Title
Spacetime C++ Core: Higher Performance Implementation of Global Object Tracker Programming Model

Author
Yu, Xiaochen

Publication Date
2021

UNIVERSITY OF CALIFORNIA,
IRVINE

Spacetime C++ Core: Higher Performance Implementation of Global Object Tracker Programming Model

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Software Engineering

by

Xiaochen Yu

Thesis Committee:
Professor Cristina V. Lopes, Chair
Associate Professor James A. Jones
Professor Sam Malek
Professor Michael Dillencourt

2021

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to thank Professor Cristina Lopes, my advisor, for her academic support, encouragement and general guidance, without which the work presented in this thesis wouldn't be possible. I would like to thank Rohan Achar, for introducing me to this project and providing valuable guidance. I would also like to thank Associate Professor James Jones, Professor Sam Malek, and Professor Michael Dillencourt, for being on this committee and always being a great pleasure to learn from and work with.

Further, I would like to express my most sincere thanks to my family, especially my mother, for the unconditional care and support I consistently receive, which truly made me who I am and made it possible for me to be where I am. I would also like to thank some of my close friends, including He, Junjie, for the great memories and inspirations they brought me, and for putting up with me in general for a long time.

Finally, I would like to thank everyone I have met and have been around for the past several years as I study at UC Irvine, for their kindness and friendship that I will always cherish: Ali, Dave, Fabian, Janet, Karen, Lewis, Muzyar, Powei, Taylor, Thomas, Tian, Yang, Yunlong, Zachary, Zichong, Zizhao. Having met and known these amazing people made my experience during this years long journey truly magical.

ABSTRACT OF THE THESIS

Spacetime C++ Core: Higher Performance Implementation of Global Object Tracker Programming Model

By

Xiaochen Yu

Master of Science in Software Engineering

University of California, Irvine, 2021

Professor Cristina V. Lopes, Chair

Distributed systems are the backbone of a wide range of highly important applications, but developing and debugging such systems become tricky with the synchronization mechanism needing to meet critical consistency and latency requirements. A range of models, frameworks and languages have been developed to tackle such complexity in allowing a shared computing state. In this thesis, based on the Global Object Tracker (GoT) distributed programming model proposed in previous research, I present a re-implementation of the core components of Spacetime, a GoT based Python framework. The C++ implementation tackles multiple limitations the pure Python implementation was faced with, and further improves the performance with redesigned core structures. The internal designs and comparison with previous designs will be discussed throughout this thesis. The C++ core also allows interoperability with other high level languages, such as javascript. Through benchmarks comparing both implementations and the Parse platform with respect to update latency, we are able to show significant performance improvement over the pure Python implementation when the job scales up, and have a competitive performance against Parse in many configurations.

Chapter 1

Introduction

Many important applications and systems are by nature distributed, and require mechanisms to properly synchronize data across multiple machines over distant physical locations. Such applications include online multiplayer video games, banking systems, self-driving cars, etc. Many of such use cases require the distributed system to synchronize a shared state of computation over the network. However, said mechanism is tricky to properly design and implement, due to the non-deterministic nature of the distributed operations. Further, with conflicts inevitably occurring, the synchronization mechanism need to satisfy a certain set of consistency constraints, adding to the complexity of designing such distributed systems. Many architectural styles, design patterns, programming models, languages, and frameworks have been proposed and applied to practical systems to assist the construction of complex distributed systems.

The Global Object Tracker (GoT) distributed programming model was previously proposed [3] primarily concerning distributed applications where components sharing a long-lived, highly mutable, and often highly contended state. GoT is inspired by popular version control software Git, as GoT tracks versions of the set of objects being synchronized, and re-

solve conflicts of states with merging operations. The GoT model uses a common consistency model, causal consistency, and optimizes for a low update latency.

Based on a purely Python implemented framework realizing the GoT model, Spacetime, this thesis presents a redesigned and re-implemented core library of Spacetime including several core components, written in C++. We will discuss the limitations the previous implementation was facing, and the internal design decisions taken in the redesigned library to circumvent such limitations, further improving the performance of the framework. Through analyzing the result of benchmarks, we observe the significant performance improvement over the previous implementation, and confirm the effectiveness of the redesign.

This thesis is organized as follows. Chapter 2 introduces a number of related work previously done in the state replication area. Chapter 3 gives an overview of the GoT model. Chapter 4 discusses the core data structure handling the version control of objects, where Chapter 5 introduces several other core components in the implementation. Chapter 6 presents the evaluation on the performance of the new implementation, and finally Chapter 7 concludes this thesis and envision possible future works on this topic.

Chapter 2

Related Work

There have been various programming models, architecture styles, and frameworks developed by both the academia world and by the industry aiming to address the hardness of building distributed systems with a shared computation state. In this chapter, we briefly review some of the existing programming models tackling the same complexity arose when designing distributed systems when the shared state becomes highly mutable. We will discuss the general categories such models belong to, and the design goals that lead to the approaches each model have taken, along with strengths and shortcomings of the models.

2.1 Shared-State Programming Models

In shared-state programming models, a set of data, or state, is shared by distributed nodes across the network, where each node may read from and write to the shared state. Traditional relational database management systems (RDBMS), such as MySQL [12], Oracle [14], and PostgreSQL [22], provide sequential consistency utilizing mechanisms including transactions. But a strong consistency guarantee comes with a trade-off of either a lower availability, or

a longer time before updates propagate to other nodes in the network. Redis [6] takes the approach where each node in a cluster of nodes have a local copy of the data set, and the availability of the system can be increased simply by adding more nodes to the cluster. However, as synchronization of the shared state among nodes in the cluster is not aggressively performed, update latency, hence the time it takes for updated date to be received by all nodes, can be high. Other NoSQL database systems, such as MongoDB [5], could perform better than SQL databse systems in terms of read/write latency, but only provides a weaker consistency guarantee: eventual consistency.

2.2 Message Passing Programming Models

In contrast to shared-state models, in message passing programming models, there does not exist an entire set of objects shared by and synchronized across all nodes of the system. Instead, nodes send messages to each other, typically domain-specific commands and updates, to coordinate the overall computation process.

One very commonly used paradigm that fit into this category would be publish-subscribe, in which events are broadcasted to subscribers by the nodes that created such events, namely the publishers in the system. With the events propagated to subscribers immediately, publish-subscribe model provides excellent update latency, but without a fully synchronized state, ensuring the correctness and consistency of the data across the system is very difficult, and require significant amount of engineering effort. For example, it would be difficult to implement multiplayer online games with publish-subscribe, as complex interactions of players would be difficult to be properly computed without a local copy of states of all the interacting objects.

Another category of programming models that can be considered to be a subset of message passing models, is the actor models. Actor models were first introduced in 1973 [15], where nodes in the system are considered actors, and would execute a set of defined tasks in parallel, communicating over events of updates and commands to other actors in the system. A recent example designed based on actor model is Anna [29], which achieves coordination-free causal consistency with the introduction of lattices.

2.3 Object Based Version Control Models

The concept behind many version control systems, including Git, is that the history of the set of data being tracked, is represented by a directed acyclic graph (DAG), where each vertex represent a version of the data, or a snapshot of the state, and edges between vertices represent the changes between the new version the edge goes to, and the old version where the edge is from. The changes between versions are commonly referred to as the "delta" [27]. When applied to object owned data, instead of text based files, conflicts of object state can also be resolved through merges, similar to that in file based version control systems. Also similar to file based version control systems, replicas of the object set is synchronized by sending and retrieving updates on the DAG maintaining the version history of the set.

Examples of object based version control models include Concurrent Revisions, which was introduced in 2010 [8] for parallel processing purposes, and follow the fork-join design pattern [9]; Cloud types [20], focusing on applications that require isolation, such as mobile games; CARMOT [18], a state replication framework in OCaml that utilizes data types that are less prone to inconsistency during merging [17]; and TARDiS [11], a distributed key-value database based on centralized version control.

Our GoT programming model also falls under this category of programming models. We will take a breif look at the GoT model in the next chapter.

Chapter 3

The Global Object Tracker Programming Model

In this chaptor, we take a look at the underlying programming model implemented by Spacetime framework, the Global Object Tracker programming model. We further discuss the general architecture of Spacetime, and the motivation for a redesign and implementation of the framework.

3.1 Introduction

In GoT, an application consists multiple nodes, performing synchronous tasks on a shared collection of objects, synchronized by underlying mechanisms. Nodes in the same distributed application can be located in separate physical machines, where the set of objects get the updated to them synchronized over the network.

Table 3.1: 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.
N/A	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.

3.1.1 Dataframe

The object repository (similar to a Git repository) containing data shared among nodes in the same distributed application is named dataframe in GoT. Each node in a GoT application holds its own instance of a dataframe, where dataframes of nodes would synchronize and reflect updates to each node. A dataframe consists of two main components: a snapshot that contains data of all objects within the current version, with which the application code interacts, and a version history being synchronized with other nodes in the application. This structure is also similar to that of Git, where a snapshot consists all files currently being tracked and operated upon, and a version history of all previous changes made to the snapshot synchronized by user operations.

Table 3.1 shows a comparison between APIs of dataframe and Git.

Snapshot

The snapshot contains the latest known state of all objects being tracked, where all modifications to object state made by the application level logic are automatically staged in the snapshot. The latest state staged in the snapshot is only visible to its local node until a *commit* operation put the latest updates into the object version graph, thus making it visible to other nodes in the system through synchronization on the version graph. As a typical object heap, the snapshot in GoT supports the creation, deletion, reading and updating of objects in the heap. The application code can also synchronize the snapshot with the latest state known by the local version graph by using the *checkout* operation.

Object Version Graph

The object version graph in each dataframe keep a record of historical versions of the set of objects. Version graph, by nature, is a directed acyclic graph (DAG), as any modification to the state would create new versions, and would never go back in time to an older version, as in Git. Each vertex in the version graph represents a version of state of all objects at some point in history, and each edge in the version graph is associated with the difference, known as diff in context of version control, between the version the edge points to and the version the edge goes out of.

One important property of the object version graph in GoT, is that the full states of the set of objects is not stored at each vertex: the version graph starts with a version named ROOT, which corresponds to an empty state, and all other changes, and only the changes, are stored as diffs associated with the edges of the graph. The latest version in the version graph is considered the HEAD vertex, once again using the same nomenclature as Git. Once newer changes are committed from the snapshot or received from remote, the newly created version becomes the HEAD vertex, replacing the previous one.

Interaction between Snapshot and Version Graph

The interaction between the two main components of a dataframe, namely the snapshot and the version graph, is done through two primitive operations, commit and checkout.

Upon commit, all changes staged in the snapshot will be stored into the version graph: a globally unique identifier is created for the new version, and a new edge from the previously tracked version to the new version, is added to the version graph, associated with the changes the snapshot has been tracking. After changes are committed to the version graph, the snapshot then starts tracking new changes made upon the newly added version, which became the new HEAD version in the version graph.

When the version graph contains a newer version than the snapshot received from remote nodes, checkout operation will retrieve the diff data required to get the snapshot up to date with the latest HEAD version, and apply the changes to the snapshot which updates the state of the objects accordingly. With the snapshot interacting with the version graph to be updated and providing objects to application state, the snapshot need to meet the requirement of read stability.

