (and Figure 11.4b) shows a conflict between three (four) operations on different coordination conditions. Figure 11.5a (and Figure 11.5b) shows the projection, along the granularity dimension, of the derived coordination lattice (arrows in the direction of more coarsening). An element in this projection is called a *granularity configuration*. The bottom element represents the initial fine-grained configuration. A successor configuration (by following an arrow) represents the coarsening of locks. Each two-dimensional lock in a granularity configuration is subscripted by the operations that acquire it, and by the corresponding coordination condition (after hyphen).

Each lock in a granularity configuration may have multiple mode and placement configurations. A single combination of granularity, mode and placement is called a coordination configuration.

11.4 Navigating the coordination lattice

Navigation of the coordination lattice is guided by the estimated dynamic characteristics of the application - its workload characteristics and service time per operation. We evaluate the performance of each point in the lattice, using the workload model. This enables the developer to select among safe coordination protocols, according to representative performance metrics.

As discussed in Subsection 11.2.3, the choice of placement determines the cost of lock acquisition at a given replica.[3] This information is needed to decide a mode configuration since we need to maximize parallelism. Once we have the placement and mode dimensions for each granularity configuration, we can navigate the lattice to select the optimal coordination configuration according to a given workload. Hence, for each granularity configuration, we use some metrics to determine the optimal configuration and then navigate the granularity dimension for selecting the optimal coordination configuration.

11.4.1 Granularity selection

Moving upwards in a coordination lattice through the granularity dimension means reducing the number of locks by coarsening. This has two effects - minimizing the number of locks to be acquired reduces lock acquisition costs, but at the same time increasing the lock contention since the operations that previously didn't require coordination for the fine-grained locks, now have to coordinate.

We define a metric *Coordination configuration nonparallelism*, CcNonParallel, (in Subsection 11.4.2) for each coordination configuration to help navigate the coordination lattice. Coordination configuration nonparallelism gives the estimate of the amount of parallelism disallowed by a given coordination configuration; it is obtained by varying mode and placement for a given granularity. The optimal coordination configuration will be the one with the minimum Coordination configuration nonparallelism.

11.4.2 Mode selection

A shared/exclusive lock enables multiple processes to hold the lock in shared mode, whereas exclusive mode blocks concurrent acquires (whether shared or exclusive) until it is released. All mutex acquires are exclusive: only one process at a time may hold a mutex lock; all concurrent acquires are blocked.

The choice of locking mode should aim to minimise the time where locks are held in exclusive mode, as this is what creates lock contention. Mode selection follows the following rules:

1. *Use shared/exclusive mode when it is safe, otherwise mutex.* If some operation op_a conflicts with another op_b, and multiple instances of op_a's don't conflict with each other, then it is safe for op_a to acquire lock $lock_{ab-P_x}$ in shared mode. In this case, $lock_{ab-P_x}$ can be a shared/exclusive lock. In all other cases, conservatively, it shall be a mutex.

[3] In this work, we are not considering the effect of caching a lock.

2. *For a shared/exclusive lock, the mode that allows maximum parallelism possible should be selected.* To explain this rule, we define some metrics. Call *Lock Holding Time* (LHT) the minimum time during which an operation holds the lock. LHT is the sum of lock acquisition time and service time.[4] $LHT(op, l, r)$ for operation op for lock l at replica r is given by

$$LHT(op, l, r) = service_time(op) + C(l, r)$$

where $service_time(op)$ is the execution time of the operation, outside of lock acquisition and release, and $C(l, r)$ is the cost of acquiring lock l at replica r. Note that LHT differs per operation and per replica. We will consider the impact of lock contention next.

Now let us study the possible parallelism, based on LHT and locking mode.

The set of operations that gets the exclusive mode of a single lock, l, serialises all executions for that particular set of operations across all replicas. We call *operation serialisation* the estimate of the expected execution time at replica r as a result of this serialisation. This provides us the time during which the operations that acquire the exclusive mode of lock l at replica r have to wait due to the execution of operations acquiring the same mode of the same lock at other replicas. For a set of operations O acquiring a lock l at replica r, *operation serialisation*, OpSerial, can be defined as follows:

$$\textsc{OpSerial}(l, r) = \sum_{op \in O, r' \in R} LHT(op, l, r') * freq(op, r') \mid r' \neq r \wedge op.l.mode = exclusive$$

where R is the set of replicas and $freq(op, r')$ is the frequency of operation op in replica r'. Operation serialization is the total execution time taken by the operations.

Recollect that according to our system model, individual replicas are sequential — irrespective of mode, all operations on a given replica run sequentially. Metric *replica serialisation* captures this behaviour. For a set of operations O acquiring lock l, *replica serialisation*, RepSerial, on replica r can be defined as

$$\textsc{RepSerial}(l, r) = \sum_{op \in O} LHT(op, l, r) * freq(op, r)$$

Replica serialisation is the total execution time taken by all operations that acquire lock l at replica r.

Let us now study the amount of parallelism allowed on other replicas by a given mode, *operation parallelism*. For each replica, this is the total time taken to execute all operations that acquires the lock in shared mode. For a set of operations O, acquiring a lock l, *operation parallelism*, OpParallel, for a replica r can be defined

[4] An operation acquiring multiple locks would hold the lock for more time (waiting for the acquisition of other locks) if the given lock was acquired first. For simplicity, we are not considering that case.

as

$$\textsc{OpParallel}(l,r) = \max_{r' \in R} \sum_{op \in O} LHT(op,l,r') * freq(op,r') \mid r' \neq r \wedge op.l.mode = shared$$

Note that OpParallel represents the maximum time taken by a single replica to execute operations that has acquired the shared mode of lock. Other replicas that also have operations with shared mode of locking can run parallel to this.

Mode selection needs to minimise *operation serialisation*, along with *operation parallelism* across all replicas.[5][6] We use these metrics in order to define *replica execution time*, RepExecTime, that indicates the average time needed to execute all the replicas when all replicas start their execution at the same time.

$$\textsc{RepExecTime}(l) = \frac{\sum_{r \in R}(\textsc{OpSerial}(l,r) + \textsc{RepSerial}(l,r) + \textsc{OpParallel}(l,r))}{|R|}$$

A coordination configuration in the coordination lattice might include more than one lock. Since a lattice contains a single conflict graph, all locks essentially are related and they need to wait for others. We extend the metrics for a single lock for a set of locks belonging to a single coordination configuration. The enhanced set of metrics, with prefix Cc, is as follows:

$$\textsc{CcLHT}(op,r) = service_time(op) + \sum_{l \in L_{cc}} C(l,r)$$

$$\textsc{CcOpSerial}(r) = \sum_{l \in L_{cc}, op \in O, r' \in R} \textsc{CcLHT}(op,r') * freq(op,r') \mid r' \neq r \wedge op.l.mode = exclusive$$

$$\textsc{CcRepSerial}(r) = \sum_{l \in L_{cc}, op \in O} \textsc{CcLHT}(op,r) * freq(op,r)$$

$$\textsc{CcOpParallel}(r) = \max_{r' \in R} \sum_{l \in L_{cc}, op \in O} \textsc{CcLHT}(op,r') * freq(op,r') \mid r' \neq r \wedge op.l.mode = shared$$

$$\textsc{CcRepExecTime} = \frac{\sum_{r \in R}(\textsc{CcOpSerial}(r) + \textsc{CcRepSerial}(r) + \textsc{CcOpParallel}(r))}{|R|}$$

where L_{cc} is the set of locks for a particular coordination configuration (granularity, mode and placement dimensions are assigned). We select the coordination configuration with minimal CcRepExecTime.

Relation to Amdahl's law: Amdahl's law [68] gives us the speed up possible by parallelising the execution on n number of processors.

$$Speedup = \frac{1}{r_s + \frac{r_p}{n}}$$

[5]Note that *replica serialisation* cannot be modified since all operations in a single replica are sequential irrespective of the mode.

[6]We add these components because the operations that acquire exclusive mode of locks need to wait for the replicas acquiring shared mode of locking.

where $r_s + r_p = 1$ and r_s represents the ratio of the sequential portion of one program. It can also be written as

$$Speedup = \frac{r_s + r_p}{r_s + \frac{r_p}{n}}$$

The speed up attainable due to parallelisation is the ratio of the execution time of the process if it were to run fully sequential to the execution time of the process parallelised on n processors.

Similar to the system model in Amdahl's law, we have a set of processors that can process operations in parallel. Instead of multiple processors, we have replicas and the number of replicas are fixed in our case. The numerator of the speed up ratio as per Amdahl's law is the execution time required if all the operations were sequential. In our case, it is similar to acquiring mutual exclusive lock for all operations. The denominator represents the execution time of the parallelised process, if any. The amount of parallelism permissible is determined by the mode of locking. If we use mutex, there is no extra parallelism possible, hence $r_s = 1$ and $r_p = 0$. This gives us the baseline for calculating speed up.

Let us discuss how our metrics relates to Amdahl's law at replica r. Let S be the total execution time at replica r when all the locks are mutual exclusive. This is essentially the sum of operation serialisation and replica serialisation at a single replica when all the locks are acquired in mutual exclusive mode.[7]

$$S(r) = \textsc{CcOpSerial}(r) + \textsc{CcRepSerial}(r) \mid \forall l \in L_{cc}, op \in O \,\bullet\, op.l.mode = exclusive$$

We now define two parameters s and p that gives the sequential part of the execution time and parallel part of the execution time respectively. s and p at replica r can be defined as follows:

$$\begin{aligned} s(r) &= \textsc{CcOpSerial}(r) + \textsc{CcRepSerial}(r) \\ p(r) &= \sum_{r' \in R, l \in L_{cc}, op \in O} \textsc{CcLHT}(op, r') * freq(op, r') \mid r' \neq r \wedge op.l.mode = shared \end{aligned}$$

With these new terms, Amdahl's law can be rewritten as

$$\begin{aligned} Speedup(r) &= \frac{s + p}{s + \frac{p}{n}} \\ \overline{Speedup} &= \frac{\sum_{r \in R} Speedup(r)}{|R|} \end{aligned}$$

where $s + p = S$ and $\overline{Speedup}$ indicates the average speed up. For mutex, $s = S$ and $p = 0$, and $\overline{Speedup} = 1$, which gives us the baseline.

