UNIVERSITY OF CALIFORNIA,
IRVINE

The Global Object Tracker: Decentralized Version Control for Replicated Objects

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Software Engineering

by

Rohan Achar

Dissertation Committee:
Professor Cristina Videira Lopes, Chair
Associate Professor James A. Jones
Professor Sam Malek
Assistant Professor Ardalan Amiri Sani

2020

TABLE OF CONTENTS

Page

LIST OF FIGURES **vii**

LIST OF TABLES **ix**

ACKNOWLEDGMENTS **x**

VITA **xi**

ABSTRACT OF THE DISSERTATION **xiii**

1 Introduction **1**
- 1.1 Application Domains . . . 2
  - 1.1.1 Online Multiplayer Video Games . . . 2
  - 1.1.2 Multi-agent Simulation . . . 3
  - 1.1.3 Geo-replicated Databases . . . 4
- 1.2 Challenges in State Replication . . . 5
- 1.3 Decentralized Version Control for Replication . . . 7
- 1.4 Organization of this Dissertation . . . 10

2 Background and Motivation **11**
- 2.1 Causal Consistency . . . 11
- 2.2 Consistency versus Latency . . . 12
- 2.3 Update Latency . . . 13
  - 2.3.1 Reducing Cost of Reconciliation . . . 14
  - 2.3.2 Rules for Reducing communication . . . 15
  - 2.3.3 Meeting the Requirements . . . 17
- 2.4 Rudimentary Forms of Version Control . . . 19
  - 2.4.1 Spacerace: Basic Design . . . 21
  - 2.4.2 Spacerace: Delta updates from the server . . . 26
  - 2.4.3 Delta updates with Isolation . . . 30
  - 2.4.4 Delta updates with Isolation and Low Memory . . . 33
- 2.5 Summary . . . 37

3 Related Work **38**
3.1 Shared-State Programming Models . . . 38
3.1.1 Databases . . . 39
3.1.2 Multiversion Databases . . . 42
3.1.3 Software Transactional Memory . . . 43
3.2 Message-Passing Programming Models . . . 44
3.2.1 Publish-Subscribe . . . 45
3.2.2 Actor Models . . . 46
3.2.3 Conflict-free Replicated Data Types . . . 46
3.2.4 Global Sequence Protocol . . . 48
3.3 Operational Transformation . . . 50
3.4 File Based Version Control . . . 51
3.5 Object Based Version Control . . . 54
3.5.1 Concurrent Revisions . . . 54
3.5.2 Cloud Types . . . 55
3.5.3 CARMOT . . . 56
3.5.4 TARDiS . . . 57
3.6 Optimizations in Shared-Space Applications . . . 58
3.7 Research Gap . . . 58
3.7.1 Update latency . . . 59
3.7.2 Version Control Systems . . . 60
3.7.3 Design Goals . . . 61

4 GoT Programming Model **62**
4.1 GoT Example: Multi-bot Space Race . . . 64
4.1.1 Data Model . . . 64
4.1.2 GoT Nodes . . . 68
4.2 Dataframe: Object Repository . . . 73
4.2.1 Snapshot . . . 74
4.2.2 Object Version Graph . . . 74
4.3 Snapshot and Version Graph Interaction . . . 75
4.4 State Replication in GoT . . . 76
4.4.1 Communication between Version Graphs . . . 78
4.4.2 Conflict Detection . . . 81
4.4.3 Conflict Resolution: Big-Step 3-way Merge . . . 82
4.4.4 Responsibilities of Merging . . . 85
4.5 Consistency Model . . . 86
4.6 Formal Specification . . . 86
4.6.1 Dataframe . . . 87
4.6.2 Data Model and Types . . . 90
4.6.3 Snapshot . . . 90
4.6.4 Object Version Graph . . . 91
4.6.5 Operational Semantics of Big Step . . . 92
4.7 GoT and Alternative Programming Models . . . 94

5 Spacetime: Implementation of GoT **97**
5.1 Components of Update Latency . . . 98
5.2 Optimizing for Update Latency . . . 100
5.2.1 Preparation of the Updates . . . 100
5.2.2 Transportation of Updates . . . 101
5.2.3 Conflict resolution . . . 105

6 Challenge 1: Peer to Peer Networks **107**
6.1 Criss Cross Merge Scenario . . . 108
6.2 Failure of Big-Step Merge . . . 110
6.3 The Root of the Problem . . . 111
6.4 Small Step Merge: Consistent P2P Version Control . . . 112
6.4.1 Success of Small-Step Merge . . . 114
6.4.2 Small-Step Merge with Three Peers . . . 116
6.5 Conclusion: Constraints and Properties . . . 117
6.6 Formal Proof of Small-Step 3-way Merge . . . 119
6.6.1 Additional Definitions . . . 119
6.6.2 Helper Functions . . . 119
6.6.3 Constraints . . . 121
6.6.4 Operations on Version Graph . . . 122
6.6.5 Single Lowest Common Ancestor . . . 124
6.6.6 Correctness . . . 125
6.6.7 Convergence . . . 128

7 Challenge 2: Unbound Growth of the Version Graph **135**
7.1 Related Work . . . 136
7.2 Basics of Garbage Collection . . . 137
7.3 Server Client Model: Practical Garbage Collection in Big Step . . . 141
7.4 P2P Model: Garbage Collection in Small Step . . . 143
7.4.1 Reducing the Version Graph . . . 143
7.4.2 Reference Passing in Peer to Peer . . . 146
7.4.3 Implementation Challenges . . . 147
7.5 Summary . . . 147

8 Challenge 3: Interest Management **150**
8.1 Basic Interest Management in GoT . . . 151
8.2 Introduction to Predicate Collection Classes . . . 152
8.3 Predicate Collection Classes: Overview . . . 155
8.4 Object Reclassification . . . 159
8.4.1 Object Model and Reclassification . . . 160
8.4.2 Reference vs. Copy Semantics . . . 161
8.4.3 Constant Consistency vs. One-Time Consistency . . . 162
8.4.4 Inheritance . . . 163
8.5 Relational Operations of PCCs . . . 164
8.5.1 Subset . . . 166

8.5.2 Projection . . . 166
8.5.3 Cross Product (Join) . . . 167
8.5.4 Union . . . 168
8.5.5 Intersection . . . 170
8.5.6 Parameterized Collections . . . 172
8.5.7 Final Note on PCC Creation . . . 173
8.6 Usage Examples . . . 174
8.6.1 K-Nearest Neighbor . . . 175
8.6.2 Car and Pedestrian . . . 178
8.6.3 Distributed Simulations using Spacetime . . . 182
8.7 Related Work . . . 184
8.7.1 Predicate Classes . . . 184
8.7.2 Other Class-Instance Associations . . . 186
8.8 Conclusions . . . 187

9 Observable Replication of Objects 189
9.1 Related Work: Debugging Distributed Systems . . . 190
9.2 Fundamental Requirements of Interactive Debuggers . . . 192
9.2.1 Requirement 1: Observing State Changes . . . 192
9.2.2 Requirement 2: Controlling the Flow of Execution . . . 196
9.3 Constraints on Distributed Systems . . . 198
9.3.1 Read Stability . . . 198
9.3.2 Separation of Published State and Local State . . . 200
9.3.3 Explicit Mechanisms for State Change . . . 200
9.3.4 GoT: Enabling an Interactive Debugger . . . 201
9.4 GoT example: Distributed Word Counter . . . 202
9.5 GoTcha . . . 208
9.5.1 Operation . . . 210
9.5.2 Observing Node State . . . 211
9.5.3 Debugging Word Frequency Counter . . . 214
9.6 GoTcha: Meeting the Fundamental Requirements . . . 217
9.6.1 Observing State Changes . . . 217
9.6.2 Controlling the Flow of Execution . . . 218
9.7 Beyond Interactivity . . . 220
9.7.1 Scalability of Interactivity . . . 220
9.7.2 Integration with Alternate Debugging Concepts . . . 223

10 Experimental Analysis and Results 225
10.1 Update Latency . . . 225
10.1.1 Setup . . . 225
10.1.2 Experiment 1: Spacetime vs Baselines . . . 231
10.1.3 Experiment 2: Breakdown of update latency in Spacetime . . . 234
10.2 Garbage Collection . . . 238
10.2.1 Setup . . . 238
10.2.2 Experiment 3A: Version count vs. the number of objects . . . 239

10.2.3 Experiment 3B: Version count vs. the number of nodes 241
10.3 Experiment 4: Garbage Collection in Peer to Peer 242
10.3.1 Setup 243
10.4 Conclusion 245

11 Conclusion and Future Work **248**
11.1 Summary 248
11.2 Discussion and Future Work 253

Bibliography **256**

LIST OF FIGURES

Page

1.1 Examples of virtual environments for AI-bot competitions. Left: game inspired by the classic Attari's Space Race game. Right: traffic simulation with different fleet operators. 2

2.1 Attari's Space Race game. 20
2.2 A simple architecture for Spacerace. 21
2.3 Simple design - Client side. 22
2.4 Simple design - Server side. 24
2.5 Delta updates over push for Spacerace. 26
2.6 Delta updates, Push only design - Client side. 27
2.7 Delta updates, Push only design - Server side. 28
2.8 Delta updates with isolation for Spacerace. 31
2.9 Delta updates, with Pull - Client side. 31
2.10 Delta updates, with Pull - Client side. 32
2.11 Delta updates with isolation and low memory for Spacerace. 33
2.12 Delta updates, Pull, low memory - Client side. 34
2.13 Delta updates, Pull, low memory - Server side. 35

4.1 Structure of a GoT node. Arrows denote the direction of data flow. 63
4.2 Spacerace using version control. 65
4.3 The data model for the multiplayer Space Race game. 66
4.4 Structure of the Space Race distributed application. Each node resides on a separate process/machine/geographic location. 67
4.5 The Physics simulator node. 68
4.6 The Player nodes. 70
4.7 The Viewer nodes. 72
4.8 Version Graph at Node N_1. 77
4.9 Inter-node Communication in GoT. 79
4.10 Conflict Detection and Resolution. 80
4.11 Programmatic conflict resolution for the Physics node. 82
4.12 3-way merge function for Counter. 83
4.13 Formal specification of Dataframe. 88
4.14 Formal Specification of Global Object Tracker (GoT) operations. 89
4.15 Map of Alternative Programming Models when compared to GoT. 94

5.1 Components of update latency. 99
5.2 Pushing updates $C \rightarrow D \rightarrow E$ from N1 to N2. 102
5.3 Pushing updates $E \rightarrow G$ from N1 to N2. 103
5.4 Delta transformation with commutative merges δ_1 and δ_2 106

6.1 Broken Version Control in Peer to Peer Networks. 109
6.2 Revised Version Control in Peer to Peer Networks. 113
6.3 Three concurrent updates with Revised Version Control. 116
6.4 Four scenarios that can arise during $G.put(\partial G)$. 130

7.1 Basic Garbage Collection. 138
7.2 Garbage Collection of Concurrent Update. 140
7.3 Garbage Collection in Small Step. 145

8.1 Object model: an instance of a class is a combination of state and behavior (methods). Different behavior may be dynamically associated with the same instance state. 159
8.2 PCC creation by reference (left) vs. by copy (right). 160
8.3 Urban simulation. 183

9.1 Information propagation in single-thread systems. 193
9.2 Information propagation in distributed systems. 194
9.3 Network topology of an example distributed word counting application built on Spacetime. 202
9.4 The types used by the Word Counting application. 203
9.5 The Grouper node. 205
9.6 The Word Counter node. 207
9.7 Merge function used at the Grouper node. 208
9.8 Architecture of GoTcha. 209
9.10 Debugger showing the network topology of the application. 211
9.11 Debugger view showing version history at the end of a *commit*. 212
9.12 Debugger view at Grouper showing response to a *push* request. 214
9.13 Merge function used at the Grouper node. 216

10.1 Median read latency vs write loads (All models). 229
10.2 (A)Median update latency vs write loads (All models). (B) Median update latency vs write loads (low latency models). 230
10.3 Median update latency vs Number of nodes. 234
10.4 Version count at Server with varying number of objects. 240
10.5 Version count at Server with varying number of nodes. 242
10.6 Version count in peer to peer with two peers and no garbage collection. . . . 244
10.7 Version count in peer to peer with two peers and garbage collection. 244

LIST OF TABLES

Page

3.1 Design Goals for GoT 59

4.1 API table for a dataframe 64
4.2 Metavariables 87

6.1 Additional Metavariables 120

8.1 Summary of decorators used by PCCs. 156
8.2 Dimensions rules (Γ) for PCCs, which dictate the fields of PCCs. 165
8.3 Extension rules (Ψ) for PCCs, which define which instances are in the PCC. 165

9.1 Mapping the primitives of GoT to the types of State changes 217

10.1 Time taken by operations at the writer node 235
10.2 Time taken by operations at the server node, when receiving updates. 235
10.3 Time taken by operations at the server node, when responding to fetch. . . . 235
10.4 Time taken by operations at the reader node. 235

ACKNOWLEDGMENTS

Many people have given me time and support and played essential roles in the completion of this dissertation. I thank all my friends, family, peers, and well-wishers for helping me through this significant milestone. The first among many, I would like to thank, is my advisor, Cristina Lopes. Right from giving me the creative freedom to do what I thought was best to giving me critical feedback to identify and improve upon my shortcomings, it is her academic, financial, and emotional support that has made all of this dissertation possible. I especially thank her for the patience she showed when reviewing every one of my publications and academic submissions.

I would also like to thank my parents for the emotional support they have given me. They have always pushed me to be the best person I could be, never believing that I would fail. I am eternally grateful to have their faith and support. I would also like to thank my girlfriend, Dheeru Dua. She has been there for me throughout this journey, the unsung hero. Every idea I have had, every zany scheme I came up with, she was my partner in crime, helping me sort through my thoughts and refine my designs.

Finally, I thank all my collaborators and peers: Cristina Lopes, Pritha Dawn, and Shrijith Venkatramana for the time and the effort they have put in to help me with this dissertation. I thank Pritha Dawn for her contribution and for co-authoring published papers used in chapter 9 in this dissertation. I also thank Shrijith Venkatramana for helping me design and implement the algorithms described in chapter 7.

VITA

Rohan Achar

EDUCATION

Doctor of Philosophy in Software Engineering **2020**
University of California, Irvine *Irvine, California*

Masters of Science in Informatics **2015**
University of California, Irvine *Irvine, California*

Bachelors of Technology in Computer Science and Technology **2011**
National Institute of Technology, Karnataka *Surathkal, Karnataka*

RESEARCH EXPERIENCE

Graduate Research Assistant **2014–2020**
University of California, Irvine *Irvine, California*

TEACHING EXPERIENCE

Teaching Assistant **2013–2016**
University of California, Irvine *Irvine, California*

REFEREED JOURNAL PUBLICATIONS

Predicate Collection Classes **Feb 2017**
Journal of Object Technology

REFEREED CONFERENCE PUBLICATIONS

A simulation analysis of large contests with thresholding agents. **Dec 2019**
WSC 2019

GoTcha: An Interactive Debugger for GoT-Based Distributed Systems **Oct 2019**
Onward!

Toward understanding the impact of user participation in autonomous ridesharing systems. **Dec 2018**
WSC 2018

50K-C: A dataset of compilable, and compiled, Java projects. **May 2018**
MSR 2018

CADIS: Aspect-Oriented architecture for collaborative modeling and simulation. **Dec 2016**
WSC 2016

Automatic Builds of Large Software Repositories **June 2015**
UC Irvine

SOFTWARE

Spacetime `https://github.com/Mondego/spacetime`
Python implementation of the Global Object Tracker Programming Model.
Spacetime P2P `https://github.com/Mondego/spacetime/tree/peer2peer`
Python implementation of small-step merge for peer to peer GoT.
Predicate Collection Classes `https://github.com/Mondego/pcc`
Python implementation for Predicate Collection Classes.
SourcererJBF `https://github.com/Mondego/SourcererJBF`
Tool to automate the building of Java projects in large software repositories.

ABSTRACT OF THE DISSERTATION

The Global Object Tracker: Decentralized Version Control for Replicated Objects

By

Rohan Achar

Doctor of Philosophy in Software Engineering

University of California, Irvine, 2020

Professor Cristina Videira Lopes, Chair

Synchronization in distributed applications with shared, highly mutable replicated state often requires complex engineering to maintain low latency for the propagation of updates. At their core, distributed applications with replicated state need to implement some form of version control to be able to deal with concurrent updates and still maintain some notion of consistency. This need leads to complex and ad-hoc code that is hard to maintain and easy to break. To simplify the creation of such applications, I propose the Global Object Tracker (GoT) model, an object-oriented programming model based on causal consistency, whose design and interfaces mirror those found in decentralized version control systems: a version graph, working data, diffs, commit, checkout, fetch, push, and merge. I have implemented GoT in a framework called Spacetime, written in Python.

GoT is designed to ensure causal consistency with low latency updates in all sorts of distributed application topologies. These include peer-to-peer and applications subject to, or requiring, temporary isolation of nodes. The key to being able to do this is a novel approach to storing and merging concurrent updates within the version graph. GoT stores the version history as a graph of updates instead of calculating the whole state at each version. This allows GoT to send delta updates between nodes without having to calculate the deltas. The ability to send delta updates along with the lack of heavy serialization and deserialization allows for

the updates to be quickly available. The final state at each node is only constructed when needed.

In addition to the lower latency of updates, GoT addresses three concerns. First, object-based version control systems are typically not suitable for peer to peer networks. The merge strategies used are incompatible with the communication patterns seen in peer to peer. GoT addresses this concern with a novel approach to merging concurrent updates, called the small-step merge. Second, GoT makes the state changes in the system more visible, allowing for powerful tools such as interactive debuggers to be built and integrated, helping developers find and fix errors. Finally, GoT provides these benefits to not just push-style communication but also pull-style, allowing for isolation and partition recovery.

In its traditional form, GoT is impractical for real systems, because of the unbounded growth of the version graph. I present my solution to this problem that adds constraints to GoT applications but makes the model feasible in practice. I show that Spacetime is not just feasible, but viable for real applications, and present performance results showing low update latency.

Chapter 1

Introduction

Distributed computing is the backbone of a wide variety of large-scale applications seen today: online games, e-commerce sites, video conferences, self driving cars, search engines, banking, and many, many more. All of these distributed computing scenarios revolve around a shared computation state that is worked upon by distributed components that communicate over the network. Over the years, through the learnings in both academia and the industry, several common architectural styles have emerged for distributed applications: client-server, peer-to-peer, map-reduce, etc. Different architectural styles are suited for different categories of distributed applications. State synchronization, however, is a common problem that all distributed systems need to address in some form or another and different architectural styles impose different constraints on how to solve it.

In this work, I am particularly interested in the problems associated with state synchronization for a specific category of distributed applications that are characterized by first, the existence of many components collaborating, or competing, over a shared, long-lived, and highly mutable state, second, use a specific form of communication guarantee popularly called causal consistency [3], and finally, have a need to account for, or benefit from network

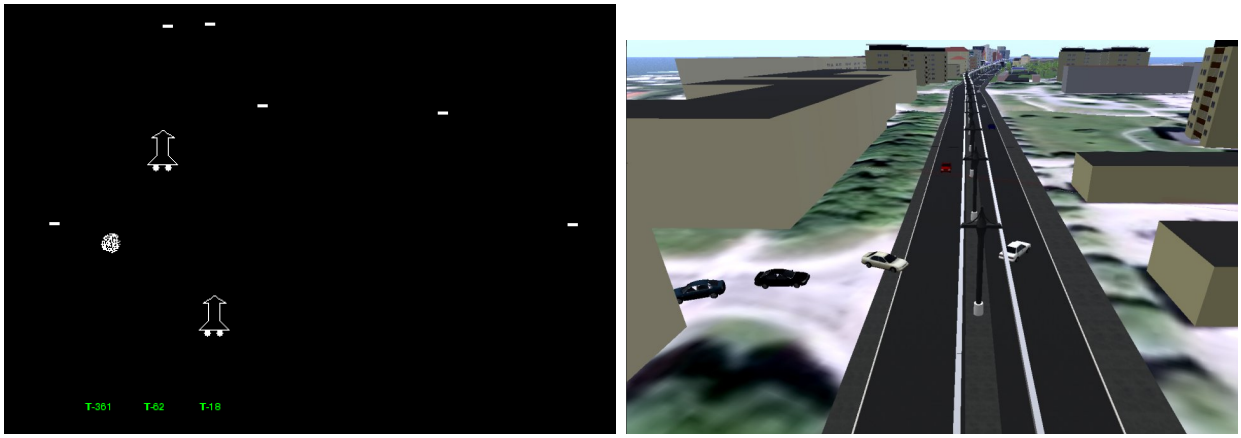


Figure 1.1: Examples of virtual environments for AI-bot competitions. Left: game inspired by the classic Attari's Space Race game. Right: traffic simulation with different fleet operators.

isolation. Examples of such systems include online multiplayer gaming, distributed multi-agent simulations, geo-replicated databases and more. Examples of such applications are shown in Figure 1.1.

1.1 Application Domains

1.1.1 Online Multiplayer Video Games

Ever since Spacewar! was programmed and released in 1962, video games have been gaining in popularity. For most of their existence though, videos games have primarily been applications designed to be operated from a single machine or device. In 1974, Mazewar became the first networked game, where players at different computers, connected over a serial cable, could interact with the same virtual space. Throughout the 1970s and 80s, a number of networked games were released, but it was not until the ubiquity of the Internet in the early 2000s, that online multiplayer games became popular. Today, almost every video game released has some form of multiplayer built into it. eSport competitions using online multiplayer games such as League of Legends, Fortnite, and Overwatch are a rapidly growing industry generating more

than $1 Billion in revenues in 2019.[1]

Online multiplayer games, both big and small, are, by definition, distributed systems that have shared, long-lived, highly mutable state. For each player in the game to take the best decision that they can, they need to know the latest state of the world constantly, so the shared state needs to be available locally. When the state of the world changes, those changes need to quickly propagate to all players, as new decisions by these players might be affected by the updates. Additionally, inconsistent changes made by different agents need to be resolved and the appropriate semantics for resolving inconsistencies is highly dependent on the logic of the game. The consistency model, which defines the rules for the communication of updates in a distributed system, is therefore, an important software engineering consideration in multiplayer games.

1.1.2 Multi-agent Simulation

Modeling and simulations are vital tools in strategic and tactical decision making, especially when the decisions have to be made over systems that are dynamic and complex. They help improve the understanding of these systems and in many cases make non-trivial predictions of possibilities within the system.

Agent based simulations are a type of simulation strategy where modeling researchers describe the low level rules that define the behaviors of simulated actors, called agents, and then observe the structures and states reached by the interaction of these agents with other agents and their environment. Distributed multi-agent simulations deploy these low level agents in different machines that interact over the network.

Like online multiplayer games, the agents in any multi-agent simulations also have a long

[1] `https://newzoo.com/insights/trend-reports/newzoo-global-esports-market-report-2019-light-version/`

lived, shared space that mutate often. Since the goal of the simulation is to make accurate predictions over the shared state, the agents need the latest state and updates from each agent must propagate to all other agents as quickly as possible. Concurrent updates have to be reconciled in a manner that is specific to each simulation. Finally, the consistency model is an important consideration as wrongly ordered updates can negatively affect the validity of the simulations.

1.1.3 Geo-replicated Databases

Geo-replication is a strategy often used by online service providing companies to support improved access to their content for users around the globe. The underlying reason is that, since the latency of network communication increases with the physical distance between the user and the servers, quality of service provided can be improved by moving the physical servers closer to the users, reducing this latency. The content on these servers is kept up-to-date using data synchronization between database nodes, and can be processed to be network efficient.

The acceptable delays of replication between these databases depends highly on the use of these databases themselves. For example, geo-replicating the number of likes on a youtube video does not require sub-second transfers, however, banking transactions or physical resource allocation systems, such as booking flight tickets or hotel rooms, might require faster transfers in order to avoid the rollbacks.

Depending on the application, geo-replicated databases can have large amounts of data, which is the shared state, and might have to handle a large number of read or write transactions per second. Reconciliation of conflicting updates is also often required. Significant research in both academia and industry has been made to build consistency models that aim to optimize the network efficiency of geo-replication.

1.2 Challenges in State Replication

All these categories of distributed applications have had many decades of industry attention, especially in the gaming industry and in military R&D. The main problem that needs to be solved in these applications is correct data synchronization with as good performance as possible. While the problem is easy to formulate, its solution in practice poses many challenges.

First, there is the problem of network latency: a typical 'ping' between the US West Coast and Amsterdam is 130ms, of which 67% is spent in fiber optic cable at almost the speed of light[2]. Any attempts to implement strong data consistency will result in severely slowing down the local execution of each component. For that reason, strong consistency is usually not followed; instead, these applications use weaker consistency approaches such as sequential consistency [59], eventual consistency [107], causal consistency [3], and others, all of which makes dealing with the shared state more difficult, but provide higher availability. Second, network delays increase noticeably with the amount of data that is sent across. While compression helps, ultimately the data that needs to be sent depends heavily on the application itself. This knowledge has the perverse effect of increasing the complexity of the applications, as performance-oriented engineers look for application-specific opportunities to reduce the amount of data that is transferred across the network. For example, a server may need to maintain complex information about the state of each client, so to tailor state updates for each of them. Third, there are no general rules for when and how to synchronize the data; the right approach depends on the application. The naive approach of synchronizing every single state change may result in unacceptable application performance. Typically, well-engineered games and simulations buffer changes that are sent only and exactly when needed – again, typically, at the expense of increased complexity in the code.

[2]https://wondernetwork.com/pings

These problems are well encapsulated in the most famous theorems in distributed computing: the CAP theorem [45], and its more generalized variation, the PACELC theorem [1]. In informal terms, these theorems state that software engineers building distributed systems must choose between latency – the time taken to receive a response for a request over the network – and consistency – the ability to read the latest write that has been made to the shared state.

Another common problem in distributed systems with a highly mutable shared space is the problem of network independence and isolation. In some distributed systems such as mobile games, collaborative documents, and geo-replicated datastores, it is not always possible for the components to stay connected. In the design of such distributed systems, isolation of components is an important aspect to consider. Take for example a collaborative document, shared between multiple users. When one user edit, the changes made must be propagated to all other users. A user that is offline or chooses to be offline should still be capable of making edits that can synchronize when reconnected. Moreover, resources should not be wasted by online components in trying to connect to the offline ones. A model where the offline components choose when to synchronize their state is better suited for these applications but designing such a system typically implies that the servers (or peers in a non server-client architecture) must hold the state of every node that connects to it. The engineering of these systems can be complex and error-prone.

While these are system design aspects that make state replication hard, the software engineering problems should not be forgotten. Distributed systems are notoriously difficult to get right. The non-determinism of update orderings due to network conditions, including partitions makes reasoning over the changes to shared state hard, and bugs hard to track. Efficient observation of the shared state and the operations over it is necessary.

1.3 Decentralized Version Control for Replication

My approach to tackling these problems is based on a simple idea: that synchronizing mutable state among distributed components can be modeled as a problem of version control. This idea is not new. Version Control has been proposed as a model to synchronize state replication of application objects in distributed systems before. Examples include Concurrent Revisions [19, 20], TARDiS [30], and Irmin [53]. The core idea in all these approaches is that the history of the updates to the state of the replicated objects is stored as a Directed Acyclic Graph (DAG) known as version graph (also known as a revision graph, revision history, or version history). Replicas synchronize state by sharing updates to the DAG, often as diffs, reducing the network overhead significantly. Version control gives each node in the system both a means to reason about updates over time which allows them to detect and resolve write-write conflicts, and a means to reduce the cost of communication by sharing delta encoded information.

All the models mentioned, however, use a centralized approach for version control. History of operations is maintained in a central server, and components in the distributed system synchronize changes over this version history. In contrast, in my approach I model the synchronization of mutable state between distributed components as a *decentralized* version control system with a copy of the version history locally available to every component in the distributed system, an idea made popular by tools such as Git. For example, if two components, C_1 and C_2, have identical copies of the same object O_1, and then both change the state of that object locally, both changes are locally valid and recorded in the local version history I call **dataframes** ("repositories" in Git). Components can push/pull the changes to other dataframes in the network, at which point changes will be merged and conflicts resolved.

Modeling shared-space distributed computing as a distributed version control problem also

has the added advantage of providing a clear state of the replicated objects at each distributed node along with a clear understanding of the changes that occur to the version graph in the form of an update history. This allows developers reason over past state changes, something that is difficult in many distributed programming models.

I take this simple idea, and formalize a programming model called Global Object Tracker (GoT) that captures a simplified version of Git for in-memory applications objects replicated among multiple components.[3] In essence, *GoT is the formalization of an object-oriented programming model based on causal consistency with application-level conflict resolution strategies whose elements and interfaces are taken from decentralized version control systems.* GoT is at the core of a framework I have developed called Spacetime that supports distributed components doing sub-second commit, push, and pull operations. Spacetime is currently implemented in Python, as that is the main language of the Artificial Intelligence community, but GoT is inherently language independent.

While the idea is simple and appealing, there are a few challenges associated with modeling replicated objects as a version control problem. A large category of those challenges is related to the feasibility of the model in practice. The important feasibility problems are: (a) how to deal with merge conflicts without live human intervention (b) how to deal with the fast explosion of revisions that are stored in the version graph and (c) how to maintain the correctness of decentralized version graphs in complex network topologies such as peer to peer networks.

With these challenges in mind, I formulate my **thesis statement** in the following way:

> GoT, a programming model for replicated objects, is capable of supporting causal consistency with isolation including in peer to peer networks, while also supporting observability and low update latency. No existing approach is capable

[3] According to `https://github.com/git/git/blob/master/README.md`, one of the several meanings of Git is Global Information Tracker. I re-appropriated that meaning for objects.

of supporting all of this.

It is not my intention to replicate and formalize the complexity of Git. For example, I do not explore Git branches or capabilities of going "back in time" by checking out older versions while the application is running. The focus of this work is the precise formalization, implementation, and assessment of the basic operations of decentralized version control on a uni-directional branch for purposes of giving a strong organizing principle to replicated distributed objects. To that end, my work makes the following contributions:

- I present a new programming model for replicated objects based on a popular decentralized version control system, Git. While Git is widely-known for versioning files, its use in real-time replicated objects manipulated by programs poses a number of challenges, which are discussed and addressed in GoT.
- I identify and solve the challenges of GoT in order to make it feasible in practice.
- I present a formal specification of this Git-like programming model, GoT. To the best of my knowledge, this is the first time such a model has been formalized.
- I use my formal model to prove that my unique approach to solve the challenges of GoT, works in theory.
- I demonstrate using micro-benchmarks that GoT is feasible in practice and that Spacetime performs remarkably well and can be tuned for consistency at the lowest possible latency.

Finally I provide a replication package for Spacetime that is publicly available at `https://github.com/Mondego/spacetime`.

1.4 Organization of this Dissertation

The thesis is organized into eleven chapters. Chapter 2 provides a strong motivation for using version graphs for distributed systems. Chapter 3 provides a broad background in the alternate approaches taken by both the industry and academia to optimize consistency and latency in distributed systems. Chapter 4 introduces the Global Object Tracker (GoT) programming model, details the developer-facing features that it provides, and a formal model. Chapter 5 talks about the implementation of GoT – Spacetime – and the optimizations made in it. Chapters 6, 7, and 8 elaborate on three challenges to the implementation of GoT. In Chapter 9 I discuss observability of state in GoT and Spacetime with the help on an interactive debugger built on GoT called GoTcha. Finally, I discuss several experiments and micro-benchmarks performed on Spacetime in Chapter 10 and conclude in Chapter 11.

Chapter 2

Background and Motivation

Before we dive into the GoT programming model, it is essential to understand the role it needs to play in the engineering of the systems discussed in the previous chapter. The most critical design considerations are the speed of propagation and the correct application of the updates. We argue that the engineering efforts to deal with these considerations are typically rudimentary forms of version control and that distributed applications that have causally consistent replicated objects have, at their core, a problem of version control.

2.1 Causal Consistency

Causal consistency is a consistency model first formulated in 1995 [3], where updates are ordered in a partial order that reflects the causal relationship between updates. Any update made at a node is causally related to the updates read by the node at the time of the update. A remote node can only receive this update when it has read all the causally linked updates that happened before it. Updates that are not causally linked are considered concurrent updates and can be received by nodes in any order. In systems like multiplayer games and

distributed simulations, this causal relation is essential as it preserves the importance of the actions performed by the players or the agents.

Causal consistency is typically ensured by padding updates with metadata that are vector clocks [70], version vectors [82], etc. help construct dependency graphs. The dependency graphs in causally consistent distributed systems are identical to the version graphs built and maintained by the version graphs in a version control system. The version graphs capture the causal dependency of updates made to the file-system or the shared object space. The root of the version graph is the initial state of the system, and each update builds on the version that it reads to create a new version that is causally related. Concurrent updates create concurrent versions in the graph that act as forks in the version graph, similar to any dependency graph.

2.2 Consistency versus Latency

The famous CAP theorem [45] by Eric Brewer states that when building distributed stores, engineers can only choose to optimize for two out of three properties: Consistency, Availability, and Partition Tolerance. In this context, consistency is the ability to read the latest write. Availability guarantees that all requests to non-failing nodes return a response and Partition tolerance is the ability for the system to continue operation over unpredictable network failures and delays. In any large scale system, however, Partition tolerance is a must. Therefore, the choice is between Consistency and Availability.

It is important to note that the CAP theorem, though often cited, is a rule that defines the choices of the system only under the condition of partition. Without partitions, both Consistency and Availability, as defined, can be provided. Therefore, a more relevant theorem is the PACELC theorem [1], which is an enhancement to the CAP theorem. It states that a

distributed system, when not partitioned, must choose between latency and consistency, with latency defined as the time taken for a component to receive a response to a request from the distributed store. In the case of replicated shared-space systems like multiplayer games, simulations, and geo-distributed databases, there is typically no single data store. Instead, they have replicas at each component in the system, for which both consistency and latency are essential. For example, in an online multiplayer game, the updates made by one player have to be visible (consistency) as soon as possible (latency) by another player.

2.3 Update Latency

Optimizing for lower latency at the cost of consistency implies that components may receive stale data during requests, which is detrimental to the performance of shared space applications that need consistency at the lowest latency possible, a notion we refer to as update latency in the rest of the paper. *Update latency* is the time taken for an update to the state in one component in a distributed system to be observed in another component.

Update latency is different from the notion of both consistency and traditional latency. Consistency only includes the notion of the latest write and does not include the concept of time. Traditional latency only includes the notion of the time for response and provides no information about the content of the response. Update latency aims to encapsulate both concepts, and shared space applications require systems optimized for update latency. Update latency can be broken down into many independent steps, each having a different impact on update latency, and require different solutions.

The first step is to prepare an update, which may include serializing state or encoding operations into formats that can be transported over the network. The smaller the update, the less time is typically spent in this stage. Optimizations such as difference calculators

typically used in geo-replicated databases can increase this cost. The second step is the transportation of an update. While the network latency is unavoidable, the size of the update is again significant. Finally, the update received has to be deserialized or decoded and reconciled with the local state at the receiver. It can be seen that update latency can be reduced if the cost of communication and reconciliation is made low.

As a driving example to explain how to reduce update latency, let us consider a multiplayer game where players, Alice and Bob, experience the same virtual space. Online multiplayer games have strong requirements on the availability of the shared objects, as the responsiveness of the environment is of utmost importance. As such, object replication is a necessity. So is latency minimization.

2.3.1 Reducing Cost of Reconciliation

When Bob takes an action, the local state of the game is updated. When Bob and Alice's game states synchronize, the updates that Bob made are sent to Alice. When Alice receives these changes, they must be applied to the local state. The process of applying a remote update to a local state such that it is then available for the local code to use is what we call reconciliation. Reconciliation of states is not the same as conflict resolution, although conflict resolution is an integral part of it. When multiple updates are concurrently made and sent to a node, the differing states must be resolved and is called conflict resolution. In addition to conflict resolution, reconciliation includes the cost of deserializing updates and the cost of applying an update, conflicting or not, into the local state.

Reconciliation is typically made fast by eliminating, limiting, or ignoring the conflicts that can occur. For example, the last write wins reconciliation strategy, by design, ignores all conflicts. Instead, every write takes precedence over the update that preceded it. Conflict-free replicated data types [100] (CRDTs) and Global Sequence Protocol [46] are other examples

of a reconciliation strategy that are fast as they avoid conflicts altogether. Unfortunately, it is not always possible to eliminate or ignore conflicts. The conflict resolution strategy used varies from domain to domain and is tied to the system's business logic. Some domains, for example, banking, might require expensive conflict resolution strategies. While this is not to say that conflict resolution should be forgotten, a better approach would be to leave the conflict resolution as a choice to the developers who can optimize their approach based on domain knowledge.

Another way to reduce the cost of reconciliation is to avoid heavy processing of the updates that are introduced. Massive operations or large state deltas are typically expensive to reconcile and can slow down the availability of updates at the receiving nodes. Sending or receiving redundant states can, therefore, increase the cost of reconciliation.

2.3.2 Rules for Reducing communication

Reducing the cost of communication is a crucial design goal for reducing update latency in distributed systems. Often, the nodes that need to synchronize the state are located geographically far apart, and latency between these nodes is unavoidable. As explained earlier, the network latency between Frankfurt, Germany, and Los Angeles, USA, is about 145ms. If Alice and Bob, are located in Frankfurt and Los Angeles, respectively, updates made by Bob will only be visible to Alice, at best, 145ms later. This latency makes communicating redundant information costly, as unnecessary data may further delay the reception of relevant information. Reducing communication also allows for faster processing of updates as serialization and deserialization costs are typically lower when updates are smaller.

While techniques such as compression certainly help reduce the data transmitted, to effectively reduce the cost of communication, nodes must make only the required synchronizations and

with minimal redundancy. To understand how to achieve this, we identify three rules that any node in the system must follow.

- R1: A Node must take/receive only *objects it needs.* (Interest Management)
- R2: A Node must take/receive only *changes it does not already have.* (Delta Updates)
- R3: A Node must take/receive *only when it requires updates.* (Isolation)

We explain each of these ideal requirements in more detail.

R1: Interest Management

The nodes in the system should receive updates to only the slice of the state that is relevant. This requirement requires techniques that partition updates by interest space. This requirement is known in games engineering as *interest management.* If Alice and Bob are in different zones of the game world, or they are visually blocked by a virtual wall, the updates that Bob makes do not need to be sent to Alice, as they do not affect her view of the world. Multiplayer games, especially massively multiplayer games, use many techniques for interest management. Outside multiplayer games, techniques such as subscription channels (publish/subscribe), views (relational databases), and event-driven programming, are often used to slice the replicated state for each node in the system.

R2: Delta Updates

This requirement deals with avoiding sending state that did not change and requires techniques that partition updates by time. For example, when Bob moves in the game world, it is unnecessary to send to Alice the entire state related to Bob; only his new position needs to be sent. This update can be communicated either as delta-state, like in δ-CRDTS [4], or as a

domain-specific command that encodes some semantic meaning (e.g., MOVETO(200, 300)). When Alice's node receives the update, it can apply the update to its local replica allowing Alice to see the action that Bob took.

R3: Isolation

This requirement also calls for nodes to have autonomy in deciding when to incorporate changes made by other nodes. For example, let us assume that Alice's game refreshes at 30 frames per second. (the value is typically tied to the hardware capabilities of Alice's machine.) This means that the game renders a new frame on the screen every 34ms. Let us now assume Bob is moving in a straight line and generates four MOVETO commands in 34ms. When Bob sends the updates to Alice, only the last update is incorporated, and the rest are discarded. This means that three of the updates were redundant information. Even if the command was one that requires the previous state, for example, MOVEBY(100, 0), the four MOVEBY commands could have been combined into one MOVEBY(400, 0) update. The need for autonomy can also come from the unavoidable event of network partitions. If the connections between the players break, it should be possible, in some cases, for players to continue to play in disconnected mode and synchronize later. When they do synchronize after the partition, the node should receive all the individual updates that it missed, but instead receive a single, much larger update.

2.3.3 Meeting the Requirements

The difficulty of meeting these three requirements differs with the style of communication used. In general, there are two styles of communication: push (Alice receives updates made by Bob when Bob sends them to Alice), and a pull (Alice receives updates made by Bob when Alice gets them from Bob); different requirements are easier to implement in each style.

In both push and pull, the requirement (R1), interest management, is relatively straightforward. The only challenge is in defining the interest, especially when it changes dynamically. The node generating the update needs to know if the remote nodes receiving the update require the information. Predefined or programmable interests (subscribe channels, for example) are a way to achieve this.

Requirement (R2), delta updates, is much easier to implement in push style than in pull style. In push style, the creator of the update initiates the communication with the other nodes. Because the node that pushes is the node where changes occurred, it is relatively easy to know what changed since the last update was sent. Even in the case of partition, a simple acknowledgment based delivery protocol can ensure delta based communication. In pull style, however, delta updates are a lot harder. Since the receiver of updates initiate requests, the sender must keep track of all updates sent to the receiver, for every receiver, since the last request.

Requirement (R3), autonomy and recovery from network partitions, is hard to achieve in push style, but trivial in pull style. In many cases, nodes pushing updates do not know if and when the receiving nodes want updates. For example, if Alice enters a "freeze" mode, Bob will not know that it should stop pushing updates until Alice's node informs Bob's node. In pull style, however, Alice's node will simply make the request to Bob's node when needed, allowing requirement (3) to be trivially satisfied.

Moreover, in the face of network partitions, it becomes the responsibility of the pusher node to store the potentially increasing set of updates while trying to re-establish the connection. Additionally, if a consumer restarts and requires the entire state to be synchronized, the producer has no way of knowing that the consumer requires the entire state. The updates that were in the queue have just been invalidated. In pull communication, the consumers are responsible for requesting updates. If there is a network partition, the producers can continue while the consumers have to stop. No effort is lost by the producers in trying to re-establish

connections. Consumers can express the type of synchronization required and can request the complete state after restarts. This partition tolerance is especially useful for peer to peer applications that can continue operations while some nodes are down or unreachable.

Since sending state changes in the form of delta updates is an important and necessary optimization for replicated state applications, most replicated state applications use push-style communications for the state that requires immediate attention, and leave pull style only for large, more permanent data such as immutable assets. However, as argued above, push-style makes it hard to achieve node autonomy.

2.4 Rudimentary Forms of Version Control

In this section, we argue that the engineering efforts to deal with the considerations mentioned are typically rudimentary forms of version control and that distributed applications that have causally consistent replicated objects have, at their core, a problem of version control. To understand this, we use a simple virtual environment example: Attari's-inspired multi-bot Spacerace. We re-use this same example again when demonstrating the features of the Global Object Tracker programming model.

The Spacerace game is about a player-controlled spaceship navigating the dangers of an asteroid field. The player typically starts at the bottom of the screen and works their way up past asteroids that are moving at varying speeds along the path. The player is given a win if they manage to reach the top of the screen without crashing. In a multiplayer version, multiple players share the same level and control different spaceships. They may compete to reach the top of the screen first.

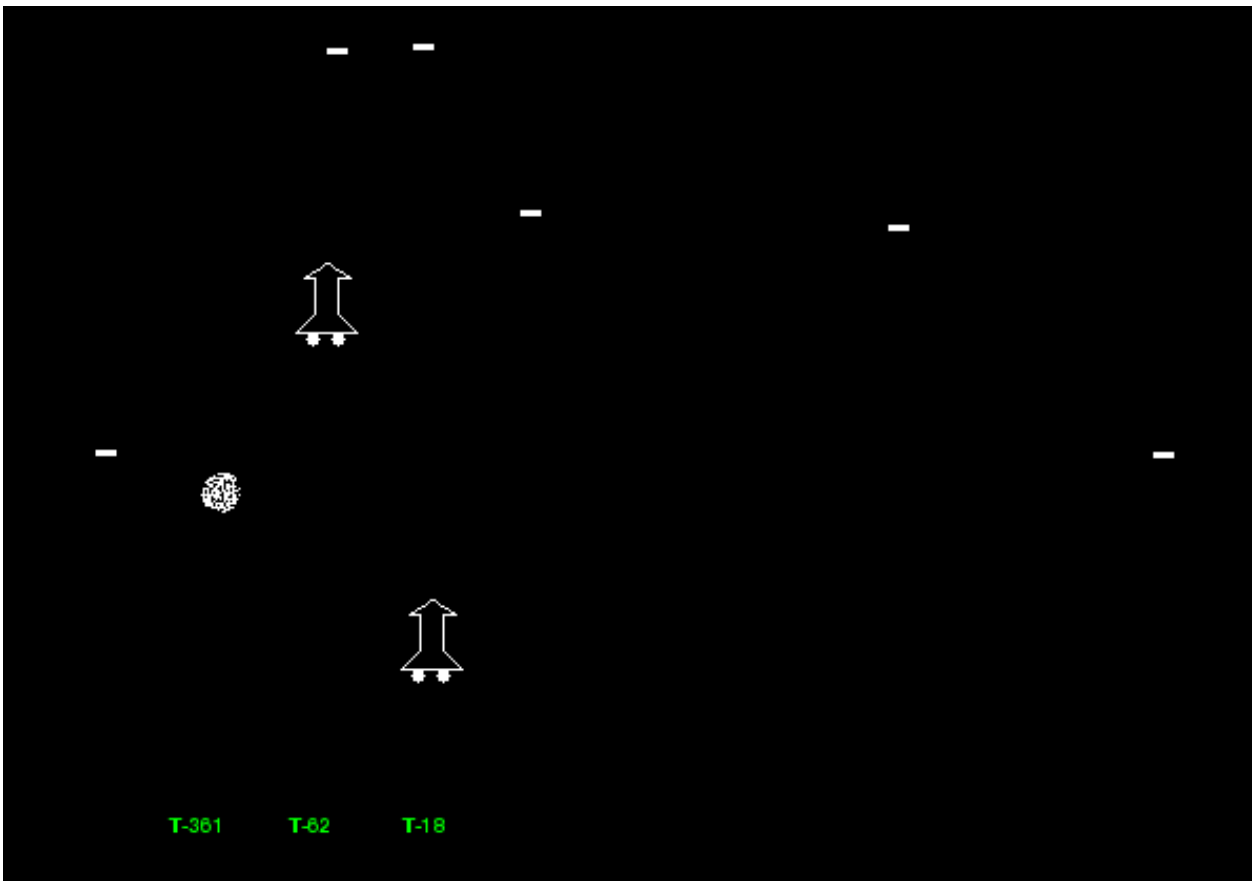


Figure 2.1: Attari's Space Race game.

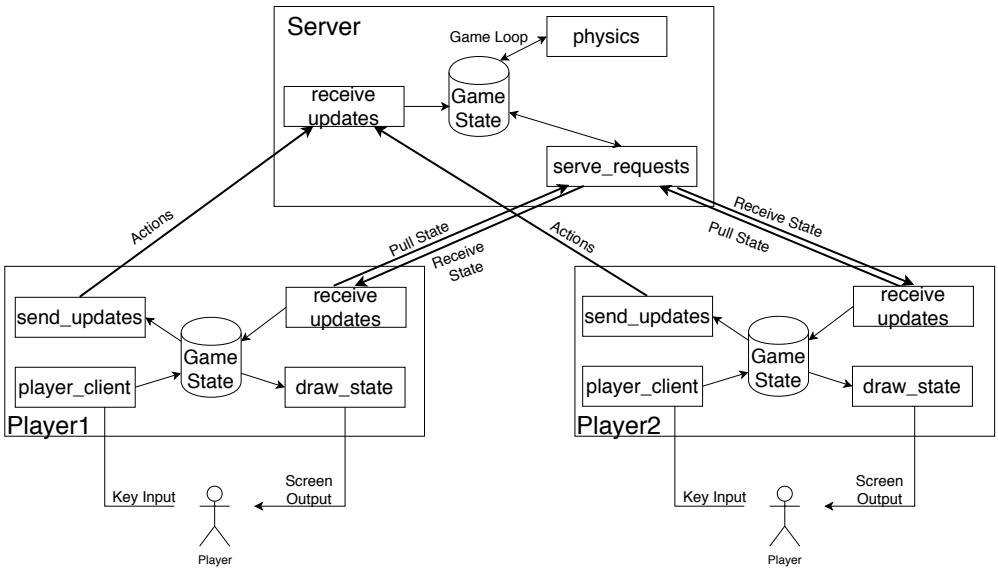


Figure 2.2: A simple architecture for Spacerace.

2.4.1 Spacerace: Basic Design

The most straightforward design that can be used to implement Spacerace is a traditional server-client topology, with the server being a data store acting as the shared-memory for the environment. The components coordinating over this shared-memory are the player clients to take input and play the game, visualizers to have visual feedback of the game state, and a physics server component, responsible for enforcing the rules of the game, that can house the in-memory data store. The interaction between the components is shown in Figure 2.2

Let us first take a look at the client-side. In Figure 2.3 we see four small functions, a player_client function (line 1) which acts as the main function, send_updates (line 11) that sends updates made by the player to the server every second, receive_updates (line 20) that synchronizes the state of the local game with the server, and draw_state (line 21) that shows the state of the game to the player. The local state of the game is maintained in the game_state parameter (line 1) that is then passed around to all other functions.

The player_client first connects to the physics servers (line 2) and then launches the other three functions as concurrent threads (lines 3 – 6). It then runs on a loop, taking input from

```
1  def player_client(server, game_state, player_id):
2      server.connect()
3      Thread(target=send_updates,
4             args=(server,game_state,player_id)).start()
5      Thread(target=receive_updates,args=(server, game_state)).start()
6      Thread(draw_state,args=(game_state,frame_time)).start()
7      while True:
8          key = take_input()
9          update = game_state.record_key(key)
10
11 def send_updates(server, game_state, player_id):
12     sync_time = 1  # Send updates once every second.
13     while True:
14         start = time.perf_counter()
15         delta_update = game_state.get_and_clear_delta()
16         if delta_update:
17             server.send(player_id, delta_update)
18         sleep_remaining(start, sync_time))
19
20 def receive_updates(server, game_state):
21     sync_time = 1 # Assuming sync with server every second.
22     while True:
23         start = time.perf_counter()
24         resp = server.send_pull_request()
25         resp.wait_for_complete()
26         game_state.replace(resp.asteroids, resp.ships)
27         sleep_remaining(start, sync_time)
28
29 def draw_state(window, frame_time):
30     window = GameWindow(frame_time)
31     frame_time = 1.0/20
32     while True:
33         start = time.perf_counter()
34         window.predict(frame_time)
35         window.draw()
36         sleep_remaining(start, frame_time)
```

Figure 2.3: Simple design - Client side.

the user and applying it to the game_state (lines 8, 9). The send_updates thread runs on a loop and sends all unsynchronized delta changes that have been applied to the game_state by the inputs from the player, to the physics server (lines 14 – 18). The receive_updates thread sends a pull request to the server asynchronously (line 24) and replaces the asteroids and ships in the game_state with the list of asteroids and ships that it receives in response (line 26). The draw_state thread visualizes the game state every 50ms trying to achieve a frame rate of 20 frames per second. When there are no new updates in a draw, it extrapolates the state to keep the animations looking smooth at the client-side (line 34).

On the server-side, as seen in Figure 2.4, the server (line 18) is the driving code. The state of the entire game is stored in an object of type ServerGameState (definition in lines 1 – 16). It consists of two sets of objects - ships and asteroids (lines 3, 4). In the server, the game_state object holds an instance of ServerGameState (line 19). Three parallel threads are launched (line 20 – 22) to handle three aspects of the server, after which the main thread only listens for new clients that connect (line 25, 26). When a new client connects, it is immediately added to the ServerGameState object using the add_client method (line 9).

The first thread is the physics of the game (line 27), which executes on a loop (typically called the game loop). The physics function first waits for at least two ships to join (line 28–29). When at least two ships are in the state, it iterates over the game, invoking the tick method in game_state until the game_state has signaled the end of the game (line 32–35). In each tick, the game_state moves the asteroids and ships and checks if any player has won the game or if every player has crashed. If either of those conditions is true, the game is over, and the game_state signals this to the physics thread (lines 13 – 16). The serve_requests (line 38) thread receives requests for the game state (line 40) and responds by sending the current state of the asteroids and ships (line 41). The third thread, receive_updates (line 43), is responsible for receiving the updates from the clients and applying them to the game_state (line 45).

```
class ServerGameState():
    def __init__(self):
        self.asteroids = [Asteroid() for i in range(NO_ASTEROIDS)]
        self.ships = dict()

    def add_client(self, client, ship):
        self.ships[client.cid] = ship

    def apply_client_update(self, client, update):
        self.apply_delta(update)

    def tick(self):
        self.move_asteroids()
        has_winner = self.move_ships()
        game_over = check_collisions(asteroids, ships)
        return has_winner or game_over

def server():
    game_state = ServerGameState()
    Thread(target=physics,args=(game_state)).start()
    Thread(target=receive_updates,args=(game_state)).start()
    Thread(target=serve_requests,args=(game_state)).start()
    while True:
        client = receive_connection()
        game_state.add_client(client, Ship(client))

def physics(game_state):
    while len(game_state.ships) < 2:
        time.sleep(1)
    has_winner = False
    done = False
    while not done:
        start = time.perf_counter()
        done = game_state.tick()
        sleep_remaining(start, frame_time)
    game_state.print_winner()

def serve_requests(game_state):
    while True:
        client = receive_state_request()
        client.send(serialize(game_state.ships, game_state.asteroids))

def receive_updates(game_state)
    while True:
        game_state.apply_delta(receive_delta())
```

Figure 2.4: Simple design - Server side.

This simple approach is straightforward in its design. However, all players are synchronizing the entire state at each frame. If there are a large number of asteroids or a large number of ships, the amount of data being transferred will be prohibitively large.

To analyze this design further, let us look at the three requirements of reducing communication. The first requirement, interest management (R1), is not really relevant as both ships and asteroids are required by both the physics and player nodes.

Looking at the second requirement, delta updates (R2), we see that writes from the player to the physics server are in the form of delta updates. Each player does not send the entire state of objects over the network to the server. Instead, it sends only updates to the game_state made by the keystroke pressed by the user. However, the updates made at the server are communicated to the player node in the form of the entire state of the objects. The server does not use delta updates. For example, let us say that a ship is absolutely stationary, and the state of a ship has not changed for a few seconds. A player already knows the position of that ship and therefore does not need to receive any update on the position of the ship from the physics server. Since the player does not know if the ship has moved or not, the player has to make the pull request for the state. Since the physics server does not keep a record of what the game state at the player is, it has to send the entire state to the player. It cannot send only delta updates (say the new positions of the asteroids) back to the player. This is a familiar story in stateless server systems. Many database stores such as MySQL, Redis, etc., allow clients to write to the database in the form of delta changes (Update commands typically only require the primary key and any fields that updates), but the clients cannot receive only those set of changes that have happened in the database since their last read or write.

The third requirement, isolation (R3), is met from the client side as each client synchronizes with the server at their own pace. The server is rather passive, waiting for requests from the clients. This means that if a client disconnects, the server does not spend resources trying to

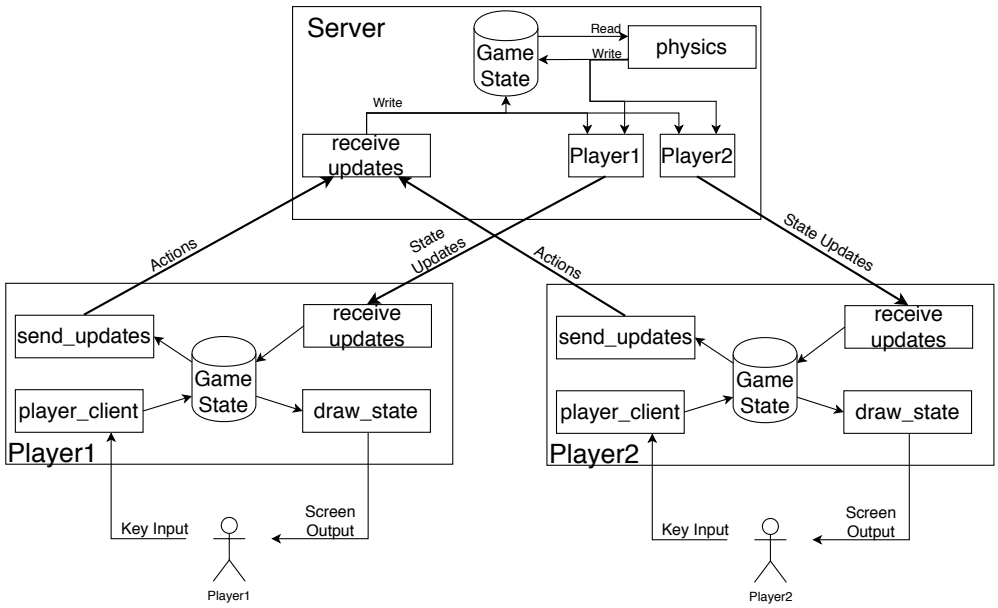


Figure 2.5: Delta updates over push for Spacerace.

send the client's data. However, as explained above, this also means that the server cannot send clients delta updates.

