

# DisViz: Visualizing real-world distributed system logs with space time diagrams

by

Josiah McMenamy

B.S., Computer Science and Engineering  
Massachusetts Institute of Technology, 2024

Submitted to the Department of Electrical Engineering and Computer Science  
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2025

©2025 Josiah McMenamy. All Rights Reserved

The author hereby grants to MIT a nonexclusive, worldwide, irrevocable, royalty-free license to exercise any and all rights under copyright, including to reproduce, preserve, distribute and publicly display copies of the thesis, or release the thesis under an open-access license.

Authored by: Josiah McMenamy  
Department of Electrical Engineering and Computer Science  
May 14, 2025

Certified by: Upamanyu Sharma  
Doctoral Candidate  
Thesis Supervisor

Certified by: M. Frans Kaashoek  
Charles Piper Professor  
Thesis Supervisor

Accepted by: Katrina LaCurts  
Chair, Master of Engineering Thesis Committee

# DisViz: Visualizing real-world distributed system logs with space time diagrams

by

Josiah McMenamy

Submitted to the Department of Electrical Engineering and Computer Science  
on May 14, 2025, in partial fulfillment of the  
requirements for the degree of  
Master of Engineering in Electrical Engineering and Computer Science

## Abstract

This thesis aims to provide an intuitive debugging and learning tool for distributed systems that communicate by message passing. Understanding and debugging distributed systems can be challenging and slow to iterate on, so there is a need for tools that can speed up the time it takes to diagnose the root cause of a bug. There exists significant prior work in creating tools that can aid in the visualization and debugging of distributed system executions, such as the ShiViz log visualizer [13]. This work builds on top of these tools to provide more debugging information, handle large log files, and be easily instrumented in existing systems. We demonstrate using the tool to debug issues in an implementation of the Raft consensus algorithm [34].

Thesis Supervisor: Upamanyu Sharma  
Title: Doctoral Candidate

Thesis Supervisor: M. Frans Kaashoek  
Title: Charles Piper Professor

## Acknowledgments

Without the consistent support, encouragement, and guidance of many people, this thesis would not have been possible. I would like to thank Upamanyu Sharma for his constant mentorship and our frequent meetings, Frans Kaashoek for guiding this research and helping determine its scope and goal, and the Parallel and Distributed Operating Systems Group as a whole for being welcoming and open. Thank you to my parents for getting me to MIT and always prioritizing my education.

# Contents

<b>List of Figures</b>	<b>7</b>
<b>List of Tables</b>	<b>9</b>
<b>1 Introduction</b>	<b>10</b>
1.1 Distributed Systems in Go . . . . .	11
1.2 Visualizing Logs . . . . .	12
1.3 ShiViz . . . . .	13
1.4 GoVector . . . . .	14
1.5 DisVis . . . . .	15
1.6 Roadmap for this Thesis . . . . .	15
<b>2 Background and Challenge</b>	<b>16</b>
2.1 ShiViz . . . . .	16
2.1.1 Space-time diagram . . . . .	17
2.1.2 Exploring the Visualization . . . . .	20
2.1.3 Parser Settings . . . . .	21
2.2 GoVector . . . . .	22
2.2.1 Vector Clocks . . . . .	23
2.2.2 API . . . . .	24
2.3 Challenge . . . . .	25
<b>3 DisViz</b>	<b>28</b>
3.1 Ptest . . . . .	29

3.2	GoVector . . . . .	31
3.3	DisViz Client Server Model . . . . .	33
3.4	Search in DisViz . . . . .	34
3.5	Meeting the design goals . . . . .	35
<b>4</b>	<b>Implementation</b>	<b>37</b>
4.1	Ptest . . . . .	37
4.2	Changes to GoVector . . . . .	38
4.2.1	Zap Primitives . . . . .	39
4.2.2	Logging with Zap . . . . .	39
4.2.3	Initialization . . . . .	42
4.3	Changes to ShiViz . . . . .	43
4.3.1	Server . . . . .	43
4.3.2	Client . . . . .	47
<b>5</b>	<b>Case Studies</b>	<b>52</b>
5.1	etcd . . . . .	52
5.2	6.5840 Raft . . . . .	55
<b>6</b>	<b>Evaluation</b>	<b>58</b>
6.1	Leader Election . . . . .	58
6.2	Committing Logs . . . . .	62
6.3	Snapshotting . . . . .	66
<b>7</b>	<b>Related Work</b>	<b>71</b>
<b>8</b>	<b>Conclusion</b>	<b>73</b>
8.1	Future Work . . . . .	73
8.1.1	Change the Server Side Language . . . . .	73
8.1.2	TypeScript . . . . .	74
8.1.3	JavaScript Runtime . . . . .	74
8.1.4	Support Clustering . . . . .	74

8.1.5	File API . . . . .	75
8.1.6	Remote Hosting . . . . .	75

# List of Figures

1-1	A dynamic visualization of the Raft consensus algorithm . . . . .	13
2-1	ShiViz space-time diagram . . . . .	17
2-2	Display of identical logs when hiding and not hiding a process . . . .	17
2-3	Display of identical logs when collapsing and not collapsing consecutive events . . . . .	18
2-4	ShiViz left and right panels supporting the main diagram . . . . .	19
2-5	ShiViz Text Search Input . . . . .	19
2-6	ShiViz Structure Search Input . . . . .	19
2-7	ShiViz Pairwise Execution Display . . . . .	20
2-8	Options to change how the log file is parsed and displayed . . . . .	21
2-9	Latency for ShiViz to create a space time diagram of a Raft test for leader election, varying the number of elections . . . . .	25
3-1	System diagram for generating logs and viewing them in DisViz . . .	29
3-2	Command line options for ptest . . . . .	30
3-3	Live updates during a Go test run using ptest . . . . .	30
3-4	Final ptest output after all tests complete . . . . .	31
3-5	ptest output files . . . . .	31
3-6	DisViz display of log information . . . . .	34
3-7	DisViz controls for shifting the subset of the log file being displayed .	34
3-8	DisViz search bar during active search . . . . .	35
3-9	Latency to create visualization for Raft test in ShiViz and DisViz . .	35