Conflict Resolution

When the application code want to synchronize with nodes across the network, dataframe provides two primitive operations: *pull* and *push*. As with their counterparts with the same name in Git, the pull operation retrieves the latest version from a remote machine, where push operation updates the remote machine with the latest locally committed changes.

Conflicts are detected when a split, or branch, is seen in the object version graph, either by pulling from remote with changes made to an older version than the latest local committed version, or when a remote repository receives a push update from another node with changes

made to a earlier version known by that repository. In short, conflicts arise when nodes synchronize after concurrent write operations were performed. With the version graph, conflicts are quite easy to detect: whenever a vertex has more than one outgoing edges, there is a conflict, and subsequently, will be resolved by merging.

One difference GoT has from Git is that in Git, when conflict arise, hence when two a local repository have committed changes applied to an older version than the HEAD of the remote repository, the push of such changes will be rejected, and the local repository need to pull the latest changes, resolve conflicts locally with merge, rebase, or put local changes into a new branch which effectively defers the resolution of the conflict, before being able to push to the remote repository and have the update reflected. In GoT, as a low update latency is desired, creating new branches would leave states at different nodes further apart, leading to more required work during merge steps, and even the possibility of merging not being able to catch up with the updates, as rapid distributed updates are being performed by multiple nodes. Thus in GoT, conflict resolution in form of merge is performed by the receiving end of a push operation, in addition to local repositories pulling from remote. This allows merges to be completed as early as possible, and avoiding starvation of pushing nodes.

Similarly, with updates being performed rapidly by all nodes in the distributed application, when merging conflicting states together, having to wait for human input to decide the solution to a three way merge becomes unfeasible. To address this issue, a custom merge function on conflicting object states provided by the user can be used as a mechanism to automatically resolve conflicts during merge operations. However, the model also handles most of merging of a state without requiring a merge function: GoT utilizes a variant of operational transformation [13]. In contrast to original OT where functions transforming conflicting states into a merged state would be required for every data type, in GoT, as delta changes are being stored instead of the complete state, the approach where combining delta

changes to a merged delta change that would get the original state to the latest state can be applied to different data types without changing the operational transformation logic.

Example: Distributed Counter

To demonstrate how GoT handles concurrent updates and resolve the conflicts, we consider this simple example: a set of nodes share a simple integer counter object, where each node can make additions to the shared counter (and as a matter of fact, subtractions too, being considered addition of a negative value). Each node (including the remote node itself) performs an arbitrary number of additions to the counter with an arbitrary value, and each node attempt to synchronize with the remote repository as frequently as possible.

Suppose the shared counter is named x, where there are three nodes, N_1, N_2 and N_3, where both N_2 and N_3 use N_1 as the remote repository. Initially, the shared counter object is created by N_1, and both N_2 and N_3 have pulled from N_1 to operate on the shared object. All nodes have made a concurrent update (addition) to the counter, and Figure 3.1 shows how the synchronization among nodes impact the version graph in all nodes.

At time $T = 1$, each node has a latest version unknown by other nodes, and by pushing of N_2 at $T = 2$ and N_3 at $T = 3$, N_1 receives updates from the other nodes, and creates extra edges from the new version from the previous latest version of the local and remote repository to a new merged version, each accompanied with delta updates such that when applied would update the branching versions to the merged version. We can see an example of this at $T = 3$: to get from version C to version E, the same update that gets the root version to D need to be applied to version C, thus an edge from C to E is added with combined delta from root to A and A to D. Similarly, the update to get from root to C is applied to D to get to the merged version E. This process of continuously merging branching versions into

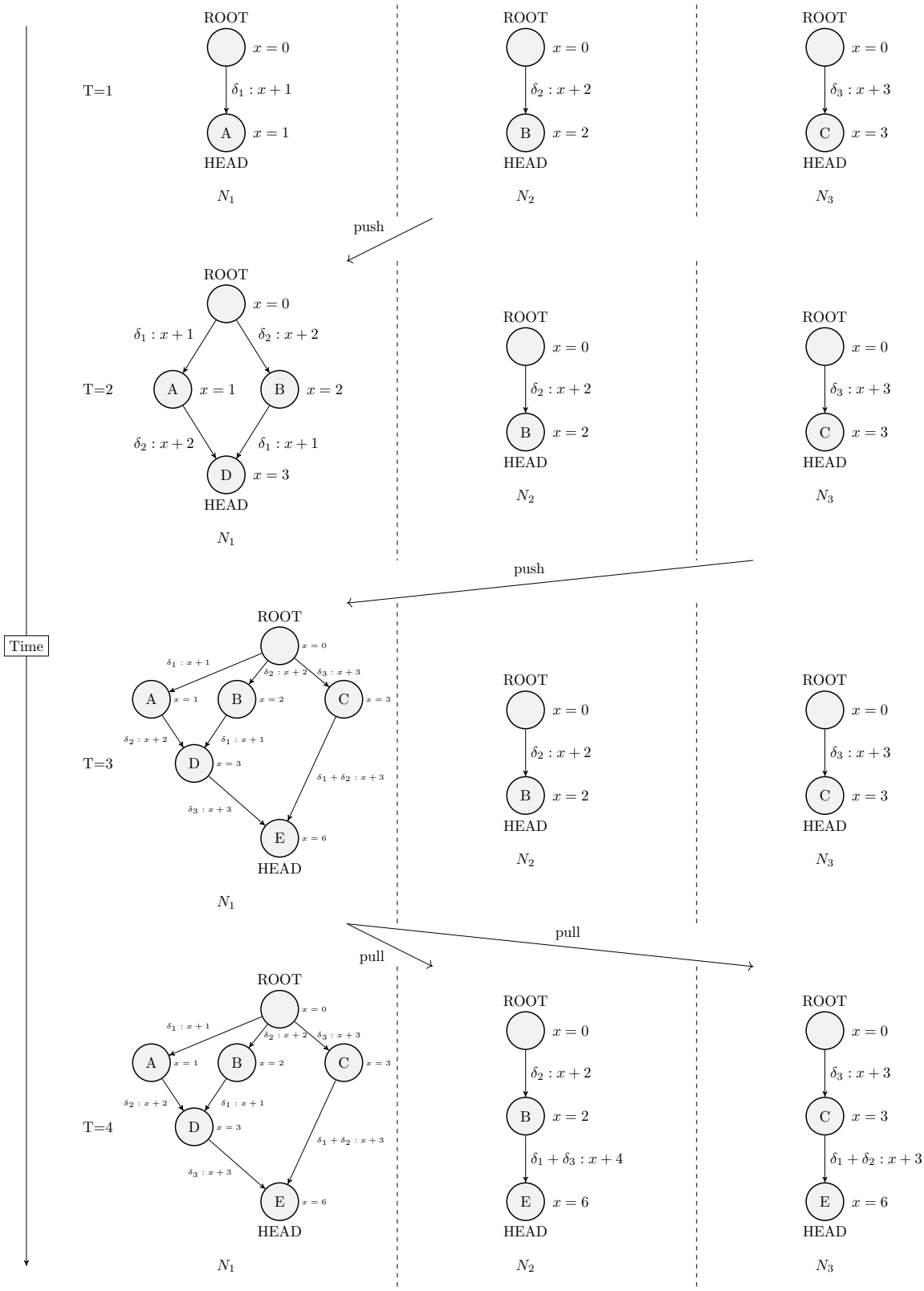


Figure 3.1: Communication in Distributed Counter Example.

one branch ensures when nodes pull for updates at $T = 4$, both are provided delta changes that get them to the converging merged version.

It should be noted the semantics in this example is for demonstration only: in actual applications, conflicts on one single field would need to invoke the custom merge function to resolve, as the semantics would differ from application to application. A fully automated merging on conflicting updates on different objects, or different dimensions, hence member fields, can be performed without the need of the invocation of the custom merge function, in a similar fashion to that shown in this example.

Unbound State Growth and Garbage Collection

As can be seen previously in Figure 3.1, with each node simultaneously performing updates, each push/pull operation performed by a node may create new state history, hence vertices in the version graph. For file based version control systems such as git, the version history grows only at a speed at which human operates, as the artifacts being tracked are generally human created source code. In GoT, however, updates and commits are made by distributed programs running sets of tasks, which will potentially lead to tens or hundreds of versions being created per second, especially when the number of nodes increase. As time goes by, the size of the version graph grows rapidly and may consume all available memory of the system. This is a significant issue for frameworks and models that take the version control approach need to address in the design. In GoT, a garbage collection mechanism on unneeded versions is used to limit the growth of the size of the version graph while still retaining the functionality and satisfying the consistency model.

In order to design a correct and efficient garbage collection mechanism, we first need to define a set of rules that distinguish needed and unneeded vertices, hence versions, in the version graph. To be able to eliminate any unneeded versions, the repositories must keep track of

which versions are being tracked by which of the other nodes: without this knowledge, it would be impossible to prove any version can be eliminated, as it would be possible some other nodes are tracking said version and will further be needed locally to apply diffs when updates arrive from a remote node.

With the information of the relationship between versions and remote nodes that are keeping track of them, one basic rule we can easily observe is that if a version is known by all nodes, it can be eliminated, as nodes will never query or operate on older versions. When a version is removed, a new edge connecting the previous and next version will be created with the combined diff from the two previous edges, shortening the path length by one. With only this rule, we still might see unbounded growth of state: if a remote node is lagging behind in the version it is tracking, all versions after that tracked version is not known by that node. It is important that we also observe that on a path without branches, as updates always synchronize remote nodes to the latest version, any version between the current version being tracked and the latest version will not be tracked by that remote node even when a synchronization occurs: the remote node would then simply track the latest version instead. As such, this rule can be expanded to when a version is on a non-branching path and not being tracked by any remote node, it can be eliminated.

In a more complex case, where there are branches and merges in the graph, we can observe that eliminating a branching or merging vertex is intuitively tricky: when eliminating a version in the graph, we combine the delta updates that proceed and succeed said version, and as merging and/or branching versions have more than one preceding and succeeding edges representing conflicting changes, an attempt to remove such versions would not be able easily find a correct combining scheme. Thus the second rule of our garbage collection mechanism is the branching/merging versions cannot be eliminated. However, these versions will not persist forever: each path between a pair of branching/merging vertex will eventually be shortened as nodes tracking the intermediate versions synchronize. With all paths between