This example is simple and easy to understand but is inefficient. As explained above, the entire state of the system is transferred at each request as the servers do not maintain any notion of what state the clients are in, and the clients do not self identify this information. This can be solved by making the server stateful and moving to a push only model of communication.

2.4.2 Spacerace: Delta updates from the server

To cater to strict delta updates, we modify the client not only to send updates but also to receive updates that are pushed by the server. Figure 2.5 shows the interaction between the components and Figure 2.6 shows the changes made to the code shown in Figure 2.3. Unlike in the previous version, the thread receive_updates does not make pull requests to the physics server. Instead, it actively listens for any incoming messages from the server

```
1  def player_client(server, game_state, player_id):
2      ...
3
4  def send_updates(server, game_state, player_id):
5      ...
6
7  def receive_updates(server, game_state):
8      while True:
9          delta = server.receive_delta()
10         game_state.apply_delta(delta)
11
12 def draw_state(game_state):
13     ...
```

Figure 2.6: Delta updates, Push only design - Client side.

(line 9). The message it receives is parsed as a delta and applied to the game_state (line 10). The rest of the operations at the client are the same as before. At no point does the client perform a pull for updates. The client wholly relies on the server sending messages to synchronize the state.

The server-side, especially ServerGameState, become more complex in this method, as seen in Figure 2.7. When a new client is added to the game_state, in addition to adding a new ship, all other clients are immediately notified of the new ship (line 7,8). The clients receiving this update via their receive_updates thread, discussed above, and will update the local game_state replica accordingly. The draw_state thread would then visualize this ship in the game window, informing the player that competitors have joined the game.

As the game progresses through game ticks, as seen in the physics thread, the state of the game changes. All changes are recorded and distributed to all the clients (line 23, and 10–13). Every client's receive_update thread receives these updates and merges them into the game_state for draw_state function to render in the next frame.

This design goes all in for delta updates. The delta updates are tracked in both the server and client as they happen. The client updates are sent to the server, which immediately

```
class ServerGameState():
    def __init__(self, asteroids, ships):
        self.asteroids = [Asteroid() for i in range(NO_ASTEROIDS)]
        self.ships, self.clients = dict(), dict()

    def add_client(self, client, ship):
        self.clients[client.cid], self.ships[client.cid] = client, ship
        self.distribute(ship, except=client)

    def distribute(self, data, except=None):
        for cid, other_client in self.clients.items():
            if except and cid != except.cid:
                other_client.send_delta_async(serialize(data))

    def apply_client_update(self, client, update):
        self.apply_delta(update)
        self.distribute(update, except=client)

    def tick(self):
        delta1 = self.move_asteroids()
        delta2, has_winner = self.move_ships()
        delta3, game_over = check_collisions(asteroids, ships)
        self.distribute(delta1 + delta2 + delta3)
        return has_winner or game_over

def server():
    ...

def physics(game_state):
    ...

def receive_updates(game_state)
    while True:
        client, delta = receive_delta()
        game_state.apply_delta(delta)
        game_state.distribute_delta(delta, except=client)
```

Figure 2.7: Delta updates, Push only design - Server side.

distributes them to all other clients. The updates made by the physics thread are also distributed to all clients.

There are two significant problems with this approach. First, since the server has to maintain a state for each client, if a client disconnects or has network delays, the server keeps trying to send the updates. This wastes computing resources, even if the actual data transfer is asynchronous. Additionally, if due to the network partition between a player and the server, some updates to the player are lost, when the player reconnects, the entire state must be transferred once again. This state can be quite large, locking up resources and can slow down the propagation of ongoing updates. The problem is exacerbated when there are a large number of clients to whom the updates have to be distributed. The appeal of stateless systems (like the previous version and RESTful [38] architectures in general) is that the clients are in control of the synchronization. If the client drops out of the network, the server does not need to do anything different. This highlights the benefits of supporting isolation.

The second problem is that out of order updates can be received by each client. Let us take, for example, a Spacerace game with two players Alice and Bob. Seeing an asteroid in her path, Alice changes her trajectory slightly. This update is sent over to the physics server. When the physics server receives it, the update is distributed to Bob and applied to the game_state. In the next game loop, and before Bob has received Alice's new movement, the physics server calculates a new update, including a new position for the asteroid. It then distributes this new update. There are two concurrent and competing updates for Bob to receive. If Bob receives the new asteroid position first and then Alice's position, the game_state drawn in Bob's game will show Alice crashing into the asteroid, setting her velocity to zero and not rendering any further movement. When Alice's change in trajectory is received, it could be rejected as Alice has been destroyed and cannot make any further action. If the updates are, instead, received the other way around, Bob sees Alice escape the asteroid, and the state is consistent with the state in the physics server and in Alice's machine. Such inconsistencies

can only be fixed if updates that are distributed are associative and commutative – a hard constraint to meet in games.

In the previous basic version of Spacerace, the players pulled the entire state of the physics server and replaced their state with the incoming state. This negated any requirement over the order of the updates and provided a consistent state. However, in the push-based version discussed, there is no guarantee on the order in which updates are received, and therefore, inconsistent states can be reached. Ensuring that the updates are received in the right order requires that the order of updates to the server game_state and the order in which the updates are distributed are the same. This can be accomplished in a number of ways: transactions, locks, a single active thread reading updates from a queue, etc. However, these solutions add significantly to the engineering effort of the system. For example, the methods add_client (line 6, Figure 2.7), apply_client_update (line 15, Figure 2.7) and tick (line 19, Figure 2.7) in ServerGameState can be protected by a lock to ensure that updating the server state and distributing the update is an atomic step. This can, however, as one can imagine, significantly slow down the execution.

2.4.3 Delta updates with Isolation

To tackle the first problem, lack of isolation, minor modifications can be made to both the server and client code to bring back the pull-style communication that was implemented in the first basic version. The modified interaction is shown in Figure 2.8 and the code modifications are shown in Figure 2.9 and 2.10. The receive_updates function in the client has returned. The client makes a pull request to the physics server, but instead of receiving the entire state on a response, it receives a delta response (line 13, Figure 2.9). On the server-side, there is no network activity in the distribution of the updates (line 6, Figure 2.10). Instead, the updates are placed in a client-specific queue. There has to be a separate queue

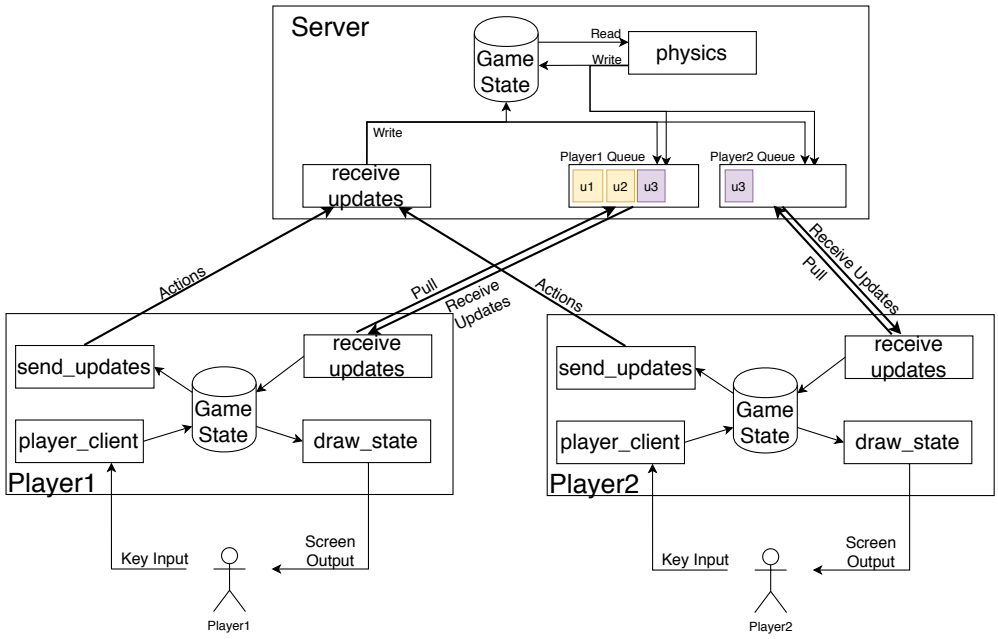


Figure 2.8: Delta updates with isolation for Spacerace.

```
def player_client(server, game_state, player_id):
    ...

def send_updates(server, game_state, player_id):
    ...

def receive_updates(server, game_state):
    sync_time = 1  # Pull at most once per second
    while True:
        start = time.perf_counter()
        resp = server.send_pull_request()
        resp.wait_for_complete()
        game_state.apply_delta(resp.delta)
        sleep_remaining(start, sync_time)

def draw_state(game_state):
    ...
```

Figure 2.9: Delta updates, with Pull - Client side.

```
class ServerGameState():
    ...
    def distribute(self, data, except=None):
        for cid, other_client in self.clients.items():
            if except and cid != except.cid:
                other_client.add_to_queue(serialize(data))

    def get_deltas(self, client):
        if client.cid in self.clients:
            return self.clients[client.cid].flush_queue()
    ...

def server():
    ...

def physics(game_state):
    ...

def serve_requests(game_state)
    while True:
        client = receive_state_request()
        client.send(merge_deltas(game_state.get_deltas(client)))

def receive_updates(game_state)
    ...
```

Figure 2.10: Delta updates, with Pull - Client side.

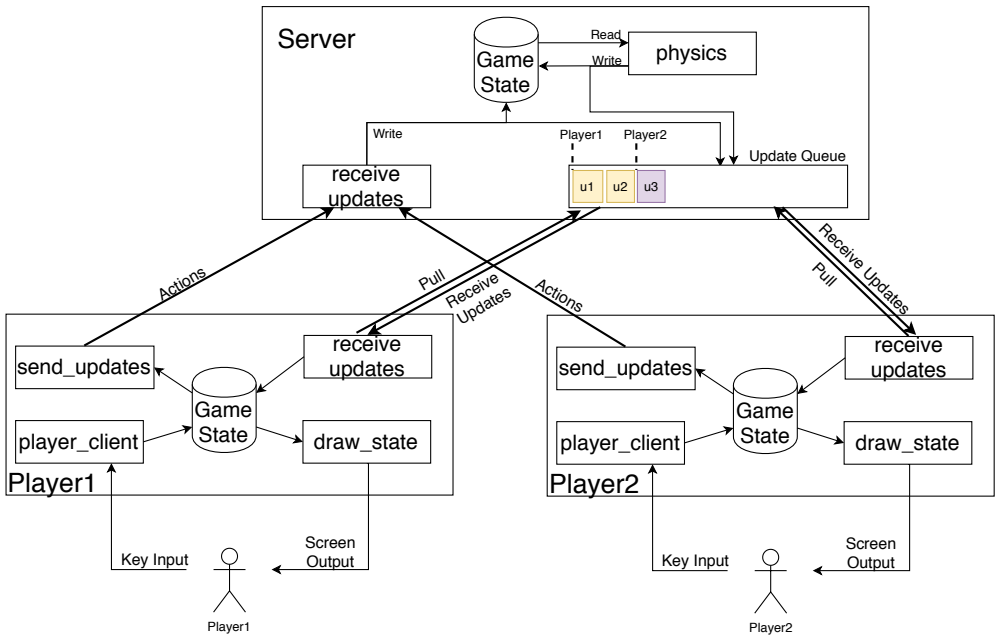


Figure 2.11: Delta updates with isolation and low memory for Spacerace.

for each client as different clients synchronize at different rates and might have a different set of updates to pick up. When a pull request reaches the server (line 21, Figure 2.10), the deltas present in the client-specific queue are picked up and flushed (line 10). They are then merged together into one update and sent back to the client (line 22).

As we can see, the clients have isolation, and the server is not performing unnecessary network tasks. However, in the event of a network failure or delay for a client, the size of the client queue at the server will increase indefinitely. While setups such as timeouts and complete synchronization can be put in place, these are simply patches over a problematic design. Furthermore, the second problem discussed in the previously – ordering of updates – still exists in this version of the system.

2.4.4 Delta updates with Isolation and Low Memory

To solve the problem of having multiple copies of the same update, one at each client's update queue, at the server, we can merge each individual update queue into a single update queue.

```
1  def player_client(server, game_state, player_id):
2      ...
3
4  def send_updates(server, game_state, player_id):
5      ...
6
7  def receive_updates(server, game_state):
8      sync_time = 1  # Pull at most once per second
9      last_sync = None
10     while True:
11         start = time.perf_counter()
12         resp = server.send_pull_request(last_sync)
13         resp.wait_for_complete()
14         game_state.apply_delta(resp.delta)
15         last_sync = resp.last_sync
16         sleep_remaining(start, sync_time)
17
18 def draw_state(game_state):
19     ...
```

Figure 2.12: Delta updates, Pull, low memory - Client side.

The component interactions of this version of the system can be seen in Figure 2.11 and the code is seen in Figures 2.12 and 2.13. When the server makes an update or receives an update from one of the players, it places the update in am update queue (line 9) that is marked with a certain identifier, which could be as simple as an array index. The clients that synchronize with the server receive an update position that marks the last know update it received (line 15, Figure 2.12). In any pull request, the client first supplies this position (line 12). The server can pick up the appropriate set of updates from the single update queue (line 12 – 17 Figure 2.13) and send it to the client (line 30, Figure 2.13).

This is very similar to publish-subscribe. This approach keeps the servers relatively stateless and independent of the clients. The clients have isolation to synchronize with the servers at their own rate, and every communication is in the form of delta updates. The problem of out of order updates is also solved as a definitive queue of updates is always available. It is important to note, however, that in this process, updates made by a client can be put into

```
class ServerGameState():
    def __init__(self, asteroids, ships):
        self.asteroids = [Asteroid() for i in range(NO_ASTEROIDS)]
        self.ships, self.clients = dict(), dict()

        self.update_queue = OrderedDict()

    def distribute(self, data):
        self.update_queue[uuid()] = data

    def get_deltas(self, pos):
        new_pos = pos
        final_data = list()
        for update_id, data in self.update_queue[pos:]:
            final_data.append(data)
            new_pos = update_id
        return final_data, new_pos
    ...

def server():
    ...

def physics(game_state):
    ...

def serve_requests(game_state)
    while True:
        client, pos = receive_state_request()
        data, new_pos = game_state.get_deltas(client, pos)
        client.send(merge_deltas(data), pos)

def receive_updates(game_state)
    ...
```

Figure 2.13: Delta updates, Pull, low memory - Server side.

the update queue and received once again when the client synchronizes. These updates can be discarded when they are received again, but that would mean that the order of updates applied at each node is, once again, different. The alternative is to use the order that is synchronized from the server, and apply updates only at the end of the pull, thereby not letting clients read their own writes. This technique is the basis of the Global Sequence Protocol [46], which, in addition, allows local updates to be tentatively applied until they are confirmed.

It is important to note that the size of the update queue can keep growing indefinitely as well. In order to control that growth, updates that have already been received by every player must be deleted. While this does control the growth of the updates under normal network conditions, when there is a partition, and certain players have not received their updates, the number of updates can once again be indefinite. There are garbage collection techniques that can be employed to deal with this unbound growth and are discussed in detail later.

The approach described here, which is also at the heart of many low-latency games, is a form of rudimentary version control. The identifiers for updates are equivalent to version tags. A node pulling changes identifies the version of its replica and receives a diff (delta update) of changes that, when applied, bring the replica to the latest version. In fact, the additional benefit of using an actual version graph is that instead of an update queue, we have an update graph that allows for branches. These branches are essentially alternate orderings of the same update, preventing us from getting into a situation where clients cannot read their own writes.

Recognizing these engineering patterns as primitive forms of version control is a good stepping stone to more straightforward solutions. Modeling replicated state synchronization as a version control problem from the start allows us to avoid all this ad-hoc complexity. In fact, if we go back to the ideal requirements stated at the beginning of this section, we can see that (1) interest management maps to tracking/untracking objects in version control systems;

(2) delta state maps to diffs in version control systems; and (3) autonomy and recovery from isolated operation maps to explicit inter-node synchronization actions such as push/fetch, that merge independent evolutions from common ancestor versions in version control systems. Further custom and optimized domain-specific conflict resolution can also be achieved using the three-way merges that are popular in file-based version control systems.

2.5 Summary

To summarize, we defined causal consistency, update latency, and isolation as essential goals for highly mutable shared space applications. We showed that the engineering efforts to minimize update latency and allow for isolation are typically rudimentary forms of version control. Modeling such systems using version control can simplify the programming model allowing application developers to reason over the state at each node and program inter-node communication precisely.

In this dissertation, we take the design principles in distributed version control and adapt it into a programming model called the Global Object Tracker (GoT) that is suitable for highly mutable shared space applications. It provides causal consistency with isolation and low update latency.

Chapter 3

Related Work

In this chapter, we will look at the programming models create and used in both the industry and academia from the viewpoint of the support they can offer highly mutable shared-space applications. In particular, we will look at their support for consistency, update latency (including all the requirements established in the previous chapter), and isolation.

Most programming models can be classified as either a shared-state programming model or a message-passing programming model.

3.1 Shared-State Programming Models

A shared-state programming model typically has all of the synchronized application state hosted by a predefined set of components. This synchronized application state is the shared state of the system. Distributed computing nodes read, execute, and update the shared space using many coordinating mechanisms, including, and not limited to, transactions, locks, and versioning. We must first clarify the difference between shared-state models and shared-space applications before we see how shared-state programming models can help build

highly mutable shared-space applications.

Shared-space applications are a type of application. The business logic of the application demands a notion of a shared environment or data. For example, an online multiplayer game has a virtual world that is shared and consistent between all players. The same is true for multi-agent simulations. In geo-replicated databases, the content of the database itself is the shared space. All geo-replicated database instances behave as if they were a single entity to the clients that use the data within these databases.

The shared-state programming model is a type of programming model where the state of the system is stored in a common location independent of whether the application has a shared space or environment in its logic. An online multiplayer game or multi-agent simulation built on a shared-state model would have a database or a node controlling the virtual world. Components that work on manipulating or computing over the virtual world would need to read and write from the shared state.

We look at two specific types of shared-state programming models: databases and software transactional memory and see how they help create shared-space applications with highly mutable data.

3.1.1 Databases

Databases and Data Stores are the most fundamental tools that exist in the utility belt of distributed system designers. In fact, many times, the term distributed systems is used only to mean distributed access to databases. Databases are an organized collection of information. There are many types of databases and multiple concurrency models supported, but for this dissertation, we will be looking at two aspects of databases. First, we will look at databases as a means to provide read and write access to the highly mutable shared-space required

by applications that we have identified as our concern (multiplayer games, multi-agent simulations, and geo-replicated databases). Second, we will also look at a specific class of databases – multi-version databases – that provides version control like guarantees to the access to data.

Highly mutable shared-space over Databases

Databases support a wide variety of consistency models. On one end of the consistency spectrum are ACID databases like MySQL [33], PostgreSQL [76], etc, that support strong consistency synchronizations. These databases offer sequential consistency and guarantee that every read includes the latest possible write with techniques like transactions. On the other, weakly consistent and highly available end is eventually consistent databases such as Redis [15], Riak [9], etc. These databases are typically deployed as geo-distributed clusters of nodes. The clients that connect to these nodes can read and write from any one of the database nodes in the cluster. This database distribution improves latency at scale as there more nodes are available to respond to requests. However, the writes made at one database node must be replicated to every other node, which does not happen instantaneously. This lack of instantaneous write means that while the latency of response from the database cluster is low, update latency might be high as the response received might not have any new update. Since highly mutable shared-space applications require low update latency, eventually consistent databases are not typically used to implement these applications.

Databases, especially relational databases, are excellent at interest management. With the help of techniques such as views, managed views, and dynamic querying, the clients reading from the database can pick and choose at low granularity, the data that they require. For example, let us consider a rural simulation using a relational database as the host of the shared space. A traffic simulator and a livestock simulator could be two separate simulations executing over the same shared space. However, the traffic simulator is interested in the

Cars table, and the livestock simulator is only interested in the Livestock table. This per table subscription is already quite good interest management. However, relational databases can go even further. There can be many cars that are simply parked and not moving. The traffic simulator does not need to know about updates to those cars until they move. For example, if the ownership of a parked car changes, the traffic simulation does not need to know it. By using view or dynamic filters in select commands, the traffic simulation can obtain precisely the information that it needs, i.e., all cars that have a velocity greater than zero. Join operations can make for even more granular interest management. For example, the traffic simulation does not need to know about chickens in the simulation up until a chicken decides to cross the road. While the traffic simulation can never tell why the chicken decided to cross the road, it can take a dependency on a join between the Livestock table and the Cars table to only obtain information on those chickens who are in danger. This level of granularity, if used correctly, can often lead to minimal transfers of information between nodes vastly improving update latency.

Databases do not typically support delta updates. Databases are typically built for persistent data, and these systems are typically engineered to be stateless. As such, they cannot know what information a client has, which means that they typically cannot offer delta updates. While timestamps can be added to the records and queries can be constructed to include only the records that fall after a particular timestamp, correct delta updates would also include the ability to obtain only the set of column values within a record that has changed. Without knowing the exact state of the clients, databases cannot offer delta updates. There has been some work on database support for temporal tables [58], which expose additional fields that can be used to query for delta updates.

Databases primarily support pull-based communication. They are agnostic to the state of the clients that connect to them and do not actively push updates to these clients. The clients can choose the rate at which they communicate with the databases (up to a service

upper bound set by the database). However, since they are a shared-state system, nodes in the system do not keep the state locally. As such, if there is a network partition from the databases, the nodes will no longer have access to the shared state, potentially halting these operations. The significant engineering effort of maintaining offline replicas is required to make this form of isolation possible.

Another important consideration is the notion of time. Eventual consistency provides no causal relation between updates, and therefore, additional safeguards have to be placed over the state's correctness. Causal consistency is the weakest consistency model that still supports a notion of time. Many databases such as MongoDB [7], Cassandra [23], and Riak [9] offer optional causal consistent modes of operations. These modes offer causal consistency using techniques such as vector clocks [70] or hash histories [55], which allows for the detection of concurrent updates. When concurrent updates are detected, these systems typically retain both updates and leave the resolution to the application code via two-way merges. There are databases such as AntidoteDB [99] that explicitly support causal consistency.

3.1.2 Multiversion Databases

Multiversion databases are those that use multi-version concurrency control (MVCC) as a transaction mechanism. MVCC was first proposed in 1970 [87] is a concurrency scheme and a transaction mechanism for databases that aims to maximize the parallelism without sacrificing serializability. In MVCC, the database maintains multiple copies of each tuple in the database to allow parallel operations on the tuples. Read-only transactions are allowed to access older versions concurrently with read-write transactions that create newer versions. The work done by Wu et al. [113] provides a good understanding of the types of MVCC mechanisms seen in popular databases. In all of these versions, while the basic principle is version control, the underlying mechanism does not take advantage of versioning to allow

clients to read delta updates. This choice, however, is understandable. Databases are typically designed to hold large quantities of data. The clients, however, require a small subset of this data. Interest management and fast concurrent writes are more useful than delta encoding. However, for shared-space applications with highly mutable data and updates that have to propagate fast, these MVCC based systems that do not provide support for delta updates, fall short.

3.1.3 Software Transactional Memory

Software Transactional Memory (STM) is a synchronization protocol for controlling access to shared memory in distributed systems. It is an alternative to pessimistic approaches such as locks and allows the programmer to specify groups of commands to be executed atomically in the shared space. Concurrent processes or nodes that use software transactional memory are optimistic and always assume that the threads are rarely in conflict. If a conflict, however, does occur, the transaction is canceled.

These allow shared-space applications to be built with relative ease as concurrency is relatively straightforward to implement. The synchronization mechanisms used in the distributed versions often rely on commands being sent over the network, which is a form of delta encoding. Only the state used in the application code at a node is fetched from the shared memory, and the application code has control over when the state is read. However, the main drawback is that concurrent access is relatively slow. In applications like multiplayer games and simulations where the state of the shared space changes rapidly over time, hard synchronization methods such as transactions are always bound to fail. Recovery from these failures is challenging in these applications.

While many STM models have been proposed over the years, in 1983, David Reed [88] first proposed a version control mechanism for controlling the parallel reads and writes to shared

memory. Each write creates new versions that are then read and modified by other nodes in the system. The shared memory is read stable until the end of the transaction, after which a new version is created. Nodes can either choose to read the newer version or continue reading the older version. Since then, several other versioning models such as Distributed Multiversioning [69], Decent STM [14], etc.

For shared-space applications, the behavior of STMs is similar to in-memory databases and offers the same advantages and disadvantages. One major drawback is that STMs are generally not useful when the network latencies between components in the system are unpredictable (as is the case for many online multiplayer games).

3.2 Message-Passing Programming Models

Message-passing programming models, as opposed to shared-state programming models, are a broad group of programming models that distribute the state of the system between all the nodes in the system. The nodes of the application are all independent processes that perform their tasks over their local state. There is no single set of objects that are synchronized and shared. Communication and coordination between these nodes happen via messages. These messages are usually domain-specific commands or operations. Message-passing programming models are prevalent today to scale operations in large distributed systems. Maintaining and operating over a large share state is not efficient or beneficial for many distributed applications. Let us take a map-reduce system, for example. The mappers execute tasks in parallel over a small subset of the data. The output from each mapper is then reduced to get a single output. If each mapper had a shared state, time could be wasted in coordinating access to the state. Instead, a more efficient setup is to send a message to each mapper with precisely the information it needs. There are many types of message-passing programming models that have been created. We take a look at some of the ones that are relevant to GoT

as a programming model.

3.2.1 Publish-Subscribe

Publish-Subscribe is arguably the most common form of message-passing programming models used. It is a form of a message queue pattern. Messages are considered to be events that are created by the publishers and received by the subscribers. Message events are categorized into pre-determined channels that signify some form of collective interest. For example, all logging messages can be grouped into the logging channel. The nodes in the system know which type of messages they need and can subscribe to the right channels.

The primary advantage it offers is the high scalability of the system and the loose coupling of the system's components. More nodes of the same type can be thrown to tackle bottlenecks in the system. For example, a distributed crawler can be implemented in Publish-Subscribe, where URLs to be downloaded are published to the channel by nodes that download and scrape URLs. Each node subscribes to the URLs channel, receives one URL, downloads it, and scrapes new URLs from the downloaded resource. The scraped URLs are then published into a validator channel, and the downloaded resource is published to a downloaded channel. A URL checker can subscribe to the validator and removes all downloaded entries, publishing the rest back into the URLs channel. An indexer can subscribe to the downloaded channel to process all newly downloaded documents. Depending on the speed of the entire process, one or more channels in the system can be a bottleneck, and more copies of the subscribers to the channel can be added to improve efficiency. Such systems can typically be very fined tuned.

While this model is excellent at computing through distributed applications that have tasks that can fit in a workflow, the primary drawback is that using such a model for distributed systems that require shared replicated states, such as multiplayer games or distributed simulations, is difficult. The primary difficulty is in ensuring that the replicated state at

each node is correct. Since there is no common location where the state is stored, significant engineering effort is sometimes required to ensure that the messages reach the same state when applied in each node. Publish/Subscribe offers low update latency, and therefore seems like the right choice for these systems. However, they require complicated and hard to understand mechanisms to maintain correctness of the state at each node.

3.2.2 Actor Models

Actor models are another form of message-passing programming models, introduced in 1973 [47], that is very popular in both research and industry. In an actor model, the nodes performing tasks are considered to be actors. The actors are responsible for executing a well-defined set of tasks and receive and send messages only on those tasks. There is no assumed sequence of operations, and the actors all execute independently and concurrently.

This programming model is excellent at interest management as nodes only receive the information that they are interested in and nothing more. Since it is a message-passing protocol, the messages sent by the actors are usually either partial updates or commands that are interpreted by the actors receiving the update, thus support delta updates. Supporting, isolation, however, is complicated as actor models typically only use push-based communication.

3.2.3 Conflict-free Replicated Data Types

On the heels of OT, a new approach emerged that is both a generalization and a departure from OT: Conflict-free Replicated Data Types (CRDT) [100]. The underlying observation in CRDT, which was already visible in [81], and became even more evident in Treedoc [85], is that there seem to exist data structures that lend themselves to the coordination of

state evolution in a network of peers without central concurrency control, but that not all data structures have that capability. For example, a simple sequence, with normal `ins`, del operations, is not capable of that. However, a sequence without `del` and holding more complex nodes representing a renderable string has that capability. Other examples include grow-only sets, monotonic counters, and many others, including combinations of simpler CRDTs.

CRDTs come in two flavors: state-based and operation-based, the latter more closely resembling OT. They have been proven to be equivalent [100], and so we will focus on state-based CRDT. Formally, a state-based CRDT object is a tuple $< S, s^0, q, u, m >$, where S is the state of all of its replicas, s^0 its initial state, q and u are methods for querying and updating local replicas, and m is a method for merging local replicas with state from peer replicas. It has been proven (again, in [100]) that a sufficient condition for state-based CRDT to achieve eventual consistency in a network of peers is for the merge method to be commutative, associative, and idempotent, and for the state to be monotonically non-decreasing across updates. Again, this is quite a high bar, but many data types can meet it. Mainly, CRDTs implement the illusion (via rendering) of mutable state with data types that never really forget anything, therefore requiring, in principle, an ever-increasing amount of memory. Optimizations can be done that reduce the amount of entropy in these objects via distributed garbage collection, which requires global coordination (e.g., [114]).

CRDTs are a mathematically sound mechanism to distribute data in cases where eventual consistency of data is needed, and they have been implemented with great success in industry-grade middleware – e.g., Redis [15] and Riak [9]. These industrial adoptions use CRDTs not for user-bound peer components, but as a mechanism to keep internal database replicas (the peers) eventually consistent.

However, as the scale of the system grows, it takes longer to achieve synchronization because CRDTs require the transmission of the full state of the object. Recently, [5] proposed delta-state CRDT (δ-CRDT), which is capable of transmitting the state incrementally. It is

interesting to note that delta state, a concept that has existed in VCS for over 40 years, was only recently added to CRDTs.

CRDTs can be made causally consistent instead of eventually consistent by enforcing a causal ordering at each node (usually using techniques such as vector clocks or dependency graphs). Recently, Shapiro et al., defined "just-right" consistency [99] that uses the principles of CRDTs along with causal ordering to ensure conflict-free causal consistency. They have implemented their concepts in the Antidote database. In such causally consistent CRDTs, the garbage collection of unneeded updates becomes a problem to solve. Causality stability [8] has been proposed as a technique to clean up updates that have been received by every node. However, there are still opportunities to combine intermediate operations that are unnecessary. In this dissertation, we put forward an approach that can also be used to garbage collect CRDTs built over causal graphs.

3.2.4 Global Sequence Protocol

Global Sequence Protocol [46, 21] (GSP) is a message-passing programming model for distributed systems. The nodes in the system synchronize their state and coordinate their actions using two sequences of updates. A global sequence, where the order of the updates is globally accepted, and that can be applied to the local state of each application; and a pending sequence, to which new updates are added, awaiting global ordering. It relies on a total ordered broadcast of the updates to build the global sequence. The most straightforward example of a system that ensures total ordered broadcast is a server-client model. As an implementation with optimization, the authors described a system where updates are stored as delta changes in each application until they are pushed through a channel to a server where they reside in a buffer set up individually for each client. The server frequently iterates through open buffers, collects updates, applies them to a persistent state, if present, and

pushes these updates through the channels of other clients. When a client application pulls, it merely applies the deltas in its incoming buffer to the local state and clears the buffer.

GSP is an eventual consistency model. Being a message-passing model, it allows for effective interest management as nodes just need to receive the updates they are interested in (and in global order). GSP also allows for easy delta updates. However, isolation is a significant concern as both sequences of updates would grow in size if a client were not accepting updates or was partitioned from the rest of the system.

The resolution of conflicts is mostly ignored in GSP. The update sequences made are globally ordered and are applied in the same order at all locations. The result is that if two contradictory updates are made, one of them is chosen by the global order and becomes the canonical version; the other is ignored. The authors suggest that updates can be designed in a way to mitigate the effects of this. For example, instead of sending the new absolute value of a counter as an update, which can then be incorrectly overridden, the application can send the update as the operation 'add(1)', which is commutative and gives the same result. While this approach works in many simple cases, it can quickly be challenging to apply this to general operations. They also use additional primitives to block execution on an application until specific updates have been added to the global order. These primitives help ensure that the changes sent have been applied before continuing on the execution. However, it does not guarantee that the update accepted was also not overridden by an update that soon followed. Without an explicit merge function like in version control based systems, or exposing the conflicting versions for future resolution, like in databases such as Riak [9], programmers of the system have lesser tools at their disposal to guard against automatic overwrites.

3.3 Operational Transformation

In the late 1980s, just before the dawn of the Web, a new breed of distributed applications started getting considerable attention from researchers and industry, namely "groupware." Groupware are collaborative environments, where users at different locations can interact with a shared virtual space. Included in these are collaborative editors and widgets (e.g., GROVE [34], Jupiter [79]), and more complex virtual worlds (e.g., DIVE [22]), which were the basis for online gaming.

From early on, groupware systems identified a concrete technical challenge that would define this line of work for several decades: how two or more users can edit the same text without getting it scrambled due to conflicting edits. A simple example of such a conflict is as follows. Consider the shared string "abcd", with two users having identical replicas of it. If both users delete the character at position 1 ('a') in their replicas, the synchronization of the shared state, if done carelessly, may produce an unintended result: "cd". This problem was first identified in [34], where a solution was also proposed: Operational Transformation (OT). The basic idea of OT is to perform a special transformation on the operations sent by remote peers before applying them to the data locally. In the concrete case of our simple conflict over "abcd", the transformation could be for the other peer's delete operation to change to a no-op.

In general, OT defines a transformation matrix T of size mxm, where m is the number of distinct operations on the data, and where each cell is a function that transforms operations into other operations, but with some strict constraints. Specifically, given two operations o_i and o_j, let $o'_j = T(o_j, o_i)$ and $o'_i = T(o_i, o_j)$. Then T is such that $o'_j \circ o_i = o'_i \circ o_j$, i.e. the operations must commute. The original OT paper describes a distributed (peer-to-peer) OT scheme that requires the maintenance of request logs in every peer of the network. A request $< j, s, o >$ in L_i (the log for peer i) indicates that, while in state s, peer i executed the

operation o requested by peer j. This historical information, along with the transformation matrix T, supports, in principle, the de-conflicting merge of any operations on the shared data, so that the data is eventually consistent in the entire network.

Unfortunately, the original OT algorithm was later shown to fail in some crucial cases [89]. In fact, the details of OT have proven to be quite elusive, leading that research community into the chase of a general proof of correctness for over a decade that resulted in several papers making claims that were later shown not to hold, including those in [89] (see [86]).

Taking a step back, it seems that OT is trying to do a form of operation-based version control where merge conflicts over edits on a string can be, not just automatically resolved locally, but also assured to lead to eventual consistency in a network of peers. This is quite a high bar. The only solution that seems to have hit that bar is the one proposed in [81], which adds tombstones – i.e., markers for invisible characters that have been deleted but that need to exist and be passed around along with the data for consistency purposes. This has some significant consequences for OT: first, the data grows indefinitely, because no character is ever deleted; then, the string that was directly manipulated by OT ceases to be a simple string and becomes a much more complicated data structure, for which an additional rendering function needs to be defined. [81] describes a solution to the indefinite growth problem that adds even more complexity to the data structure.

3.4 File Based Version Control

Version control systems (VCS) aim to track the evolution of files in the file system, as humans change them. They exist since, at least, the 1970s [90] (see [36] for a good overview of the history of software configuration management tools). The concept of delta state (a.k.a. *diffs*) has been central to these systems since early on [49, 48].

Distributed VCS was an evolutionary step from non-distributed VCS. The first distributed VCS publicly known as such was Code Co-op [74], a decentralized, peer-to-peer system that supported the propagation of changes over many communication channels, including email. Since then, many other systems became very popular, some of which centralized (e.g., CVS, SVN), others decentralized (e.g., Git, Mercurial).

All of these systems have been designed to keep track of changes made by people to files. As such, the rate of change is extremely slow. While the latency of updates is not irrelevant, it is also not critical. For example, it is acceptable to wait 10 seconds when pulling from GitHub. This kind of latency is not acceptable for quasi-real-time state synchronization. Another non-problem in VCS is the size of history data: very few projects have a number of commits in the millions, but even for those that do, that represents a database (on disk) of a few Gigabytes, at most. In contrast, programs can generate millions of state changes in just a few seconds, leading to an unbearable explosion of historical data.

During the early days of version control, dealing with concurrent updates was considered to be painful [73, 94]. These old version control systems employed file locking mechanism to keep concurrent updates out [24]. Merge strategies at that time were often thought to be tedious and not worth the effort. With the advent of tools to help visualize changes being merged, such as diff3 [56], merge strategies became mainstream. Today most version control systems use some form of merge strategy. VCS assume that inconsistent edits are unavoidable, and therefore merge conflicts will occur. Files are mutable without any constraints. For example, when, locally, one person changes the first character of the first line of a file to the letter 'a', and another person, concurrently, replaces that same character by the letter 'z', when the state is synchronized, only one of these changes must prevail. Different VCS differ on the kind of delta encoding that they use, and, therefore, some may see inconsistencies in edits where others do not. However, none of them can resolve all inconsistencies, like two concurrent changes to the same character. VCS refrain from solving these inconsistencies

automatically, and instead, prompt the user about them.

Baudiš [10] provides a good overview of the types of merge strategies present in version control systems. By far, the most common strategy is the 3-way merge. This merge strategy takes two diverging revisions of files and constructs a diff of each version against a single least common ancestor (LCA) so that an external actor (a person or a program) can decide what to do. 3-way merge is known to have an error condition called the "criss-cross" merge, where a single LCA cannot be calculated, potentially leading to inconsistent version graphs. Version control systems such as Git use a recursive merge strategy to try and eliminate the problem. However, some variations cannot be resolved. [1]

Other approaches include weave merge [10] and mark merge [10], which do well on the basic cases of the criss-cross merge but cannot handle the more complicated case. More importantly, these merge strategies are predicated on lines and their ordering on files, making it difficult to be applied in the context of objects in state replication.

Finally, an interesting case is that of the DARCs version control system [91]. DARCs explores the semantics of patch algebra and treats a version as the ordered combination of all diffs before it. Diffs that are concurrent are permuted in any order when commutative and raised to the user for conflict resolution when they are not. The result of the conflict resolution is added to the end of the patch history. While DARCs is not immune to the criss-cross problem since DARCs stores patches as first-class citizens, additional information, present in the patches help avoid the simple criss-cross merge problems. The model in DARCs has been formalized [65]. Our formal model presented in this paper builds on this work.

[1] This blogpost gives a good summary of the problems with Git `https://bramcohen.livejournal.com/74462.html`

3.5 Object Based Version Control

There have many variations of object version control proposed over the last decade. In this section, we will explore them.

3.5.1 Concurrent Revisions

Concurrent Revisions was introduced in 2010 by Burckhart et al. as a programming model for parallel processing. It was inspired by version control and built around the fork-join design pattern, which was first introduced and explained in 1963 [27]. This programming model is particularly useful when building applications that require several heterogeneous tasks to read or modify shared data concurrently without the need for complex mechanisms like locks or actors. The idea is similar to typical map-reduce, where many parallel tasks are provided with their own snapshot of the state and are free to modify them as they see fit. At the end of each task's execution, snapshot changes are merged automatically, and conflicts are resolved deterministically (similar to the reduce phase in map-reduce) with the use of type-specific three-way merges. Unlike map-reduce, however, the tasks that are deployed concurrently need not perform the same task.

The revision graph in concurrent revisions is centralized. The main thread, from which all other concurrent tasks fork, controls the graph. When a concurrent task is forked, a new referenced revision is created in the main thread and copied over to the task. The task then makes updates to the task that are within their branch of the revision tree. The tasks can fork again if needed, allowing for a hierarchical branching of the state. When a join occurs, the main thread reconciles the two revisions using a three-way merge. The original revision is known to the main thread as a reference to it is maintained by the concurrent task.

The differences between concurrent revisions and GoT are many. While concurrent revision is

primarily a programming model for parallel computing, GoT focuses on distributed computing. Further, since concurrent revisions is a fork-join model, the approach uses centralized version control. GoT uses a distributed version control with a version graph at each independent node that is performing tasks. Finally, the semantics of concurrent revisions expressly prohibit certain forms of communication (task join patterns) so as not to encounter a situation where the merge cannot be computed. In GoT, we explore the cause of these situations and propose a technique to ensure that the merge is always computed.

3.5.2 Cloud Types

Cloud types [62] extends the programming model introduced in Concurrent Revisions and introduces it to distributed computing. The model is geared towards applications that require object replication in the face of isolation, such as mobile games. The servers host revision graphs that replicate at the clients that fork from them. Each component can then work on the local snapshot and join the changes made back into the revision graph that it forked.

The programming model supports two main primitives: push, pull, and a secondary primitive: flush that gives stronger guarantees. The revision graph in Cloud Types is partly distributed. There can be multiple servers where revision graphs can exist in the system. Each of these servers can connect to multiple clients that do not have revision graphs themselves. Unlike concurrent revisions, the three-way merge is no longer used to resolve write-write conflicts. Instead, specific data types such as Cint (Cloud integers) have been created to handle merges automatically. We believe this takes control away from the user and makes it less general purpose. In Cloud types, these restrictions are in place because the entire replicated state is forked at each revision. If the replicated state can have any data type, creating delta updates is difficult. In GoT, states are never compared. Instead, all changes are marked and recorded as they happen and shared as a patch. The version graph exists as a graph

of patches as opposed to a graph of snapshots. Finally, like concurrent revisions, there are strict communication patterns built into the communication protocol that disallows cross-join patterns. We discuss these patterns in this dissertation and provide a solution called a small-step merge that lifts these constraints in GoT.

3.5.3 CARMOT

CARMOT [54] is a version control based state replication framework, like GoT, built-in OCaml. The API exposes a git-like interface for nodes in a distributed system to push and pull changes between nodes. Write-write conflicts are resolved using custom three-way merge functions. A global version graph exists which tracks the provenance of each version. The applications themselves can pick and choose from different versions and make updates from that. When an update is made, it is checked against any conflicting newer version and merged against that.

While CARMOT is similar to GoT, at the API level, there are very different design choices under the hood. CARMOT, unlike GoT, maintain full copies of the state that are indexed by their SHA1 hash. GoT, on the other hand, stores the version graph as a graph of updates. Each version in GoT is reconstructed from the set of versions preceding it from the ROOT version. Furthermore, CARMOT seems to rely on centralized version control as every sync checks for the presence of globally conflicting updates. The system uses delta communication to communicate between the nodes, but the diffs are calculated, at each step, from the full versions. Finally, CARMOT does not seem to have a garbage collection strategy. Since nodes can traverse back in time to an older update, garbage collection is not possible. The examples they use – blockchains – explicitly have to remember every transaction and require that the versions and the updates be stored in more permanent storage. GoT, on the other hand, is purely designed for applications with a shared state that is quickly mutating and has to

quickly share many updates.

CARMOT, however, takes a unique approach to solving the patterns banned by concurrent revisions and cloud types. They provide certain data types with restrictive alternate forms of representation that are more suitable for reconciliation into remote states. These restrictive forms can compose together correctly despite the inconsistencies introduced by the join patterns. They propose Mergeable Replicated Data Types [52] that take this principle and show how it can be applied in a more general context. This system is inherently compatible with the GoT programming model as the merge functions in GoT are entirely left to the design of the programmers letting them implement mergeable replicated data types style merges for consistent merges.

3.5.4 TARDiS

TARDiS [30] is an asynchronously replicated, multi-master key-value store that tries to address some of the problems of ensuring scalable causal consistency using concepts in centralized version control. Many of the design choices in GoT are also found in TARDiS and for similar reasons. Neither TARDiS nor GoT abstract the interaction with the data as sequential operations. Both have explicit primitives to observe and resolve concurrency and distribute changes.

There are, however, several significant differences. TARDiS is, first and foremost, a database. A database like TARDiS expects multiple applications to connect to them. Each application interacting with a TARDiS store is given its own branch of a shared DAG to execute in, and merging conflicts becomes a shared task. In GoT, the repository of objects present in each node is not a database and is not shared by multiple nodes. Each node is solely responsible for resolving the conflicts in its repository, much like in Git. Another significant difference between TARDiS and GoT is that in GoT, the version graph is stored as a graph of deltas

while in TARDiS, they are stored as full copied revisions. With full revisions instead of deltas, calculating deltas for patching is costly. Since the number of entries in a delta is less than or equal to the number of entries in the full version, it is faster to apply a delta on a version than to create a delta by diff-ing two versions.

3.6 Optimizations in Shared-Space Applications

The topic of update latency, though it has not been explicitly called that, has always been among many problems in the networking of multiplayer games and multi-agent distributed simulations [102, 51]. There exist several architectural solutions to different aspects to optimizing update latency for online multiplayer gaming [57, 116, 42, 84, 61, 13, 31, 26] and multi-agent simulations [72, 29, 93, 92]. Each of these solutions and more, optimize different aspects of the problem and combining them together into a single solution is quite hard without a proper programming structure. Our work with the GoT model attempts to provide this by leaning on the familiar concept of version control systems. By presenting inter-node communication control as a version control system, we can tactically employ several of the industry and academic strategies that exist for reducing update latency.

3.7 Research Gap

In this section, I identify the research gap in the related literature by answering the following research questions:

- Update latency: How do existing programming models help programmers optimize for update latency in shared-space applications?
- Version Control Systems: Are existing version control models sufficient?

Table 3.1: Design Goals for GoT

	Interest Management	Delta Updates	Isolation	Non-restrictive Updates	P2P
Message-passing					
Actor Models	✓	✓	X	✓	✓
GSP, Cloud Types	(✓)	✓	X	✓	✓
Shared-state					
Server mediated-replication	(✓)	(X)	✓	✓	X
BASE Datastores	✓	X	✓	(✓)	(✓)
ACID Datastores	✓	X	✓	✓	X
CRDTs					
State-CRDTs	✓	X	✓	X	✓
Op-CRDTs	✓	✓	X	X	✓
δ-CRDTs	✓	✓	(X)	X	✓
Version Control					
Concurrent Revisions	✓	(X)	✓	✓	X
TARDiS	✓	(X)	✓	✓	X
CARMOT	✓	(X)	✓	✓	(X)
Global Object Tracker	✓	✓	✓	✓	✓

3.7.1 Update latency

The shared-state programming models, discussed in this chapter, are designed to keep the state of the distributed system at a common location. The features built into shared-state programming models typically focus on the access control of data, the correct application of concurrent updates, and the availability of the data store to requests. Since only a small subset of applications require the fast propagation of updates, shared-state programming models optimize for goals such as the scalability of the data store to a large number of clients or low latency of response at the cost of update latency.

Message-passing programming models, discussed in this chapter, are designed for the quick propagation of updates. However, since no shared state exists for the distributed system, the programmer has to ensure the correctness of the state changes at each node in the system. Additionally, the tepid support for pull based communication and isolation in these models makes it such that the programmer has to engineer solutions to guard against network

partitioning and recovery. In shared-space applications, this can lead to significantly more complex systems as the programmer has to build mechanisms to keep the shared-space consistent at every node and over byzantine network failures.

There is a need for a programming model to combine the fast update propagation of message-passing with the ease of manipulating the state of the system that comes with shared-state programming models.

3.7.2 Version Control Systems

Version control is a model, as we discussed in Chapter 2, that can combine fast update propagation with easy state manipulation, by using state replication. The model explicitly defines procedures for the correct replication of states.

While there is potential in version control models, existing version control models like TARDiS, concurrent revisions, and version based shared memory transactions do not cater to update latency. Their primary design objective is to provide snapshot isolation, where concurrent updates are made to the same shared-space (using replicas) but do not affect each other until they are resolved using powerful conflict resolution strategies. Their focus is on the isolation of the updates. The quick propagation of the delta differences between the snapshots (which is the update) is, however, ignored.

The delta difference between snapshots is utilized in file-based version control systems where entire snapshots can be of significant size. The same approach should be applied to object-based version control models to allow for the quick propagation of updates. Such a model, however, does not exist.

3.7.3 Design Goals

Table 3.1 summarizes the support provided by the models and technologies, discussed in this chapter, for the requirements of shared-space applications. We can see that message-passing models do not support isolation while shared-state models do not support delta updates. CRDTs like message-passing do not support isolation. However, since the messages are considered to be idempotent, a single queue of updates would be enough in δ-CRDTs to provide isolation. However, as they support eventual consistency, they restrict the type of updates that are possible. Complex state manipulations are normal in shared-space applications and therefore, δ-CRDTs are not suitable for shared-space applications such as multiplayer games.

Version control models, can theoretically support interest management, delta updates, isolation, and non-restrictive updates. However, the design choices made in existing version control models, make delta updates difficult. Nevertheless, the effort of engineering delta updates over these solutions should not be high. Version control systems, however, fail when put in peer to peer network topologies. Version control systems, including file-based systems cannot handle specific concurrent scenarios that we elaborate upon later in the dissertation.

My goal with the Global Object Tracker (GoT), is to provide a programming model for shared-space applications that meet all the goals shown in Table 3.1.

Chapter 4

GoT Programming Model

A GoT application consists of many nodes, called GoT nodes, that perform tasks asynchronously within the distributed application. The GoT nodes share among themselves a collection of objects. Each of these nodes can be executed in different machines, communicating the changes to the replicated objects via the network. What is unique about GoT is that the synchronization of object state among the distributed nodes is seen as a version control problem, modeled after Git.

All GoT nodes that are part of the same Spacetime application have a repository of the shared objects, called a dataframe. The dataframe is similar to a normal, non-bare Git repository,[1] and, like a Git repository, it has two components: a snapshot and a version history, as shown in Figure 4.1. The snapshot, analogous to the staging area in Git, defines the local state of the node. All changes made by the application code on the local dataframe are first staged in this snapshot. The version history, on the other hand, is the published state of the node. Like in Git, changes can be moved from the staging area to the version history using the *commit* primitive, and the snapshot can be made up to date with the latest version in the

[1] In Git, repositories may be bare or non-bare. Bare repositories contain just the history, but not a copy of the files. We do not have bare dataframes, although those may be introduced in the future if they prove to be useful.

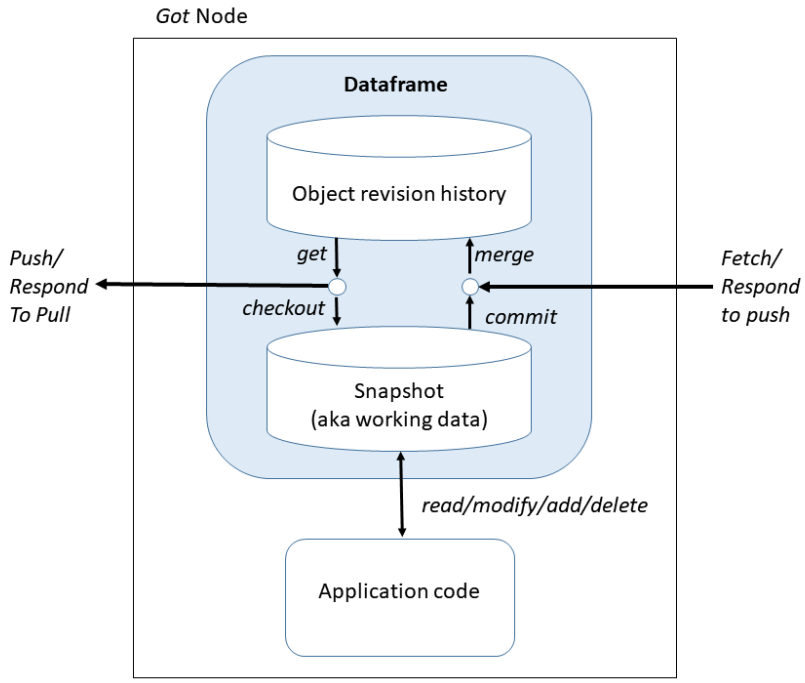


Figure 4.1: Structure of a GoT node. Arrows denote the direction of data flow.

version history by using the *checkout* primitive. Inter-node communication happens using *push* and *fetch* requests, used to communicate updates in version histories between nodes.

When the version history at a node receives changes (via *commit*, *push* or *fetch*), a conflict with concurrent local changes is possible and must be resolved. In Git, conflicts are resolved manually by the user, and only on a *fetch*. However, in GoT, conflicts are resolved automatically, at the node receiving the changes, and irrespective of the primitive used. Automatic conflict resolution is achieved via programmer-defined three-way merge functions that are invoked when conflicts are detected.

The APIs supported by the dataframe is shown Table 4.1; this table can be used as a quick reference for the API calls in the example explained next.

Table 4.1: API table for a dataframe

Dataframe API	**Equivalent Git API**	**Purpose**
read_{one, all}	N/A	Read objects from local snapshot.
add_{one, many}	git add [untracked]	Add new objects to local snapshot.
delete_{one, all}	git rm [files]	Delete objects from local snapshot.
	git add [modified]	Objects are locally modified which is tracked by the local snapshot.
commit	git commit	Write staged changes in local snapshot to local version history.
checkout	git checkout	Update local snapshot to the local version history HEAD.
push	git push	Write changes in local version history to a remote version history.
fetch	git fetch && git merge	Get changes from remote version history to local version history.
pull	git pull	fetch and then checkout.

4.1 GoT Example: Multi-bot Space Race

Before getting into the underlying design of GoT, we begin by using the example established in Chapter 2: Attari's-inspired multi-bot Spacerace, to explain the structure of GoT applications. Figure 4.2 shows the same example, rewritten using version control.

4.1.1 Data Model

The development of GoT applications starts by identifying the types of objects that will be tracked by the dataframe and shared under version control. In these applications, "objects" are actual programming language-level objects. As such, they are defined via classes or types. These classes/types need to define which parts of the objects should be tracked, and which parts should not. We have chosen to do this statically and declaratively, as changing the data model at runtime would complicate the data synchronization model unnecessarily. Figure 4.3 shows the two classes of objects shared in the Space Race game, along with their most important methods.