4-1	An abridged call chain for making a logging call with a <code>GoLog</code> object	42
4-2	A highlighted event during a <code>DisViz</code> search . . . . .	48
4-3	Time spent in the browser when rendering all log events for an execution	50
6-1	Last log from <code>TestManyElections3A</code> failure . . . . .	60
6-2	Logs from <code>TestInitialElection3A</code> failure . . . . .	62
6-3	Displaying a raft object in <code>DisViz</code> . . . . .	63
6-4	<code>AppendEntries</code> failure from <code>raft_1</code> to <code>raft_0</code> . . . . .	64
6-5	Raft followers successfully committing a log entry . . . . .	65
6-6	Raft follower ignoring redundant logs from leader . . . . .	67
6-7	Raft follower after processing an <code>InstallSnapshot</code> RPC . . . . .	68
6-8	Raft leader updating its commit index . . . . .	68



# List of Tables

1.1	Survey of open-source systems implemented in Go . . . . .	11
5.1	Code changes required to integrate GoVector . . . . .	57

# Chapter 1

## Introduction

Debugging distributed systems is a challenging task. Bugs may be due to null pointer de-references, off-by-one errors, or floating point errors. Bugs may be due to the concurrent behavior in the code such as deadlocks, race conditions, or starvation. Bugs can also be introduced by the interactions between nodes in the system: network partitions, message drops, and timing differences between nodes. Reproducing the bug reliably is also difficult because it can require simulating the network conditions that were present when the bug occurred. In order to find the cause of the bug, the programmer needs to understand the execution of the system and how the nodes were interacting when the bug occurred.

One tried and true method for debugging is to use print or logging statements to see hints about the execution of the code. With logging, the programmer has granular control over seeing what path the code has taken and what the state of the system is on that code path. Unfortunately, distributed system executions can produce massive log files that give too much information to the programmer, who must tediously read through the logs and reconstruct the execution to find the bug. Further, reading through a log file makes it difficult to understand the relative ordering between events in a distributed system.

Programmers would benefit from tools that can take this log information and help them more quickly understand what the distributed system is doing. However, these tools must be careful to help, not hinder the iterative process of debugging. The

programmer must be able to easily add information to the logs while hypothesizing the cause of a bug, in order to get more information to confirm or alter their hypothesis.

The tool used for understanding the logs should not pose restrictions on the kinds of information that can be in the logs, otherwise the tools becomes inflexible and cumbersome to use. If some structure does need to be imposed on the logs to ensure metadata information is present or a certain format is used, then the tool should provide an API in the language that helps generate logs compatible with the tool.

## 1.1 Distributed Systems in Go

System	Description
Kubernetes [8]	An open-source system for automating the deployment, scaling, and management of containerized applications.
etcd [23]	A distributed, reliable key-value store for critical data in distributed systems.
Docker [5]	A platform for containerizing applications.
CockroachDB [30]	A cloud-native, distributed SQL database designed for consistency, scalability, and resilience.
SeaweedFS [9]	A fast, simple, and scalable distributed file system for handling billions of files.
rqlite [33]	A lightweight, distributed relational database built on SQLite.
Jaeger [7]	A distributed tracing platform for monitoring and troubleshooting microservices-based systems.
Temporal [39]	A durable execution system for managing and scaling microservice applications.

Table 1.1: Survey of open-source systems implemented in Go

The choice of language impacts many characteristics of a distributed system, such as the performance, fault tolerance, and maintainability. Some languages are suited better than others for implementing a distributed system. For example, the Erlang language was designed specifically for distributed systems, with lightweight processes and hot swappable components [3, 38]. More recently, the Go programming language has risen in popularity for distributed system programming, developed at Google, a leader in scalable distributed systems [6].

Go also has lightweight processes in the form of Goroutines, as well as native support for networking such as HTTP servers, RPC, and TLS. With static typing and robust tooling including a race detector, Go makes it easier to prevent and detect bugs before they crash a program. Many modern distributed systems in use today are implemented in Go, as can be seen in Table 1.1. For this reason, this thesis focuses on tools that aid in generating and understanding logs from distributed systems written in Go. etcd is a core distributed system used in the Kubernetes container orchestration software [22, 23]. It will be used as a litmus test to determine if a tool works well in a large distributed system.

## 1.2 Visualizing Logs

In seeking to understand the events in a log file better, one natural desire is to be able to visualize what the distributed system is doing. This helps gain context and intuition for how nodes in the system are interacting. The kinds of visualizations that are possible with distributed systems are highly varied and depend on the nature of the application, but in general visualizations using graphs or nodes are applicable since a graph can intuitively represent the topology of a distributed system, giving a way to encapsulate the entire system as one object.

Visualizations may be static or dynamic. Dynamic visualizations are generally more difficult to implement because live animation introduces engineering challenges for performance and proper controls that the user can quickly learn. One example of a dynamic visualization is the visualization of the Raft consensus algorithm by the Raft authors [36, 34]. Figure 1-1 shows the visualization. The nodes in the system are arranged in a circle, the user can control how often the nodes send messages to each other, and the user can manually control the actions of an individual node, e.g. to partition it away from the system. The logs for each node are shown on the right side. This particular visualization is useful for learning Raft, but requires a working Raft implementation for the visualization to work.

