UNIVERSITY OF CALIFORNIA,
IRVINE

GoTcha: An Interactive Debugger for GOT-based distributed systems

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Software Engineering

by

Pritha Dawn

Thesis Committee:
Professor Cristina V. Lopes, Irvine, Chair
Assistant Professor Joshua Garcia
Associate Professor James A. Jones

2020

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to thank my advisor Professor Cristina V. Lopes for her invaluable guidance. I would also like to express my sincere appreciation to Rohan Achar for his support and insights along the way.

ABSTRACT OF THE THESIS

GoTcha: An Interactive Debugger for GOT-based distributed systems

By

Pritha Dawn

Master of Science in Software Engineering

University of California, Irvine, 2020

Professor Cristina V. Lopes, Irvine, Chair

Debugging distributed systems is difficult. Most of the techniques that have been developed for debugging such systems use either model checking, or postmortem analysis of logs and traces. Interactive debugging is a widely-used technique for debugging single threaded applications. But only a few interactive debuggers for distributed systems are available as the techniques used for building traditional debuggers for sequential single-threaded programs cannot be easily transferred to a distributed setting. In this thesis, I look at the topic of designing interactive debuggers for distributed systems and introduce GoTcha which is an interactive debugging tool for distributed systems based on the GoT distributed programming model. GoTcha supports the core features of traditional single-threaded debuggers like step-through debugging and breakpoints so that users can wield these powerful tools in a distributed context. I demonstrate the capabilities of GoTcha through a series of usage examples, and discuss the design and architecture of GoTcha.

Chapter 1

Introduction

Debugging distributed systems is difficult. The inherent features of distributed systems like concurrency and distributed state [5] make distributed systems hard to reason about and the non-determinism introduced in the system due to network delays and system failures make bugs difficult to reproduce. Traditionally, a wide range of methods are employed to debug distributed systems; from basic unit-testing and print statements, all the way to more sophisticated techniques like log analysis [13], record and replay [15] and formal methods like theorem proving [25] and model checking [27]. However, one technique that is quite popular for single threaded systems, interactive debugging, is almost never used. This is because building an interactive debugger for a distributed system is hard and consequently, very few interactive debuggers for distributed systems exist.

The techniques used for building traditional interactive debuggers for single-threaded programs cannot be applied to a distributed context due to the absence of a single *controller* in a distributed system unlike in a single-machine program. Any interactive debugger needs to support *step-through* debugging. To do this, the debugger needs to take control of the flow of execution so that it is able to pause execution at any point, let the user observe the state at that point, and then resume execution from that point. Traditional interactive debuggers for single-machine systems resolve this problem by using the debug APIs exposed by the native operating system or the virtual machine (e.g. JVM) - the agent which is in control of the execution of all the programs inside a machine. In a distributed system where there are multiple machines running independently and communicating over the network, no single *controller* exists which a debugger can use to take control of the execution.

This thesis presents an interactive debugger for a recently proposed distributed programming model, GoT [3]. GoTcha is an interactive debugger for GoT-based distributed systems. GoTcha provides all the core features of traditional single-threaded debuggers like step-through debugging and breakpoints so that the users can wield these powerful tools in a distributed context. GoTcha lets the user step-through the program, and inspect the new state and the change in state at each step. GoTcha supports conditional breakpoints. When the breakpoint predicate becomes True, it stops the application and provides the user with a list of the operations executed in the system till that point. In addition to these, GoTcha supports exploratory testing inasmuch it lets the user simulate dropping and delaying messages over the network.

The rest of the thesis is organised as follows. In chapter 2, I discuss related work. In chapter 3, I discuss the fundamental goals of an interactive debugger for distributed systems and examine the role of the underlying distributed programming model in the design of an interactive debugger. In chapter 4, I introduce the GoT distributed programming model and in chapter 5, I introduce GoTcha which is an interactive debugging tool for GoT-based distributed systems, and show how GoTcha achieves the goals identified in chapter 3 through a series of usage examples. In chapter 6, I discuss the detailed architecture and design of GoTcha. In chapter 7, I discuss directions for future work.

Chapter 2

Related Work

Interactive debugging of parallel and distributed systems has been discussed as early as 1981 [21], but the idea has never been fully realized, mainly because it is very hard to do. The reasons for the difficulty are described by Cheung et al [8] in detail - the problem of maintaining precise global states, a large state space, interaction among multiple asynchronous processes, communication limitations and error latency.

However, there are many non interactive tools which can help developers in debugging distributed applications. A comprehensive survey of the types of methods available can be found in Beschastnikh et al. [5]. 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. Formal methods like model checking and theorem proving can provide guarantees about the correctness of the system but struggle with scalability. Tools for record and replay [15, 18, 14], tracing [19, 11], log analysis [13], and visualizers [5], 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. Record and replay tools place a lot of overhead on the system and suffer from the probe effect [20]. Tracing is used to track data flow across process and machine boundaries but is less useful for the purpose of observing state changes in the entire system. Log analysis is light-weight but the sheer volume of information makes it harder for developers to derive useful information from the data. Visualizers help users make sense of the logs by building visualizations of the system execution out of the logs.

Many of these tools share features with interactive debuggers, as they share the common objective of exposing errors in the system to the developers. For example, tools like ShiViz [5] 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. Causeway [23] is a message-oriented post-mortem debugger which attempts to trace message flow across processes so that a user can find the source of a bug by backtracking across the chain of messages. An interactive debugger would also need to track message-passing in the system. D^{3}S [18] 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.

Comparatively few interactive debuggers for distributed systems exist as opposed to the variety of *post-mortem* tools described above. TotalView [1] and p2d2 [17] are interactive debugging tools which are basically extensions of traditional debuggers to a distributed system. As such, the amount of information they display might be overwhelming for a user as the user needs to keep track of debugging information for multiple processes at a very fine level of granularity i.e. line of code. Millipede [24] is a multi-level interactive debugger for distributed systems which attempts to reduce this information overload on a developer debugging a distributed system containing multiple processes by displaying the debugging information to the user at different levels of abstraction ranging from process-level to message-level to protocol-level. While an important task for a debugger for distributed systems is to filter out unimportant information, Millipede does not seem to provide a visualization of causal ordering between events which is helpful for a developer to be able to trace the origin of a bug.

REME-D [6]is an interactive debugging tool meant specifically for mobile applications developed using the Ambient-Oriented programming Model(AmOp). AmOp is a variant of the Actor Model with non-blocking communication and event-loop concurrency. REME-D has a decentralized design in the sense that the debugger does not attempt to control the entire application. Only one actor is paused at a time and its state inspected. This approach is useful in scenarios where a user has some intuition that the fault lies in a particular node. The user can use the debugger to observe the changes in the state of that node over the course of the execution. This can become cumbersome when the system has a large number of heterogeneous nodes and the user has to do several debugging sessions - one session per node.

Recently, a graphical interactive debugger for distributed systems called Oddity [26] was presented. Using Oddity, programmers can observe communication in distributed systems. They can perturb these communications, exploring conditions of failure, reorder 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.

Chapter 3

Designing an Interactive Debugger for Distributed Systems

In this section, we discuss the design of an interactive debugger for distributed systems at a high level. To design any system, we need to identify the requirements first. For that reason, we discuss the high-level goals of an interactive debugger first and show that the underlying distributed system model determines how an interactive debugger needs to be designed to achieve these goals. We identify features of distributed system models which facilitate interactive debugging and conclude with a discussion on the design of an interactive debugger for few widely-used models, considering these features.

3.1 Goals of an Interactive Debugger for Distributed Systems

Any interactive debugger needs to be able to control the flow of execution and expose state changes in a system. In a distributed system, the debugger also needs to show some form of

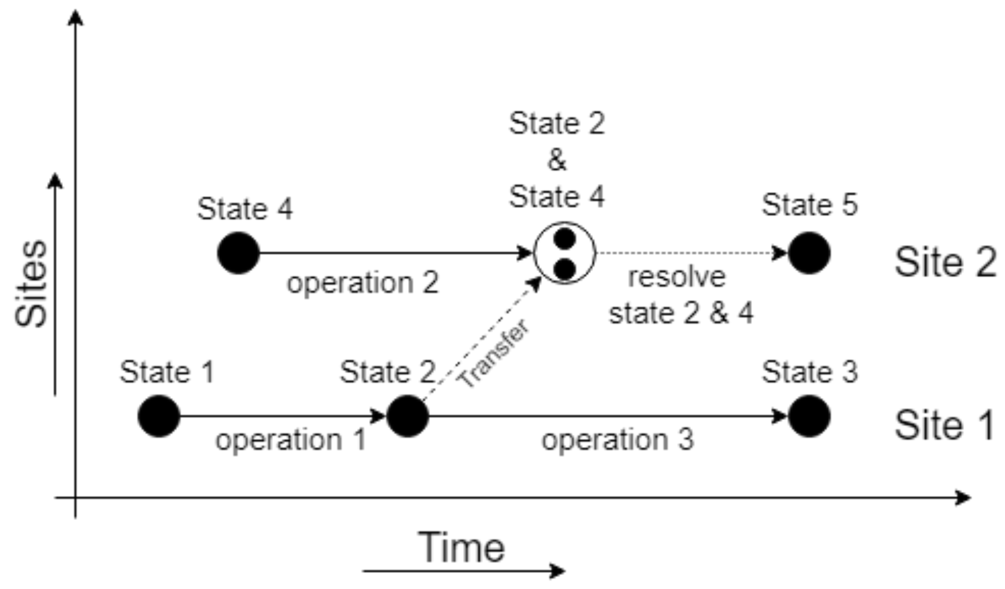


Figure 3.1: State changes in a distributed system

the *history of execution* to the user. The history, i.e. the sequence of events and state changes leading to a particular state, is needed to trace the origin of a bug. Inferring this sequence for a user is complicated in a distributed context unlike for a single-threaded program because of the inherent non-determinism of a distributed system. In this section, we discuss each of these goals of an interactive debugger for a distributed system - expose state changes, control flow of execution and expose history of execution.

3.1.1 Observe State Changes

A program is a sequence of operations on an initial state of a set of variables. Each operation causes a mutation in the previous state. So, any program execution can be described completely by the initial state, the sequence of operations and corresponding state mutations. The task of an interactive debugger is to make these operations and internal states observable to a user so that the user can detect erroneous states, if any.

The state at any node in a distributed system changes due to its local execution as well as due to its interaction with the rest of the distributed system. Fig 3.1 depicts the state changes in a distributed system. The nodes communicate with each over via different protocols. Any such communication encapsulates some information about the current state of the sending node and so, can be considered to constitute a transfer in state. On receiving the changes, the recipient node needs to process the incoming communication and *reconcile* its local state with the incoming state to construct a consistent final state.

So, the debugger in a distributed system needs to expose three types of state changes - **state changes due to local execution**, **transfer of state**, and **state changes due to reconciliation of states received from remote nodes with the local state**.

Traditional sequential debuggers already show state changes due to local execution. Interactive debuggers for a distributed system can be integrated with the traditional debuggers to expose local state changes.

To show transfer of state, the debugger would need to observe all the communications within the system. In order to do this, it becomes necessary to understand the underlying distributed system model because the *connectors* of the underlying model need to be identified. For example, let us consider a very simple model which provides the distributed system nodes with a function called *Send(message, destinationNode)* to send a message to another node. This *send* function could internally uses TCP/IP sockets to send the message in a serialized form to the recipient node. In order to enable a debugger to *snoop* on communications within this system, we could build a wrapper around this *send* function which would encapsulate two *send* calls - one to the debugger and the other to the actual *destinationNode*. So, in the debug mode, every time a node would invoke *Send* to send a message to another node, the message would get sent to both the debugger and the recipient node. In this way, once the connectors in a distributed system model are identified, the debugger can be designed to latch onto these connectors to intercept the communications.

Exposing reconciliation can be tricky. Many distributed models do not provide an explicit mechanism to nodes to accept updates from outside. These models accept updates as soon as they are received. For example, in a distributed system which uses CRDT [22] sets for synchronization, when a node A adds an element to its set, the element also gets added to the set at node B after some point in time if there is no network disruption. Node B has no control over when its local state i.e. the set gets modified by an outside update. The model has taken away this control from the node. In these systems, since reconciliation is hidden, it is complicated for a debugger to understand when an outside update modified the local state compared to models which provide an explicit mechanism to a node to accept outside events. For example, in Actor models [4], updates are not applied immediately. Incoming messages are stored in a queue first. It is left to the node to pop the message from the queue and process the message whenever it wants.It is simpler to expose reconciliation in this case because the debugger *knows* that reconciliation happens when a message is popped from the queue and so, can track the queue *pop()* operations.

3.1.2 Control Flow of Execution

As mentioned earlier, there are two main debugging use-cases. First is that the user starts the application in the debug mode and then advances the execution step by step, pausing at each step to inspect the state. Second is that the user sets a breakpoint, execution stops when the breakpoint is hit, the user inspects the state of the system and then resumes the application from the point it was stopped at. A typical debugging session is a mix of the two.

Central to both of these use-cases is the capability of the debugger to pause the execution of the program at any point. In a program executing at the single site, this is trivial but in a distributed system, the problem of exerting control over the entire distributed application is difficult. During the time interval when the debugger first issues a stop command to the point when the message reaches a node, the node will already have progressed through some state changes. Besides, the stop command might be dropped by the network or the stop command might reach different nodes at different times.

A solution to this problem is to have a centralized design where every operation at each node is routed through the debugger. Every operation is permitted to happen only when the debugger gives permission. Execution across the entire system can be stopped at once if the debugger stops granting permissions. GoTcha uses this approach.

The disadvantage is that the concurrency of the system has been sacrificed for control. A concurrent system has been transformed into a sequential system. So, a debugger should provide a user with options to reorder operations so that the user can simulate concurrency by exploring different orderings of concurrent operations. There should also be options for aborting networked operations so that the user can simulate network failures.

3.1.3 Observe History of Execution

When a breakpoint is hit, traditional interactive debuggers pause the execution and show the state at that point. They do not display the execution path i.e. the sequence of operations and corresponding state changes which led to that state. Providing this information is superfluous in a single-threaded context as the path can be extrapolated by the user. The path is implicit in the program code. The debugger displays the line of code where the breakpoint hit and the state of the system at that point. The path can be mapped by *going backwards* through the program code from that line of code.

Extrapolating the execution path in a distributed system is not trivial. Let us consider a simple example. Suppose that a program fails with a divide-by-zero exception. It is evident that the divisor is being set to zero at some point in execution. The point of a debugging exercise would be to locate the origin of the error i.e. to find the point where the variable is being set to zero. In a single-threaded context, it would be sufficient for the developer to look at the code to work that out. But, in a distributed system, an outside event could have been responsible for setting the variable to zero. The node might have received a message from another node to decrement the value of the divisor. It would be non-trivial for a developer to figure this out solely from the program code. If the underlying distributed system model abstracts away the reconciliation operation and accepts updates in the background, it would be difficult to figure out when the message was received, or what were the contents of the message, or how the message was processed.