Since we are constrained by the workload characteristics, unlike in the case of Amdahl's law where the parallelisation can be manipulated, we cannot fairly "distribute" the

[7] The sum of operation serialisation and replica serialisation would be the same across all replicas in the case of a mutex.

execution of the parallel portion across all replicas. Hence, we can substitute the term $\frac{p}{n} = \textsc{CcOpParallel}(r)$.

With our metrics, Amdahl's law becomes

$$Speedup(r) = \frac{\textsc{CcOpSerial}(r) + \textsc{CcRepSerial}(r) \mid \forall l \in L_{cc}, op \in O \,.\, l.op.mode = exclusive}{\textsc{CcOpSerial}(r) + \textsc{CcRepSerial}(r) + \textsc{CcOpParallel}(r)}$$

To maximize speedup, we need to minimize the denominator, corresponding to our CcRepExecTime metrics.

11.4.3 Placement selection

The placement of the lock service affects the response time, as network latency affects both the cost of consensus between different instances of the lock service, and cost for the client to access the service, when it is remote.

Let us note $C(l, p, r)$ the cost of acquiring lock l from replica r for placement p, and $freq(l, r)$ the frequency of acquiring lock l from replica r, then expected cost of acquisition, TC, for a chosen placement p is given by

$$TC(l, p) = \sum_{r} freq(l, r) * C(l, p, r)$$

The selected placement, $\hat{p}$, should minimise the acquisition cost over all placements, i.e.,

$$TC(l, \hat{p}) = \min_{p}(TC(l, p))$$

In Chapter 12, we discuss on applying this metrics to some examples to determine the optimal coordination configuration for a given workload.

Chapter 12

Experiments

This chapter describes some experiments that illustrate the coordination lattice. We consider two conflict graphs and the corresponding coordination lattices. We include several synthetic workloads, and show how our metrics vary, for each workload, with coordination configuration. We then compare our prediction and experimental values.

12.1 Experimental setup

The experimental setup consists of a set of configuration files provided as user input, a preprocessing stage that processes the inputs, and a simulator. Figure 12.1 shows the overall architecture of the experimental system.

12.1.1 Inputs

The inputs to the experiment are the following:

- The *Conflict graph* lists pairs of conflicting methods along with the corresponding coordination conditions.
- The *Operation service time* provides the time taken by each operation between its lock acquires and releases.
- The *Latency matrix* lists the replicas and the network latency between pairs.
- The *Workload characteristics* contains the frequency of operations per replica.

The simulator uses *Operation service time* to simulate the operation execution; the operation sleeps during the service time. The simulator uses *Latency matrix* to construct the network layer.

12.1.2 Intermediate processes

The preprocessing stage processes the configuration provided by the user to the format required by the simulator. It has two components - *DisLockGen* (Distributed Lock Generator) and *WorkloadGen* (Workload generator).

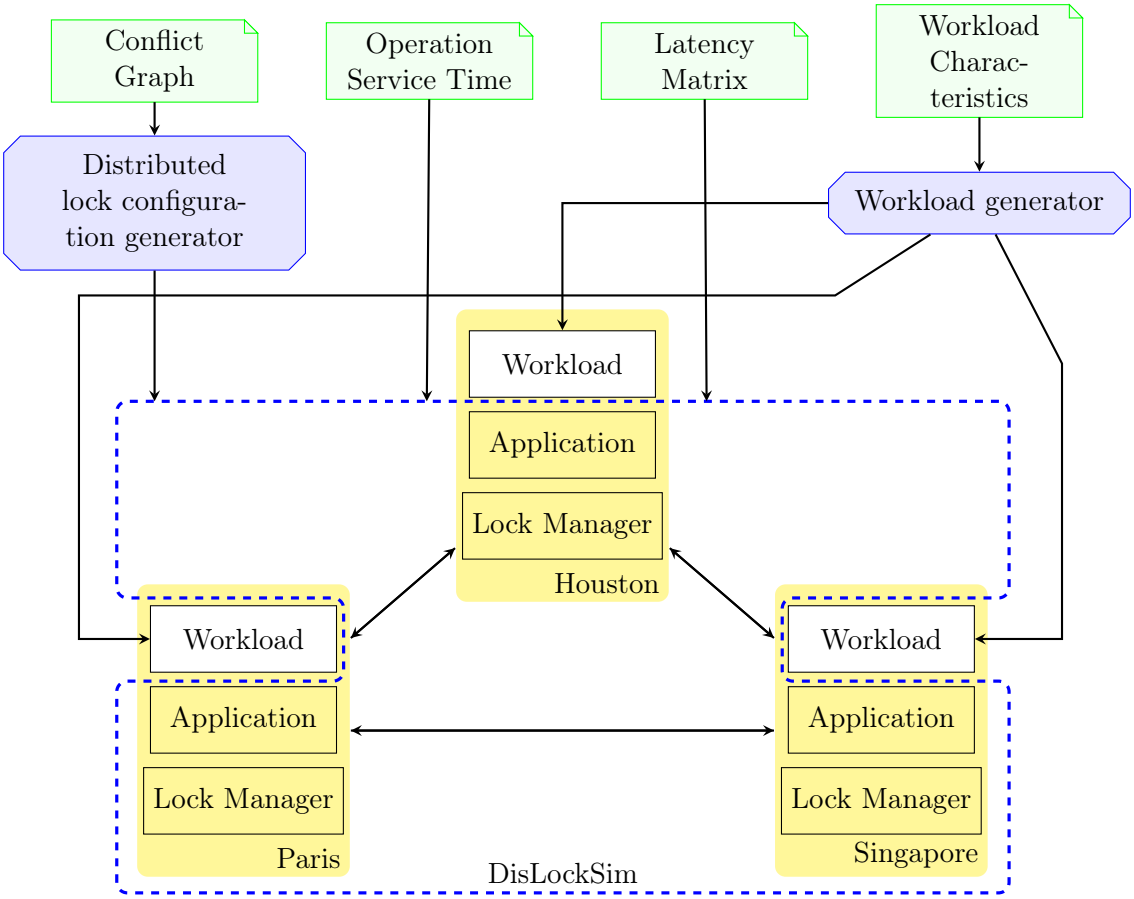


Figure 12.1: The architecture of dislock experiment. The first row lists the inputs, and the second row the preprocessing stage. The dotted box represents the simulator. The yellow shaded regions represent the physical location.

Latency (ms)	Houston	Paris	Singapore
Houston	—	55	125
Paris	55	—	75
Singapore	125	75	—

Table 12.1: Average latency between replicas

DisLockGen uses the conflict graph to generate all possible coordination configurations. The simulator uses this generated set of coordination configurations to run the application.

The Yahoo! Cloud Serving Benchmark (YCSB) [69] serves as the workload for the simulator. It issues client calls to the DisLockSim; there is a YCSB instance colocated with each replica. The workload requires trace files and workload configuration in the format for YCSB. *WorkloadGen* takes the list of operations along with their parameters and the workload characteristics (frequency of each operation per replica) to generate the configuration files.

12.1.3 DisLockSim - A simulation model for distributed lock

We present *DisLockSim* (abbreviation of Distributed Lock Simulator), a tool that allows us to observe the performance of different distributed lock configurations. It uses the user-provided inputs to instantiate the distributed application, the system on which it operates, and the expected workload.

The simulator is composed of three layers - the network layer, the application and the lock manager running on top of the network layer. The network layer specifies the number of replicas and the latency between each pair.

Each replica will have an instance of the distributed application and a lock manager running. The application layer provides APIs for client calls. Instead of actually executing the operation, it simulates the execution using the service time per operation provided by the user. To ensure safety, the application prepares the list of locks according to the coordination configuration chosen. It then issues acquire and release requests with the list of locks to the respective lock managers.

A lock manager receives lock acquisition requests from applications running on different replicas. The lock manager processes the request depending on the mode requested. Lock manager uses the lock service of etcd [70] to maintain the guarantees of the locking modes.

12.1.4 Cost of locking

Our experiments consider the replicas located in Houston, Paris and Singapore respectively. The latencies between these replicas are shown in Table 12.1.