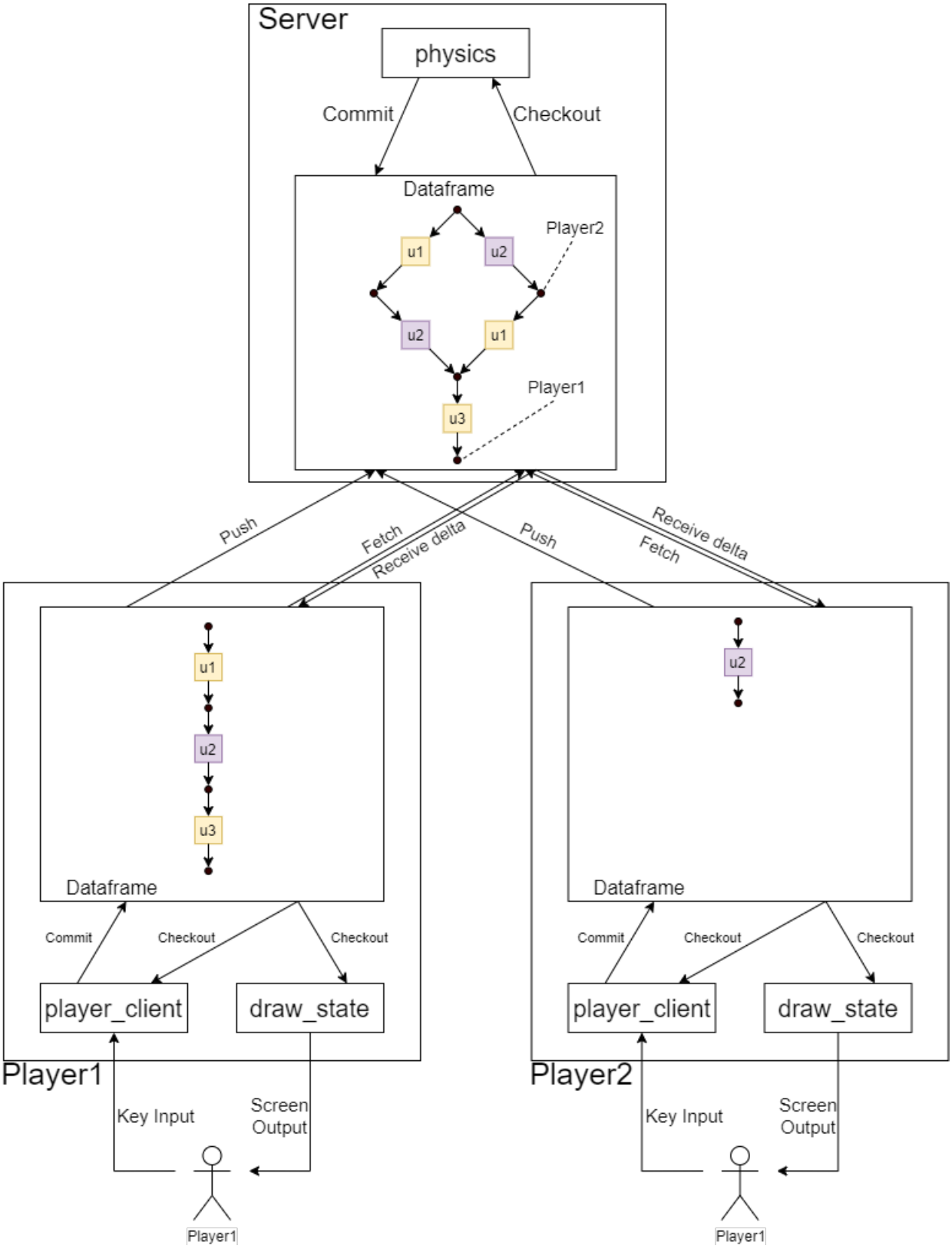


Figure 4.2: Spacerace using version control.

```
@pcc_set
class Player(object):
    oid = primarykey(int)
    crashed = dimension(bool)
    winner = dimension(bool)
    position = dimension(tuple)
    velocity = dimension(tuple)

    def __init__(self, object_id):
        self.oid = object_id
        ...other initializations,
           including non-shared fields...
        # Example of non-shared field
        self.world = World()

    def act(self, asteroids, players):
        ... do smart things with the ship...
    ...Other functions...

@pcc_set
class Asteroid(object):
    oid = primarykey(int)
    position = dimension(tuple)
    velocity = dimension(tuple)

    def __init__(self, object_id, pos, vel):
        self.oid = object_id
        self.position = pos
        self.velocity = vel

    def move(self):
        ... attempt brownian motion ...
```

Figure 4.3: The data model for the multiplayer Space Race game.

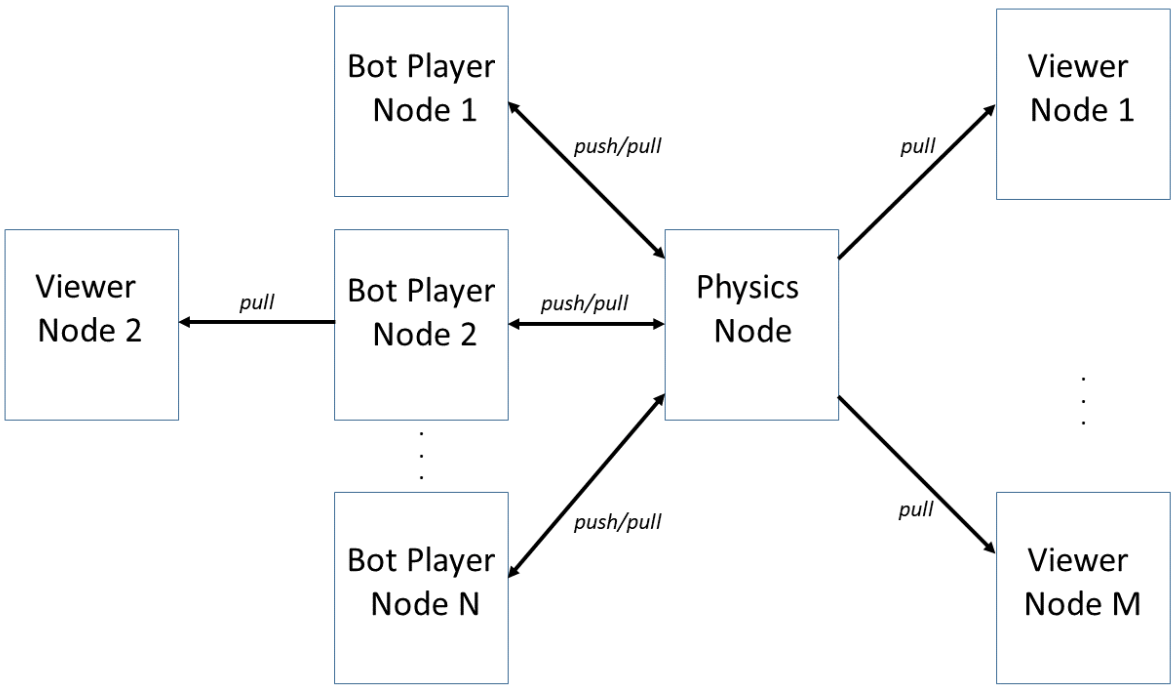


Figure 4.4: Structure of the Space Race distributed application. Each node resides on a separate process/machine/geographic location.

The first thing to notice in Figure 4.3 are the extra declarations using our language: **@pcc_set** (decorator of classes), and **primarykey** and **dimension** (special attributes). The declaration **@pcc_set** indicates that objects of that class are to be tracked. The other declarations define which fields are to be tracked. For example, the Player class defines five dimensions that must be tracked, one of which, `oid`, is the primary key that uniquely identifies the object in the entire universe of objects. All tracked objects can define a primary key field, which must be unique. If a primary key is defined, then objects can be picked up from the dataframe using the key. If it is not defined, the dataframe assigns a unique random id to the object, and the objects can only be retrieved as a part of the collection of objects of that type. The Space Race example also defines an Asteroid class that defines the properties of the obstacles faced by the players.

```
1  def physics(df):
2      df.add_many(Asteroid, [Asteroid() for i in range(NO_ASTEROIDS)])
3      df.commit()
4      while True:
5          start = time.perf_counter()
6          # Pick up new updates
7          df.checkout()
8          # state changes.
9          for a in df.read_all(Asteroid):
10             a.move()
11         for p in df.read_all(Player):
12             p.move()
13             p.crashed = is_crashed(p)
14             p.winner = reached_finish(p)
15         # Make changes public.
16         dataframe.commit()
17         # Wait for remaining game loop time.
18         elapsed = time.perf_counter()-start
19         time.sleep(DELTA_TIME-elapsed)
20
21 def main(port):
22     node = GotNode(physics, server_port=port,
23                    name="spacerace.got",
24                    Types=[Player, Asteroid])
25     node.start()
```

Figure 4.5: The Physics simulator node.

4.1.2 GoT Nodes

There are three types of GoT nodes in Space Race: (1) the world/physics simulator, (2) a simple player bot, and (3) a simple viewer that displays the state of the world. The physics simulator is to be the authoritative component on the majority of the state of the simulated objects, but not all. We explain this in the next section. The physics simulator also doubles as the enforcer of the game rules. Figure 4.4 depicts the overall structure of a Space Race deployment. Because there is only one shared world/game, there is only one physics simulator, but there can be any number of player nodes and any number of viewer nodes. There can also be other types of nodes, such as nodes with UI for human players, nodes for compiling statistics, etc., but, for simplicity sake, we do not describe those in this paper.

Figure 4.5 shows almost the entire code of the Physics node, with just a couple of methods and constants missing. Starting at the bottom, in lines 22–25, we create the GoT node for the physics simulation. `GotNode`, as the name indicates, is our main class for creating GoT nodes. Its arguments are, from left to right: the entry function to run the node, the port to listen on the network for push and pull requests, the name of the local dataframe, and the types of objects that are to be tracked in the dataframe of this node. As explained in the previous section, these need to be declared as **@pcc_set** with dimensions to be tracked. The entry function is shown in lines 1–19. These entry functions always receive a dataframe (labeled df) as an argument to perform the data versioning operations on it. In this case, we initialize the environment by adding a prebuilt list of asteroids (ASTEROIDS) into the dataframe (line 2). The dataframe from here on tracks any modifications to the objects in the list. After all changes are added to the dataframe, we commit the changes, so that the asteroids will be effectively under version control (line 3). After initialization, the game is played (lines 4–19). The game operates in a loop that mainly simulates physics, but also has some game functions. At every iteration, it performs a checkout of the objects (line 7), implements the logic (lines 9–14), and commits the changes at the end (line 16). Lines 9–14 are the main physics functions: asteroids and player ships are moved by delta time, and collisions are dealt with (lines 13–14). These operations change the shared state of these objects, which change fields that are under version control. The physics loop runs in 20 fps, so 50ms frames[2] that, in our case, also determine the rate of checkout/commit operations in the Physics node.

A simple type of player is shown in Figure 4.6, showing the most important snippets. One important difference between the Player node and the Physics node presented before is embodied in line 21 of Figure 4.6. Here, we are binding the player node to a remote dataframe given by the command line argument `physics_node` (line 21). This argument is

[2]"Frame" and "fps" are part of the standard terminology of physics and game engines, as they all operate with these precisely-timed loops explained here. Each precisely-timed iteration is a "frame," not to be confused with our dataframes (although the two concepts are related).

```
1  SYNC_TIME = 0.3 # secs
2  def player_client(df, player_id):
3      df.pull()
4      my_player = Player(player_id)
5      df.add_one(Player, my_player)
6      df.commit()
7      df.push_await()
8      while not my_player.crashed and not my_player.winner:
9          start = time.perf_counter()
10         df.pull()
11         my_player.act(df.read_all(Asteroids), df.read_all(Player))
12         df.commit()
13         df.push()
14         elapsed = time.perf_counter()-start
15         sleep_t = SYNC_TIME - elapsed
16         if sleep_t > 0:
17             time.sleep(sleep_t)
18
19 def main():
20     args = ... # parse command line args
21     player_node = GotNode(player_client, remote=[(args.physics_node)],
22                           Types=[Player, Asteroid])
23     player.start(args.player_id)
```

Figure 4.6: The Player nodes.

expected to be a `got` URL, of the form `got://somehost.edu[:port]/spacerace.got`. This method of binding is equivalent to defining the `remote` origin in Git. In this case, this is supposed to be the URL of the Physics simulator dataframe (see Figure 4.4). Pull and push operations will be directed to this remote dataframe. Our `player_client` function (starting in line 2) first pulls all the objects from the remote dataframe (line 3), then creates a Player object and adds it to the dataframe (lines 4–5), and then commits the changes locally and pushes them to the remote dataframe awaiting confirmation that the submitted change has been accepted (lines 6–7). This is so that the physics simulator/game receives the new player object when it performs a checkout from its local dataframe (line 7 in Figure 4.5). From then on, our bot is on a loop of pulling the objects (line 10), doing some local actions on them (line 11), committing the changes locally (line 12), and pushing them to the remote dataframe (line 13).

Our player node is on a timed loop of 300ms (lines 1 and 17), so it only checks the shared state every 300ms. Other player nodes may want to do something different. In any case, this is where network latency plays an important role that cannot be ignored. We are doing push/pull operations to a remote dataframe. Depending on how much data is to be synchronized, and the relative locations of the physics node and the player node, each of these operations can take anywhere from 20ms to 200ms or more. As such, the player nodes will always be "behind" the physics node in absolute time, no matter how fast they pull. This delay is an essential aspect of these kinds of applications. It has a tremendous influence on their design; the effect will be even more evident in the description of viewer nodes.

Viewer nodes are observers of the simulated world, and their local changes are not supposed to be propagated to other nodes. In our implementation, viewers are graphical components (we use `pygame`) that display the state of the simulation in a nice, smooth manner. In order to animate the objects, our viewer simulates them at 60fps. This simulation has two benefits: the animation is smooth, and state synchronization becomes simply a matter of compensating

```
1  SYNC_TIME = 0.5 # secs
2  def sync(df, world):
3      while True:
4          start = time.perf_counter()
5          df.pull()
6          world.ships = df.read_all(Player)
7          elapsed = time.perf_counter()-start
8          time.sleep(SYNC_TIME-elapsed)
9
10 def draw_state(df):
11     df.pull();     world = World()
12     world.asteroids = df.read_all(Asteroid)
13     vis = Visualizer(world)
14     threading.Thread(target=sync,args=[df,world]).start()
15     vis.run() # Run pygame loop.
16
17 def main():
18     args = ...parse arguments...
19     vis_node = GotNode(draw_state, remote=[(args.physics_node)],
20                        Types=[Asteroid,Player])
21     vis_node.start()
```

Figure 4.7: The Viewer nodes.

for drifting clocks and for acquiring the new objects that are added to the world by other nodes, rather than being the primary means of knowing the state of the world. Rather than trying to be always in lockstep with the physics simulator, something that would be unreliable, the viewer assumes its role as an autonomous component that operates on its copy of the shared state.[3]

The viewer is the most complex node in the Space Race example because it executes two threads: the data synchronization thread and the pygame thread. Figure 4.7 shows one of the two parts of the viewer nodes, namely the entry point and the data synchronization thread. Similarly to the player node, the viewer node also binds to the remote dataframe given by the `physics_node` URL in the command line (line 19). The entry function (lines 10–15) starts

[3]What we did with our viewer follows the standard practice in the gaming and simulation industries. The difference between our model and the many approaches in use today is our use of version control as the fundamental organizing principle of distributed shared objects.

by pulling the objects and storing the pulled asteroid objects in a local world context. More importantly, it spawns the data synchronization thread (line 14), and then proceeds to run the `pygame` visualization object (line 15). At that point, there are two threads: one that is updating the screen at 60fps, changing the state of asteroids and ships along the way (not shown in this listing), and one that pulls data from the remote dataframe (lines 2–8) and updating the asteroids and ships held by the pygame thread. Note that the `sync` function (lines 2–8) never commits or pushes; it just pulls.

The viewer changes the state of the shared objects for purposes of animation. However, those changes are not propagated to the physics node because the data synchronization thread in Figure 4.7 never commits or pushes. As mentioned before, this is by design: the viewer is just an observer of the shared state, so when it receives the updated state from the physics server, it merely discards its estimates.

4.2 Dataframe: Object Repository

The core of our model is centered around the dataframe. The dataframe is a specialized shared object heap used by the application code in every node in GoT. Each node in GoT gets its own dataframe to manage the state of its objects[4]. As an object heap, the dataframe must satisfy two important constraints: read stability, and deterministic state update. To satisfy these constraints and better isolate the application code from the effects of external updates, the heap is divided into two components: a snapshot that provides read stability, and a version graph that detects and manages concurrent updates. The snapshot holds the local copy of the objects used by the application code in the node. The version graph keeps track of the published history of shared objects. It can receive both updates committed locally by the node, and updates made externally by external nodes. Updates to the version

[4]For simplicity's sake since there is only one dataframe per node, we use these two words interchangeably throughout the dissertation.

graph do not affect the snapshot.

4.2.1 Snapshot

The role of the snapshot in the dataframe is to provide the local application code with a copy of the shared objects whose updates are under the control of the local code. The state of the snapshot is observable only to the local code. As is typical for an object heap in any language, the application code can read objects, create new objects, delete existing objects, and modify objects. All these reads and writes are executed upon the snapshot. All writes that are made (updates, additions, deletions) are staged in the snapshot, ready to be bundled as a diff and applied to the version control graph on a *commit*. The local application code can also update the local snapshot, using the *checkout* primitive, where new updates in the version graph (received from remote nodes) are picked up as a diff and applied to update the local object state.

4.2.2 Object Version Graph

The version graph is a Directed Acyclic Graph (DAG). It maintains the history of changes that have been applied to it. Each vertex of the graph represents a version of the state. Each directed edge from a source version to a destination version represents the delta change required to change the state at the source to that at the destination. The version graph at each node is analogous to the cloned repositories in Git, each of which maintains its history of changes in a directed acyclic graph. The vertices of the Git history are labeled snapshots of the files, and edges of the git history are commonly associated with a diff of changes.

The full representation of each version is never stored in the version graph. The start version, called ROOT, represents an empty state. A globally unique version identifier represents each

version. Only the edges and the delta changes associated with these edges are stored. The representation of the version history as a chain of diffs is not new to distributed version control systems and is the defining feature of DARCS [91] distributed version control system. The key advantages that this offers will be discussed in the next chapter.

The version graph terminates at the HEAD version (using the same nomenclature as Git), which can be considered the latest state version of all the objects at that GoT node. The version graph is the published state of the node in the distributed system and is visible to all the nodes in the system. The latest version of the node, as observed by all the other nodes, is the HEAD as the snapshot cannot be observed, and any updates staged in the snapshot are not observed. The version graph caters to two types of requests: retrieve updates (*fetch*, responding to *push*, and *checkout*), and receive updates (*push*, responding to *pull*, and *commit*). These will be discussed in detail in the next few sections.

4.3 Snapshot and Version Graph Interaction

As explained above, in addition to the standard object heap operations, the snapshot also defines two primitives that allow the Nodes to move data between the snapshot and the version graph: Commit, and Checkout.

When a commit is invoked, a new, globally unique version identifier is created, representing the current state of the snapshot. All the updates staged in the snapshot represent the transformation, from the state at the previously synchronized parent version of the snapshot to the newly created version. A new edge is created in the version graph from the previous synchronized parent version to the newly created version. Once the version graph has accepted the update, all changelogs in the snapshot are cleared, and the snapshot's previously synchronized parent version identifier is updated to the newly committed version.

The snapshot can be updated with the latest updates in the version graph by executing the checkout primitive. In a checkout, all delta changes, in the version graph, since the snapshot's previous synchronized version are merged in order (to retain their causal relation) and applied to the state stored in the snapshot. This update brings the state of the snapshot to the HEAD version in the version graph.

The interaction between the snapshot and the version graph (checkout and commit) is controlled by the local application code and can be called when required by the node. Since this is the only way for external updates to be introduced to the snapshot, the snapshot promises read stability.

4.4 State Replication in GoT

With data in the dataframe present as a graph of delta changes, sending, receiving, and requesting changes to other dataframes is simplified. Dataframes provide the application code with two primitives that allow the node to retrieve and receive data from other dataframes explicitly: pull, and push, respectively. A dataframe that wants to initiate requests to another dataframe must first register the remote dataframe. Each remote dataframe is stored as an address, which is also mapped to the last version known for that node, the default being the ROOT version.

To aid with the explanation of state replication between version graphs, we introduce another example of a GoT application – distributed counter. In a distributed counter, the replicated state at each GoT node, is reduced to a single shared counter x. Each GoT node increments the counter by an arbitrary value and shares the current state of the counter with an authoritative GoT node. According to the semantics of the counter, concurrent increments to x have to be aggregated.

As explained in the previous section, each GoT node has a version graph, a DAG, that stores the history of changes to the replicated counter x. The vertices in the version graph are versions of the state of the counter.

The directed edges between the two versions define a happened-after relation between the versions. Figure 4.8 shows the version graph of a node N_1. We see that version B happened-after version A. Each edge is also associated with an update either as a domain-specific command, or a state delta. The edge $A \rightarrow B$ is associated with the update u_1. When u_1, which is an increment to the counter by one, is applied to state A where $x = 0$, we get state B where $x = 1$.

Although every node in the system starts at the same initial state – ROOT, in this example, we show the initial state to be version A, where $x = 0$. The HEAD version is the version with no children (i.e., a version for which there are no other versions that happened-after) and in Figure 4.8, the HEAD version of N_1 is B.

If N_1 were to make an update to the local state from B using a commit, a new version identifier representing the updated state would be created, and a happened-after relation would be established from B to this new version. The update recorded would be associated with this newly created edge, and the HEAD at N_1 would be the newly created version. Each new version identifier can be assumed to be globally unique.

It is guaranteed, like in Git, that if two versions that have an edge exists in the version graphs

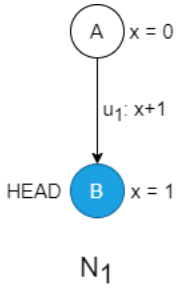


Figure 4.8: Version Graph at Node N_1.

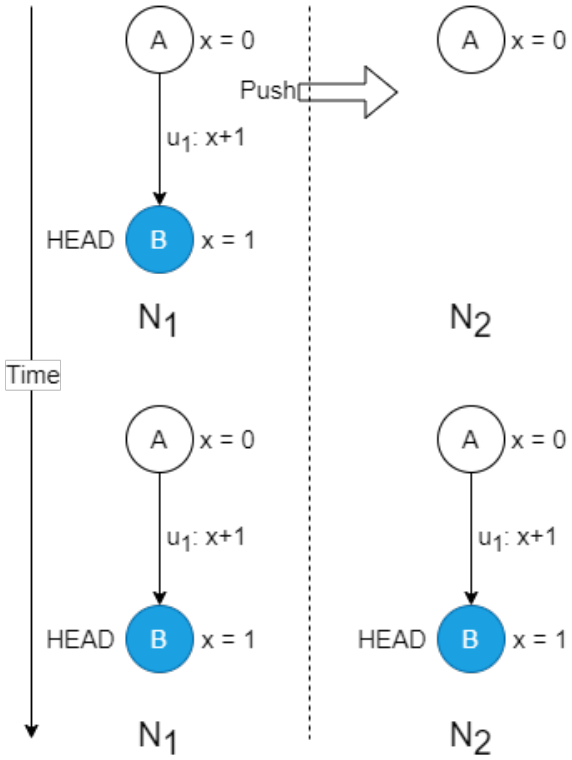
at multiple nodes, then the edge in every version graph will be associated with the same update (or diff in Git). For example, if A and B as seen in Figure 4.8 exists in more than one node, say N_1, and N_2, then the edge $A \rightarrow B$ in both N_1 and N_2 will both be associated with the update u_1. State divergence can e kept in check via inter-node communication, which can be either push or pull communication.

4.4.1 Communication between Version Graphs

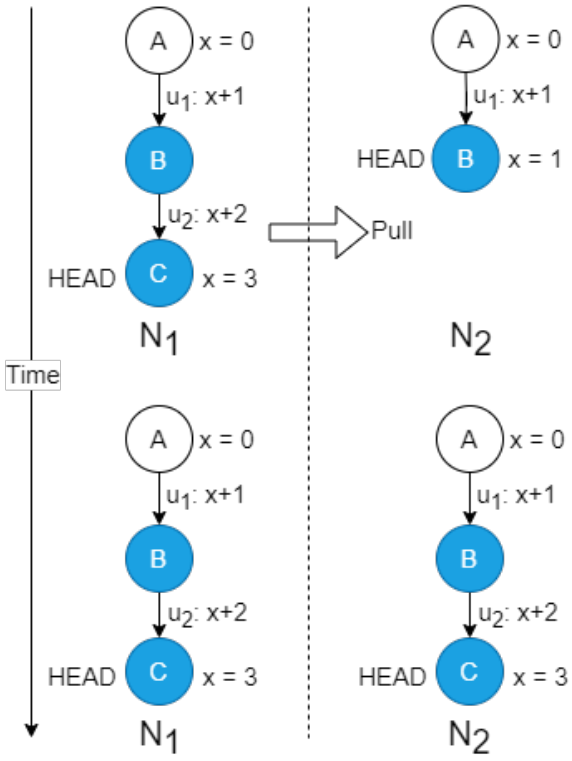
Figure 4.9 illustrates how basic inter-node communication (both push and pull) works with state replication modeled on version control systems.

In Figure 4.9a, we see the object version graphs at two nodes N_1 and N_2. Both N_1 and N_2 start at the same version A where $x = 0$. N_1 has an additional update u_1 where x is incremented by one giving a new state B that happened-after A and has $x = 1$. N_1 pushes all state changes to N_2 since version A, and therefore, sends update u_1 and version B to N_2. The version graph at N_2 is updated to reflect the result of the push. HEAD at N_2 is moved from A to B.

In Figure 4.9b, there are again two nodes N_1 and N_2 which start at the same version A ($x = 0$) as before. N_1 has two updates $u_1 : x + 1$ and $u_2 : x + 2$ that happened in sequence giving versions B ($x = 1$) and C ($x = 3$) respectively. C happened-after B that happened-after A. N_2 on the other hand, only knows the update u_1 and therefore, it only has states B that happened-after A. When N_2 pulls changes from N_1, update u_2 is sent by N_1 bringing HEAD at N_2 to state C.

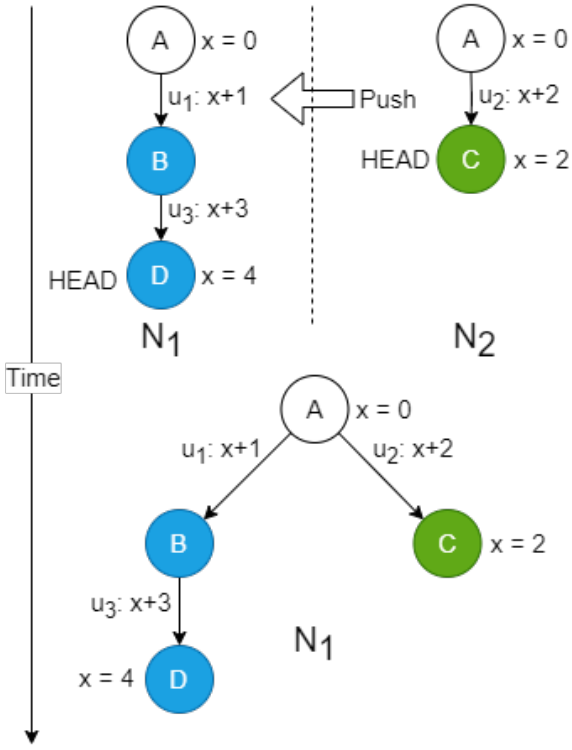


(a) Basic Push from Node N_1 to N_2.

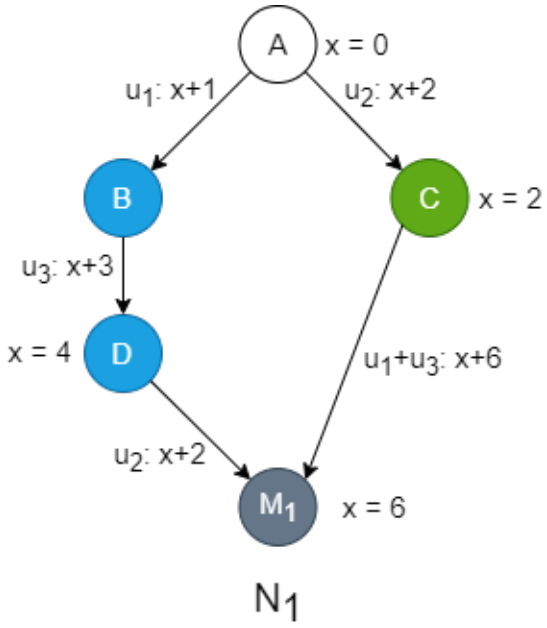


(b) Basic Pull from Node N_2 to N_1.

Figure 4.9: Inter-node Communication in GoT.



(a) Conflict Detection.



(b) Conflict Resolution.

Figure 4.10: Conflict Detection and Resolution.

4.4.2 Conflict Detection

One of the strong advantages of using a version graph to track object changes is that detecting conflicts become trivial. A conflict is detected between nodes when revision trees are merged, and divergence of state versions is detected. This means that different nodes have performed different computations after having read the same state.

In the case of the Space Race application, what we have shown so far is free from conflicts. However, conflicts would arise, for example, if the AI player would set the position of the ship directly (the player, shown in Figure 4.3, sets only the ship's velocity, and that is how all players are supposed to control the ship). This condition might happen if the player was trying to cheat, but it also might happen accidentally. In that case, there would be conflicts between the player node and the physics node on the `y` (and even `x`) fields of the player's position.

In the case of the simple counter, in both the basic inter-node communication shown above, there are no concurrent updates, and therefore, no conflicts. Figure 4.10a shows both conflict detection. In Figure 4.10a, we have two nodes N_1, and N_2 that, again, start at the same version A ($x = 0$). From that common version, however, each node has performed a different concurrent update. N_1 has made two updates $u_1 : x + 1$, and $u_3 : x + 3$ in sequence, moving the state from version A to B ($x = 1$), and version B to version D ($x = 4$) respectively. N_2, on the other hand, has made one update $u_2 : x + 2$ moving the state from A to C ($x = 2$). If N_2 pushed the update u_2 to N_1, the object version graph at N_1 would have four versions, A, B, C and D. D happened-after B that happened-after A, and C happened-after A. There is no causal ordering between (B and C) and (D and C). These versions are concurrent versions.

This state is a conflict and is relatively straightforward to detect. Note that all concurrent updates from the same version are considered to be in conflict for a version graph. The

```
def conflict_resolution(conflict_iter,
                        original_snap, your_snap, their_snap):
    for orig,yours,theirs in conflict_iter:
        if isinstance(yours,Player):
            if any(v <= World.MAX_SPEED for v in theirs.velocity):
                yours.velocity = theirs.velocity
            your_snap.resolve_with(yours)
        else:  # if it is an asteroid
            your_snap.resolve_with(your)
    return your_snap
```

Figure 4.11: Programmatic conflict resolution for the Physics node.

semantics of the state update is not taken into consideration. In this example, even though $u_1 + u_3$ and u_2 are not semantically in conflict and can be applied correctly in sequence, in any order, version control detects them as conflicting updates as both updates are applied to the same initial state A, giving two possible futures for A. To be specific, conflicts due to concurrent updates are detected when the object version graph has multiple HEAD versions (versions with no children) after receiving an update.

4.4.3 Conflict Resolution: Big-Step 3-way Merge

In Git, conflicts are exposed to the users: users have to edit the files and decide whether to keep their own version, or the remote version of the modifications. Clearly, for in-memory real-time application objects, the human-in-the-loop approach would be unfeasible. Instead, we need a mechanism for automatically resolving conflicts. Spacetime supports automatic merge conflict resolution by allowing programmers to declare functions specifically for that purpose. For example, additionally to the code seen in Figure 4.5, the physics node has the following function declared:

This strategy means that upon a merge conflict on any object of class Player, the physics node uses the velocity set by the conflicting node, if it is under the maximum allowed speed,

```
def three_way_merge(original, yours, theirs):
  return yours + theirs - original
```

Figure 4.12: 3-way merge function for Counter.

but maintains its own version of everything else. In this case, this policy makes sense, because the physics node is authoritative for purposes of deciding positions of the objects, but not their velocity – as long as the velocity honors the rules set by the physics node. In the case of asteroids, no node other than physics is supposed to change any of their parameters; as such, the merge policy simply keeps the physics' node version. For other situations, the merge policies will be different.

The merge policy used also varies from application to application. Figure 4.12 shows an example of a 3-way merge function that applies to the simple counter.

The effect of using this merge function to resolve conflicts is seen in Figure 4.10b. As we can see, with the 3-way merge returning a new state M_1 ($x = 6$), the version graph is updated. M_1 happened-after both D and C and edges are added to the version graph to reflect this relation. Version M_1 is the state of the system having applied both updates u_1, u_2 and u_3 to the state at the lowest common ancestor version A. Two orderings are present in the version graph merged with big-step 3-way merge: u_1 followed by u_2 followed by u_3 ($A \rightarrow B \rightarrow D \rightarrow M_1$) and u_2 followed by the combined effect of u_1 and u_3 ($A \rightarrow C \rightarrow M_1$).

We call this 3-way merge procedure *big-step 3-way merge*, as versions across different generations from A, are merged. The term generation is used here to define the distance from the LCA. In this case, A. C and B are in the same generation, while D is in the next generation. In a big-step 3-way merge, the HEAD versions of each branch are simply merged, independent of their generation.

An interesting observation is that since the snapshots have the entire state, semantic merges are possible. Additionally, optimizations can be made in the merge function to avoid iterating

over independent and unrelated concurrent updates. It cuts the time taken to accept concurrent updates, thereby decreasing update latency. Semantic merges are made possible by the presence of the original snapshot. Mechanisms like version vectors [83] employed in industry-standard databases, like Riak [9] and Apache Cassandra [23], are capable of detecting conflicts, but are not capable of determining the original version from which they deviated.

The state of the snapshot returned by the merge function is treated as the correctly merged state, and a new version is created for this merged state. Since this merged state "happened-after" all the diverging states, edges are created connecting these diverging edges to the merged version. Diffs are generated for each of these new edges as a difference between the two states. The computation of these diffs is simply a union of updates in each divergent path that were not in conflict and the updates that were executed in the custom merge. A plethora of built-in merging strategies can also be used for trivial cases.

This method of resolving the version graph is similar to the approach of Operational transformation [86] (OT) with a transformation diff being generated at the detection of conflict to "correct" the deviations. However, there are important differences between the approaches. In traditional OT, two functions must be created that, when applied on the divergent states, transforms them into the common resolved state. Different data types would need to be handled differently, making the creation of these functions extremely difficult. In GoT, we do not create different transformation functions. We create different diffs that are applied to the divergent states using the exact same function and bring them to an identical state. While transformation functions can be hard to generate, generating the diff required to bring one state to another is straightforward. We create the diff by making a log of all object dimensions that are in the newer state but not present, or different in the older state, and all objects that were deleted. We call this *Delta transformation*

It has been proven [20] that a version graph constructed along these conflict resolution rules

is a partially ordered set of updates that form a semi-join lattice.

4.4.4 Responsibilities of Merging

In Git, conflicts are not resolved on a push. Instead, when there are conflicts, the receiving repository rejects the push request. The user pushing the changes has two options: push the changes in as a new branch, which defers the resolution to a later time, or pull the new changes in the remote repository, resolve the merge locally, and then push the merged state. In the case of GoT, neither options are appealing. If we create new branches on conflicts and not merge, the nodes keep drifting apart until the deferred merge is invoked. If the state has branched multiple times, it becomes computationally expensive to merge states all at once. If we choose the second option (i.e., reject the push), the sender may never be able to push changes, as the remote node might progress through the states so quickly that the sender is always rejected, starving the sender. For example, a node trying to set the ship's velocity might never be able to do it, because the physics simulator keeps committing new values to the ship's position every 50ms.

To counter these effects, we allow conflict resolution also on the receiving end of a push request. That is the case shown in both Figure 4.11 and Figure 4.12. A good consequence of this is that each node has complete control over its version graph and can merge the conflict using the logic that they have. In the Spacerace example, the physics node can always decide to accept its changes over the changes pushed by the players. The visualizer can always decide to accept the changes pulled from the physics node over the changes that it has simulated. In the counter example, the resolution is the same at every node, and they aggregate the counts from concurrent updates. The rules of permissible merge functions change based on the network topologies used and will be discussed in later chapters.

4.5 Consistency Model

GoT provides causal consistency as the data consistency guarantee. The "happened-after" relation between versions in the version graph captures the causal relationship between operations in the system. Any updates made and exchanged include these causal relations and are in the order they have to be observed. This ensures that any node that reads an update also reads all the updates that causally preceded the update making the model causally consistent.

In addition to the causal consistency guarantee, the fetch_await and push_await variants of the fetch and push primitives allow for case by case modifications to the guarantee. When a node invokes the regular fetch, a request is made to the remote node requesting changes from a specific version. If there are no new changes, the fetch returns empty-handed, and the local node process continues. With fetch_await, however, the fetch request blocks until the remote node has progressed from the requested version. In many cases, this is preferable to constantly polling fetch requests as it generates much less network traffic, decreasing update latency. Along similar lines, the push_await primitive blocks the local execution of application code after a push request until an acknowledgment is received for the successful integration of the pushed diffs. The code executed after the push_await can assume that the remote node has integrated the updates into its version graph. This is particularly useful when the node making the push wants to receive updates from a potential conflict resolution immediately following a push, as in some cases, conflict resolution can be costly.

4.6 Formal Specification

We present a formalization of the GoT version control programming model that underlies Spacetime. The formal specification serves to unambiguously describe the concepts and

Table 4.2: Metavariables

Primitive metavariable	Meaning
x	Value
d	Attribute of an object.
t	A concrete type. $t : \{d_1, d_2, \ldots, d_n\}$
o	An object. $o :< t, id(o), z >$
$id(o)$	The unique primary key of the object. $id(o) : z[d_p], d_p \in t$
z	State of an object. $z : \{\forall d \in t, d \mapsto x\}$
v	An unique identifier for a version.

operations, independently of any implementation. We were unable to locate any published work with a formal specification of Git. As such, we believe GoT, which, in its current state, is a simplified version of Git, is a valuable step towards that goal.

Figure 4.14 shows the formal specification of GoT. Table 4.2 summarizes the metavariables used in the specification. The specification is divided into five parts: (1) the definition of dataframe; (2) the interface that the snapshot exposes to the application code; (3) functions of the version graph; (4) the interface that the version graph exposes to the snapshot; and (5) the interface that the dataframe exposes to remote dataframes. Figure 4.1 (page 63) is a good illustration for understanding the different parts of the formal specification, which are explained next. (A word on notation: the arrows in Figure 4.14 denote a change of state in either the snapshot or the version graph).

4.6.1 Dataframe

As discussed before, the core of our model is centered around the component called the dataframe. Each node in GoT gets its own dataframe to manage the state of its objects[5]. As an object heap, the dataframe must satisfy two important constraints: read stability, and deterministic state update. In the heap of single-threaded programming languages, these

[5]For simplicity's sake since there is only one dataframe per node, we use these two words interchangeably throughout.

D1. Dataframe:

$$
\begin{aligned}
\text{(Dataframe) } D &: && < T, G, S > \\
\text{(Types) } T &: && \{t_1, t_2, \ldots, t_n\} \\
\text{(Version Graph) } G &: && < v_h, V, E, P > \\
\text{(Vertices) } V &: && \{v_1, v_2, \ldots, v_n\} \\
\text{(Edges) } E &: && \{e_1, e_2, \ldots, e_n\} \\
\text{(Single edge) } e &: && \delta_{1\to2}, \delta_{2\to3}, \ldots, \delta_{m-1\to m} \\
\text{(Snapshot) } S &: && < v_s, s, \delta > \\
\text{(Objects in Snapshot) } s &: && \{o_1, o_2, \ldots, o_n\} \\
\text{(Changes in Snapshot) } \delta &: && \{id(o) \mapsto \{d \mapsto x\}[new|mod|del]\} \\
\text{(Remote Nodes) } P &: && \{p \mapsto v_p\}
\end{aligned}
$$

Figure 4.13: Formal specification of Dataframe.

constraints are easy to achieve, as the only way to change the state is from the sequential execution of the application code. In parallel and distributed computing, this is harder to achieve because changes can be made from outside the application. To better isolate the effects of external changes, the heap is divided into two components: a snapshot that provides read stability, and a version graph that detects and manages concurrent changes. The snapshot is the local copy of the objects used by the application code in the node. The version graph keeps track of the published history of shared objects. It can receive both changes committed locally by the node, and changes made externally by external nodes. Changes made to the version graph do not affect the snapshot without an explicit request from the application code.

Formally, a dataframe D is defined as a tuple $< T, G, S >$: a list of types T declared in a shared data model, a snapshot S, and a version graph G to track versions of the objects (Figure 4.13).

D2. Snapshot Interface:

$$\text{(New object) } S(v_s, s, \delta) \xrightarrow{S.create(o)} S(v_s, s, \delta \cup \{id(o) \mapsto z\}[new])$$

$$\text{(Delete object) } S(v_s, s, \delta) \xrightarrow{S.delete(o)} S(v_s, s, \delta \cup \{id(o) \mapsto \emptyset\}[del])$$

$$\text{(Write attribute) } S(v_s, s, \delta) \xrightarrow{o.write(d,x)} S(v_s, s, \delta \cup \{id(o) \mapsto \{d \mapsto x\}\}[mod])$$

$$\text{(Read attribute) } o.read(d): \begin{cases} \delta(id(o), d) & (id(o), d) \in \delta \\ s(id(o), d) & (id(o), d) \in s \wedge (id(o), d) \notin \delta \end{cases}$$

D3. Big Step Version Graph Functions:

(Retrieve changes since v)

$$G.get(v) : \{e_r, e_{v+1}, \ldots, e_{h-1}, e_h\}$$

(Receive changes: $v_a \rightarrow v_b$)

$$G(v_h, V, E) \xrightarrow{G.put(v_a, v_b, \{e_a, e_{a+1}, \ldots, e_b\})} \begin{cases} G(v_b, V \cup \{v_{a+1}, \ldots, v_b\}, & \\ \quad E \cup \{e_a, e_{a+1}, \ldots, e_b\}) & v_a = v_h \\ Resolve(G, v_a, v_b, \{e_a, e_{a+1}, \ldots, e_b\}) & v_a \neq v_h \end{cases}$$

(Conflict Resolution)

$$G(v_h, V, E) \xrightarrow{Resolve(G, v_a, v_b, \{e_a, \ldots, e_b\})} \begin{array}{l} G(v_m, V \cup \{v_{a+1}, \ldots, v_{b-1}, v_b, v_m\}, \\ \quad E \cup \{e_a, \ldots, e_b, (\delta_{a \rightarrow h} - \delta_{a \rightarrow b}) \cup \delta_{res}, \\ \quad\quad (\delta_{a \rightarrow b} - \delta_{a \rightarrow h}) \cup \delta_{res}\}) \end{array}$$

D4. Interaction between Snapshot and Version Graph:

$$\text{(Checkout) } S(v_s, s, \delta), G(v_h, V, E) \xrightarrow{D.checkout(G,S)} S(v_h, s \cup G.get(v_s), \delta), G(v_h, V, E)$$

$$\text{(Commit) } S(v_s, s, \delta), G(v_h, V, E) \xrightarrow{D.commit(S,G)} S(v_n, s \cup \delta, \emptyset), G.put(v_s, v_n, \delta)$$

D5. Interaction with Remote Dataframes:

$$\text{(Push) } \begin{array}{l} D(T, G_l, S_l, P), \\ D(T, G_p, S_p, \emptyset) \end{array} \xrightarrow{D.push_to(p \in P)} \begin{array}{r} D(T, G_l, S_l, v_{lh}), \\ D(T, G_p.put(P[p], v_{lh}, G_l.get(P[p])), S_p, \emptyset) \end{array}$$

$$\text{(Pull) } \begin{array}{l} D(T, G_l, S_l, P), \\ D(T, G_p, S_p, \emptyset) \end{array} \xrightarrow{D.pull_from(p \in P)} \begin{array}{r} D(T, G_l.put(P[p], v_{ph}, G_p.get(P[p])), S_l, v_{ph}), \\ D(T, G_p, S_p, \emptyset) \end{array}$$

Figure 4.14: Formal Specification of Global Object Tracker (GoT) operations.

4.6.2 Data Model and Types

All nodes share the same data model that declares a set of types (denoted by T). All objects shared between the nodes must be instances of one of these types. A type, denoted as t is declared with many dimensions d. An object o of type t is represented by the tuple $< t, id(o), z >$ (see Table 4.2). z represents the state table mapping each dimension d to a value x. Each object has an identifier $id(o)$ such that the pair $< t, id(o) >$ is globally unique. Example types, Player, Ship, and Asteroid, used in a Space Race game, are shown and described in Figure 4.3.

4.6.3 Snapshot

The role of the snapshot in the dataframe is to provide the application code with a copy of the shared objects whose updates are under the control of the code. As is typical for an object heap in any language, the application code can read objects, create new objects, delete existing objects, and modify attributes of objects (Figure 4.14 Part D2). All these reads and writes are executed upon the snapshot. All writes made (updates, additions, deletions) are staged in the snapshot (in δ), ready to be bundled as a diff and applied to the version control graph on a commit. It can also request changes (only when requested by the application code using the checkout primitive) from the version graph and apply the diff received to update the local object state. These two primitives are explained in more detail below.

Formally, we define the snapshot as the tuple $< \delta, s, v_s >$. All writes made to the snapshot are staged in δ. These deltas consist of newly added and modified objects as their state deltas (the whole state is a state delta for new objects), and the identifier for all objects that were deleted. s represents the local state of the objects. v_s indicates that the snapshot was last copied from the version graph at version v_s.

4.6.4 Object Version Graph

The version graph G is a Directed Acyclic Graph (DAG) denoted as the tuple $< v_h, V, E, P >$. Each vertex in (V) represents a version of the state of all the objects. It is never instantiated in the version graph and is only labeled by a version identifier v. Each edge in E is the part of the graph that is instantiated and is represented as $e : \delta_{a \to b}, e \in E$. It represents the diff of changes (δ) required to move the state of all the objects from the previous version (v_a) to the next (v_b). The graph terminates at the head version denoted by v_h, which can be considered the latest state version of all the objects. The version graph can receive object changes through diffs either from the local snapshot or from remote dataframes. The version graph can also receive requests for updates from the local snapshot or the remote dataframe. The version graph responds to these requests with diffs representing these updates. Finally, the version graph also maintains a map of the last know states (v_p) of each external dataframe (p) that it pushes changes to $(P : p \mapsto v_p)$. This map allows the version graph to generate the correct diff during a push.

The version graph exposes two functions: retrieve changes, and receive changes (Figure 4.14 Part D3), which are explained next.

Retrieving changes

Changes can be retrieved by specifying a version identifier $v_n \in V$. This version identifier denotes the last version of the state known by the requester. All the edges (diffs) starting from v_n up until the head version v_h are retrieved and returned in order. Applying these diffs, in order, on the state at version v_n would bring the state to version v_h.

Receiving changes

When receiving changes, the version graph receives an ordered set of edges, $\{e_a, e_{a+1} \ldots, e_b\}$, and version identifiers, v_a, v_b. The edges represent all delta changes, that, when applied to a state at version v_a, transform it to the state at version v_b.

When applying these changes, the version graph grows. If it receives change from the head version ($e : \delta_{a \to b}, v_a = v_h$), the changes are not in conflict, and the edges can simply be appended to the existing list of edges in the graph. What this means is that all the changes received were created, having first read the latest change in the graph. V is also updated with all the version identifiers that the graph can reconstruct from the delta changes it has. The new head version changes to v_b.

However, if the version graph receives a change $e : \delta_{a \to b}$ with $v_a \neq v_h$ and $v_a \in V$, i.e. a delta that moves the state from version (v_a) that is not at the head of the graph to a divergent state (v_b), then the changes received are in conflict with all the changes between v_a and v_h already present in the version graph, and must be resolved.

4.6.5 Operational Semantics of Big Step

As discussed, in GoT, the developer can write a custom merge function to resolve cases of conflict. Conflicts are resolved asynchronously and not under the control of the application code except to supply the merge function. The merge function takes in four parameters: an iterator over objects that conflict, and three snapshot states representing the node state at the point of the fork, and at the end of the diverging paths. This merge function provides rich semantics for merge write-write conflicts and maintaining cross-object semantics after resolution. Changes are all recorded in the corresponded dataframes. The delta changes in the dataframe returned by the function are picked up as the resolved delta δ_{res}. Figure 4.11

gives an example of such a merge function. Any number of default merge strategies can be employed by the programmer to automatically merge all conflicts without having to define their own custom function.

With a final resolved state (v_m) known as the result of the three-way merge, the version graph can be updated. GoT employs a variation of Operational transformation [86] (OT) to bring the diverging states to the resolved state. In traditional OT, two functions must be created that, when applied on the divergent states, transforms them into the common resolved state. Different data types would need to be handled differently, making the creation of these functions extremely difficult. In GoT, we do not create different transformation functions. We create different delta changes that are applied to the divergent states using the exact same function and bring them to an identical state. While transformation functions can be hard to generate, generating the delta required to bring one state to another is simpler. We create a delta containing all object dimensions that are in the later state but not present, or different in the former. In the formal model, we represent the delta from the head version (v_h) to the merged version (v_m) as $\delta_{h\to m} = \delta_{a\to b} - \delta_{a\to h} \cup \delta_{res}$. This delta includes all changes present in the conflicting delta ($\delta_{a\to b}$) that are not present in (and thus in conflict with) the master branch ($\delta_{a\to h}$). This difference is represented as $\delta_{a\to b} - \delta_{a\to h}$. Additionally, we add the delta changes obtained from the three-way merge function (δ_{res}). Similarly, the delta from the conflicting version v_b to the merged version v_m is represented as $\delta_{b\to m} = \delta_{a\to h} - \delta_{a\to b} \cup \delta_{res}$. This procedure gives us three new edges that are added to the graph: the conflicting change ($\delta_{a\to b}$) that was received, the change from the current head of the graph to the conflict resolved version ($< \delta_{h\to m}$), and the change from the conflicting version to the conflict resolved version ($\delta_{b\to m}$).

Any cross-object semantic preserving changes made in the custom merge function would be part of the resolved delta (δ_{res}) and, therefore, would be preserved in the final changes.

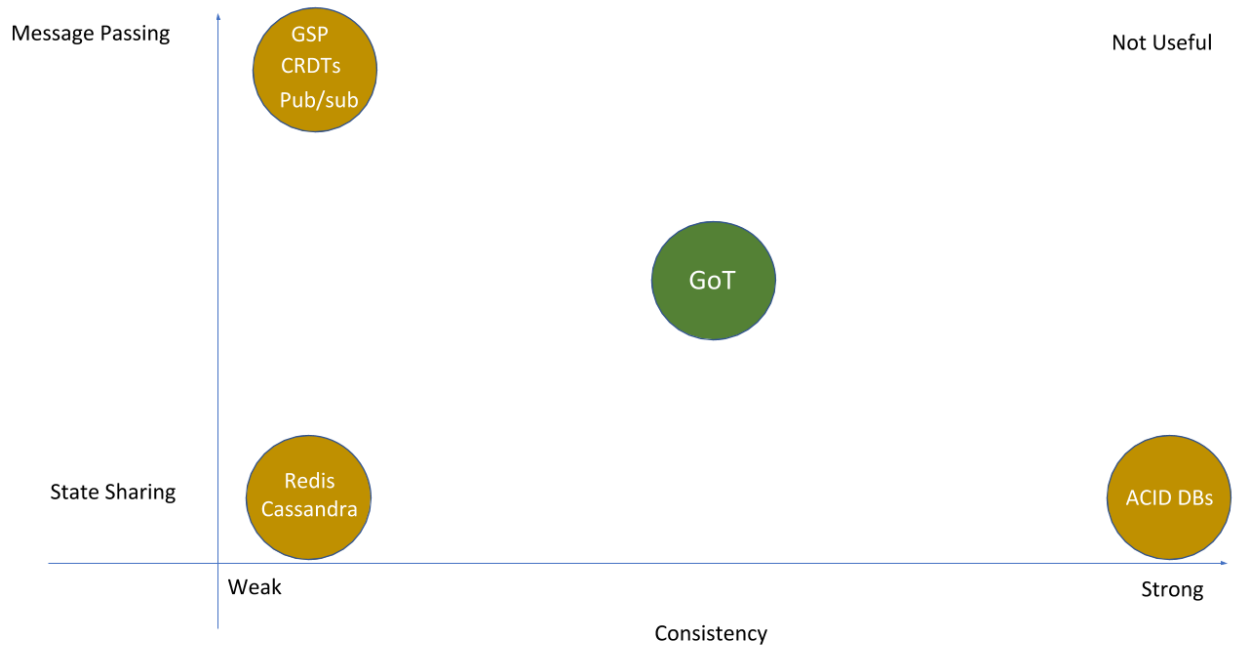


Figure 4.15: Map of Alternative Programming Models when compared to GoT.

4.7 GoT and Alternative Programming Models

With the GoT model explained, we now need to understand what type of programming model, GoT is, when conceptually compared to other distributed computing models that already exist. We chose to compare GoT to models based on two dimensions: the type of the programming model – shared-state, and message-passing, and the consistency model they support – sequential consistency, and eventual consistency. The Figure 4.15, shows a map of many technologies within these two dimensions.

GoT, and subsequently, the implementation Spacetime, as discussed above, implements causal consistency, which, by definition, is stronger than eventual consistency, but weaker than sequential consistency. The consistency model is typically relevant only with shared-state programming models with the strength of consistency defining the interaction between the nodes in the system and the datastore holding the shared-state. Databases such as Redis, Riak, Cassandra are all considered to be implementations of highly available, eventually

consistent stores. On the other end, databases such as MySQL, Oracle DB, PostgreSQL, etc. are considered to be sequentially consistent with the use of strong synchronization tactics such as transactions.

In the other dimension, message-passing programming models, typically do not implement sequential consistency as the computing would be slowed down significantly. If before making an update at a node, the node has to obtain the latest update written anywhere in the system, all concurrent updates have to wait until this one node writes. Since these nodes are not writing to one location or shared-state, coordination mechanisms have to be placed at each node to ensure this sequence. Such restrictions can realistically only be placed onto a shared-state data store, but not the states at each node in a message-passing system. Techniques that assign a global order to concurrent updates do exist (GSP [46]). However, these systems either require a node that orders the updates (server) or quorum style ordering of updates. The first implies that there is a shared-state (the state of the server). The second is slow unless applied to eventual consistency.

GoT is a hybrid programming model that shares traits with message-passing and shared-state programming models, but not wholly one or the other.

Like shared-state models, and unlike message-passing models, in GoT, there is a notion of a common set of objects or variables synchronized between different nodes – the dataframe. The state of the execution can be easily observed by looking at this shared location. This observation is not possible in message-passing systems as there is no common set of objects or variables. Each node can have a different set of variables and data. Each component keeps only the slice of the state that it needs and in the form that it can use best.

On the other hand, unlike shared-state, but similar to message-passing, the entire state of a GoT application is not with one GoT node. The state is, instead, distributed or replicated at each node. These replicas can have partial updates that have not yet been synchronized with

other nodes in the system. Even when a shared-state system is using a highly available system such as a database cluster, to all the clients, which are the nodes computing the changes to the state of the distributed application, the database cluster is seen as one component that updates are sent to, and state is read from. It, therefore, behaves as a shared-state model. The internal replication and distribution of the updates within the database cluster is a separate, shared-space application that typically uses message-passing models to share state updates.

Another similarity between GoT and message-passing models is that when nodes synchronize, they share information between each other through push and pull communication, instead of synchronizing directly over a common data store like in shared-state. Finally, in GoT, each node owns the data in the local dataframe and is responsible for keeping them correct. This trait is again shared with message-passing models, where each node, or actor is in charge of their state and delta. They receive messages and enact these messages over their state to keep them correct.

Chapter 5

Spacetime: Implementation of GoT

As defined and explained in Chapter 2, shared-space applications need consistency at the lowest latency possible, a notion we refer to as update latency. *Update latency* is the time taken for an update to the state in one component in a distributed system to be read in another component. Update latency is different from the notion of both consistency and traditional latency.

In distributed systems, consistency is defined as the communication guarantee that a node in a system will always read the latest write. In stronger consistency models like sequential consistency, this guarantee is maintained rigorously at the cost of having to wait for a write to complete. In weaker consistency models like eventual consistency, this guarantee is relaxed, and nodes can read potentially stale data in exchange for a quick response from the remote source. While consistency guarantees that the values read are at their latest state when read, it does not include a notion for the time taken to read that state. It does not include the concept of time.

On the other hand, traditional latency is the time taken for a remote node to respond to a request. This concept only carries the notion of the time for response and provides no

information or guarantees on the response's content. A node can respond with obsolete data multiple times, quickly and still have low latency.

While distributed systems typically either focus on latency or consistency, there are applications such as the shared-space applications we have been discussing, that require optimization to both.