So, in a distributed system, a debugger needs to provide the user with a condensed form of the history of execution - the sequence of operations and the corresponding state changes which led to the state which triggered the breakpoint.

3.2 The Role of the Distributed System Model in Design

Traditional single-machine interactive debuggers can be applied to any single-threaded program due to the fact that all single threaded programs share essentially the same sequential style of execution. However, to build a generalized interactive debugger applicable for any distributed system is difficult, if not impractical due to the wide variety in distributed programming styles.

To build an interactive debugger applicable for a specific distributed system model is more feasible. Distributed system models differ widely from the perspective of interactive debugging. Some models provide more support to an interactive debugger than others. In the earlier section, we have seen that the communication and state reconciliation mechanisms used by the model play a heavy role in determining the design of an interactive debugger.

In this section, we identify certain features of distributed system models which are important from the point of view of an interactive debugger. If a distributed system model has one or more of these features, designing an interactive debugger for the system becomes less complex. The presence of these features can reduce the engineering effort involved in building an interactive debugger considerably.

3.2.1 Features Which Support Interactive Debugging

Read Stability

Read stability is a property of the model where changes to the local state can only occur when the local execution context wants it to. A model achieves read stability when it does not accept remote updates immediately but stores them in some form first and accepts them later with an explicit mechanism.

Read Stability is important from the point of view of interactive debugging because it enables the debugger to easily distinguish the reconciliation operation from local state changes. As discussed in the earlier section, the debugger must be able to expose the three types of state changes as separate events. While debugging an application, when a user steps through a line of code, the user assumes that the corresponding state change is caused by that line of code. But in a model which immediately accepts remote updates e.g. Last-write-Wins, when the line of code is being executed, this assumption is violated as a reconciliation operation could be happening simultaneously in the background. A remote update might arrive and get merged with the local state. In this case, what the user will observe at the end is the aggregated state change caused by both the local execution and the reconciliation operation. But if remote updates are not immediately applied to the local state but applied later with an explicit directive, the two state changes are kept isolated and the debugger can display each state change separately to the user.

Publish Mechanism

In some models, an explicit mechanism is provided to a node to publish it's updates to the outside world. Other nodes are not allowed to observe a node's local state unless the node explicitly publishes it. This is useful because the debugger can avoid tracking each local update in case of an explicit publish mechanism. Local state changes can be tracked from one publish invocation to another instead of over each line of code because during this interval, no other node can view the changes. This makes the observable set of operations smaller and the implementation of the debugger simpler.

Shared State

Traditional interactive debuggers for single-threaded programs show the entire state to the user at each step. The user can inspect the value of any variable or any object attribute. Traditional debuggers are able to do this easily because the OS provides hooks which they can use to access the address space of any process. In case of a distributed system, to do this would be challenging due to absence of any single controller of the system as explained earlier. More importantly, to observe the state of all the variables across multiple nodes would overwhelm a user with the sheer amount of information. So, from the point of view of interactive debugging, it makes sense to identify a subset of state which is important to the user and track the changes in this subset. This simplifies the implementation significantly and it allows the user to focus on the important information. Identification of the subset of state which is important to the user would depend on the semantics and domain of the application.

For a particular class of distributed systems, it is possible to identify the subset of state which is of interest to the user without understanding the domain of the particular application. In shared state models or variants of shared state models like replicated types, all the nodes operate either on a common state or on local copies of a common state. The shared state is set of static types which is declared beforehand. For example, in multi-player online games, all the player nodes share the same game universe - cars, obstacles et cetera. For a set of clients reading and writing key-value pairs from a key-value data store, the shared state is the key-value store. In these cases, the debugger can reduce its focus to the shared state only as it is reasonable to assume that a user would be primarily interested in tracking changes in the shared state. For other models, to understand exactly which objects or variables would be of interest to a developer and thus, should be tracked by the debugger would require understanding the semantics of the specific application.

History

As discussed in section 3.1.3, an important requirement for an interactive debugger for a distributed system is to be able to display some form of the history of operations - the sequence of operations and the corresponding state changes to the user. This allows the user to infer causal relationships between events and helps the user trace the origin of a bug or defect. In order to do this, the debugger would need to log the operations and state changes that happen in the system. Some distributed systems themselves maintain history of state changes in some form. For example, there are some distributed system frameworks which model distributed system synchronization as a problem of version control(TARDIS [9], Irmin [12], GOT [3]). These systems create a new version for each update and keep older versions(until a point) in order to support sophisticated conflict resolution functions like three-way merge. In these cases, the implementation of the debugger is simplified as the debugger can directly leverage the history maintained by framework and display it to the user.

3.2.2 Design Examples

We have identified certain characteristics of distributed systems which are important from the point of view of interactive debugging. It would be instructive to analyse some popular distributed system models in the light of these characteristics (Table 3.1). A very rough estimate of the complexity of designing an interactive debugger for a distributed system model can be formed by tracking the number of features present in the model. The presence of all of the features would indicate that the model is eminently suitable from the point of view of interactive debugging while the absence of all of the features would mean the opposite.

Actor model [4] is a very commonly used model in concurrent/distributed programming where independent nodes(actors) communicate by sending messages to each other. Incoming messages are stored in a queue and processed one at a time. So, there is an explicit mechanism for accepting remote updates which is getting the message from the queue. This allows an interactive debugger to detect reconciliation by tracking the queue pop() operations. Actor model implementations also generally have an explicit mechanism for publishing changes i.e. a send primitive but this is not specified by the model. On the downside, actor model is a message-passing system and does not have a concept of shared state. So, a relevant subset of state would need to be identified and nodes would need to send their states in a serialized form to the debugger with each interaction. Actor models also do not maintain any history of the system. The debugger would need to log the messages and the states itself.

For CRDTs [22], the shared state is the replicated data structure e.g. a counter or a set. However, most CRDT implementations do not have an explicit accept or publish mechanism as updates to other replicas are automatically propagated in the background. So, to design an interactive debugger, we would need to understand the mechanisms used by the underlying CRDT implementation to send and receive messages, and establish hooks into these mechanisms to capture transfer in state and reconciliation of state. CRDTs have

Table 3.1: Properties in different systems

Model	**Read Stability**	**Publish Mechanism**	**Shared State**	**History**
Actor Model	Yes	Yes	No	No
State-based CRDT	No	No	Yes	No
Operation-based CRDT	No	No	Yes	No
GSP	Yes	Yes	Yes	Yes
TARDiS	Yes	Yes	Yes	Yes
GOT	Yes	Yes	Yes	Yes

implicit conflict resolution so a debugger needs to only track the receipt of changes and not the merger of the local state with the received changes. This merger is deterministic and happens according to well-defined rules for each CRDT structure. State-based CRDTs send out the entire state with each update while operation-based CRDTs transmit smaller update operations. An interactive debugger would be able to observe states by simply observing the messages in a state-based CRDT. In a operation-based CRDT, the debugger would need to find a different way to observe states in the system.

GSP [7, 16] is a replicated shared state model which uses Reliable Total Order Broadcast [10] to agree on a global order of update operations for the entire system. GSP provides an explicit mechanism to accept changes (*pull*). All the incoming updates are stored in a buffer called the *receivebuffer* and then applied with the *pull* directive. There is also an explicit directive to publish changes(*push*). Each node maintains a prefix of the global order of update operations called the global update sequence. This can be considered to be a form of *History* even if it does not capture the changes in the global update sequence.

Along with being shared state systems and having explicit mechanisms to accept and publish changes and, TARDiS [9] and GOT [3] maintain history of state changes in the form of version graphs.

Chapter 4

The GoT Distributed Computing Model

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 a specific distributed programming model called Global Object Tracker (GoT) [3]. Specifically, GoTcha is built as a debugger for a Python implementation of GoT called Spacetime. In this section, we describe GoT, and explain its design features that are relevant to GoTcha.

4.1 GoT: Git, but for Objects

A Spacetime 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 state of the shared 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, with a solution that is modeled after Git [3].

All GoT nodes that are part of the same spacetime application have a repository of the shared objects, called a dataframe. The dataframe is akin to a Git repository, and, like a Git repository, it has two components: a snapshot and a version history, as shown in Figure 4.2. 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 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. While in Git conflicts are resolved manually by the user, and only on a *fetch*, 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 are shown Table 4.1; this table can be used as a quick reference for the API calls in the example explained next.

4.2 GoT Example: Distributed Word Frequency Counter

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

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.

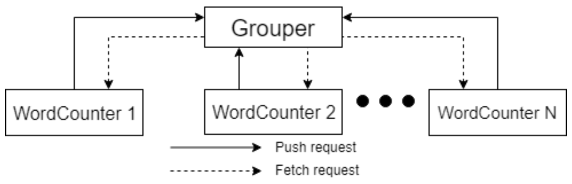


Figure 4.1: Network topology of an example distributed word counting application built on Spacetime.

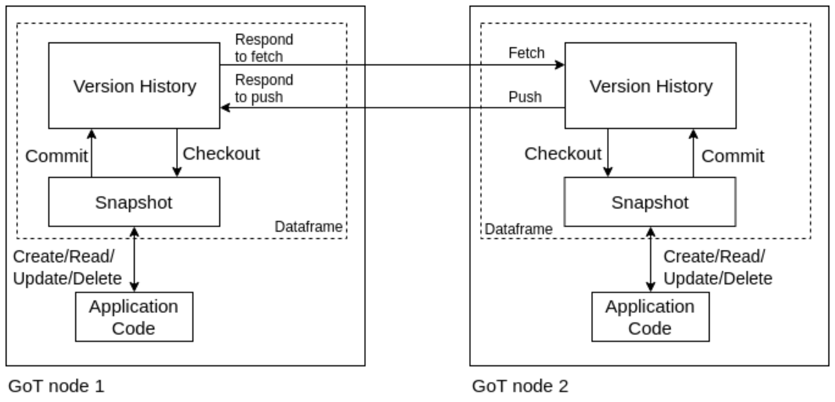


Figure 4.2: Structure of a GoT node. Arrows denote the direction of data flow.

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 4.1 and defines the network topology of our example application.

The dataframes at each WordCounter and Grouper nodes share objects of type Line, WordCount, and Stop that are shown in Listing 4.1.

```
class Line(object):
    line_num = primarykey(int)
    line = dimension(str)
    def __init__(self,line_num,line):
        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
```

Listing 4.1: The types used by the Word Counting application.

All types registered to the dataframe have list of attributes marked as dimensions of the type, that declare the attributes to be stored and tracked in the dataframe. One dimension in each type is a special attribute and is defined as the primarykey. Objects are stored and retrieved by Spacetime by the value of this primarykey attribute.

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 4.2 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.

```
 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)
```

```

if __name__ == "__main__":
    filename, ncount = sys.argv[1:]
    grouper_node = Node(
        Grouper, server_port=8000
        Types=[Line, WordCount, Stop])
    grouper_node.start(filename, ncount)
```

Listing 4.2: The Grouper node.

Listing 4.3 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 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.

```python
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()
```

Listing 4.3: The Word Counter node.

4.3 Dataframe: Object Repository

To the code in each of the GoT nodes, the dataframe acts an object heap that consists of in-memory objects that are under version control. As explained above, the dataframe in each node has two components: a snapshot and an version history.

The version history is stored as a directed acyclic graph where each vertex of the graph represents one version of the state and has a globally unique identifier that labels it and each directed edge of the graph represents a causal "happened-after" relation. Each edge is associated with a delta of state changes (diff) that when applied to the state of the objects at the source version transform it to the state of the objects at the destination version. The latest version of the node state (called the HEAD) is the state of the node that is observable to the other nodes in the application. Changes that are staged in the snapshot cannot be observed by external nodes until they are put into the version history.

An edge with an associated delta is added into the graph for each new version of the state and, therefore, memory usage of the application can potentially be high. To manage the memory, Spacetime implements an effective garbage collector in each node that cleans up obsolete versions.

Changes that are made to the objects in the application code, are staged in the snapshot. When the *commit* primitive is invoked, a new version is created in the version history, representing the newly committed state. An edge is added from the last version the snapshot was in, to the newly created version in the version history. The diff that was committed is then associated with this edge. Changes to the version history, updated by external nodes, are introduced to the snapshot via the *checkout* primitive.

Inter-node communication only happens between the version histories of the corresponding nodes. As seen in the example and like Git, nodes communicate changes between version histories using the *fetch* and *push* primitives present in the dataframe. *Fetch* retrieves changes published to the version history in a remote GoT node and applies the changes to the local version history. *Push* takes changes published to the local version history and delivers it to a remote GoT node. GoT takes advantage of the diffs stored in the version histories to communicate via delta encoding, reducing the amount of data transferred between nodes.

4.4 Conflict Detection and Resolution

Conflicts are detected (at any node that is receiving data), when an update received is a change from a version that is not the HEAD version in the local version history. Intuitively, this means that at least two different nodes committed concurrent changes after having read the exact same version. When conflicts are detected, they are resolved using a user defined three way merge function that includes the state that was read (the original), the changes already in the version history (yours) and the conflicting changes that are incoming (theirs). The output of the merge function, much like the merge resolution in git, adds a new merged version to the version history that has a happened-after relation with both the diverging versions.

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.

```
def merge(conf_iter, orig_df,
          your_df, their_df):
    your_df.update_not_conflicting(
        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  ...
```

Listing 4.4: Merge function used at the Grouper node.

An example merge function is shown in Listing 4.4. 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, and we'll correct it later.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 5.5 at the end of Section 6.

4.5 GoT: Enabling an Interactive Debugger

In Section 3.2, we had identified certain characteristics of distributed system models which make building an interactive debugger simpler - explicit primitives to accept and publish changes, shared state and history. We look at how GoT incorporates these features 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 *fetch* primitives are invoked. These are invoked by the application code at the node, and not automatically behind the scenes. So, a GOT node has read stabilty as explicit primitives have been provided by the model to accept remote updates.

Publish Mechanism: GoT also provides an explicit primitive to publish local changes - *commit*. The version history is the published state which any other node can observe by using *fetch* primitive. The version history is refreshed with the latest local updates by invoking the *commit* primitive.

Shared State: GOT is a shared state model. A GOT application defines a set of static types in the beginning and computation revolves around synchronizing objects of these types across multiple nodes.

History: Each GOT node maintains version history which captures the evolution of state. This can be exposed to the user as History.

Chapter 5

GoTcha: Introduction and Usage

As discussed in the previous section, the version histories at GOT nodes are used for inter-node communication. Remote updates are stored in the version history and then applied to the local state. But a version history is more than the published state. A version history encapsulates the *evolution* of state as well. Each node in the version history captures a previous state of the node with the HEAD of the graph containing the current state while the edges in the version history indicate causal links(*happened-after relationships*)between states. This is useful information from the perspective of debugging and a debugger can use these version histories to display state changes to the user. This is what GoTcha [2] does.