Recall that the lock manager uses the lock service of etcd [70] to maintain the lock guarantees. We run a benchmark to determine the cost of acquiring and releasing a lock for different modes and placements. The results of benchmarking the cost of locking using the etcd lock service is shown in Figure 12.2. Observe that the cost of locking from each replica is greater than four times the latency due to two calls – acquire and release. As we

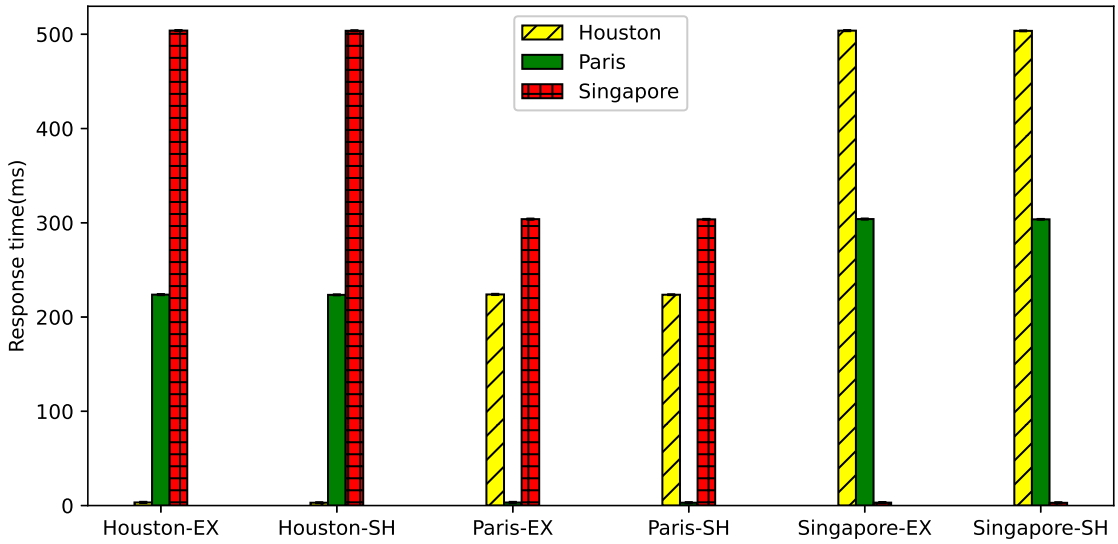


Figure 12.2: Cost of locking for different lock placements from different replicas. The X-axis shows placement and mode configurations. The Y-axis represents response time in ms.

discussed in Section 11.2, the cost of locking depends on lock placement, but not on the type of lock. We then use this benchmark to calculate the metrics discussed in Section 11.4.

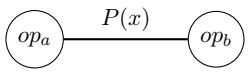
12.2 Analysing some conflict graphs

In this section, we consider some conflict graphs with some synthetic workloads.

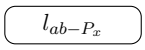
We use the metrics discussed in Section 11.4 to predict the comparative performance of each coordination configuration. For each workload, we calculate the CcRepExecTime metrics for all coordination configurations. We then measure the total execution time taken for each configuration for a given workload; we take the average from 5 runs of the experiment. We compare the metrics with experimental values. We run all the experiments on DELL PowerEdge R440 machine with 384 GB RAM and 2 x Intel Xeon Silver 4216 32 cores / 64 threads @ 2.10GHz processor.

As discussed in Chapter 11, we can use CcRepExecTime to find the optimal distributed lock configuration. In this section, we experimentally illustrate the usage of the metrics.

12.2.1 Conflict graph involving two operations



(a) Two conflicting operations



(b) Coordination lattice

Figure 12.3: Conflict graph and coordination lattice for a single lock

Consider a conflict graph containing two operations op_a and op_b conflicting on condition $P(x)$, as shown in Figure 12.3a. Since the conflict graph involves a single lock, l_{ab-P_x}, the

Mode	op_a	op_b
1	EX	EX
2	EX	SH
3	SH	EX

(a) Mode configuration. EX and SH stands for exclusive and shared modes respectively.

Placement	l_{ab-P_x}
1	Houston
2	Paris
3	Singapore

(b) Placement configuration. H, P, S stands for Houston, Paris and Singapore respectively.

Table 12.2: Coordination configurations for lock l_{ab-P_x} from Figure 12.3b

Workload	Operations	Houston	Paris	Singapore	Description
Workload A	op_a	167	167	166	Frequency per operation per
	op_b	167	167	166	replica is the same.
Workload B	op_a	0	500	0	Frequency per operation is
	op_b	0	500	0	equal, located at a single replica.
Workload C	op_a	250	250	0	Equal frequency per
	op_b	250	250	0	operation at a replica cluster.
Workload D	op_a	333	334	333	A single operation equally
	op_b	0	0	0	distributed across all replicas.
Workload E	op_a	0	1000	0	A single operation located at
	op_b	0	0	0	a single replica.
Workload F	op_a	500	500	0	A single operation executing
	op_b	0	0	0	at a replica cluster.
Workload G	op_a	0	500	0	Equal frequency per
	op_b	250	0	250	operation, different frequency across replicas.
Workload H	op_a	0	500	0	Equal frequency per
	op_b	167	167	166	operation, different frequency across replicas.

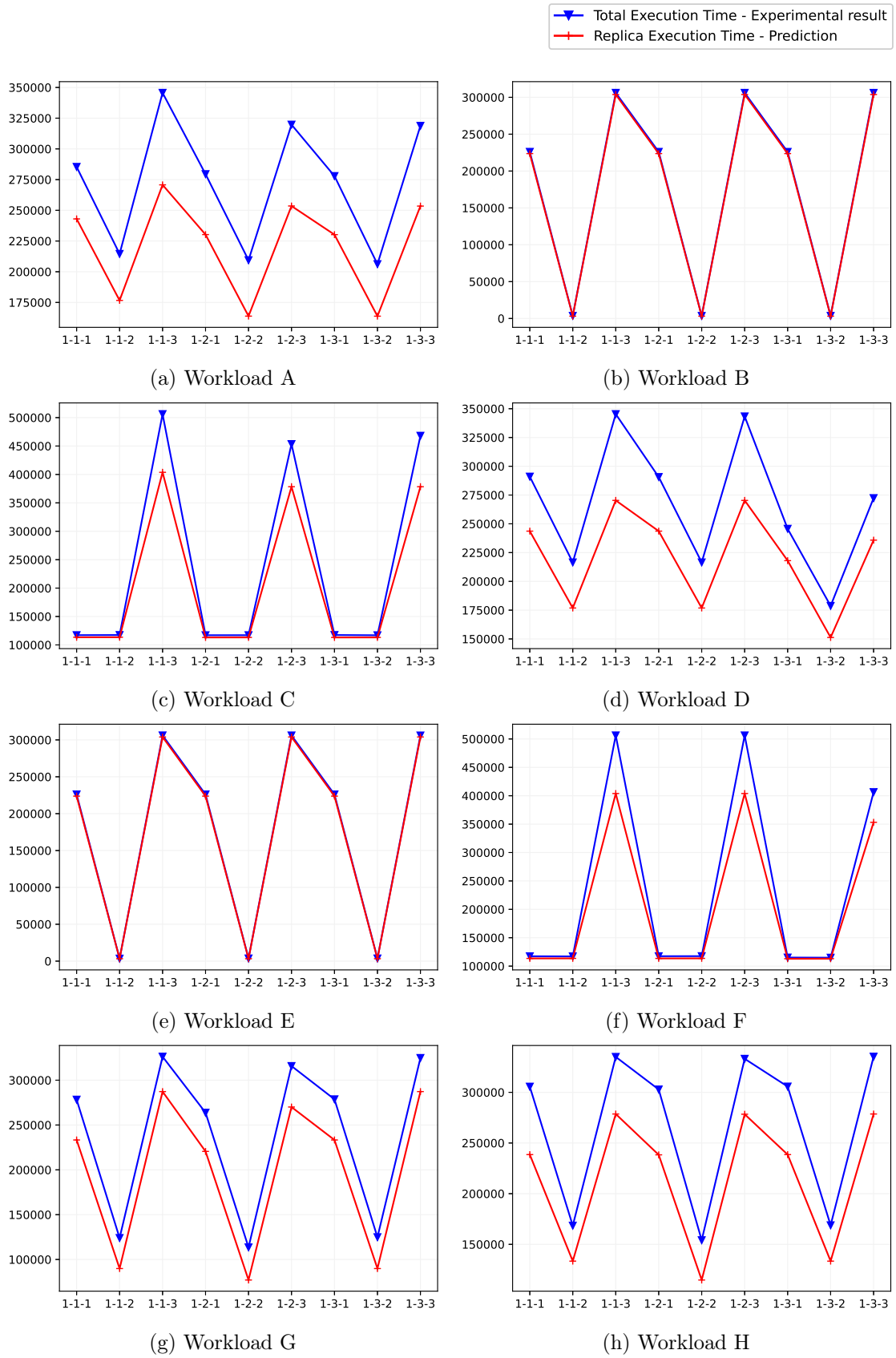
Table 12.3: Workloads for the conflict graph involving two operations.

granularity dimension of the coordination lattice is trivial with a single element. The lock is two dimensional along mode and placement. The mode and placement dimensions of a lock are as shown in Table 12.2.

We consider some synthetic workloads as listed in Table 12.3. Figure 12.4 compares the metrics CcRepExecTime and the total time taken for all the operations for each workload for a given configuration.

The point of interest for us is the pattern of the plots. Observe that the coordination configuration with minimal CcRepExecTime is also the one with the lowest total execution time for each workload.

The metric CcRepExecTime gives the maximum execution time for a single replica in the system. For workloads B and E, the metrics and the experimental results overlap since all the operations happen at a single replica. For workloads C and F, we observe a slight deviation from the metrics because there are operations happening on two replicas. The difference between the lines in the plot varies as the workload is being distributed



(a) Workload A

(b) Workload B

(c) Workload C

(d) Workload D

(e) Workload E

(f) Workload F

(g) Workload G

(h) Workload H

Figure 12.4: Plots of CcRepExecTime and total execution time obtained from the experiments for different workloads for the conflict graph in Figure 12.3.