An example of a static visualization is a space-time diagram, as seen in Figure 2-1.

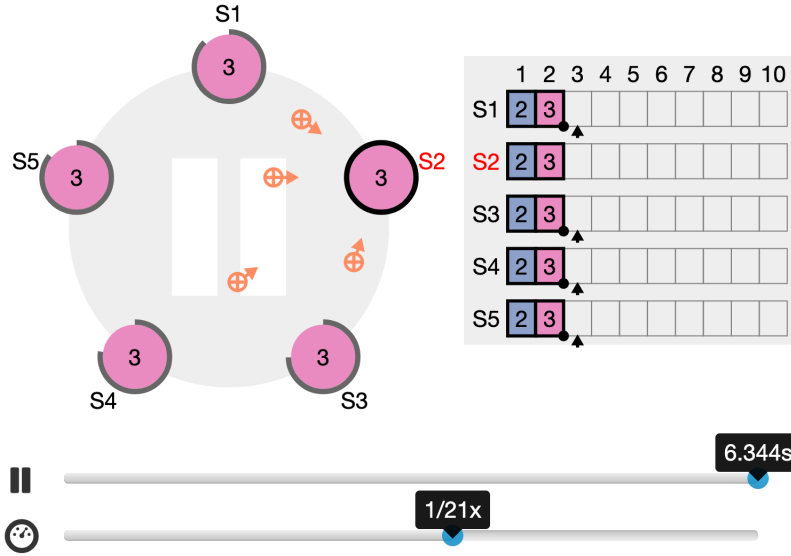


Figure 1-1: A dynamic visualization of the Raft consensus algorithm

Here, the members in the system are arranged along one axis. Perpendicularly, the flow of time in the execution is depicted, and messages between nodes are shown as nodes, with lines between them depicting one member sending a message to another. A static visualization has the advantage of requiring less for the user to learn to interact with it, being simpler to implement and extend, and allowing the user to process the information at their own pace. A space-time diagram also makes it obvious when a member stops responding to others, which may be difficult to observe in a dynamic visualization with many members.

### 1.3 ShiViz

Some prior work has been done in creating a space-time diagram to visualize distributed system logs. The ShiViz system, seen in Figure 2-1, was created for that purpose, by Ivan Beschastnikh, et. al [13, 16]. This is an entirely client side application that allows the user to load a log file from their machine with a format given by a user defined regular expression. It creates a partial ordering of events from that log file, displaying it as a space-time diagram.

As we see in more detail in section 2.1, there are some limitations to the existing version of ShiViz that prevent it from being used to debug real-world distributed systems. Most importantly, loading and processing the entire file on the client side introduces too much latency as the log file becomes large. With a log file that has 12,000 events, there is a latency of around 7 seconds just to load the file, seen in Figure 2-9, and the webpage becomes slow to respond to user actions thereafter. Further, using a regular expression to parse each log event from the file is too rigid and inflexible. If a user wants to put information specific to a single log call in the file, they must resort to a catch-all capture group in the regex to contain all fields custom for that log call, which becomes difficult to visually parse on the webpage.

## 1.4 GoVector

As mentioned above, using ShiViz requires having a log file that can be parsed with a user defined regular expression. By itself, this would present an annoyance for the user, who must implement a way to carefully generate log files that conform to a particular regex. Along with ShiViz, Beschastnikh, et. al also created logging libraries to do that in Go, C, C++, and Java [12]. We focus on the Go logging library, GoVector [15].

GoVector includes a built in RPC library for sending messages. This allows GoVector to add vector clock and other information to the RPC payload, but contrains the developer too much. For an implementation that is already using a different API to send RPCs over the network, it requires non-trivial effort to correctly instrument the system in a backwards manner. In this situation, functionality already exists to make the RPC payload and send it across the network, but GoVector requires wrapping each payload to include the vector clock of that node, so the payload sent across the network contains the original payload the system wanted to send. For an implementation of the Raft consensus algorithm, the total number of additional lines needed to make this change is not large, around 80, however it is error prone, time consuming, added in different places in the code, and introduces an unnecessary layer of complexity to sending RPCs.

## 1.5 DisVis

In order to rectify these issues and build a tool that can be used with real-world distributed systems, we propose DisViz, which extends ShiViz and GoVector, focusing on the Go programming language. The main design goals for DisViz are to:

- be able to display large log files with low latency
- be compatible with any library used to send RPCs
- make it easy to add information to individual logs

In order to ensure that these changes help with debugging existing real-world distributed systems, we demonstrate using DisViz to find bugs in a Raft implementation.

## 1.6 Roadmap for this Thesis

Chapter 2 of this thesis provides more background on debugging distributed systems. Chapter 3 provides a detailed summary of the features of DisViz, as well as changes made from the ShiViz and GoVector. Chapter 4 dives into the implementation details and technical challenges encountered to create DisViz. Chapter 5 shows two case studies of instrumenting an existing system to generate DisViz-compatible logs, in etcd and a Raft implementation from MIT’s distributed systems course 6.5840. Chapter 6 evaluates DisViz by demonstrating its usability to investigate bugs in the Raft implementation. Chapter 7 discusses related work in debugging distributed systems. Chapter 8 concludes this thesis and proposes ideas for avenues for future work on DisViz.

# Chapter 2

## Background and Challenge

Using GoVector and ShiViz to debug a distributed system in Go locally, the process is:

- Refactor or create the system to send RPCs using the GoVector library
- Run tests of the system using any preferred method or framework. Tests running in parallel must be careful to not use identical names for log files for GoVector logs.
- Run a GoVector script manually to glue all the separate log files for each node in the system into one log file
- Open the ShiViz webpage in a browser and upload the combined log file. All the processing of the file happens on the client side

### 2.1 ShiViz

ShiViz is a static site that can be run locally, but it is also hosted by Beschastnikh et. al and can be experimented without additional setup [14].