GoTcha is an interactive debugger for GOT-based distributed systems which exposes the changes made to the version history at each node in a Spacetime application. We present GoTcha's key functionalities, interface and usage in this section through a series of examples. As discussed in section 3.1.1, there are three types of state changes in a distributed system. Similarly, errors in a distributed system can be classified according to their origin - **errors in local execution**, **errors in communication** and **errors during reconciliation of remote changes with the local state**. However, there is an additional category - **error**

```
foo
bar
bar
baz
bar
bar
bar
bar
```
Listing (5.1) Input file.

```
foo 1
bar 6
baz 1
```
Listing (5.2) Expected output.

```
foo 1
bar 10
baz 1
```
Listing (5.3) Observed output.

in the network topology of the system itself. For each category, we provide an example and show how GoTcha can be used to detect the source of the error. We also provide an example to show GoTcha can be used for **exploratory testing**. We start our discussion with errors in reconciliation as these bugs are particularly hard to detect and this serves as an effective example for the purpose of demonstrating GoTcha's capabilities.

5.1 Observing State and History

We will use the distributed word-counting application introduced in the previous section, but with a bug in the merge function. A test input file is created consisting of six lines, each with one word – see Listing 5.1. The word frequencies for the words foo, bar, and baz are one, six, 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 4.2, 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 4.2). WordCounter-0 reads, tokenizes, and counts words on lines 1, 3, and 5. WordCounter-1 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 5.2. However, a different output is observed, shown in Listing 5.3. The observed output is wrong, and GoTcha can be used to trace the origin of the error.

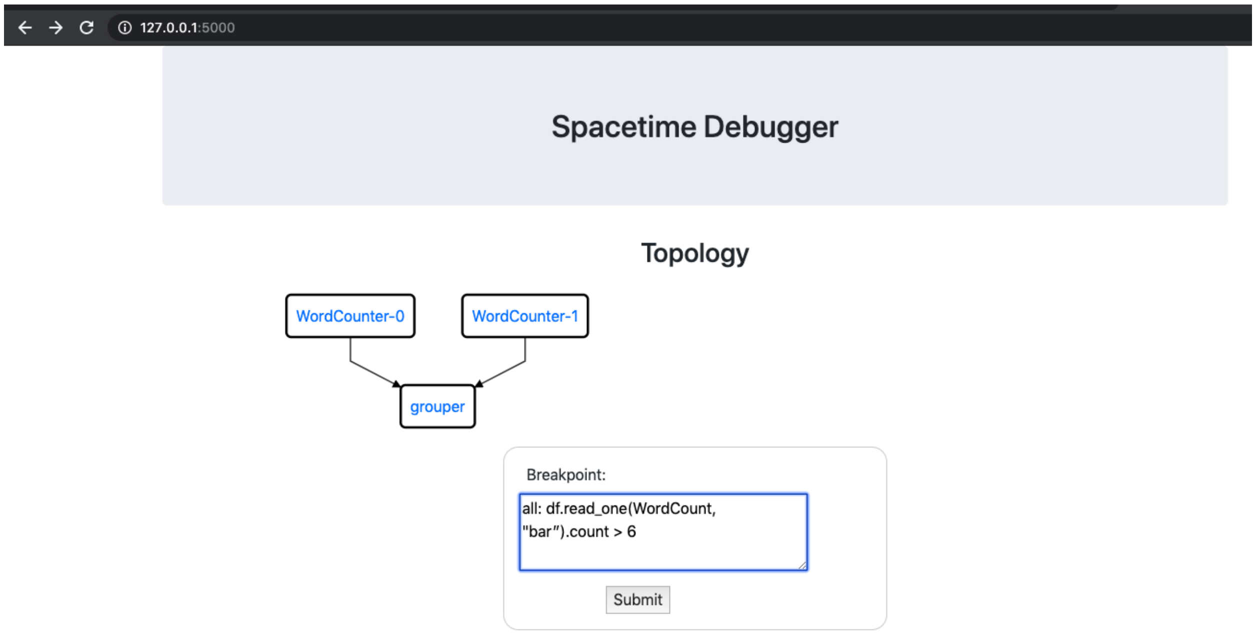
To start debugging the application using GotCha, the debugger needs to be instantiated and started first – see Listing 5.4. The debugger is an application by itself and is launched before any application nodes are launched. The shared application types need to be passed as a parameter to the start command. For the word counter applications, these are Line, WordCount and Stop. Then the application nodes are launched in the debug mode - they are instantiated with an extra flag called *Debug* set to the network address of the debugger, in order to register them with the debugger.

```
debugger_server = GotchaDebugger(port=...)
debugger_server.start_async([Line, WordCount, Stop])

 grouper_node = Node(grouper, server_port=...,
 Types=[Line, WordCount, Stop],
 debug=(... , ...))
```

Listing 5.4: Invoking the debugger.

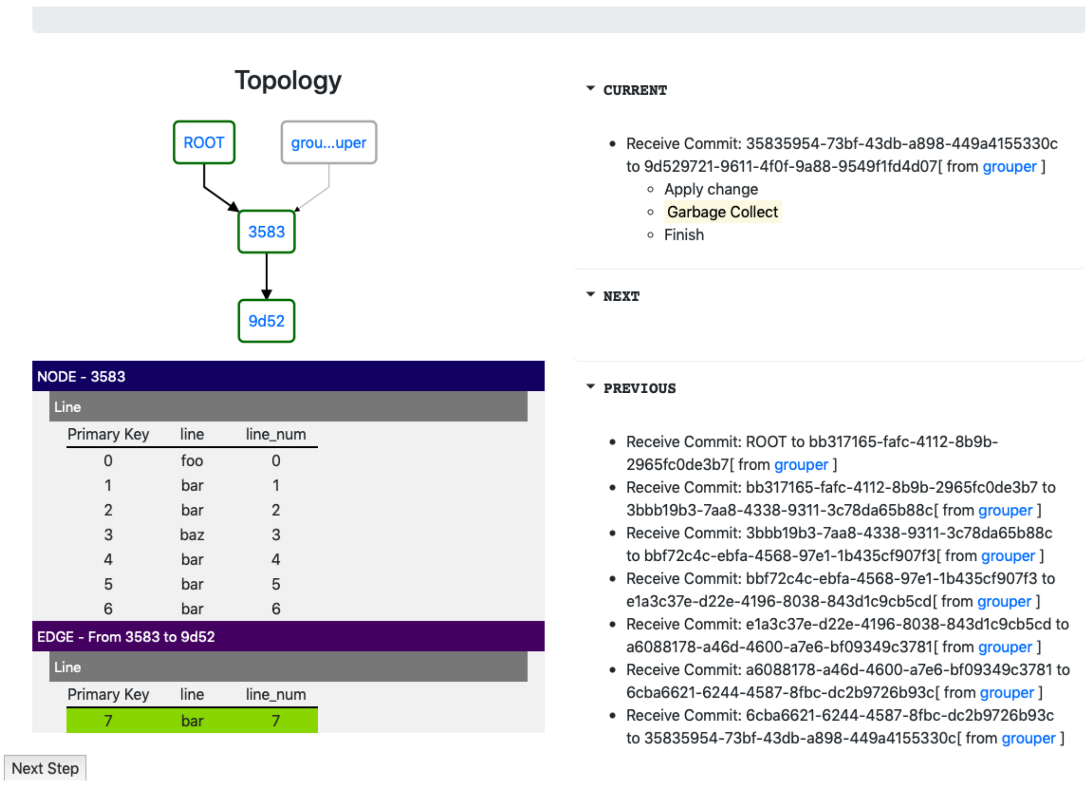
5.1.1 Network Topology Page

GoTcha's UI is a web application. 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 5.2. In this figure, on the left, the user sees that two WordCounter nodes and one Grouper node are being controlled by the debugger. 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 bottom, 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 primary key *bar* with the count dimension greater than six, in the dataframe, at any GoT node.



```
all: df.read_one(WordCount,
"bar").count > 6
```

Figure 5.2: Debugger showing the network topology of the application.



NODE - 3583

Line

Primary Key	line	line_num
0	foo	0
1	bar	1
2	bar	2
3	baz	3
4	bar	4
5	bar	5
6	bar	6

EDGE - From 3583 to 9d52

Line

Primary Key	line	line_num
7	bar	7

CURRENT

- Receive Commit: 35835954-73bf-43db-a898-449a4155330c to 9d529721-9611-4f0f-9a88-9549f1fd4d07[from grouper]
  - Apply change
  - Garbage Collect
  - Finish

NEXT

PREVIOUS

- Receive Commit: ROOT to bb317165-fafc-4112-8b9b-2965fc0de3b7[from grouper]
- Receive Commit: bb317165-fafc-4112-8b9b-2965fc0de3b7 to 3bbb19b3-7aa8-4338-9311-3c78da65b88c[from grouper]
- Receive Commit: 3bbb19b3-7aa8-4338-9311-3c78da65b88c to bbf72c4c-ebfa-4568-97e1-1b435cf907f3[from grouper]
- Receive Commit: bbf72c4c-ebfa-4568-97e1-1b435cf907f3 to e1a3c37e-d22e-4196-8038-843d1c9cb5cd[from grouper]
- Receive Commit: e1a3c37e-d22e-4196-8038-843d1c9cb5cd to a6088178-a46d-4600-a7e6-bf09349c3781[from grouper]
- Receive Commit: a6088178-a46d-4600-a7e6-bf09349c3781 to 6cba6621-6244-4587-8fbc-dc2b9726b93c[from grouper]
- Receive Commit: 6cba6621-6244-4587-8fbc-dc2b9726b93c to 35835954-73bf-43db-a898-449a4155330c[from grouper]

Figure 5.3: Debugger view showing version history at the end of a *commit*.

5.1.2 Version History Page

The current version history of any node can be observed by clicking on the node in the topology graph in Figure 5.2. Figure 5.3 shows the node view of the Grouper node, observing the result of the execution of the *commit* primitive at line 5 of Listing 4.2. The version history is shown on the top left. The history shows three versions: ROOT, 3583, and 9d52. 9d52 happened-after 3583 which in turn happened-after ROOT (the start version of every version history).

Clicking on a version shows the state of objects at that version in tabular form. The table shows that the state at version 75ac has six objects of type Line, and shows the values of the two dimensions (line_num, line) for the Line objects.

Similarly, clicking on a edge shows the delta change (diff) associated with that edge. The diff associated with the edge 3583 → 9d52 is also shown on the bottom. In this case, the diff consists of a single object of type Line with the dimensions line_num, and line having the values 7, 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 4.2). Modifications are uncolored, and deleted objects are marked in red. The state of every version, and the diff associated with every edge can be observed. The grey line relation shows us that the state of the snapshot of the Grouper node is known to be at version 3583.

On the right of Figure 5.3, we see the state of the actions being executed on the dataframe at the Grouper node. The user sees the current active step being executed(CURRENT), a list of steps that have to be executed next(NEXT)as well as a list of previous steps that have been executed on the version history(PREVIOUS). 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 current *HEAD* version (3583) to the newly created version (9d52). 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 3583) are cleaned up.

Clicking on Next Step button at the bottom would allow the garbage collect phase of *commit* at the Grouper node to be executed.

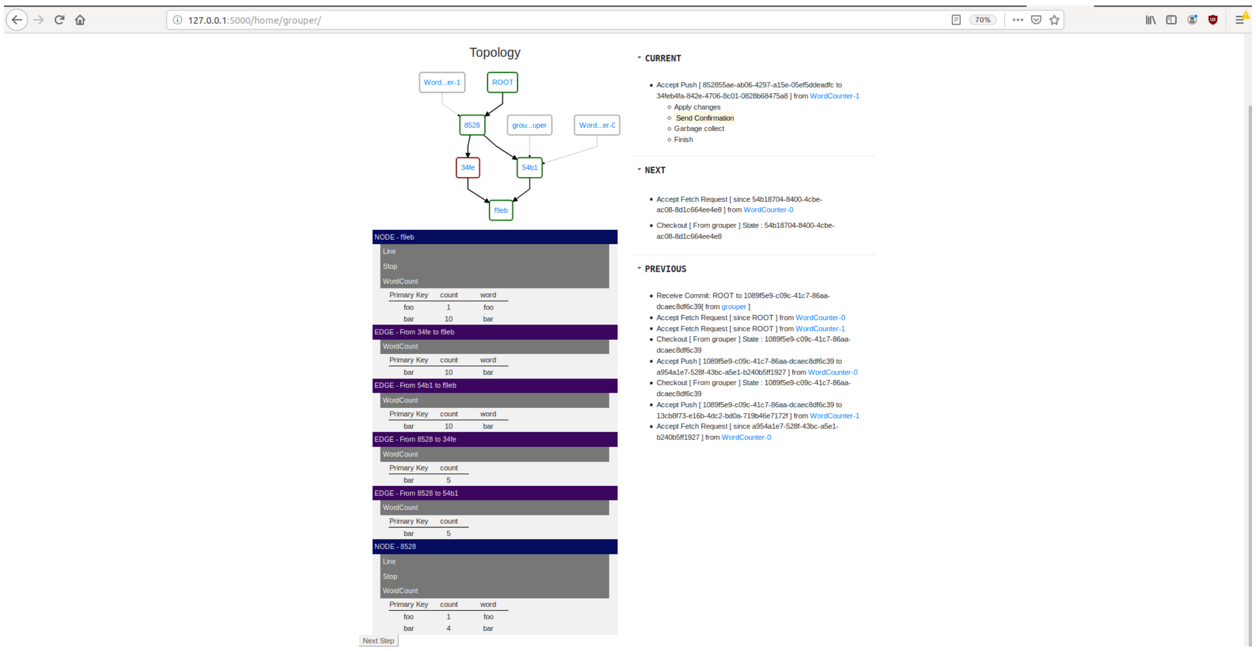
Since the *fetch*, and *push* primitives of the dataframe span across multiple GoT Nodes, they are broken up into two sets of operations each: *fetch* and *respond to fetch*, *push*, and *respond to push*, to observe the state changes at both the node making the request and the node receiving the request.

5.2 Observing Reconciliation

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 5.2, and hit the Submit button. We put in this specific condition because the correct count for *bar* is six and we want to observe the state of the system at the exact point when the count becomes incorrect i.e. greater than six.

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 5.4. Here, we see that the current step being executed is a *push* request from WordCounter2 from version 8528 to version 34fe. The execution is paused at the start of the send confirmation phase.