The concept of update latency is quite similar to the concept of probabilistically bounded staleness [6] (PBS) proposed by Bailis et al. Given a response to a remote request, PBS defines the probability of getting a stale response. In contrast to update latency, PBS first assumes weak consistency and then provides a probability of getting the latest write on each response. For strongly consistent systems, the probability would be high. It gives the software developers a sense of how inconsistent the state at each node is. Update latency is the other side of the coin. It first assumes consistency and then measures the time taken to achieve this consistency. It gives the software developers a sense of the time taken for an update to propagate. While they represent similar notions, we feel that for shared-space applications, update latency provides a clearer picture. For example, in an online multiplayer game, showing the players the median update latency for the updates they make is more relevant than providing them with the probability of receiving new updates. The usefulness of PBS is limited to eventually consistent systems.

5.1 Components of Update Latency

To understand how GoT enables low update latency we need to understand its components. Figure 5.1 shows the space and time diagram of three nodes of a distributed system A, B and C. Real time is represented on the horizontal axis and the state space of different nodes is represented in the vertical axis. The three nodes are at initial states $A1$, $B1$, and $C1$

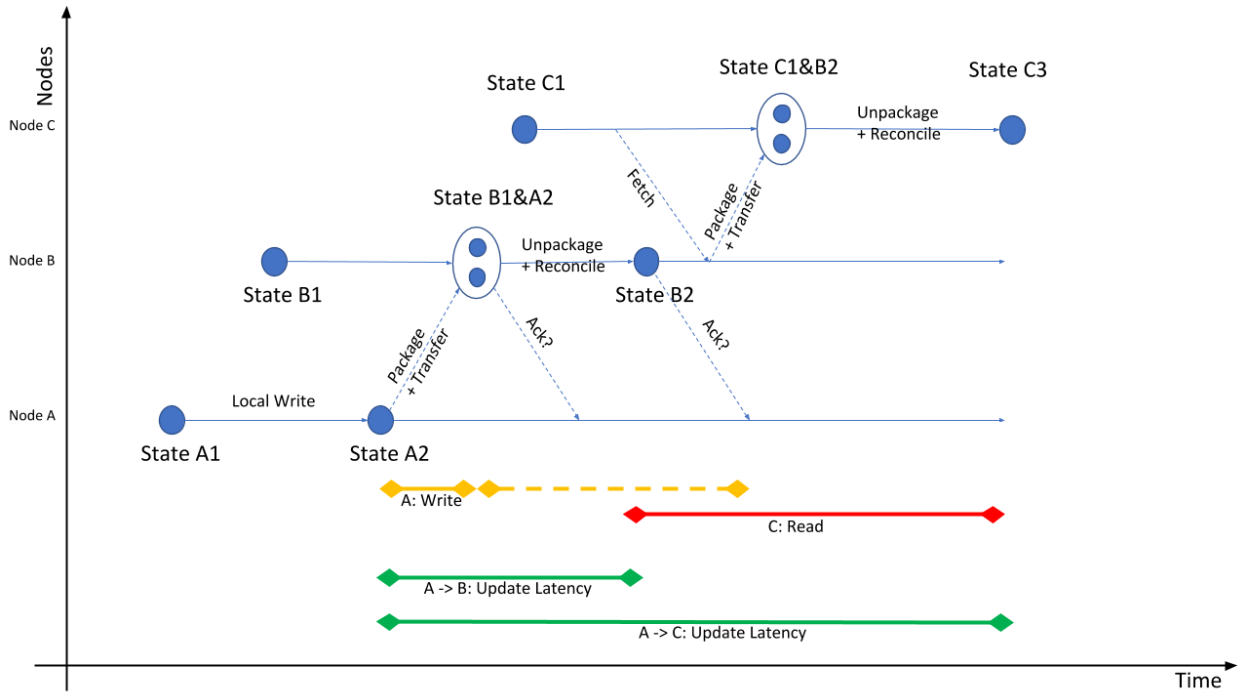


Figure 5.1: Components of update latency.

respectively.

Node A makes a commit and moves the local state to state $A2$. This process takes some time, and the update is only available locally. This update has to now propagate to the other two nodes. Update latency is calculated from this point in time. Node A packages the update and transfers it to node B over the network. This propagation is not instantaneous and therefore, must travel through both space and time to reach node B. This time taken is the write latency of node A to node B. Although node A cannot record this latency.

At this moment, the state at node B does not include the information that has just been introduced from node A. The update has to be unpackaged and reconciled into the local state. Reconciliation might include conflict resolution if the updates received are incompatible with the local state. The reconciled state is state $B2$. The update latency between A and B for the update made by A is the time between the creation of state $A2$ and the creation of state $B2$, as shown by the green line.

Let us now consider how fetch requests influence update latency. Node C makes a fetch request to node B. The node receives the request, and a response comprising of state $B2$, which includes $A2$, is packaged and sent back. When the node receives it, unpackages and reconciles the update, the local state is updated to state $C3$. The update latency for update $A2$ between node A and C is the time between the creation of $A2$ and $C3$.

5.2 Optimizing for Update Latency

Spacetime is an implementation of the GoT programming model, developed in python. It was designed for shared-space distributed applications that require low update latency such as multiplayer games, multi-agent simulations, etc. Using the GoT programming model, Spacetime optimizes each stage or gives the programmer choices in the process of transmitting an update, all in a bid to reduce update latency.

5.2.1 Preparation of the Updates

As an optimization, delta updates in GoT are bundled as and when tracked objects are made, updated, or deleted. They are then bundled into a single update when the commit command is invoked. The choice for the programmer is in the granularity of the bundle.

For example, suppose a thousand cars have to be added by an initialize node, to a simulation, for a taxi simulation to use. The initialize node can prepare one update with thousand objects making all thousand objects available at once, but at the same time taking longer to package the update. Alternately, it can publish the objects one at a time, making the updates numerous but smaller and letting the taxi simulation get some of the cars while the others are being added. Each update, however, is staged while it is being created to avoid the computation cost of creating a diff from full versions during packaging time.

5.2.2 Transportation of Updates

Since the updates are always transferred as deltas, and not as full objects, the updates only carry the necessary information to the remote nodes. The amount of information is dependent on the changes made to the version graph at the time of the transportation.

Additionally, to optimize the delivery of updates in pull communication, GoT gives the choice to the developers to use two variants of the fetch – fetch and fetch_await. In fetch, the node attempts to pull any update that exists from the remote node, even if there are no updates. This method allows for low latency, but potentially stale responses. In fetch_await, the node makes a pull request that the remote node serves only when there is an update to be sent across. This method allows for stricter consistency at the expense of having to wait for an update. The fetch_await blocks the application code in the node that invokes it. Using fetch_await instead of fetch allows nodes to receive updates that they know will be available, as soon as they are present in the remote version graph. Since pull communication in GoT also uses delta updates, redundant information is not transferred.

For example, a programmer should code the player bot in Spacerace to proceed with the game, only when the player state has been set to ready. If the programmer is only using fetch and not fetch_await, multiple redundant fetch requests will be made, leading to higher traffic load at the physics server. With fetch_await, only updates with new information are obtained, and the player bot can only synchronize over new information until the game can begin. Another example in the Spacerace, the visualizer nodes use pull instead of pull_await as it is not critical for them to receive the latest update, and the visualizer can continue estimating new positions even with a slightly stale update.

Finally, as another optimization from GoT, in Spacetime, the deltas transported are compressed into a single delta and edge. This optimization makes sense in the big-step Spacetime-world as the topology restrictions ensure that the node receiving the updates will never require

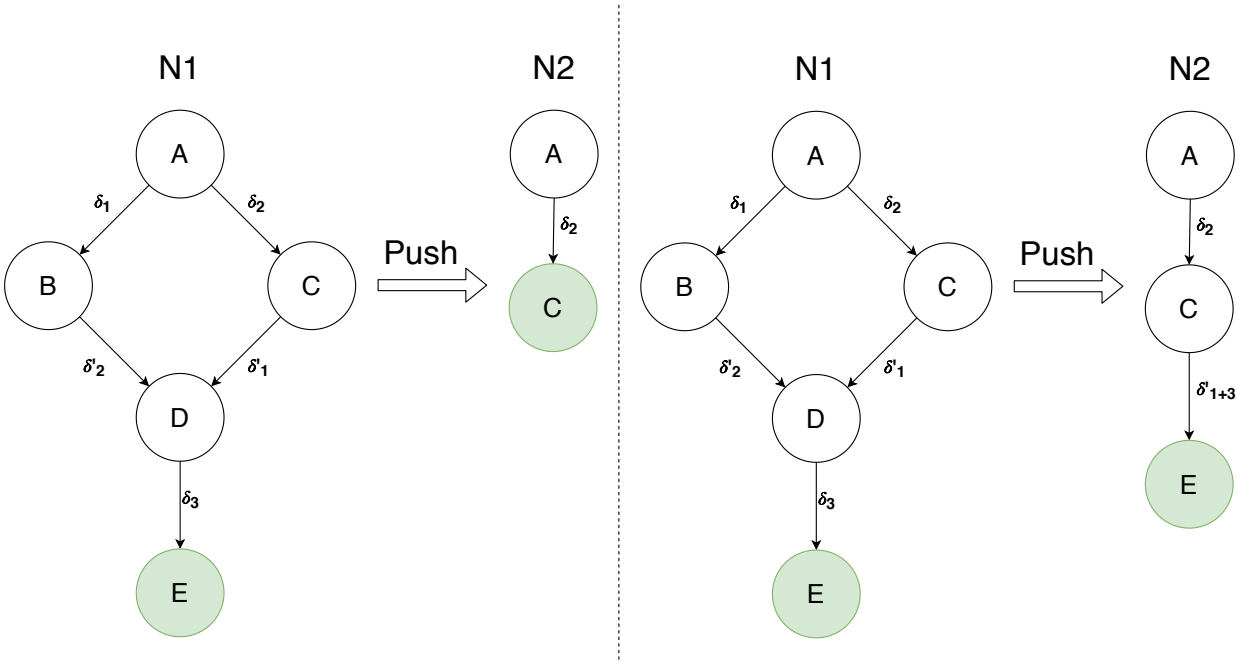


Figure 5.2: Pushing updates $C \rightarrow D \rightarrow E$ from N1 to N2.

the intermediate updates in the delta. This is simply because the onus of communication is one way. Nodes do not push to each other or fetch from each other. One node receives a push or fetch request, and the other node makes the request. This means that it is only one node that dictates from which version to send updates and from which version to receive the updates. Since nodes cannot also roll back in time and they are always expected to synchronize to the HEAD version, they can never visit the intermediate states.

Let us consider the following example shown in Figure 5.2. There are two nodes $N1$ and $N2$ that are synchronizing. $N1$ pushes updates to $N2$. $N2$ never pushes updates back to or fetches updates from $N1$. The updates δ_1 and δ_2 are concurrent in $N1$, with a delta transformation states B, C, and D present in the graph. In addition, there is an extra update from $D \rightarrow E$ with delta δ_3. The versions marked in green are the HEAD versions. When $N1$ pushes to $N2$, the node $N2$ identifies its last state synchronized with $N1$ as version C. $N2$ needs to receive updates to bring the state of $N2$ to state E. There are two updates along that way, $\delta_1^{'}$ and δ_3. The edges $A \rightarrow B \rightarrow D$ are not relevant to $N2$ to reach the HEAD

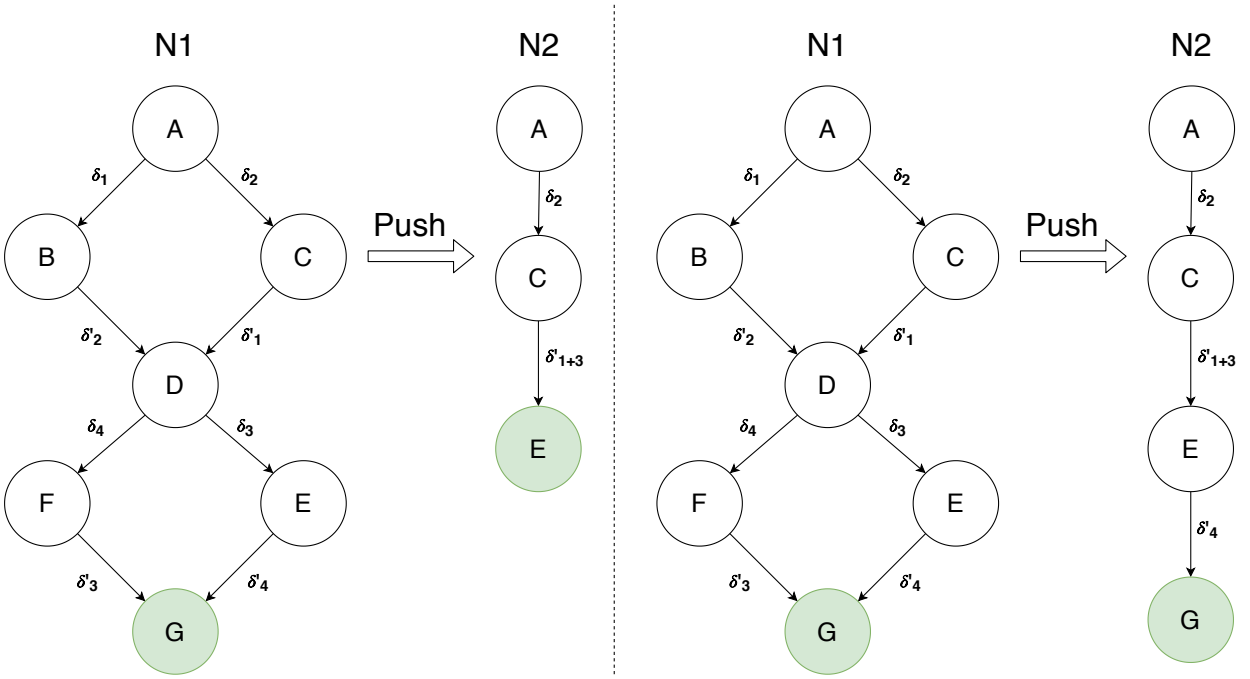


Figure 5.3: Pushing updates $E \rightarrow G$ from N1 to N2.

version E. Therefore, this does not need to be synchronized. The edges $C \rightarrow D \rightarrow E$ are combined together into one update $C \rightarrow E$ with the combined delta δ'_{1+3}. The state D is completely an intermediate state and is not required by $N2$ if certain topological constraints are met.

First, there must be no cycles in the topology. Nodes $N1$, $N2$, and $N3$ cannot be set up in a way that $N1$ pushes/fetches from $N2$ which in turn pushes/fetches from $N3$ which pushes/fetches from $N1$ again. Second, each node can have only one parent. These two rules ensure that there is only one authoritative replica for each node and that there are no cycles in the authority chain.

With these rules, we can answer two questions: (1) why the versions B, and D, although not known to $N2$, will never be needed by $N2$ and (2) why $N2$ will never need the edges $A \rightarrow B \rightarrow D$.

The versions B, and D are present in $N1$, but not in $N2$. Since $N1$ is pushing to $N2$, $N2$

cannot push back to $N1$ (banned by the rules over the topology). This essentially means the $N2$ is the authoritative replica of $N1$. The versions B and D could only have been created in two ways: $N1$ could have committed the update itself, or $N1$ received an update from a child node for whom it is the authoritative replica. Since there cannot be any loops in the authoritative chain, any child replica of $N1$ is not a direct child replica of $N2$. This means that any update created by the children of $N1$ cannot reach $N2$ without passing through $N1$. Since $N1$ is in control of updates being sent to $N2$, it knows the updates that are with $N2$ and those that are not. It knows the last synchronized version of $N2$. Therefore, $N1$ can ignore versions B and D and directly send the updates from $C \rightarrow E$. Any further concurrent updates from B or D are merged into a version that happened-after E. That update will be then be received by $N2$ as an update from E on subsequent synchronizations. This process is shown in Figure 5.3. A concurrent update from $D \rightarrow F$ is seen in $N1$, the merge of which is version G. When $N1$ synchronizes with $N2$, it sends the missing concurrent update $\delta_4^{'}$ with the edge $E \rightarrow G$. Thus we can see that this answers the question (1).

To answer the question (2), $N2$ requires the edges $A \rightarrow B \rightarrow D$ only if there are delta updates in there that are missing from $N2$, or if there is a possibility that $N2$ might receive an update from version B. The latter was shown not to be true above, so the only reason to transfer $A \rightarrow B \rightarrow D$ is if there are missing updates. The updates in $A \rightarrow B \rightarrow D$ are δ_1 and δ_2. We can see that $N2$ already has the update δ_2, and has yet to receive δ_1. The edges $C \rightarrow D \rightarrow E$, however, contain the update δ_1. So from the standpoint of correctness, the push update it is receiving in Figure 5.2 will contain the required deltas, answering the question (2).

This method of merging updates significantly reduces both the amount of data being transferred, as redundant intermediate states are discarded, and reduces the amount of processing required to merge the update that is being received. The latter is because on every push and pull, instead of synchronizing multiple edges, only one compressed edge is ever sent.

5.2.3 Conflict resolution

Another phase in the path of update latency is conflict resolution. Every update has to be resolved against concurrent updates, if any. Many architectures enforce that this resolution takes place before the conflict can be observed in a bid to avoid conflicts in the first place. For example, quorum based techniques, global sequence protocol [46], CRDTs [100] all define rules for concurrent updates in such a way that conflicts are eliminated systemically (last-write-wins, etc.) or their application is redundant (CRDTs).

GoT, instead uses the three-way merges of the concurrent changes to accept the changes when the updates are observed. When a custom three-way merge is used, the complexity of the operation is determined by the complexity of the custom function in finding the resolved state. Optimizations to avoid iterating over objects present in independent updates can easily be made.

When there are no concurrent changes, updates are fast, as it is a simple operation of extending the graph. Since the version graph is maintained edge first, the graph can be extended without actually observing the state changes in the delta update. No version state has to be calculated.

In the case of conflict, the deltas must be read at the granularity chosen by the programmer. A blanket rule such as “accept yours” or “accept theirs”, as seen in Git, is extremely efficient as the deltas do not have to be processed. A custom merge rule, however, can be slow if the algorithm to determine the merged result is slow.

CRDTs were explicitly designed to avoid this problem. By knowing that the updates compose correctly, irrespective of the order, the contents of the updates do not need to be processed. However, the same rules can be applied to Spacetime. The delta transformation can just use the existing deltas if the deltas are known to be commutative and associative.

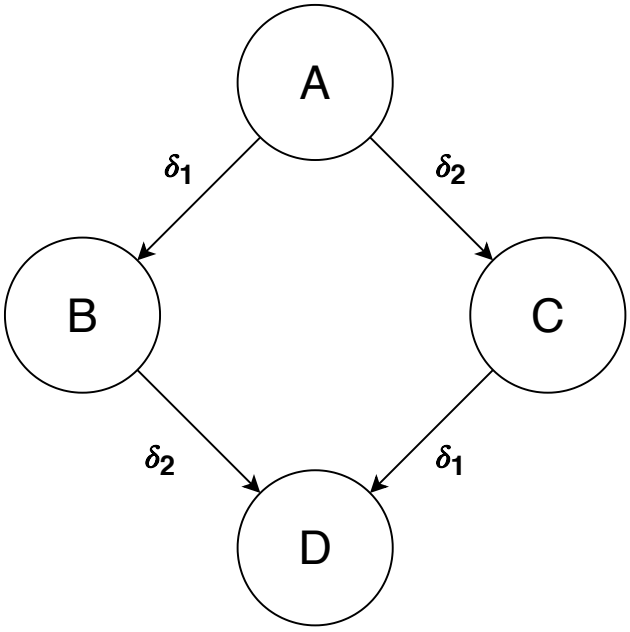


Figure 5.4: Delta transformation with commutative merges δ_1 and δ_2

For example, in Figure 5.4, if deltas in concurrent updates δ_1 and δ_2 are associative and commutative, then applying them in any order reaches the same state. Therefore, to reach state D from B, we just need to apply δ_2, and to reach D from C, we need to apply δ_1. These deltas then do not have to be computed.

The presence of the three-way merge, and the choice to use them at any granularity, including an approach similar to CRDTs, allows optimizations that keep conflict resolution fast. Spacetime refrains from observing and processing the state in the updates unless necessary for resolution, thus improving the speed of conflict resolution and ultimately update latency.

Chapter 6

Challenge 1: Peer to Peer Networks

As discussed in the previous chapter, and for more reasons explained in this chapter, there is a fundamental problem in existing version control systems that makes their use quite limited in an important category of network topologies, namely in peer to peer applications. Version control systems often rely on 3-way merges, as explained in the previous chapter, to resolve write-write conflicts which require that two conflicting versions have a least common ancestor (LCA) version to compare them to. In certain situations, (known as the "criss-cross merge" in git[1]), a single LCA cannot be computed. In those situations, existing VCSs resort to one-off solutions that are not elegant or that burden the user with large diffs that are irrelevant. While these situations are rare corner cases in the use of version control of files, they will be common if GoT is used in peer to peer applications. For that reason, this fundamental problem should be addressed head-on, rather than being treated as a rare corner case.

In this chapter, we illustrate the criss-cross problem with version control state replication in a peer to peer setting, with the example of the distributed counter described in the Chapter 4, in the simplest peer to peer setup: two nodes, N_1 and N_2, that both push updates to each other. We then explain the inadequacies in big-step merge to deal with the criss-cross merge,

[1] `http://www.gelato.unsw.edu.au/archives/git/0504/2279.html`

and show how modifications to the merge function can help mitigate that problem in peer 2 peer. With a simple formal proof we prove that the new merge function, called small-step three way merge can eliminate the criss-cross merge problem in peer to peer GoT applications.

6.1 Criss Cross Merge Scenario

Figure 6.1a shows the initial states of the version graphs at N_1 and N_2. The initial state in both N_1 and N_2 is A ($v = 0$). N_1 has further updated the state to B ($v = 1$) via update u_1, while N_2 has made a concurrent update bringing the state to C ($v = 2$) via update u_2. All updates made by N_1 locally are colored in blue while that of N_2 are colored in green. A which is the common root of the version graph is left uncolored. To illustrate the problem, let the following events take place in order.

Event 1: The two nodes communicate updates with a push to each other. N_1 sends $A \rightarrow B$ to N_2, and N_2 sends $A \rightarrow C$ to N_1.

Event 2: While the push is still in the network, N_1 and N_2 both make concurrent updates u_3, bring HEAD to B, and u_4, bringing HEAD to D respectively. Figure 6.1b shows the version graph at N_1 and N_2 after this update.

Event 3: The push that was previously sent in Event 1 arrives at each of the nodes. A conflict is detected in both nodes and resolved with the creation of merge nodes using the big-step 3-way merge. As we can see in Figure 6.1c, N_1 resolves versions conflicting versions D, C, and LCA A to get merge version M_1.

The update associated with the edge $D \rightarrow M_1$ is equivalent to u_2 and the update associated with $C \rightarrow M_1$ is the combined effect of two updates u_1, and u_3 represented as $u_1 + u_3$. The combined effect is a new delta (or command) that is equivalent to the two individual delta

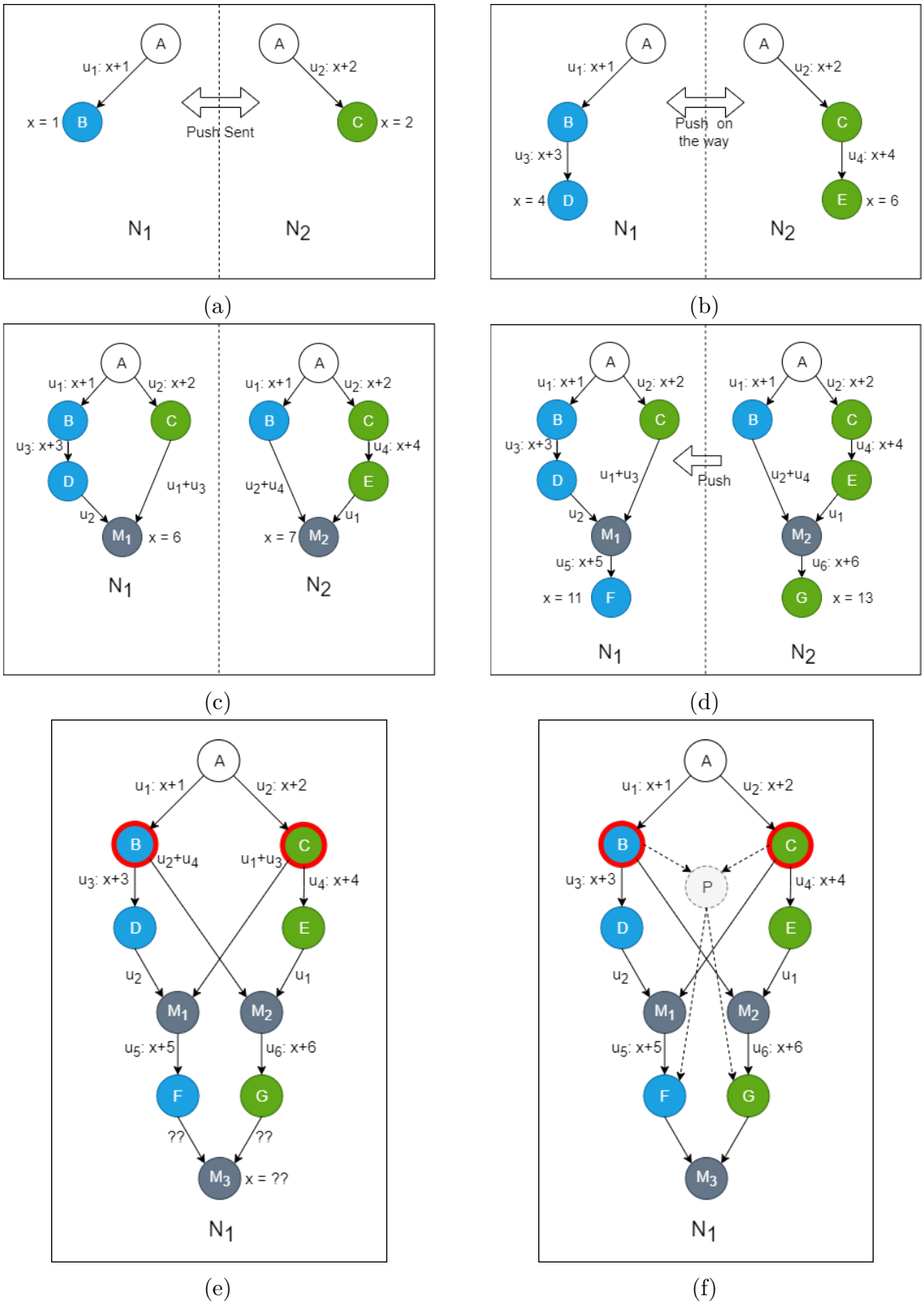


Figure 6.1: Broken Version Control in Peer to Peer Networks.

updates. In the case of the counter, $u_1 + u_3 = v + 4$. Similarly, N_1 resolves versions conflicting versions E, B, and LCA A to get merge version M_2. The merge versions are colored grey to better identify them. The value of the counter v at M_1 and M_2 is six $(u_1 + u_2 + u_3)$ and seven $(u_1 + u_2 + u_4)$ respectively. Both values are correct.

Event 4: Let each of the nodes then make another set of concurrent updates u_5 and u_6, bringing the HEAD of N_1 and N_2 to F and G respectively, as seen in Figure 6.1d.

Event 5: Node N_2 then pushes updates to N_1 which is received. What we see, in Figure 6.1e, is the result of that push in node N_1. N_1 first detects a conflict as the version graph has two HEAD versions, F, and G. In order to invoke the 3-way merge and resolve this conflict, a LCA of F and G is needed. There are, however, two such ancestors: B and C (marked with a red circle). This is where the big-step 3-way merge fails.

6.2 Failure of Big-Step Merge

The solution is not to pick any one of the ancestors. For example, if N_1 chooses B, then the update $F \to M_3$, which is equivalent to the updates $B \to M_2 \to G$, is set to $(u_2 + u_4) + u_6 = v + 12$ and the update $G \to M_3$, which is equivalent to the updates $B \to D \to M_1$ which is $u_3 + u_2 + u_5 = v + 10$. This would set the value of v at M_3 to be 23. If we instead choose, C as the least common ancestor for the merge, the update $F \to M_3$ is equivalent to the updates $C \to E \to M_2$ which is $u_4 + u_1 + u_6 = v + 11$ and the update $G \to M_3$ is equivalent to the updates $C \to M_1 \to F$ which is $(u_1 + u_3) + u_5 = v + 9$. The value of v at M_3 would then be 22. Both values are wrong as the real set of updates to M_3 is $u_1 + u_2 + u_3 + u_4 + u_5 + u_6 = 21$.

This problem exists in file based version control systems like Git. Git employs a lazy, recursive 3-way merge strategy: when it encounters the criss-cross merge, it creates a pseudo version P, as seen in Figure 6.1f which is the automatic merge of B and C, and acts as the least

common ancestors for F and G. If B, and C have no single common LCA, the process is repeated recursively. This recovery process, by which the causality of smaller state updates is being calculated, can be extremely costly. In cases where criss-cross merges are frequent, such as in peer to peer applications, this strategy becomes unfeasible.

6.3 The Root of the Problem

If we were to look at the path $A \rightarrow M_3$, via B, we can see that it includes the update u_2 twice (the same path via C includes u_1 twice). The updates $F \rightarrow M_3$, calculated with B as the ancestor, is the combined effect of $B \rightarrow M_2$ and $M_2 \rightarrow G$. $B \rightarrow M_2$ itself is the combined effect of $A \rightarrow C$ (u_2) and $C \rightarrow E$ (u_4). But the update u_2 is already included in the path $B \rightarrow F$ at $D \rightarrow M_1$. In order to detect that u_2 is being applied twice, it is necessary to include that information in the edge $B \rightarrow M_2$. This means that every edge must carry information of the constituent updates. This is unfeasible. Take for example, if the next update N_1 were to receive would be a concurrent update from F. The resolve of that conflict would need to include all the updates that are included with $F \rightarrow M_3$, and so on. As updates progress, each edge generated via conflict resolution would need to store more and more information.

Since the crux of the problem is the loss of information when updates are combined together during conflict resolution, our solution is to rework conflict resolution to ensure that each edge only carries the information of one update. This solution is called **small-step merge** and is discussed next.

6.4 Small Step Merge: Consistent P2P Version Control

To demonstrate the small-step 3-way merge, we use the same example that was used in the precious section. Events 1 and 2 remain the same and we pick up at the start of Event 3. The version graph at N_1 and N_2 at the start of the Event 3, is depicted in Figure 6.1b, and the two pushes from Event 2 are still in the network and have not reached the two nodes.

Figure 6.2a shows the state of the version graphs at N_1 and N_2 after the push is received by each, but before the resolution occurs. Looking at node N_1, big-step merge rules for conflict resolution performs a 3-way merge between versions C, D, and least common ancestor A to get a merged version. However, we have already seen that the update from C to this merged version becomes the combined update $u_1 + u_3$, compressing information irretrievably (Figure 6.1c). In our algorithm, however, we first perform a 3-way merge between C, B and the least common ancestor A (Figure 6.2b), to obtain the intermediate merge node M_1 that happened-after B, and happened after C, but has no relation with D. The only information that is known is that both M_1 and D happened-after B and are concurrent updates to each other. The next step within the same conflict resolution, is another 3-way merge between D, M_1, and least common ancestor B, to obtain M_2, which becomes the HEAD version for N_1 (Figure 6.2c). The same process is also applied to N_2 as can be seen in Figure 6.2c. In this process we can observe three things.

- First, every edge is only ever associated with one update. This solves one part of the problem identified in the previous section. It becomes trivial to detect if the same update is being included multiple times in each resolution.
- Second, any path from A to M_2 in N_1 will always include one u_1, one u_2, and one u_3, but in different orderings. For example, $A \to B \to D \to M_2$ applies the updates in the order:

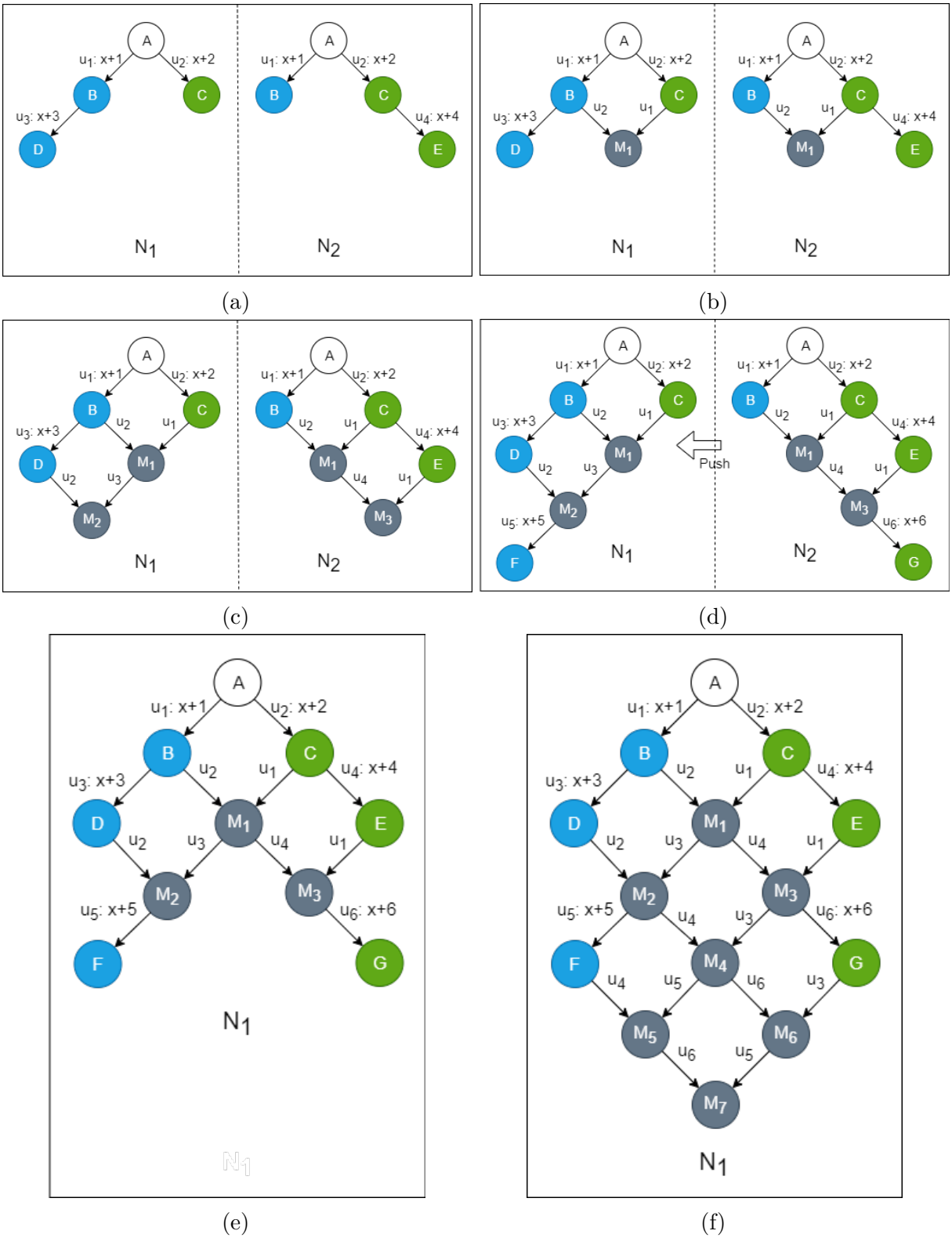


Figure 6.2: Revised Version Control in Peer to Peer Networks.

u_1, u_3, and u_2. This path explores u_2 occurring after u_1 and u_3. $A \rightarrow B \rightarrow M_1 \rightarrow M_2$ applies the updates in the order: u_1, u_2, and u_3 and explores u_2 occurring in between u_1 and u_3. $A \rightarrow C \rightarrow M_1 \rightarrow M_2$ applies the updates in the order: u_2, u_1, and u_3 and explores u_2 occurring before u_1 and u_3. Since u_2 is an update that occurred concurrently with u_1 and u_3, it can occur before or after u_1 and u_3. However, u_3 will always occur after u_1. That partial order of updates is maintained. It also follows that the version graph, as shown here, preserves the partial ordering of updates and when visualized as shown, is the Hasse diagram [108] for the partial order of updates.

- A final observation is that both nodes N_1 and N_2 merge the nodes A, B, and C (Figure 6.2b) into M_1. This requires that the 3-way merge function at both N_1 and N_2 be (1) identical, to consistently produce the same result, and (2) commutative, to ensure that the order of concurrent updates from the same version do not matter. The second condition is important because the order of the observation of versions B and C is different in N_1 and N_2. N_1 sees B first, followed by C. N_2 sees them in the opposite order. The notion of "yours" and "theirs", using the terms from Git, is reversed for the 3-way merge functions at both nodes, but they must still produce the same merged state. Therefore, merge functions need to be commutative.

During Event 4, in Figure 6.2d, N_1 and N_2 both make concurrent updates u_5 and u_6 respectively. In Event 5, N_2 pushes updates to N_1.

6.4.1 Success of Small-Step Merge

Figure 6.2e shows the state of the version graph at N_1 after the push from N_2 during Event 5, but before the merge algorithm resolves the detected conflict. In this version graph, concurrent versions F, and G have a least common ancestor M_1, however, our method of conflict resolution, no longer performs a 3-way merge between these three versions. Instead,

first M_2, M_3, and ancestor M_1 are merged to produce M_4. Then M_4, F, and ancestor M_2 are merged to produce M_5; M_4, G, and ancestor M_3 are merged to produce M_6. Finally, the version graph converges to a single HEAD version with the 3-way merge between M_5, M_6, and ancestor M_4, giving us merged node M_7 which becomes the HEAD version of N_1

We can again see that every edge is only associated with one update (a property that would have been broken if F, G, and M_1 were merged via a 3-way merge); any path from A to M_7 includes every update and only once, making the state at M_7 be correct with the value of the counter $v = 21$. The Hasse diagram of partially ordered updates includes the relations between the additional updates that were pushed by N_2 (u_4 and u_6). u_5, for example, only occurs after u_1, u_2, and u_3 while being concurrent to u_4, and u_6. Similarly, u_6 occurs after u_1, u_2, and u_4, while being concurrent to u_3 and u_5.

The crux of the algorithm is the merge strategy that is followed. Instead of merging conflicting HEAD versions (versions that have no other versions that happened-after) with the least common ancestor (if there is only one), a version is only ever merged with its immediate siblings. To understand this constraint, let us once again look at Figure 6.2a. The newly added conflicting update is $A \rightarrow C$. The immediate sibling of C is B. D is not an immediate sibling of C. We merge B, C, and A to obtain M_1. This newly created M_1, which is in conflict with D, is then merged with D as D is now the immediate sibling of M_1. We will prove, formally, in the next section that two versions that are immediate sibling will have one and only one common parent. Intuitively, the only nodes that can have multiple parents are the merge nodes (nodes in grey) since each real update happened-after the state of a single version state. If two versions are immediate siblings of multiple parents, both versions must have been created by the 3-way merge of these parents. If the merge functions are identical and commutative in every node, the two versions can be considered to be the same, and conflated as one version, resolving our situation. In Figure 6.2d, if the merge of B C and A was not identical and commutative, different versions would have been created as a result

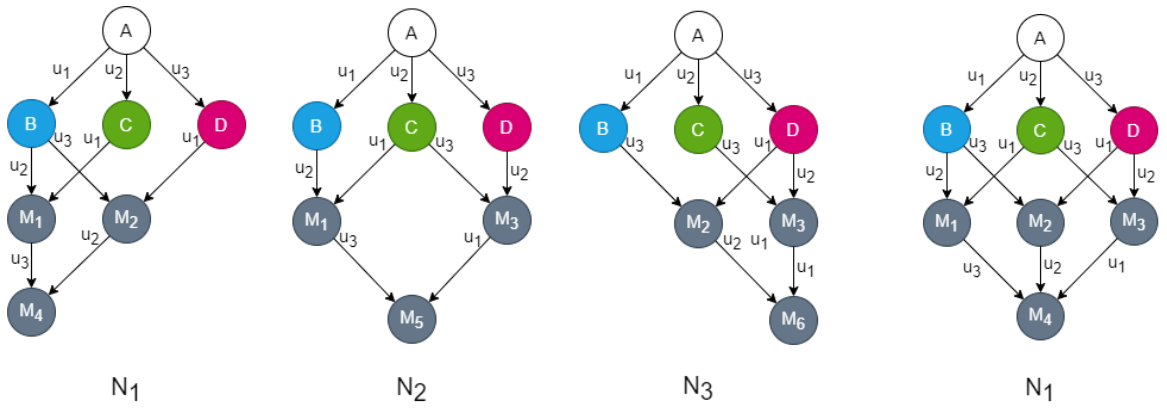


Figure 6.3: Three concurrent updates with Revised Version Control.

of the merge. When N_2 pushes to N_1 in Figure 6.2e, in the version graph at N_1, B and C would have two different merge versions as children. These merge versions would be siblings that share more than one parent, breaking our requirement of a single least common ancestor for each 3-way merge. If the merge functions returned identical state, the two versions can be conflated.

6.4.2 Small-Step Merge with Three Peers

We have seen how merge scenarios work in peer to peer systems with two peers. Figure 6.3a gives an example of a three peer system. Let N_1, N_2, and N_3 be three peers in the system, each making the update u_1, u_2, and u_3, respectively. These updates are then shared. We stated that a version is only ever merged with its immediate siblings. To be more precise in the constraint, a version is only ever merged with one immediate sibling from each parent. At Node N_1, versions B, C, and D are siblings of each other. There is no guarantee on which node between C and D would arrive at node N_1. Let us assume that it was C. C and B are siblings, and are merged to M_1. When D then arrives, D has two siblings with parent A. Either of those two siblings can be used to generate the next merge. In Node N_1, it was B that was chosen. M_1 and M_2 are then merged to M_4 using B as the least common ancestor.

Different nodes can merge the same three updates in different ways. In the Figure, the three nodes show all three combinations possible. Let us then assume that N_2 pushes changes to N_1. N_1 now learns of a new merge node M_5 that happened after M_1 using the same update u_3. Both M_4 and M_5 are versions created from version M_1 using update u_3 and therefore, must be equivalent. The graph updated is shown in Figure 6.3b.

This merge node M_4 from Figure 6.3b with three parents can be considered to be equivalent to the result of a four-way merge between B, C, D and the least common ancestor A. Since only 3-way merges are defined by the version control system, multiple 3-way merges are required to obtain the effect of a four-way merge. Implementing four-way merges instead, or N-way merges instead is not practical as the size of N is determined by the number of nodes in the distributed system and would additionally involve waiting for all N updates to arrive before resolving conflicts.

It is important to note that $M_1 = M_4$ will only be true if the 3-way merge function at each node is also Associative. It is interesting that the version graph implementing the rules that we have detailed can detect during runtime when the merge function is not identical, commutative or associative – something that eventual consistent approaches like CRDTs cannot detect. In CRDTs, only when the updates have stopped, and applied by every node, can states be compared to detect inconsistencies. In our approach, when differing versions are immediate siblings and have more than one common parent but encode different states, then non commutative or non associative merge function can be flagged.

6.5 Conclusion: Constraints and Properties

We summarize the constraints to the version graph, that were discussed in our small-step 3-way merge approach, as follows:

- **C1:** The version graph at every node in the system must originate from the same root version.
- **C2:** The merge function at each node must be: identical, commutative, and associative.
- **C3:** If a version v has siblings from a parent p, there must exist a merge version between v and another child of p.

With these three constraints, we state that the version graph has the following two important properties:

- **Convergence:** After a write operation to a version graph at a node, satisfying the given constraints, the version graph will have only one HEAD version. Convergence is required, as each node in the version graph should have only one latest state. Multiple HEAD versions implies there are multiple possible versions of the replicated state, and that would be inconsistent.
- **Correctness:** After a write operation to a version graph at a node, satisfying the given constraints, every path from any version to any other reachable version always includes the same set of updates, and creates the same state. In a distributed system that is weakly consistent (in the case of version control replication, causally consistent), updates can reach modes in different orders. The goal is to ensure that the same state is arrived at irrespective of the order in which concurrent updates are applied. In the version graph, which has been shown to be a partial ordering of updates forming a semi-join lattice, multiple paths from one version to another reachable version is interpreted as the latter happening causally after the former, with concurrent updates along the way. The different paths define the possible orderings of updates. Since the goal is state consistency after all updates are applied, in any order, the different paths must lead to the same state.

6.6 Formal Proof of Small-Step 3-way Merge

In this section, we prove that by setting the constraints as defined in the previous section, we achieve a graph that is convergent and correct. To do this, we use the formal model of the version graph defined in Chapter 4 Section 4.6.

6.6.1 Additional Definitions

To recap, the version graph at each node is defined as a directed acyclic graph, $G = (V, E)$. V is the set of vertices in the graph. V is the set of vertices in the graph. Each vertex $v \in V$ is a single version of the shared state replicated at every node. It is defined as $v = \{d \rightarrow x | \forall d \in D\}$ where D is the set of attributes that are stored in the shared replicated state and $d \in D$ is one such attribute. x is a value that is associated with each attribute in a single version. Each edge $e(v_i, v_j) \in E$ defines that v_j happened-after v_i. Another way we represent that relation is $j > i$. This relation is transitive. Therefore, if $k > j$ and $j > i$, then $k > i$. However, this does not mean that there exists an edge $e(v_i, v_k) \in E$. It means that there is at least one ordered set of edges $ep(v_i, v_k)$ that define a path along the directed edges from v_i to v_k. For example, if there are edges $e(v_i, v_j), e(v_j, v_k) \in E$ then $ep(v_i, v_k) = (e(v_i, v_j), e(v_j, v_k))$. There can be multiple such paths (depending on the branches that exist) between v_i and v_k. This set of all edge paths between v_i and v_k is denoted as $EP(v_i, v_k)$. Each edge $e(v_i, v_j) \in E$ is also associated with a state delta and is denoted as $|e(v_i, v_j)| = \delta_{i \rightarrow j}$.

6.6.2 Helper Functions

We define three functions for each version: $Pa(v)$, $Ch(v)$, and $Si(v)$ that define the set of immediate parents, children and siblings of a version v in a graph. We define $T(v, s, p)$ as

Table 6.1: Additional Metavariables

Metavariable	Meaning
D	Set of attributes in the replicated state at every node.
∂G	A partial directed acyclic version graph. $\partial G : (\partial V, \partial E)$
$v[d]$	Value x of the attribute $d \in D$ at version v.
r	The root version of every G.
H	The set of HEAD versions in G.($H \subseteq V$)
$e(v_i, v_j)$	A single directed edge in G from v_i to v_j.
$ep(v_i, v_j)$	An ordered set of directed edges in G forming a path from v_i to v_j.
$EP(v_i, v_j)$	Set of all possible $ep(v_i, v_j)$ in G from v_i to v_j.
$\delta_{i \to j}$	A state delta record, associated with $e(v_i, v_j)$

the user defined 3-way merge function used in every node of the system that is associative and commutative.

The function T takes in three inputs: a version $v \in V$ its sibling $s \in Si(v)$, and a least common ancestor $p \in Pa(v) \cap Pa(s)$ and returns a new merged version v_m. v_m happened-after both v and s. This function is used by the merge operation $M(v, s)$ that resolves write-write conflicts in the graph. This function M merges the version v_m generated by T into the version graph by adding the both the new version and two edges $e(v, v_m), e(s, v_m)$. This process is shown in Equation 6.1

$$M(v,s) = (\{v_m\}, \{e(v, v_m), e(s, v_m)\}) : v_m = T(v, s, Pa(v) \cap Pa(s)), v \in V, s \in Si(v) \quad (6.1)$$

To define the transitions via state deltas, we define two binary operations $+_c$ and $-_c$:

$$v_i +_c \delta_{i \to j} = \begin{cases} d \mapsto \delta_{i \to j}[d] & d \in dom(\delta_{i \to j}) \\ d \mapsto v_i[d] & d \notin dom(\delta_{i \to j}) \end{cases} : \forall d \in dom(v_i) \cup dom(\delta_{i \to j}) \quad (6.2)$$

$$\delta_{i\to j} +_c \delta_{j\to k} = \begin{cases} d \mapsto \delta_{j\to k}[d] & d \in dom(\delta_{j\to k}) \\ d \mapsto \delta_{i\to j}[d] & d \notin dom(\delta_{j\to k}) \end{cases} : \forall d \in dom(\delta_{i\to j}) \cup dom(\delta_{j\to k}), k > j > i \quad (6.3)$$

$$v_j -_c v_i = \delta_{i\to j} = \{d \mapsto v_j[d], v_j[d] \neq v_i[d]\} \quad (6.4)$$

Equation 6.2 defines state transitions through state deltas. A new state is constructed when a delta is applied by using the binary operation $+_c$ over the previous state. Delta values associated with edges can be combined into a single delta using the same function, as seen in Equation 6.3. This binary operator is associative but not commutative. Finally, a state delta can be obtained by diff-ing the state at two versions using the binary operation $-_c$ as shown in Equation 6.4.

6.6.3 Constraints

The first constraint that we have is that each version graph in a system has the same initial version r called the ROOT version. This is an invariant of the version graph in every node of the distributed system and is formally defined as:

C1: $\{v | v \in V, P(v) = \emptyset\} = \{r\}$

The ROOT version of each version graph stores the same initial state $\forall d \in D$. There can only be one ROOT version and each node in the system has the same ROOT version. All updates that every node makes have happened-after this ROOT version.

The second constraint **C2** is that the merge function T at each node must be: identical (across nodes), commutative, and associative. This is necessary to ensure that the state of the merge versions created from same parent versions at different nodes is the same irrespective of the order in which concurrent updates were observed at each node.

In any graph $G(V, E)$, we define the set of the pairs of versions that need to be merged using the 3-way merge function as K. In big-step 3-way merge, the precondition for conflict resolution is when G has more than one HEAD version after a write operation. i.e. $|H| > 1$. In fact, in systems which resolve conflicts with 3-way merges, the value $|H| = 2$ as merge is called on the first conflict detected. In such systems, $K = \{(v_i, v_j) \in H\}$. We have seen in Section 6.1 that this can lead to situations where there are multiple LCAs.

In order to avoid this problem, we instead set the post condition to be constraint $C3$. Formally we define the constraint as:

$$\textbf{C3: } \underbrace{\exists M(v,s) \in Ch(v) \cap Ch(s)}_{\text{There exists a merge for } v,\ s} : \quad \underbrace{\forall p \in Pa(v), \exists s \in Ch(p), v \neq s}_{\text{for at least one sibling } s \text{ of } v \text{ per parent } p \text{ of } v}$$

It states that a version v must be merged with at least one sibling s from every parent p it has. The version v will have at least as many merge versions as it has parents with siblings. The set K is defined as the pairs of versions v and one such sibling s that do not satisfy the constraint **C3**. A resolved version graph will have $K = \emptyset$.

6.6.4 Operations on Version Graph

In a break from the formal model described in Chapter 4, we define two simpler operations over the version graph G: get, and put, as shown in equations 6.5. Operation O1, $G_i.get(G_j)$, retrieves a partial graph of all versions and edges that are present in G_i but not present in G_j. This is the read operation on the version graph.

D6. Small Step Version Graph Functions:

$$\begin{aligned} &\text{(O1. Retrieve changes) } G_i.get(G_j): && \partial G(V_i - V_j, E_i - E_j) \\ &\text{(O2. Receive changes) } G \xrightarrow{G.put(\partial G)} && R(G \cup \partial G \cup B(G, \partial G)) \end{aligned} \tag{6.5}$$

$$\begin{aligned} &\text{(Bridge Version) } B(G, \partial G): && M(v_i \in V, v_j \in \partial V) : v_i \in Si(v_j) \\ &\text{(Resolve) } R(G): && \begin{cases} G & K(V) = \emptyset \\ R(G \cup \{M(v,s) | \forall (v,s) \in K(v)\}) & K(V) \neq \emptyset \end{cases} \end{aligned} \tag{6.6}$$

D7. Simplified Version Graph Functions:

$$\begin{aligned} &\text{(O3. Push) } G_i, G_j \xrightarrow{G_i.push(G_j)} && G_j.put(G_i.get(G_j)) \\ &\text{(O4. Pull) } G_i, G_j \xrightarrow{G_i.pull(G_j)} && G_i.put(G_j.get(G_i)) \\ &\text{(O5. Local Commit) } G \xrightarrow{G.commit(v_{new})} && G_i.put(\partial G(\{v_{new}\}, \{e(v_{head}, v_{new})\})) \end{aligned} \tag{6.7}$$

$G.put(\partial G)$, the write operation of the version graph, is a function that incorporates new versions and edges in ∂G into G. In this process, the new versions and edges are added into the graph. An initial bridge version $B(G, \partial G)$, if required, is then incorporated into the graph. To create B, we choose a v_i from the original set of versions V and a v_j from the incoming set of versions ∂V such that v_i and v_j are siblings of each other but do not have a merge version. If such a v_i and v_j does not exist, then the merge of G and ∂G has no conflicts, and the bridge version B is not required. B, when created, is incorporated into G before resolving any other conflicts. The recursive function $R : G \rightarrow G$ is then executed to resolve the remaining conflicts. In R, the merge procedure M is called for all mergeable candidates in K. For each candidate, the 3-way merge function T is invoked to get the merged version. This merged version, and edges connecting this version to G are created and incorporated into G. These merge versions can also be new candidates in K and so the function R is called recursively, until no such candidates remain.

The effect of both internode communication, and local updates on the version graph can be expressed as a combination of *get* and *put* as shown in Equations 6.7.

6.6.5 Single Lowest Common Ancestor

Theorem 6.1. *Every small-step 3-way merge will always have a single LCA.*

Proof: Every small-step 3-way merge only merges a pair of sibling versions. Let v_i and v_j be two siblings ($v_i \in Si(v_j) \wedge v_j \in Si(v_i)$) that have to be merged using the small-step 3-way merge. Since v_i, and v_j are siblings, the LCA is the set of their common parents. i.e $Pa(v_i) \cap Pa(v_j)$. To have a single LCA, they must have only one common parent.

$$|Pa(v_i) \cap Pa(v_j)| = 1 : v_i \in Si(v_j), v_i \neq v_j$$

Let us assume that $|Pa(v_i) \cap Pa(v_j)| > 1$. i.e There are more than two common parents for v_i, and v_j. By definition, a local commit (**O5** Equation 6.7) creates a version v_{new} that has only one parent. The only way for a version to have more than one parent is if it was a merge node. Therefore, v_i and v_j must be merge nodes.

By definition, A merge node is not created in the same node more than once. Therefore, either v_i, or v_j are versions created by the merge function T at different nodes in the system independently, and concurrently.

We know by definition that the same small-step 3-way merge function T is used by all nodes in the system, and T is commutative and associative. Therefore, two nodes merging the same set of versions, must give the same merged node. Therefore $v_i = v_j$. This means that two versions $v_i \neq v_j$ cannot have more than one common parent.

This common parent is the least common ancestor for v_i and v_j. Therefore, every small-step 3-way merge will always have a single least common ancestor (LCA).

6.6.6 Correctness

We define correctness in the version graph as the property that guarantees that the state at a version v_i can be transformed to the state at version v_j by applying all deltas, in order, along any path from v_i to v_j. Therefore, all paths must yield the exact same state. Formally we define this property to be

$$v_i +_c \sum ep(v_i, v_j) = v_j : \forall ep(v_i, v_j) \in EP(v_i, v_j), j > i \tag{6.8}$$

This property ensures that the same version v in the version graph at every node, has the exact same state. To prove that the version graph remains correct after every operation, we need to first prove that the state transitions are correct.