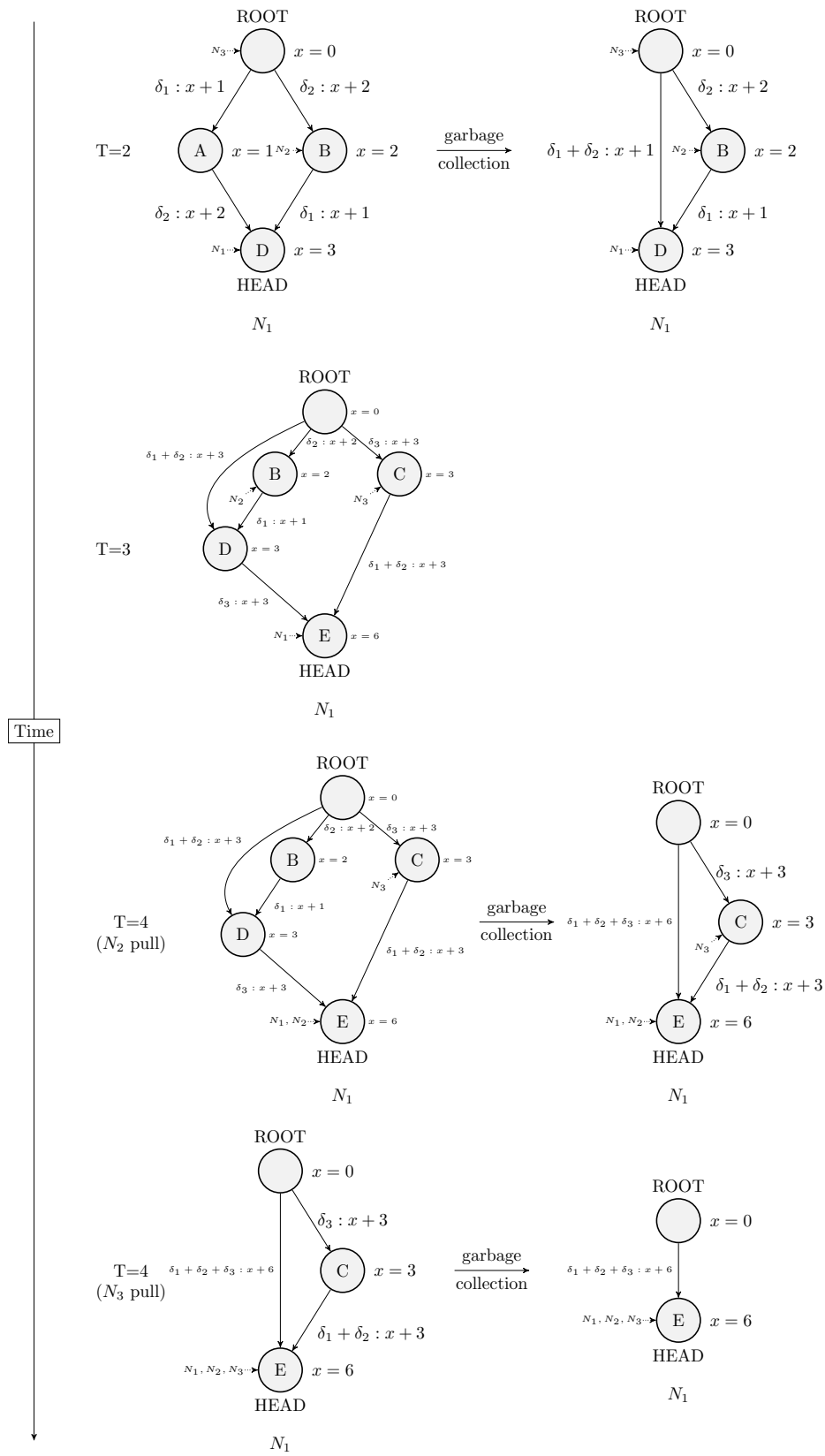


Figure 3.2: Garbage Collection in Distributed Counter Example.

a pair of branching and merging vertices combined to be the same delta change, we only need to keep one, instead of all paths, thus eventually taking away the branches and leave us with a non-branching path, on which we can eliminate all intermediate versions that are not being tracked by any remote node. By default, the path that appeared in the version graph first gets kept, while other paths get eliminated along with the vertices on them.

Let us consider the same counter example as in Figure 3.1 being garbage collected, shown in Figure 3.2. All interactions are the same as in Figure 3.1, and since in this specific example all merged versions are created in N_1, we focus on version graph in that node to see how the garbage collection process limits the version graph size growth. Markers representing the relationship between the remote nodes and the versions they track are added to the figure.

At $T = 2$, version A is not tracked by any node, as N_1 have moved on to track the latest merged version D, and N_2 only tracks version B. Version A gets eliminated, combining changes on edge ROOT $\rightarrow A$ and $A \rightarrow D$ into a single edge.

At $T = 3$, no garbage collection can be done as each vertex is either tracked by a node or a split/merge vertex. If path ROOT $\rightarrow B \rightarrow D$ is kept instead of ROOT $\rightarrow D$, version D can be eliminated, but as we keep the path that exists in the graph for the longest time (hence the edge ROOT $\rightarrow D$), D remains a merge vertex.

At $T = 4$, after N_2 pulled hence now tracks the latest E vertex, vertex B is eliminated with the path, leaving D a non-merge vertex, further causing it to be eliminated, combining the changes, resulting in the changes on edge ROOT $\rightarrow E$. After N_3 pulled, all three nodes now track version E, thus allowing us to eliminate version C. With path from ROOT to E already existing, the changes preceding and succeeding vertex C is also eliminated.

In node N_2 and N_3, the only changes on version graph after the garbage collection is in both nodes, the intermediate versions (B and C) are eliminated after the pull operation. In short, with garbage collection, after the interactions shown in this example, all three nodes end up

with version graphs containing only the ROOT and the latest version, and a path length of one, demonstrating the effectiveness of the garbage collection mechanism limiting the state growth.

3.2 The Spacetime framework

Spacetime is a prototype framework realizing the GoT programming model implemented in pure Python. The overall architecture follows quite closely to that of the GoT model, where a dataframe can be created and owned by each node of the application, and each dataframe contains a object heap, and a object version graph. Specifically, the version heap utilizes a subset of operations provided by rtypes, a library realizing the concept of Predicate Collection Classes (PCCs) [21], to handle reflection in object attribute accessing. On the other side, the version graph communicates with version graphs to synchronize latest object versions through a set of socket based client/server components.

In the implementation, an additional constraint is placed on the garbage collection component of the system, where eliminations of versions only occur during writing operations onto the object version graph. This makes reading operations truly read only in terms of versions and diffs (hence the vertices and edges in the version graph), while only possibly changing the markers on remote node version tracking. This allows the reading operations to never need to acquire the graph level writing lock and allows for more concurrent reading operations to be handled.

3.3 Motivation for Re-implementation of Spacetime

Several major drawbacks provide a strong motivation for creating a re-implemented version of Spacetime.

The most significant limitation imposed on the pure Python version of Spacetime is that, the multi-threaded performance of Python is limited by the infamous Global Interpreter Lock (GIL) whenever the job being executed is CPU-bound. While conventional thinking suggests that a network based framework like Spacetime would be network-bound, in reality with the use of reader-writer locks, we see many CPU-bound tasks when a large number of clients attempt to concurrently read the same version graph at a node and the GIL proves to be a significant bottleneck. One that is unnecessarily imposed by the language since reading from the version graph is thread safe, guaranteed by the version graph design.

Further, by being a pure Python library, induction into other programming languages and their runtime environments is limited, making it difficult for applications written in other languages to utilize the framework. In contrast, libraries written in C/C++ are often used as native extensions by runtime environments of other high level programming languages, as with the reference implementation of Python.

In next chapters, we explain in more details the design taken in the new version of Spacetime, and comparing with the original Spacetime.

Chapter 4

Version Graph Core Data Structure

The version graph data structure contains a significant part of the core logic of dataframe, and the efficiency of operations performed on the version graph greatly impacts how many concurrent requests a node can handle promptly, and how low of a latency we can get out of the system. In this chapter, we take a look at the design of the new implementation of version graph, and compare with the previous design to demonstrate the changes made to optimize the performance.

4.1 Design of Version Graph in New Implementation

The overall API of the version graph structure stays quite similar to the previous implementation: adding a new edge to the graph with a specified delta update, either creating vertices or connecting the edge to existing ones; iterating through the version graph from a specified version identified by a universally unique identifier (UUID) tag; and updating markers so the version graph can have its vertices garbage collected.

4.1.1 Structure of Version Graph

The general structure of the version graph consists of vertices and directed edges connecting vertices, forming a directed acyclic graph. The version graph is optimized for safe concurrent read an write operations, where there will only be up to one concurrent writing operation at any given time, but unlimited concurrent reading operations. Reading and writing operations can be performed concurrently as long as no writing operation is performed on nodes and edges that are being read from.

For garbage collection purposes, smart pointers are heavily used in the version graph data structure. Specifically, `std::shared_ptr` and accompanying `std::weak_ptr` are used [16]. The important property of `std::shared_ptr` is that the reference to the same object is counted, and when the last shared pointer instance pointing to some object is deleted, the object being pointed to is also deleted. This use of smart pointers will be discussed in more details in later sections when we implement garbage collection.

Node

Each node in the version graph always contain a finite amount of data, no matter how complex the version it represents is: this is due to GoT only storing delta data, which is stored on the edge, instead of the nodes. As such, a node primarily contains a tag, usually a UUID in a string format, except for the ROOT node. Each node points to two edges, the previous edge and the next edge, except for ROOT and HEAD versions. It it worth noting the node only points to one previous edge and one next edge, while split and merge nodes exist in the version graph: we would like to keep track of the first edge added to either side of the node, and consider that to be the mainline path related to the node. This is helpful when paths between split/merge nodes are being deleted to select which path to keep, and when traversing the version graph. Additionally, the number of preceding and succeeding

nodes are stored in each node, to keep track of all split/merge nodes for garbage collection purposes.

Edge

Each edge in the version graph contains an associated delta update that goes between two versions the edge connects, and in the implementation, this is stored as a json object in the edge, the payload. The edge is designed to not be updated during its lifetime. Once an edge is created, it points to two vertices (the Node objects) without the need to be updated. The payload on the edge will not be changed either, as the version history will never be modified, except for when unneeded versions are garbage collected, and at which point the edges connecting to the version being eliminated will be merged together, hence also deleted.

When nodes in the graph points to the edges, `std::shared_ptr` is used, as the lifetime of the edge can be perfectly captured by these pointers: an edge will only be pointed to by up to two vertices, and the number of reference only decrease to zero when the garbage collection process eliminate either vertex pointing to the edge. On the other hand, when edges point to nodes, we cannot use `std::shared_ptr` as we will have a circular reference between the edge and the node, rendering garbage collection impossible. Instead, `std::weak_ptr` is used, which does not guarantee the object it points to still exists, while also not increasing the reference count on that object. When Access of the object is necessary, a shared pointer can be created by the weak pointer when the object still exists, or a null pointer would be returned and we know the object no longer exists. In the version graph data structure, the limitation where only the writing operations performs garbage collection guarantees the vertices pointed to by the edge always exists during the lifetime of the edge.

Let us consider the example version graph shown in Figure 4.1, ignoring garbage collection for the moment. In this example, we have four vertices and four edges connecting the

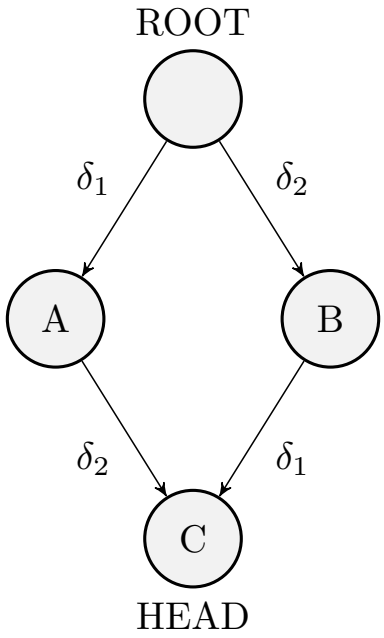


Figure 4.1: Example Abstract Version Graph.

four versions. Figure 4.2 shows how we represent this abstract version graph in the new implementation, where nodes point to edges with shared pointers, and edges point to nodes with weak pointers. Additionally, the ROOT and HEAD nodes are pointed by standalone shared pointers owned by the version graph object, which both provides entry points to the graph, point of comparison when traversing the graph, and guarantees these vertices will not be deleted.