The version history contains seven versions. Starting from the top, we have ROOT again



NODE - f9eb

Line

Stop

WordCount

Primary Key	count	word
foo	1	foo
bar	10	bar

EDGE - From 34fe to f9eb

WordCount

Primary Key	count	word
bar	10	bar

EDGE - From 54b1 to f9eb

WordCount

Primary Key	count	word
bar	10	bar

EDGE - From 8528 to 34fe

WordCount

Primary Key	count
bar	5

EDGE - From 8528 to 54b1

WordCount

Primary Key	count
bar	5

NODE - 8528

Line

Stop

WordCount

Primary Key	count	word
foo	1	foo
bar	4	bar

CURRENT

- Accept Push [852855ae-ab06-4297-a15e-05ef5ddeadfc to 34feb4fa-842e-4706-8c01-0828b68475a8] from WordCounter-1
  - Apply changes
  - Send Confirmation
  - Garbage collect
  - Finish

NEXT

- Accept Fetch Request [since 54b18704-8400-4cbe-ac08-8d1c664ee4e8] from WordCounter-0
- Checkout [From grouper] State : 54b18704-8400-4cbe-ac08-8d1c664ee4e8

PREVIOUS

- Receive Commit: ROOT to 1089f5e9-c09c-41c7-86aa-dcaec8df6c39[from grouper]
- Accept Fetch Request [since ROOT] from WordCounter-0
- Accept Fetch Request [since ROOT] from WordCounter-1
- Checkout [From grouper] State : 1089f5e9-c09c-41c7-86aa-dcaec8df6c39
- Accept Push [1089f5e9-c09c-41c7-86aa-dcaec8df6c39 to a954a1e7-528f-43bc-a5e1-b240b5ff1927] from WordCounter-0
- Checkout [From grouper] State : 1089f5e9-c09c-41c7-86aa-dcaec8df6c39
- Accept Push [1089f5e9-c09c-41c7-86aa-dcaec8df6c39 to 13cb8f73-e16b-4dc2-bd0a-719b46e7172f] from WordCounter-1
- Accept Fetch Request [since a954a1e7-528f-43bc-a5e1-b240b5ff1927] from WordCounter-0

Figure 5.4: Debugger view at Grouper showing response to a *push* request.

as the start version. Version 8528 happened-after ROOT. All versions that were present between 8528 and ROOT, have been garbage collected.

At 154b, we see a fork in the path. Both versions 34fe and 54b1 happened-after 8528, and are siblings. These are concurrent updates and were performed on different GoT nodes. Version 54b1 is bordered in green while 34fe is bordered in red. This means that update 8528 $\rightarrow$ 34fe was received by the version history at Grouper after the update 8528 $\rightarrow$ 54b1. The GoT node resolved such conflicts using the custom merge function written in Listing 4.4. The output of the merge function was a new version f9eb. Since f9eb happened-after both the concurrent versions, 54b1 and 34fe, the graph was updated with the happened-after relations and f9eb has two in-edges. Additionally, each of these edges are associated with a diff that transforms the previous version to the version at f9eb. f9eb is the 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 54b1. Additionally, the last known versions of WordCounter1 and WordCounter2 are 8528, and 54b1, respectively.

We can click on any node of the graph to get the exact state at that version. Similarly, we can click on any edge to get the state change associated with the edge. The state at version f9eb is shown in the table below the version history graph. The table shows us two WordCount objects and the WordCount object for the word 'bar' has a count of ten, showing us why the conditional breakpoint was hit. Looking at the version at the start of the merge, 8528, we see that the count of 'bar' is four. The diffs associated with 8528 → 34fe and 8528 → 54b1 both update the count of 'bar' to five. This means that both WordCounter0 and WordCounter1 had the count of 'bar' as four, and observed a 'bar' token updating the count concurrently to five. At the end of the merge function, this count is set to ten, and can be see in the diffs for both 34fe → f9eb and 54b1 → f9eb. This means that the error is in the merge function. We can see that the merge function in Listing 4.4, on detecting a conflicting count, simply adds up the counts. So receiving two counts of five, would result in a count of ten. 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 5.5 and the word counting application gives the right output.

```
def merge(conf_iter,orig_df,
               your_df,their_df):
    your_df.update_not_conflicting(
         their_df)
    for orig,yours,theirs in conf_iter:
         assert isinstance(
              yours,WordCounter)
         yours.count += theirs.count
         if orig: # False if new objects.
              yours.count -= orig.count
    return your_df
```

Listing 5.5: Merge function used at the Grouper node.

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.

5.3 Observing Communication

Inter-node communication is typically a source of problems in distributed applications: the contents of the message may be incorrect, the message may get dropped or delayed by the network, or the sending of the message itself may never have been initiated. In order to illustrate the debugging of these types of bugs, we will use a very simple producer-consumer application.

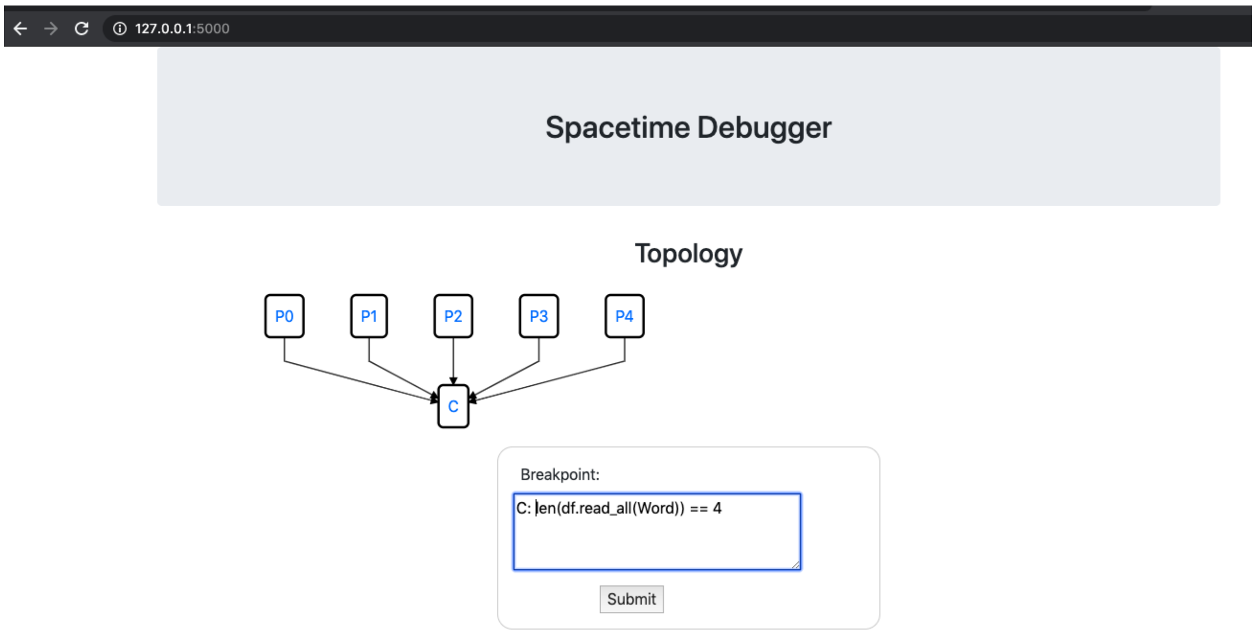
5.3.1 Example: Distributed Producer-Consumer

There are multiple *producers* in the application, each of which generating a random string, and a single *consumer*, which *consumes* the generated strings. The *consumer* prints out a received string to the screen if it is a string it has not seen before. Listing 5.6 and Listing 5.7 show the application code for a producer node and the consumer node respectively.

```
def producer(df):
    df.add_one(Word, Word(randomString()))
    df.commit()
    df.push()
```

Listing 5.6: A Producer.

```
def consumer(df) :
    existingWords = set()
    while True:
        df.checkout()
        newWords = df.read_all(Word)
        for word in newWords:
```



C: len(df.read_all(Word)) == 4

Figure 5.5: Debugger view showing the network topology for the producer-consumer application

```
                if word not in existingWords:
                    existingWords.add(word)
                    # print the word if it is a new word
                    print(word)
```

Listing 5.7: The Consumer.

```
def buggyProducer(df):
    df.add_one(Word, Word(randomString()))
    df.commit()
```

Listing 5.8: The buggy Producer.

There are five producers, each in a different machine. The network topology of the producer-consumer application is shown in Fig 5.5 The fifth producer has a bug in its application code. It generates a random string but does not send it, as shown in listing 5.8

Due to this bug, the consumer will only display four strings when the expected output is five since there are five producers.

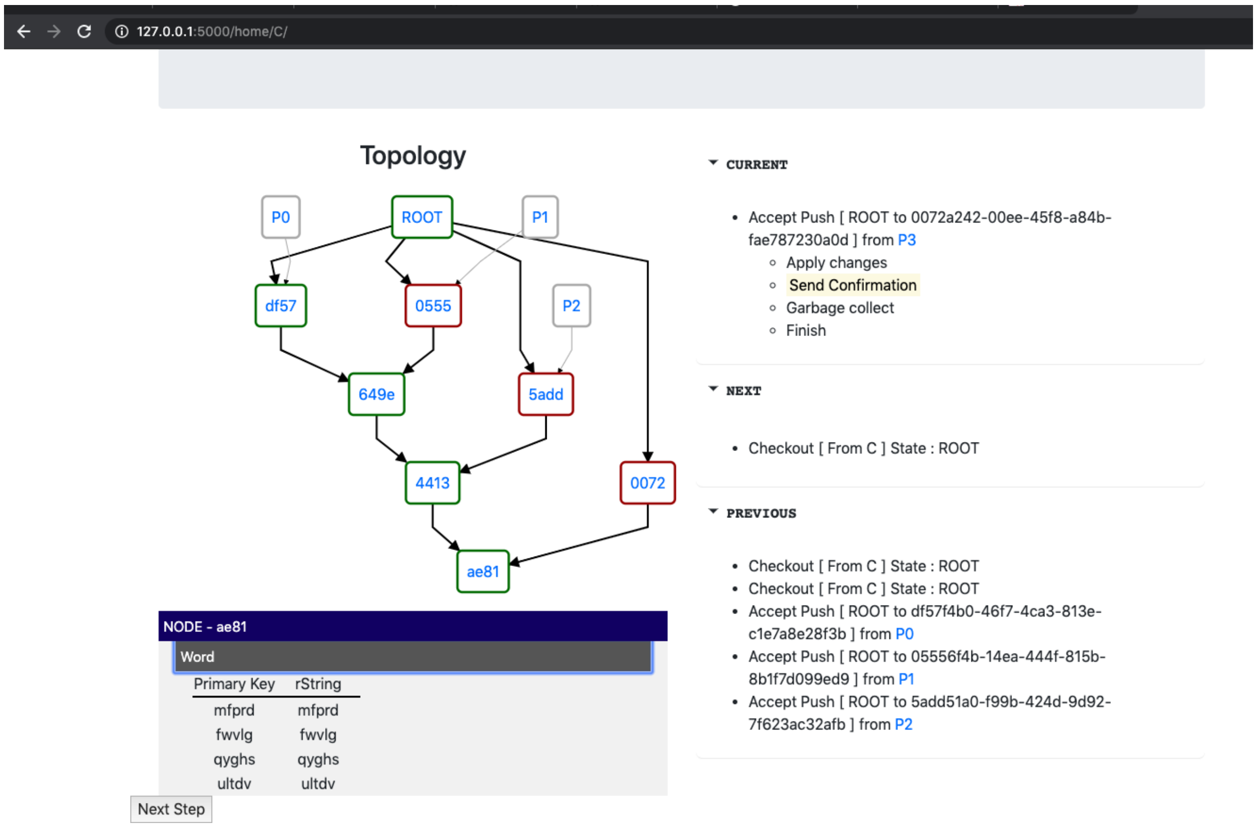
5.3.2 Detecting The Bug

To debug, we put in the condition for breakpoint as shown in Fig 5.5 and hit the Submit button. We put a breakpoint on the number of words in the consumer's(C)dataframe to the effect that the execution should pause when the number of words becomes four. The expected number of words the consumer should display is five but the actual output only displays four words. So, we want to observe the state of the system when the consumer has already received the four strings. We can then advance the execution step by step to understand why the consumer is not able to get the last string.

When the breakpoint is matched, GoTcha shows the view of the consumer node 5.6. Looking at the CURRENT section, we note that the consumer is paused in the middle of processing a Push request from producer P3 from version ROOT to 0072. We observe there are four branches from the ROOT indicating that four concurrent updates have been received by the consumer. We click on the node ae81 which is the HEAD of the graph to view the current state of the consumer to confirm that the consumer has received four strings. Looking at the PREVIOUS section in the bottom right of the page, we observe that Push requests have already been received from each of the producers P0-2.

We click the next step button to complete the remaining stages of the Push from P3 to get the state as shown in 5.7.

From Fig 5.7, we notice that P4 is not in the graph which means that no request has been from received from P4 yet. We also note that no update from P4 is pending in the NEXT section. We conclude that P4 has an issue and go to the version history page for P4 – Fig 5.8 by clicking on the node for P4 in the network topology page. We see that P4 has generated its random string but that no Push request is pending in the NEXT section. Also, we can see from the PREVIOUS section that P4 has not attempted any Push requests in the past. Hence, we can conclude that the issue is that P4 is not sending its updates to the consumer.



CURRENT

- Accept Push [ROOT to 0072a242-00ee-45f8-a84b-fae787230a0d] from P3
    - Apply changes
    - Send Confirmation
    - Garbage collect
    - Finish

NEXT

- Checkout [From C] State : ROOT

PREVIOUS

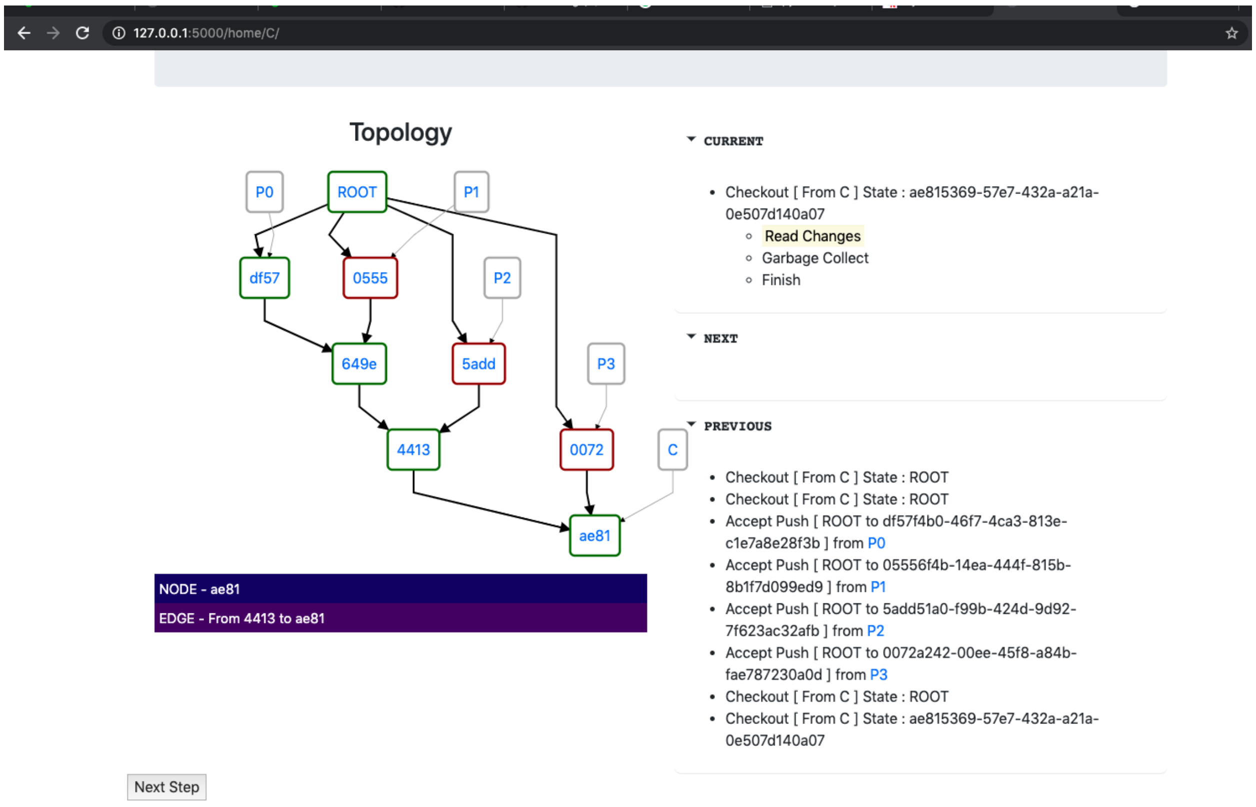
- Checkout [From C] State : ROOT
- Checkout [From C] State : ROOT
- Accept Push [ROOT to df57f4b0-46f7-4ca3-813e-c1e7a8e28f3b] from P0
- Accept Push [ROOT to 05556f4b-14ea-444f-815b-8b1f7d099ed9] from P1
- Accept Push [ROOT to 5add51a0-f99b-424d-9d92-7f623ac32afb] from P2