Theorem 6.2. *If δ is the state delta created using the binary operator $-_c$ on two versions $(v_i, v_j) : j > i$, then δ can be applied to version v_i using the binary operator $+_c$ to recover v_j.*

$$\delta = v_j -_c v_i \implies v_i +_c \delta = v_j$$

Proof: Let us first consider $\delta = v_j -_c v_i$. This δ is a map of attributes d to their values in version v_j, which is $v_j[d]$, for all the attributes where $v_j[d] \neq v_i[d]$. We know from Equation 6.4 that $\delta = \{d \mapsto v_j[d], v_j[d] \neq v_i[d]\}$

We also know from Equation 6.2 that:

$$v_i +_c \delta = \begin{cases} d \mapsto \delta[d] & d \in dom(\delta) \\ d \mapsto v_i[d] & d \notin dom(\delta) \end{cases} : \forall d \in dom(\delta) \cup dom(v_i)$$

$v_i +_c \delta$ is the map of attributes d to its value in δ, if present, otherwise maps d to its value in v_i. The attributes $d \in dom(\delta)$ are all the attributes where $v_j[d] \neq v_i[d]$. For the attributes $d \in dom(\delta)$, the value $\delta[d] = v_j[d]$. Therefore, the equation becomes:

$$\begin{aligned} v_i +_c \delta &= \begin{cases} d \mapsto v_j[d] & v_j[d] \neq v_i[d] \\ d \mapsto v_i[d] & v_j[d] = v_i[d] \end{cases} \\ &= v_j \end{aligned}$$

Therefore, the δ created using $-_c$ on two versions $(v_i, v_j) : j > i$ can be applied to version v_i using the operator $+_c$ to obtain v_j.

Fundamentally, this means that when δ is constructed as $v_j - v_i$ at one node, sent across the network to another node, and applied to the state at version v_i, we recover the same state v_j at the later node.

Theorem 6.3. *Correctness of a version graph G is maintained after every small-step 3-way merge.*

Proof: We prove the correctness by induction. Let us consider a graph G at one node in the system. Let us define $G(n)$ as the version graph after the n^{th} 3-way merge function is executed and integrated into G.

$G(0)$ represents the version graph before the first merge has occurred. Such a graph will not

have any branches, by definition. The graph will entirely be composed of a single chain of versions from the ROOT to the singular HEAD. There is, therefore, only one path from any version to any other reachable version in the graph. Therefore, our graph is correct.

$G(1)$ represents the version graph right after the first merge has occurred. 3-way merge function is called only if there exists immediate siblings v, s that share a common parent p but are not merged. The merge node v_m that is created is added to the graph as a child of both v and s. The state at the merge node v_m is determined by the user defined 3-way merge, and the deltas $\delta_{v \to v_m}$ and $\delta_{s \to v_m}$ are constructed from this predetermined version. We have already seen from Theorem 6.2 that the binary operation $+_c$ can be used to apply the delta correctly. Therefore, we can see that:

$$\begin{aligned} v +_c \delta_{v \to v_m} &= s +_c \delta_{s \to v_m} = v_m \\ p +_c \delta_{p \to v} +_c \delta_{v \to v_m} &= p +_c \delta_{p \to v} +_c \delta_{s \to v_m} = v_m \end{aligned} \tag{6.9}$$

The graph, before adding the merge version v_m, is correct as there are no other branches and this is the first merge seen by the graph. Any path through the new merge version v_m has to pass through either v or s, both of which have been shown to be correctly reach the state v_m. Additionally, v_m, which is a newly generated version, has no children. Therefore, the graph after $G(1)$ must be correct.

Now let us assume that the graph $G(n)$, which is the graph G after the n^{th} merge, is correct. When the $n+1$ merge is made, again a new node, say v'_m, is created. There are two possibilities going forward, either the version v'_m already exists and has been created as a result of a merge with either v' or s' with a different sibling, or the version v'_m does not exist and has to be added.

If it is the former case, we know that the old paths to v'_m are correct as $G(n)$ was correct and therefore, we need only concern ourselves with the new paths that were created. These new

paths must pass through the versions v' and s' that were used to create it. Using the same logic explained for $G(1)$, we can say that all possible paths from any version to the newly created version v'_m through v' and s' are correct. Therefore, all possible paths, old and new to v'_m are correct.

If v'_m already existed before, it may have children. Since the graph was correct before the merge, all paths v'_m as source are correct. In the same way, all paths that existed with v'_m as an intermediate remain correct, and all new paths with v'_m as intermediate must pass through v' and s' and therefore, must be correct.

We showed that $G(0)$, and $G(1)$ are correct graphs. We also showed that if $G(n)$ is a correct graph, $G(n+1)$ is also a correct graph. Therefore, by induction we can say that the graph is correct after every small-step 3-way merge.

6.6.7 Convergence

Version graphs at different nodes can temporarily diverge and have different states. However, when such graphs are synchronized, either through push or pull communication, the version graphs must converge. In big-step 3-way merge, convergence is a trivial property to satisfy. When two version graphs are synchronized, the divergent HEAD versions are explicitly merged using the 3-way merge giving one merge node that becomes the new singular HEAD node in the version graph. We have already discussed in detail the flaws with this approach. In our approach, however, we recursively merge only immediate sibling versions in order to avoid losing important information by merging multiple updates. We will now prove that our approach also guarantees the same convergence. At the end of every write operation $G.put(\partial G)$, there is only one HEAD version. i.e. $|H| = 1$. In order to prove convergence we need to establish additional properties of the version graph.

Theorem 6.4. *The total number of merge versions added for n concurrent updates at a Node*

is $n(n-1)/2$

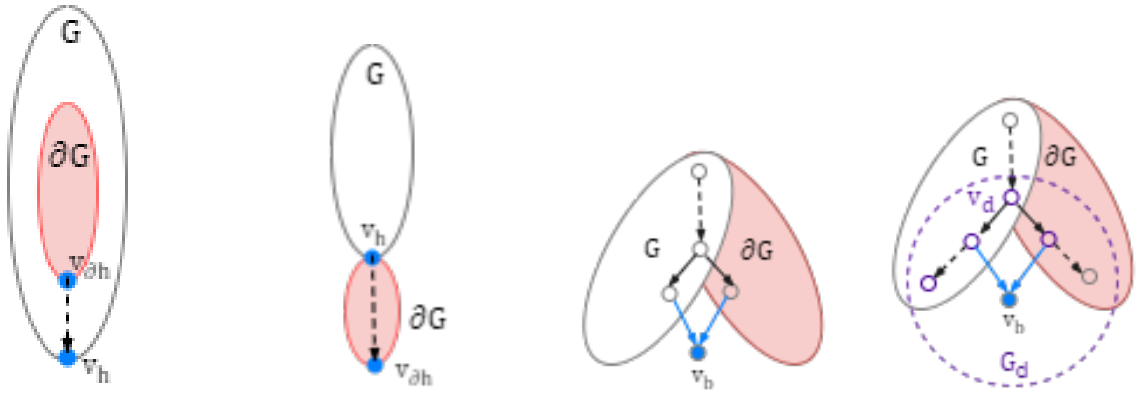
When the first update $G.put(\partial G(\{v_1\}, \{e(v, v_1)\})$ is made, the version graph gains a version v_1 and an edge $e(v, v_1)$ but is not in conflict. Therefore no merges are performed. v_1 is the new HEAD. When v_2 is added next, v_2 and v_1 are siblings of the common parent v and are merged giving us our first merge version: $M(v_1, v_2)$. $M(v_1, v_2)$ has no siblings, and therefore, the conflicts are resolved. For two concurrent versions, one merge version was added.

When v_3 is added to G, it has two siblings v_1, and v_2 with parent v. It can choose any one sibling, say v_2 to create a merge version $M(v_2, v_3)$. The version $M(v_2, v_3)$ is a sibling of $M(v_1, v_2)$ via the parent v_2 and is merged to obtain a new merge $M(M(v_1, v_2), M(v_2, v_3))$. When the third concurrent version is added, three additional merge versions are created. When we add the ith concurrent version, we need to perform $i-1$ merges. Therefore when we add n concurrent versions, the total number of merge versions added is $1+2+\cdots+(n-1) = \sum_{k=1}^{n-1} k = n(n-1)/2$, thus proved.

After every write operation *G.put* on a version graph G at a node, the version graph should converge to one version that happened-after every other version in the graph. This is the HEAD version, and there must only be one HEAD. i.e. $|H| = 1$. This HEAD version is the current state of the replicated object at the node upon which new local commits can be made.

Theorem 6.5. *The version graph G after any G.put(∂G), has only one HEAD version*

Assumptions: We assume that ∂G is created by either by $G'.get(G)$ where G' is a remote version graph, satisfying all constraints (**C1**, **C2**, and **C3**) and having one HEAD version $v_{\partial h}$, that is sending changes to be merged into version graph G, (Operation O1, Equation 6.5) or a $G.commit(\{v_n, \{e(v_h, v_n)\}\})$ (Operation O5, Equation 6.7) where v_n is a unique version that is created and v_h is the current HEAD of G. The final assumption is that, unlike file based version control systems, there are no rollbacks to state during distributed computation in this system.



(a) Scenario 1 (b) Scenario 2 (c) Scenario 3 (d) Scenario 4

Figure 6.4: Four scenarios that can arise during $G.put(\partial G)$.

Proof: We prove this by induction. Let us consider the state of the graph G before the first *put* is invoked (represented for this proof as $G(0)$ The graph has only one version, the root version r, and no edges. $G(0) = G(\{r\}, \emptyset)$.

At $G(1)$, $\partial G = G' - G$. Since G has no edges, $\partial G = G' - (\{r\}, \emptyset\}$. When $G.put(\partial G)$ is invoked, $G = G'$. Since HEAD at G is the same as HEAD at G' which by assumption is just one, $v_{\partial h}$. Therefore, after $G(1)$, G has only HEAD version.

Let us now assume that after the nth $G.put$, $G(n)$ has only one HEAD version v_h. Let us now observe the state of G after $G(n+1)$.

We can see that from Operation O2, Equation 6.5 that there are four steps to the the *put* operation.

- **Step 1:** The new edges and versions in ∂G are added to G. $(G \cup \partial G)$
- **Step 2:** A bridge version $B(v_i, v_j)$ is created if condition $(v_i \in G, v_j \in \partial G, v_i \in Si(v_j))$ is met.
- **Step 3:** The bridge version B is added to $G \cup \partial G$.
- **Step 4:** Versions that break the constraint **C3** $(K \neq \emptyset)$ are recursively merged using

the 3-way merge until $K = \emptyset$.

There are four possible scenarios for the operation *put* (Figure 6.4).

Scenario 1: At Step 1, $\partial G \subseteq G$. The versions and edges being put into G already exist with G (Figure 6.4a)[2]. That means that $v_{\partial h} \in G$. By definition, v_h happened-after every version in G. Therefore, $h > \partial h$ and $G \cup \partial G$ has only one HEAD version: v_h.

Scenario 2: At Step 2, a bridge version does not exist $B(G, \partial G) = \emptyset$ (Figure 6.4b). This means that there two sibling versions v_i, and v_j that meet the condition ($v_i \in G, v_j \in \partial G, v_i \in Si(v_j)$) do not exist in $G \cup \partial G$. This means that all $j > i$. It implies that ∂G has not diverged from G, but is merely a chain extension to G. Therefore, $\partial h >= h$. Thus, $v_{\partial h}$ becomes the sole HEAD version of the graph $G \cup \partial G$ and no conflicts need to be resolved.

Scenario 3: At Step 2, there exists a bridge version $v_b \in B(G, \partial G)$, but at Step 4, there are no versions that are candidates for a merge in the graph $G \cup \partial G \cup B(G, \partial G)$. i.e. $K = \emptyset$. (Figure 6.4c). In this scenario, the Step 2, bridge version v_b was the only candidate for a merge. After Step 3, the graph satisfied constraint **C3**. This can occur only when the bridge version v_b is $M(v_h, v_{\partial h})$. ∂G and G diverged only at the last update. The bridge version v_b is the sole HEAD version and all conflicts have been resolved.

Scenario 4: The final scenario is when, at Step 2, there is at least one bridge version $v_b \in B(G, \partial G)$ and at step 4, there are more candidates for a merge in the graph $G \cup \partial G \cup B(G, \partial G)$ i.e. $K \neq \emptyset$. (Figure 6.4d). Since each of the graphs G, and ∂G individually satisfy the constraint **C3** (from assumptions), every version in both graphs, taken independently, have been merged with at least one sibling from each parent. Therefore, if $K \neq \emptyset$, then $v_b \in K$. This occurs when there have been more updates in both G and G' since the point of divergence (which is called v_d from now on)

[2]Since ∂G is defined as $G' - G$, for some version graph G', the *put* scenario $\partial G \subseteq G$ can only occur when node with G' is resending ∂G having not received any acknowledgement. It caters to network partition tolerance.

Let us say that the bridge version v_b happened-after two versions $v_1 \in G$ and $v_2 \in \partial G$. Additionally, $v_1 \in Ch(v_d)$, and $v_2 \in Ch(v_d)$. Since this was the point of divergence, the merge of v_1 and v_2 was never created until now.

To understand the creation of the rest of the merge nodes, let us observe the operations of the recursive function R (Step 4, Operation O2, Equation 6.5) on the subgraph $G_d \subseteq G$, rooted at v_d, including all versions that are reachable from v_d (see Figure 6.4d). Both v_h and $v_{\partial h}$ are going to be reachable from v_d. The merge operations as a result of R will only act on these versions.

Let $L(v)$ be a function that defines the number of edges along any directed path from v_d to version v and $Z(l)$ be a function that counts the number of versions in the subgraph G_d that have $L(v) = l$. All versions that have the same $L(v) = l$ are said to be at level l. By definition, $L(0) = 1$

If a node makes an update from version v_i to v_j, then $L(v_j) = L(v_i) + 1$. Since all merges in our approach are applied to sibling nodes, using the common parent, each edge only ever represents one update. Therefore, all versions having the same $L(v)$ value are versions that have the same number of concurrent updates applied to them and are concurrent versions. They are said to be at the same level.

Since a node in the system cannot perform rollbacks (by assumption), a node could only ever have made one update from one of the versions with the same $L(v)$ value. Once it has made an update, the state at that node is at a version v' with $L(v') = L(v) + 1$. Therefore, the maximum number of updates that can be found from versions with the same $L(v)$ in G_d is the number of nodes in the system n (**F1**).

From Theorem 6.4, we know that n concurrent updates at version v, produces $n - 1$ versions at the next level $L(v) + 1$.

Combining Theorem 6.4 and (**F1**), we can see that the number of versions at a one level l given by $Z(l)$, is the number of updates made by nodes in the system (upper bounded by n), and the number of merge nodes at that level (upper bounded by the number of nodes at the previous level $Z(l-1)-1$. With this, function $Z(l)$ can be partially defined as:

$$Z(l) \leq \begin{cases} 1 & l = 0 \\ Z(l-1) - 1 + n & 0 \leq l \leq max(L(v_h), L(v_{\partial h})) \end{cases} \tag{6.10}$$

Up until the recursive merge reaches the max depth of graph G_d, which is given by $dep = max(L(v_h), L(v_{\partial h}))$, the number of merge versions created can increase to the order of $O(n \times dep)$. When it reaches max depth, there are no more updates from other nodes in the system to consider. From there, the only versions at each level are the previously merged versions.

Therefore, function $L(z)$ can be defined as:

$$L(z) \leq \begin{cases} 1 & z = 0 \\ L(z-1) - 1 + n & 0 \leq z \leq max(Z(v_h), Z(v_{\partial h})) \\ L(z-1) - 1 & z > max(Z(v_h), Z(v_{\partial h})) \end{cases} \tag{6.11}$$

The number of versions at each level decreases after max depth has been reached, up until it becomes one. The last merge version created becomes the sole HEAD version.

Since in all four scenarios of *put*, the newly created graph G has one HEAD version, the version graph $G(n+1)$ after the $(n+1)$th call to $G.put(\partial G)$ operation converges to one HEAD version.

We showed that, the version graph had only one HEAD node at $G(0)$, and $G(1)$. Additionally, when we assumed G had one HEAD version after $G(n)$, we found that it had one HEAD version after $G(n+1)$. Therefore, by induction, we prove that after every $G.put(\partial G)$, the version graph G always converges to one HEAD version.

Chapter 7

Challenge 2: Unbound Growth of the Version Graph

While the aspects of GoT we have touched upon till now involve state replication, we have not yet addressed the elephant in the room. As changes are committed, the version graph grows. For Git, which operates at human speeds, this is not a problem, as the graph grows very slowly. For Spacetime applications, the version graph can grow rapidly, resulting in processes running out of memory. For example, the physics node shown in Listing 4.5 is doing 20 commits per second! This is a significant drawback of version control based state replication. Optimizations must be made to keep the version graph in check while still retaining its existence and function. In this chapter, we discuss the garbage collection strategies of Spacetime that allow the GoT programming model to be feasible. It is important to note that the garbage collection strategy used depends on the type of merge function used and the GoT application's network architecture. As such, we explain the garbage collection strategies employed in both big step and small step merge. These optimizations are necessary for realistic implementations of GoT in current computers but are not part of the conceptual model.

7.1 Related Work

The topic of garbage collection in version control closely resembles the work done in CRDTs on causal stability [8] and message obsoletion [95, 106].

Casual stability in CRDTs determines which messages have been received by every node in the system and, therefore, can be safely deleted. Systems such as vector clocks [70], version vectors [82], or even simple tombstones have been used to detect and delete messages that are known to every node in the system.

While deleting messages received by every node in the system is a step in the right direction, the concept can be taken even further. Deleting obsolete messages is not the same as deleting messages that are received by every node. There can be intermediate messages that are required by the same set of nodes. Such messages can be potentially composed to reduce communication. This process is stated as the eventual goal of causal stability.

Many techniques exist for deleting unused variables in shared transactional memory systems and is explained in detail by Wiseman [111]. However, these techniques do not translate well to replicated states where there is no single shared state.

File-based version control systems such as Git allow users to squash the version history to shorten the version graph. Such techniques are usually employed to keep the version graph in large scale projects manageable and easy to clone. In object-based version control systems, only TARDiS [30] has some form of garbage collection. The garbage collection employed in TARDiS is heuristic with stale versions that are beyond a threshold age are garbage collected. This garbage collection is simple to build and use. The staleness threshold can be tuned to the requirement of the application. However, it suffers from three significant drawbacks.

First, it can only be used when the version graph tracks the full version. If old versions are to be deleted and not merged, then the oldest version present after garbage collection must

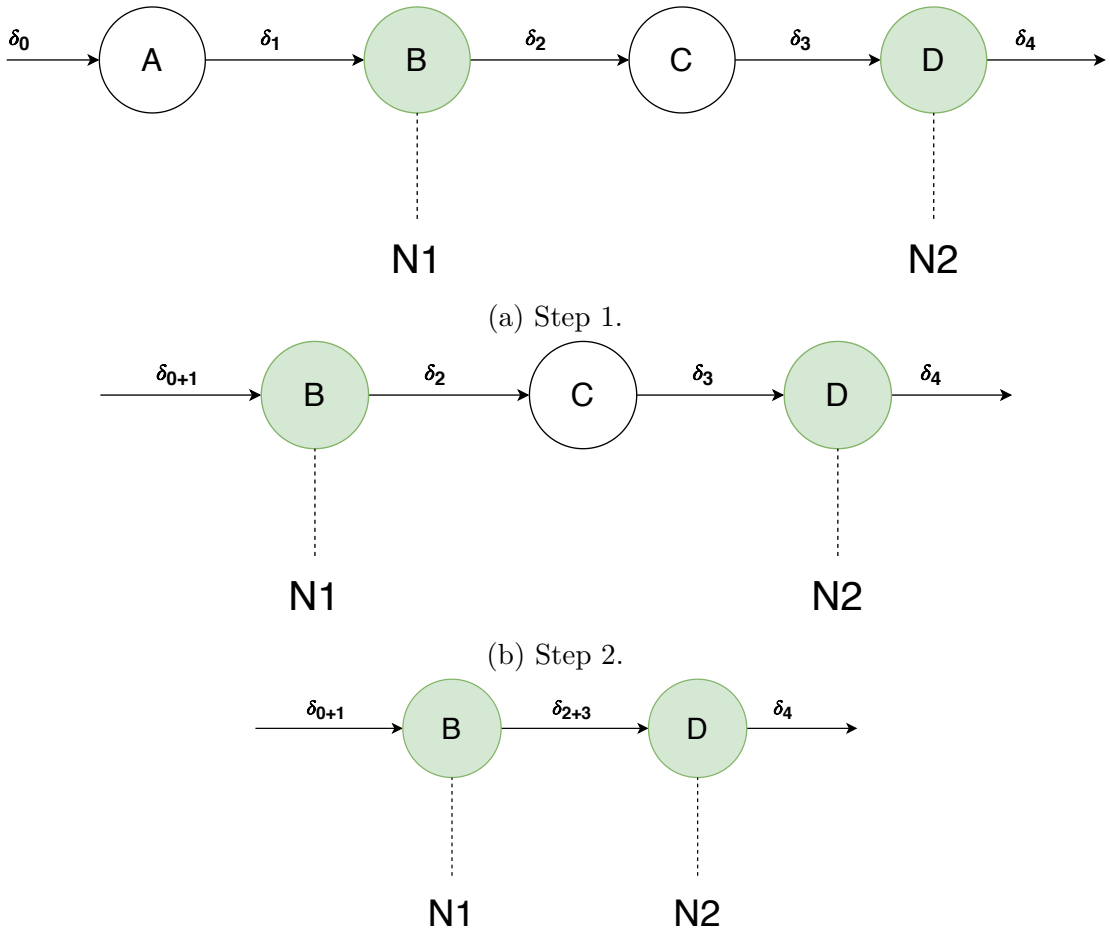
have the complete state of the system. Otherwise, new nodes synchronizing with the system cannot obtain the whole state on full sync.

The second problem is that this approach is not partition tolerant. If a version is maintained by a replica that is partitioned for a long time, it is in danger of being garbage collected before the replica rejoins the network. Once the replica rejoins, any offline updates made by the replica cannot be distributed as the version that it is causally related to is no longer present.

Finally, TARDiS employs this garbage collection as a periodic process that cleans up all unneeded versions. Such a process can be intrusive and can cause the system to slow down. A garbage collection process that marks and deletes versions as they become obsolete is more efficient as the cost of garbage collection is accrued only when a version has to be deleted. Process time is not wasted scanning the versions in the version graph periodically, even if there are no versions to delete.

7.2 Basics of Garbage Collection

The purpose of garbage collection is to remove the unneeded versions from the version graph. In order to do that, we must first understand when a version is no longer required. Let us take the example shown in Figure 7.1a. Here we have four versions A, B, C, and D such that D happened-after C, which happened-after B, which again happened-after A. The versions that happened before A and the ones that happened after D are not shown in this version graph. The deltas shown at each edge is the update for that transition of versions. Finally, we have two nodes that are synchronizing on this particular version graph: $N1$ and $N2$. The last versions synchronized with $N1$ and $N2$ are versions B and D, respectively, as shown by the dotted markers. This knowledge is essential. Without knowing which node is at which



(a) Step 1.

(b) Step 2.

(c) Step 3.

Figure 7.1: Basic Garbage Collection.

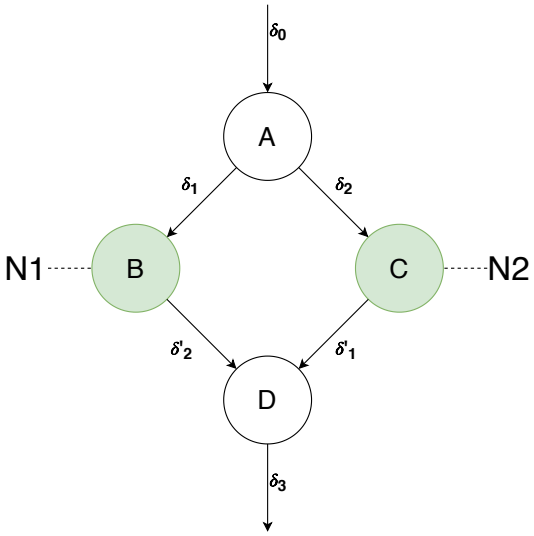
version, the version graph cannot prune unneeded versions.

The first and simplest rule that can be formed is that when a version is known to all the intended recipients, the version can be deleted. For example, version A can be deleted as both $N1$ and $N2$ know version A. This is shown in Figure 7.1b. For all nodes that are making a fresh synchronization with the version graph, the delta changes to version A is the combined delta δ_{0+1}. This is a good starting rule and is the basis of the techniques such as tombstones. However, in a system aiming to provide causal consistency, it can lead to potential problems. For example, if $N1$ has crashed or has been network partitioned, the versions after version B are no longer garbage collected.

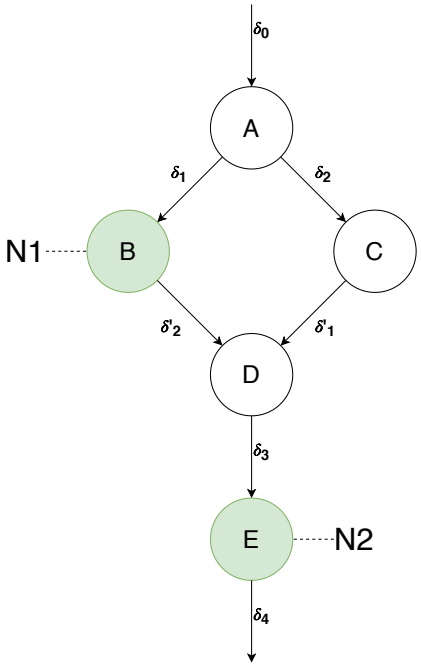
To address this, we can add a second rule that says that all versions in between versions that are referenced by nodes can also be deleted. With this, we can say that version C can be deleted, and the delta change from B to C is δ_{2+3}, shown in Figure 7.1c. If $N2$ were to synchronize, it would only receive the versions after D, and therefore, version C is not required. If $N1$ were to synchronize, it would pull all changes from B to the latest state. Version C is an intermediate state that it does not require. $N1$ can directly skip to version D and beyond using δ_{2+3}. With these two rules, we can see that the number of versions would remain a function of the number of Nodes that are communicating.

As a note, this method only works if individual updates can be composed together into larger updates. In several models of communication, updates are shared in the form of operations and not deltas. Operations in subsequent updates may often be incompatible to such a composition and, therefore, might require to stay separate. In GoT, however, updates are shared as state deltas and are composable. Additionally, a composition such state deltas might have the effect of decreasing the size of the update. For example, if an object was created in update δ_2 and subsequently deleted in δ_3, then the composed update δ_{2+3} will have no record of that object whatsoever, thereby decreasing the size of the update. At worst, the combined delta is as large as the sum of the individual deltas.

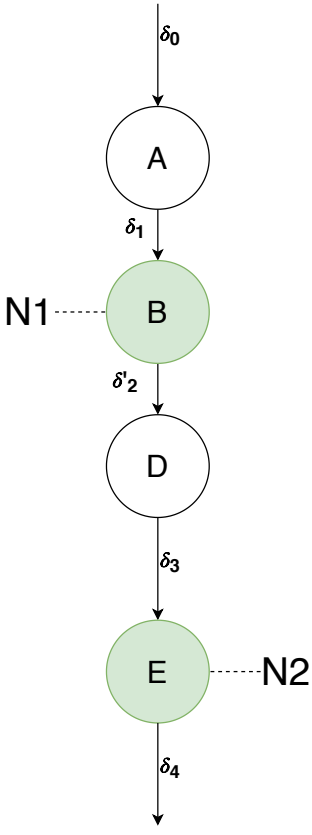
While these rules work for a sequential set of updates, the concurrent nature of the updates in a version graph demands more rules. If we look at Figure 7.2a, we can see a classic case of concurrency. Updates δ_1 and δ_2 are concurrent updates from version A. Besides the preceding δ_0 update, nodes $N1$ and $N2$ know the updates δ_1 and δ_2 respectively. Even though no node references A and D, they cannot be garbage collected. When one node synchronizes and the reference is updated out of the branch (shown in Figure 7.2b, N2 moves to version E), then the branch is deleted. Additional edges are not required between versions A and D as there already exists a path from the two versions ($A \rightarrow B \rightarrow D$). We see the outcome of this step in Figure 7.2c. We can also see that version D can now be deleted as well, using the rules we



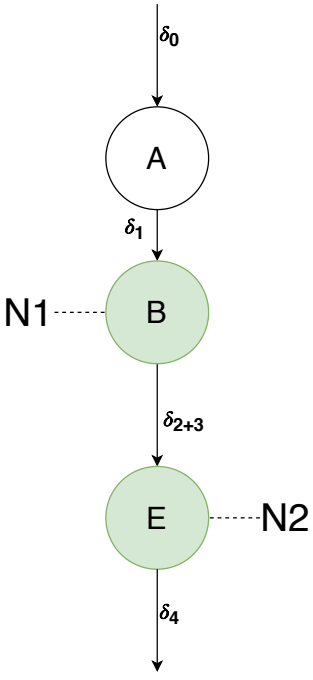
(a) Initial state.



(b) Push from $N2$.



(c) Step 1.



(d) Step 2.

Figure 7.2: Garbage Collection of Concurrent Update.

have described before, which brings the version graph to what is shown in Figure 7.2d.

The core purpose of the garbage collection of versions is to ensure that the number of versions remaining is to the order of the number of nodes synchronizing with the version graph while ensuring that each node synchronizing with the version graph does not miss any updates that it needs. Techniques that use tombstones only delete the versions that have been received by every node. In contrast, the garbage collection of versions in GoT also deletes unnecessary intermediate versions.

7.3 Server Client Model: Practical Garbage Collection in Big Step

In the previous section, we determined that if we had a reference marker in the version graph mapping each remote node that interacts with the version graph to its latest version synchronized, we could effectively prune the version graph out of unneeded and unnecessary nodes. In this section, we will go over the garbage collection process from an operations perspective.

Each GoT node contains a reference map that maps every remote node that it interacts with to the last known version sent or received. These reference maps are updated both when the node itself pushes or fetches updates, or when the node receives push and fetch requests. Let us consider two nodes, $N1$ and $N2$ with head versions h_1 and h_2. When $N1$ receives updates, it is either a response to a fetch request that $N1$ made to $N2$ or a push that it received from $N2$. During this operation, the node $N2$ first reads its version graph up until h_1. It then updates its reference map to map $N1$ to h_1. The older reference is maintained during the transfer until the update is received and acknowledged by node $N1$. This delay in moving the markers ensures that a partition in the network does not cause a preemptive deletion

of required versions. The receiving node $N1$ applies the updates to its version graph, and updates its reference map, pointing $N2$ to h_2. This reference map update is made even if the updates received were conflicting with h_1 and a merge version was created. This is because while node $N1$ has merged the conflicting updates, $N2$ has not, and has only sent updates up to h_2.

After the updates are written into the version graph at $N1$, and the reference map is updated, $N1$ performs the garbage collection operation described in the previous section. Even though the reference map at $N2$ was updated, $N2$ does not perform the garbage collection up until the end of the next write. This constraint ensures that the version graph only changes when updates are written to it. Even though there are potential versions that could be garbage collected after a read, since a read cannot increase the number of versions in the graph, the space requirement of the version graph remains unchanged. Aggressively applying garbage collection even on a read, only serves to hurt performance, as a mutual exclusion lock has to be now applied on a read as well, slowing down the number of concurrent operations the version graph can handle.

In the systems that communicate using operations, consolidating updates as described requires domain-specific knowledge and can sometimes be impossible. For example, the process of consolidating two additions $+2$ and $+3$ to $+5$ is different from consolidating two multiply operations $x2$ and $x3$ to $x6$. It requires knowledge of what the operations being consolidated are.

In Spacetime and delta state-based systems, however, the delta state updates can be consolidated very easily. The delta update does not encode an operation but a partial state of attributes. Let us take for example, a series of operations changing the state from $A : (x = 5, y = 5)$ to $B : (x = 10, y = 10)$ to $C : (x = 20, y = 10)$. The delta state from state $A \rightarrow B$ would be $(x = 10, y = 10)$ and the delta state from $B \rightarrow C$ would be $(x = 20)$. If B is to be deleted, then the delta from $A \rightarrow C$ is $(x = 20, y = 10)$. The intermediate step

where x was 10 is obsolete information and can be removed.

While the basic garbage collection rules work for big-step merge in server-client topologies, new rules are required to handle garbage collection in small-step merge and peer to peer topologies

7.4 P2P Model: Garbage Collection in Small Step

To understand garbage collection in small-step, we need to look at two aspects of garbage collection independently. First, the reduction of a small step version graph at a single node assuming that the reference map is correct. Second, the construction and maintenance of the reference map.

It is important to note that this approach to garbage collection is the initial step in designing a complete garbage collector. Due to the inherent difficulties in implementing this model, the approach remains mostly unverified. The work to refine the approach is still ongoing. Explained in this section are the initial set of rules.

7.4.1 Reducing the Version Graph

The basic rule in a version graph maintained by the small-step merge is that every update is only ever resolved against a sibling version. This ensures that the version graph is a full lattice of partially ordered updates. Every path from the ROOT version, to the HEAD version has the same set of updates. Unlike big-step, where several updates can be composed together in different paths. For example, in Figure 6.1d in Chapter 6, we see that in version N_1, the path $A \to B \to D \to M_1$ has the updates u_1, u_2, and u_3, whereas the path $A \to C \to M_1$ has the updates u_2, and $u_1 + u_3$. The composed update $u_1 + u_3$ cannot be broken back down into its

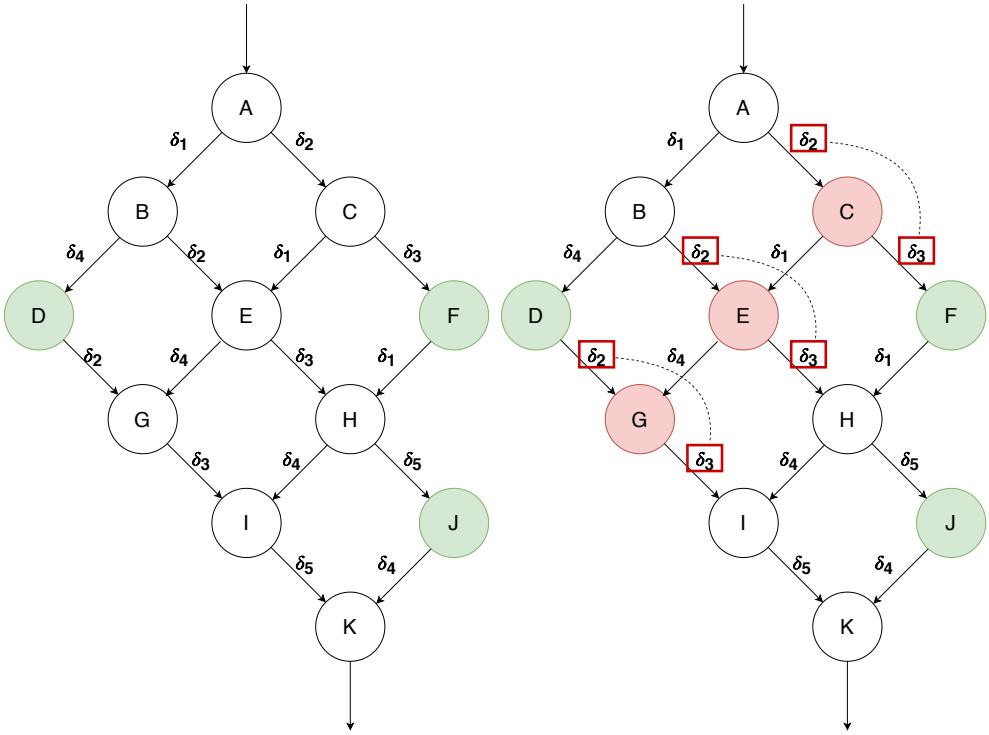
constituent updates. In the small step merge at Figure 6.2c, every path has the same set of three updates u_1, u_2, and u_3. The only difference in each path is the order of the updates.

When garbage collecting the version graph, this property that every path from ROOT to HEAD must have the same set of updates must be maintained. To satisfy this constraint, when updates are composed, all occurrences of those updates must be composed. Let us understand this garbage collection by using an example. In Figure 7.3a we see a version graph that has been merged with small step merge. We see eleven versions, from A to K. References to versions D, F, and J are maintained in the the node's reference map. There are four updates in the lattice: δ_1, δ_2, δ_3, δ_4, and δ_5.

The first step is to identify the causal relation between these updates. To do this, we traverse the version graph in breadth-first order. The first time we encounter an update, we know that it happened-after the set of updates that are incident on the parent. For example, in the breadth-first order, we first encounter edges $A \rightarrow B$ and $A \rightarrow C$. The updates δ_1 and δ_2 are considered concurrent updates as there are no updates that are incident on version A. δ_4 is first encountered at the edge $B \rightarrow D$ which establishes that update δ_4 happened after δ_1. Similarly, δ_3 is first encountered at the edge $C \rightarrow F$ which establishes that update δ_3 happened after δ_2; and δ_4 is first encountered at the edge $H \rightarrow J$ which establishes that update δ_5 happened after both δ_1 and δ_3. The update δ_5 is also indirectly causally linked to δ_2, but we are interested in only the immediate causal relation.

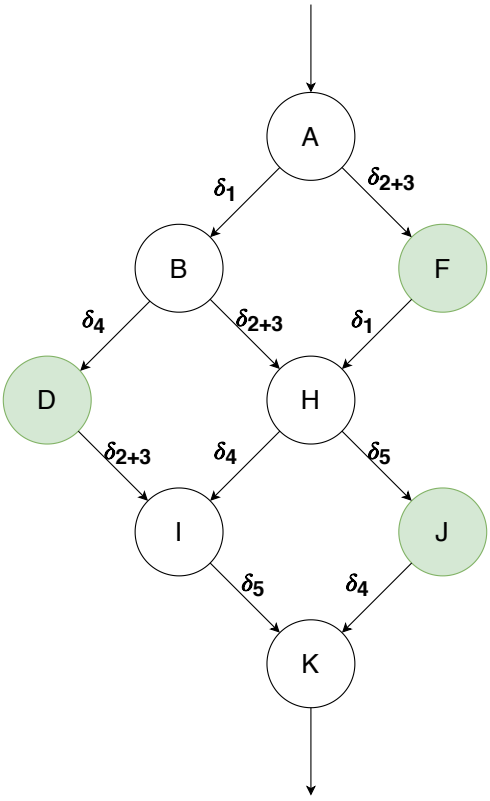
The second step is to identify, for each update in the graph, the set of nodes that require the update. The the given example, update δ_2 is required by only $N1$. The update δ_3 is also only required by $N1$. Update δ_1 is not known by $N2$. Update δ_4 is not known to both $N2$ and $N3$ while δ_5 not known to $N1$ and $N2$.

Using the causal relation between updates from step one and the knowledge of missing updates from step two, we can see that causally related updates δ_2 and δ_3 are known to both



(a) Initial state.

(b) Identifying composable groups.



(c) Merging composable groups.

Figure 7.3: Garbage Collection in Small Step.

$N2$ and $N3$ and not known to $N1$. Since these updates are causally related and are either known to nodes, or both unknown to nodes, they can be combined into one update. Updates δ_1 and δ_4, though causally related cannot be combined as $N3$ knows δ_1 but not δ_4. Similarly, δ_5 cannot be combined with both δ_3 and δ_1 as different nodes require different parts of these updates. If we combine updates, we must combine all occurrences of these updates. Since there are nodes that require a section of these updates, they cannot be combined.

Having recognized that updates δ_2 and δ_4 can be combined, we find all occurrences of the two updates with each other. If the updates are found independently, but not together, they cannot be collected, and the entire group is not collected. In this example, as shown in Figure 7.3b, Edge groups $(A \rightarrow C \rightarrow F)$, $(B \rightarrow E \rightarrow H)$, and $(D \rightarrow G \rightarrow I)$ are all updates δ_2 followed by δ_4. These edges will be merged, and the versions C, E, and G can be deleted. The merged edges are treated as a new update δ_{2+3}. We can see the result of such a garbage collection in Figure 7.3c.

7.4.2 Reference Passing in Peer to Peer

As discussed in Chapter 6, small step merge is a technique that is used in peer to peer network topologies. At the operations level, in server-client and tree-based topologies, where big-step is used, each GoT node can have many clients connected to it but connect to only one server. This means that a node can perform its push or fetch on to one server and receive many push and fetch requests from multiple clients. Since the same node can not be both a server and a client for a GoT node, only one version per node is maintained in the reference map.

In small-step, however, a node can initiate push and fetch requests with the same node from which it is receiving push and fetch requests. Since these operations can execute concurrently, the two nodes can read different HEAD versions and merge the concurrent HEADs independently. To keep track of both these versions at each node, we need multiple

versions maintained for the same node in the reference map. The intuition for which versions need to be maintained by each node in the reference counter is that only versions from which the node could make an update must be maintained. In this scenario, the update can be made from the HEAD version sent, HEAD version received, or the semilattice-join of these two versions

7.4.3 Implementation Challenges

While, in theory, this seems easy to implement, in practice, due to the non-determinism of updates and the arbitrary partitioning of networks, maintaining the reference map at each node can be complicated. We have achieved some success in the garbage collection of versions in a system containing two nodes, that push and pull from each other. However, scaling to a larger number of nodes introduces many complicated errors yet to be solved. Stabilization of the process in Spacetime is an ongoing, laborious process. In the future, we aim to address these issues and improve the theory behind garbage collection of versions in version graphs that use small-step merge.

7.5 Summary

The viability of Spacetime in shared-state applications that last long and update frequently depends on its ability to keep the version graph under control. The number of versions in the version graph has to be bounded as an unbounded version graph is essentially a memory leak.

In this section, we showed the basic principles of garbage collection. While causal stability and message tombstones are excellent techniques to use, they fail when there are network partitions as they do not help remove intermediate updates. When a node is partitioned from updates, all the updates that it is missing, but has been received by other nodes cannot

and should not be deleted. They must instead be merged into a single update.

We showed that garbage collection in big-step merge of Spacetime, with the help of a basic version reference map at each node, implements three rules:

- First, all versions that are known to every node are deleted. This condition is similar to those in causal stability and tombstoning updates. The versions to be deleted can be detected using a simple breadth-first scan from the ROOT node, up to the earliest versions being held in the version reference map.
- Second, all versions that are intermediate steps in between versions that are referenced by the nodes can also be deleted.
- Third, branches in the version graph, which are concurrent updates, cannot be deleted until one branch has no version held by nodes.

With these three rules, the versions created in the version graph that use big-step merge can be maintained to the order of the number of nodes in each reference map. We prove this, using microbenchmarks that are explained and discussed along with other experiments in Chapter 10.

These rules are, however, inadequate in peer to peer applications that need to use small-step merge. In garbage collection of version graphs maintained by small-step merge, we first identify the causally related groups of updates that are either known to every node, or not known by precisely the same set of nodes. Every instance of these updates is then identified and combined. These rules represent preliminary directions and intuitions involved in the correct garbage collection of obsolete versions in small-step. In practice, the maintenance of the reference map in asynchronous peer to peer is hard, and we have not fully identified the safeguards and processes required when there are more than two nodes in the system. In the future, we aim to identify the entire set of rules required to perform correct garbage

collection in small-step merge. In Chapter10, we discuss garbage collection in version graphs using small-step merge using microbenchmarks over two peer nodes.

Chapter 8

Challenge 3: Interest Management

The contents of this chapter have published in the Journal of Object Technology [66].

In the context of Spacetime, as we have discussed up until now, all objects have to be shared between nodes. This can be problematic when the shared-state is large and all nodes do not need the entire state. Partitioning the data for the nodes reduces the network traffic, as only data that a node needs will be synchronized. It also satisfies the interest management requirement for reducing communication and reducing update latency. Additionally, it provides the shared-state a level of security, as nodes that should not have access to certain data need not receive it (commonly know as Principle of Least Privilege). The disadvantage, however, is that the diffs are more expensive to compute and to merge. The computation directly depends on the number of partitions but the partitions can be synchronized independently.

Interest management in version control would mean synchronizing on a smaller subset of the shared repository. File-based version control systems provide varying degrees of support for such a task.

Git, for example, tracks the whole repository in the version graph, and it is impossible to

synchronize only over a subset of the repository. This is because updates to the repositories are better tracked on the whole tracked filesystem rather than on a per file or folder basis. Centralized repositories like Perforce [110], on the other hand, provide the option to synchronize over a subset using hard branches, but that comes at the cost of being very complex and hard to manage. Furthermore, the granularity that Perforce offers is at the file level.

In shared space distributed systems, true interest management would translate to the ability to pick and choose down to the dimensions the data required to be synchronized. This principle is heavily ingrained in the share-state programming models. For example, databases typically offer mechanisms to query the data stored and retrieve a smaller subset of the database. Relational algebra is typically used to define these slices dynamically at each request. For example, it is possible for a node synchronizing over a table of Cars in a database to only pick up those cars that have velocity greater than zero (or are moving). This is a dynamic query. Assuming the cars are mutable, if the query is made multiple times, the set of cars that is returned can be different each time. Such intricate query mechanism do not typically work nicely with version control systems.

8.1 Basic Interest Management in GoT

In the case of GoT, as explained till now, nodes can synchronize over the types that they need. Let us take for example, a distributed simulation of a city. The shared space of an GoT application can be composed of two types: People and Cars. Let there be three types of nodes in the simulation: pedestrian simulator, traffic simulator, and a taxi service. The pedestrian simulator only needs objects of type people, the traffic simulator needs objects of type Cars and the taxi service required both to match people who are waiting for a ride to a free cab which is a car.

As is, GoT supports synchronizing over individual types. So while the full shared-space is hosted on the authoritative node (say the taxi service), the traffic simulator only pulls changes to the cars. When the traffic simulator makes a pull request, the taxi service picks up any delta updates that the traffic sim does not have, and then filters it to the types that are required and sends it. If there are no updates to cars, but there are updates to pedestrians, then an empty delta is still sent to update the latest version at the traffic sim. This kind of division is easy to implement as a car can never change its membership. It can never become part of the People set.

However, dynamic queries are still not possible. Not every car in the set of Cars is an actively moving car. Many cars can be parked. The traffic sim does not need to know about these cars until they move. Additionally, it does not need to know all the properties of these cars that are not relevant. e.g. ownership, color, engine VIN number, etc. Therefore, in some cases, type level synchronization can still introduce redundancy in communication.

To fix this problem, we introduce predicate collection classes, by taking inspiration from relational algebra. Predicate collection classes are a typing system that works with dataframe to allow nodes to specify synchronization over dynamic collections of objects.

8.2 Introduction to Predicate Collection Classes

In many branches of study, including mathematics, the word *class* is, formally or informally, used to denote sets of things that can be unambiguously defined by a property that all of its members share [63]. In this sense, *classification* is the process of placing data with common properties into sets or collections. For historical reasons, in OOP languages, the word *class* has a related, but slightly different, meaning: a *class* is a template for *constructing* objects that have one property in common, namely, they are modeled after the same nominal pattern

of state and behavior, the *class* itself [18]. In OOP, we can look at a class C as a function that takes data and returns an object with a specific set of fields and methods, as defined by some pattern; all objects instantiated using C are said to be of the same class.

We are interested in expression mechanisms that allow us to classify data depending on runtime conditions, or *predicates*, associating it with behavior dynamically. We use the words *classify* and *classification* to denote both the identification of data with certain properties (i.e. the broader meaning) and the process of associating that data with certain pre-existing patterns (i.e. the OOP meaning). The idea is that of *predicate collection classes* (PCCs, from here on), those whose extents are automatically determined by a predicate, rather than being explicitly manipulated, and whose behavior is given by [OOP] classes. Such expression mechanisms are extremely convenient, especially in long-lived, complex applications where the data changes dynamically and may need to be reclassified often, such as simulations [40] and the processing of streaming data [78].

As a motivating example, consider two functionally independent, but data dependent, simulations: a traffic simulation and a pedestrian simulation. The traffic simulation moves cars along roads, performing collision avoidance between them, using a rich model of cars and roads. The pedestrian simulation moves pedestrians in sidewalks who can, at points, cross roads. The process of detecting possible collisions between cars and pedestrians establishes a data dependency between the two otherwise independent simulations: information about cars must flow, directly or indirectly, to the pedestrian simulation, or vice-versa.

There are several alternatives to model this situation. One option is for the car simulation to import/access, at every simulation step, the entire collection of pedestrians from the pedestrian simulation; another option is for data to flow the other way around. None is these options is ideal, for two reasons: (1) the vast majority of cars and pedestrians are not at risk of colliding, so it will be wasteful to replicate/access them all when only a *subset* of them is of interest; and (2) the information that the car simulation needs of pedestrian objects, or

that the pedestrian simulation needs of car objects, is only a small portion (i.e., a *projection*) of the potentially rich data models used by the respective simulations – essentially only their positions are needed. A better alternative is for one of the simulations to only import/access portions of the other simulation's objects that are within a certain range of its own objects. But, in this case, an even better alternative is to not import or access the other simulation's objects at all, and, instead, rely on an external substrate of all data that can help model and classify the internal data more expressively. For example, the pedestrian simulation can model the concept of *pedestrian in danger*, which corresponds to the subset of its own pedestrians that are at a certain distance of any car in the other simulation. In doing so, no car flows explicitly into the pedestrian simulation; the collection of *pedestrians in danger* is simply a subset of the existing pedestrians predicated on state that exists on another collection elsewhere. Pedestrians in danger behave differently than all other pedestrians, for example, they may move faster or stop. Once they move out of danger ranges, they become regular pedestrians again.

```
1  pclass PedestrianInDanger(Pedestrian pedestrian, List<Car> cars) :
2      predicate : :
3          foreach c ∈ cars :
4              if pedestrian.near(c) :
5                  return True
6              end
7          end
8          return False
9      end
10     method avoid() :
11         // Move the pedestrian out of the road!
12     end
13 end
```

Algorithm 1: Potential Pedestrian and Car collision avoidance code.

The pseudo-code in Algorithm 1 sketches the main idea of what we want to achieve: we want to create and classify collections of objects automatically from other collections. In this case we want the collection class `PedestrianInDanger` to contain `Pedestrian` objects predicated

on a given list of `Car` objects, and we want this predicate collection class to have its own specific methods – in this case, method `avoid`.

Proposals for how to achieve similar goals can be traced to the early days of OOP, and include predicate classes [25], multiple most specific classes [11], dynamic reclassification of objects [32], and dependent classes [41]. We take a fresh look at this idea, placing it in the context of modern applications and frameworks. Two characteristics make our approach different from what has been proposed before: (1) our focus on *collections* of objects (predicate classes), rather than on construction of individual instances (attributive classes); and (2) our approach to handling object state changes.

In languages with mutable state, PCCs pose many challenges, some of which are critical to both the semantics and the implementation of the concept. An object that is classified as an instance of some predicate class C at one point in time, may see its internal predicate become invalid some time later. In the example above, a call to the method `avoid` (line 10) may change the position of the pedestrian in danger, violating the property that made that object exist in the first place. From that point on, it is unclear what should happen to that object. Should it cease to exist? Should it continue to exist but in a zombie state? Should it continue to exist within a certain scope, but with the understanding that the predicate may be invalid? We chose this latter option, as it is the closest to the semantics of modern collection classes.

8.3 Predicate Collection Classes: Overview

This section gives an overview of the main design elements of PCCs using as an example the simplest of all operations, subsetting. Given that our first implementation of PCCs is in Python, the examples are written in Python. The PCC capabilities are embedded without

Decorator	**Target**	**Origin**
@subset($Class$)	class	PCC
@projection($Class, Field_1, ..., Field_m$)	class	PCC
@join($Class_1, ..., Class_N$)	class	PCC
@union($Class_1, ..., Class_N$)	class	PCC
@intersection($Class_1, ..., Class_N$)	class	PCC
@parameter($Class, mode$)	class	PCC
@dimension($Type$)	property	PCC
@$Property$.setter	property	Python
@staticmethod	method	Python

Table 8.1: Summary of decorators used by PCCs.

changing the language, using our own decorators as well as some pre-existing ones. Table 8.1 summarizes these decorators, the elements they apply to, and whether they are our own or Python's.

In general, PCCs are collections of objects created from one or more collections of objects of certain classes or types, for which certain predicates hold. A PCC defines both a class and a collection of instances of that class. Listing 8.1 shows one example where the PCC ActiveCar is defined as a subset of instances of the regular class Car.

This first example lends itself to a few observations, all of which are applicable to all PCCs, not just subsets.

- The data of the elements in these collections is given by properties tagged as **@dimension** (see lines 2, 5, 8).[1] The set operations are established by declarations immediately above the class declarations, in this case **@subset(Car)** (line 13). Other operations will be introduced in the next section.

- Besides the operation declaration (in this case, Subset), the other user specification pertaining to PCC is the predicate, **___predicate___**, a static method, which, in this case, establishes that a Car c is active if its Velocity field is neither the zero vector nor

[1]For simplicity sake, we omit the setter methods in these examples, but they need to exist if the values of properties are to be changed.

```
1  class Car:
2    @dimension(int)
3    def ID(self): return self._ID
4
5    @dimension(list)
6    def Position(self): return self._Position
7
8    @dimension(list)
9    def Velocity(self): return self._Velocity
10
11   #...methods of Car...
12
13 @subset(Car)
14 class ActiveCar:
15   @staticmethod
16   def __predicate__(car):
17     return  not (car.Velocity == [0,0] or car.Velocity == None)
18
19   def Move(self):
20     self.Position[0] += self.Velocity[0]
21     self.Position[1] += self.Velocity[1]
22     if self.Position == (100, 100): self.Velocity = None
23
24 cars = []
25 # ...fill in the list of cars...
26 while (True):
27   foreach aCar in pcc.create(ActiveCar, cars) as aCars:
28     aCar.Move()
```

Listing 8.1: Subsetting

None (lines 16–17). The arguments for predicate depend on the operation declaration, and this is enforced by the PCC language processor; in the case of subsets, there is only one argument, one whose type is that of the subsetting declaration above (in this case, Car). This allows the programmer to define the filter for determining which elements of the original set are to be selected.

- PCCs construct instances of their type, not of the type of their supersets. In this case, the ActiveCar PCC will create instances of ActiveCar, not of Car. In other words, PCCs are not just about selecting objects from lists and returning the objects that honor in invariant; for each object that honors the predicate, a new instance of the PCC class will be created.

- PCCs can define their own fields, properties and methods. In the example above, the PCC ActiveCar defines a method Move() (lines 19–22) that doesn't necessarily exist in class Car (if it exists, it is an unrelated method). In this sense, PCCs are regular OOP classes, with their own behavior that is independent of the original objects' classes.

- PCCs acquire dimensions (i.e. properties) defined in the original classes of their supersets. In this case, ActiveCar acquires all the dimensions of class Car; that can be seen in the body of method Move, which refers to the fields Velocity and Position defined in class Car (lines 19–22). This mechanism is not inheritance, even though for subset operations, in particular, the dimension acquisition semantics is very similar to inheritance. For other operations, the differences will become much clearer in the following sections.

- Lines 24–27 show how PCCs are created and used: a function pcc.create takes a PCC class name (in this case ActiveCar) and a collection of objects from which to create reclassified instances, and returns those new instances.

The next two sections cover object reclassification and the different kinds of PCCs, respectively, in more detail.

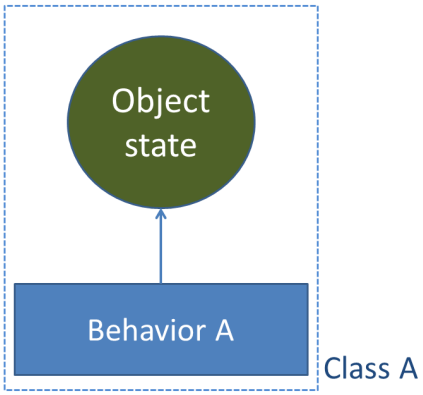


Figure 8.1: Object model: an instance of a class is a combination of state and behavior (methods). Different behavior may be dynamically associated with the same instance state.

8.4 Object Reclassification

In order to explain object reclassification, we will use the example of the previous section involving collections of cars and subsets of active cars (see Listing 8.1). We now focus on the bottom part of that snippet, the infinite loop (lines 26–28). Within that loop, in each iteration, a collection of reclassified objects is created given a list of cars (line 27). The collection aCars is a subset of the collection of cars that honor the predicate defined in ActiveCar. For every active car in this collection, the method Move is invoked (line 28); that method, which does not exist in class Car, changes the active car's position (lines 19–22), and may change its velocity to None (line 22). Those changes may or may nor be propagated to the original car instances, depending on whether copy or reference semantics is used; both semantics will be explained next. In either case, in the next iteration, the next reclassification will get a new collection of active cars.