It should be noted that while the graph is generally considered a DAG, the actual links in the graph are bidirectional, similar to that of the doubly linked lists. This is to allow both forward and reverse iterators to exist for the version graph. This functionality is essential for the rest of the dataframe to be implemented, as all reading operations on the version graph revolve around traversing the mainline path from a given version, either forward or backward, exactly what the iterators provide, which we will discuss in the next part.

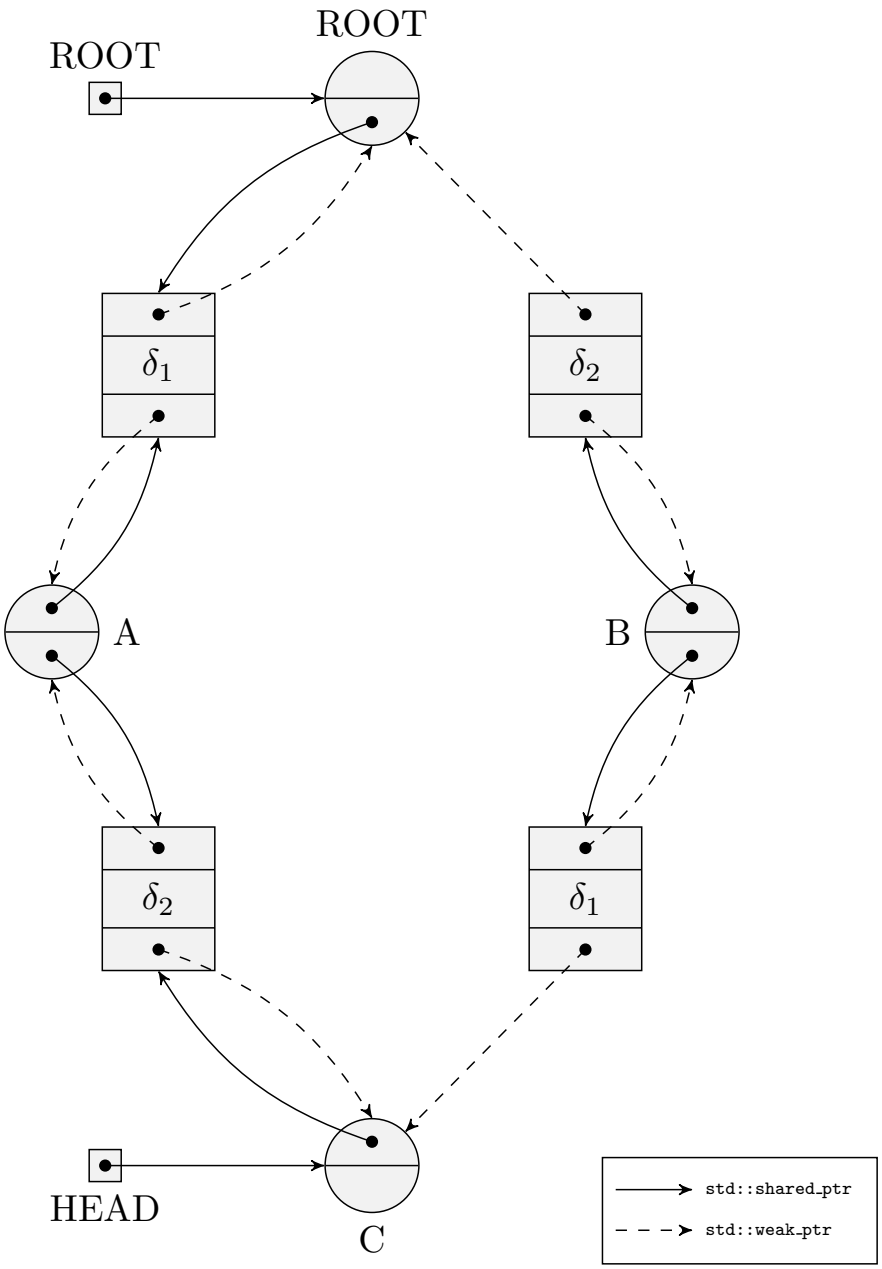


Figure 4.2: Example Version Graph Representation in New Implementation.

Iterators

Both forward and reverse iterators are implemented in the version graph. As required by the API of the version graph, iterators start at a specified version, and ends at either end of the version graph, either the ROOT node, or the HEAD node. Iterators return a pair, where the first element is the tag of next coming version, and the second element is the delta change on the edge between the current version and the next version. For example, if a forward iterator is initialized with version A, dereferencing that iterator will return tag C, and δ_2 on the edge $A \rightarrow C$. Similarly, if a reverse iterator is initialized with version A, dereferencing returns the tag ROOT and δ_1 as the change on the edge ROOT $\rightarrow A$.

With the API only allowing traversing the version graph through iterators, each of the concurrent reading operations would hold own an iterator and traverse the graph using it. As such, it is important for the iterators to properly lock the resources required to perform the traversal steps. We will discuss how we ensure the correctness of the version graph under concurrent read and write operations.

4.1.2 Concurrency Control

With the version graph allowing one writing operation simultaneously with unlimited reading operations being performed, it is important to make sure the writing operation is not in conflict with any reading operations. We first observe what resources each operation requires to stay unchanged during the operation, and describe the concurrency control scheme used to ensure the operations interact without race conditions.

Writing operations include either adding new vertices and edges, or removing vertices and edges during garbage collection and possibly reconnecting vertices to a new merged edge. In the first case, pointers to nodes are retrieved from the graph if they already exist, or

created if they are not yet created. Then, a new edge is connected to nodes on both of its ends, possibly setting the next pointer of the previous node and the previous pointer of the next node to the new edge. In the second case, edges connected to the node being garbage collected is combined into a new edge, and connected to the preceding and succeeding nodes, if the garbage collected version is considered mainline by the adjacent nodes. From both types of writing operations, we can observe the fact that no writing operation would change both the previous and next pointer of the same node.

For reading operations, what an iterator really need during a reading step is to have the edge to be read from, and pointers pointing to that edge not to be changed before the increment operation on the iterator. As such, the next pointer in the node where the current edge come from and the previous pointer in the node the current edge goes to need to stay constant. We can observe that neither node would need both of the pointers stay constant during one step of the iteration, which when combined with the previous observations on writing operations, a locking scheme in the version graph seems to be a very logical choice, where each half of each node is the finest granularity of locking in the graph. Each iterator would hold a reading lock on the half of the current node, and in each write operation, the halves of nodes that need to be modified will be locked by writing locks. This locking scheme makes it possible to have as many concurrent operations as possible under the constraints of the overall system, while ensuring no race conditions or deadlocks arise during execution.

4.1.3 Garbage Collection

By using shared pointers, the garbage collection mechanism can be built without manually implementing a reference counting utility, but rather simply by correctly controlling the creation and deletion of shared pointers to nodes in the version graph.

Recall the fact that whenever there does not exist a shared pointer to a node, it is deleted, hence garbage collected. We would like to make sure no needed nodes get deleted, so we have to manually maintain shared pointers pointing to needed nodes: for versions tracked by remote dataframes, maps where the remote node name maps to a shared pointer pointing to the version graph node is maintained, and updated whenever a remote node pulls or pushes from the local repository; for split/merge nodes, with manually maintained counters in each node, we add shared pointer pointing to that node to the splitting node set when a node has more than one adjacent new versions, or to a merging node set when it has more than one adjacent older versions; finally, as the ROOT and HEAD nodes shall never be deleted during a dataframe's lifetime (however, the HEAD version changes as new versions are created), and we need entry points to the version graph, two special pointers are owned by the graph pointing these two nodes. It is also important for iterators to hold shared pointers to both ends of the current edge, as either end being garbage collected is not desirable during a read operation.

If we reconsider the example in Figure 4.2, but with garbage collection in mind, we would see neither node A nor B is being pointed to by any shared pointer, thus would have been deleted, and that would be an accurate observation. However, if version B is tracked by a remote node N_2, in the remote reference map there would be a shared pointer pointing to node B which prevents it being garbage collected, as long as the remote node still tracks that version. By definition, we would also have a shared pointer pointing to the ROOT node in the split node set, and a shared pointer pointing to node C in the merge node set. For version A, if it is not tracked by any remote node at the moment, or any iterators, then there wouldn't be any additional shared pointers pointing to it, and Figure 4.2 correctly shows that it would be deleted, where the edges connected to it would be merged and connected to ROOT and C.

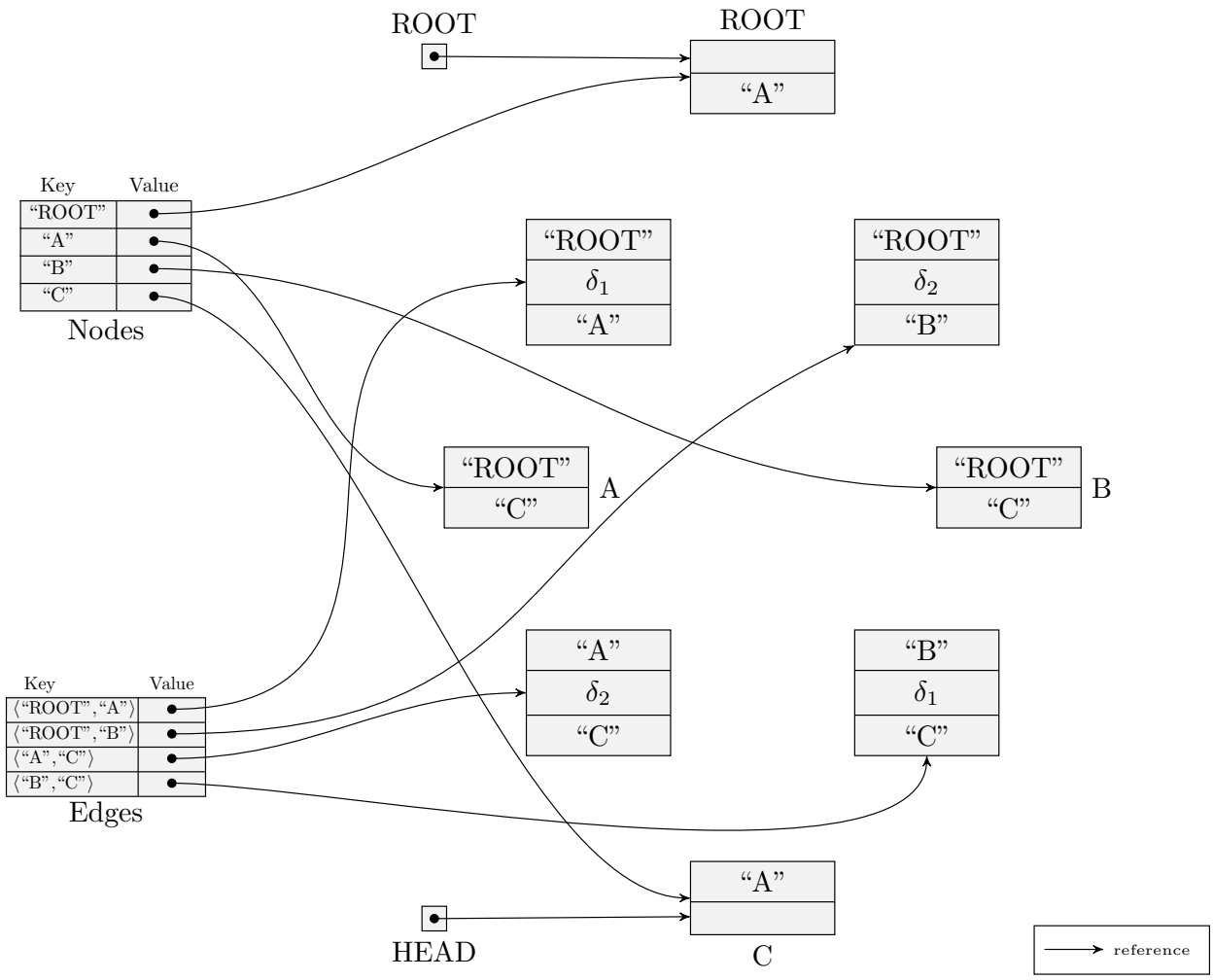


Figure 4.3: Example Version Graph Representation in Original Implementation.