Mode	l_{ab-P_x}		l_{bc-Q_y}	
	op_a	op_b	op_b	op_c
1	EX	EX	EX	EX
2	EX	EX	EX	SH
3	EX	EX	SH	EX
4	EX	SH	EX	EX
5	EX	SH	EX	SH
6	EX	SH	SH	EX
7	SH	EX	EX	EX
8	SH	EX	EX	SH
9	SH	EX	SH	EX

(a) Mode configuration for fine grained locks

Placement	l_{ab-P_x}	l_{bc-Q_y}
1	Houston	Houston
2	Paris	Houston
3	Singapore	Houston
4	Houston	Paris
5	Paris	Paris
6	Singapore	Paris
7	Houston	Singapore
8	Paris	Singapore
9	Singapore	Singapore

(b) Placement configuration for fine grained locks

Mode	$l_{abc-P_xQ_y}$		
	op_a	op_b	op_c
1	EX	EX	EX
2	EX	SH	EX
3	SH	EX	SH

(c) Mode configuration for coarse lock

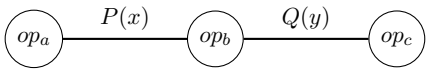
Placement	$l_{abc-P_xQ_y}$
1	Houston
2	Paris
3	Singapore

(d) Placement configuration for coarse lock

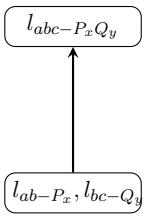
Table 12.4: Coordination configurations for the coordination lattice in Figure 12.5b

across the replicas; but the pattern remains the same.

12.2.2 Conflict graph involving three operations



(a) Three conflicting operations



(b) Coordination lattice

Figure 12.5: Conflict graph and coordination lattice for two locks

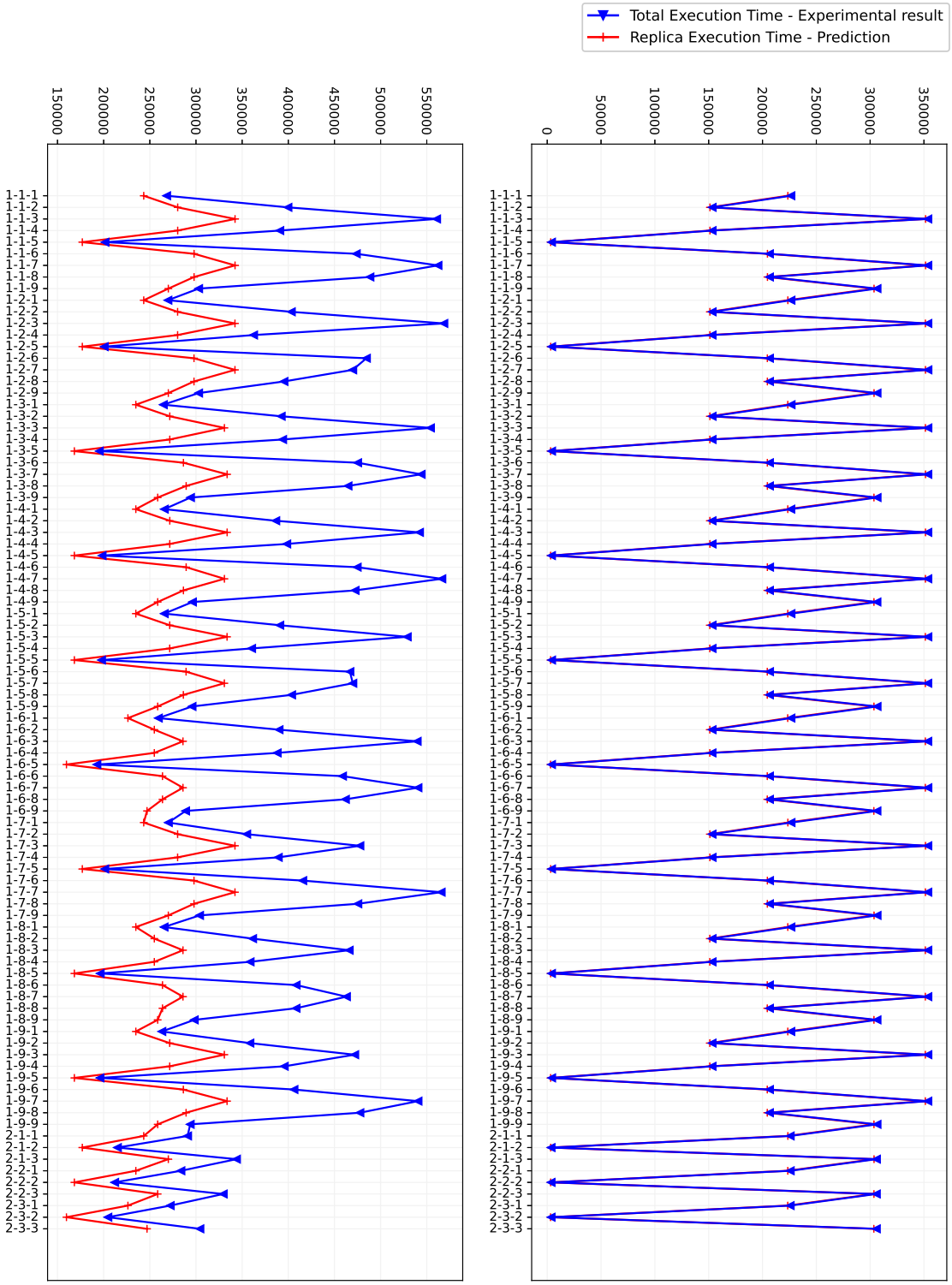
Consider a conflict graph containing three operations op_a, op_b and op_c conflicting on condition $P(x)$ and $Q(y)$ respectively, as shown in Figure 12.5a. The coordination lattice has two elements — fine locks, l_{ab-P_x} and l_{bc-Q_y}, and coarse lock $l_{abc-P_xQ_y}$. Each lock is two dimensional along mode and placement. The mode and placement dimensions are shown in Table 12.4.

Consider the example of online auction with operations *start auction*, *place bid*, *remove bid* and *close auction*. The operation *start auction* is safe with other concurrent operations whereas, *close auction* need to coordinate with both *place bid* and *remove bid* operations to ensure that the highest bid wins. This resembles the conflict graph in Figure 12.5a where *place bid* is op_a, *close auction* is op_b, and *remove bid* is op_c.

We consider some synthetic workloads listed in Table 12.5. The first three workloads

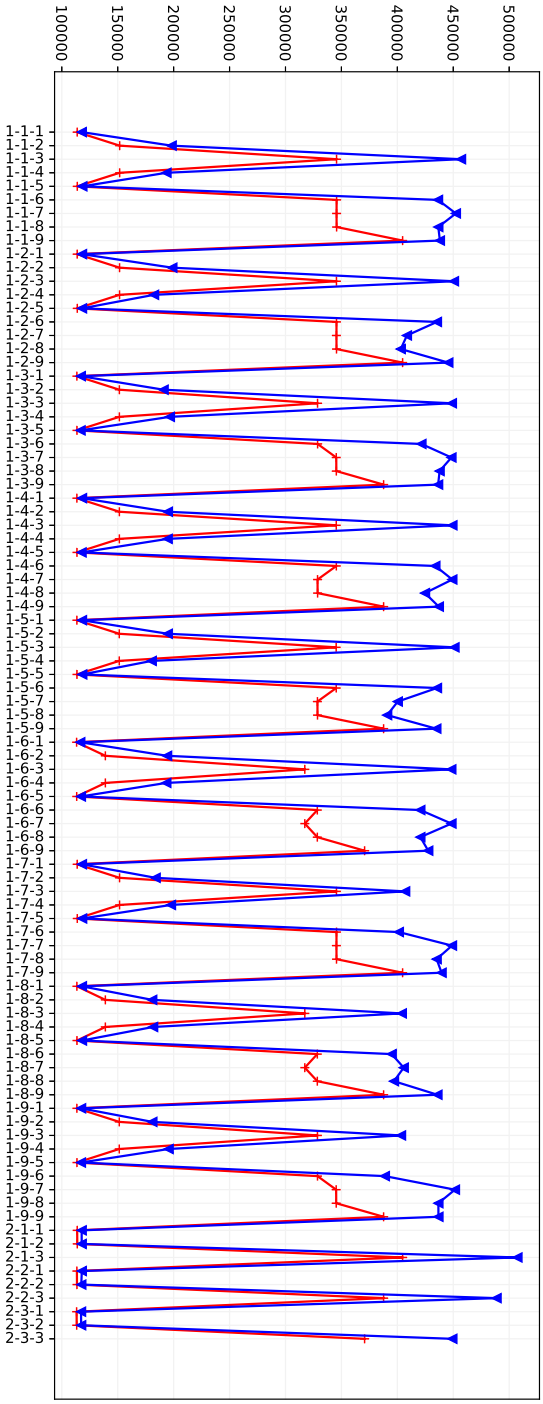
Workload	Operations	Houston	Paris	Singapore	Description
Workload A	op_a	111	111	111	Frequency per operation per replica is the same.
	op_b	111	111	111	
	op_c	111	111	111	
Workload B	op_a	0	333	0	Frequency per operation is equal, located at a single replica.
	op_b	0	333	0	
	op_c	0	333	0	
Workload C	op_a	167	167	0	Equal frequency per operation at a replica cluster.
	op_b	167	167	0	
	op_c	167	167	0	
Workload D	op_a	303	303	303	Highest frequency operation acquires a single lock, equal frequency across replicas.
	op_b	25	25	25	
	op_c	5	5	5	
Workload E	op_a	0	909	0	Highest frequency operation acquires a single lock at a single replica.
	op_b	0	75	0	
	op_c	0	15	0	
Workload F	op_a	455	455	0	Highest frequency operation acquires a single lock at a replica cluster.
	op_b	37	37	0	
	op_c	8	8	0	
Workload G	op_a	0	1	0	Operation acquiring multiple locks most frequent.
	op_b	100	100	100	
	op_c	0	1	0	
Workload H	op_a	100	100	0	Operation acquiring multiple locks least frequent.
	op_b	0	0	0	
	op_c	0	0	50	

Table 12.5: Different workloads for the conflict graph involving three operations.