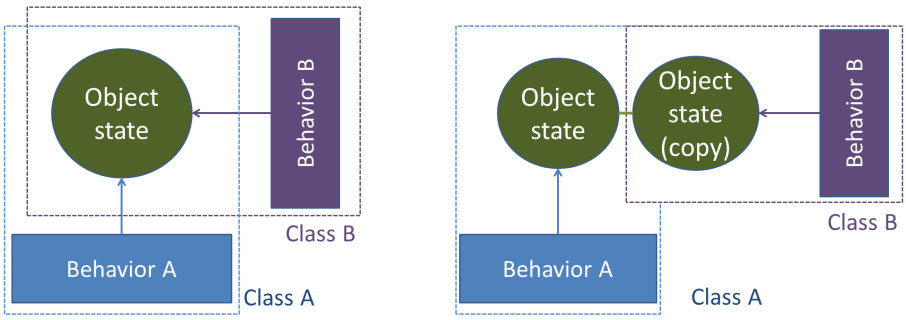


Figure 8.2: PCC creation by reference (left) vs. by copy (right).

8.4.1 Object Model and Reclassification

Figure 8.1 illustrates the required object model upon which PCCs can be defined. The creation of a class instance, for example aCar = Car(), results in two components being created: a set of fields that hold the object's state and a set of methods that access that state. The object state exists independently of the methods, and is a first-class object in itself – i.e. it can be referenced.

Several object-oriented languages provide this object model, or some version of it; Python is one of them. Other languages such as Java and C# do not expose the object's state as first-class, but the set of fields and properties of the object can be accessed via reflection, which make it possible for PCCs to be implemented in those languages too.

PCC objects go through a process of reclassification, which means that they are instances of different classes than their originals. In the Cars example (8.1), in line 29, the car instances that honor the predicate (line 17) will be available as instances of ActiveCar. The relation between the reclassified instance and its original depends on whether a reference or copy semantics is used, which is the topic of the next subsection.

8.4.2 Reference vs. Copy Semantics

There are two mechanisms by which reclassification can be achieved: by referencing the state of the original object directly, or by copying data between the original and the reclassified instance. Figure 8.2 illustrates the two semantics. Both are possible, and we have experimented with both, but ultimately, we decided to adopt a reference semantics for the core of PCCs (Figure 8.2, left). Nevertheless, here, we discuss the two possibilities, their strengths and weaknesses, and the context where copy semantics is required.

If the state is used by reference, the PCC object and the original have different types (and therefore different interfaces) but point to the same object state. State changes can be done via any of the interfaces. In Python, we achieve reference semantics between the state of different instances of different classes simply by setting the ___dict___ field of reclassified instances to that of their original counterparts.

If the reclassification is done with a copy semantics, then the creation of a PCC involves making deep copies of the objects' state at that point in time. From then on, the state of the PCC instance becomes independent of original. In order to prevent copying more than necessary, only attributes marked as dimensions are copied. Additionally, dimensions that are object references are followed and copied recursively. The copying process keeps track of which objects have been ingested, so that no two copies of the same object are included.

Reference semantics provides true multi-classification of objects, which makes many problems easier to model. It also has much better performance than copy semantics, since no memory is copied when creating PCCs. As such, even though we experimented with both, the current implementation of PCCs supports only reference semantics.

In multi-threaded applications, the reference semantics suffers from the general problems associated with the lack of isolation. In programs where changes to the reclassified objects

do not need to be reflected back to the originals, copy semantics might be preferred and that is a drawback of our choice to support only reference semantics.

Copy semantics can be achieved with an additional layer on top of PCCs. For example, in one of our main applications of PCCs involving distributed simulations, we designed and implemented a layer on top of PCCs that serializes and deserializes PCCs, and that merges changes made to PCC objects back with the state of their originals. We call this extra layer a **dataframe**. Dataframes could also be implemented in a non-distributed setting, but, so far, we have no strong justification for doing so, since all non-distributed implementations of PCCs can be done much more easily with reference semantics. While interesting in its own right, especially with respect to merging changes, the dataframe layer is not part of PCCs, and hence it is not described here.

8.4.3 Constant Consistency vs. One-Time Consistency

As the example shows, PCC instances can, at points, be in violation of the predicate that classified them and placed them in the resulting collection. In this example, when active cars reach the final point, their Velocity is set to None (Listing 8.1, line 22). This, however, does not remove them from the collection, or from the class – ActiveCar – even though from that point on they are in violation of the predicate.

This is a feature, not a flaw, of the design of PCCs. PCCs are designed to be used in iterative computations. As such, at each step, we want stability of the data in the collections. As such, we support **one-time consistency**. The semantics of one-time consistency is as follows: **at the time of creation of a PCC**, before any processing occurs, the state of the instances that make it to the PCC is guaranteed to be consistent with the predicate of the PCC. Once the objects start being processed, however, there are no guarantees of consistency with the predicate that placed them there.

While this choice of semantics may at first seem strange, we note that a similar choice has also been done in mainstream collection libraries. Consider, for example, the iteration over a hash map/dictionary; in most modern programming languages (Java, Python, C#, etc.) the collection cannot be changed during iteration, even if some elements of the collection logically stop belonging to it or new elements logically become members of it. The same happens in sorted collections based on the objects' hashcode, or comparator method: given that the hashcodes and comparisons are based on the objects' state, when that state changes after the original insertion, the position of the object in the collection may be temporarily inconsistent until explicit adjustments are made.

An alternative to this design choice would be to keep strict constant consistency of the objects with the PCC predicates, and to move objects in and out of collections immediately as their state changes. Constant consistency would make certain instances be reclassified in, and even disappear from, the collection before the processing step would be over. While feasible, this choice would be considerably more complex to implement, slow, and, most importantly, potentially confusing. We decided to keep with the design choices of mainstream collection libraries.

8.4.4 Inheritance

The relational operations underlying PCCs, of which subsetting is only one, have interesting implications for the traditional concept of inheritance in OOP. Even before presenting the other operations, it is important to clarify the relation between PCCs and inheritance; subsetting serves as a good illustration of the subtleties of this relation. More subtleties will emerge when we present other PCCs. The overall design principle is that PCCs can use inheritance, but do not try to reappropriate it. When inheritance is desired, programmers can complement the PCC declaration with the inheritance features provided by the host

language.

In the example of Listing 8.1, the PCC ActiveCar is a subset of Car, but does not inherit from the class Car. Because there is no inheritance relation, ActiveCar is not a subtype of Car, nor does it inherit any method from Car. The only elements reused from Car by ActiveCar are the fields explicitly marked as dimensions, in this case ID, Position, and Velocity. The lack of implicit inheritance is intentional: it may very well be that programmers want completely different behavior for ActiveCar objects than that present in Car. When behavioral subtyping is desired, the programmer can add it in the usual manner. In the example, if subtyping had been desired, we would have declared ActiveCar as:

```
@subset(Car)
class ActiveCar(Car):
...
```

In this case, ActiveCar would be a subclass of Car, and all methods and fields would be inherited.

8.5 Relational Operations of PCCs

A PCC is both a class of objects and a specification for collection of objects of that class. As such, it is defined by both a dimensions rule Γ (Gamma) and an extension rule Ψ (Psi). The predicate collections result from relational operations on collections of existing objects. This section revisits subsets and presents the other main relational operations: projection, cross product (join), union, intersection, and parameterization of collections. A summary of these rules is presented in Tables 8.2 and 8.3. This section contains the explanation of these rules.

	Γ **(Fields Rule)**
Subset $\text{Sub}_P(A)$	$\gamma(Sub_P(A)) \supseteq \gamma(A)$
Projection $\text{Proj}(A : f_1, ..., f_m)$	$\gamma(Proj(A : f_1, ..., f_m)) \supseteq \{f_1, ..., f_m\}$
Join $A_1 \times_P A_2$	$\gamma(A_1 \times_P A_2) \supseteq \gamma(A_1) \cup \gamma(A_2)$
Union $A_1 \cup A_2$	$\gamma(A_1 \cup A_2) \supseteq \gamma(A_1) \cap \gamma(A_2)$
Intersection $A_1 \cap A_2$	$\gamma(A_1 \cap A_2) \supseteq \gamma(A_1) \cap \gamma(A_2)$

Table 8.2: Dimensions rules (Γ) for PCCs, which dictate the fields of PCCs.

	Ψ **(Extension Rule)**
Subset $\text{Sub}_P(A)$	$\psi(Sub_P(A)) = \{a \in \psi(A) \mid P(a)\}$
Projection $\text{Proj}(A : f_1, ..., f_n)$	$\psi(Proj(A : f_1, ..., f_m)) = \psi(A)$
Join $A_1 \times_P A_2$	$\psi(A_1 \times_P A_2) = \{(a_1, a_2) \in \psi(A_1) \times \psi(A_2) \mid P(a_1, a_2)\}$
Union $A_1 \cup A_2$	$\psi(A_1 \cup A_2) = \{a \in \psi(A_1) \cup \psi(A_2)\}$
Intersection $A_1 \cap A_2$	$\psi(A_1 \cap A_2) = \{a \in \psi(A_1) \cap \psi(A_2)\}$

Table 8.3: Extension rules (Ψ) for PCCs, which define which instances are in the PCC.

```
1  class Person:
2    @dimension(int)
3    def ID(self): return self._ID
4    @dimension(str)
5    def Name(self): return self._Name
6    # plus 20 other dimensions
7
8  @projection(Person, Person.ID, Person.Name)
9  class PersonInfo:
10   def PrintSummary(self):
11     print ''ID='' + str(self.ID) + '' Name='' + self.Name
12
13   @Name.setter
14   def Name(self, value): self._Name = value
15   # More fields and methods for PersonInfo
```

Listing 8.2: Projection

8.5.1 Subset

Listing 8.1 presented an example of a subset PCC. More formally, given a collection A of objects of class A, a subset PCC of A for a certain predicate P, written $Sub_P(A)$, which is also a class, is ruled by the following:

Γ **Rule:** $Sub_P(A)$, as a class, includes all dimensions of class A, and can include additional dimensions of its own.
Ψ **Rule:** $Sub_P(A)$, as a set, consists of instances of class $Sub_P(A)$ constructed from instances in A that honor the given predicate P.

8.5.2 Projection

Given a collection A of objects of class A, a projection PCC of A over a subset of dimensions of A, written $Proj(A : f_1...f_m)$, is defined by the following:

```
1  @join(Person, Card, Transaction)
2  class RedAlert:
3    def __init__(self, p, c, t):
4      self.p = p
5      self.c = c
6      self.t = t
7
8    @staticmethod
9    def __predicate__(p, c, t):
10       p.id==c.owner and t.card==c.id and
11       t.amount > 2000 and t.holdstate==False
12
13   # Dimensions of Person, Card and Transaction are available
14   # For example:
15   def Protect(self):
16     self.c.holdstate = True
```

Listing 8.3: Cross Product (Join)

Γ **Rule:** $Proj(A : f_1...f_m)$, as a class, includes the dimensions of class A onto which it is being projected, $f_1...f_m$, and can include additional dimensions of its own.
Ψ **Rule:** $Proj(A : f_1...f_m)$, as a set, consists of instances of class $Proj(A : f_1...f_m)$ constructed from all instances in A.

Listing 8.2 shows an example where a class Person, with multiple dimensions, is projected as PCC PersonInfo, in which the objects have only two of the dimensions of Person (line 8). PersonInfo can be tasked with methods not available to the objects of the class Person. For example, PersonInfo can have exclusive set access to Name (line 13). This means that any function that uses Person will not be able to change the Name, until the projection PersonInfo is used. By making this explicit we are enforcing a certain protocol for data access that is much more expressive than simple accessibility qualifiers.

8.5.3 Cross Product (Join)

Given a collection A of objects of class A, and a collection B of objects of class B, the join PCC of A and B under a certain given predicate P, written $(A \times_P B)$, is ruled by as:

```
1  class Fruit:
2    @dimension(float)
3    def size(self): return self._size
4    ...
5
6  class Lemon(Fruit): ...
7  class Orange(Fruit): ...
8  class Banana(Fruit) ...
9
10 @union(Lemon, Orange)
11 class Citrus(Fruit):
12   def MakeJuice(): return size/50
13   # more fields and methods
```

Listing 8.4: Union

Γ **Rule:** $A \times_P B$, as a class, includes all dimensions of class A and all dimensions of class B, either directly as the same dimensions or possibly indirectly via instances of A and B as fields, and can include additional dimensions of its own.
Ψ **Rule:** $A \times_P B$, as a set, consists of instances of class $A \times B$ constructed from instance pairs $(a \in A, b \in B)$ that honor the given predicate P.

The cross product of two sets is defined as the set of all combinations of elements of the first set and elements of the second, possibly with some additional constraints. In Listing 8.3, the declaration **@join(Person, Card, Transaction)** establishes that RedAlert is a cross product operation between instances of those three classes. The predicate P, in this case, establishes some constraints regarding identifiers, amounts and status of the transactions.

8.5.4 Union

Given a collection A of objects of class A, and a collection B of objects of class B, the union PCC of A and B, written $(A \cup B)$, which is also a class, is ruled by the following:

Γ **Rule:** $A \cup B$, as a class, includes all dimensions resulting from the intersection of the dimensions of class A and class B, and can include additional dimensions of its own.
Ψ **Rule:** $A \cup B$, as a set, consists of instances of class $A \cup B$ constructed from all instances $a \in A$ and $b \in B$.

While it is possible to define the union of two sets with elements of the same type (e.g. List<Car> c1 ∪ List<Car> c2), it is more interesting to add sets with elements of different types, producing a union that can have meaningful semantic differences from the original sets. Listing 8.4 shows one such example, where the PCC Citrus defines a set containing all instances of Lemon and Orange, but not of Banana.

As the example illustrates, Unions are an expression a mechanism that can be seen as post-hoc inheritance, i.e. commonalities that are established later rather being modeled in. In the example of Listing 8.4, using traditional inheritance, a Citrus class would be defined as inheriting from Fruit, and then classes Lemon and Orange would inherit from Citrus. With the union PCC, such strict hierarchies are not necessary, as new unions can be added relating existing classes to each other externally to their inheritance definitions.

Dimension compatibility in unions is determined by their names. The less fields the elements have in common, the less fields will be available for the methods of the union PCC to use. In the extreme, when the only thing in common between the elements is that they are objects, no fields are available in the union PCC; but the union is still valid.[2]

Because PCCs relate to their domain sets structurally, the union of sets applies to any sets, not just those whose elements have a common user-defined super type, as in the example. The following code shows the union of two sets that have elements with no type in common, other than Object, but that have one field in common, `size`:

[2]Many SQL engines support attribute renaming in order to enrich the expression of unions and intersections. For the time being, our model and language does not support field renaming, although it can easily be added in the future.

```
class Car:
  @dimension(float)
  def size(self): return self._size

class Lemon:
  @dimension(float)
  def size(self): return self._size

@union(Lemon, Car)
class Prize:
  def Box(): return size + 10


class Car: ...

@subset(Car)
class ActiveCar(Car): ...

@subset(Car)
class RedCar(Car):...

@union(ActiveCar, RedCar)
class RedOrActive(Car): ...
```

In cases where the two sets have some overlap, the union, by default, will include only one of the duplicate objects (so, the DISTINCT semantics of union in SQL). For example:

In this example, there may be cars that are both active and red. By default, those objects appear only once in the resulting collection RedOrActive.

8.5.5 Intersection

Given a collection A of objects of class A, and a collection B of objects of class B, the intersection PCC of A and B, written $(A \cap B)$, which is also a class, is ruled by the following:

```
1  class Car: ...
2
3  @subset(Car)
4  class ActiveCar(Car): ...
5
6  @subset(Car)
7  class RedCar(Car): ...
8
9  @intersection(ActiveCar, RedCar)
10 class RedActiveCar(Car): ...
```

Listing 8.5: Intersection

Γ **Rule:** $A \cap B$, as a class, includes all dimensions resulting from the intersection of the dimensions of class A and class B, and can include additional dimensions of its own. (Similar to union)
Ψ **Rule:** $A \cap B$, as a set, consists of instances of class PCC_I constructed from all instances that belong to the intersection of A and B.

As stated above, the Γ rule is the same as for unions. This requires some explanation, as it may be surprising. Unions and intersections must operate on structurally compatible objects, hence the intersection of the fields in both cases, which gives the minimum common structure. The differences are in the extension (Ψ) rule, i.e. the objects that end up in the resulting set. Our design decision for unions and intersections follows that of SQL's unions and intersections.

Listing 8.5 shows an example where the set of active cars is intersected with the set of red cars, resulting in a set of cars that are both active and red.

Although the concept of an object being an instance of ActiveCar and of RedCar simultaneously may seem strange, this is explained by the fact that PCCs create objects that entangle with their originals: a specific car that is both active and red will have an incarnation as ActiveCar and as RedCar; but, due to entanglement, object identification is never lost. Therefore, two PCC instances derived from the same original object pass the equality test.

```
class Node(object):
  @dimension(int)
  def id(self): return self._id
  ...
class Edge(object):
  @dimension(Node)
  def start(self): return self._start
  @dimension(Node)
  def end(self): return self._end
  ...

@parameter(Node, mode=Singleton)
@subset(Edge)
class InEdge(Edge):
  @staticmethod
  def __predicate__(e, n): return e.end.id == n.id
```

Listing 8.6: Parameterized subset with single parameter

8.5.6 Parameterized Collections

Additionally to the basic algebraic operations on collections, we also support parameterization of queries when constructing PCCs. Listing 8.6 shows one example in the domain of graphs (nodes and edges). The PCC collection InEdge is defined as a parameterized subset. InEdge is a subset of Edge. However, the subset cannot be instantiated without providing specific run-time context: the Node to which the subset collection of Edges are incident upon. Node is therefore the parameter. A Node has to be passed during the creation of the InEdge collection for it to be successful.

Parameters can be also collections of objects. Listing 8.7 illustrates this with the concept of a pedestrian in danger of being hit by a car. The parameter is a collection (list) of cars.

```
@parameter(Car, mode=Collection)
@subset(Pedestrian)
class PedestrianInDanger(Pedestrian):
  @staticmethod
  def __predicate__(p, cars):
    for c in cars:
      if abs(c.Position.X - p.X) < 130 and c.Position.Y == p.Y:
          return True
    return False
```

Listing 8.7: Parameterized subset with a list as parameter

8.5.7 Final Note on PCC Creation

This section showed the different relational operations that underlie the different kinds of PCCs, but it didn't yet cover how these PCCs are constructed from existing collections of objects. As briefly mentioned in Section 8.3, PCCs are created with the `pcc.create` function – see Listing 8.1, lines 23–27. This function takes a PCC name (the class to be constructed), one or more lists of objects from which to select the members of the PCC, and possibly additional parameters. For completeness' sake, the following shows examples of how to obtain the different PCCs explained in this section.

Type	Listing	Creation example
Subset	1	cars = *list of cars* acars = pcc.create(ActiveCar, cars)
Projection	2	persons = *list of persons* pinfos = pcc.create(PersonInfo, persons)
Join	3	persons, cards, trans = *lists of persons, cards, and transactions* ralert = pcc.create(RedAlert, persons, cards, trans)
Union	4	fruits = *list of several types of fruits* citrus = pcc.create(Citrus, fruits)
Intersection	5	cars = *list of cars* acars = pcc.create(ActiveCar, cars) rcars = pcc.create(RedCar, cars) racars = cc.create(RedActiveCar, acars, rcars)
Parameterization	6	edges = *list of edges*; node = Node() inedges = pcc.create(InEdge, edges, params=(node,))
Parameterization	7	peds = *list of pedestrians*; cars = *list of cars* pdanger = pcc.create(PedestrianInDanger,peds,params=(cars,))

8.6 Usage Examples

We have used PCC in both small algorithms and as a component of Spacetime. This section illustrates the expressiveness of PCCs using four of those examples. All of these examples are available from the PCC repository in Github (See `https://github.com/Mondego/pcc`). Parts of the code are omitted, so that the code shown here can fit in one page. The indentation shown here deviates from the stylistic indentation of Python code, for the same reason. Please see the project repository for the complete code.

8.6.1 K-Nearest Neighbor

Listings 8.8 and 8.9 give the PCC implementation of the K Nearest Neighbor algorithm as applied to the clustering of flowers.[3] The goal here is to identify the type of a flower using four characteristics of flowers: the width and length of the sepal and petals. The example code is shown in two parts, the data model (Listing 8.8) and the procedural part that uses it (Listing 8.9).

The main class in this example is flower (Listing 8.8, lines 1–15). A parameterized subset knn is defined (Listing 8.8, lines 17–36), whose purpose is to model the K nearest neighbors of any given flower. As such, it requires two parameters: a flower, and the number of nearest neighbors to be selected. A few facts are interesting about the knn PCC:

- The knn PCC does not inherit from flower, but it is a subset of flower collections. As such, all dimensions of the flower class are available in knn instances. Methods, however, are not inherited.
- The ___query___ defined in the knn PCC is tasked with sorting the training set of flowers using the Euclidean distance with the characteristics of the given flower as dimensions (Listing 8.8, lines 21–25). It selects and returns the nearest K flowers. The default ___query___ cannot be used to create this subset, because the final subset is a limited collection that depends on sorting order. Additional controls like ___order_by___ and ___limit___ would be needed in order to provide this functionality, something we still do not support. Advanced queries such as this one can be directly defined by the programmer.
- The knn PCC defines an additional method vote (Listing 8.8, line 36), which returns the label of the flower as neighbor of some other flower. In other words, voting is not a

[3] Adapted from `http://machinelearningmastery.com/tutorial-to-implement-k-nearest-neighbors-in-python-from-scratch/`.

```
class flower(object):
      @dimension(float)
      def sepal_length(self): return self._sepal_length
      @dimension(float)
      def sepal_width(self): return self._sepal_width
      @dimension(float)
      def petal_length(self): return self._petal_length
      @dimension(float)
      def petal_width(self): return self._petal_width
      @dimension(str)
      def fl_type(self): return self._fl_type
      @dimension(str)
      def predicted_type(self): return self._predicted_type
      def __init__(self, sl, sw, pl, pw, tp):
         # initialize fields

@parameter(flower, int)
@subset(flower)
class knn(object):
    @staticmethod
    def euclideanDistance(fl1, fl2):
        return math.sqrt(pow((fl1.sepal_length - fl2.sepal_length),2)
                         + pow((fl1.sepal_width - fl2.sepal_width),2)
                         + pow((fl1.petal_length - fl2.petal_length),2)
                         + pow((fl1.petal_width - fl2.petal_width),2))
    @staticmethod
    def __query__(training_flowers, test, k):
        final_items = sorted([tr_f for tr_f in training_flowers],
                             key=lambda x:knn.euclideanDistance(test,x))
        return [final_items[i]
                for i in range(len(final_items))
                if knn.__predicate__(i, k)]
    @staticmethod
    def __predicate__(index, k): return index < k

    def vote(self): return self.fl_type
```

Listing 8.8: K Nearest Neighbor – data model

```
1  def getResponse(knns):
2      classVotes = {}
3      for one_neighbour in knns:
4          response = one_neighbour.vote()
5          classVotes[response] = classVotes.setdefault(response, 0) + 1
6      sortedVotes = sorted(
7          classVotes.iteritems(),key=lambda x:x[1],reverse=True)
8      return sortedVotes[0][0]
9
10 def main():
11     trainingSet, testSet, predictions = [], [], []
12     split, k = 0.67, 3
13     loadDataset("iris.data", split, trainingSet, testSet)
14     for one_flower in testSet:
15         with pcc.create(knn,trainingSet,params=(one_flower,k)) as knns:
16             one_flower.predicted_type = getResponse(knns)
17     print('Accuracy:' + repr(getAccuracy(testSet)) + '%')
18
19 main()
```

Listing 8.9: K Nearest Neighbor – algorithm

general behavior of flowers; it's only a behavior of flowers that are K-distance similar to some other flower.

Listing 8.9 shows the procedural part. After loading the data and dividing it into a training and test set (Listing 8.9, line 13), we iterate over every flower that has to be labeled (Listing 8.9, lines 14–16). For each of those flowers, we create the corresponding knn subset with the flower to be labeled and the number of neighbors to look for as the parameters (Listing 8.9, line 15). We then decide what type of flower this is by polling its nearest neighbors for votes (Listing 8.9, line 16). In the getResponses function, each of the neighbors votes on a label (line 4), and the majority wins (lines 6–8).

8.6.2 Car and Pedestrian

We also used PCCs to develop proof-of-concept simulations. The simulation example we worked on involved cars and pedestrians moving towards each other. When a pedestrian is in danger of colliding with a car, it gets out of the way. In this simulation, PCCs were used to drive the state changes for cars and pedestrians, rather than being used to calculate intermediate steps like in the previous example. PCCs are both creators and consumers of the state change.

Listing 8.10 shows the base classes that were used for this example. Car and Pedestrians have their own classes (lines 1–17 and lines 19–45 respectively). Both classes are derived from the Sprite class in pygame, a module that was used to render the car and pedestrians in the simulation viewer. The attributes required for visualization are not registered as dimensions, and so they will not be copied when PCCs are created from Car and Pedestrian.

Listing 8.11 details the mechanism needed to move a car. We track two states: InactiveCar (defined in lines 1–9) and ActiveCar (defined in lines 11–21). An InactiveCar is one that has its Velocity dimension set to zero. ActiveCar is naturally the opposite. Both are Subsets of Car, but they do not inherit from Car. This means that they get the dimensions from Car (as they are declared as Subsets) but they do not inherit the methods or the non dimension attributes from Car. This is useful in this case as Car is also a sprite object and has properties and methods that are not needed.

ActiveCar and InactiveCar also define their own methods relevant to the state of the car that it models. An ActiveCar cannot be started, and an InactiveCar cannot be Moved. When an InactiveCar is started, in the next iteration it gets classified as an ActiveCar and gains the ability to move. This is done by two parallel threads, one that starts InActiveCars (lines 26–35) and one that moves ActiveCars (lines 37–44). All the pcc changes are done under the scope of the dataframe object. The dataframe object performs two tasks. First, it uses

```
class Car(pygame.sprite.Sprite):
    # The class that shows the image of a car.
    # Normal class that holds the state of a car.
    FINAL_POSITION = 500
    SPEED = 10
    @dimension(str)
    def ID(self):  return self._ID
    @dimension(tuple)
    def position(self):  return self._position
    @dimension(tuple)
    def velocity(self):  return self._velocity

    def __init__(self, position):
        # Constructor

    def update(self):
        # Changes the position of the car in the graphics window.

class Pedestrian(pygame.sprite.Sprite):
# base pedestrian class
    INITIAL_POSITION = (400, 0)
    SPEED = 10
    @dimension(str)
    def ID(self):  return self._ID
    @dimension(int)
    def X(self):  return self._X
    @dimension(int)
    def Y(self):  return self._Y

    def __init__(self):
        # Constructor

    def Move(self):
        self.X -= Pedestrian.SPEED
        if self.X <= 0:
            self.Stop()

    def Stop(self):
        self.X, self.Y = Pedestrian.INITIAL_POSITION

    def SetPosition(self, x):
        self.X = x

    def update(self):
        # updates the graphics with changes to state.
```

Listing 8.10: Normal classes needed for the Car and Pedestrian Simulation

```
1  @subset(Car)
2  class InactiveCar(object):
3      # Car that is not moving, Velocity is zero
4      @staticmethod
5      def __predicate__(c):
6          return c.velocity == (0, 0, 0) or c.velocity == None
7
8      def Start(self):
9          self.velocity = (Car.SPEED, 0, 0)
10
11 @subset(Car)
12 class ActiveCar(object):
13     # car that is moving, velocity is not zero
14     @staticmethod
15     def __predicate__(c):
16         return not (c.velocity == (0, 0, 0) or c.velocity == None)
17
18     def Move(self):
19         x,y,z = self.position
20         xvel, yvel, zvel = self.velocity
21         self.position = (x + xvel, y + yvel, z + zvel)
22
23     def Stop(self):
24         self.position, self.velocity = (0,0,0), (0,0,0)
25
26 def StartInactiveCars(cars, MainWindow):
27     # Starts inactive cars every 5 secs
28     while True:
29         with dataframe(carlock) as df:
30             iacs = df.add(InactiveCar, cars)
31             for car in iacs:
32                 car.Start()
33                 register(car.ID, cars, MainWindow)
34                 break
35         sleep(5)
36
37 def MoveActiveCars(cars, MainWindow):
38     # Moves active cars every 300 ms
39     while True:
40         with dataframe(carlock) as df:
41             acs = df.add(ActiveCar, cars)
42             for car in acs:
43                 car.Move()
44         sleep(0.3)
```

Listing 8.11: PCC classes and its usage needed to move the Car

```
@subset(Pedestrian)
class StoppedPedestrian(Pedestrian):
    # A person that is not moving
    @staticmethod
    def __predicate__(p):
        return p.X, p.Y == Pedestrian.INITIAL_POSITION

@subset(Pedestrian)
class Walker(Pedestrian):
    # A person who is walking.
    @staticmethod
    def __predicate__(p):
        return p.X, p.Y != Pedestrian.INITIAL_POSITION

@parameter(list)
@subset(Pedestrian)
class PedestrianInDanger(Pedestrian):
    # A person who is in danger of colliding with a car
    @staticmethod
    def __predicate__(p, cars):
        for c in cars:
            cx, cy, cz = c.position
            if cy == p.Y and abs(cx - p.X) < 70:
                return True
        return False

    def Move(self):
        self.Y += 50

def StartPedestrian(peds, MainWindow):
    # Make a stopped pedestrian walk every 3 secs.

def MovePedestrian(peds, cars, MainWindow):
    # Make a Walker move.
    while True:
        with dataframe(pedlock) as df:
            pids = df.add(PedestrianInDanger, peds, params = (cars,))
            wks = df.add(Walker, peds)
            for p in (pids + wks):
                pid.Move()
        _sleep(0.5)
```

Listing 8.12: PCC classes and its usage needed to move Pedestrians

a synchronization lock to make the computation within its scope thread safe. Second, it allows us to create PCC objects using the copy semantics, and provides a mechanism to copy changes performed on the copied object back to the original. This merge operation is performed at the end of its scope.

Listing 8.12 shows us the PCC classes, and the mechanisms needed to move pedestrians, and make them avoid danger when they are close to cars. StoppedPedestrian (lines 1–6), and Walker (lines 8–13) are similar to ActiveCar, and InactiveCar classes. They are subsets of Pedestrian that track the state of the pedestrian object. A StoppedPedestrian is one that has not moved from the initial position. A Walker is a pedestrian that has moved from the initial position. There are two threads that create and use these objects in the same way as cars. To make the walkers avoid cars, the PedestrianInDanger class (lines 15–28) was created. This is a parameterized subset of Pedestrians, that takes a list of Cars as a parameter. To shorten the search space, it can also be made a subset of Walker, and parameterized on a list of ActiveCar. The Move function in the PedestrianInDanger class overrides the Move in the Pedestrian class. So when an object gets classified as a PedestrianInDanger, the Move that is executed is different, and it will avoid the car (lines 37–40).

8.6.3 Distributed Simulations using Spacetime

The previous example is a proof-of-concept of the much more complex usage within Spacetime. This is, by far, our most meaningful application of PCCs.

Background: Modeling and Simulation

Modeling and simulation is a mature field in both research and development. When studying some real-world process or activity, one starts with developing a model for it, typically a

Figure 8.3: Urban simulation.

mathematical model of some kind that abstracts away unnecessary complexity; then, when the model does not have a closed form solution, computer simulation can be used to study its characteristics (behavioral and performance). This approach to studying the real world is used in almost all branches of Science and Engineering.

Microscopic model simulations can capture richer real world scenarios than macroscopic ones, but they require much greater computational resources. The finer the level of granularity and the larger the number of individual units modeled, the more resources they need. For that reason, these simulations tend to be limited in size and/or purpose. There is no easy way of designing and implementing multi-purpose simulations; each study requires not just its own separate model, something that would be expected, but also a separate simulation.

Over the past few years, we have been involved in simulation projects that challenge this state of affairs – see picture in Figure 8.3 showing one of the 3D simulated cities. These projects have evolved towards multi-purpose microscopic simulations of urban areas. It has become clear that there are lines of expertise for the different subsystems, and that, for that reason alone, we need a decentralized simulation architecture allowing different groups to provide relatively independent simulations of their models, but in a coordinated way, because the models are mutually interdependent.

PCCs were designed to address this application domain, and its need to support separation of concerns at the systems design level.

8.7 Related Work

Since the early days of OOP, the simple use of classes and single inheritance has been questioned. From multiple inheritance to traits to predicate classes (and dispatch), many alternatives have been proposed over the years. Here we cover some of the work most related to PCCs.

8.7.1 Predicate Classes

Some of the inspiration for PCCs, including the name, comes from predicate classes [25]. The idea behind predicate classes is for objects to be dynamically classified, taking new/different behavior as they change state. Objects that satisfy predicates specified in predicate classes automatically become instances of those classes; when they stop satisfying those predicates, they stop being instances of those classes. This is very similar to PCCs, but there are some important differences.

The major difference is that PCCs, as the name indicates, pertain to collections of objects, rather than to individual objects. This difference changes the focus and the capabilities of the basic idea substantially. In the case of simple predicate classes, the programmer simply states the predicate to be satisfied (e.g. a car whose velocity is zero), but there are no handles for collections of objects that satisfy those predicates at any point in time. The PCCs' focus on collections not only exposes these handles but also enables the full expressive power of relational operations on collections, such as subsetting, projection, cross product, etc. Simple predicate classes express implicit subsets only: subsets, because predicates on field values constrain the state of the parent objects; implicit, because there is no handle for that subset.

More importantly, one of the goals underlying simple predicate classes is to always ensure the satisfiablity of the predicate. For example, given a normal class buffer and a predicate class

empty buffer, if the last element is removed from a buffer object, that object immediately[4] becomes an instance of the empty buffer predicate class; similarly, if an element is added to an empty buffer object, that object immediately stops being an instance of the empty buffer predicate class. Classification is always consistent with the state of the objects. That is not a goal of PCCs. PCCs classify the objects at some point in time, at which point the predicate is guaranteed to be satisfied; but once included in the collection, the state of the objects may later change, possibly becoming inconsistent with the predicate that placed them in the data frame. That is not just acceptable: it is a desired semantics. PCCs are meant to hold a fixed collection of objects whose state can change. Take, for example, the case of a collection of non-empty buffer objects; if during subsequent processing all elements are removed from a given buffer object in that collection, we still want that buffer object to be in the collection, even though it is empty; we don't want it to suddenly disappear because it became empty. So the semantics of predicates in PCCs is quite different from that of predicates in simple predicate classes: always true (in the case of simple predicate classes) vs. true at the time of data frame creation (in the case of PCCs).

The relaxation of satisfiability is also what makes it possible to implement relational operations in practice, not just subsetting. Unlike subsetting, that looks only at internal state of the objects, joins (cross product) and parameterization pertain to combinations of objects. Take for example, a join between cars and their owners. If at some point in the framed computation the car ownership changes from one person to another (or to no one), we would need to search combinations of a car and persons again to check whether the resulting join object satisfies the predicate. Strict satisfiability of predicates for objects involved in join operations, as well as parameterizations, would be prohibitive to implement.

[4] "Immediately" here includes lazyness, i.e. not necessarily instantly but as soon as classification is needed.

8.7.2 Other Class-Instance Associations

Besides predicate classes, the OOP literature presents a considerable number of ideas aimed at making the instance-class relationships more flexible. We describe some of them here, and how they relate to PCCs.

Fickle [32] includes another idea for dynamic object reclassification that is not based on predicates, but on explicit reclassification by the programmer. The Fickle language provides a construct to reclassify instances of special "state" classes that can be used by programmers. These special classes, however, cannot be used as types for fields of parameters, as that would violate type safety. The main difference between PCCs and this older work is, again, the focus on collections rather than on individual instances. Additionally the Fickle reclassification construct is not declarative but imperative in nature. In contrast, PCCs are defined using declarations (the predicates).

First introduced in Flavors by Moon [77], and then in CLOS, mixins (abstract subclasses) are a powerful way of combining object behavior, as they can be applied to existing superclasses in order to create a related family of modified classes. Bracha and Cook introduced mixin-based inheritance [16], a form of class inheritance formulated as a composition of mixins. Mixin layers [101] are a form of decomposition that targets the encapsulation of class collaborations: each class encapsulates several roles, where each role embodies a separate aspect of the class's behavior. A cooperating suite of roles is called a collaboration. Mixins are only vaguely related to PCCs in that they make reuse of behavior more flexible than inheritance, allowing objects to be given several different roles. But the classification is still static, meaning that it is established before any instances are created. In contrast, PCCs serve to reclassify objects at runtime.

Self [104], and many languages inspired by it, including recent ones such as YinYang [71], include the concept of dynamic inheritance, which "allows the inheritance graph to change

during program execution." While PCCs focus on the dynamics of program execution, our goal is not to change the inheritance hierarchy at runtime, but to change the classification of objects among existing (fixed) classes at runtime. In our model, the class hierarchy is static, as it reflects the important activity of designing and modeling the application entities; but the classification of data can change at runtime based on specific predicates on the state of the objects.

Bertino and Guerrini proposed a technique that allows objects to belong simultaneously to multiple [most specific] classes [11]. The motivation was data modeling situations in which a single instance (e.g. a person) is naturally associated with multiple classes (e.g. student, and female). Although similar to multiple inheritance, the technique proposed in that paper aimed at avoiding the proliferation of subclasses that are simple combinations of other classes. This work built on the idea of mixin-based inheritance [16], and it predates traits [96, 80]. Traits are another way of reusing behavior. PCC instances do not have traits, but rather they are instances associated with a single class, that take the state from existing objects.

Finally, virtual classes [68, 35], dependent classes [41], and generics [75, 17] are mechanisms to parameterize classes. That work is vaguely related to PCCs in that it is particularly useful for collection classes such as lists, sets, etc. But the purpose of parameterized classes is quite different from that of predicate [collection] classes: the former targets the generalization of type definitions (types parameterized on other types), whereas the latter targets the association between instances and their classes.

8.8 Conclusions

This paper has introduced the concept of Predicate Collection Classes, PCCs for short. PCCs are a declarative mechanism of selecting objects from collections, reclassifying them along

the way. PCCs are both classes and specifications of collections of objects of those classes. Composition of collections can be expressed very easily using concepts from relational algebra such as subsetting, projection, cross product, union and intersection. PCCs are useful for filtering and manipulating collections, including when the elements of those collections may behave differently depending on which collection they are placed.

PCCs are in Spacetimevia Python's decorators. We have successfully used PCCs to model several iterative algorithms that use collections heavily. This showed us that PCCs are an expressive mechanism for declaring operations on collections while, at the same time, establishing new behaviors for objects that fall into those collections. PCCs are a core component Spacetime as they allow the declarative specification of optimized dataframes that are streamed from a data server to distributed simulation components, allowing each component to give their own behavior to the data – that behavior may change over time. In other words, interest management.

PCCs are inspired by relational query languages, which have proven to follow a timeless model for manipulating collections of data.

Chapter 9

Observable Replication of Objects

The contents of this chapter have been published in Onward! 2019 [2].

Debugging faults and errors in distributed systems is hard. There are many reasons for the inherent difficulty, and these have been extensively discussed in the literature [98, 12, 105]. In particular, the non determinism of concurrent computation and communication makes it hard to reliably observe error conditions and failures that expose the bugs in the code. The act of debugging, itself, may change the conditions in the system to hide real errors or expose unrealistic ones.

To mitigate the effects of these difficulties, several approaches have been developed. These approaches range from writing simple but usually inadequate test cases [103], to rigorous but expensive model checking [115] and theorem proving [109]. More recently, techniques such as record and replay [44], tracing [67], log analysis [39], and visualizations [12] have been used to locate bugs by performing postmortem analysis on the execution of distributed systems after errors are encountered.

None of these tools are interactive debuggers. Interactive debuggers in a single threaded

application context are powerful tools that can be used to pause the application at a predetermined predicate (often a line of code in the form of a breakpoint) and observe the state of the variables and the system at that point. They can observe the errors as they happen, and can be quite effective in determining the cause. Controls are often provided to execute each line of code interactively and observe the change of state after each step. Many modern breakpoint debuggers provide the ability to roll back the execution to a line that was already executed. This, along with the ability to mutate the state of the system from the debugger, can be used to execute the same set of lines over again and observe the state changes without having to restart the application.

Traditional interactive debuggers, however, become inadequate when used in parallel or distributed systems. Techniques used in single threaded applications do not translate well to a parallel or distributed context because information creation and consumption is not sequential. To create an interactive debugger for a distributed context, information flow must be modeled differently.

9.1 Related Work: Debugging Distributed Systems

Interactive debugging of parallel and distributed systems has been discussed as early as 1981 [97], but the idea has never been fully realized, mainly because it is very hard to do. However, there are many non interactive tools aid developers in debugging distributed applications. A comprehensive survey of the types of methods available can be found in Beschastnikh et al. [12]. In this survey, existing methods are grouped into seven categories: testing, model checking, theorem proving, record and replay, tracing, log analysis, and visualization. Each of these types of tools offers different insights for the developer to find bugs in the application. Tools for record and replay [44, 64, 43], tracing [67, 37], log analysis [39], and visualizers [12], try to parse the artifacts of execution such as logs, execution

stack traces, and data traces to understand the change of state in a run of the distributed system.

Many of these tools share features with interactive debuggers, as they share the common goal of exposing errors in the system to the developers. For example, tools like ShiViz [12] provide developers a way to observe the information exchanged in a distributed system by parsing logs, inferring causal relations between messages in them, and then visualizing them. Similarly, interactive debuggers for distributed systems would also need to provide a way to visualize the information being exchanged. The D^3S [64] tool allows programmers to define predicates that, when matched during execution, parse the execution trace to determine the source of the state changes. In interactive debugging these predicates are known as breakpoints and are the fundamental concept in interactive debugging.

Recently, a graphical, interactive debugger for distributed systems called Oddity [112] was presented. Using Oddity, programmers can observe communication in distributed systems. They can perturb these communications, exploring conditions of failure, reordered messages, duplicate messages etc. Oddity also supports the ability to explore several branching executions without restarting the application. Oddity highlights communication. Events are communicated, and consumed at the control of the programmer via a central Oddity component. However, the tool does not seem to capture the change of state within the node, it only captures the communication. Without exposing the change of state within the node due to these communications, the picture is incomplete. With this tool, we can observe if the wrong messages are being sent, but we cannot observe if a correct message is processed at a node in the wrong way.

9.2 Fundamental Requirements of Interactive Debuggers

Whether for a single threaded application or a distributed application, there are two fundamental requirements that interactive debuggers must provide. First, the debugger should be able to observe, step by step, the change of state of the application. Second, the debugger must give the user control over the flow of execution, so that alternative chains of events across the distributed application components can be observed. In this section, we discuss the design choices available to us when trying to achieve both goals.

9.2.1 Requirement 1: Observing State Changes

The most important goal of an interactive debugger is to observe, step by step, the change of state of the computing system. Therefore, it is important to understand how change of state can occur. A change of state can be abstracted to the consumption and creation of information. For example, over the execution of a line of code in a single threaded application, the current state of the variables in the application's heap is consumed, and a new state is created. In such an application, we only have one dimension over which information is created and consumed: time – not necessarily world time, but time modeled as the sequence of operations, or causal time. Figure 9.1 shows this progression. The only task that an interactive debugger designed for single threaded applications has to do is to show the change of state over the sequential execution of each line of code.

In the context of a parallel or distributed system, we have an additional dimension to think about: the site of execution (thread in parallel systems, nodes in distributed systems). Information is not only generated and consumed over lines of code at a site, it is also transmitted from one site to another and then made available at that site. Figure 9.2 shows

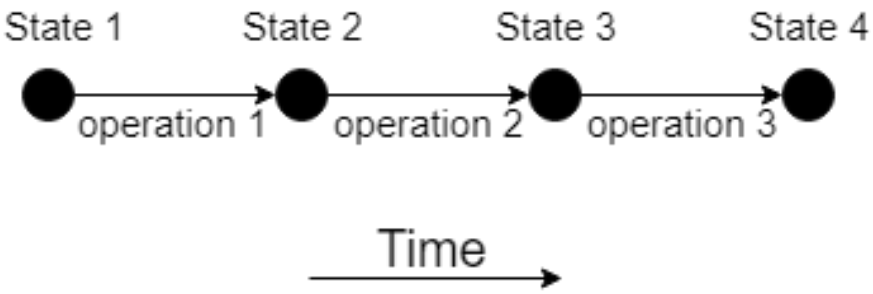


Figure 9.1: Information propagation in single-thread systems.

this information exchange. An interactive debugger for such a system must model three effects: the change of state over execution at each site, the transfer of state between sites, and the reconciliation of the states received from remote sites with the state at that site. It is possible to consider reconciliation as part of the state changes through the execution of code at a site. However, it is more useful, in the context of interactive debugging, to keep them separate. Reconciliation does not always occur through dedicated lines of code. Often asynchronous operations accept communications and update states. It could also just be a side effect of receiving a transmission of state. For example, when multiple clients concurrently update the same keys of the database using the last write wins reconciliation strategy, the old state is simply replaced with the new state as a part of the transfer. The overwriting of information is not implicitly recorded. Writes that are lost to this become hard to track. Interactive debuggers have a hard time highlighting these lost writes and so developers cannot use the debugger effectively when fixing related bugs. Since the point of the debugger is to enable the developer to reason over state changes and detect errors, it is better to expose reconciliation separately.

In summary, in a distributed or parallel system, an interactive debugger needs to expose three types of state changes: changes due to local execution, transfer of state between sites, and changes due to reconciliation of multiple states at a site. We discuss each of these next.

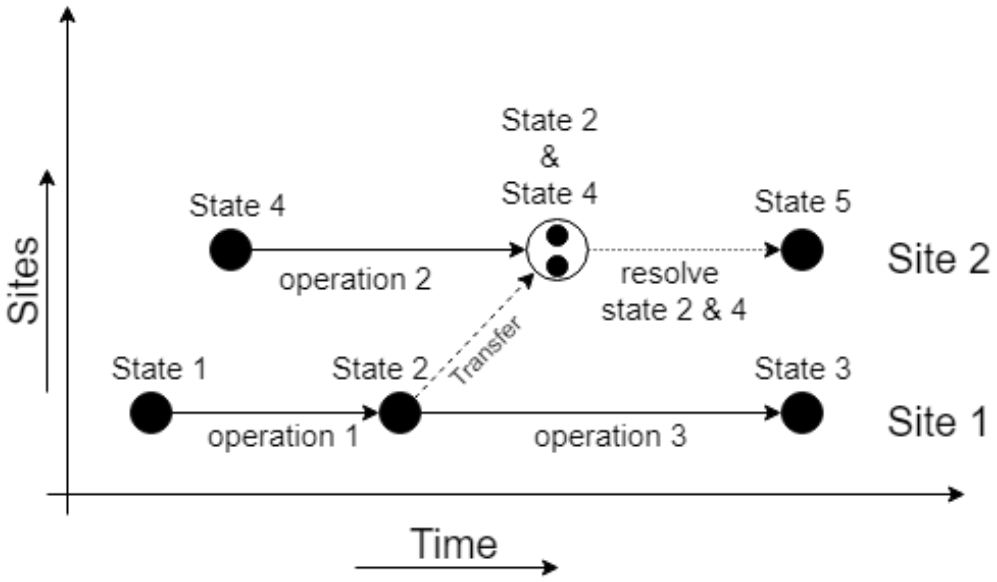


Figure 9.2: Information propagation in distributed systems.

Exposing State Changes Due to Local Execution

Designing to expose state changes due to local execution is quite straightforward, and traditional interactive debuggers do it already. There is, however, an issue of scale. Since there are multiple sites to track, there are many code paths to follow. The developer can easily get overwhelmed by this. The debugger needs to filter out the unimportant parts. One way to do this would be to treat execution paths from one point of inter-site communication to the next as one unit to step through. Doing this cuts down the number of local state updates the distributed application debugger needs to follow.

Exposing Transfer of State Between Sites

There are only two ways in which state can be transferred, and every form of communication falls into one of them: pushing and pulling changes. A web browser receiving website data is pulling changes from a server. A node sending events to all other nodes that have subscribed to the events is pushing data. Since these are the only two ways in which state can be

transferred, the debugger must pay special attention to these two primitives in any distributed model and expose the calls to these primitives explicitly to the developer.

Pull and push operations consist of one or more phases. At a minimum, the sequence for a pull operation includes a request for information, and then information is received in response to the request; similarly, the minimum for a push operation is one command wherein the information is sent. More robust implementations, however, have multiple phases with acknowledgements. Several distributed models optimize by making these calls asynchronous and sometimes going to the extent of taking the control over communication away from the programmer. For example, in the publish-subscribe model, the subscriber of data receives data via a push operation when the data is published at a different site.

Exposing Changes Due to Reconciliation of Multiple States

When a node receives state changes from another site, it needs to reconcile the state. It is important to understand reconciliation, and differentiate it from conflict resolution.

Reconciliation is a two step process. The first step is to receive the information of state change from another site. The second step is conflict resolution where the information received is meaningfully merged with the information already present in the site, to make the local state coherent for the next local execution. Different distributed models deal with these two steps in different ways, making them particularly tricky to observe.

In most distributed models, the two steps happen together. Remote changes are evaluated as soon as they are received and decision is taken regarding their incorporation into the local state, e.g. last-write-wins, CRDTs [100]. In these models, observing conflict resolution is the same as observing reconciliation. In some distributed models, however, state changes received are stored, and conflict resolution, if any, is deferred to a later time. For example, total store ordering [50], global sequence protocols [21, 46], TARDiS [30], Irmin [52], concurrent

revisions [20], and GoT, are a few models that first store the incoming changes, and provide the programmer control over when these changes are resolved and introduced into the local state. From the point of view of the user of an interactive debugger observing reconciliation in these models, the user must both observe when information is received from a remote site, and when the information is accepted and incorporated in the local state.

Looking at conflict resolution in particular, there are a myriad of ways in which concurrent state updates are resolved, and it entirely depends on the underlying distributed model. Some models such as the last-write-wins, total store ordering [50], global sequence protocols [21, 46], etc. resolve conflicts implicitly. Since many of the models do not retain causal relations between reads and writes of state, it is hard to tell if an overwrite was an intended update, or the result of implicit conflict resolution. As such, it becomes quite difficult for an interactive debugger to expose the point of conflict resolution in such models. Other models such as TARDiS, Irmin, concurrent revisions, and GoT resolve conflicts explicitly using programmer-written merge functions. Although in many models these merge functions are called asynchronously, there exist specific execution paths which deal with conflict resolution, and this can be exposed by the interactive debugger.

9.2.2 Requirement 2: Controlling the Flow of Execution

Interactive debuggers, as the name suggest, must allow the user to debug the distributed application interactively. To be interactive, the debugger must take control of all forms of state changes present in the distributed system and hand this control over to the user. In a single threaded system, with only one form of state change and executed at a single location, taking control of the execution and handing it over to the user is relatively straightforward. However, in a distributed system, this is harder. There are more forms of state changes as described above, and these state changes can execute over multiple sites. If the interactive

debugger is controlled by the user on only one site, and the rest of the sites are free to execute, then the user has control over only one form of state change: changes occurring due to local execution.

In order for the user to have control and observe all forms of state change, the user needs to be able to pause all sites when one site is paused. The problems associated with distributed computing are inherited by the debugger trying to exercise global control over the system. An easy solution, one that is present in traditional interactive debuggers when trying to debug multi-threaded programs, is to pause all threads of execution when one thread is paused. For example, the GDB debugger has an all-stop mode[1] that behaves in that manner. While this solution can work in simple multi-threaded applications operating out of the same machine, this approach, used as is, becomes difficult when moving to the distributed context when sites are located at different machines. Triggering a pause on one machine would have to be made instantly visible to all nodes, which is hard, as there are inevitable network delays, leading to unintended state changes after the user has tried to exert control.

The problem of exerting global control is even more pronounced when each site communicates with multiple sites, such as in peer-to-peer applications. In a server-client model where all clients only communicate with the server, pausing the server could allow us to pause the clients. A solution then, perhaps, could be to transform the distributed system into a server-client model with an interactive debugger as the central component. All forms of state changes could be rerouted through this central component, giving this component the ability to observe and control all these forms of state change.

Rerouting both networked and local state changes would significantly alter the network conditions for the system. This is fine, as long as the user of the system is able to leverage the interactivity gained to explore state changes in the application related to different orderings of concurrent operations. The advantage offered depends on the system being developed. If

[1] https://sourceware.org/gdb/onlinedocs/gdb/Thread-Stops.html

there are too many variations or ordering possible (e.g. a large system with many sites), the developer might not be able to observe them all.

While exploring the design choices available to us when fulfilling these two requirements, it becomes clear that the underlying distributed model on which the interactive debugger is to be built on is absolutely dominant. The communication, and state reconciliation methods used by the model play a heavy role in determining the capabilities that the interactive debugger has. That said, in the next section, we explore what support from the distributed model is necessary to make an interactive debugger viable.

9.3 Constraints on Distributed Systems

There are a large number of distributed computing models. Each model optimizes for different goals and, therefore, makes different design choices for different aspects under its control. Some of the choices can help an interactive debugger expose the state changes of the system accurately to the developer, while others can hinder it. If interactive debugging is a goal, the underlying distributed system model must be constrained in specific ways. In this section, we discuss at least three constraints that a distributed model must abide in order to support the requirements of interactive debuggers discussed in the previous section.

9.3.1 Read Stability

As discussed in the previous section, there are three ways in which information is created and consumed in a distributed context: state change through local execution, transfer of state between nodes, and state change through reconciliation of multiple states at a node. The distributed model must expose these three ways as separate events on their own. An important constraint that the model must have in order to be able to separate these three

scenarios is read stability.

Read stability – a concept typically associated with isolation in database transactions [60] – is a property of the model where changes to the local state can only occur when the local execution context wants it to. There are two ways to affect change to the local state: writes from local execution, and reconciliation of local state and a state that is received from an external node. Read stability can be easily achieved if the model, after the transfer of information between sites, does not reconcile the information immediately but instead stores the information (cache, queue, etc.) and waits for the local execution to accept these changes. For example, a multi-threaded system with multiple threads writing concurrently to shared variables, without locks, breaks the read stability requirement.

Without read stability, the interactivity of the debugger is heavily curtailed. For example, when the user of an interactive debugger debugging an application paused at a certain execution point, steps through one line of execution, the expectation from the user is that the state change being observed is a state change due to the execution of that one line. However, in a system without read stability, this might not be true. The local state change could also have changed via reconciliation with external changes, essentially giving rise to a situation where there are multiple types of state change being executed in the same step. Interactive debugging is a debugging method that aims to give the control of execution to the user with the aim of letting the user observe all forms of state change explicitly. Therefore, having multiple types of state change being executed in a single step is not acceptable. Read stability solves this problem by ensuring that updates to the local state are explicitly defined and occur only when the local execution context expects it.

There are many methods to achieve read stability. For example, in the global sequence protocol (GSP) model [46, 21], the operations that are distributed to all remote sites are placed in an pending queue, ready to be applied through an explicit primitive given to the local application. In TARDiS [30], concurrent writes are placed under version control in

separate branches to avoid interfering with each other until one context wants to introduce these concurrent changes to its branch using the resolve primitive.

9.3.2 Separation of Published State and Local State

When a local site makes changes locally, some models immediately make these changes available for other sites to observe. This means that the local state of a site at the end of every line of code is potentially a site that is going to be observed by a remote node, which also means that each of these states have to be tracked by the debugger. Observing state change over local execution over every line of code in all sites of a distributed system is definitely not scalable and the information can be overwhelming to the user observing it.

Some models (for e.g. publish-subscribe), do not make changes, made locally, immediately available for other sites. These changes are made ready for consumption only when they are specifically published. The local state of the site is kept separate from what the site wants to share. An interactive debugger for the system just needs to take control over the points where the site shares information, grouping all changes to local state in between as a single operation. This makes the observable set of operations smaller where we trade in the fine granularity of observing change of state over every line of code to look at the execution in broader strokes which makes the implementation feasible. Therefore, the separation of published state and local state is critical when attempting to create an interactive debugger at scale.