With the additional constraint where garbage collection only occurs during writing operations, an additional shared pointer disposal set is owned by the version graph, where all the used shared pointers will be disposed into, either by iterators go to when the iterator moves forward, or by the remote reference map when pull operations are performed. This disposal set will only be cleared by writing operations, which ensures reading operations will not make any changes to the nodes in the version graph.

4.2 Comparison with Previous Design

Figure 4.3 shows the same abstract version graph structure shown in Figure 4.1, but represented by the original Python implementation. If we compare this representation with the one shown in Figure 4.2, we see a lot more indirection in accessing adjacent nodes and edges: in the original design, hash tables containing all nodes and all edges are created, where nodes and edges themselves only store the string tag to the nodes they are adjacent to. When accessing an edge adjacent to a node, a pair of strings, the tags of the current node and the node the edge connects to, will have to be used to query the edges hash table to find the corresponding edge.

Further, this original design performs garbage collection in a less efficient way compared to the new implementation: at each writing operation, the hash maps containing edges and nodes are manually iterated through to check for nodes that can be garbage collected. Compared to the automated reference counting mechanism used in the new design, this garbage collection mechanism clearly becomes much less efficient when the size of the version graph grows, which occurs when the number of GoT nodes in the application grows.

Chapter 5

Implementation of the Dataframe

While we have discussed the core version graph data structure, several other components are required to implement the API of dataframe. In this chapter, we take a look at other components in the re-implemented Spacetime core, and discuss important design decisions taken in each of the components.

5.1 Conflict Resolution

The version graph core data structure discussed previously does not handle or resolve conflicts, but simply maintain edge and node information received by function calls. To resolve conflicts, a version of operational transformation [13] is used to create merge versions and constructing delta updates between the merged version and the conflicting versions. Operational transform takes the branching version, the existing changes from the branching version up to the current latest version, and the new conflicting changes, and return the corresponding changes needed to be applied to the conflicting versions to a merged version. This process may involve calling a custom merge function provided by the programmer, as

the semantics of the values prevents conflicting values to be merged automatically. All delta updates are represented by json objects in Spacetime, where the deltas contains the updates on different scopes: data type, object, and dimension, hence field. A predefined set of rules is applied by OT to ensure the resulting delta updates satisfy causal consistency after the transformation and lead to a common merged state.

This process of creating a merged version from a total of three versions, with the two delta updates between the three versions, is referred to as a 3-way-merge step, and a similar technique is used in Git to resolve conflicts manually.

5.2 Asio Server and Client: Think Async

The dataframe provides pull and push operations to synchronize the local version graph with a remote repository, where the remote repository need to update the remote version graph, and respond accordingly. Thus a set of client and server component handling the communication is needed in implementing dataframe. In this section, we discuss the server and client implemented with the Asio C++ library.

5.2.1 Asio Library and the Proactor Pattern

Asio is a cross-platform C++ library for network and low-level I/O programming that provides developers with a consistent asynchronous model using a modern C++ approach [1]. Asio has been included as part of the Boost library since version 1.35.0, but is also distributed as a standalone library. In this implementation, we used the standalone version of Asio which has the additional benefit of being a strictly header only library.

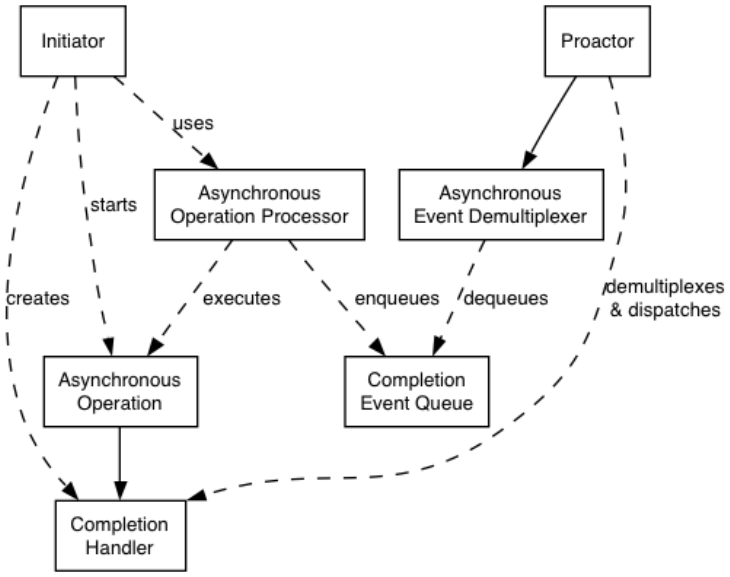


Figure 5.1: Proactor Pattern Used by Asio Library [2].

When handling asynchronous operations, Asio has an architecture designed around the proactor design pattern [25], a event driven handling design pattern providing high performance and great portability. Figure 5.1 shows the participants and the interactions between them in the proactor pattern. In the proactor pattern, asynchronous I/O operations are invoked by the application, and completion event queue will be notified when the operation is completed, where the proactor receives the event from the demultiplexer and calls the appropriate completion event handler.

5.2.2 Asynchronous Server Design

Both the server and the client are implemented based on TCP sockets. When the server starts, an asynchronous accept operation on the specified port is invoked, waiting for an incoming client connection, upon the completion of which a connection handler will start handling the requests from the newly connected client. Each client will repeatedly send push or pull requests, and the server would acknowledge the request in case of push, and retrieve the latest updates and reply the client with the data in case of pull. As both push and

pull requests are sent in a json format that has variable length (more precisely, the data is encoded in CBOR [7], a more concise binary representation), the length of the request data is sent before the actual request data; the same happens for the response to pull requests. This request-response exchange continues until the client sends a zero as request length, indicating the termination of the connection.

We use a thread pool structure in the server, where a finite number of threads are dispatched asynchronous event processing jobs, where each task typically involves the work that need to be done between asynchronous I/O steps. The overall handling of a connection is sliced by the I/O operations into smaller chunks that get executed as jobs in the job queue. However, the properties of the object version graph makes it desirable to adopt a special design with two job queues, which we will discuss in the next section.

A Tale of Two Queues

One important property of the object version graph data structure is that on the graph level, while writing operations are mutually exclusive, reading and writing operations can be concurrently executed, unless they are contending on the same nodes. This makes the simple design of using a single job queue quite inefficient: suppose there simultaneously exists reading and writing requests to be handled, where the number of both reading and writing operations is more than the size of the thread pool. In this case, if all writing operations are dispatched before any reading operations, there will only be one thread actually executing at any given time until the number of remaining number of writing operations is less than the size of the thread pool: handling threads of the writing operations are all waiting to acquire the lock on the graph. However, as we already know the writing operations on the graph can only be processed one at a time, we should be able to only have one thread process the writing operation, with all other threads in the thread pool concurrently process the reading operations, which are not mutually exclusive with the writing operation on the graph level.

If we are able to achieve this, the reading requests will receive their response much earlier than if the thread pool is essentially clogged by all writing operations.

The design we used in the asynchronous server is a two job queue design, where the first queue contains incoming connection acceptance, response, and reading request jobs, and the second queue only contains jobs that correspond to the step in handling a push request where writing operation to the object version graph is performed. The first queue is handled by a thread pool of a specified size, where the second queue is only handled by a singular thread, as only one writing operation can be performed on the graph. Initially, all connection would have a corresponding handler object, owned by a task in the first job queue, and when the handler sees a writing request, a job is created in the second queue owning the same handler, only performing the version graph writing step. After the writing step, a job is put into the first queue, handling the acknowledgement of the writing operation if needed, and the continuation of handling requests from the same client.

This double job queue design allows the server to handle as many requests concurrently as possible without increasing the size of the thread pool, as we exploited the fact where only one writing operation on version graph can be performed at any given time, and essentially prioritizing the reading requests to be handled by the server. Combining this and the other optimizations made in the server design along with the proactor pattern provided by Asio library, we have a quite efficient server constructed for the dataframe.

Specification of Server Behavior

The behavior of the server in perspective of one connected client is shown in Figure 5.2, where the details of the transitions and states in the figure is specified and explained in Table 5.2 and Table 5.1. It is worth noting that there are await versions of these operations

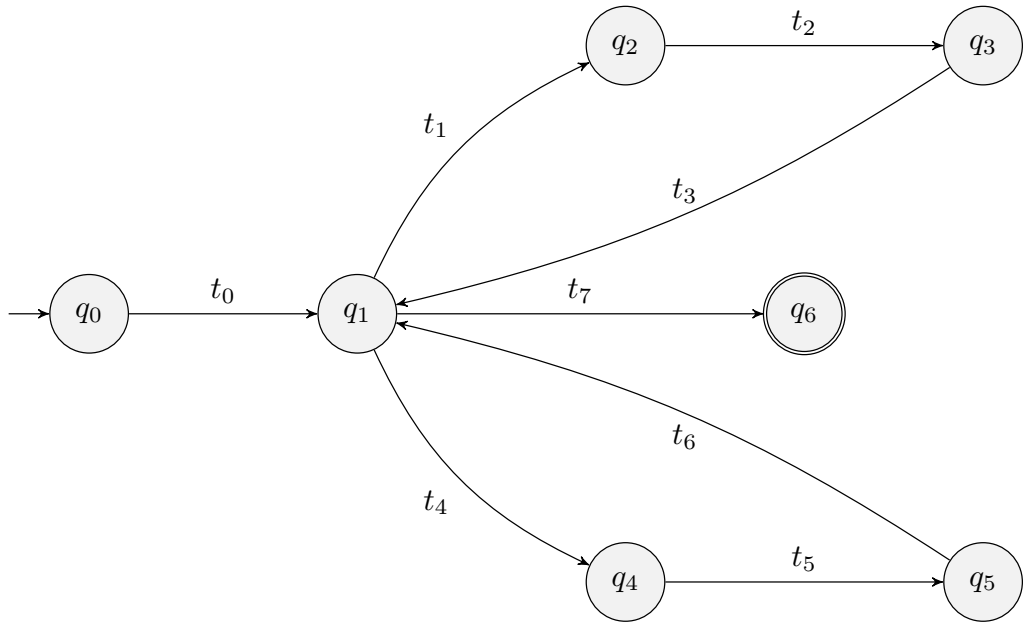


Figure 5.2: State Machine Representing the Server Behavior.

Table 5.1: Transitions of the Server.