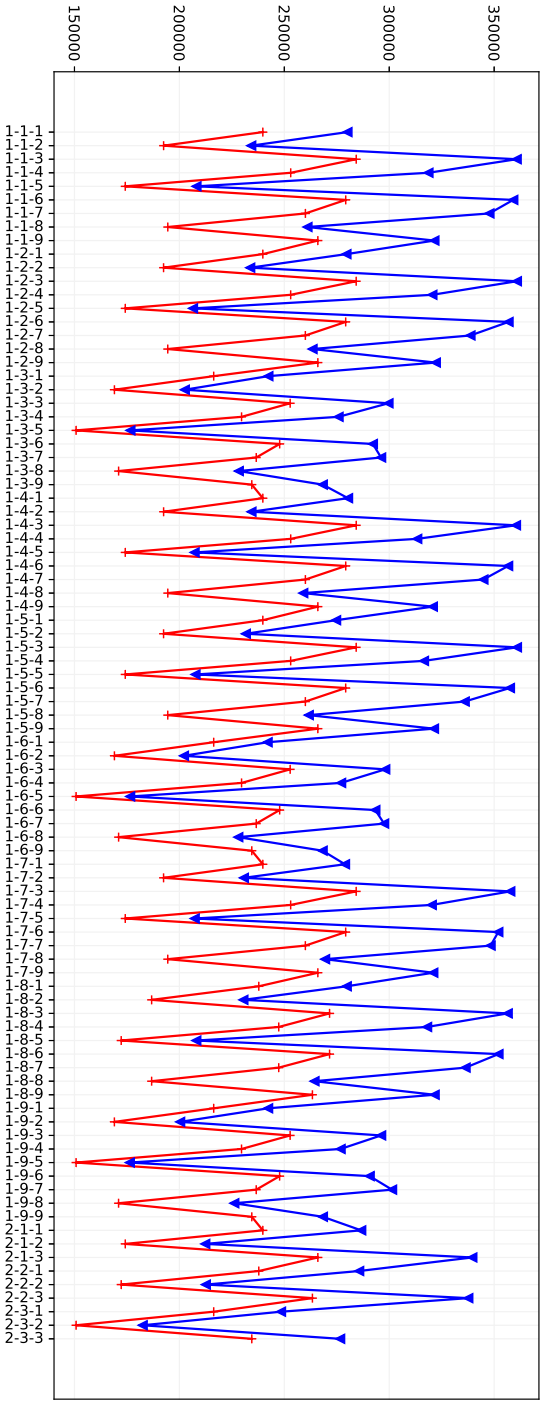


(a) Workload A

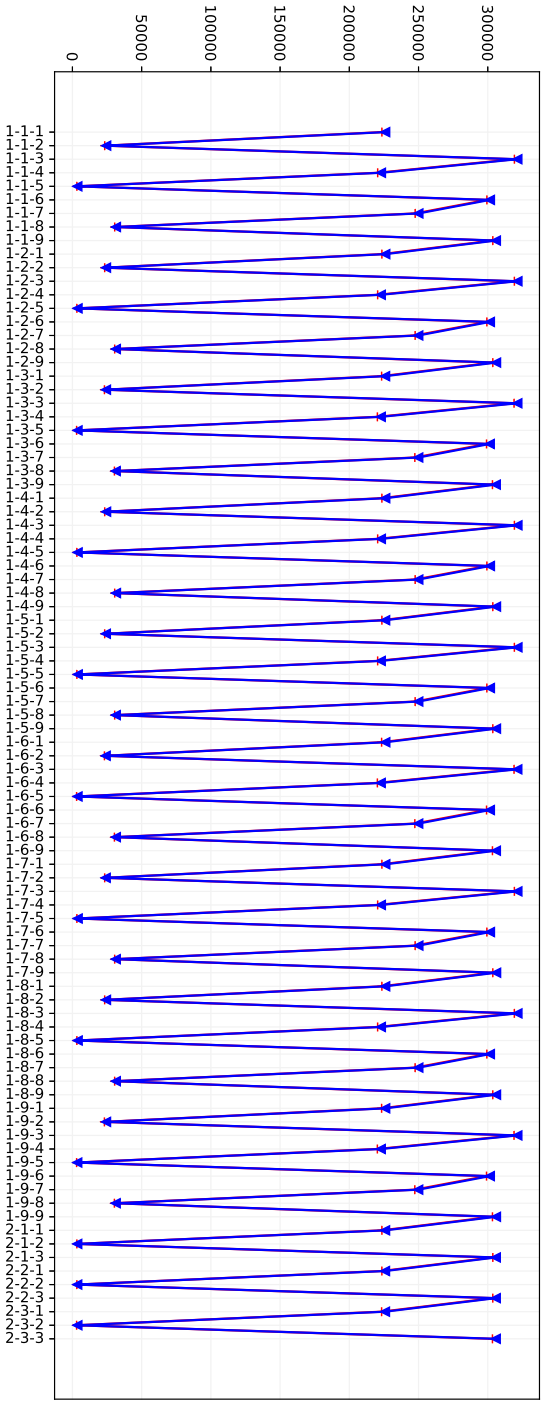
(b) Workload B



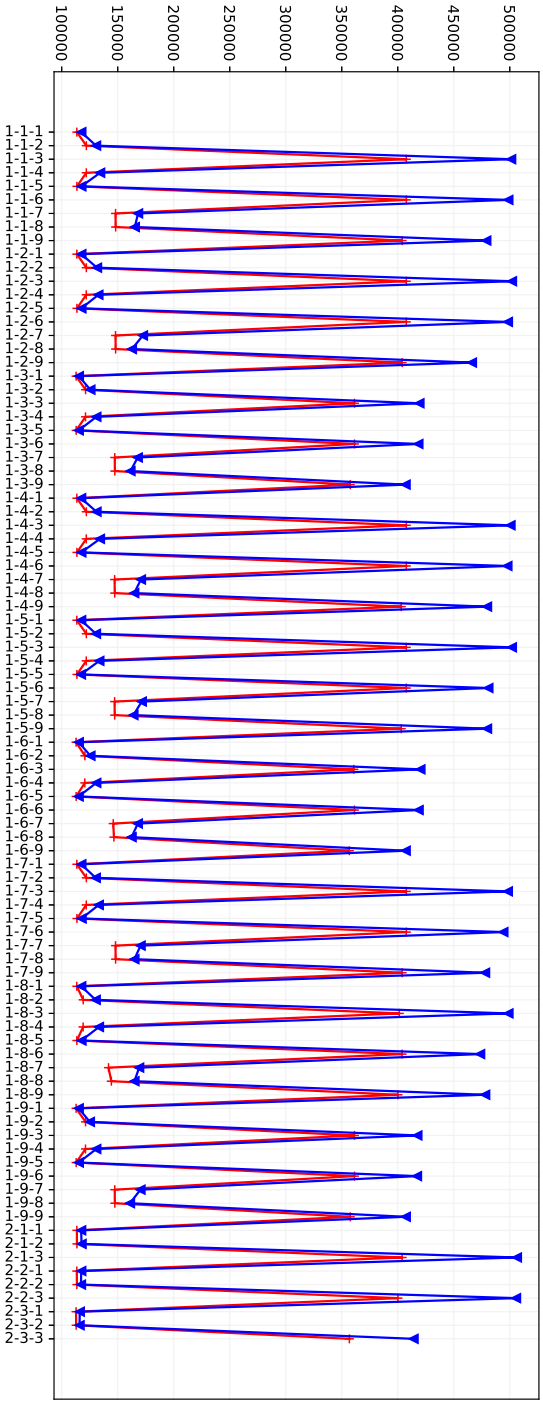
(c) Workload C



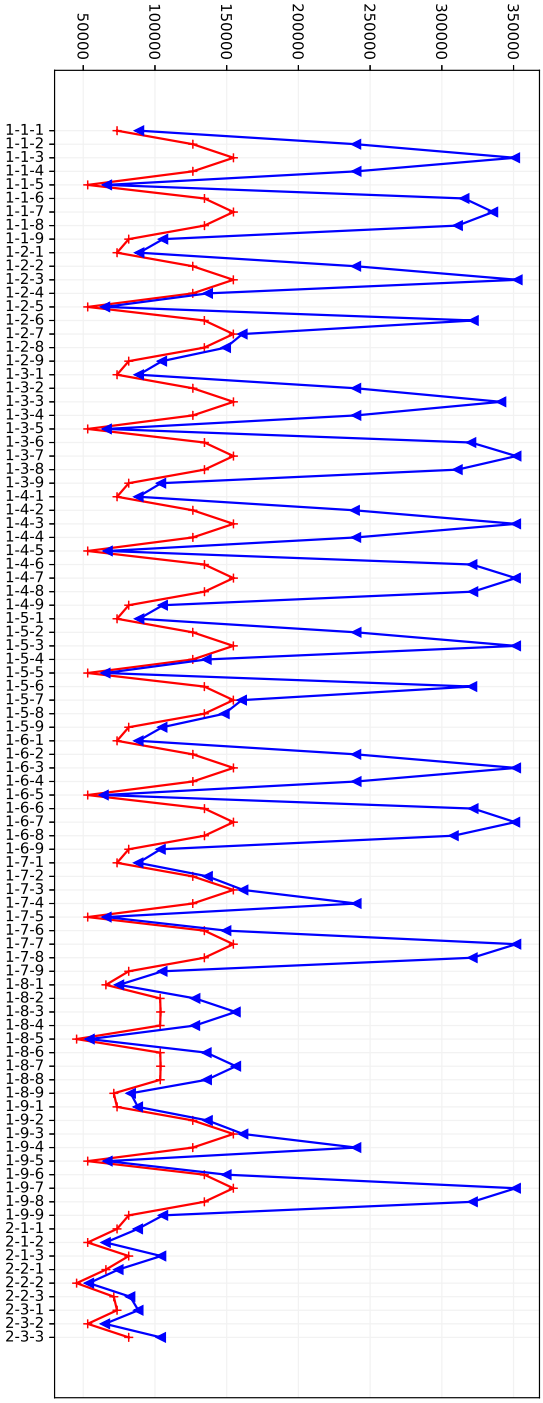
(d) Workload D



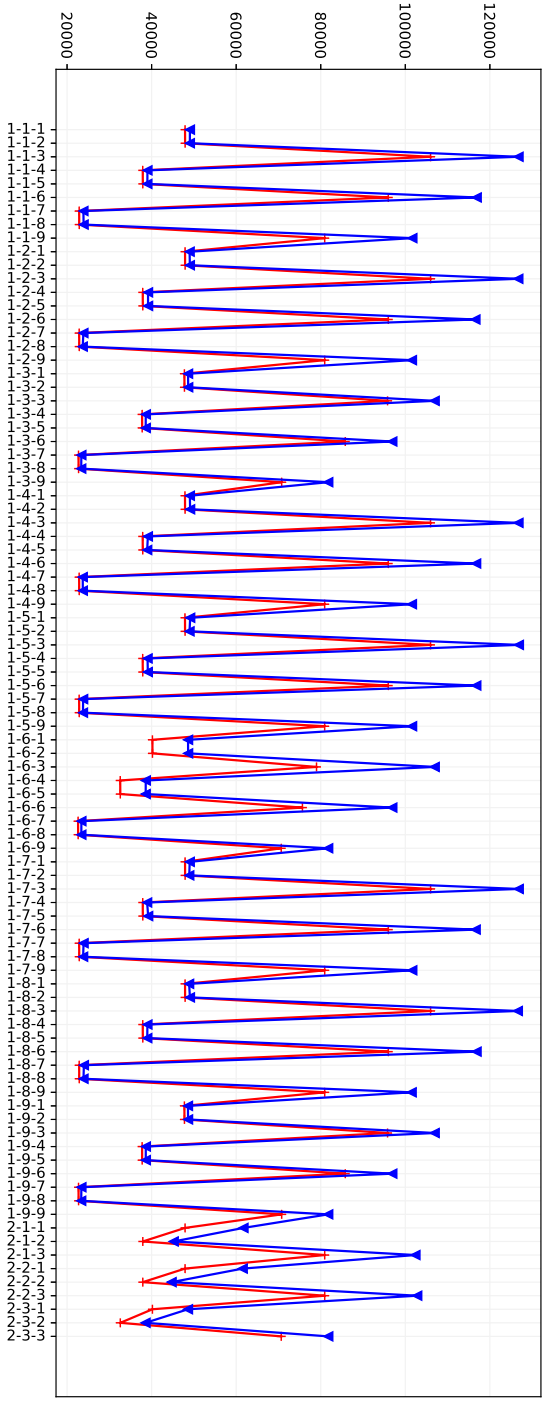
(e) Workload E



(f) Workload F



(g) Workload G



(h) Workload H

Figure 12.6: Plots of CcRepExecTime and total execution time obtained from the experiments for different workloads for Figure 12.5.

is a hypothetical scenario where all operations have equal probability of execution. The remaining workloads, except the last two, resemble the auction example we discussed, with place bids as the most common operation. The last two workloads show a synthetic case where coarsening of two locks helps or hinders the performance respectively.

Figure 12.6 shows the CcRepExecTime metrics and the experimentally obtained total time taken for all the operations for each workload for a given configuration. Workloads B and E follow the predictions since they are a single replica workload. As the workload gets distributed, the experimentally obtained total execution time across all replicas will be more than the CcRepExecTime metrics. Observe that the relative ordering of CcRepExecTime and the total execution time from the experiments are almost equivalent.

All these experiments show the usefulness of our metrics to suggest a coordination configuration according to the workload of a distributed application.

Chapter 13

Related work

Several works have addressed the safety of distributed applications [6, 9, 11, 14]. The tools based on these proof rules [7, 8, 12] provide pairs of conflicting methods. Nair [71] provides the conditions under which the methods conflict.

The next step is to determine how to use the pairs of conflicting methods to design coordination policies that minimizes overhead. Gotsman et al. [6] use the abstract notion of tokens for coordination control. Token is a point of global synchronization. In this work, we materialize this concept of tokens as distributed locks.

Houshmand and Lesani [9] present coordination protocols parameterized by the analysis results. Using the pairs of conflicting methods, they construct a conflict graph. The nodes represent the operations and the edges represent the conflict. Based on the conflict graph, they instantiate two protocols, a blocking one and a non-blocking one. The blocking protocol resembles a shared/exclusive lock and the non-blocking protocol resembles a mutual exclusive lock. We are inspired by the idea of conflict graph they generated and using the graph parameters to instantiate coordination protocols. However they do not consider the other dimensions of concurrency control such as, placement and granularity. They also neglect the impact of workload. As we have seen in Chapter 12, all three dimensions impact the overhead of locking.

Houshmand and Lesani [9] uses minimum vertex cover to parametrize their blocking protocol. We are inspired by this approach, which we use to assign the mode of a lock.

Xie et al. [72], Su et al. [73] present federated concurrency control mechanisms, called Modular Concurrency Control (MCC), for maintaining the ACID properties of a transaction. MCC partitions transactions into small groups and each group applies the best concurrency control independently. We take inspiration from their work to select the best configuration for each lock in a modular fashion.

There are several lock services that do distributed locking [62–67, 74–76]. We are interested in the lock configuration itself, which can be realized with these lock services.

Grzesik and Mrozek [77] presents a comparative study of RedLock [62], locks in Zookeeper [67], etcd [66], and Consul lock service [78]. They study the performance, safety, deadlock-free and fault tolerance properties. They conclude that for all lock services except RedLock, concurrent access to the same set of keys had little performance impact. We are interested in studying the performance of different configurations of a distributed lock, regardless of

the underlying lock provider.