9.3.3 Explicit Mechanisms for State Change

Just as the local execution context should have control over the introduction of changes to its state, the local execution should also have control over when the local changes are being

transmitted to other nodes. Having an explicit primitive to start the transfer of changes helps the debugger expose the start point of transfer. Having explicit primitives deal with receiving these changes, and then introduce these changes to the local state helps the debugger expose reconciliation between the transferred state and the local state.

9.3.4 GoT: Enabling an Interactive Debugger

As explained in the previous section, the distributed model needs to have the following properties to make an interactive debugger feasible: first, the nodes need to have read stability; second, the published state at each node must be separate from the local state used by the application code at that node; finally, the model must expose primitives for the transfer and reconciliation of states between nodes. We explain how GoT incorporates these properties in its design.

Read Stability: Each GoT node computes only on the objects in the snapshot. The snapshot can only be updated with external changes when the *checkout* or pull primitives are invoked. Since these are invoked by the application code at the node, and not automatically behind the scenes, the snapshot does not change unless it is directed to by the application code. Therefore, GoT supports read stability.

Separation of published state and local state: GoT also separates the published state (the version history) from the local state (the snapshot). All changes received via a *fetch* or *push* request will only include changes that have been committed by the sender node.

Explicit mechanisms for communication and reconciliation: The dataframe has an explicit mechanism for inter-node communication (*fetch*, *push*) and conflict resolution (merge functions) that can be tracked and used by the debugger to observe both the transfer of state and the reconciliation of state updates between nodes.

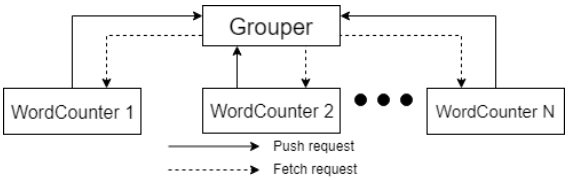


Figure 9.3: Network topology of an example distributed word counting application built on Spacetime.

9.4 GoT example: Distributed Word Counter

As mentioned before, the underlying distributed computing model has a dominant influence on the feasibility of an interactive debugger. Our interactive debugger, GoTcha, is designed for GoT. Specifically, GoTcha is built as a debugger for the implementation of GoT called Spacetime. In this section, we describe a GoT example that is used in subsequent sections to explain the features in GoT that are relevant to GoTcha and to show case the abilities of GoTcha itself.

The example that we use is a distributed word frequency counter. The application takes a file as input and shards it by line. These lines are distributed to several remotely located workers which tokenize and count the frequency of the tokens in the lines. The partial counts are then aggregated and presented by the application as the final word frequency tally.

The distributed word frequency counter application has two types of GoT nodes: WordCounter and Grouper. The Grouper node controls the execution of this Spacetime application. It is responsible for sharding the input files into lines and aggregating partial word frequency counts to reach the final tally. The WordCounter node is responsible for tokenizing and counting the word frequencies in each line.

WordCounter nodes are responsible for the communication in this Spacetime application. Every WordCounter node must *fetch* changes from, and *push* changes to the Grouper node. This relation between these nodes is shown in Figure 9.3 and defines the network topology of

```
1  class Line(object):
2      line_num = primarykey(int)
3      line = dimension(str)
4      def __init__(self,line_num,line):
5          self.line_num = line_num
6          self.line = line
7      def process(self):
8          # a simple tokenizer
9          return self.line.strip().split()
10
11 class WordCount(object):
12     word = primarykey(str)
13     count = dimension(int)
14     def __init__(self,word,count):
15         self.word = word
16         self.count = count
17
18 class Stop(object):
19     index = primarykey(int)
20     accepted = dimension(bool)
21     def __init__(self, index):
22         self.index = index
23         self.accepted = False
```

Figure 9.4: The types used by the Word Counting application.

our example application.

The dataframes at each WordCounter and Grouper nodes share objects of type Line, WordCount, and Stop that are shown in Listing 9.4.

Objects of type Line are shards of the input file. The dimension line_num (defined in line 2) is the primary key, of type integer, that represents the line number in the input file. The dimension, line (line 3), stores the contents of the line as a string. Objects of type WordCount store the word frequency for a unique token and have two dimensions: the primarkey, word (line 12), is a string representing the token, and count (line 13), is an integer representing the frequency of that token. WordCounter Nodes communicate completion using objects of type Stop having two dimensions: the, primarykey, index (line 19), an unique identifier representing a single WordCounter worker, and accepted (line 20), which is set to True by

the WordCounter node signalling the completion of its task. Any state in attributes outside these dimensions is purely a local state, and is not tracked and shared by the dataframe.

Listing 9.5 shows part of the application code for an instance of the Grouper node. The grouper_node is instantiated (lines 18-20) with the application code, defined by the function Grouper, along with the types to be stored in the dataframe, and the port on which to listen to incoming connections. The grouper_node is launched using the blocking call, start (line 21), and takes in the parameters that must be passed to this instance of the Grouper node: the input file and the number of WordCounter nodes that are going to be launched.

The Grouper function (line 1) that is executed receives the repository, dataframe, as the first parameter, and all run-time supplied parameters as the additional parameters. The node iterates over each line in the input file, creating new Line objects for each line. The Line objects are added to the local dataframe (line 4), similar to how new files are added to a changelist in git. After each Line object is added, these staged changes are committed to the dataframe (line 5) and are available to any remote dataframe that pulls from it. After all Line objects are added, Stop objects are added, one for each WordCounter worker in the application, and committed to the dataframe (line 6-7). Grouper now has to wait for all WordCounter workers to finish tokenizing the lines that it has published, and the state of the Stop object acts as that signal (Lines 9-12). Once every worker has accepted the Stop object associated with it, the Grouper reads all the WordCount objects in the repository and displays the word frequency to the user.

Listing 9.6 shows part of the code for an instance of the WordCounter. Multiple instances of the WordCounter node are instantiated with the remote address of the Grouper node, and the same Types that Grouper uses (lines 32-35). Each instance is started asynchronously with the parameters that it needs (line 36).

The application code for WordCounter (function WordCounter shown in lines 1-26) also takes

```
1  def Grouper(df,filename,ncount):
2      i = 0
3      for line in open(filename):
4          df.add_one(Line, Line(i,line))
5          df.commit(); i += 1;
6      df.add_many(Stop,
7        [Stop(n) for n in range(ncount)])
8      df.commit()
9      while not all(
10             s.accepted
11             for s in df.read_all(Stop)):
12         df.checkout()
13     for w in df.read_all(WordCount):
14         print(w.word, w.count)
15
16 if __name__ == "__main__":
17     filename, ncount = sys.argv[1:]
18     grouper_node = Node(
19         Grouper, server_port=8000
20         Types=[Line,WordCount,Stop])
21     grouper_node.start(filename,ncount)
```

Figure 9.5: The Grouper node.

the dataframe as the first parameter. An independent and new dataframe is created for each instance of WordCounter, and they all have the same Grouper node as the remote node. The WordCounter keeps pulling changes from the remote node (line 4) for as long as there is a new line to read in the updated local dataframe and until a Stop object associated with the instance is read in the local dataframe. In each pull cycle, the WordCounter reads a Line object from the local dataframe, using index (line 5), and tokenizes it (line 7). For every word in the token list, the node retrieves the WordCount object associated with the word from the dataframe (line 9), creating and new object if it does not exist (line 11-14), and increments the count dimension in the object by one (line 16). These updates (both new objects, and updates to existing objects), staged in the local snapshot, are committed to the local dataframe and pushed to the remote Grouper node (line 23). The WordCounter ends operations if after pulling updates from Grouper, a Stop object is present in the dataframe, and there are no new Line objects to read. The stop object is accepted by setting the accepted

dimension to True and this update is committed and pushed to Grouper as the last operations by the WordCounter node.

In the WordCounter example, the state of the WordCount objects created and updated by different WordCounter nodes can be in conflict with each other when changes are pushed to the Grouper node. For example, if two WordCounter nodes concurrently read the same word in two different lines and the word has not been observed before, both the nodes would create a new WordCount object for the word. When both changes are pushed to the Grouper node, a conflict is detected and a merge function is called.

An example merge function is shown in Listing 9.7. This function is called asynchronously when a conflict is detected, and is used to only to reconcile conflicting state updates. The merge function receives four parameters: an iterator of all objects that have direct contradictory changes that cannot be auto resolved (*conf_iter*), as well as three snapshots of the state, one for the point where the computation forked (*orig_df*), and two for the version at end of the conflicting paths (*your_df*, *their_df*).

In the merge function shown, objects (Line, WordCount, and Stop objects) that are new or modified in the incoming change but do not have conflicting changes in the local history are first accepted (line 3). For the objects that are in conflict (only WordCount objects can be in conflict), we read the states at three versions of the objects: the state at the fork version, and the two states at the conflicting versions – *orig*, *yours*, and *theirs* from the iterator (line 3), respectively. Then, the dimension count in objects that have been updated together are added up and stored in the object tracked by the *your_df* snapshot. At the end, this modified version of *your_df* is considered to be the reconciled state and returned to the version history.

There is a bug in this merge function as it does not add counts correctly. We will use this bug to demonstrate the capabilities of the interactive debugger. For quick reference, the correct merge function is shown in Listing 9.13 at the end of Section 9.5.

```
def WordCounter(df,index,ncount):
    line_num=index; stop=None; line=None
    while not stop or line:
        df.pull()
        line = df.read_one(Line,line_num)
        if line:
            for word in line.process():
                # reads from the snapshot
                word_obj = df.read_one(
                    WordCount,word)
                if not word_obj:
                    word_obj = WordCount(
                        word,0)
                    df.add_one(word_obj)
                # changes only snapshot
                word_obj.count += 1
            line_num += ncount
        stop = df.read_one(Stop,index)
        # commit changes
        # to version history
        # and push these changes
        # to remote node.
        df.commit(); df.push()
    stop.accepted = True
    df.commit()
    df.push()
if __name__ == "__main__":
    workers = []
    address = sys.argv[1]
    num_workers = int(sys.argv[2])
    for i in range(num_workers):
        wnode = Node(
            WordCounter,
            Types=[Line,WordCount,Stop],
            remote=(address, 8000))
        wnode.start_async(i, num_workers)
        workers.append(wnode)
    for w in workers:
        w.join()
```

Figure 9.6: The Word Counter node.

```
1  def merge(conf_iter,orig_df,
2            your_df,their_df):
3      your_df.update_not_conflicting(
4          their_df)
5      for orig,yours,theirs in conf_iter:
6          assert isinstance(
7              yours,WordCounter)
8          yours.count += theirs.count
9      return your_df
10
11 ...
12     # Updated Node initialization
13     grouper_node = Node(
14         Grouper,
15         Types=[Line,WordCount,Stop],
16         conflict_resolver=merge,
17         server_port=8000)
18 ...
```

Figure 9.7: Merge function used at the Grouper node.

9.5 GoTcha

GoT nodes execute their tasks over shared objects that are stored in the version history in the dataframe. These version histories are primarily used, in GoT, to facilitate the communication between nodes using delta encoding and to detect and resolve conflicts. The version history is an internal component of the dataframe and is, therefore, typically not exposed to the programmer.

In version control systems, the version histories are more than just a datastores for files. They document the evolution of the files stored in the repository over time. There are many tools available, such as GitKraken [2] and SourceTree [3], that expose this evolution to users. Observing the version history, through these tools, not only tells us the current state of the repository, but also all the changes that were made to the repository in the past and in the order that they were made. This same principle can be leveraged in GoT, to expose the

[2]https://www.gitkraken.com
[3]https://www.sourcetreeapp.com/

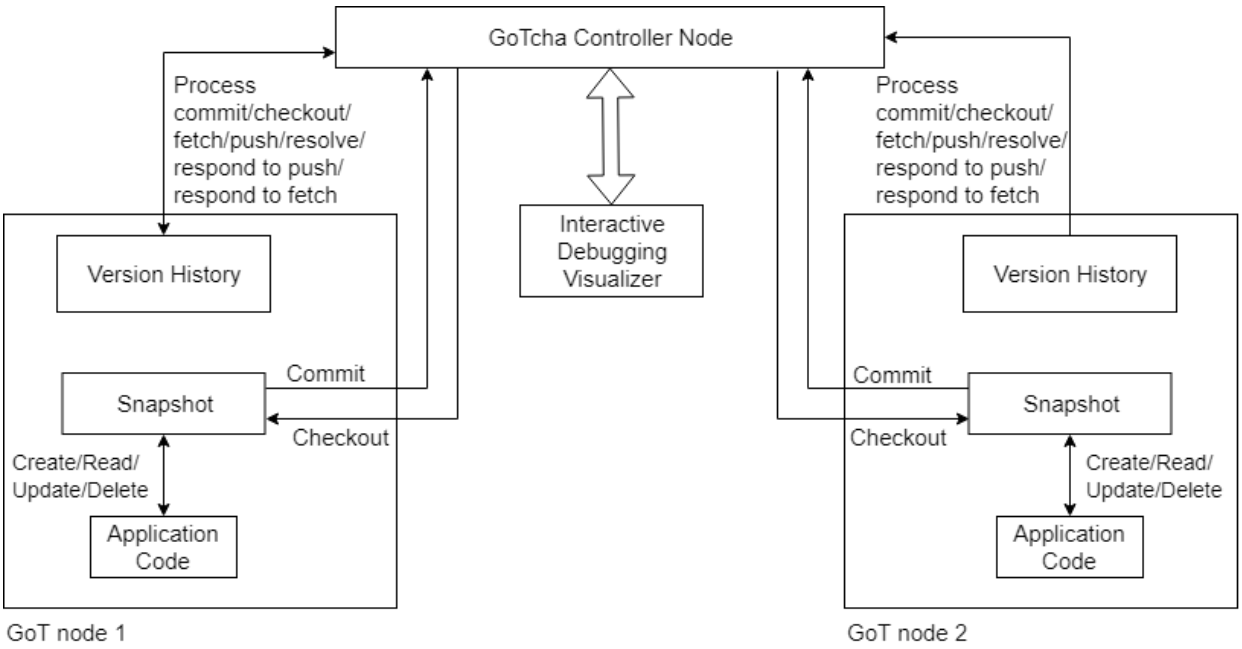


Figure 9.8: Architecture of GoTcha.

version history of object changes to the user. A debugger for GoT can expose the version history allowing users to observe the evolution of the state at a node, and detect errors that have already occurred. In addition to viewing errors in the version history, live and interactive debugging becomes possible, as the updates to the version history is driven explicitly by the application code, and performed via a small API in the dataframe (see Table 4.1). By taking control of these APIs and giving this control of the execution to the user, the user can stop the application, observe the state of the version history at each node, resume and observe the change of state over the execution of the dataframe operations. This, along with the ability to observe variations in the order of execution, will assist the user in observing errors as they occur.

We created an interactive debugger called GoTcha, to expose the changes made to the version history at each node in a Spacetime application. In this section, we explain the features of GoTcha. We will continue to use the distributed word frequency counter example, detailed in the previous section, to showcase the features of the debugger.

```
foo
bar
bar
baz
bar
bar
```

Listing (9.1) Input file.

```
foo 1
bar 4
baz 1
```

Listing (9.2) Expected output.

```
foo 1
bar 6
baz 1
```

Listing (9.3) Observed output.

For the purpose of the example, a test input file was created consisting of six lines, each with one word – see Listing 9.1. The word frequencies for the words foo, bar, and baz are one, four, and one respectively. The application consists of two WordCounter nodes and one Grouper node that are launched in different machines. During execution, as shown in Listing 9.5, the Grouper node adds six Line objects and two Stop objects into its dataframe, and waits for the Stop objects to be accepted by the WordCounter nodes (Listing 9.5). WordCounter1 reads, tokenizes, and counts words on lines 1, 3, and 5. WordCounter2 does the same for lines 2, 4, and 6. Finally, both WordCounter nodes accept their Stop objects, and execution completes. The expected output is shown in Listing 9.2. However, a different output is observed, shown in Listing 9.3. The observed output is wrong, and this is where GoTcha can help.

9.5.1 Operation

GoTcha follows the centralized service approach, discussed in Section 9.2, to have complete control over the nodes and expose the version history to the user at each node. This central service is an application by itself and is launched before any application nodes are launched. We call this application the GoTcha Controller Node (GCN from now on). The GCN is a web service to both the User Interface (UI), and the nodes in the application. When GoT nodes are launched in debug mode, they register themselves with the GCN.

When in debug mode, the architecture of the application is modified, during runtime, from the original GoT structure to what is shown in Figure 9.8. The primitives that read or write

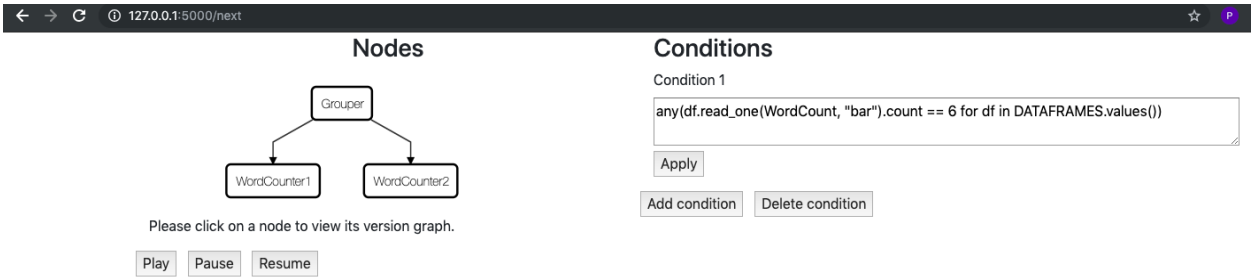


Figure 9.10: Debugger showing the network topology of the application.

from the version history at each GoT node (*commit*, *checkout*, *fetch*, *push*) are all rerouted through the GCN. With each interaction, the version history at each node is also sent to the GCN to be shown to the users. While a traditional interactive debugger for a single threaded application would observe the change of state between each line of code, GoTcha observes state changes over each action of read or write performed on the version history at each node.

At the start of the debugging session, after every node has been launched in debug mode, the user is shown the network topology of the application, as shown in Figure 9.10. In this figure, on the left, the user sees that two WordCounter nodes (WordCounter1 and WordCounter2) and one Grouper node are being controlled by the GCN. The Grouper node is the authoritative node in the application, with both the WordCounter nodes making *fetch* and *push* requests to the Grouper node. On the right, there is an input field for the user to add one or many breakpoint conditions to the debugger. The breakpoint condition shown here, returns True if there exists any WordCount object with the count dimension set to six, in the dataframe, at any GoT node.

9.5.2 Observing Node State

The current version history of any node can be observed by clicking on the node in the topology graph in Figure 9.10. Figure 9.11 shows the node view of the Grouper node, observing the result of the execution of the *commit* primitive at line 5 of Listing 9.5 (during

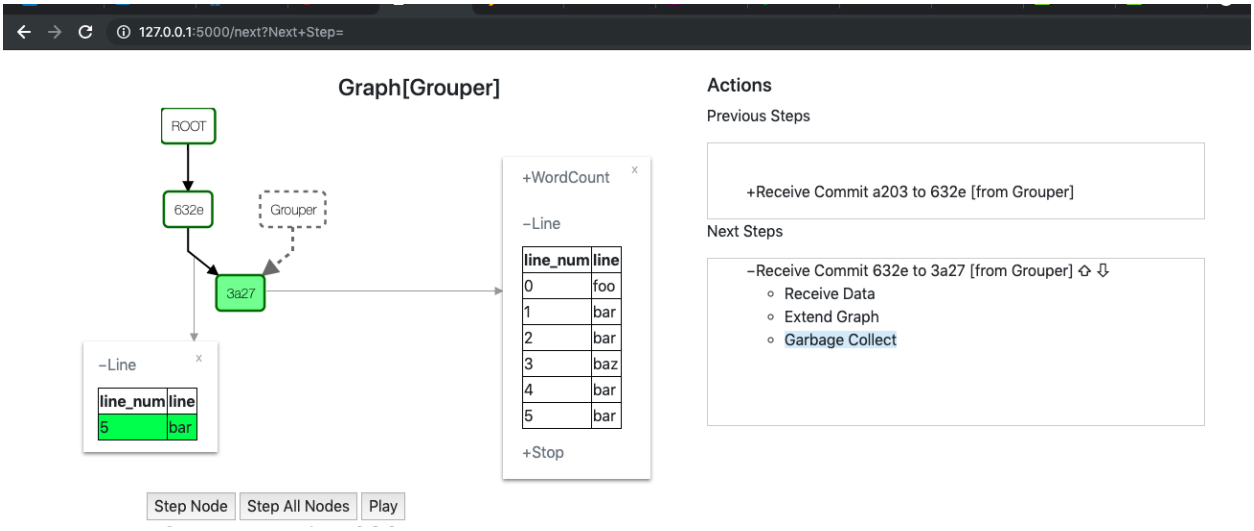


Figure 9.11: Debugger view showing version history at the end of a *commit*.

the last iteration of that loop). The version history is shown on the top left. The history shows three versions: ROOT, 632e, and 3a27. 3a27 happened-after 632e which in turn happened-after ROOT (the start version of every version history). The HEAD version of the graph, highlighted in green, is 3a27.

Selecting the version brings up a tooltip that shows in tabular form, the state of objects at that version. The tooltip shows that the state at version 3a27 has six objects of type Line, and shows the values of the two dimensions (line_num, line) for the Line objects. Though the tables for Stop and WordCounter have been collapsed, as shown by the plus shaped user interface element, there are no objects of those types present yet.

Selecting the edge brings up a tooltip that shows the delta change (diff) associated with that edge. The diff associated with the edge 632e $\rightarrow$ 3a27 is also shown on the bottom left. In this case, the diff consists of a single object of type Line with the dimensions line_num, and line having the values 5, and 'bar' respectively. The entry is also marked in green, which signifies that the entry is a newly added object (added in line 4, Listing 9.5). Uncolored entries are considered to be modifications, and entries marked in red are considered to be deleted objects. The state of every version, and the diff associated with every edge can be

observed. The dotted line relation shows us that the state of the snapshot of the Grouper node is known to be at version 3a27.

On the right of Figure 9.11, we see the state of the actions being executed on the dataframe at the Grouper node. The user sees both a list of previous steps that have been executed on the version history, and a list of steps that have to be executed (next steps). At the top of the next steps list is the current active step being executed. Each step directly maps to one of the dataframe primitives rerouted through GoTcha and is broken up into several phases.

We can see that the *commit* primitive has three phases. The first phase is receive data where a *commit* request is made using the diff staged in the snapshot. Stepping through this phase brings us to the extend graph phase, where the version history graph is extended from the HEAD version (632e) to the newly created version (3a27) and the new version is marked as the new HEAD. The last phase of commit, which is yet to be executed, is the garbage collect phase where obsolete versions in the graph (in this case 632e) are cleaned up.

At the bottom, we see three buttons: Step Node, Step All Nodes, and Play. Clicking on Step Node, would allow the garbage collect phase of *commit* at the Grouper node to be executed. Clicking on the Step All Nodes, would allow all nodes to execute the next phase of the step that they are paused at, if any. Play allows the user to fast forward the execution up until the next breakpoint condition is hit.

Since the *fetch*, and *push* primitives of the dataframe span across multiple GoT Nodes, they are broken up into two sets of operations each: *fetch* and *respond to fetch*, *push*, and *respond to push*, to observe the state changes at both the node making the request and the node receiving the request.

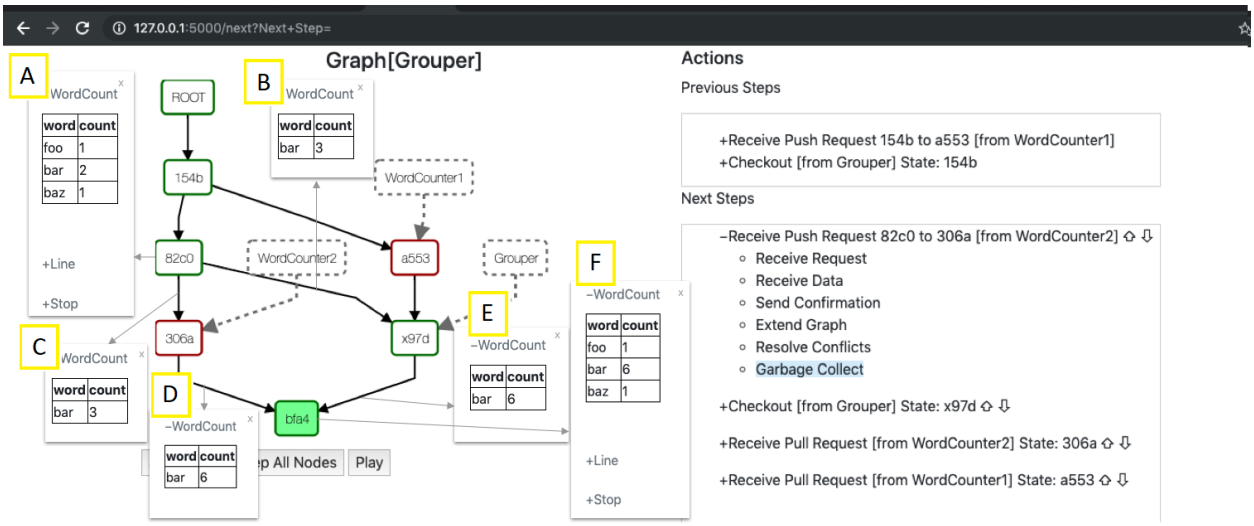


Figure 9.12: Debugger view at Grouper showing response to a *push* request.

9.5.3 Debugging Word Frequency Counter

To debug the mismatch between the expected and observed output of the application, we first put in the condition for the breakpoint as seen in Figure 9.10, and hit the play button. All nodes are executed in debug mode and reroute their primitives through the GCN. At each of these rerouted steps, the GCN observes the states of the dataframe in each Node and executes the breakpoint conditions.

When the breakpoint is matched, the execution of all nodes is paused and GoTcha shows the view of the Grouper node where the condition matches, shown in Figure 9.12. Here, we see that the current step being executed is a *push* request from WordCounter2 from version 82c0 to version 306a. The execution is paused at the start of the garbage collect phase.

The version history contains seven versions. Starting from the top, we have ROOT again as the start version. Version 154b happened-after ROOT. All versions that were present

between 154b and ROOT, like the version 632e and 3a27, have all been garbage collected.

At 154b, we see a fork in the path. Both versions 82c0 and a553 happened-after 154b, and are siblings. These are concurrent updates and were performed on different GoT nodes. Version 82c0 is bordered green while a553 is bordered in red. This means that update 154b → a553 was received by the version history at Grouper after the update 164b → 82c0. The GoT node resolved such conflicts using the custom merge function written in Listing 9.7. The output of the merge function was a new version x97d. Since x97d happened-after both the concurrent versions, 82c0 and a553, the graph was updated with the happened-after relations and x97d has two in-edges. Additionally, each of these edges are associated with a diff that transforms the previous version to the version at x97d.

Over the course of execution, another concurrent update was performed with the update 82c0 → 306a being concurrent with previously resolved conflict. Another conflict resolution is performed using the merge function. A new resolved version bfa4 is created having a happened-after relation with both 306a, and x97d. The version history is updated to show these relations and the version bfa4 is marked as the current HEAD version of the version history at Grouper.

Looking at the dotted line relations, we see that the snapshot at Grouper is at the version x97d. Additionally, the last know versions of WordCounter1 and WordCounter2 are a553, and 306a, respectively.

The state at version bfa4 is shown in the tooltip F. The tooltip shows us three WordCount objects and the WordCount object for the word 'bar' has a count of six, showing us why the conditional breakpoint was hit. Looking at the version at the start of the merge, 82c0, in the tooltip A, we see that the count of 'bar' is two. The diffs associated with 82c0 → 306a (tooltip C) and 82c0 → x97d (tooltip B) both update the count of 'bar' to three. This means that both WordCounter1 and WordCounter2 had the count of 'bar' as two, and observed

```
1  def merge(conf_iter,orig_df,
2            your_df,their_df):
3      your_df.update_not_conflicting(
4          their_df)
5      for orig,yours,theirs in conf_iter:
6          assert isinstance(
7              yours,WordCounter)
8          yours.count += theirs.count
9          if orig: # False if new objects.
10             yours.count -= orig.count
11     return your_df
```

Figure 9.13: Merge function used at the Grouper node.

a 'bar' token updating the count concurrently to three. At the end of the merge function, this count is set to six, and can be see in the diffs for both 306a → bfa4 (tooltip D) and x97d → bfa4 (tooltip E). This means that the error is in the merge function. We can see that the merge function in Listing 9.7, on detecting a conflicting count, simply adds up the counts. So receiving two counts of three, would result in a count of six. However, the actual increment in each update is actually just one. The right way to merge counts would be to find the total change in count and add it to the original count. We can fix the code as shown in Listing 9.13 and the word counting application gives the right output.

This bug was found quite easily because GoTcha exposes the version history. By looking at the evolution of the version history, even though the error had already occurred, we could see in which type of state change the error occurred in. In this case, we could see that the version state was correct before the merge function, but after the reconciliation of two correct states, the state was wrong, telling us that the error was in the custom merge function. GoTcha exposes bugs in a Spacetime application in the same way a tool viewing git history can help find the commit that caused a bug in the code. Instead of the evolution of the files being looked at, GoTcha looks at the evolution of the state at each node.

Table 9.1: Mapping the primitives of GoT to the types of State changes

Type of state change	**GoT Primitives**
Change in local state	Commit, Checkout
Inter-node state transfer	Push, Respond to Fetch
Reconciliation of states	Fetch, Respond to Push, Commit, Checkout

9.6 GoTcha: Meeting the Fundamental Requirements

In Section 9.2 we describe, in detail, the fundamental requirements that an interactive debugger must fulfill. To summarize, the debugger must expose to the user all forms of state changes in the application while minimizing the interference in the natural flow of execution. In this section, we discuss how GoTcha meets these fundamental requirements.

9.6.1 Observing State Changes

There are three forms of state changes present in a distributed system that are relevant to an interactive debugger: state changes at a node due to local execution, transfer of state between nodes, and the reconciliation of the state received via transmission and the local state at each node. Table 9.1 maps the GoT primitives to the type of state change that it facilitates. In what follows, we explain how GoTcha exposes all these types of state changes to the user.

Observing changes in local state: In GoT, the "local state" is the snapshot. The snapshot is updated by write operations directly from the local application code. These kinds of state updates can be observed by traditional debuggers. However, as mentioned in Section 9.2.1, the amount of state changes in a distributed system can overwhelm the user, and a distributed systems debugger should reduce the number of such updates shown. GoTcha does not track the change of state over every line of code at each GoT node. Instead, it tracks the change in

the snapshot over consecutive interactions (commit, and *checkout*) between the snapshot and the version history. All changes in between these interactions is purely local and grouped together as one update by both GoT and GoTcha.

Observing the transfer of state: The local state of a GoT node is transferred to remote nodes in two ways: a *push* from the local node to the remote node, or the response by a remote node to a *fetch* from the local node. The user can step through these primitives to observe this communication. Specifically, the user can see when such requests are made, and the delta changes that are transferred as a part of these requests.

Observing reconciliation of multiple states: When a node receives state changes transferred from a remote node, it needs to reconcile the states changes. As explained in Section 9.2.1, reconciliation is a two step process: first, receiving changes from a remote node, then introducing these changes to the state of the local node. GoTcha must expose both steps to allow the user to observe reconciliation correctly. The first step is observed in GoTcha when observing the state changes on receiving deltas either at the end of the *fetch*, or when responding to a *push* request. The acceptance of these changes can be observed during the *fetch*, response to a *push*, *commit*, or *checkout*. Conflicts are resolved using custom merge functions that are observed by GoTcha. Changes can also be accepted, as is, without conflicts through a *checkout*. All ways of receiving delta changes and observing the acceptance of these changes can be observed by GoTcha allowing the user to observe reconciliation of multiple states.

9.6.2 Controlling the Flow of Execution

GoTcha follows the centralized debugger design explained in Section 9.2.2. The central component, GCN, takes control of all GoT primitives that read or modify the version history. This means that even *commit* and *checkout* primitives, which are normally local operations,

are also routed through the GCN. Control over the execution of the changes to the version history is given to the user. The user can reorder and interleave requests that have to be processed and can explore possible execution variations. This would allow the user to observe if, for example, the conflict resolution functions are performing as intended. The user interface for reordering or interleaving execution steps is shown in Figure 9.12, where there are additional steps that are pending at the Grouper node. The developer can reorder and interleave these pending operations using the promote and demote arrows shown on the right side next to each step.

Roll backs are an additional and useful tool to explore different state changes without having to restart the entire execution. Since we have the entire history of execution given to us by the version history, we support roll backs to a previous version. When a roll back in performed, the state in the version history is reverted to an older version. It is important to note that the local state and the execution of the application code is not rolled back. This means that state changes observed after roll backs are only meaningful when the application code at each node is stateless and performs the same action iteratively. However, reconciliation can be observed well using roll backs.

Rolling back the execution state at a node, along with the state of the dataframe, would require that we either take control of the programming language runtime in each node, which suffers the same problems of coordinating distributed control as discussed in Section 9.2.2, or we integrate GoTcha with a traditional single-threaded debugger at each GoT node. While the first is unfeasible, the second can be a future possibility and is discussed in the next section.

9.7 Beyond Interactivity

GoTcha is a good first step into the interactive debugging of distributed applications. By relying on the idea of version control of objects, and exposing the version history at every node, we meet the minimum requirements for observing all forms of state changes in distributed applications. GoTcha fills a hole – interactivity – in the tools available for debugging distributed applications. Interactivity has been an elusive piece in this ecosystem, and not much is known about how it can be used in a distributed context. With GoTcha, we see both potential and challenges in the future development of interactive debuggers for distributed applications. Moreover, we envision the development of powerful tools by combining GoTcha with existing concepts related to distributed debugging. In this section, we expand on the potential and challenges of this vision.

9.7.1 Scalability of Interactivity

An inherent property of traditional interactive debuggers is their reliance on the user to explore the possible paths of failures. This is no less true for GoTcha. However, in large systems with large number of possibilities to observe, this interactivity can potentially be overwhelming to the user. Interactive debuggers for single threaded systems ignore this complexity by design, hoping that the advantages offered by the live exploration of the execution of code compensates for the disadvantage of not being able to explore every path and fixing all issues. This exploration space is much larger in distributed systems when compared to single threaded systems because there are more types of state changes that have to be considered. Therefore, in GoTcha, the advantages offered by live exploration of the execution heavily depends on the scalability of both the debugging system, and the user interface, when increasing the number of GoT nodes in the application being debugged.

Scaling the Debugging System

In an application with a few number of nodes, the exploration of the execution can be easily visualized and followed and the centralized approach of GoTcha does not hinder the debugging process. However, in applications with a large number of nodes, which is common in a distributed setting, the centralized approach can be a bottleneck. Since every primitive of the dataframe involved in reading or writing to the version history has to be rerouted through the GCN, execution of the application through the debugger is much slower. It would also take longer to hit the conditional breakpoints potentially making the whole debugging process tedious. An easy solution, and one that works with GoTcha as it is, would be to reduce the number of GoT nodes in the application during debugging. The user could debug issues in this much smaller application, fix the problems, and then scale the number of nodes back up. However, this approach might not be always possible and, therefore, changes to the debugging architecture might be needed to solve this problem.

To understand the difficulties involved, let us look at the flow of interactive debugging. There are essentially two modes that GoTcha executes in. First, a "free-run" mode, where the application executes as it would under normal conditions until it hits a breakpoint. Second, we have the slow and more deliberate "step-by-step" mode which activates when the free-run mode matches a breakpoint, or if the user is exploring execution paths. In the second mode, the user has control over the execution and is observing a small and very specific part of the entire application. The design of GoTcha is tailored towards the step-by-step mode. The central GCN helps coordinate these steps, and visualizations are created with this mode in mind. However, the same central GCN which enables total control during the step-by-step mode, is a bottleneck in the free-run mode when the number of GoT nodes becomes too large. A distributed approach to debugging would be as scalable as the distributed application it is debugging, but only during the free-run mode. However, such an approach would again have a hard time scaling with the number of nodes when the debugger has to control

the application in the step-by-step mode. This incompatibility of designs and the multiple modes that interactive debuggers work in, is the underlying reason why the advantages of interactivity in debugging distributed systems diminish with scale.

A possible solution is to change the architecture of GoTcha in each of the modes, matching the strength of each design with the mode that they work best with. In the free-run mode, nodes could communicate directly with each other, and log their activity with the GCN. Each node also receives the list of breakpoint conditions. When a break point is hit, the GCN receives this information and then instructs every other node to switch to the step-by-step mode, taking control of the application and handing it over to the user.

Scaling the User Interface

With a large number of nodes, we also encounter the problem of the user interface having too much information. The network topology is certainly going to be difficult to read making it difficult for the user to take a deep dive into the GoT nodes and explore specific execution paths. Conditional breakpoints become the only way to explore the execution meaningfully making the debugger strictly for finding bugs whose symptoms are already known. A possible solution is to use grouping algorithms to show the topology of the application concisely. Alternately, algorithms like PageRank can be used to show only nodes that are heavily connected.

The view of the version history at each node can have a lot of information when there is significant interaction with the node. For example, a view of the version history at a Grouper node, working with a thousand WordCounter nodes, could potentially have a thousand steps pending at Grouper and waiting to be stepped through. To enhance the navigation of execution, the user interface could allow users to attach breakpoints to the end of the steps that are pending, allowing the user to skip large batches of steps without necessarily having

to artificially promote specific the step that they wish to see, to the top of the pending list. Such a breakpoint would be a close match to the non-conditional breakpoints that exist in traditional debuggers.

9.7.2 Integration with Alternate Debugging Concepts

GoTcha is built as a stand alone system, over the GoT model, that helps debug the application by exposing the changes to the version history at each node. State changes within the application are observed from a version control point of view and is observed in broad strokes over several lines of code. The bugs to be found are, however, in these lines of code and integration with traditional interactive debuggers can help find these bugs. As such, GoTcha does not interfere with the use of traditional interactive debuggers at a single node. A single threaded interactive debugger can break down the state changes due to local execution and allow the user to debug the lines of code, while also being sure that the state cannot change in unpredictable ways from one line to the next.

Integration with non interactive forms of distributed debugging are also possible. For example, GoTcha, during the free-run mode, is similar to a tool for record and replay. When a conditional breakpoint is hit, it would be possible for the user to cycle through the previous steps and observe the previous states of the application along with the interactions that occurred between the nodes. Cycling through the previous steps is important because conditional breakpoints are usually used to find the execution point where the symptom of the error manifests. This may not always be the point where the error is. The user can find these errors by observing previous states of the application. If the user does not put a conditional breakpoint and executes the entire application in free-run mode, the entire execution is recorded and can be replayed. Existing tools and research on record and replay can add value to the free-run mode of GoTcha and make it a more powerful tool. The

integration of non interactive debugging tools would enhance the approach of finding errors during free-run as most of these tools deal with postmortem analysis of execution, while the interactivity of GoTcha would allow the user to observe errors during the exploration of live execution of the application.

Chapter 10

Experimental Analysis and Results

In this chapter, we will focus on observing the aspects of Spacetime that make it effective: update latency, and garbage collection.

10.1 Update Latency

We conducted two sets of experiments to show how the optimizations GoT, and Spacetime enable low update latency in shared-space applications. First, we ran microbenchmarks to compare the update latency in specific scenarios in Spacetime with the update latency in the same scenarios in alternate competing programming models. Second, we observe the effect the various stages through which an update passes, have on update latency.

10.1.1 Setup

To understand if the GoT model, and Spacetime, are suitable for applications that require low update latency, we ran micro-benchmarks that compared the update latency observed

in the implementation of GoT, Spacetime, against comparable alternate models. We chose the alternatives based on two dimensions that we discussed in Chapter 4: the type of the programming model – shared-state, and message-passing, and the consistency model they support – sequential consistency, and eventual consistency.

To represent shared-state programming models in these experiments, we chose a Redis cluster as our eventually consistent shared-state data store, and a MySQL server as our sequentially consistent shared-state data store.

To represent message-passing programming models, we chose two types of models. First, we chose a typical push only message-passing model (MP Push), similar in design to the push only approach to Spacerace described in Chapter 2. The second is a model that supports both push and pull communication by buffering updates at the server for each client (MP Buffered Pull). This model is similar in design to the Push with Isolation approach to Spacerace, again described in Chapter 2

Workload

Spacetime The Spacetime application consists of three types of nodes: reader nodes, writer nodes, and a single server node. The server node is the remote node for all reader and writer nodes. They synchronize over objects of the type BasicObject, having two dimensions: primary key 'oid', and 'create_ts' which stores the timestamp of object creation. The writer bot runs in a loop. In each iteration, the bot creates one timestamped object of type BasicObject and immediately pushes the change to the server. When all the objects have been created and pushed, the writer stops. The writer does not sleep between successive loops. The reader bot also runs in a loop. In each iteration, the bot pulls (not pull_await) from the server. A timestamp is recorded at the end of the pull and mapped to the list of objects of BasicObject currently under the reader bot's dataframe. The reader does not sleep between

iterations. After the bots receive all objects, they determine the update latency of each object using their creation timestamp and the timestamp when the object was first observed at the reader. The server performs no computation of its own. However, the dataframe at the node serves as the authoritative copy for the state of the benchmarks. Writer bots push new objects to the server node, and the reader bots read changes in the server node.

We ran the experiments with two variations of Spacetime. First is *Spacetime (fetch)*, where the fetch requests made to the server node by the reader node do not wait until any new update is available. Second is *Spacetime (fetch_await)*, where the fetch requests made to the server node by the reader node waits until there is an update that the reader node has not received. The second has a stronger consistency guarantee, with the trade of potentially having to wait for the new update.

MySQL: The MySQL setup consists of a single MySQL database, and two types of clients that connect to the database: reader clients, and writer clients. Both the reader and writer clients perform the same role as their Spacetime counterparts and synchronize writes and reads over rows of a single table with two columns: a primary key 'oid', and the 'create_ts' timestamp. Update latency of the records created by the writer clients is observed at the reader clients.

Redis: The Redis setup consists of a single Redis master database instance, several replica instances of Redis, and two types of clients written in python: reader clients, and writer clients. The writer clients write hash groups ('BasicHash::{oid}') with two keys in the group: a primary key oid and create_ts timestamp, into the master database instance. There are as many replica Redis databases in the cluster as there are reader clients. The reader clients read from the replicas. The synchronization between the replicas and the master is left to Redis. Update latency of the hash groups created by the writer clients is observed at the reader clients.

Push only Message Passing (MP Push): The MP Push setup consists of a single server node, and two types of clients that connect to the server node: reader clients, and writer clients. Both the reader and writer clients perform the same role as their Spacetime counterparts but synchronize in different ways. The writer client nodes also create objects of type BasicObject and push them to the server node. The server node receives these messages containing objects and distributes them to every reader node. The reader node connects to the server node and then actively listens for any incoming messages. When it receives a message, it records the update latency of the objects received.

Push and Pull Message Passing (MP Buffered Pull): The MP Buffered Pull setup is very similar to the MP Push setup, with two exceptions. First, the server node does not distribute the messages received to all the reader nodes immediately. Instead, it puts the messages into a client-specific update buffer. It also actively listens for any communication from the readers. The reader nodes pull changes from the server, which takes all the updates in that client's buffer, merges the updates and returns the merged delta update. The objects in the merged delta update are promptly observed, and their update latency is recorded.

Hardware and Network Conditions

We used two machines to conduct the experiments. One machine was an Amazon EC2 HVM instance located in Germany, running Ubuntu 20.04 with kernel 5.4.0-1009-aws with 1GB of RAM, and one core of an Intel Xeon CPU E5-2676 v3 @ 2.4GHz processor with no hyperthreading. We used this machine to run the server (Spacetimeserver node, MySQL, and Redis, MP Push server, MP Buffered Pull server). The second machine, used for launching the reader and writer clients, was a workstation running CentOS 7.5.1804 with kernel 3.10.0-862.11.6.el7.x86_64 with 252GB of RAM, and an Intel Xeon CPU E5-4650 v4 processor with 56 cores (112 with hyperthreading). This second machine was located in the West Coast of the US, at the University of California, Irvine. All the client nodes were launched from the

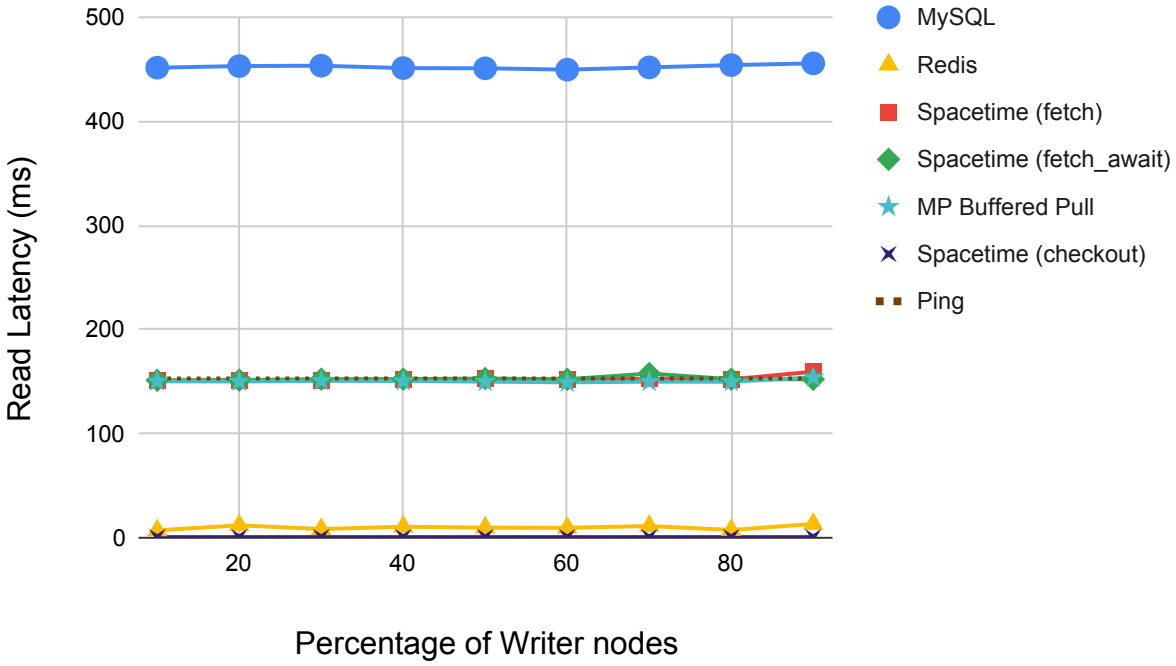


Figure 10.1: Median read latency vs write loads (All models).

same physical machine in order to make meaningful comparisons between read and write timestamps. Ping time between the two locations is typically 153ms, on average.

We used MySQL Community Edition Ver 8.0.17 and Redis 5.0.4 as the servers for their respective workloads. Python v3.8.2 was used in the EC2 instance to host the server node, and all clients were run on Python v3.7.2. The MySQL and Redis queries were both launched via CLI commands sent from their respective python libraries. The cache size of MySQL was increased to keep the table cached in memory, removing any need for slow disk access.

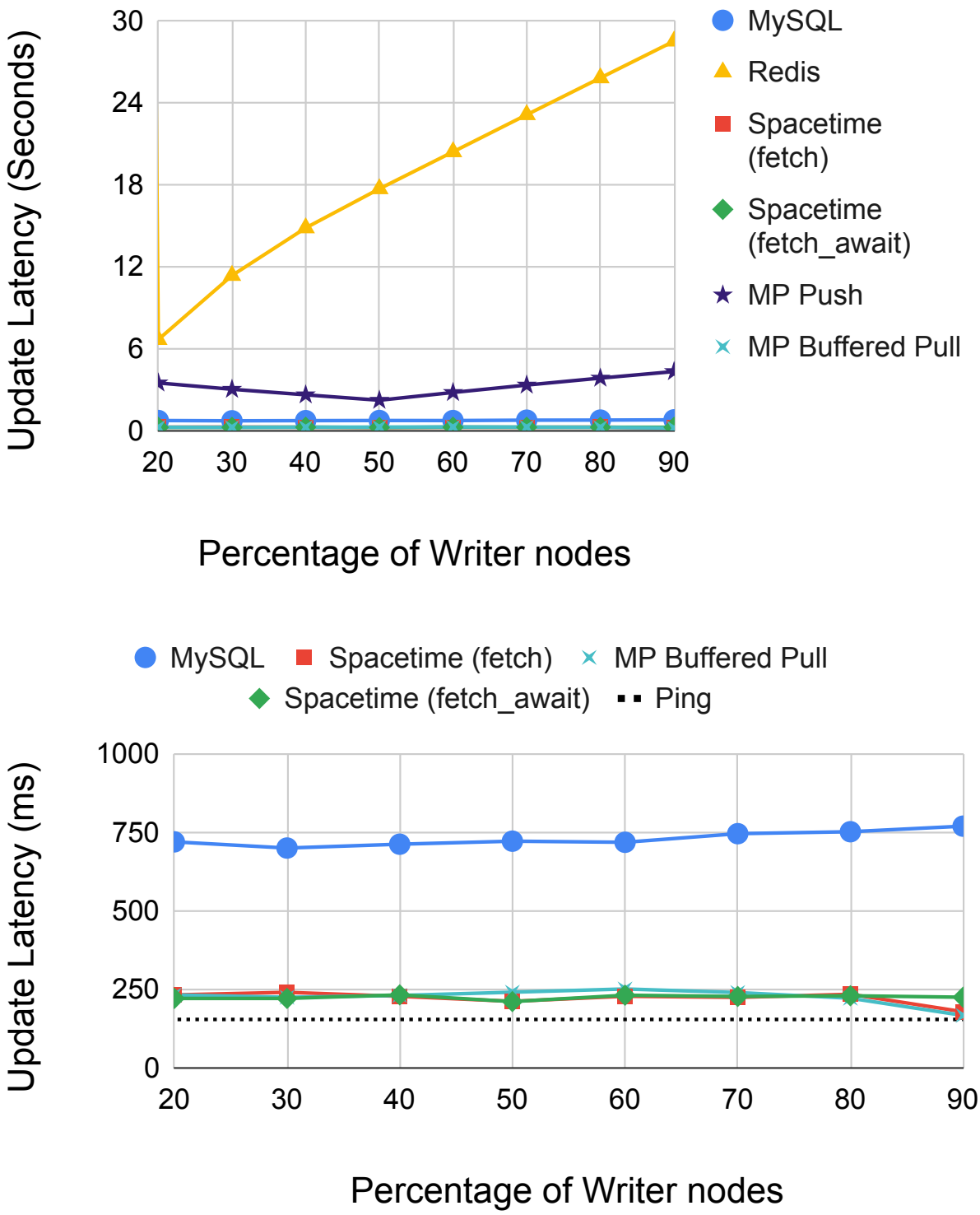


Figure 10.2: (A)Median update latency vs write loads (All models). (B) Median update latency vs write loads (low latency models).

10.1.2 Experiment 1: Spacetime vs Baselines

Experiment 1A: Update Latency vs Write Load

Taking a cue from the load testing Yahoo Cloud Serving Benchmarks [28], we observe the update latency of all six (two Spacetime, and four competing) setups at varying distributions of readers and writers going from read-heavy to write-heavy scenarios. The independent variable is the percentage of nodes that are writers, from 20% to 90% of the nodes, in increments of 10%. The number of objects/records/hashgroups was kept constant at 100. The total number of client nodes (readers + writers) was kept at 20. So in the read-heavy scenario, 4 of the client nodes are writers creating objects that 16 client nodes are reading, while in the write-heavy scenario, 18 of the client nodes are writers, and 2 are readers. The dependent variable measured was the median update latency for all updates in each scenario.

Results: Figure 10.1 shows us the read latency for the reader clients/nodes in each setup. We can see that Redis has the low read latency (10ms) as the Redis database that the reader node is reading data from is present locally, and highly available. Variations of Spacetime, MP Push and the MP Buffered Pull setup have around the same read latency (152ms), which is exactly at the average ping observed between the server and the client machines. This means that these scenarios do not need to wait at the server for their response. However, since there is a local replica of the state at each node, much like Redis, the read latency from the local replica at the dataframe, denoted by Spacetime (checkout) in the graph, is also very low (0.5ms). The MySQL setup, however, has a large read latency. This is because, MySQL cannot respond with delta updates, and must send the entire state across. Further more, the transactional nature of MySQL can slow the reads while writes are concurrent. MP Push setup did not have any read latency as the reader nodes in that setup receive updates directly from the server and do not make pull requests of their own.

Looking at update latency in Figure 10.2 (A). We see that Redis has extremely high update latency. The read latency for Redis is low, and the reader client can perform many read requests operations each second. However, a majority of these reads do not have any new updates for the reader. This scenario is common for Eventual Consistency. There are two reasons Redis nodes take significant time to receive updates. First, the Redis replica delays the application of updates it has received from the master when read requests are in bound. This choice caters to high availability but the cost of update latency. Second, the master replica employs asynchronous push communication to keep the replicas in sync. Since the machine hosting the master database has only a single core and thread, distributing these pushes to multiple clients is still sequential. This pattern is also seen in the MP Push scenario. Since the server node in the MP Push is also single-threaded, the server cannot keep up with the distribution of new updates. A multi-threaded server can potentially solve this bottleneck, but therein lies the problem with scaling this solution. A single server should not be responsible for the distribution of updates to all its clients, a design point strictly adhered to in RESTful architectures.

We can take out the high update latency scenarios: MP Push, and Redis, and take a closer look at the update latency of the remaining scenarios. The median update latency for all remaining setups are less than one second (Figure 10.2 (B)). MySQL has the highest update latency at around 750ms. We see that read latency is around 450ms and update latency is 750ms. The update latency is at least as high as the read latency. The remaining contributors to update latency are the packing and transmitting cost for the writers, which are concurrent to the read latency, and the cost of transferring, unpackaging the update in the reader client and introducing it to the local state. Since MySQL cannot communicate with delta updates, a large amount of data is transferred increasing this cost. There is also some cost to introducing the update to the MySQL tables and maintaining the state at the database.

The update latency for all the other setups are comparable at around 220ms. With read

latency at 152ms and a ping of 152ms, conflict resolution, packaging and unpacking the updates at both the server and the reader clients takes around 70ms. The substantial reduction is attributed to the use of delta updates. While non-existent in MP Buffered Pull, conflict resolution is invoked at each update received at the Spacetime server as they are all considered concurrent updates (the writer nodes never pull the state from the server). Therefore, the 70ms in update latency also includes conflict resolution and the creation of the merge nodes.

The percentage of writer nodes does not affect the update latency in any setup other than Redis, a product of the design for high availability.

Experiment 1B: Update Latency vs Node Count

In the second experiment, we observe the effect the number of nodes in the system have on the update latency of all six setups. The independent variable here is the number of client nodes in the system: 10, 20, 50, 100. Although by database standards the number of clients is low, it is quite realistic for a multiplayer game or a multi-agent simulation to have at most a 100 clients working together. The read/write ration of the clients nodes was set to 50% and the object count was set to a 100 objects. The dependent variable measured was the median update latency for all updates in each scenario.

Results: Figure 10.3 shows that there is a lot of variation in the update latency for both Redis and MP Push with an increase in concurrent clients which means that both these setups are throttled by having to respond to incoming connections. As discussed, the server being single-threaded severely limits the capabilities of push only communication.

The rest of the scenarios are not affected by the number of clients, showing the scalability of such systems.