Transition	**Events or Operations the Transition Corresponds to**
t_0	Receiving a connection from the client
t_1	Receiving a push request from the client
t_2	Sending an acknowledgement message regarding the push request
t_3	Writing to the local object version graph
t_4	Receiving a pull request from the client
t_5	Constructing a pull response message through reading operations on local object version graph, and sending the constructed message
t_6	Waiting for acknowledgement of the pull response message from the client
t_7	Received a end of connection request (request with zero length)

Table 5.2: States of the Server.

State	**Program State the State in State Machine Corresponds to**
q_0	Waiting for connection from client to accept
q_1	Connection established, waiting for a request from the client to handle
q_2	Handling a push request
q_3	Push request have been acknowledged, pending writing operation to local object version graph
q_4	Handling a pull request
q_5	Waiting for connection from client to accept
q_6	Stopped state, handler to be deleted

implemented by the server but are not shown in Figure 5.2. The await versions of these operations have slightly sequencing than the standard versions, but are largely similar.

More specifically, for the await versions, upon push, the acknowledgement is sent after writing to the version graph, whereas in the standard push it is sent before writing; and upon pull, the server will wait until some change to the version graph occur, and a response will only be constructed after the change, whereas in the standard pull, if no changes are present, essentially an empty response will be sent.

5.2.3 Client Design

Unlike the asynchronous server, the client making requests is designed to be synchronous. This matches with the behavior of the dataframe, where both pull and push operations are blocking operations to the application.

To an extent, the operations of the client can be considered the complement of the server shown in Figure 5.2, not in the strict finite automaton sense, but rather whenever the server is waiting for the server, the client will be performing some I/O operations, either acknowledging the response from the server, making a connection, or making a request; and whenever the server is sending messages, either responses or acknowledgements, the client will be waiting for and receiving such messages. As such, the general flow of operations of the client is similar to that of the server, but for the reverse of waiting and responding steps.

5.3 Core Library and Interoperability

Built as a core library, it is essential to have the ability for the re-implemented core of Spacetime to be wrapped into frameworks in a variety of high level languages. In this

section, we take a look at how the core library is wrapped into a Python framework which has a API that is essentially the same as the previous implementation, and discuss how the core library can be further used to construct libraries in other languages in the future to further expand the availability of Spacetime, and make the GoT programming model truly a cross programming language model.

5.3.1 Binding: the C++ side

The standard implementation of Python is written in C, thus in both C and in C++, native extensions for the Python runtime can be built. The C++ binding layer wrapping the core library is considered by the interpreter to be a new Python type, where upon the invocation of a method from the Python side, a C-style function would receive Python object pointers to all the parameters. With the core library accepting C++ types and the Python runtime only providing Python types, the data that go across the language boundary can only be binary data in order for the core library to have a language independent API. Thus, the C++ side of the binding receives the serialized binary data, and is in charge of deserializing the data back into C++ json type. All data passed between the binding layer and the core library are in form of either json data, or data of pritimitive types, including strings and integers.

One caveat in the implementation of the C++ side binding is the reference counter used by the standard Python implementation to garbage collect need to be correctly maintained to avoid memory leaks. Using a modern C++ programming idiom, resource acquisition is initialization (RAII), a Python object reference counter guard is designed in a similar fashion to that of the standard lock guard type in the standard library, tying the reference count to a scope in C++, eliminating the chance of a missing reference count decrease. Additionally, when the custom merge process is required by the library during conflict resolution, a Python

callback function need to be invoked by the C++ binding. When the Python function is executed, the global interpreter lock need to be acquired by the binding layer, otherwise the interpreter may see inconsistent internal states for GIL, which would cause exceptions for the interpreter. With these caveats taken care of, the C++ layer in the binding can properly handle interactions with Python interpreter.

5.3.2 Binding: the Python side

While it is technically possible to build the binding layer in a single layer, it would involve having C++ code manipulate pure Python types and objects, and becomes extremely tedious through the C API of Python. Thus a two-layered design is used in the implementation of the binding. The Python side of the binding holds an instance of the C++ layer extension, and handles a couple of extra tasks, namely type information handling and creating temporary objects when the custom merge function is being invoked.

Type Information Handling

With the type information required to be passed through the language boundary into the C++ core library, the binding layer on the Python side need to preprocess the type and dimension information into a json object, which then get serialized and passed to the C++ binding layer. This type information is used by the core library to manage the interest in types in the object set, where upon receiving a diff update on the entire set, the type list included in the request would be compared with the local type list to only process and respond with relevant objects with the given type.

Custom Merge Function Handling

The custom merge function provided to Spacetime through the provided API takes the form of a function taking three object of the same type as parameter, and return a single merged object. The core library and the version graph in general stores json data representing updates, in contrast to the state, in object version graph. When updates are combined into a complete state, the data stored in the tracked objects are present, but in order for the provided merge function to be invoked, temporary objects containing the actual data need to be constructed and passed to the merge function.

The core library takes a callback function in C++ that operates on json data, provided by the C++ binding layer wrapping a Python function into a C++ function which when invoked calls the Python callback function with appropriate serialization, where the Python layer of the binding creates a Python function with a desired API from the custom merge function, to be pointed to and called by a C++ functon. This process includes deserialization of passed data, injecting the data into temporary objects, calling the merge function, and creating corresponding diff from the final object. Table 5.3 further explains this complex callback wrapping process in detail.

Table 5.3: Chain of Callback for Custom Merge Function.

Provided By	**Parameters**	**Operations**
User	Python objects	Returns merged object
Python binding	Binary data	Deserialize data, creates temporary Python objects and stores data into objects, calls previous function, creates diff from the returned object, and serialize the diff to be returned to next level
C++ binding	C++ json objects	Serialize json objects into binary data, calls previous function, gets the binary data returned by previous function, and deserialize into C++ json object to return to core library

5.3.3 Potential Interoperability with Other Languages

One great benefit of re-implementing the core library in C++ is that libraries written in C/C++ can also be easily incorporated into native extensions of other languages, including the standard implementation of Python, the extension of which we have already wrapped the C++ version into.

However, it should be noted that implementing such a binding layer is not quite trivial, and to create a complete and usable version of Spacetime for a new language requires several language specific components created for that language, namely the object heap, the utility to transform type information into json data for the core library to handle and query, and the ability to have callback interoperability between the core library and the runtime environment of the higher lever language.

While not being straightforward, adapting the core library into a framework in a new language is definitely feasible: this is shown by creating a demo in javascript [28]. The existence of such a demo still illustrates the potential flexibility of the core library being adapted into frameworks written in different languages that can synchronize over the same set of objects across languages through the underlying library.

Chapter 6

Experiments and Evaluation

In this chapter, we examine the results of a set of benchmarks designed to illustrate the scalability of the re-implemented Spacetime, comparing with the Python version, and Parse [24, 19], an open source framework implemented in javascript that provides object storage and synchronization. The set of benchmark take inspiration from Yahoo Cloud Serving Benchmarks (YCSB) [10], and measures the update latency of changes made to the set of shared objects, hence the time it takes for an update to be propagated through the network for other nodes to see the changes.

6.1 Benchmark design

The application consists of three types of nodes: reading, writing, and the server. A set of objects are shared amongst all the nodes, where initially we have an empty set containing no objects. Each object contains two dimensions, primary key 'oid', the ID of the object, and 'create_ts', which stores the time of creation of the object. When the benchmark is executed, all writing nodes start to create new objects locally, and push the update after each creation

of a new object. The reading nodes repeatedly pull from the server node, calculating the update latency upon receiving updates containing newly created objects.

Various variables are changed with a range of configurations, including the total number of objects collectively being created by all writer nodes, the number of client nodes, and the percentage among the client nodes that are writer nodes. For each configuration, the median of update latency calculated by the reader nodes are stored, and used to analyze the performance. A very similar set of benchmark is created for Parse with the live query feature.

6.2 Benchmark Setup

The server node of the benchmark is located in a Amazon EC2 c5.2xlarge instance, which come with 4 CPU cores and 8GB of memory. Sets of benchmarks are ran with the server located in different regions to analyze the change of performance when the physical distance, hence the inherit network latency, changes. The server was deployed to AWS regions including us-west-1 (N.Virginia), us-east-1 (N. Virginia), ap-northeast-1 (Tokyo), eu-central-1 (Frankfurt), and ap-southeast-1 (Singapore). The relation of the regions and a measurement of the round trip network latency with the average value of 50 pings to the server can be seen in Table 6.1. All update latency are measured from the west coast of US, the UC Irvine campus.

Table 6.1: Ping Latency of Servers Located at Different Regions.

Server Region	**Ping Latency (ms)**
Norhtern California	12.102
Norhtern Virginia	68.023
Tokyo	130.646
Frankfurt	162.939
Singapore	202.731

The client nodes, including the reading nodes and the writing nodes, would need to be located in the same physical machine in order for meaningful update latency comparisons. As we aim to evaluate the performance of the system with the workload scaling up, we need to have a machine capable of running hundreds of processes simultaneously to simulate having hundreds of independent client nodes across the network. Thus we used a machine with four Intel Xeon CPU E5-4650 v4 processors, a total of 56 cores (112 with hyperthreading), and has 256GB of RAM. This machine is located on the west coast of the US, Irvine.

The C++ core library used in the benchmark was compiled with GCC 10.2.0, and ran with Python 3.9.0. Parse is deployed and ran on node 15.3.0, with the server deployed with the latest docker image.

6.3 Results and Analysis

In this section, we look at the results from the benchmarks and analyze the data with multiple plots in several dimensions, including varying the number of clients, the writer percentage with the same number of client nodes, the physical distance between the server and client machine, and a broader view on the performance comparison of the two versions of Spacetime.

6.3.1 Varying Number of Clients

We can see a comparison between the two versions of Spacetime when the number of clients varies, while all other variables are kept constant, in Figure 6.1. With the same 1000 objects, and with the server machine located in Northern California, and always having 20 percent of client nodes being writers, the update latency of the three frameworks are measured for having 10, 20, 50, 100, 200, 300, 500, and 800 client nodes. The set of benchmarks ran for

Parse only goes up to 200 clients, as setting up and shutting down the server side for Parse have proven to take a significant amount of time, and the primary focus is to compare the two version of Spacetime. Figure 6.1a only shows the test cases with client node count up to 200 to show the details, and is a subset of Figure 6.1b which shows the whole set of results.

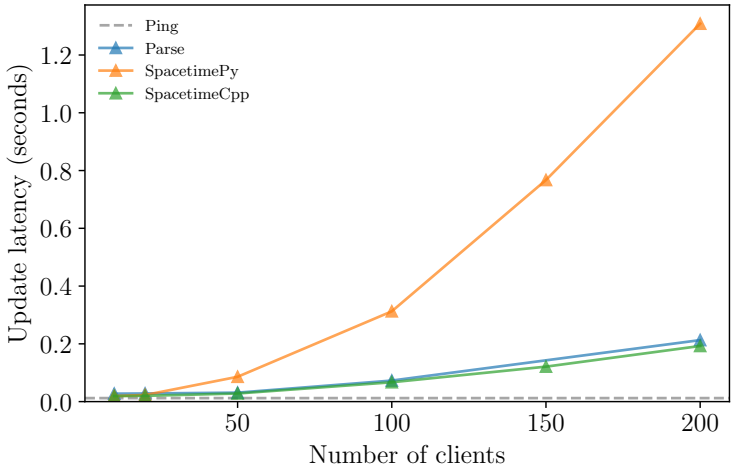
With a somewhat small but reasonable percentage of writer nodes, specifically 20 percent, among all client nodes, we can see in Figure 6.1a that with this scale of workload and client nodes, we see the Python version of Spacetime suffers greatly when the number of clients increase to more than 100 when compared with the other frameowkrs, where the new implementation performs at an extremely similar level to that of Parse. As the number of client nodes further increase in Figure 6.1b, we see the performance advantage of the new version persist.