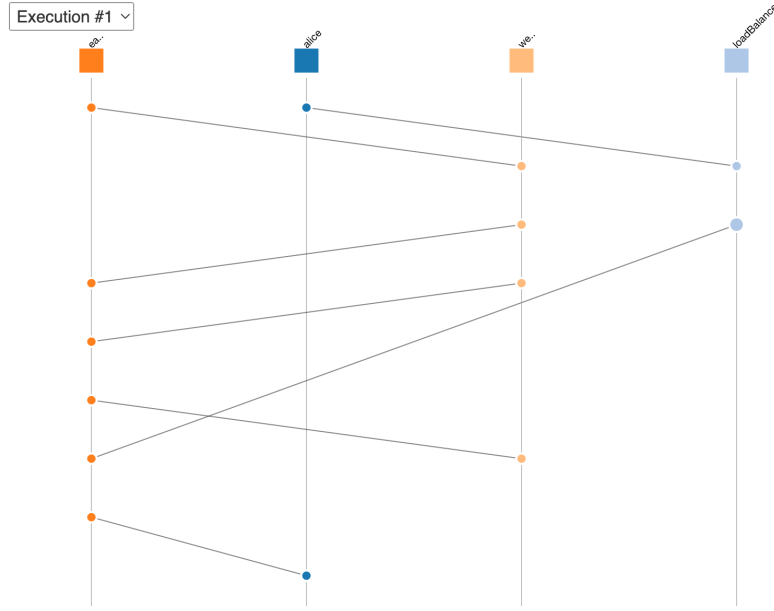


Figure 2-1: ShiViz space-time diagram

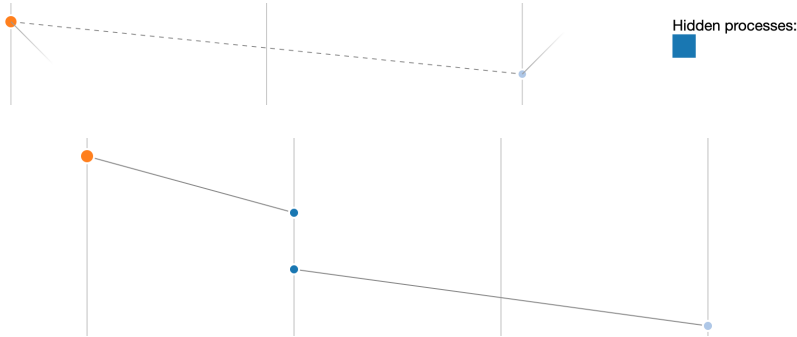


Figure 2-2: Display of identical logs when hiding and not hiding a process

### 2.1.1 Space-time diagram

ShiViz centers around a space-time diagram of events in the logs. Time flows from top to bottom. The main middle panel, seen in Figure 2-1, displays a directed acyclic graph of the partially ordered vector timestamps parsed from the input log. The row of squares at the top represents all the nodes in the system, a singular thread of execution. Clicking on a process allows the user to hide that process from the diagram, as seen in Figure 2-2, or filter the diagram to show only those processes/events that communicate with the filtered process. Multiple processes can be hidden and filtered

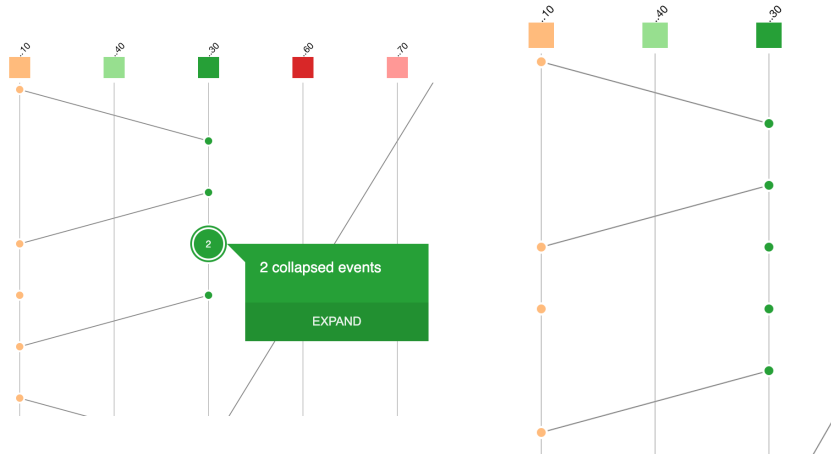


Figure 2-3: Display of identical logs when collapsing and not collapsing consecutive events

at the same time. The circles below a process are events that were logged by the process. Lines connecting two circles represent the happened-before relation between the events: the higher of the two events happened before the lower event. Dashed lines represent transitive communication edges. These only appear when a process is hidden, and two processes that are not hidden communicated indirectly through this process. ShiViz collapses adjacent process events not incident on any communication edges into larger circles, seen in Figure 2-3. These have a number inside of them, indicating the number of events that they represent. The user can click on these events to expand the events. Radiating lines that fade out represent communication edges to processes that are currently hidden from view. Each process is associated with a unique color.

The left and right panels beside the main diagram are seen in Figure 2-4. The left panel shows the event capture group from each line in the log, as well as the line number in the original log file it parsed from, with the same color as the corresponding host in the diagram. The right panel shows all capture groups parsed from this log event, except for the vector clock. Clicking on a circle in the diagram shows the same information displayed on the right panel.