NODE - ae81

Word

Primary Key	rString
mfprd	mfprd
fwvlg	fwvlg
qyghs	qyghs
ultdv	ultdv

Figure 5.6: Debugger view showing the view at the consumer node in the middle of Push from P3



NODE - ae81

EDGE - From 4413 to ae81

CURRENT

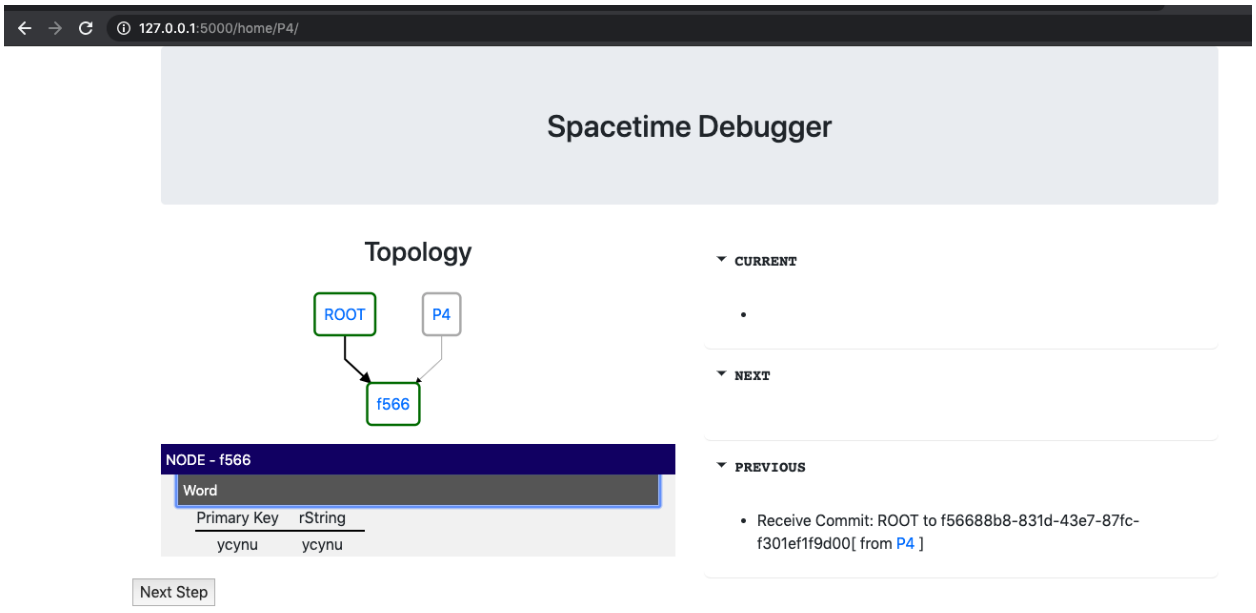
- Checkout [From C] State : ae815369-57e7-432a-a21a-0e507d140a07
  - Read Changes
  - Garbage Collect
  - Finish

NEXT

PREVIOUS

- Checkout [From C] State : ROOT
- Checkout [From C] State : ROOT
- Accept Push [ROOT to df57f4b0-46f7-4ca3-813e-c1e7a8e28f3b] from P0
- Accept Push [ROOT to 05556f4b-14ea-444f-815b-8b1f7d099ed9] from P1
- Accept Push [ROOT to 5add51a0-f99b-424d-9d92-7f623ac32afb] from P2
- Accept Push [ROOT to 0072a242-00ee-45f8-a84b-fae787230a0d] from P3
- Checkout [From C] State : ROOT
- Checkout [From C] State : ae815369-57e7-432a-a21a-0e507d140a07

Figure 5.7: Debugger view showing the view at the consumer node after the Push from P3 is complete



Primary Key	rString
ycynu	ycynu

- Receive Commit: ROOT to f56688b8-831d-43e7-87fc-f301ef1f9d00[from P4]

Figure 5.8: Debugger view showing the view at the Producer node P5

5.4 Observing Local Execution

Until now, we have looked at how GoTcha can be used to detect errors during communication and reconciliation. That is, we have seen how GoTcha is able to expose two out of the three types of state changes discussed in section 3.1.1 - transfer of state and reconciliation of remote state with local state to the user. The third type of change in state which should be exposed to the user is the change in state due to local execution, which GoTcha does, to a certain extent. While GoTcha does not let the user observe change in state over each line of code, it lets the user observe local state change from one dataframe operation to next with the guarantee that the state change across this interval is solely due to local execution. This information is useful for debugging as the user can locate the node or even the section of the application code in the node which is problematic, and use a traditional sequential debugger to locate the exact line of code which is the source of the error. So, GoTcha can be used in conjunction with a traditional sequential debugger to detect errors due to local execution.

5.4.1 Scenario

We will illustrate this through another example. Going back to our distributed word-counter example, see section 4.2, we will introduce a bug in line 16 of the word-counter (Listing 4.3) . The line is changed as shown below:

```
word_obj.count += 2
```

In other words, a Word Counter node, upon finding a word, increments the count of the word by 2 instead of 1. On running the application, the counts are doubled and we get an incorrect output as shown in 5.9. The expected counts are shown in 5.3.

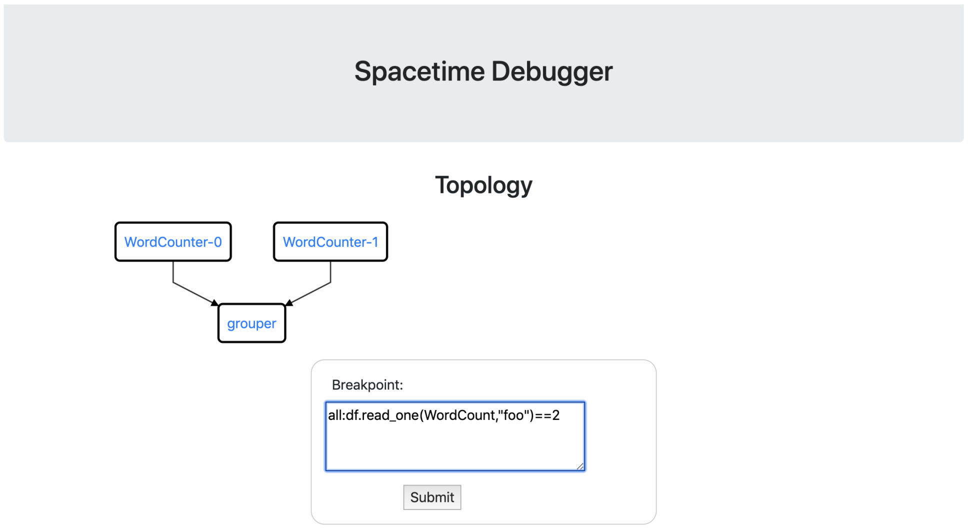
5.4.2 Detecting The Bug

To debug, we launch GoTcha and put in a breakpoint as shown in 5.9. We put a breakpoint on the word *foo* such that the breakpoint is hit when the count of *foo* becomes two because we know that this count is incorrect. We would want to observe the precise state of the system when the count becomes two and the history of operations till that point. When the breakpoint is hit, we are shown the state at the WordCounter0 node– see Fig 5.10. We observe that the WordCounter0 is paused in the middle of a commit operation. The *Garbage Collect* stage is highlighted in yellow which means that the WordCounter0 is about to start the *Garbage Collect* stage and that it has completed the *Apply Changes* stage of the commit operation.

We click on the HEAD of the graph-afd1 to check the current state of the WordCounter0

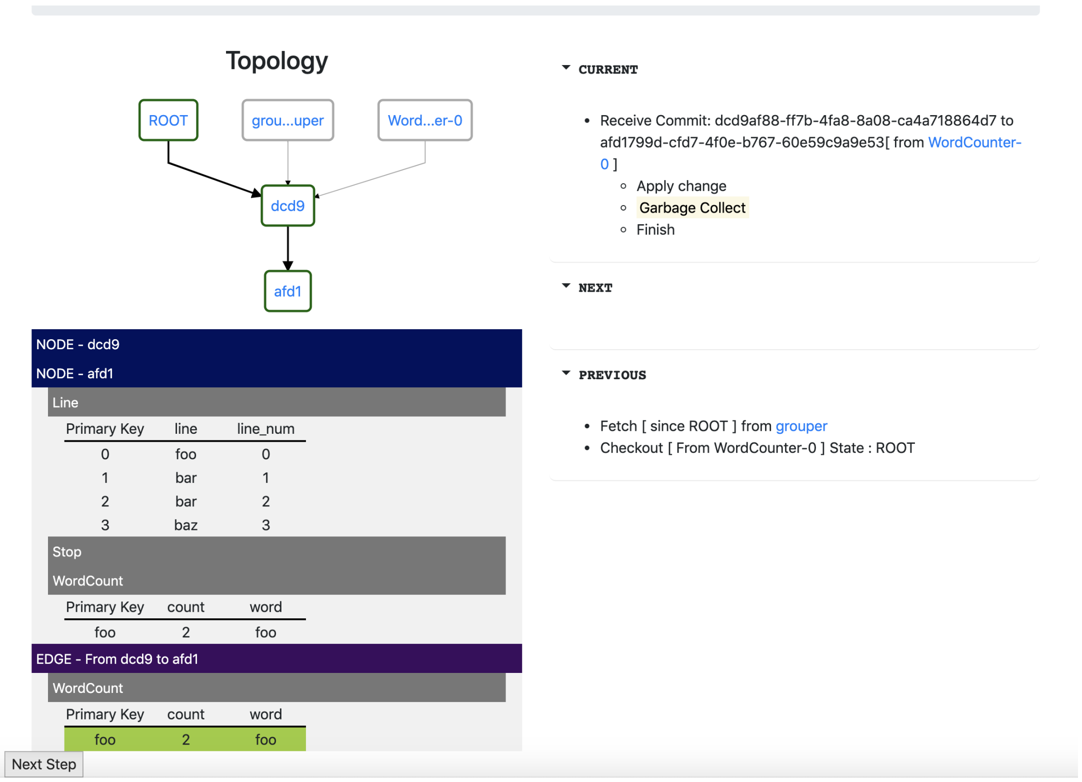
```
foo 2
bar 12
baz 2
```

Listing 5.9: Incorrect output.



```
all:df.read_one(WordCount,"foo")==2
```

Figure 5.9: Debugger view showing breakpoint on the count of foo



- Receive Commit: dcd9af88-ff7b-4fa8-8a08-ca4a718864d7 to afd1799d-cfd7-4f0e-b767-60e59c9a9e53[from WordCounter-0]
  - Apply change
  - Garbage Collect
  - Finish

- Fetch [since ROOT] from grouper
- Checkout [From WordCounter-0] State : ROOT

NODE - dcd9

NODE - afd1

Line

Primary Key	line	line_num
0	foo	0
1	bar	1
2	bar	2
3	baz	3

Stop

WordCount

Primary Key	count	word
foo	2	foo

EDGE - From dcd9 to afd1

WordCount

Primary Key	count	word
foo	2	foo

Figure 5.10: Debugger view showing the state of the Word Counter application when breakpoint is met.

node and see that the WordCounter0 node does have the word *foo* and that it's count is two. But we also notice that out of the four line objects that WordCounter0 has, only one line contains the word *foo*. This finding is significant. If there is only one line with a single occurrence of *foo*, the count of *foo* should not be two.

To find the source of the discrepancy, we click on the edge dcd9 $\rightarrow$ afd1 to check whether the delta state change represented by this edge includes the change in count of *foo*. Our intuition is confirmed when we see that the table for the edge dcd9 $\rightarrow$ afd1 contains *foo* with the count 2. This means that the transition from version dcd9 to afd1 caused the value of *foo* to be modified to 2. We note that the current commit operation itself is responsible for this transition as the CURRENT section in the top right of the page shows the *from* version and the *to* version of the commit operation- dcd9 and afd1 respectively. Since a commit is a local operation where the changes made by the local code are saved to the version history, we can conclude the source of the bug is inside the application code of the WordCounter node between a commit and a checkout operation. We can now use a normal debugger to debug the application code of WordCounter line by line and pinpoint the bug to line 16 of the WordCounter code Listing 4.3.

5.5 Observing Network Topology

GOT enforces a variant of the server-client model of communication in which multiple servers are allowed instead of a single server. Each node (*client*) pushes and pulls changes from its *server* node. Each *client* when initialized needs to explicitly declare its *server* so that a communication channel can be established between them. GoTcha exposes the network topology of the system to the user. When a system is started in debug mode, GoTcha first shows the nodes in the system and the communication links between them. This makes it easy for a user to detect bugs in the network topology of the system itself.