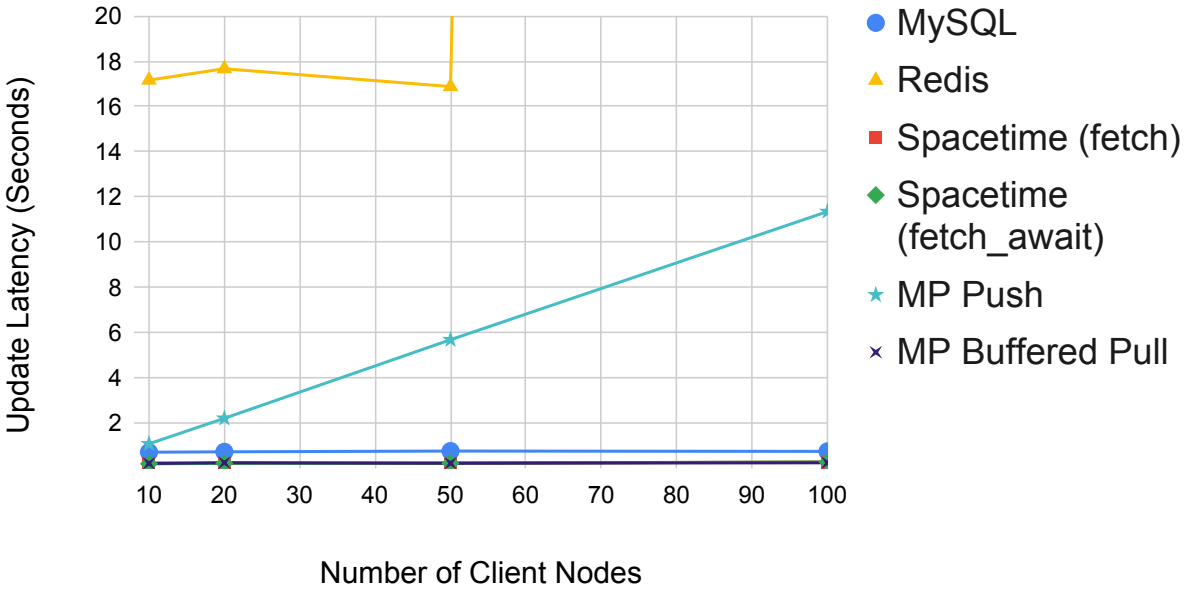


Figure 10.3: Median update latency vs Number of nodes.

10.1.3 Experiment 2: Breakdown of update latency in Spacetime

Having seen the low update latency in Spacetime, when compared to other models, we take a closer look at the contribution of each step in update latency. To this purpose, we executed microbenchmarks to instrument the functions corresponding to these steps. We present our experiments and results.

We instrumented commit, checkout, push, fetch, accept_push, and accept_fetch processes in Spacetime. The setup used is similar to the Spacetime (fetch) scenario in the previous examples, with a server node in Germany, and the reader, writer client nodes in California, US. The number of client nodes was set to twenty with ten nodes as writers, and the remaining as readers. A hundred objects of BasicObject were synchronized between the writers and readers via the server node. We ran the same setup ten times and collected all the timings.

Operation	Median Time (ms)	Standard Deviation	99.9%-ile Time (ms)
Commit	0.728	0.194	1.380
Garbage collection	0.168	0.064	0.645
Packaging update	0.176	0.083	0.959
Send update (n/w)	151.767	2.642	164.208

Table 10.1: Time taken by operations at the writer node

Operation	Median Time (ms)	Standard Deviation	99.9%-ile Time (ms)
Acquire write lock	0.010	0.278	3.626
Write update	0.013	0.024	0.408
Resolve conflicts	0.175	0.487	4.174
Garbage collection	0.213	0.301	3.580

Table 10.2: Time taken by operations at the server node, when receiving updates.

Operation	Median Time (ms)	Standard Deviation	99.9%-ile Time (ms)
Acquire read lock	0.009	0.011	0.166
Packaging update	0.393	0.287	4.622

Table 10.3: Time taken by operations at the server node, when responding to fetch.

Operation	Median Time (ms)	Standard Deviation	99.9%-ile Time (ms)
Fetch Request (n/w)	152.485	2.976	178.213
Acquire write lock	0.020	0.046	0.126
Write update	0.032	0.082	0.249
Resolve conflicts	0.146	0.116	2.465
Garbage collection	0.307	0.323	3.106
Checkout	0.506	0.280	3.272

Table 10.4: Time taken by operations at the reader node.

Results

Tables 10.1 – 10.4 show the latency of the operations in the writer, server, and reader nodes during push and fetch operations. It is important to note that these are not process times, where thread sleeps are ignored, but actual time including any time that the threads involved were preempted and/or rescheduled by the operating system. Finally, we cannot directly compare the time taken for the operations at the server and the clients as they are executed on different machines with different processing capabilities. The purpose is to illustrate the critical paths of both push and fetch operations.

Starting with the write side, in Table 10.1, we see the time taken by four operations that are executed by the writer nodes: Commit, garbage collection, packaging the update and sending the update over the network. The most significant time is spent in sending the update over the network. This network latency is unavoidable as the average ping between the server and the node is 153ms. Garbage collection occurs at the end of every commit and invokes a very low overhead (median 0.168ms).

Table 10.2 shows the time taken by the operations at the server when an update is received from the writer. A lock is first acquired. There are ten writers that are contesting for this lock, and therefore we can see that while the median time spent waiting at the lock is low (0.01ms), the standard deviation is relatively high. However. 99.9 percent of the time, the time spent at the lock was less than or equal to 3.625ms. After the lock is acquired, the update is written. This update is quick as it merely extends the graph and does not require further processing of the delta. Conflicts have to be resolved. In this setup, every update pushed from the writers is a conflict as the writers never read from the server. However, 99.9 percent of the conflicts resolved were under 4ms of time with the median time taken being less than a millisecond (0.175ms). Garbage collection at the server is the most intense of the operations as it has to deal with the states pushed by and read by multiple nodes. 99.9

percent of the garbage collection operations took less than 3.5ms with the median being less than a millisecond again (0.213ms).

Table 10.3 shows the time taken by the operations at the server when a fetch request is received from a reader node. A read lock is acquired, which takes very little time (median 0.009ms, 99.9 percentile of 0.166ms) as the locks are at the level of each version, and only garbage collection can conflict with such a read. Packaging the update takes a median of 0.393ms, with 99.9 percent of the operations taking less than 4.622ms.

Finally, Table 10.4 shows the time taken by the operations at the reader node. The reader node makes a fetch request which is transferred over the network to the the server where the operations listed in Table 10.3 occur, and a response is sent back. This entire round trip time, including the time spent at the server takes a median of 152.485ms with 99.9 percent of the operations taking less than 178.213ms. As we can see from the time taken by the operations at the server and the average ping being around 153ms, the majority of the time is spent over the network. When the update is at hand, the reader acquires a write lock, which does not contest with any operation, making it fast (median 0.02ms, 99.9 percentile 0.126). The update is then written to the version graph and conflicts are resolved. Since the reader does not make any updates of its own, there are never any conflicts. Therefore, the time instrumented for conflict resolution is the time taken for detecting conflicts. As we can see, the detection of conflicts is fast (median 0.146ms, 99.9 percentile 2.465ms). Garbage collection, and checkout also usually take less than a millisecond.

The take away from this, is that network time becomes the significant cost in Spacetime when the latency between twenty nodes and the server is more than even a 10ms. The operations run at a median time of less than a millisecond for all operations that are not over the network. The true additional cost of update latency is in the asynchronous nature of the push and fetch operations being made. This can be solved using a push model where updates can be pushed as soon as they are received. However, experiments 1A and 1B show

that the push approach does not scale with the number of nodes in the system.

10.2 Garbage Collection

In the previous section, we have shown that Spacetime performs remarkably well when trying to transport updates over the network between nodes. However, that is not the only aspect of feasibility. If the memory usage of the nodes, keep increasing over time, then, even if the update latency achieved is low, the system is not feasible for shared-space applications.

A typical concern when using version graphs is the ever-increasing versions or revisions maintained by the version graph. In file-based systems like Git, where updates happen in the order of hours, days, or weeks, this increasing set is not that critical. However, when used for highly mutable object replication, where updates happen in milliseconds, memory usage can quickly be a problem if not addressed. In Chapter 7, we discussed the garbage collection strategy of Spacetime. In this section, we show the effectiveness of garbage collection.

10.2.1 Setup

Workload

Like in the setup for Spacetime (fetch) in the previous section, the Spacetime application for these benchmarks also consists of three types of nodes: reader nodes, writer nodes, and a single server node. The server node is the remote node for all reader and writer nodes. They synchronize over objects of the type BasicCounter having two dimensions: primary key 'oid', and 'count' which stores a monotonically increasing counter. The writer bot runs in a loop. In each iteration, the bot updates the counter of one BasicCounter object by one. The writer does not sleep between successive loops. The reader bot also runs in a loop. In

each iteration, the bot pulls (not pull_await) from the server. The reader also does not sleep between iterations. After running continuously for 1 minute, the application is shut down. The server performs no computation of its own. The dataframe at the node, however, serves as the authoritative copy for both the reader and the writers. It performs the additional task of merging the counter values provided by different writers.

At the end of each change in the version graph at the server and the clients, we recorded the number of versions in the version graph. If these number of versions is growing over time, then the version graph will eventually be too big to fit into memory causing failure. The number of versions has to remain stable and constant over time for the system to be feasible.

Hardware and Network Conditions

One machine to conduct the experiments and host the server, reader, and writer nodes. The machine was a workstation running Ubuntu 20.04 with kernel 5.4.0-29-generic with 32GB of 3000MHz DDR3 RAM, and an Intel Haswell-E i7-5820K CPU @ 3.30GHz processor with 6 cores (12 threads with hyperthreading). Python v3.8.2 was used to run all the Spacetime nodes.

10.2.2 Experiment 3A: Version count vs. the number of objects

In this experiment, we fixed the number of nodes communicating with the server to 20. The number of readers and writers were equal at ten each. We varied the number of objects that the readers and writers are synchronizing on [1, 10, 100, 1000]. The number of versions at the server was recorded after every put and read operation. The experiment was run for one minute for each scenario.

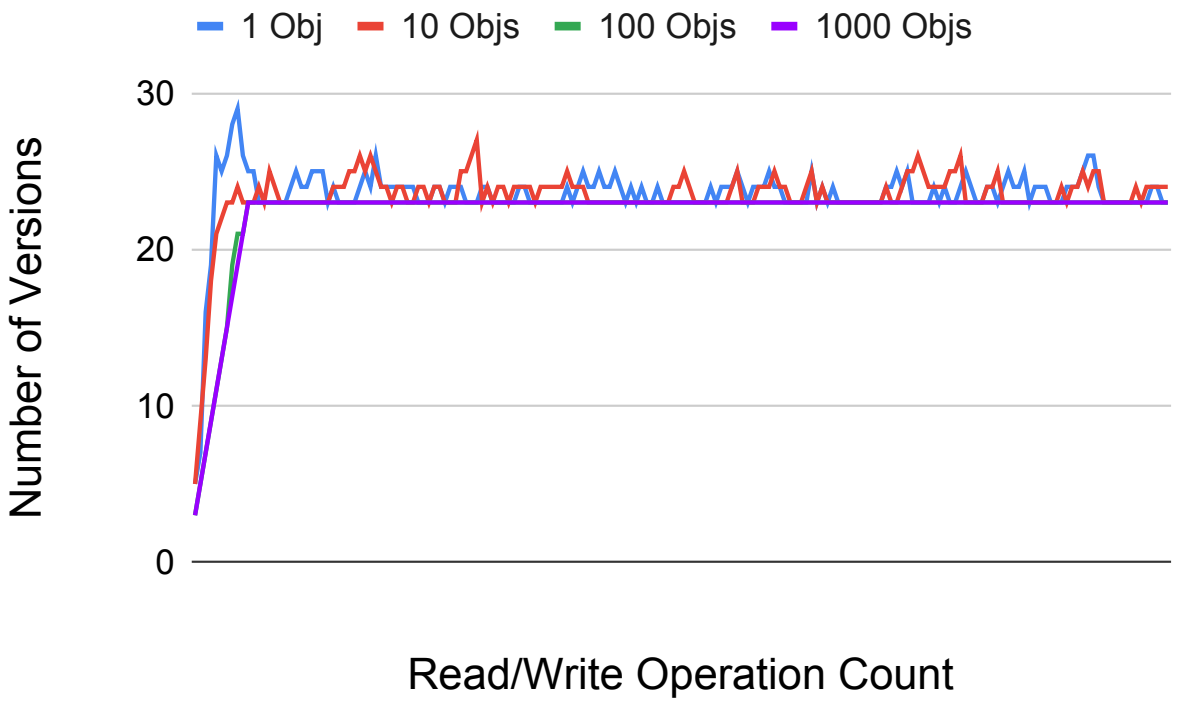


Figure 10.4: Version count at Server with varying number of objects.

Results

Figure 10.4 shows the number of versions in the version graph at the server node over the number of read/write operations on the server version graph, for each setup with varying number of objects. As we can see, the number of versions in the server increases up to a certain point. Beyond this point, the version count remains more or less steady. Fluctuations can be seen in the number of versions when there are low number of objects. The application code at each writer node iterates over all BasicCounter objects in the dataframe and updates count. With more number of objects, the application code is spending more time updating the objects. This means that the rate at which the version graph at the server receives push requests and therefore new versions is higher with lower number of objects. When a reader makes a fetch request, the reference map at the server is updated for that reader to a the HEAD version, but the old reference is not deleted until the reader acknowledges the update.

With a high volume in writes, multiple updates can occur in the version graph before the acknowledgement is received from the readers. This means that there is a temporary increase in the number of references held in the reference map at the server. Every node can have up to two references in the reference map. This, however, still bounds the number of versions to the order of the number of nodes in the system.

The number of objects being synchronized does not change the number of versions maintained in the version graph. This is because the entire state is versioned, and not the objects. Each version can have varying number of objects.

10.2.3 Experiment 3B: Version count vs. the number of nodes

In this experiment, we fixed the number of objects being updated with the server to 100 and vary the number of nodes synchronizing with the server [2, 20, 50, 100]. Half the number of nodes are assigned to be readers with the rest being writers. The number of versions at the server was recorded after every put and read operation.

Results

Figure 10.5 shows the number of versions in the version graph at the server node over the number of read/write operations on the server version graph, for each setup with varying number of nodes. As we can see, the number of versions in the server increases up to a certain point. Beyond this point, the version count remains steady. The number of versions the version graph at the server stabilizes at is different with different number of nodes. We can see that the number of versions being maintained is roughly equal to the number of nodes synchronizing with the server. However, there is no unbound growth in the number of versions in the graph.

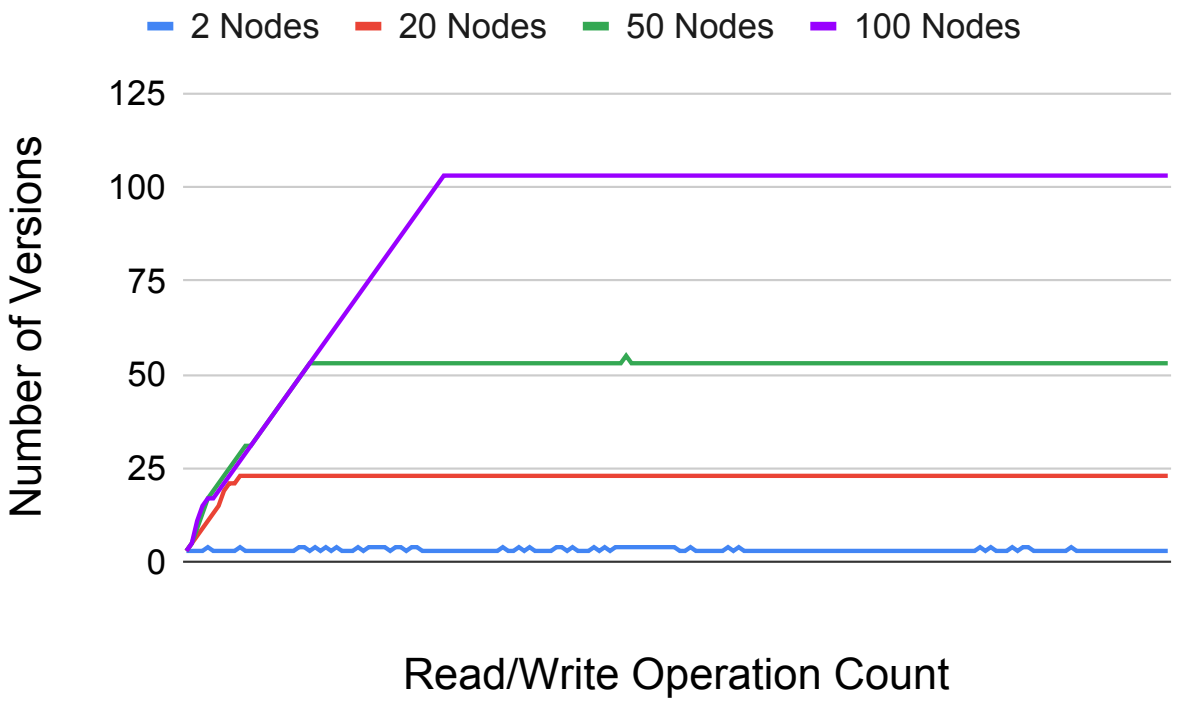


Figure 10.5: Version count at Server with varying number of nodes.

10.3 Experiment 4: Garbage Collection in Peer to Peer

In the previous section, we see that the garbage collector in Spacetime keeps the number of versions in the version graph to a bounded value. In those experiments, the conflict resolution strategy used was big-step merge and the network topology was server-client. Peer to Peer networks, however, require a different conflict resolution strategy, called small-step merge and with that a new garbage collection method. While the garbage collector for small-step is still in preliminary development, it works remarkably well in simple peer to peer cases. In this experiment, we observe the effect of small-step merge on garbage collection.

10.3.1 Setup

Workload

In this experiment, we use two nodes Peer 1 and Peer 2 that have identical application code. They synchronize with each other over objects of type BasicObject having just dimension: primary key 'oid'. Both the peers runs in a loop. In each iteration, the peer creates a random object of type BasicObject. It then commits the object, and pushes the update to the other peer node. Since the nodes push updates asynchronously to each other, the setup is not server-client. Each peer node is using small-step merge as the conflict resolution strategy, and the corresponding garbage collection prototype discussed. Each of these peers run the loop described for a minute.

Hardware and Network Conditions

One machine to conduct the experiments and host both Peer 1 and Peer 2 nodes. The machine was a workstation running Ubuntu 20.04 with kernel 5.4.0-29-generic with 32GB of 3000MHz DDR3 RAM, and an Intel Haswell-E i7-5820K CPU @ 3.30GHz processor with 6 cores (12 threads with hyperthreading). Python v3.8.2 was used to run all the Spacetime nodes.

Experiment and Results

The two peers executed the same workload described above for one minute. At the end of each loop, the number of versions in the version graph was recorded. We ran the experiment both with and without the garbage collection.

Figure 10.6 shows the number of versions in the version graph at Peer 1 with both garbage

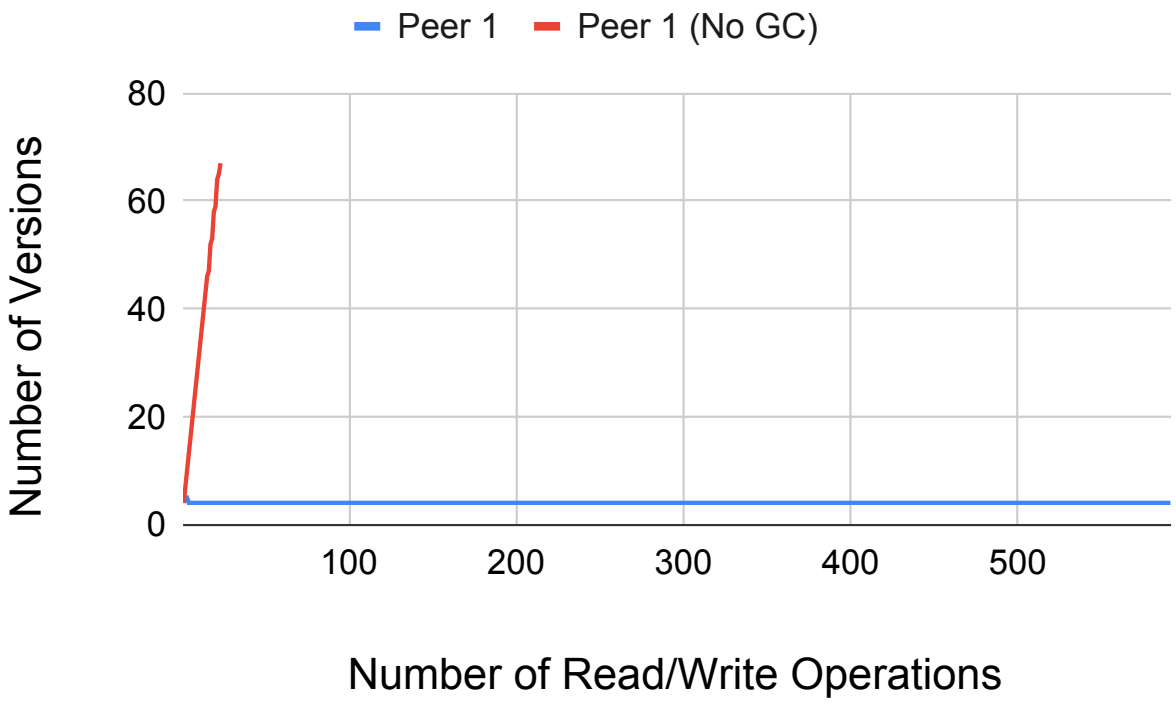


Figure 10.6: Version count in peer to peer with two peers and no garbage collection.

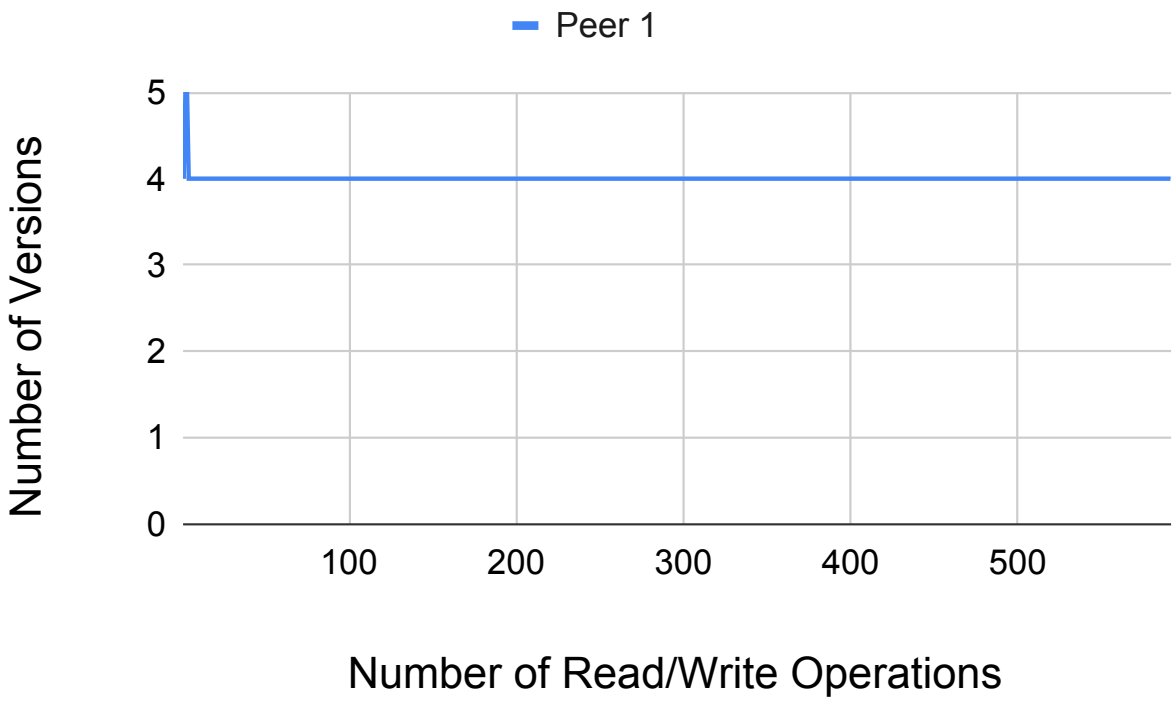


Figure 10.7: Version count in peer to peer with two peers and garbage collection.

collection turned on and off. The number of versions of Peer 2 were identical to that of Peer 1. On the x-axis we have the total number of operations performed in one minute. On the y-axis we have the total number of versions in the version graph, recorded after each operation. As we can see, when there is no garbage collection, the number of versions in the version graph increase rapidly and is unsustainable. The rise in the number of versions is same for both peer nodes. With garbage collection turned on, however, the number of versions flattens out and stays at around four versions. The four versions are the the ROOT, Peer 1's latest update, concurrent to Peer 2's latest update, merged to a common merge version. This can be seen in Figure 10.7.

Both workloads – with garbage collection, and without it – were run for a minute and the x-axis shows the number of operations that were executed in that minute. We can see that without garbage collection, the number of operations are much much lower. Each operation takes more time as the number of versions increases as each update has to be combined in small-step fashion with an ever-growing set of updates, making the system progressively slower.

10.4 Conclusion

In this section, we look at the Spacetime from the view point of its efficiency in the propagation of updates (update latency), and its feasibility for long-lasting applications with highly mutable data (garbage collection).

We see from experiment 1A that Spacetime achieves as low update latency as competing message-passing programming models. The advantages, however, are causal consistency, an observable replicated data store for the state of the system, and nearly stateless communication that supports isolation. We saw that shared-state programming models fared worse as delta

updates are hard to calculate (MySQL) and eventually consistent data stores optimize for read latency at the cost of update latency.

We also saw from experiment 1B that the number of client nodes, at least up until 100 nodes did not have much affect on update latency in Spacetime. The server node, even running on a single-threaded machine, was able to keep up with the requests from a 100 nodes. Competing systems such as Redis and push-based programming models are notoriously CPU intensive, and perform poorly with lower resources. This is primarily because the server is responsible for pushing the updates to its clients. The more number of clients, the more work the server has to do for each update. In a pull-based model, the clients take updates at the rate they can handle, and multiple updates are often merged together within a single response. Therefore, from experiment 1A and 1B, we can say that Spacetime achieves its goal of low update latency.

Experiment 2 allowed us to look at the processes involved in the path of update latency and the costs associated with each of them. Network costs are typically the highest, and out of the control of Spacetime, with the rest of the processes often taking a median time of less than a millisecond. Garbage collection and conflict resolution constitute significant percentages of the time spent.

Experiment 3 shows us the effectiveness of the garbage collector. We have already seen from experiment 2, that the cost for garbage collection is less than a millisecond when there are twenty clients reading and writing to a version graph. In experiments 3A and 3B, we see that the number of versions in a version graph stabilizes over time and does not increase indefinitely. The maximum number of versions is to the order of the number of nodes reading and writing from the version graph. The number of objects in the state, however, does not affect the number of versions. In experiment 4, garbage collection in the peer to peer mode of Spacetime, using the small-step merge, was observed in a limited setup containing two peers continually and concurrently pushing updates to each other. With garbage collection on, the

version graph stayed at a constant number of versions in both peers. When garbage collection is turned off, the number of versions rockets to unsustainable levels very quickly. This explosive growth further deteriorated the system's performance by increasing the time taken for each operation to complete, increasing update latency. With garbage collection, however, many more operations were performed as operations are much faster and therefore, update latency is lower. Overall, experiment 3 shows us that Spacetime is feasible for long-lasting shared-space applications.

Chapter 11

Conclusion and Future Work

11.1 Summary

Shared-space applications, like multiplayer games, distributed simulations, and geo-replicated databases, are hard distributed systems to engineer. They are often characterized by highly-mutable, long-lived, shared environments that are simultaneously and independently changed by many nodes in the system. While the CAP theorem and the updated PACELC theorem have made it clear that consistency – getting the latest write, and latency – getting a fast response to a remote request are competing goals that engineers have to choose one or the other to optimize, shared-space applications need both. Since the state of the system changes constantly, and decisions need to be made on these updated state, consistency is important. However, these updates need to reach the nodes as quickly as possible as well. Therefore, latency is also important. To address this specific concern, I defined the term update latency.

Update latency is the time taken for an update generated at one node, to reach the remote node where it is going to be used. It is not consistency, as there is a notion of time, and it is not latency as there is a notion of what update is obtained.

Chapter 2, takes a good look at the constituents of update latency and identifies that the determining factor in update latency is the cost of communication. Since network latency is unavoidable, it is in the application's best interest to communicate only that which is necessary. There are necessarily three rules that must be followed to achieve this.

First, nodes must take or receive only the information that they need. This is typically called interest management and is quite a common tactic used in the database world. For example, the SQL querying language allows database clients to use relational algebra when reading the tables to select the exact dynamically generated slice of information relevant to them.

The second rule is that nodes must take or receive only information that they do not have. This rule is called delta updates, and it includes both state deltas and operation based messages. Many systems, such as databases, do not respond to queries in the form of delta updates. It is quite tricky, without creating complex constructs, such as temporal tables, for a node to get only information that is new since the client's last time executing a query on the tables. However, message-passing programming models specialize in sending messages or operations in the form of delta updates.

The third rule is that nodes must take or receive information only when they need it. This rule is called isolation. If a node processes information at a rate slower than the rate at which updates are made, then all updates do not need to be received as they are created. The node should be able to choose the rate at which it synchronizes. Isolation can allow for the numerous delta updates intended for a node to be merged, removing intermediate steps that are of no use. Isolation is also useful for those applications that need to or choose to deal with partitions. If a node suffers a network partition, messages cannot be sent or received. When the node comes back into the network, it should not be inundated by all the messages it did not receive. Further, it should not need to re-synchronize from the very beginning. By supporting isolation nodes that rejoin the network after a partition can synchronize their state from their state, allowing for partitions to be gracefully handled.

I showed in Chapter 2 that the engineering efforts to build a system that caters to all of these requirements are rudimentary forms of version control. It is with this observation that I present the Global Object Tracker (GoT), a programming model that models distributed computing for spared space applications as a distributed version control system.

In Chapter 4, I showed the basics of GoT. A GoT application consists of many independent nodes called GoT nodes. Each GoT node, like Git, owns a repository of objects called the dataframe. The dataframe represents the replicated shared space at each GoT node. The application code at each node can read objects from the dataframe, and make changes to the state. The changes made are staged in the dataframe and written into a version control graph on a commit. This version graph is a directed acyclic graph containing the history of changes written to it. It is contained within the dataframe and is local to the GoT node. Got nodes communicate changes to the version graph with remote GoT nodes via both push and fetch methods. Both forms of communication pick up delta changes from one version graph and apply it to another. Concurrent updates are detected as conflicts, and resolved via a programmer-defined three-way merge (similar to Git) and added to the version graph in a process called delta transformation.

GoT, like typical version control systems, preserves a notion of time by enforcing causal ordering to the updates within the version graph. It enforces causal consistency, which is widely considered to be the weakest consistency that still retains a notion of time.

GoT is a hybrid programming model that shares traits with both message-passing programming models and shared-state programming models. Like shared-state models, there is a notion of a repository of objects in GoT. However, this repository is replicated and independently updated at each node, much like message-passing programming models.

GoT is implemented practically in a framework called Spacetime that is written in python and available for use freely. Spacetime makes many optimizations on top of the GoT programming

model to make it both feasible and optimal for update latency. In Chapter 5, I explore the optimizations that are made in Spacetime for update latency.

Version control systems, whether file-based or in-memory and object-based, are excellent ways to track the evolution of state over time. They explicitly track the causal relations between states and allow for deterministic reasoning over concurrency. Concurrent updates are reconciled, when needed, by merge functions. The most popular style used is the three-way merge, which allows for rich semantic reasoning over the right composition of concurrent updates. There are, however, some scenarios in which version graphs are unmergable using three-way merges. These scenarios, which mainly occur in peer-to-peer, are rare for file-based version control systems but can be quite common in distributed computing.

In Chapter 6, I showed that the root cause for the failure of traditional 3-way merges (what I called big-step 3-way merges) was in the loss of information when multiple updates are combined during a merge. This composition of updates makes it difficult for version control systems using it to, in specific scenarios, find a single least common ancestor for merge. The critical insight there was that intermediate versions that capture intermediate states is needed. I proposed an alternate approach in peer to peer, called small-step 3-way merge, where I merge updates one step at a time, allowing us to preserve causal orderings while not composing updates together prematurely. Using a small formal model, I showed that by enforcing these constraints, the resulting version graph is valid. i.e., the versions that are stored are the correct application of the updates that came before it, and that the version graph converges to a single HEAD version.

The pure form of GoT, however, is impractical for real systems. Unchecked version growth can quickly put a strain on resources. Sharing every intermediate version of the version graph on synchronization is wasteful and inefficient. In Chapter 7, I show how Spacetime eliminates this problem by aggressively tackling unchecked version growth using a reference counting garbage collector that merges multiple deltas that have already been seen into a

single large delta. The side effect of having this garbage collector is that nodes have to keep a small amount of the state of the other nodes that interact with it. Intermediate versions of the graph are eliminated before synchronization to make efficient use of network resources. Additionally, I propose initial steps towards building a garbage collection for peer to peer networks using small-step merge. This garbage collection for small-step has not been fully verified and is planned for the immediate future.

GoT and Spacetime provide full support for both isolation (in the form of fetch communication), and delta updates (as diffs between versions). Simple interest management is implemented, by design, in Spacetime, as nodes can choose the types of objects that they can synchronize on. However, that does not support dynamic interest management. In Chapter 8, I introduced the concept of Predicate Collection Classes, PCCs for short. PCCs are a declarative mechanism of selecting objects from collections, reclassifying them along the way. PCCs are both classes and specifications of collections of objects of those classes. Composition of collections can be expressed very easily using concepts from relational algebra such as subsetting, projection, cross product, union, and intersection. PCCs are useful for filtering and manipulating collections, including when the elements of those collections may behave differently depending on which collection they are placed. The type system in Spacetime can be augmented to use PCCs as the declarative specification of types. This allows nodes to synchronize over dynamic collections of objects, further reducing the transfer of redundant information.

Another advantage of modeling distributed computing as a version control system is the increased observability of the system. To demonstrate this observability, and explain this use, I discuss the creation of an Interactive debugger on Spacetime, called GoTcha in Chapter 9. Interactive debuggers for distributed systems is an understudied area of research. In the chapter, I discuss the major goals that interactive debuggers for distributed systems should meet. In addition to exposing state changes at a node through local processes

like traditional interactive debuggers, interactive debuggers for distributed systems should expose the communication between nodes, and the integration of this information that is communicated, at each site. The debuggers should be able to expose these information exchanges while giving the user complete control over the execution of the system. In order for interactive debuggers to meet these requirements, support is needed from the underlying programming model. I discuss the specific features in the GoT programming model that facilitate interactive debuggers and put our theory to test by describing the implementation of GoTcha. I discuss the design of GoT, GoTcha, and describe a simple debugging process using the example of a distributed word frequency counter.

Finally, in Chapter 10, I present several experiments that show the efficiency of the optimizations in GoT for update latency when compared to competing programming models, the feasibility of Spacetime, and the effectiveness of the version garbage collector. Spacetime performs as well as or better than both message-passing programming models and shared-state programming models (both sequentially consistent and eventually consistent) with respect to update latency. The garbage collection strategy used in Spacetime is a low overhead approach that bounds the number of versions in the version graph at any node to the order of the number of remote nodes synchronizing with it.

The underlying design choices and the feasibility brought by effective garbage collection, make GoT and its implementation, Spacetime highly suitable for engineering causally consistent, long-running, highly mutable shared-space applications.

11.2 Discussion and Future Work

In this dissertation, I do not touch upon the human aspects of using a programming model. While I believe that GoT is an intuitive model for developers to use, user studies have not

been conducted to validate this hypothesis. In my opinion, as the object replication, in GoT, closely follows the rules of systems that they already use – distributed version control –, the model should be intuitive to follow and implement upon. The primitives supported by the dataframe allow for developer-controlled communication over the network, the semantics of which, including conflict resolution, are clearly defined. The underlying complexity of maintaining consistent replication, garbage collection, and delta updates are hidden from the programmer, and only the manipulation of shared data and the control flow of communication is left in the hands of the programmer.

Further, there are programming models with similar end-user interfaces. For example, frameworks such as JavaScript Parse platform [1] and Google Firebase [2] offer save and load mechanisms for the live synchronization of objects between nodes in the system. While they use push-based publish-subscribe and not version control, the interface for communicating changes between nodes is similar to that in GoT. These frameworks are quite successful and used by many developers in mobile and web applications. Extensive user studies could, however, validate this hypothesis in the future.

Version control as a programming model is useful when there is a shared-space that is mutated by one or more nodes to the benefit of all other nodes. It does not fit the applications that employ distributed programming as a means to divide and conquer a large task. Let us take, for example, a word counting map-reduce. Ideally, the large initial state with multiple lines of code has to be divided and distributed to workers that compute over the lines they receive. Each worker should only receive the line that they need to process. I define this as interest management in this dissertation and can be achieved using PCCs (Chapter 8). However, using PCCs, all we can do is to pre-partition a list of lines for each worker node. In typical map-reduce, a line is picked up by a free worker and processed. The workers that get shorter lines or work faster pick up more work throughout execution. Such a control-flow

[1] https://docs.parseplatform.org/js/guide/
[2] https://firebase.google.com/

mechanism cannot be easily applied to GoT, even with PCCs, as that would require some form of transactions. When a node reads a state and picks up a line, that line must no longer be available for other nodes to read or use. In a shared-space that becomes hard to achieve as reading the state (reading a line) also modifies the state (setting the state such that no other node will process that line). I call this operation 'take'. The use of triggers (similar to database triggers) that allow programmers to write atomic functions that are executed with a read or a write can be explored in the future as a way to introduce 'take' semantics to state replication. While version control can theoretically help improve the update latency of these applications that require 'take' semantics, the use of version control in these applications is mostly unexplored.

Finally, Spacetime is currently written in python. In the future, I hope to see implementations of the core framework in multiple languages, including, and not limited to, C++, Java, C#, and Javascript. The transfer protocol of Spacetime is language-agnostic, which means that heterogenous systems with nodes written in multiple languages can communicate over a shared data model, using the GoT programming model. I am particularly interested in an implementation in javascript for the browser, which would allow the GoT programming model to be a viable alternative to asynchronous javascript. I envision GoT supporting long-lasting, highly-mutable, shared-space applications like browser-based multiplayer games, collaborative document editors, and more, on the browser.

Bibliography

[1] D. Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer*, 45(2):37–42, 2012.

[2] R. Achar, P. Dawn, and C. V. Lopes. Gotcha: An interactive debugger for got-based distributed systems. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2019, page 94–110, New York, NY, USA, 2019. Association for Computing Machinery.

[3] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, Mar 1995.

[4] P. S. Almeida, A. Shoker, and C. Baquero. Efficient state-based crdts by delta-mutation. In A. Bouajjani and H. Fauconnier, editors, *Networked Systems*, pages 62–76, Cham, 2015. Springer International Publishing.

[5] P. S. Almeida, A. Shoker, and C. Baquero. Delta state replicated data types. *Journal of Parallel and Distributed Computing*, 111:162 – 173, 2018.

[6] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica. Probabilistically bounded staleness for practical partial quorums. *CoRR*, abs/1204.6082, 2012.

[7] K. Banker. *MongoDB in action*. Manning Publications Co., 2011.

[8] C. Baquero, P. S. Almeida, and A. Shoker. Pure operation-based replicated data types. *arXiv preprint arXiv:1710.04469*, 2017.

[9] Basho. Distributed data types - riak 2.0, 2014. Retrieved Apr-2018.

[10] P. Baudiš. Current concepts in version control systems. Bachelor Thesis, Univerzita Karlova, Matematicko-fyzikální fakulta. https://arxiv.org/abs/1405.3496, 2008.

[11] E. Bertino and G. Guerrini. Objects with multiple most specific classes. In *ECOOP'95-Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995*, pages 102–126. Springer, 1995.

[12] I. Beschastnikh, P. Wang, Y. Brun, and M. D. Ernst. Debugging distributed systems. *Commun. ACM*, 59(8):32–37, July 2016.

[13] A. R. Bharambe, J. Pang, and S. Seshan. Colyseus: A distributed architecture for online multiplayer games. In *NSDI*, volume 6, pages 12–12, 2006.

[14] A. Bieniusa and T. Fuhrmann. Consistency in hindsight: A fully decentralized stm algorithm. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–12, 2010.

[15] C. Biyikoglu. Under the hood: Redis crdts (conflict-free replicated data types). Retrieved Apr-2018.

[16] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings of the European Conference on Object-oriented Programming on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA/ECOOP '90, pages 303–311, New York, NY, USA, 1990. ACM.

[17] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the java programming language. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, pages 183–200, New York, NY, USA, 1998. ACM.

[18] K. Bruce. *Foundations of Object-Oriented Languages: Types and Semantics*. The MIT Press, 2002.

[19] S. Burckhardt, A. Baldassion, and D. Leijen. Concurrent programming with revisions and isolation types. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'10)*. ACM SIGPLAN, October 2010.

[20] S. Burckhardt and D. Leijen. Semantics of concurrent revisions. In G. Barthe, editor, *Programming Languages and Systems*, pages 116–135, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[21] S. Burckhardt, D. Leijen, J. Protzenko, and M. Fähndrich. Global Sequence Protocol: A Robust Abstraction for Replicated Shared State. In J. T. Boyland, editor, *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 568–590, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[22] C. Carlsson. DIVE A multi-user virtual reality system. *Virtual Reality Annual International*, pages 394–400, 1993.

[23] A. Cassandra. Apache cassandra. *Website. Available online at http://planetcassandra.org/what-is-apache-cassandra*, page 13, 2014.

[24] P. Cederqvist, R. Pesch, et al. Version management with cvs, 1992.

[25] C. Chambers. Predicate classes. In *ECOOP'93-Object-Oriented Programming*, pages 268–296. Springer, 1993.

[26] W.-C. Chang and P.-C. Wang. A write-operation-adaptable replication system for multiplayer cloud gaming. In *2017 IEEE Conference on Dependable and Secure Computing*, pages 334–339. IEEE, 2017.

[27] M. E. Conway. A multiprocessor system design. In *Proceedings of the November 12-14, 1963, Fall Joint Computer Conference*, AFIPS '63 (Fall), page 139–146, New York, NY, USA, 1963. Association for Computing Machinery.

[28] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[29] G. Cordasco, R. De Chiara, A. Mancuso, D. Mazzeo, V. Scarano, and C. Spagnuolo. A framework for distributing agent-based simulations. In M. Alexander, P. D'Ambra, A. Belloum, G. Bosilca, M. Cannataro, M. Danelutto, B. Di Martino, M. Gerndt, E. Jeannot, R. Namyst, J. Roman, S. L. Scott, J. L. Traff, G. Vallée, and J. Weidendorfer, editors, *Euro-Par 2011: Parallel Processing Workshops*, pages 460–470, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[30] N. Crooks, Y. Pu, N. Estrada, T. Gupta, L. Alvisi, and A. Clement. Tardis: A branch-and-merge approach to weak consistency. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1615–1628, New York, NY, USA, 2016. ACM.

[31] Z. Diao. Consistency models for cloud-based online games: the storage system's perspective. *Grundlagen von Datenbanken*, 2013.

[32] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. Fickle: Dynamic object re-classification. In *ECOOP 2001-Object-Oriented Programming*, pages 130–149. Springer, 2001.

[33] P. DuBois and M. Foreword By-Widenius. *MySQL*. New riders publishing, 1999.

[34] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, SIGMOD '89, pages 399–407, New York, NY, USA, 1989. ACM.

[35] E. Ernst, K. Ostermann, and W. R. Cook. A virtual class calculus. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 270–282, New York, NY, USA, 2006. ACM.

[36] J. Estublier, D. Leblang, G. Clemm, R. Conradi, W. Tichy, A. van der Hoek, and D. Wiborg-Weber. Impact of the research community on the field of software configuration management: Summary of an impact project report. *SIGSOFT Softw. Eng. Notes*, 27(5):31–39, Sept. 2002.

[37] Y. Falcone, H. Nazarpour, M. Jaber, M. Bozga, and S. Bensalem. Tracing distributed component-based systems, a brief overview. In C. Colombo and M. Leucker, editors, *Runtime Verification*, pages 417–425, Cham, 2018. Springer International Publishing.

[38] R. T. Fielding and R. N. Taylor. *Architectural styles and the design of network-based software architectures*, volume 7. University of California, Irvine Irvine, 2000.

[39] Q. Fu, J.-G. Lou, Y. Wang, and J. Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *2009 ninth IEEE international conference on data mining*, pages 149–158. IEEE, 2009.

[40] R. M. Fujimoto. *Parallel and Distributed Simulation*. Wiley, 2000.

[41] V. Gasiunas, M. Mezini, and K. Ostermann. Dependent classes. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 133–152, New York, NY, USA, 2007. ACM.

[42] C. GauthierDickey, D. Zappala, V. Lo, and J. Marr. Low latency and cheat-proof event ordering for peer-to-peer games. In *Proceedings of the 14th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, NOSSDAV '04, pages 134–139, New York, NY, USA, 2004. ACM.

[43] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica. Friday: Global comprehension for distributed replay. In *NSDI*, volume 7, pages 285–298, 2007.

[44] D. M. Geels, G. Altekar, S. Shenker, and I. Stoica. *Replay debugging for distributed applications*. PhD thesis, University of California, Berkeley, 2006.

[45] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[46] A. Gotsman and S. Burckhardt. Consistency Models with Global Operation Sequencing and their Composition. In A. W. Richa, editor, *31st International Symposium on Distributed Computing (DISC 2017)*, volume 91 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 23:1–23:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[47] C. Hewitt, P. Bishop, and R. Steiger. Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence. In *Advance Papers of the Conference*, volume 3, page 235. Stanford Research Institute, 1973.

[48] J. J. Hunt, K. P. Vo, and W. F. Tichy. An empirical study of delta algorithms. In I. Sommerville, editor, *Software Configuration Management*, pages 49–66, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[49] J. W. Hunt and M. McIlroy. An algorithm for differential file comparison. Technical Report 41, Bell Labs, June 1976.

[50] S. I. Inc and D. L. Weaver. *The SPARC architecture manual*. Prentice-Hall, 1994.

[51] X. Jiang, F. Safaei, and P. Boustead. Latency and scalability: a survey of issues and techniques for supporting networked games. In *2005 13th IEEE International Conference on Networks Jointly held with the 2005 IEEE 7th Malaysia International Conf on Communic*, volume 1, pages 6–pp. IEEE, 2005.

[52] G. Kaki, S. Priya, K. Sivaramakrishnan, and S. Jagannathan. Mergeable replicated data types. *Proc. ACM Program. Lang.*, 3(OOPSLA), Oct. 2019.

[53] G. Kaki, K. Sivaramakrishnan, and S. Jagannathan. Version Control Is for Your Data Too. In B. S. Lerner, R. Bodík, and S. Krishnamurthi, editors, *3rd Summit on Advances in Programming Languages (SNAPL 2019)*, volume 136 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:18, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[54] G. Kaki, K. Sivaramakrishnan, and S. Jagannathan. Version Control Is for Your Data Too. In B. S. Lerner, R. Bodík, and S. Krishnamurthi, editors, *3rd Summit on Advances in Programming Languages (SNAPL 2019)*, volume 136 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:18, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[55] B. B. Kang, R. Wilensky, and J. Kubiatowicz. The hash history approach for reconciling mutual inconsistency. In *Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on*, pages 670–677. IEEE, 2003.

[56] S. Khanna, K. Kunal, and B. C. Pierce. A formal investigation of diff3. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 485–496. Springer, 2007.

[57] B. Knutsson, H. Lu, W. Xu, and B. Hopkins. Peer-to-peer support for massively multiplayer games. In *IEEE INFOCOM 2004*, volume 1. IEEE, 2004.

[58] K. Kulkarni and J.-E. Michels. Temporal features in sql:2011. *SIGMOD Rec.*, 41(3):34–43, Oct. 2012.

[59] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, Sep. 1979.

[60] P.-Å. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *Proc. VLDB Endow.*, 5(4):298–309, Dec. 2011.

[61] S. W. K. Lee and R. K. C. Chang. Enhancing the experience of multiplayer shooter games via advanced lag compensation. In *Proceedings of the 9th ACM Multimedia Systems Conference*, MMSys '18, pages 284–293, New York, NY, USA, 2018. ACM.

[62] D. Leijen, S. Burckhardt, B. P. Wood, and M. Fahndrich. Cloud types for eventual consistency. In *Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP)*. Springer, June 2012.

[63] A. Levy. *Basic Set Theory*. Springer-Verlag, 1979.

[64] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D3s: Debugging deployed distributed systems. 2008.

[65] A. Löh, W. Swierstra, and D. Leijen. A principled approach to version control. *Preprint. Available at: http://www. andres-loeh. de/VersionControl. html.(last accessed date 25/08/2016)*, 2007.

[66] C. V. Lopes, R. Achar, and A. Valadares. Predicate collection classes. *Journal Of Object Technology*, 16(2):3–1, 2017.

[67] J. Mace, R. Roelke, and R. Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. *ACM Trans. Comput. Syst.*, 35(4):11:1–11:28, Dec. 2018.

[68] O. L. Madsen and B. Moller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '89, pages 397–406, New York, NY, USA, 1989. ACM.

[69] K. Manassiev, M. Mihailescu, and C. Amza. Exploiting distributed version concurrency in a transactional memory cluster. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, page 198–208, New York, NY, USA, 2006. Association for Computing Machinery.

[70] F. Mattern et al. *Virtual time and global states of distributed systems*. Citeseer, 1988.

[71] S. McDirmid and J. Edwards. Programming with managed time. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, pages 1–10, New York, NY, USA, 2014. ACM.

[72] D. Mengistu, P. Tröger, L. Lundberg, and P. Davidsson. Scalability in distributed multi-agent based simulations: The jade case. In *2008 Second International Conference on Future Generation Communication and Networking Symposia*, volume 5, pages 93–99. IEEE, 2008.

[73] T. Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, May 2002.

[74] B. Milewski. Distributed source control system. In R. Conradi, editor, *Software Configuration Management*, pages 98–107, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[75] R. Milner, L. Morris, and M. Newey. *A Logic for Computable Functions with Reflexive and Polymorphic Types*, pages 371–394. IRIA-Laboria, 1975.

[76] B. Momjian. *PostgreSQL: introduction and concepts*, volume 192. Addison-Wesley New York, 2001.

[77] D. A. Moon. Object-oriented programming with flavors. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPLSA '86, pages 1–8, New York, NY, USA, 1986. ACM.

[78] S. Muthukrishnan. *Data Streams: Algorithms and Applications*. Now Publishers, 2005.

[79] D. A. Nichols, P. Curtis, M. Dixon, and J. Lamping. High-latency, low-bandwidth windowing in the jupiter collaboration system. In *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology*, UIST '95, pages 111–120, New York, NY, USA, 1995. ACM.

[80] M. Odersky and M. Zenger. Scalable component abstractions. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 41–57, New York, NY, USA, 2005. ACM.

[81] G. Oster, P. Molli, P. Urso, and A. Imine. Tombstone transformation functions for ensuring consistency in collaborative editing systems. In *Proceedings of the International Workshop on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, 2006.

[82] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, SE-9(3):240–247, 1983.

[83] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, SE-9(3):240–247, May 1983.

[84] J. N. Plumb, S. K. Kasera, and R. Stutsman. Hybrid network clusters using common gameplay for massively multiplayer online games. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*, FDG '18, pages 2:1–2:10, New York, NY, USA, 2018. ACM.

[85] N. Preguiça, J. M. Marques, M. Shapiro, and M. Letia. A commutative replicated data type for cooperative editing. In *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, ICDCS '09, pages 395–403, Washington, DC, USA, 2009. IEEE Computer Society.

[86] A. Randolph, H. Boucheneb, A. Imine, and Q. Alejandro. On consistency of operational transformation approach. In *Proceedings of the 14th International Workshop on Verification of Infinite-State Systems*, pages 1–15, 2012.

[87] D. P. Reed. *Naming and synchronization in a decentralized computer system.* PhD thesis, Massachusetts Institute of Technology, 1978.

[88] D. P. Reed. Implementing atomic actions on decentralized data. *ACM Trans. Comput. Syst.*, 1(1):3–23, Feb. 1983.

[89] M. Ressel, D. Nitsche-Ruhland, and R. Gunzenhäuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *Proceedings of the 1996 ACM Conference on Computer Supported Cooperative Work*, CSCW '96, pages 288–297, New York, NY, USA, 1996. ACM.

[90] M. J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, 1(4):362 – 370, 1975.

[91] D. Roundy. Darcs: distributed version management in haskell. In *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 1–4. ACM, 2005.

[92] A. Rousset, B. Herrmann, C. Lang, and L. Philippe. A communication schema for parallel and distributed multi-agent systems based on mpi. In S. Hunold, A. Costan, D. Giménez, A. Iosup, L. Ricci, M. E. Gómez Requena, V. Scarano, A. L. Varbanescu, S. L. Scott, S. Lankes, J. Weidendorfer, and M. Alexander, editors, *Euro-Par 2015: Parallel Processing Workshops*, pages 442–453, Cham, 2015. Springer International Publishing.

[93] A. Rousset, B. Herrmann, C. Lang, and L. Philippe. A survey on parallel and distributed multi-agent systems for high performance computing simulations. *Computer Science Review*, 22:27–46, 2016.

[94] N. B. Ruparelia. The history of version control. *SIGSOFT Softw. Eng. Notes*, 35(1):5–9, Jan. 2010.

[95] S. K. Sarin and N. A. Lynch. Discarding obsolete information in a replicated database system. *IEEE Transactions on Software Engineering*, SE-13(1):39–47, 1987.

[96] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. *ECOOP 2003 – Object-Oriented Programming: 17th European Conference, Darmstadt, Germany, July 21-25, 2003. Proceedings*, chapter Traits: Composable Units of Behaviour, pages 248–274. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[97] R. D. Schiffenbauer. Interactive debugging in a distributed computational. 1981.

[98] W. Schütz. Fundamental issues in testing distributed real-time systems. *Real-Time Systems*, 7(2):129–157, Sep 1994.

[99] M. Shapiro, A. Bieniusa, N. Preguiça, V. Balegas, and C. Meiklejohn. Just-right consistency: reconciling availability and safety. *arXiv preprint arXiv:1801.06340*, 2018.

[100] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In X. Défago, F. Petit, and V. Villain, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 386–400, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[101] Y. Smaragdakis and D. S. Batory. Implementing layered designs with mixin layers. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, ECCOP '98, pages 550–570, London, UK, UK, 1998. Springer-Verlag.

[102] J. Smed, T. Kaukoranta, and H. Hakonen. Aspects of networking in multiplayer computer games. *The Electronic Library*, 20(2):87–97, 2002.

[103] A. Ulrich and H. König. *Architectures for Testing Distributed Systems*, pages 93–108. Springer US, Boston, MA, 1999.

[104] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '87, pages 227–242, New York, NY, USA, 1987. ACM.

[105] A. Valadares, E. Gabrielova, and C. V. Lopes. On designing and testing distributed virtual environments. *Concurrency and Computation: Practice and Experience*, 28(12):3291–3312, 2016.

[106] P. Verissimo and L. Rodrigues. *Distributed systems for system architects*, volume 1. Springer Science & Business Media, 2012.

[107] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, Jan. 2009.

[108] H. Vogt. *Leçons sur la résolution algébrique des équations*. Nony, 1895.

[109] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 357–368, New York, NY, USA, 2015. ACM.

[110] L. Wingerd. *Practical perforce*. " O'Reilly Media, Inc.", 2005.

[111] S. R. Wiseman. *Garbage collection in distributed systems*. PhD thesis, Newcastle University, 1988.

[112] D. Woos, Z. Tatlock, M. D. Ernst, and T. E. Anderson. A graphical interactive debugger for distributed systems. *CoRR*, abs/1806.05300, 2018.

[113] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proc. VLDB Endow.*, 10(7):781–792, Mar. 2017.

[114] G. T. Wuu and A. J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, PODC '84, pages 233–242, New York, NY, USA, 1984. ACM.

[115] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. Modist: Transparent model checking of unmodified distributed systems. 2009.

[116] S. Zander, I. Leeder, and G. Armitage. Achieving fairness in multiplayer network games through automated latency balancing. In *Proceedings of the 2005 ACM SIGCHI International Conference on Advances in Computer Entertainment Technology*, ACE '05, pages 117–124, New York, NY, USA, 2005. ACM.