Log lines

Motifs

Clusters

Execution #1

/timeline uid=alice location=kansas

Initiating sync dest=204.15.23.252

Request for timeline uid=alice location=

79 Received sync request src=69.63.191.255

5

/timeline uid=alice location=kansas dest

Sending confirmation dest=69.63.191.255

Sync confirmed src=204.15.23.252

Initiating sync dest=69.63.191.255

Received sync request src=204.15.23.252

PAIRWISE

Received sync request s

rc=69.63.191.255

ip: 204.15.23.252

date: 5/27/2013 10:52:24 AM

action: INFO

host: westDC

Figure 2-4: ShiViz left and right panels supporting the main diagram

Search the visualization

Search History

Text Search

Structured Search

Text search: search for log lines/events that match a text query.

Examples:

sync

Find events containing the text "sync" in any field.

priority=CRITICAL && text=/fail.\*/

Find events with "priority" field set to "CRITICAL" and with "text" field matching the specified regex. Supported logical operators are: &&, ||, ^

ip="216.58.216.174"

String literals containing non-alphanumeric symbols must be surrounded by quotes.

location="EC2" || (isMobile && useragent=/. \*Webkit.\*/)

Use parenthesis to specify order of operations.

Figure 2-5: ShiViz Text Search Input

#structure=[{"host":"a","clock":{"a":1}}, {"host":"b","clock":{"b":1,"c":1,' X

Search History

Text Search

Structured Search

Search for a custom structure: draw a graph structure below (add processes, events, click and drag to add inter-event edges).

+

Search for a pre-defined structure: Select one of the options below to find the specified structure.

REQUEST-RESPONSE

BROADCAST

GATHER

Figure 2-6: ShiViz Structure Search Input

19

## 2.1.2 Exploring the Visualization

ShiViz supports keyword search across the parsed fields, allowing for logical connectives and regular expressions, seen in Figure 2-5. The user can also search for subgraphs or communication topologies of interest. ShiViz supplies pre-defined structures like broadcast or request-response, but custom communication patterns can also be defined, seen in Figure 2-6. After searching, the visualization will grey out all events that do not match the search criteria, and allow the user to jump to the different results of the search.

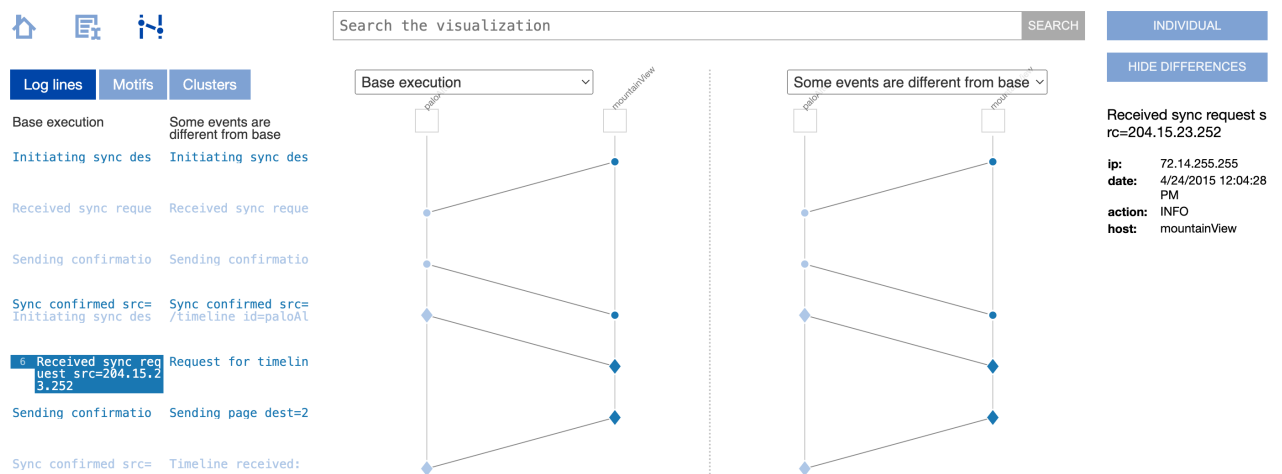



Figure 2-7: ShiViz Pairwise Execution Display

When viewing two executions side-by-side, by clicking the "pairwise" button in Figure 2-4, clicking on "show differences" highlights the differences between two executions, seen in Figure 2-7. Hosts that are not common to both executions are represented as rhombuses. Processes present in both executions have their events compared by the event capture group, and different events are drawn as rhombuses.

For logs with multiple executions, clicking on the "clusters" tab separates executions into different groups based on a chosen metric. Clustering by the number of processes groups executions by the midpoint between the smallest and largest number of processes. Clustering by execution comparison gives an overview of how executions differ from a selected base.

Clicking on the "motifs" tab highlights frequently occurring communication patterns within and across executions. The user can search for 2, 3 or 4-event motifs that occur in at least 50% of the executions or that appear at least 5 times within a single execution.

### 2.1.3 Parser Settings



The screenshot shows a web form titled "Options". It contains several input fields and controls:

- Select a file:** A button labeled "Choose File" next to a text box containing "No file chosen". A help icon (question mark in a circle) is to the right.
- Log parsing regular expression:** A text box containing the regular expression: `(?<ip>(\d{1,3}\.){3}\d{1,3}) (?<date>('`
- Multiple executions regular expression delimiter:** A text box containing the delimiter: `^=== (?<trace>.*) ===$`
- Sort processes in** a dropdown menu set to "descending", followed by **order by:** with two radio buttons: ☒ "# events" and ☐ "appearance in log".
- A large blue button labeled "VISUALIZE" at the bottom.