Kulkarni et al. [79] presents a lattice-based definition of commutativity specifications for a data structure. A commutativity specification gives a set of conditions that needs to be satisfied for each pair of operations in a data structure to commute. The commutativity lattice is constructed based on the amount of parallelism the commutativity specification permits. They discuss a trade-off between parallelism allowed and the overhead incurred. This concept of a commutativity lattice described by Kulkarni et al. [79] inspired us for creating a coordination lattice where the opposing forces are lock contention and cost of lock acquisition.

Diniz and Rinard [80] presents a lock coarsening policy for parallel computing systems. They explain the trade-off between the cost of lock acquisitions and lock contention. The basic insight is that lock coarsening decreases the cost of lock acquisitions, while increasing the lock contention. They propose two different coarsening methods - data coarsening, where the lock is coarsened when multiple objects are accessed together, and computational coarsening, when a single operation acquires a single lock multiple times. Our coordination lattice navigation is inspired by Diniz and Rinard [80], especially the insight on using trade-off as a metric to navigate through the lattice.

Chapter 14

Conclusion of Part III and Future work

Choosing a coordination configuration for a distributed application that minimizes the performance impact is far from trivial. It requires navigating different dimensions of the configuration and is dependant on the workload. We presented the concept of a *coordination lattice*, which helps systematically navigate the granularity, mode and placement dimensions of distributed locking.

We present a set of metrics that helps the user to choose a coordination configuration from the three dimensions.

We present a tool, DisLockGen, that suggests optimal distributed lock configuration for a given workload. To aid the study, we present a simulator named DisLockSim. The simulator helps to test the performance of different coordination configurations against a given workload.

With some sample conflict graphs, we show how our prediction pattern correlates with the actual measurement of the execution time for a given workload.

14.1 Future work

Given the exploratory nature of this work, this can be continued in many directions.

The first direction is regarding the system model. To resemble the real world more closely, we need to use probabilistic models for the workload. The probability of conflicting operations to happen concurrently will impact the prediction metrics. The probabilistic model of the workload would also help us understand if caching locks inside a replica would be beneficial, and if so, on parametrizing the time to cache. The performance of the lock configurations can be modelled using performance modeling techniques like the works of Agrawal et al. [81], Aksenov et al. [82], Witt et al. [83], and Nudd et al. [84].

The second path is to extend the solution to include dynamic selection of coordination configuration according to the workload. In this case, the workload would be monitored dynamically and the configuration would be adjusted online similar to the works of Diniz and Rinard [85], and Tang and Elmore [86].

For fault tolerant design, we must take other coordination methods into consideration. A distributed lock might be acquired by a process that then becomes dead. If the lock is held forever by the dead process, the system cannot progress. This can be overcome either by introducing leases (timed locks) or by using fault-tolerant consensus. In the case of leases, the lock expires after a certain amount of time, so even if the lock holder dies, the system can progress. This inclusion will add another non-trivial dimension to our coordination lattice - optimal time for lease.

Yet another useful line of work is to have a lock service that uses different distributed locks according to the semantics and guarantees required by the application. This can be done by studying the trade-offs of different lock providers and choosing the lock provider with the optimal configuration for locking.

Instead of limiting ourselves to distributed locking as the coordination method, we can broaden the work to include other coordination strategies such as consensus, total order broadcast, etc.