This significant performance improvement exactly meets our expectations: as the number of concurrent connections increase, the Python version becomes severely limited by the global interpreter lock, when we have concurrent CPU intensive operations, including concurrent reading and writing operations on the object version graph. The new implementation of Spacetime allows for true concurrency among threads handling connections from client nodes, and an efficient architecture enables more effective asynchronous I/O operations.

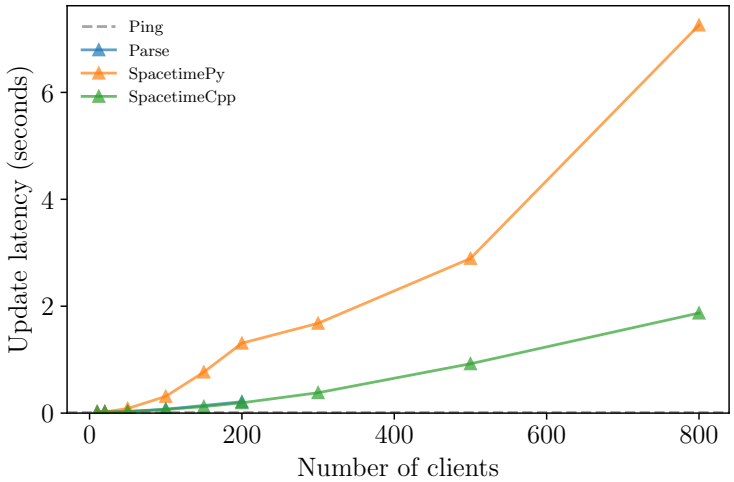
6.3.2 Varying Writer Percentage

We also examine how the frameworks under test behave with differing percentage of the client nodes being writer nodes, hence the performance under differing level of contention on the object version graph.

We primarily focus on test cases where the server is located close to the client machine (more precisely, Northern California), as increasing the physical distance between the server and



(a) Up to 200 Connected Clients.

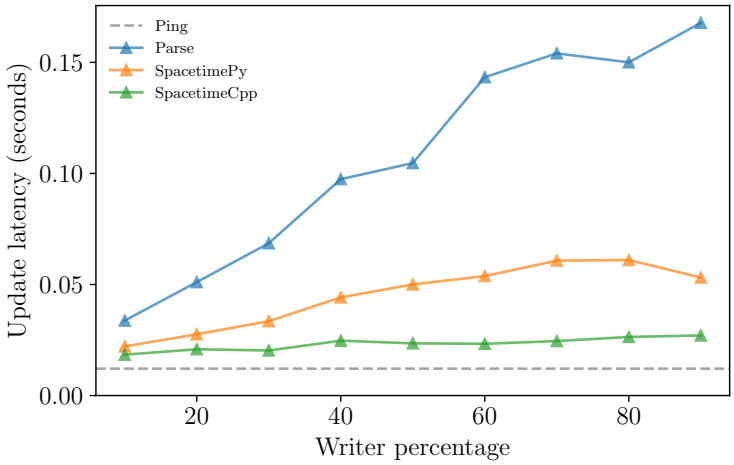


(b) Up to 800 Connected Clients.

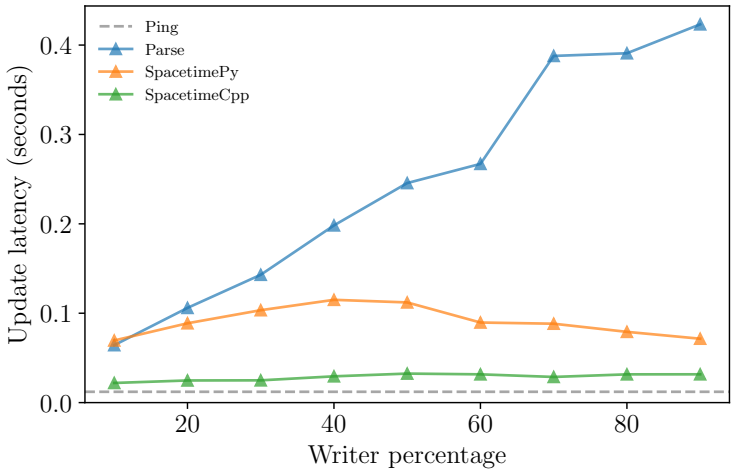
Figure 6.1: Update Latency vs. Client Count, N.California, 1000 Objects, 20% Writer Nodes.

client may have the side effect of decreasing the level of contention on the server side, which we will also discuss in the next section. With the writer percentage ranging from 10 percent to 90 percent, Figure 6.2 compares the performance of the two versions of Spacetime against Parse.

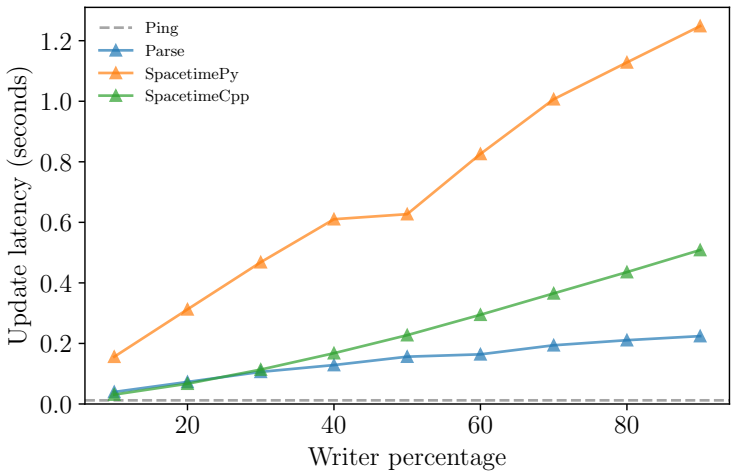
We see both versions of Spacetime stacks against Parse very well when we have a comparatively small size for the set of objects, as in Figure 6.2a and 6.2b, where there are a total of 100 objects created. As the percentage of writer nodes increase, we see a steady increase in the update latency of Parse, where both versions of Spacetime do not see such a linear



(a) 100 Objects, 50 Client Nodes.



(b) 100 Objects, 100 Client Nodes.



(c) 1000 Objects, 100 Client Nodes.

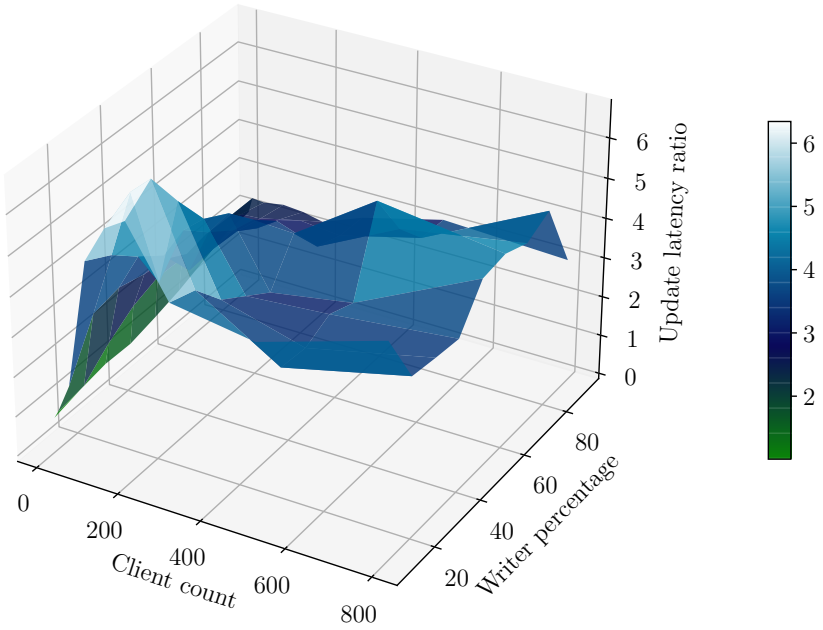
Figure 6.2: Update Latency vs. Writer Percentage, N.California.

increase in the latency. Specifically, the new implementation only see minor variation over the percentage of writers being deployed, and always maintains a latency that is very close to the ping latency, the best possible latency for the given distance between the server and the client, compared to the other two frameworks.

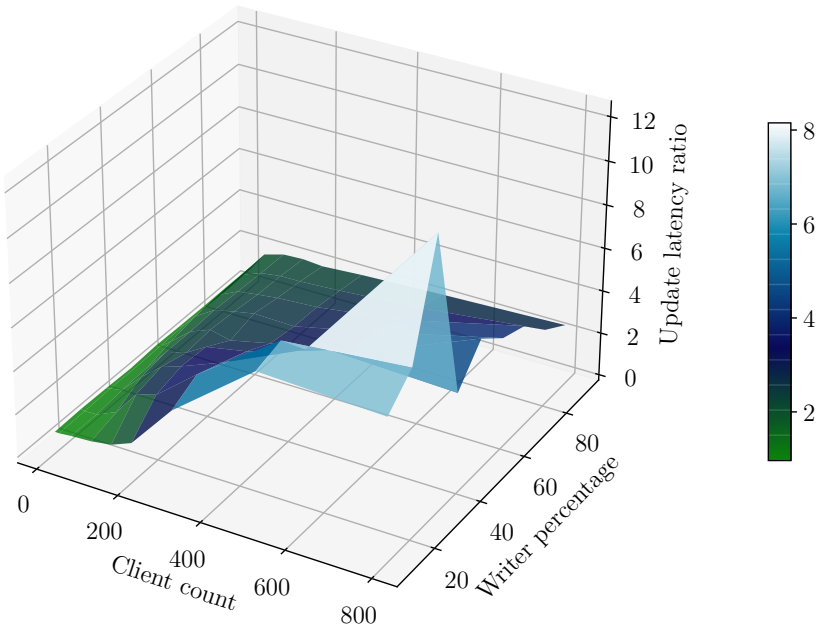
In Figure 6.2c, when we increase the number of objects to 1000, both versions of Spacetime get outperformed by Parse. This behavior, however, is to be expected: Parse uses MongoDB as the underlying DBMS, which guarantees object level eventual consistency, a weaker consistency model when compared to causal consistency on the entire object set level, which is provided by Spacetime. With the objects stored in MongoDB indexed by identifiers, the increase of the size of the object set does not negatively impact the performance of Parse. For Spacetime, however, with the increase of the objects set size, the complexity of combining delta updates increase and thus increase the update latency of both versions of Spacetime. On the bright side, we can see in Figure 6.2c that the new implementation of Spacetime still consistently outperform the original Python implementation by a decent margin. At lower level of contention, the C++ implementation still performs at a similar level to Parse.

To further compare the two implementations of Spacetime, we examine how the ratio of update latency of the Python version over the new implementation change over combinations of the number of connected clients and the percentage of writer among the clients. Figure 6.3 includes the result of two set of benchmarks, where Figure 6.3a included data collected from having the server located in Northern California, and Figure 6.3b has the server located in Tokyo. We can observe from both of these figures the new implementation consistently outperforms the original version by a significant margin as the work scales up, and can verify the effectiveness of the new designs taken in implementing the core library.