Figure 2-8: Options to change how the log file is parsed and displayed

Before the log file is parsed and visualized for the user, they must tell ShiViz how to parse events from the log. This is done with the settings seen in figure Figure 2-8. Most importantly, a regular expression is entered by the user (or may be parsed from the log file) that must contain at least three named capture groups, for the host, vector clock (discussed more in subsection 2.2.1), and the event that occurred. Users can add additional capture groups, but every log event must conform to this regular expression. A delimiter can be input to identify where different traces start in the log file, if the file contains logs from multiple executions of a test, for example. The rest of the settings let the user change the order that hosts in the system are shown from left to right.

Using a regular expression with capture groups is not flexible enough for the iterative process of debugging. A programmer often needs to add specific information or variables to logs at different points in the code, such as the current number of votes that a raft node has while asking its peers to become leader. If they want this information to be nicely displayed as a separate field similar to the host and event, then they need to add a capture group for the field to the regular expression. They have to either add this information to every other log in the program, which is not feasible, or carefully structure the regex to let this capture group match 0 characters in the log event, without disrupting the matching of the rest of the capture groups. This is error prone and would introduce bugs, which is the opposite of what the tool should be doing.

Instead, the standard practice for this problem is using a catch-all capture group, and putting all desired custom information in that capture group. This works, but the point of the ShiViz tool is to display the log information in a helpful way for the user, and a catch-all capture group would display as a large block of text, which is not easily parseable as more information is added to a particular log. This is one pain point that DisViz addresses.

## 2.2 GoVector

GoVector is an open-source library implemented in Go that provides logging of vector clocks [15]. By using vector clocks, GoVector enables ShiViz to order events across multiple processes without relying on a centralized clock. The repository is structured as follows:

- `govec/govec.go`: Contains the main API for the `GoLog` struct that clients will call to log events.
- `govec/vclock`: Implements a vector clock library
- `govec/vrpc`: Provides an integration between Go's RPC and the `GoLog` object

For serialization when sending RPCs, GoVector uses an implementation of the MessagePack format [31]. If a different serialization method is needed, the user can customize this behavior by providing encoding and decoding callback functions:

```
1 func (gv *GoLog) setEncoderDecoder(  
2     encoder func(interface{}) ([]byte, error),  
3     decoder func([]byte, interface{}) error,  
4 ) {  
5     gv.encodingStrategy = encoder  
6     gv.decodingStrategy = decoder  
7 }
```

The serialization and `vrpc` APIs are useful for a simple project that does not have strict RPC needs, but for a larger project like `etcd`, GoVector is not a sufficient library for sending RPCs between nodes. `etcd` uses a Go implementation of gRPC, which is a RPC framework that allows for multiplexed, low-latency transport and uses Protocol Buffers for compact, strongly-typed message serialization [25, 24]. It offers features such as bidirectional streaming, deadline propagation, built-in authentication, load balancing, and automatic code generation. None of those features are supported by `vrpc`, which supports only unary request-response communication. The `vrpc` library could be extended to support these features or become an implementation of gRPC, but from a design perspective, this is mixing unrelated goals. In DisViz, we address this problem by focusing on logging without dictating the RPC library.

### 2.2.1 Vector Clocks

The core mechanism in GoVector is the vector clock. The library implements the vector clock algorithm to maintain a partial order of events by:

- Initializing a vector clock in each process with an entry for the process identifier (PID) and starting at time zero.
- Incrementing the local clock when a local event occurs.
- Merging received vector clocks with the local vector clock upon message reception.

The merge operation takes the entry-wise maximum of the clock values.

This approach ensures that every logged event is stamped with a vector timestamp. The log messages include the process identifier and the current state of the vector clock at that point in time. The `VClock` type has methods for interacting with the vector clock, and the underlying type is simply `map[string]uint64`.

### 2.2.2 API

It is possible to use `GoVector` without using the `vrpc` functionality, but the API methods to create an RPC payload wrap the desired buffer to send in a custom struct that contains the `GoVector` specific information:

```
1 d := vclock.VClockPayload{
2     Pid: gv.pid, VcMap: gv.currentVC.GetMap(), Payload: buf
3 }
```

This can present an issue when the user has their own RPC library and serialization method. For a small distributed system where it is feasible to wrap the RPC library with code that uses this `VClockPayload`, this approach works and is relatively easy to implement. But for a larger system with a more complex RPC library, this approach becomes counterproductive. For example, Protocol Buffers use strongly typed structs to aid in defining data types between nodes, and the types also help performance of the serialization. Having to use this `vclock.VClockPayload` erases type information of the `buf` that is sent, which can decrease performance of serialization methods like Protocol Buffers. A logging library should not introduce friction with message serialization. `DisViz` addresses this problem.

Each log output from `GoVector` includes a message, PID, and vector clock. The primary `GoVector` API functions include:

- `InitGoVector(processid string, logfilename string, config GoLogConfig):`  
Initializes a new logger instance for the specified process.
- `PrepareSend(msg string, buf interface, opts GoLogOptions):` Called before sending an RPC, this function increments the local vector clock, logs



an event, and encodes the payload together with the vector clock using a user defined serialization strategy (or MessagePack by default).

- `UnpackReceive(msg string, buf []byte, unpack interface, opts GoLogOptions):`  
Called upon receipt of a message, this function decodes the byte array to extract the payload and vector clock, merges the vector clock with the local vector clock, and logs an event.
- `LogLocalEvent(msg string, opts GoLogOptions):` Used for logging events that occur locally. It increments the local vector clock and logs the event.

There are also configurable logging options through the `GoLogConfig` (e.g. buffered logging, encoding/decoding strategy) and `GoLogOptions` (e.g. log priority) structures.