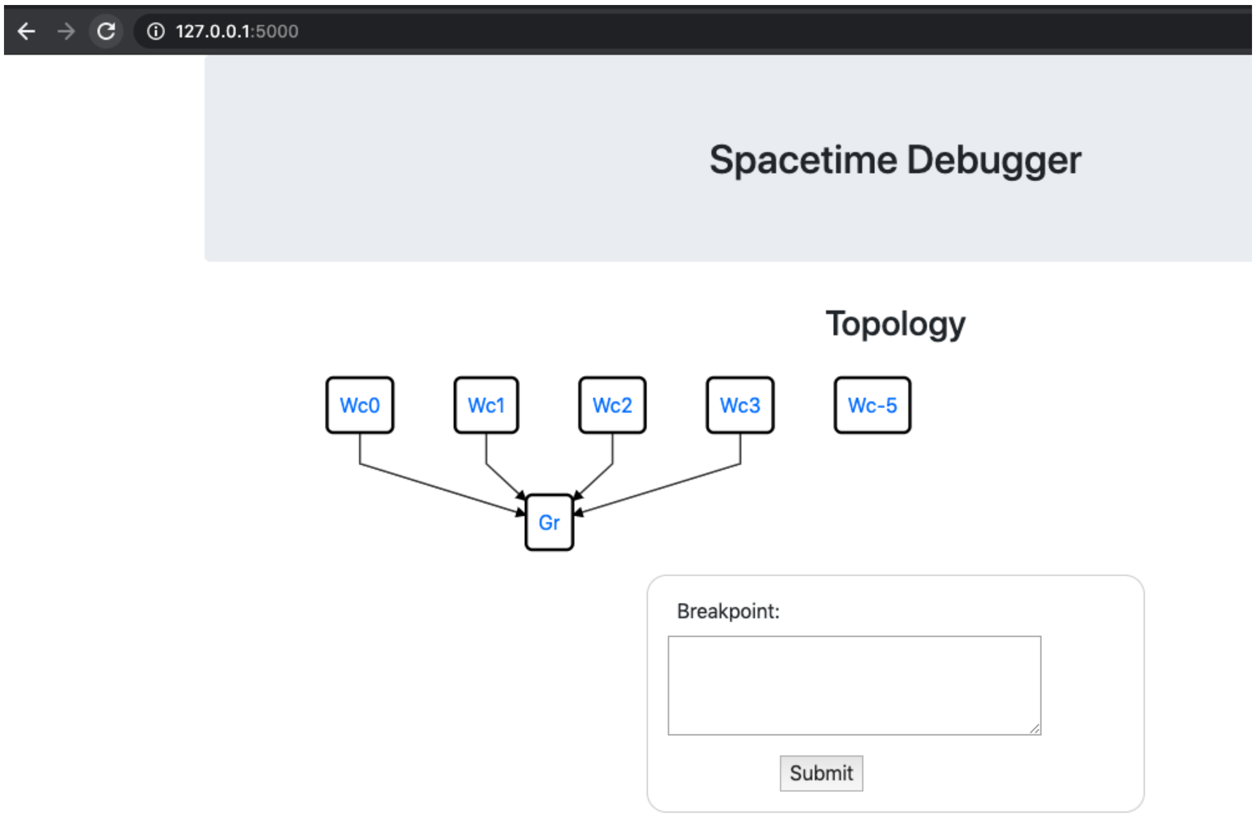


Figure 5.11: Debugger showing a fault in the network topology

Continuing the distributed word-counter example, a Grouper node is initialized and then five WordCounter nodes are initialized, each in a different machine. The WordCounter nodes are initialized as shown in listing 5.10. The *server* paratmeter is set to the IP address and port of the Grouper node. This is critical as this establishes the communication link between the WordCounter node and the Grouper node. If a WordCounter node is initialized without the *server* parameter as shown in listing 5.11, that would be an error in the program as the WordCounter node is now disconnected from the Grouper. GoTcha enables a user to detect this error easily as it displays the network topology of the entire application in its homepage, see Fig 5.11. A WordCounter node (*Wc-5*) is not connected to the Grouper node (*Gr*).

```
wcNode = Node(
            WordCounter,
            Types=[Line, WordCount, Stop],
            server =(ServerIP, ServerPort),
            )
```

Listing 5.10: Correct WordCounter Initialization.

```
wcNode = Node(
            WordCounter,
            Types=[Line, WordCount, Stop],
            )
```

Listing 5.11: Incorrect WordCounter Initialization.

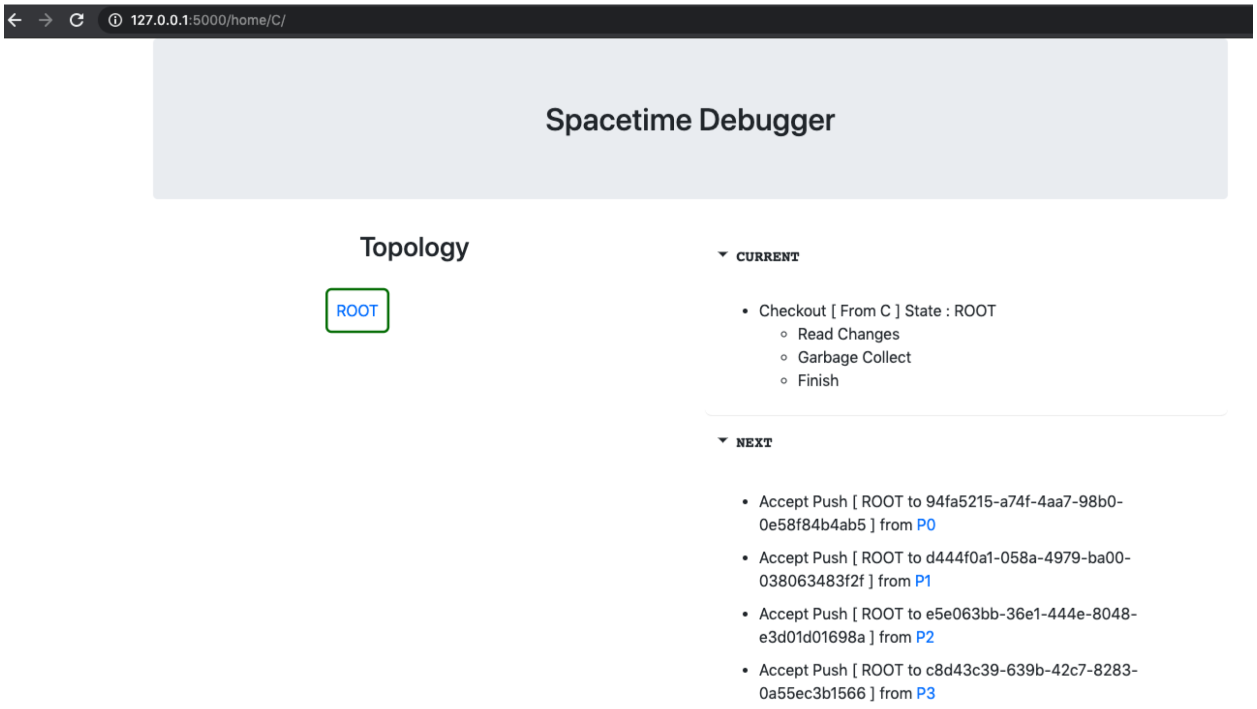
5.6 Exploratory Testing

An interactive debugger by its very definition interferes with the natural execution of a system. This is a well-understood phenomenon and is called the probe effect [20]. So, any interactive debugger, especially for distributed systems, should provide the user with tools to overcome the probe effect by allowing the user to easily explore multiple *runs* of the system. To that effect, GoTcha enables exploratory testing - it lets the user simulate the concurrency and non-determinism of a distributed system by letting the user reorder the pending operations at a node. This also enables a user to check the correctness of the system under edge cases.

5.6.1 Interleaving Operations

We will use the distributed producer-consumer example(section 5.3.1) to illustrate how GoTcha enables exploratory testing. Fig 5.12 shows the version history page at the Consumer node. We see that Consumer node is at the version ROOT, indicating that it has not received any updates yet. Looking at the NEXT section, we see that four *Accept Push* operations from *P0*, *P1*, *P2* and *P3* are waiting to be accepted by the Consumer in that order.

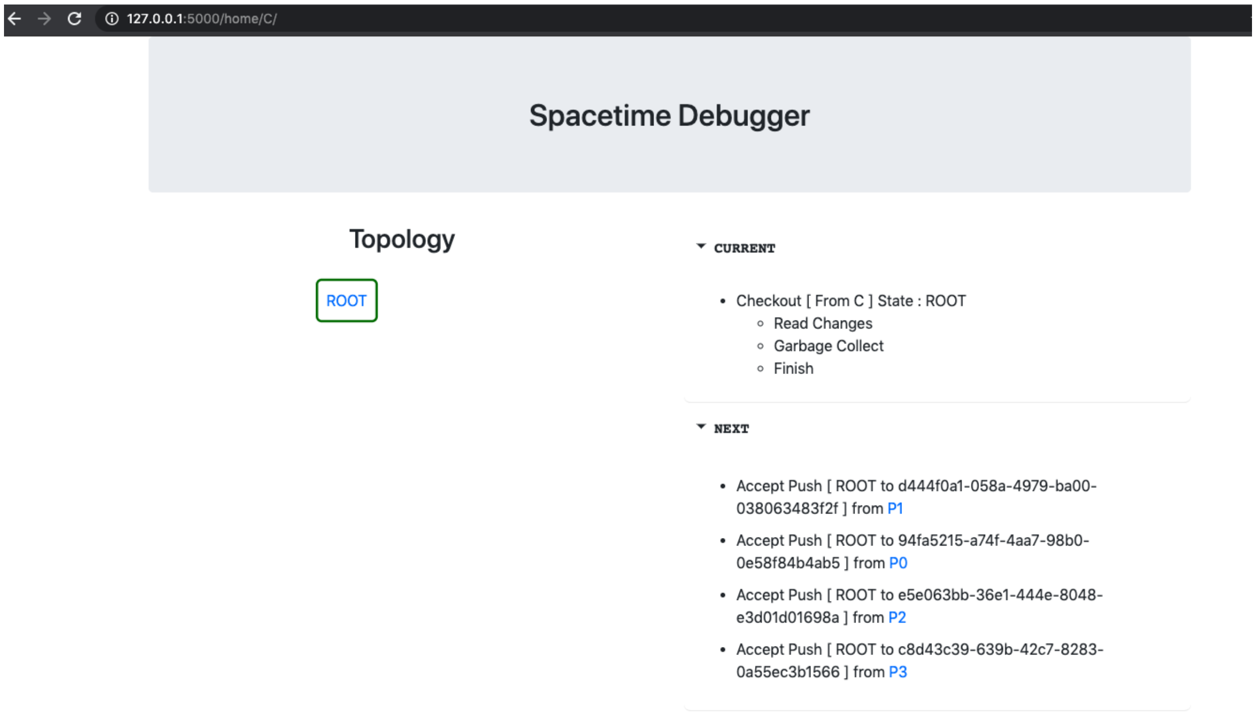
If we keep clicking the *Next Step* button, the updates will get applied in the same order - *P0* -> *P1* -> *P2* -> *P3*. In this case, we will be simulating the scenario where the Consumer gets the message from *P0* first, then *P1*, then *P2*, and finally *P3*. However, that is only one of the possible scenarios in an actual execution. The message from *P0* might be delayed by the network so that the message from *P1* reaches the Consumer first. Any one of the 4! permutations could happen. GoTcha lets the user explore all of these multiple scenarios. The user can drag and drop the pending operations in the NEXT section in order to reorder them. Fig 5.13 shows the version history page after we have dragged and dropped the *Accept Push* operation from P1 to the P0's place at the top. So, we can now explore the scenario where the Consumer receives the update from P1 first i.e. *P1* -> *P0* -> *P2* -> *P3* and continue the execution from there.



- Checkout [From C] State : ROOT
  - Read Changes
  - Garbage Collect
  - Finish

- Accept Push [ROOT to 94fa5215-a74f-4aa7-98b0-0e58f84b4ab5] from P0
- Accept Push [ROOT to d444f0a1-058a-4979-ba00-038063483f2f] from P1
- Accept Push [ROOT to e5e063bb-36e1-444e-8048-e3d01d01698a] from P2
- Accept Push [ROOT to c8d43c39-639b-42c7-8283-0a55ec3b1566] from P3

Figure 5.12: Debugger view showing the Consumer with four pending remote updates



- Checkout [From C] State : ROOT
  - Read Changes
  - Garbage Collect
  - Finish

- Accept Push [ROOT to d444f0a1-058a-4979-ba00-038063483f2f] from P1
- Accept Push [ROOT to 94fa5215-a74f-4aa7-98b0-0e58f84b4ab5] from P0
- Accept Push [ROOT to e5e063bb-36e1-444e-8048-e3d01d01698a] from P2
- Accept Push [ROOT to c8d43c39-639b-42c7-8283-0a55ec3b1566] from P3

Figure 5.13: Debugger view showing the Consumer after reordering the pending remote updates

Chapter 6

GoTcha: Architecture and Implementation

In the previous Chapter, we looked at the usage of GoTcha. In this section, the architecture and implementation of GoTcha is described in detail.

6.1 Centralized Architecture

In section 3.1.2, we saw how a centralized design where all the operations are routed through the debugger can be used to control the flow of execution in a distributed system. GoTcha's design is based on this approach. A variation of this approach is used where each dataframe operation which reads or writes from the version history (*checkout*, *commit*, *fetch*, *push*) is routed through a central node called the Global Controller Node (GCN). With each operation, the version history at each node is also sent to the GCN. The GCN, in turn, forwards it to the web-based UI to be shown to the user. In debug mode, the architecture of the application is modified from what is shown in Figure 4.2 to what is shown in Figure 6.1. While a traditional

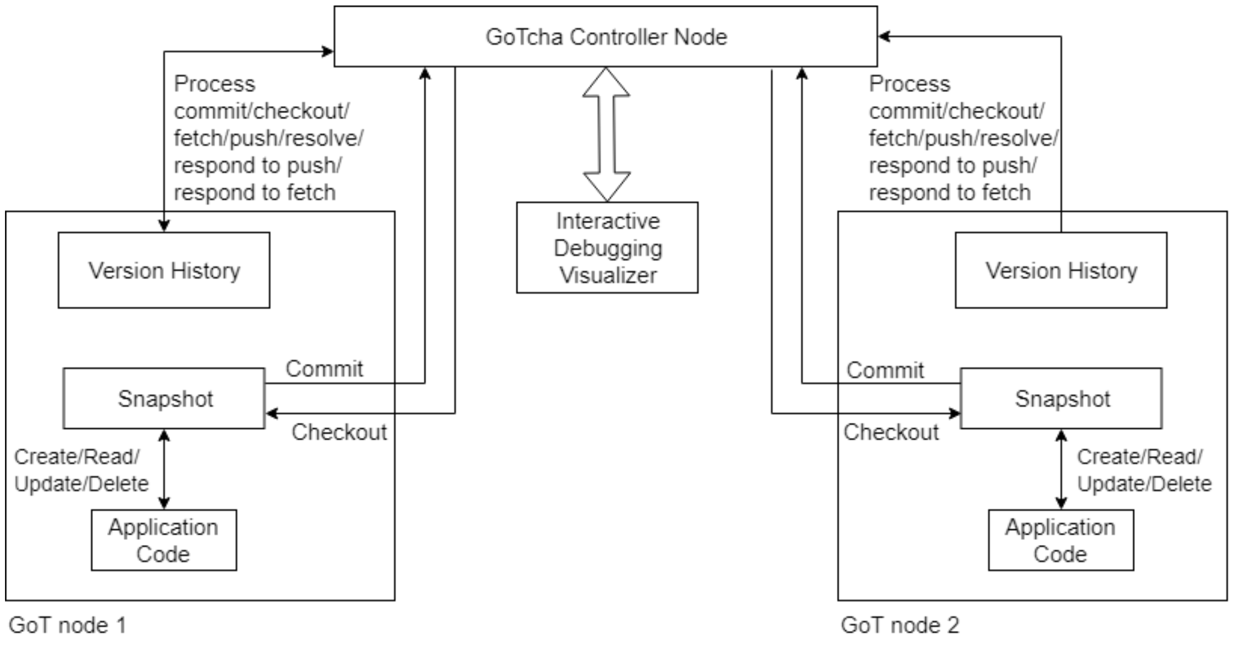

GoTcha Controller Node
Process commit/checkout/ fetch/push/resolve/ respond to push/ respond to fetch
Process commit/checkout/ fetch/push/resolve/ respond to push/ respond to fetch
Interactive Debugging Visualizer
Version History
Version History
Commit
Checkout
Commit
Checkout
Snapshot
Snapshot
Create/Read/ Update/Delete
Create/Read/ Update/Delete
Application Code
Application Code
GoT node 1
GoT node 2

Figure 6.1: Architecture of GoTcha.