Bibliography

[1] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002. ISSN 0163-5700. doi: 10.1145/564585.564601. URL `https://doi.org/10.1145/564585.564601`.

[2] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems*, pages 386–400, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-24550-3.

[3] Mahsa Najafzadeh, Marc Shapiro, and Patrick Eugster. Co-design and verification of an available file system. In *VMCAI*, volume 10747, pages 358–381, Los Angeles, CA, USA, January 2018. URL `https://doi.org/10.1007/978-3-319-73721-8_17`.

[4] Carlos Baquero, Paulo Sérgio Almeida, Alcino Cunha, and Carla Ferreira. Composition in state-based replicated data types. *Bulletin of the EATCS*, 123, 2017. URL `http://eatcs.org/beatcs/index.php/beatcs/article/view/507`.

[5] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Delta state replicated data types. *J. Parallel Distrib. Comput.*, 111:162–173, 2018. URL `https://doi.org/10.1016/j.jpdc.2017.08.003`.

[6] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'Cause I'm Strong Enough: Reasoning about consistency choices in distributed systems. In *POPL*, pages 371–384, St. Petersburg, FL, USA, 2016. URL `http://dx.doi.org/10.1145/2837614.2837625`.

[7] Mahsa Najafzadeh, Alexey Gotsman, Hongseok Yang, Carla Ferreira, and Marc Shapiro. The CISE tool: Proving weakly-consistent applications correct. EuroSys 2016 workshops, London, UK, April 2016. URL `http://dx.doi.org/10.1145/2911151.2911160`.

[8] Gonçalo Marcelino, Valter Balegas, and Carla Ferreira. Bringing hybrid consistency closer to programmers. PaPoC '17, pages 6:1–6:4, Belgrade, Serbia, 2017. ACM. URL `http://doi.acm.org/10.1145/3064889.3064896`.

[9] Farzin Houshmand and Mohsen Lesani. Hamsaz: Replication coordination analysis

and synthesis. *Proc. ACM Program. Lang.*, 3(POPL):74:1–74:32, January 2019. ISSN 2475-1421. URL `http://doi.acm.org/10.1145/3290387`.

[10] Sreeja S. Nair, Gustavo Petri, and Marc Shapiro. Invariant safety for distributed applications. In *Proceedings of the 6th Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362764. doi: 10.1145/3301419.3323970. URL `https://doi.org/10.1145/3301419.3323970`.

[11] Sreeja S. Nair, Gustavo Petri, and Marc Shapiro. Proving the safety of highly-available distributed objects. In Peter Müller, editor, *Programming Languages and Systems*, pages 544–571, Cham, 2020. Springer International Publishing. ISBN 978-3-030-44914-8.

[12] Sreeja S Nair, Gustavo Petri, and Marc Shapiro. Soteria. `https://github.com/sreeja/soteria_tool`, 2019.

[13] Carlos Baquero and Francisco Moura. Using structural characteristics for autonomous operation. *Operating Systems Review*, 33(4):90–96, 1999. URL `https://doi.org/10.1145/334598.334614`.

[14] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Coordination avoidance in database systems. *Proceedings of VLDB*, 8(3): 185–196, November 2014. URL `http://dx.doi.org/10.14778/2735508.2735509`. 2015, Waikoloa, Hawai'i, USA.

[15] E.W. Dijkstra. *A discipline of programming*. Prentice-Hall series in automatic computation. Prentice-Hall, 1976. ISBN 0-13-215871-X.

[16] Cliff B. Jones. Specification and design of (parallel) programs. In R.E.A. Mason, editor, *Information Processing 83*, volume 9 of *IFIP Congress Series*, pages 321–332, Paris, France, September 1983. IFIP, North-Holland/IFIP.

[17] Stephen Brookes and Peter W. O'Hearn. Concurrent separation logic. *SIGLOG News*, 3(3):47–65, 2016. URL `https://dl.acm.org/citation.cfm?id=2984457`.

[18] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free Replicated Data Types. Research Report RR-7687, July 2011. URL `https://hal.inria.fr/inria-00609399`.

[19] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*, FMCO'05, pages 364–387, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-36749-7, 978-3-540-36749-9. URL `http://dx.doi.org/10.1007/11804192_17`.

[20] K. Rustan M. Leino and Rosemary Monahan. Reasoning about comprehensions with first-order SMT solvers. In *Proceedings of the 2009 ACM Symposium on Applied*

Computing, SAC '09, pages 615–622, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-166-8. URL http://doi.acm.org/10.1145/1529282.1529411.

[21] Joseph M. Hellerstein and Peter Alvaro. Keeping CALM: when distributed consistency is easy. *CoRR*, abs/1901.01930, 2019. URL http://arxiv.org/abs/1901.01930.

[22] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: Specification, verification, optimality. In *POPL*, pages 271–284, San Diego, CA, USA, January 2014. URL http://doi.acm.org/10.1145/2535838.2535848.

[23] Sebastian Burckhardt. Principles of eventual consistency. *Foundations and Trends in Programming Languages*, 1(1-2):1–150, 2014. URL https://doi.org/10.1561/2500000011.

[24] KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative programming over eventually consistent data stores. PLDI '15, pages 413–424, Portland, OR, USA, 2015. URL http://doi.acm.org/10.1145/2737924.2737981.

[25] Gowtham Kaki, Kapil Earanky, KC Sivaramakrishnan, and Suresh Jagannathan. Safe replication through bounded concurrency verification. *Proc. ACM Program. Lang.*, 2 (OOPSLA):164:1–164:27, October 2018. ISSN 2475-1421. URL http://doi.acm.org/10.1145/3276534.

[26] Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. A framework for establishing strong eventual consistency for conflict-free replicated datatypes. *Archive of Formal Proofs*, 2017, 2017. URL https://www.isa-afp.org/entries/CRDT.shtml.

[27] Radha Jagadeesan and James Riely. Eventual consistency for CRDTs. In Amal Ahmed, editor, *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10801 of *Lecture Notes in Computer Science*, pages 968–995. Springer, 2018. ISBN 978-3-319-89883-4. URL https://doi.org/10.1007/978-3-319-89884-1_34.

[28] Sreeja Nair and Marc Shapiro. Improving the "Correct Eventual Consistency" tool. Rapport de recherche RR-9191, Paris, France, July 2018. URL https://hal.inria.fr/hal-01832888.

[29] Gowtham Kaki, Kartik Nagar, Mahsa Najafzadeh, and Suresh Jagannathan. Alone together: Compositional reasoning and inference for weak isolation. In *POPL*, volume 2, pages 27:1–27:34, Los Angeles, CA, USA, December 2017. URL http://doi.acm.org/10.1145/3158115.

[30] Marc Shapiro, Masoud Saeida Ardekani, and Gustavo Petri. Consistency in 3D. In Josée Desharnais and Radha Jagadeesan, editors, *Int. Conf. on Concurrency Theory*

(CONCUR) 2016, volume 59, pages 3:1–3:14, Québec, Québec, Canada, August 2016. URL `http://dx.doi.org/10.4230/LIPIcs.CONCUR.2016.3`.

[31] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975. doi: 10.1145/360933.360975. URL `https://doi.org/10.1145/360933.360975`.

[32] Geoff W Hamilton. Generating loop invariants for program verification by transformation. *arXiv preprint arXiv:1708.07223*, 2017.

[33] Boogie. URL `https://github.com/boogie-org/boogie`.

[34] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78800-3.

[35] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011. `https://hal.inria.fr/hal-00790310`.

[36] Sylvain Conchon, Albin Coquereau, Mohamed Iguernlala, and Alain Mebsout. Alt-Ergo 2.2. In *SMT Workshop: International Workshop on Satisfiability Modulo Theories*, Oxford, United Kingdom, July 2018. URL `https://hal.inria.fr/hal-01960203`.

[37] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, page 171–177, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 9783642221095.

[38] The Coq Consortium, 2020. URL `https://coq.inria.fr/`.

[39] Sreeja S Nair, Filipe Meirim, Mário Pereira, Carla Ferreira, and Marc Shapiro. A coordination-free, convergent, and safe replicated tree. Research Report RR-9395, LIP6, Sorbonne Université, Inria de Paris ; Universidade nova de Lisboa, February 2021. URL `https://hal.archives-ouvertes.fr/hal-03150817`.

[40] Vinh Tao, Marc Shapiro, and Vianney Rancurel. Merging semantics for conflict updates in geo-distributed file systems. In *SYSTOR*, pages 10.1–10.12, Haifa, Israel, May 2015. URL `http://dx.doi.org/10.1145/2757667.2757683`.

[41] Nikolaj Bjørner. Models and software model checking of a distributed file replication system. In *Formal Methods and Hybrid Real-Time Systems*, pages 1–23, 2007. URL `http://dx.doi.org/10.1007/978-3-540-75221-9_1`.

[42] Martin Kleppmann, Victor B. F. Gomes, Dominic P. Mulligan, and Alastair R. Beresford. OpSets: Sequential specifications for replicated datatypes (extended version). *CoRR*, abs/1805.04263, 2018. URL `http://arxiv.org/abs/1805.04263`.

[43] Hagit Attiya, Sebastian Burckhardt, Alexey Gotsman, Adam Morrison, Hongseok Yang, and Marek Zawirski. Specification and complexity of collaborative text editing. In *PODC*, pages 259–268, Chicago, IL, USA, July 2016. URL `http://dx.doi.org/10.1145/2933057.2933090`.

[44] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. Session guarantees for weakly consistent replicated data. In *PDIS*, pages 140–149, Austin, Texas, USA, September 1994.