## 2.3 Challenge

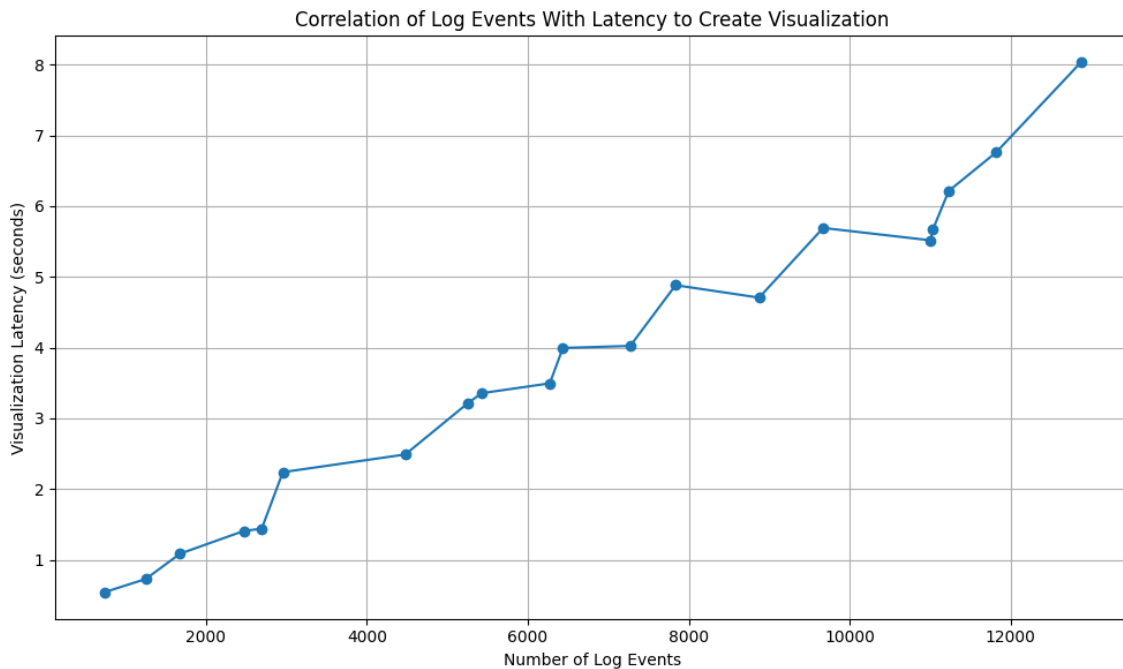


Figure 2-9: Latency for ShiViz to create a space time diagram of a Raft test for leader election, varying the number of elections

## Reduce latency in DisViz

As mentioned in section 1.5, DisViz has three main design goals. The most important goal is reducing latency when viewing the logs. Since ShiViz is a static site, it must do all the processing of the logs within the browser environment, and redraw the entire visualization any time a change needs to be made to it (e.g. collapsing nodes). This issue becomes apparent when trying to view logs with thousands of events or more. There is significant latency in preparing the visualization, and latency with any interaction like searching, clustering, or collapsing nodes. In order to make the debugging process as fast as possible, latency in the debugging tool must be low.

Figure 2-9 shows how that the latency to create the visualization changes linearly with the number of log events. This data was collected by running a test of a Raft implementation that does leader election for a group of 7 nodes. Each iteration for a varying number of iterations per test, three random servers are disconnected, the test makes sure there is still a leader, then those three are reconnected. etcd tests often create log files with tens of thousands of events, mainly from the number of RPCs passed between nodes, rather than from local events. For complex distributed system, this latency is not acceptable for debugging. The machine used to to run ShiViz and measure latency was a 2020 MacBook Air with a 1.1 GHz Dual-Core Intel Core i3 processor and 8 GB 3733 MHz LPDDR4X RAM. The browser used to run the ShiViz website was the Brave browser, which is built on Chromium and uses the V8 JavaScript engine.

## Make GoVector more flexible

For existing systems that already have their RPC library implemented, the GoVector API is too cumbersome to easily use in their system to generate DisViz compatible logs. It is possible to use GoVector without the vrpc functionality, but wrapping the RPC payload with a GoVector defined struct adds an unnecessary layer of complexity that may even reduce performance of the serialization strategy. The second design goal of DisViz corrects this, allowing GoVector to be used with any RPC library.

## **Add different information in logs**

The last design goal of DisViz is to make it easy for users to add information to specific GoVector logging calls, and for that information to be displayed well in the visualization. The use of regular expressions in ShiViz is the main reason for this pain point. Using a regular expression with capture groups that every log must adhere to is too strict. In the GoVector API all information is added to one message field for an event, so adding information unique to one log means adding it to a catch-all capture group that will display everything in that group as one string. This will be tedious for the user to program and annoying to parse through in the visualization. DisViz fixes this by changing the GoVector API to accept any number of key-value pairs for a log, which will be formatted as JSON in the log file and easily parsed by DisViz. No regular expressions are used, and each key-value pair is separately and cleanly displayed for each log.

# Chapter 3

## DisViz

A diagram for the DisViz system can be seen in Figure 3-1, which illustrates the entire process a user will go through to generate and visualize the log file for a test of their distributed system. The diagram starts with ptest, a lightweight parallel testing framework discussed more in section 3.1. This allows users to easily execute parallel runs of multiple tests and bundle the DisViz log files from each node in the system during a test. When a test is running, the source code uses the updated GoVector API to log when an RPC is sent or received, as well as log any local events useful for debugging. The logging functions accept any number of key value pairs, and GoVector adds the vector clock and PID metadata to each log. GoVector then uses the Zap library to format those key value pairs into valid JSON logs and write to a log file. Zap is a Go logging library from Uber for fast, structured, leveled logging [40]. With the log file, the user can then start up DisViz's server and navigate to the DisViz website in a browser. Once on the website, they can load the recent log file, and the website's JavaScript will communicate with the server to view small sections of the log file at a time. The user can then navigate through the log file and search for text or communication patterns present in the file. The rest of this chapter will give more detail about each step in the diagram.