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.

6.2 Global Controller Node (GCN)

The rerouting of the dataframe operations through the GCN is the key functionality that enables GoTcha to achieve the goals of an interactive debugger. This is achieved through the GoT programming model itself by using shared GoT objects of special types called the debugger types. Each dataframe operation is mapped to a specific debugger type(see table 6.1). When any dataframe operation is initiated in the debug mode, the application node creates an object of the corresponding debugger type and uses this object to communicate and co-ordinate with the GCN at each stage of the operation. Thus, we re-use the GoT model in GoTcha's design for synchronization between the nodes and the GCN.

Table 6.1: Mapping the dataframe operations to the debugger types

Dataframe Operation	**Debugger Type**
Commit	CommitOp
Checkout	CheckoutOp
Push	PushOp
Fetch	FetchOp
Respond to Push	AcceptPushOp
Respond to Fetch	AcceptFetchOp

The GCN is a GOT node itself. But unlike a regular GOT node which has a single dataframe, GCN maintains a separate dataframe for each application node– see Fig 6.2. Each dataframe acts as the remote repository for the debugger objects (instances of the debugger types) created by a particular application node. The separation of dataframes ensures that there is a separate channel of communication between the GCN and each node. No application node can receive a debugger object created by another node.

The GCN's application code encapsulates two separate threads of execution. The first thread keeps polling for new application nodes. Whenever a new node registers with the debugger, a new dataframe with the debugger types is created in the GCN for this particular node. The second thread runs the main fuctionality of the GCN. It launches the web server which is responsible for getting the user input from UI, and controls the execution of the application as per user input.

6.3 Wrapped Dataframe

In the debug mode, instead of the usual dataframe, the application node receives an instance of a *WrappedDataframe*. *WrappedDataframe* is a wrapper around the dataframe class and has the same API as dataframe so that from the point of view of the node, there is no change. The WrappedDataframe encapsulates two dataframes - the usual *Application Dataframe* which acts as the datastore for the objects declared by the application and another dataframe -*Debugger*

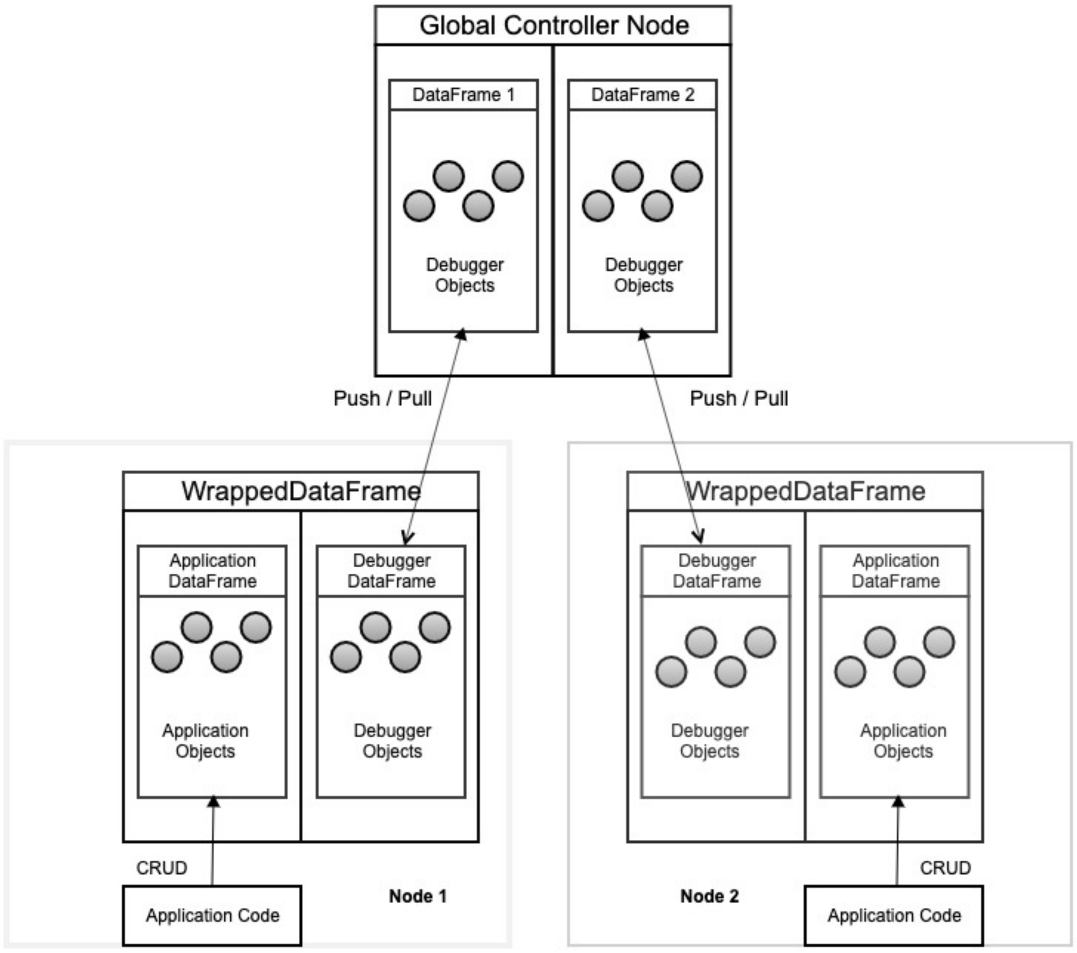


Figure 6.2: GOT Node in Debug Mode

Dataframe which is the store for the debugger objects and which acts as a communication channel to the GCN. So, in the debug mode, a GOT node's architecture gets changed from Fig 4.2 to Fig 6.2. Instead of a *Application Dataframe* which can communicate with the outside world, a GOT node now has a *WrappedDataframe* which contains the *Application Dataframe* as well as a *Debugger Dataframe*. The *Application Dataframe*'s usual communication capabilities are disabled. The *Debugger Dataframe* is the link to the outside world. And even the *Debugger Dataframe* is allowed to only communicate with the GCN. In this way, the topology of the application gets transformed into a true server-client model or a star topology where the GCN is the server and the nodes are the clients, and the clients only communicate with the server.

As the *WrappedDataframe* has direct control over *the Application Dataframe* and thus, the internals of the dataframe operations, *Wrapped Dataframe* is able to pause or resume an operation as per instructions from the GCN which it receives via the debugger object in *Debugger Dataframe*. For example, a commit operation means applying the local changes to the version history. Internally, this happens in two main stages - the dataframe reads the changes from the snapshot and then applies the changes to its version history. In debug mode, the *WrappedDataframe* inserts a *wait* after each stage where it awaits permission from the GCN to advance to the next stage.

Each *WrappedDataframe* runs a separate thread of execution to receive *respond to fetch/respond to push* requests. As mentioned in section 5.1.2, fetch and push operations are divided into two sub-operations - fetch/push at the sender node and respond to fetch/respond to push at the receiver node. There are separate debugger types for these sub-operations– see Table 6.1. When the GCN is overseeing a fetch/push operation, it needs to receive the fetch/push request from the sender and forward the request to the receiver. However, the issue is that the GCN cannot forward the requests directly to the receiver. We have explained earlier how the GCN and the *Debugger Dataframes* in the application nodes use the GoT programming model to coordinate with each other using shared *debugger objects* and that the GoT model enforces a server-client topology. Due to this, the GCN, which is the *server* in this case, cannot forward the request to the receiver node directly. So, each *WrappedDataframe* needs a separate thread to keep pulling changes from the GCN to detect any new respond to push/respond to fetch objects.

6.4 Debugger Types

The definition of *PushOp* - the debugger type for the push operation is shown in listing 6.1. The *state* attribute is used by the the application node and the GCN to coordinate the execution of the operation. The push operation has four stages - Read Changes, Send Changes, Wait for confirmation from receiver and Garbage collect. The GCN instructs the node to start the execution of a specific stage by seting *state* to an appropriate value. After the stage is complete, the node lets the debugger know that that the execution of the stage is complete by setting *state* to the corresponding value .

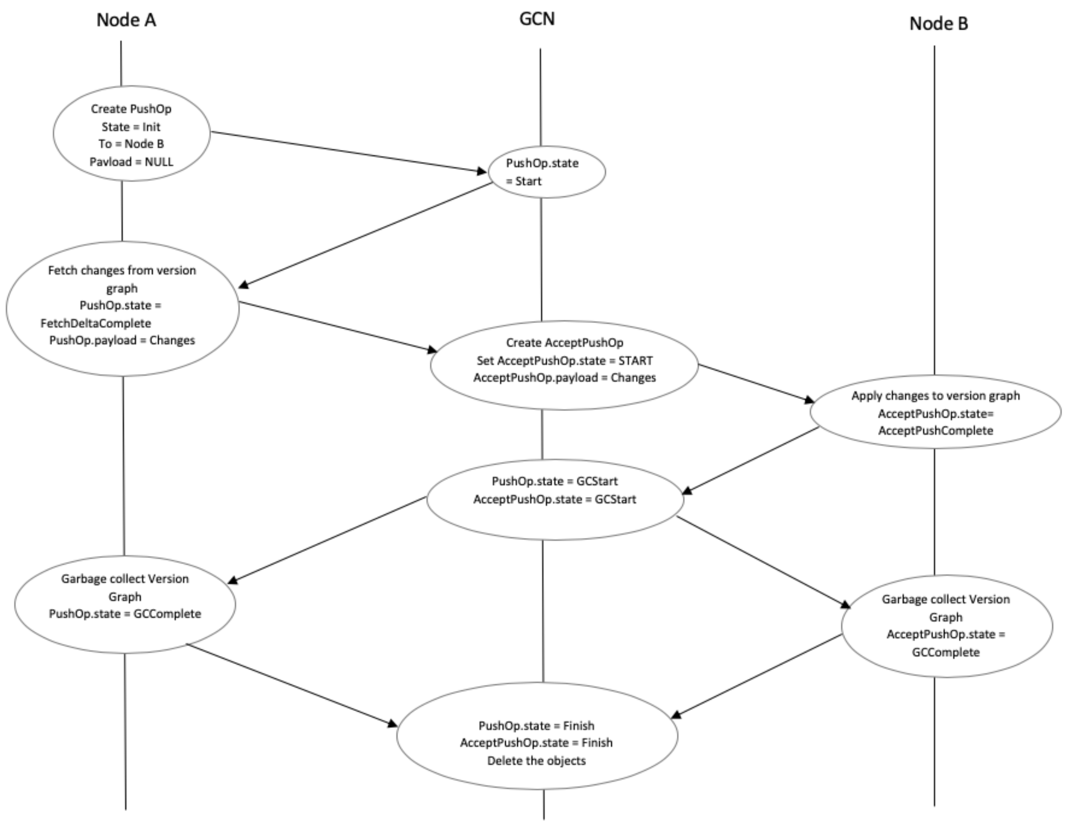
The *node* attribute stores the application name. A push operation implies sending state changes from a particular start version to an end version (which is the head of the version graph at the sender node). The *fromVersion* and *toVersion* store the version IDs for the start version and the end version respectively. The *delta* stores the actual state changes from the *fromVersion* to the *toVersion* in binary format.

```
class PushOp:

    oid = primarykey(str)

    # Logistics
    sender_node = dimension(str)
    receiver_node = dimension(str)
    state = dimension(int)
    # Initial payload
    fromVersion = dimension(str)

    # Response to request
    toVersion = dimension(str)
    delta = dimension(bytes)
```
Listing 6.1: PushObj

6.5 Example: Controlling A Push Operation

To illustrate the interaction between the GCN and application nodes in detail, Fig 6.3 shows the sequence diagram for a *Push* operation in debug mode. When a node A decides to push changes to another node B, it invokes the push API as usual. The *WrappedDataframe* creates an object of the debugger type*PushOp*, sets the relevant attributes, adds it to *Debugger Dataframe*, commits and pushes the changes to the GCN. The *WrappedDataframe* then waits till the GCN provides permission to start the push operation. The GCN in turn keeps checking its dataframes. When GCN *checksout* the dataframe for Node A and detects the *PushOp* object, it sets its state attribute to *Start*. Node A gets the modified object when it next pulls changes from the GCN, checks that *state* is set to *Start* and starts the push operation accordingly. It fetches the changes from its version graph, puts the changes i.e. the state change delta in the *PushOp* object's *payload* attribute. It then commits the updates and pushes the updates to GCN.

Next, the GCN asks node B to accept the changes. The GCN does this by initializing an object of the type *AcceptPushOp*. The GCN gets the state delta from the *payload* of the *PushOp* object, puts the delta in the *payload* of the *AcceptPushOp* object and commits the changes. On detecting the *AcceptPushOp* object, the Node B applies the state delta to its version graph on receiving the *AcceptPushOp* object and sets the *AcceptPushOp* object's *state* to *AcceptPushComplete*.



Node A
GCN
Node B
Create PushOp
State = Init
To = Node B
Payload = NULL
PushOp.state = Start
Fetch changes from version graph
PushOp.state = FetchDeltaComplete
PushOp.payload = Changes
Create AcceptPushOp
Set AcceptPushOp.state = START
AcceptPushOp.payload = Changes
Apply changes to version graph
AcceptPushOp.state= AcceptPushComplete
PushOp.state = GCStart
AcceptPushOp.state = GCStart
Garbage collect Version Graph
PushOp.state = GCComplete
Garbage collect Version Graph
AcceptPushOp.state = GCComplete
PushOp.state = Finish
AcceptPushOp.state = Finish
Delete the objects

Figure 6.3: Sequence Diagram for a Push operation in Debug mode

The GCN then sets the *state* of the both the objects to *GCStart* which is a signal to the nodes to start garbage collection on their respective version graphs. When garbage collect is complete, the nodes set the status of the objects to *GCComplete* which the GCN takes to mean that the operation is complete and that the *PushOp* and *AcceptPushOp* objects can be deleted from their respective dataframes.