An interesting observation we can make from these figures is the fact that in Figure 6.3a, as the number of clients increase, the ratio takes a peak with the number of clients at around 200, whereas in Figure 6.3b, the ratio does not change as significantly until the client count



(a) Server Located in Northern California.



(b) Server Located in Tokyo.

Figure 6.3: Update Latency Ratio (Python version / C++ Version) vs. Client Count and Writing Percentage, with 1000 Objects.

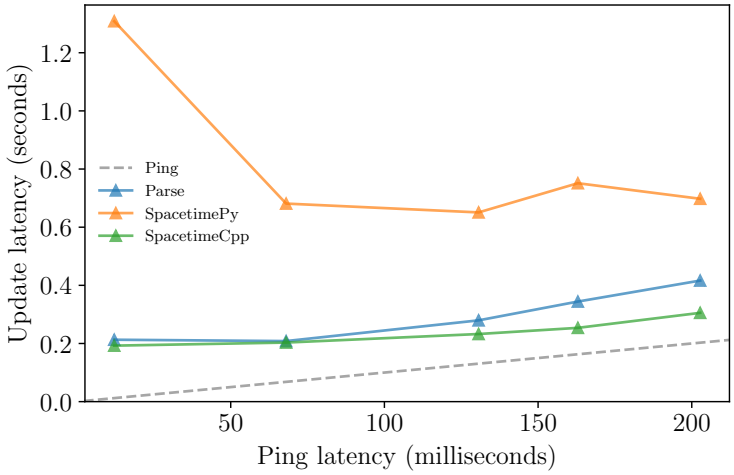
reaches 500 and 800. One likely explanation of this behavior is based on the hypothesis of the saturation of the server's ability to handle concurrent clients: there exists a threshold of the amount of concurrent requests the server handles, under which the latency increases slowly, and the throughput of the server increases as concurrent requests increase; however as the number of requests goes above the threshold, hence when the server is saturated, the throughput ceases to increase, and in turn the latency would increase rapidly. This general behavior has been observed and modeled in behaviors of servers [23], and even going back to behavior of transistors [26].

The peaks in these figures can be explained by this hypothesis as follows. When the server located close to the clients, with more than 300 clients, both versions of Spacetime are saturated, and due to the C++ implementation having the benefit of not being limited by GIL, and a more optimized design, the performance is consistently several times better. When there are between 100 and 300 clients, with a lower capacity of handling requests concurrently, the Python version of Spacetime saturated its capacity which cause the latency to increase significantly, where the C++ version is still within its threshold, thus the latency only increase slightly, causing a peak in the latency ratio. When the server is further away, as in Figure 6.3b, the level of contention is overall lowered, further delaying the saturation of the new implementation of Spacetime, thus causing a spike only after the client count exceeds 500. We will also see the effect of increased network latency on delaying the saturation in the next section.

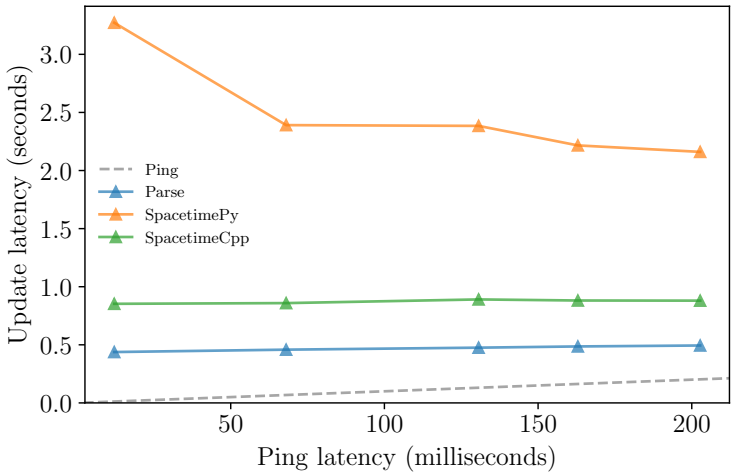
6.3.3 Varying Network Latency

As shown in Table 6.1, the same set of benchmarks is ran separately with the server located in five regions under a range of network latency conditions. In Figure 6.4, we compare the update latency of the three frameworks under varying network latency conditions. The

conceptually optimal latency, hence the latency reported by ping, is shown as a dashed grey line in the plots.



(a) 20 Percent Writer Nodes.



(b) 60 Percent Writer Nodes.

Figure 6.4: Update Latency vs. Ping Latency, 1000 Objects, 200 Clients

In Figure 6.4a, we have a total of 1000 objects, 200 clients, and 20 percent of the nodes, hence 40 out of the 200 total nodes, as writer nodes. A quite peculiar, and somewhat counterintuitive behavior of the Python version of Spacetime is when the network latency increases from 12 milliseconds (Northern California) to 68 (Northern Virginia), the update latency actually decreased significantly. However, as we hypothesized earlier, this quite likely is a result of lower network latency causing a much higher level of contention, which

in turn pushed the server into saturation, causing the performance to degrade. Higher level of contention also creates more intermediate versions in the version graph, causing more merging to be performed, especially when the server struggles to handle previous requests promptly, which further explains the saturation. As the network latency increases, the level of contention lowers, and considering the routing process over the network, requests may arrive in a more sequential fashion on the server side compared to the situation under lower network latency. These conditions result in the Python implementation being in a less saturated state when the network latency increases, hence better performance with longer physical distance between the server and the client.

While the new implementation marginally outperforms Parse overall in Figure 6.4a, when the writer percentage is increased to 60 percent as in Figure 6.4b, both versions of Spacetime see a bigger hit in performance compared to Parse, as we have seen in Figure 6.2c, as we do have a larger object set. However, the new implementation of Spacetime still consistently outperform the original Python version regardless of the physical distance between the machines the GoT application is deployed to.

6.3.4 Final Verdict

We evaluated the performance of both versions of Spacetime and Parse with respect to update latency, hence the time taken for a local change to be propagated to reading nodes. As we examine several independent variables, including the number of client nodes, the percentage of nodes being writers, and the network latency between the client and the server machine, it has been shown that whenever the workload scale up, the new implementation of Spacetime in C++ consistently outperforms the Python version, and perform at a similar level, or even outperform a popular open-source framework, Parse, in many configurations.

Overall, this set of benchmark demonstrates the significant improvement brought by the new version, and shows promising performance in general. We are also able to hypothesize on the reasons behind the behavior of each framework when compared against each other.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, a redesigned and re-implemented version of Spacetime, a framework realizing the Global Object Tracker (GoT) programming model, is presented. We briefly discussed the underlying model GoT that approaches distributed object replication with version control techniques similar to that used by Git, and the automated conflict resolution approach taken in the model. We further discussed the architecture and internal design decisions taken in the new implementation wrapping the core components of Spacetime into a library reusable by various high level languages. We further examine the performance of the C++ implementation against the original Python implementation, and the Parse framework, where the benchmark have shown significant performance improvement over the previous version of Spacetime, and comparable or better performance to Parse in many configurations. Adding the performance improvement to the higher level of flexibility provided by having a lower level library implementing the core components of GoT, we have a promising improved version of Spacetime over the original implementation.

7.2 Future Work

In this section, we introduce several interesting possible future development based on the work presented in this thesis.

The set of benchmark used to evaluate the performance of the frameworks do provide insightful results, but further evaluations on how the frameworks perform under more realistic workload, where the application logic is more complex, object size increase, and when nodes are distributed across the network over multiple regions, would be very helpful in assessing the usefulness of the re-implemented Spacetime framework in practice.

Previously, an interactive debugger has been developed for the Python version of Spacetime [4], which potentially can be redesigned to work with the core library, allowing a more flexible adaptation of the debugger in multiple languages and environments, when the core library get incorporated into frameworks in languages other than Python.

While a peer-to-peer (P2P) version of the GoT model has been theorized and proposed [3], a fully working prototype is still in the works. When the P2P prototype get completed, with the increased amount of version merging being performed, a highly efficient implementation would surely be essential for the P2P version to become practically useful. The general design and important optimization mechanisms taken in the work shown in this thesis should provide a helpful guidance when said efficient implementation for the P2P model need to be constructed.

In conclusion, there are a wide range of interesting future research topics stemming from the work done on GoT programming model and Spacetime framework, and it would be exciting to see the upcoming work being done in the area.

Bibliography

[1] Asio C++ Library. `https://think-async.com/Asio/`. Accessed: 2021-04-14.

[2] The Proactor Design Pattern: Concurrency Without Threads. `https://think-async.com/Asio/asio-1.18.1/doc/asio/overview/core/async.html`. Accessed: 2021-04-14.

[3] R. Achar. *The Global Object Tracker: Decentralized Version Control for Replicated Objects*. PhD thesis, UC Irvine, 2020.

[4] 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.

[5] K. Banker. *MongoDB in action*. Manning Publications Co., 2011.

[6] C. Biyikoglu. Under the hood: Redis crdts (conflict-free replicated data types). Retrieved Apr-2018.

[7] C. Bormann and P. Hoffman. Concise Binary Object Representation (CBOR). RFC, RFC Editor, December 2020.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] P. DuBois and M. Foreword By-Widenius. *MySQL*. New riders publishing, 1999.

[13] 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.

[14] S. Feuerstein and B. Pribyl. *Oracle pl/sql Programming*. " O'Reilly Media, Inc.", 2005.

[15] 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.

[16] Programming languages — C++. Standard, International Organization for Standardization, Geneva, CH, Dec. 2020.

[17] G. Kaki, S. Priya, K. Sivaramakrishnan, and S. Jagannathan. Mergeable replicated data types. *Proc. ACM Program. Lang.*, 3(OOPSLA), Oct. 2019.

[18] 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.

[19] C. G. Kim. A study of utilizing backend as a service (baas) space for mobile applications. In *International Conference on Intelligence Science*, pages 225–236. Springer, 2019.

[20] 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.

[21] C. V. Lopes, R. Achar, and A. Valadares. Predicate collection classes. *Journal Of Object Technology*, 16(2):3–1, 2017.

[22] B. Momjian. *PostgreSQL: introduction and concepts*, volume 192. Addison-Wesley New York, 2001.

[23] J. Neilson, C. Woodside, D. Petriu, and S. Majumdar. Software bottlenecking in client-server systems and rendezvous networks. *IEEE Transactions on Software Engineering*, 21(9):776–782, 1995.

[24] P. Nguyen. Mobile backend as a service: the pros and cons of parse. 2016.

[25] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-oriented software architecture, patterns for concurrent and networked objects*, volume 2. John Wiley & Sons, 2013.

[26] A. S. Sedra, D. E. A. S. Sedra, K. C. Smith, and K. C. Smith. *Microelectronic circuits*. New York: Oxford University Press, 1998.

[27] D. Spinellis. Git. *IEEE software*, 29(3):100–101, 2012.

[28] S. Venkatramana. Actualizing the got model as a multi-language, multi-platform distributed system through interoperable spacetime dataframes. Master's thesis, UC Irvine, 2021.

[29] C. Wu, J. Faleiro, Y. Lin, and J. Hellerstein. Anna: A kvs for any scale. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 401–412, 2018.