[45] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 – Where Programs Meet Provers. In *ESOP'13 22nd European Symposium on Programming*, volume 7792 of *LNCS*, Rome, Italy, March 2013. Springer. URL `https://hal.inria.fr/hal-00789533`.

[46] Filipe Meirim, Mário Pereira, and Carla Ferreira. CISE3: Verifying weakly consistent applications with Why3, 2020. URL `https://arxiv.org/abs/2010.06622`.

[47] Jean-Christophe Filliâtre. Deductive software verification. *International Journal on Software Tools for Technology Transfer*, 13(5):397, Aug 2011. ISSN 1433-2787. URL `https://doi.org/10.1007/s10009-011-0211-0`.

[48] Vinh Thanh Tao. *Ensuring Availability and Managing Consistency in Geo-Replicated File Systems*. PhD thesis, Sorbonne-Université–Université Pierre et Marie Curie, Paris, France, December 2017. URL `https://hal.inria.fr/tel-01673030`.

[49] Filipe Meirim, Mário Pereira, Carla Ferreira, Sreeja S Nair, and Marc Shapiro. Proofs of Maram, 2020. URL `https://fmeirim.github.io/Maram_proofs/`.

[50] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. Making operation-based CRDTs operation-based. volume 8460, pages 126–140, Berlin, Germany, June 2014. URL `http://dx.doi.org/10.1007/978-3-662-43352-2_11`.

[51] Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Mooly Sagiv. Eventually consistent transactions. In *ESOP*, Tallinn, Estonia, March 2012.

[52] Stéphane Martin, Mehdi Ahmed-Nacer, and Pascal Urso. Abstract unordered and ordered trees CRDT. Research Report RR-7825, INRIA, December 2011. URL `https://hal.inria.fr/hal-00648106`.

[53] Puneet Kumar and M. Satyanarayanan. Log-based directory resolution in the coda file system. In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, PDIS '93, page 202–213, Washington, DC, USA, 1993. IEEE Computer Society Press. ISBN 081863301.

[54] Sujay Jayakar. Rewriting the heart of our sync engine, 2020. URL https://dropbox.tech/infrastructure/rewriting-the-heart-of-our-sync-engine.

[55] Gowtham Kaki, Swarn Priya, KC Sivaramakrishnan, and Suresh Jagannathan. Mergeable replicated data types. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019. URL https://doi.org/10.1145/3360580.

[56] Farzin Houshmand and Mohsen Lesani. Hamsaz: Replication coordination analysis and synthesis. In *POPL*, volume 3, pages 74:1–74:32, Cascais, Portugal, January 2019. URL http://doi.acm.org/10.1145/3290387.

[57] Sreeja S Nair, Gustavo Petri, and Marc Shapiro. Invariant safety for distributed applications. In *PaPoC*, pages 4:1–4:7, Dresden, Germany, March 2019. URL https://doi.org/10.1145/3301419.3323970.

[58] Cheng Li, Nuno Preguiça, and Rodrigo Rodrigues. Fine-grained consistency for geo-replicated systems. In *Proceedings of the 2018 USENIX Annual Technical Conference*, pages 359–372, Boston, MA, USA, July 2018. URL https://www.usenix.org/conference/atc18/presentation/li-cheng.

[59] Cheng Li, João Leitão, Allen Clement, Nuno M. Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. Automating the choice of consistency levels in replicated systems. In Garth Gibson and Nickolai Zeldovich, editors, *Proceedings of the 2014 USENIX Annual Technical Conference*, pages 281–292, Philadelphia, PA, USA, June 2014. URL https://www.usenix.org/node/183990.

[60] Sreeja S Nair and Marc Shapiro. Improving the "Correct Eventual Consistency" Tool. Research Report RR-9191, Sorbonne Université, July 2018. URL https://hal.inria.fr/hal-01832888.

[61] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, June 1981. ISSN 0360-0300. doi: 10.1145/356842.356846. URL https://doi.org/10.1145/356842.356846.

[62] Distributed locks with Redis. URL https://redis.io/topics/distlock.

[63] Ensar Basri Kahveci. Distributed locks are dead; long live distributed locks!, April 2019. URL https://hazelcast.com/blog/long-live-distributed-locks/.

[64] Martin Kysel. Distributed transactional locks, 2018. URL https://nuodb.com/blog/distributed-transactional-locks.

[65] Alexander Patrikalakis and Sasha Slutsker. Building distributed locks with the DynamoDB lock client, 2017. URL https://aws.amazon.com/blogs/database/building-distributed-locks-with-the-dynamodb-lock-client/.

[66] etcd concurrency API reference. URL https://etcd.io/docs/v3.2.17/dev-guide/api_concurrency_reference_v3/.

[67] Zookeeper. Zookeeper lock recipe. https://zookeeper.apache.org/doc/r3.4.4/recipes.html#sc_recipes_Locks.

[68] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), page 483–485, New York, NY, USA, 1967. Association for Computing Machinery. ISBN 9781450378956. doi: 10.1145/1465482.1465560. URL https://doi.org/10.1145/1465482.1465560.

[69] Yahoo! cloud serving benchmark. https://github.com/brianfrankcooper/YCSB/releases/tag/0.17.0, 2020.

[70] Sreeja S. Nair. Read-write lock, 2021. URL https://github.com/sreeja/etcd-exp/tree/main/rwlock.

[71] Sreeja S Nair. Evaluation of the CEC (Correct Eventual Consistency) Tool. Research Report RR-9111, Inria Paris ; LIP6 UMR 7606, UPMC Sorbonne Universités, France, November 2017. URL https://hal.inria.fr/hal-01628719.

[72] Chao Xie, Chunzhi Su, Cody Littley, Lorenzo Alvisi, Manos Kapritsos, and Yang Wang. High-performance acid via modular concurrency control. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 279–294, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450338349. doi: 10.1145/2815400.2815430. URL https://doi.org/10.1145/2815400.2815430.

[73] Chunzhi Su, Natacha Crooks, Cong Ding, Lorenzo Alvisi, and Chao Xie. Bringing modular concurrency control to the next level. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 283–297, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450341974. doi: 10.1145/3035918.3064031. URL https://doi.org/10.1145/3035918.3064031.

[74] Michael Burrows. The Chubby lock service for loosely-coupled distributed systems. pages 335–350, Seattle, WA, USA, November 2006. URL http://www.usenix.org/events/osdi06/tech/burrows.html.

[75] Shedlock. URL https://github.com/lukas-krecan/ShedLock.

[76] Locking functions in MySQL. URL https://dev.mysql.com/doc/refman/5.7/en/locking-functions.html.

[77] Piotr Grzesik and Dariusz Mrozek. Evaluation of key-value stores for distributed locking purposes. In Stanisław Kozielski, Dariusz Mrozek, Paweł Kasprowski, Bożena Małysiak-Mrozek, and Daniel Kostrzewa, editors, *Beyond Databases, Architectures and Structures. Paving the Road to Smart Data Processing and Analysis*, pages 70–81, Cham, 2019. Springer International Publishing. ISBN 978-3-030-19093-4.

[78] HashiCorp. Consul lock, 2020. URL https://www.consul.io/commands/lock.

[79] Milind Kulkarni, Donald Nguyen, Dimitrios Prountzos, Xin Sui, and Keshav Pingali. Exploiting the commutativity lattice. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 542–555, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. URL `http://doi.acm.org/10.1145/1993498.1993562`.

[80] Pedro C. Diniz and Martin C. Rinard. Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs. *Journal of Parallel and Distributed Computing*, 49(2):218 – 244, 1998. ISSN 0743-7315. URL `http://www.sciencedirect.com/science/article/pii/S0743731598914411`.

[81] Rakesh Agrawal, Michael J. Carey, and Miron Livny. Concurrency control performance modeling: Alternatives and implications. *ACM Trans. Database Syst.*, 12(4):609–654, November 1987. ISSN 0362-5915. doi: 10.1145/32204.32220. URL `https://doi.org/10.1145/32204.32220`.

[82] Vitalii Aksenov, Dan Alistarh, and Petr Kuznetsov. Brief Announcement: Performance Prediction for Coarse-Grained Locking. In *PODC 2018 - ACM Symposium on Principles of Distributed Computing*, Egham, United Kingdom, July 2018. doi: 10.1145/3212734.3212785. URL `https://hal.inria.fr/hal-01887733`.

[83] Carl Witt, Marc Bux, Wladislaw Gusew, and Ulf Leser. Predictive performance modeling for distributed batch processing using black box monitoring and machine learning. *Information Systems*, 82:33–52, May 2019. ISSN 0306-4379. doi: 10.1016/j.is.2019.01.006. URL `http://dx.doi.org/10.1016/j.is.2019.01.006`.

[84] G. R. Nudd, D. J. Kerbyson, E. Papaefstathiou, S. C. Perry, J. S. Harper, and D. V. Wilcox. Pace–a toolset for the performance prediction of parallel and distributed systems. *Int. J. High Perform. Comput. Appl.*, 14(3):228–251, August 2000. ISSN 1094-3420. doi: 10.1177/109434200001400306. URL `https://doi.org/10.1177/109434200001400306`.

[85] Pedro C. Diniz and Martin C. Rinard. Dynamic feedback: An effective technique for adaptive computing. *SIGPLAN Not.*, 32(5):71–84, May 1997. ISSN 0362-1340. URL `http://doi.acm.org/10.1145/258916.258923`.

[86] Dixin Tang and Aaron J. Elmore. Toward coordination-free and reconfigurable mixed concurrency control. In *Proceedings of the 2018 USENIX Annual Technical Conference*, USENIX ATC '18, page 809–822, USA, 2018. USENIX Association. ISBN 9781931971447.