6.6 Modes of Operation

GoTcha has two modes of operation - **interactive** mode and **free-run** mode. We have seen both the modes in the examples in the previous section. In the interactive mode, the user executes the application step by step, pausing at each step to inspect the state. The *Next Step* button in the version history page of the GoTcha UI lets the user run GoTcha in the interactive mode. In the free-run mode, the user does not control the execution step by step but lets it run freely till the end or till a condition become true. The breakpoint functionality available in the network topology page of the UI enables the user to run GoTcha in the free-run mode.

As discussed in section 6.2, GCN controls the execution of each dataframe operation. In the interactive mode, the GCN executes a step on receiving the *Execute Next Step* instruction from the user, then waits for the next *Execute Next Step* instruction and so on. In this way, the user is able to run the application step-by-step.

In the free-run mode, the GCN executes one step after another without waiting for instruction from the user. If a conditional breakpoint has been set, at each step, the GCN loops through all the dataframes(if the breakpoint specifies *all*)or checks a specific dataframe(if the breakpoint mentions a particular dataframe)and evaluates the condition. If the breakpoint evaluates to true, its pauses the execution.

6.7 Implementation

We describe the implementation of GoTcha briefly here.

UI: GoTcha's UI is written in basic HTML, CSS and Javascript. The dagred3.js [1]library is used for rendering the graphs for network topology and version history.

Backend: GoTcha's backend is currently implemented in Python3. We use the Flask web application framework [2] for a light-weight web server.

[1]https://github.com/dagrejs/dagre-d3
[2]https://palletsprojects.com/p/flask/

6.8 Experiment

To estimate the overhead that GoTcha imposes on a system, below experiments were performed.

Experiment 1: A program where a node generates a certain number of objects and exits, was run, see Listing 6.2. The program was run in two modes - using GoTcha and without GoTcha(normal mode). The number of objects the node generates before stopping was varied across executions. The time taken for the node to finish producing all the objects was noted. The number of objects the node produces is equal to the number of commit operations the program does. The results are shown in Fig 6.4.

Experiment 2: A program where multiple *producers* are producing objects and a *consumer* is consuming the objects was run with GoTcha and without GoTcha,see Listing 6.3. The time taken till the *consumer* receives all the objects was noted. The number of objects was kept fixed across executions. The number of nodes was varied across executions. The results are shown in Fig 6.5.

The experiments were run on a physical machine with a dual-core 3.07 GHz Intel(R) Xeon(R) processor with 125 GB of RAM.

Analysis: From experiment 2, we observe that the GoTcha puts a significant overhead on the normal execution of the program and that this overhead increases steeply with the increase in the number of nodes. This is not surprising since GoTcha has a centralized design and increasing the number of nodes implies increasing the number of concurrent dataframe operations happening in the system. More and more operations get queued up at the GCN which can only process them sequentially. From experiment 1, we observe that the overhead also increases with the increase in the number of objects which is also expected as the increase in the number of objects implies increase in the number of dataframe operations being done

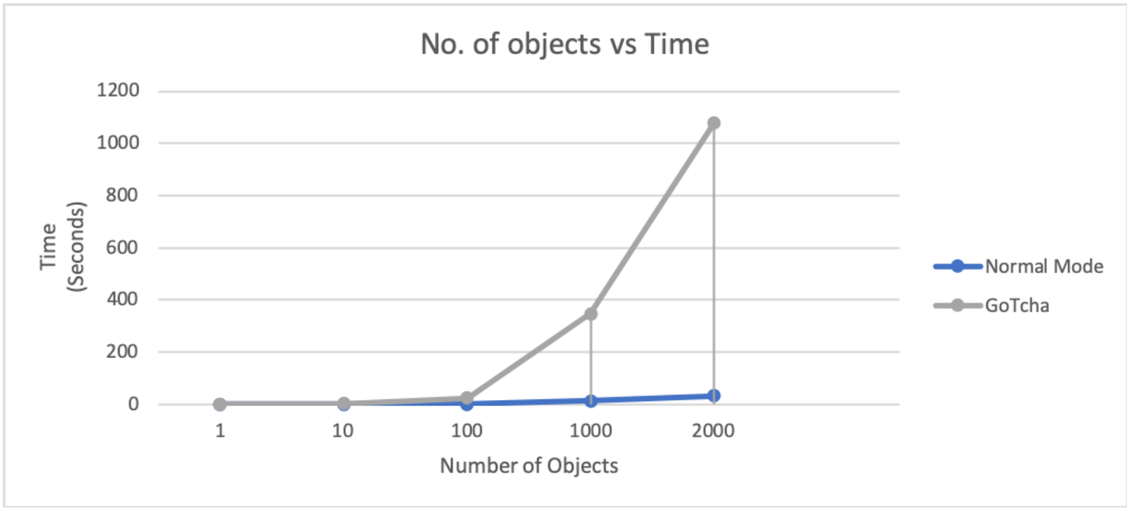


Figure 6.4: Results for experiment 1

in the system.

```
@pcc_set
class Foo:
    i = primarykey(int)

    def __init__(self, i):
        self.i = i

    def __str__(self):
        return "Foo_"+ str(self.i)

def producer(df):
    start = None
    count = 0
    while count < numObjects:
        df.add_one(Foo, Foo(count))
        df.commit()
        count += 1
producer = Node(producer, Types=[Foo])
producer.start_async()
producer.join()
```

Listing 6.2: Experiment 1

```
def producer(df, i, numProducers):
    for i in range(i,numObjects,numProducers):
        df.add_one(Foo, Foo(i))
        df.commit()
        df.push()
```

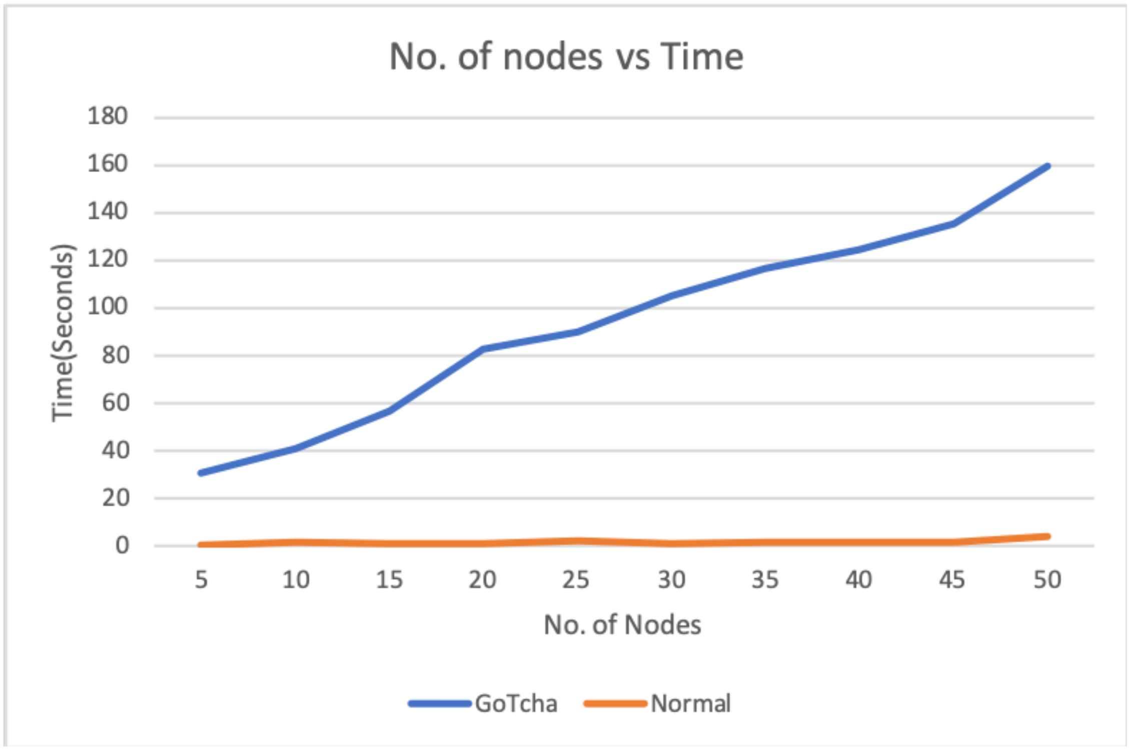


Figure 6.5: Results for experiment 2

```

def consumer(df, start):
    foos = list()
    while True:
        df.checkout()
        foos = df.read_all(Foo)
        if len(foos) >= numObjects:
            break
```

Listing 6.3: Experiment 2

6.9 GoTcha: Achieving the goals

In Section 3, we describe the fundamental goals that an interactive debugger must achieve. 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, and also provide the *history of execution* to the user. In this section, we discuss how GoTcha meets these fundamental

Table 6.2: 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

requirements.

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 6.2 maps the GoT primitives to the type of state change that it facilitates.

Since GoTcha intercepts each of these GOT primitives, it is able to let the user observe all the three types of state changes. We have shown through examples how GoTcha exposes all the three types of state changes– see sections 5.2, 5.3, 5.4.

Controlling the Flow of Execution

GoTcha follows the centralized debugger design explained in Section 6.1. 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, as shown in section 5.6.

Observing History As discussed earlier in section 3.1.3, an interactive debugger needs to provide the *history of execution* to the user. The *history of execution* contains the sequence of operations and the resulting state changes from the beginning of execution till the current state.

GoTcha shows this information to the user through the version history page. All the previous operations that have happened at the node are listed in the PREVIOUS section while the user can (mostly) infer the corresponding state changes from the version history graph. The version history graph represents a condensed form of the *state change history* as the version graph is garbage-collected and obsolete versions are deleted after each operation. So, a version graph might not contain all the state changes that have happened since the beginning of execution till that point. But it does encapsulate valuable information for backtracking from an erroneous state to its origin - a sequence of states from the beginning to the current state, even if some intermediate states are missing.

Chapter 7

Future Work and Conclusion

7.1 Future Work

We would like to explore several avenues of future work. A brief description of each is provided below.

7.1.1 Decentralized Design

GoTcha leverages its centralized design to control the flow of the execution of the system. All the operations are routed through the debugger providing the debugger with complete control over the system. But this becomes a disadvantage in large systems due to two reasons. First, the execution of the application in the debug mode becomes slower as each operation needs to go through the debugger. Time taken to hit the conditional breakpoints also increases, potentially adversely affecting the user experience. Second, it becomes increasingly hard to manually simulate the different *runs* of the system. As mentioned earlier, GoTcha converts a concurrent distributed system into a sequential system in order to be able to control each step of the execution. In order to compensate for the lost concurrency, GoTcha provides the user with tools for reordering operations. We leave it to the user to manually explore all the different execution paths of the system using these tools. But manual exploration of all the different permutations of operations becomes increasingly difficult and tedious as the size of the system increases. In the future, we would like to explore a different trade-off between control and concurrency, and explore a decentralized version of GoTcha where the debugger doesn't control each node directly, and the debugger needs to issue a stop to each node over the network to pause execution.

7.1.2 Rollbacks

We would like to extend GoTcha to support *rollbacks*. This is possible as we have the version history graph and we can go back to a previous version by reverting to the state encapsulated by that previous version. It is important to note that by doing this, state would be rolled back for the version history only, the local execution would not be rolled back.

7.1.3 Graph Visualization

GoTcha's UI in its current version functions well with a limited number of nodes. However, issues arise as the number of nodes in the system increase. The UI finds it hard to render the network topology or the version graph if the system has a large number of nodes. We would like to incorporate advanced graph visualization techniques in order to make the UI scalable.

7.1.4 Breakpoints

The current implementation supports only a rudimentary conditional breakpoint functionality. A user needs to input Python code as breakpoint. This leaves the system open to injection attacks. We would like to remove this vulnerability by implementing input-checking. We would also like to enhance the basic breakpoint functionality by supporting advanced querying and search techniques like regex.

7.2 Conclusion

Debugging distributed systems is difficult - interactive debugging, even more so. In this thesis, I presented GoTcha, an interactive debugger for distributed systems based on the GoT programming model. I discussed the high-level goals of an interactive debugger for distributed systems and explained how GoTcha achieves these goals through a series of usage examples. I described the detailed architecture and implementation of GoTcha, and ended with a discussion on how GoTcha can be extended in the future.

Bibliography

[1] *TotalView*. `https://computing.llnl.gov/tutorials/totalview`.

[2] *Onward! 2019: Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, New York, NY, USA, 2019. Association for Computing Machinery.

[3] R. Achar and C. V. Lopes. Got: Git, but for Objects. *arXiv e-prints*, page arXiv:1904.06584, Apr 2019.

[4] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

[5] I. Beschastnikh, P. Wang, Y. Brun, and M. D. Ernst. Debugging distributed systems. *Commun. ACM*, 59(8):32–37, July 2016.

[6] E. G. Boix, C. Noguera, T. V. Cutsem, W. D. Meuter, and T. D'Hondt. REME-D: a reflective epidemic message-oriented debugger for ambient-oriented applications. In W. C. Chu, W. E. Wong, M. J. Palakal, and C. Hung, editors, *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC), TaiChung, Taiwan, March 21 - 24, 2011*, pages 1275–1281. ACM, 2011.

[7] 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.

[8] W. H. Cheung, J. P. Black, and E. Manning. A framework for distributed debugging. *IEEE Software*, 7(1):106–115, Jan 1990.

[9] 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.

[10] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, Dec. 2004.

[11] 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.

[12] B. Farinier, T. Gazagnaire, and A. Madhavapeddy. Mergeable persistent data structures. In D. Baelde and J. Alglave, editors, *Vingt-sixièmes Journées Francophones des Langages Applicatifs (JFLA 2015)*, Le Val d'Ajol, France, Jan. 2015.

[13] 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.

[14] 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.

[15] D. M. Geels, G. Altekar, S. Shenker, and I. Stoica. *Replay debugging for distributed applications.* PhD thesis, University of California, Berkeley, 2006.

[16] 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.

[17] R. Hood. The p2d2 project: Building a portable distributed debugger. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, SPDT '96, page 127–136, New York, NY, USA, 1996. Association for Computing Machinery.

[18] 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.

[19] 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.

[20] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Comput. Surv.*, 21(4):593–622, Dec. 1989.

[21] R. D. Schiffenbauer. Interactive debugging in a distributed computational. 1981.

[22] 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.

[23] T. Stanley, T. Close, and M. S. Miller. Causeway: a message-oriented distributed debugger. Technical report, HP Labs, 2009. HP Labs tech report HPL-2009-78.

[24] E. Tribou and J. Pedersen. Millipede: A multilevel debugging environment for distributed systems. In H. R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2005, Las Vegas, Nevada, USA, June 27-30, 2005, Volume 1*, pages 187–193. CSREA Press, 2005.

[25] 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.

[26] D. Woos, Z. Tatlock, M. D. Ernst, and T. E. Anderson. A graphical interactive debugger for distributed systems. *CoRR*, abs/1806.05300, 2018.

[27] 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.