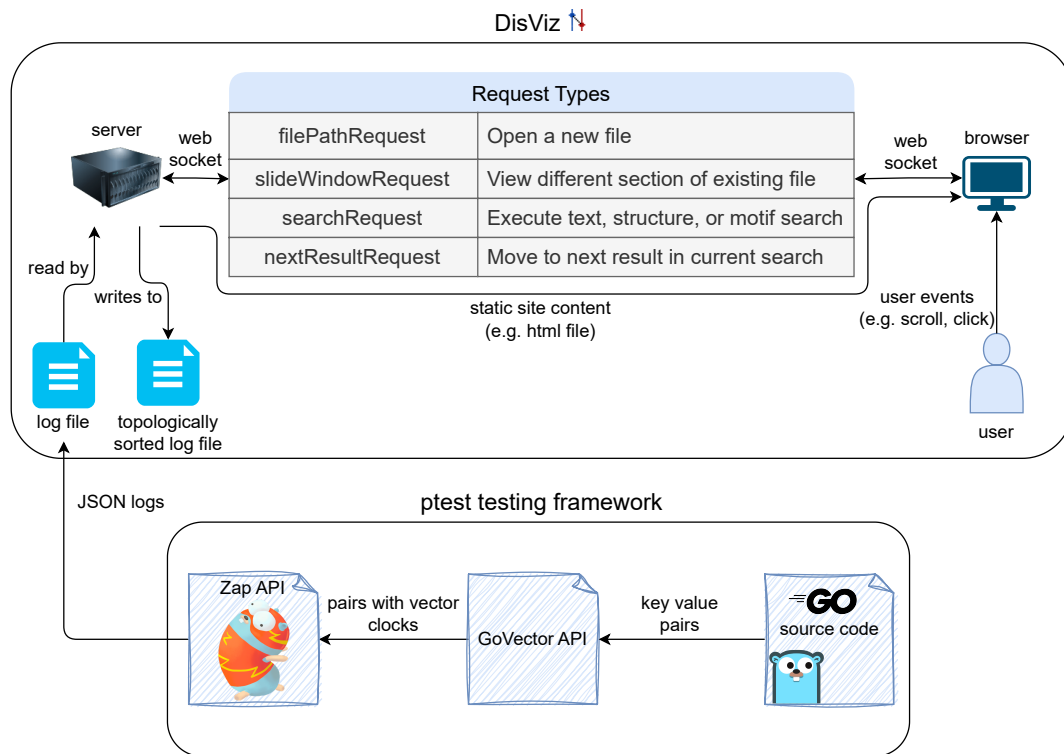


Figure 3-1: System diagram for generating logs and viewing them in DisViz

### 3.1 Ptest

Ptest is a command line tool for running Go tests that gives the user granular control over things like the number of parallel workers running tests, which tests to run, and the number of iterations for each test. When debugging a distributed system, a race condition may cause a test failure infrequently. For a bug related to snapshots in the Raft algorithm, introduced in section 6.3, a failure is only observed around twice in every 100 test runs. It is necessary to be able to run a particular test many times in parallel while debugging, in order to iterate as quickly as possible. The native Go testing framework makes it easy to designate that a test can be run in parallel with `t.Parallel()`, but this signals that this test is to be run in parallel only with other parallel tests; multiple instances of a single test never run in parallel with each other [10]. In order to run an individual test in parallel, the test itself must be implemented

with this in mind using more of the Go testing library, such as `t.Run(name string, f func(t *T))` along with `t.Parallel()` to run subtests in parallel. When tests have not been specifically implemented for running in parallel, a testing framework like ptest is helpful. ptest uses and extends the dstest framework described by a Teaching Assistant of 6.5840 at MIT [35]. dstest provides the command line options seen in Figure 3-2, displays live results of the tests as they are running (Figure 3-3), and gives a summary of the results once all the tests are done running (Figure 3-4). These figures are the output of running this command on a Raft implementation:

```
ptest -iter 5 -workers 10 TestManyElections3A TestBasicAgree3B TestReliableChurn3C
TestSnapshotBasic3D
```

Arguments			
<b>*</b>	<b>tests</b>	<b>TESTS...</b>	[default: None] [required]
Options			
<b>--sequential</b>	<b>-s</b>		Run all test of each group in order
<b>--workers</b>	<b>-p</b>	<b>INTEGER</b>	Number of parallel tasks [default: 1]
<b>--iter</b>	<b>-n</b>	<b>INTEGER</b>	Number of iterations to run [default: 10]
<b>*</b>	<b>--output</b>	<b>-o</b>	<b>TEXT</b> Output path to use, required [default: None] [required]
	<b>--verbose</b>	<b>-v</b>	<b>INTEGER</b> Verbosity level [default: 0]
	<b>--archive</b>	<b>-a</b>	Save all logs instead of only failed ones
	<b>--race</b>	<b>-r</b>	Run with race checker [default: no-race]
		<b>--no-race</b>	<b>-R</b>
	<b>--loop</b>	<b>-l</b>	Run continuously
	<b>--growth</b>	<b>-g</b>	<b>INTEGER</b> Growth ratio of iterations when using <b>--loop</b> [default: 10]
	<b>--timing</b>	<b>-t</b>	Report timing, only works on macOS
	<b>--timeout</b>	<b>-z</b>	<b>INTEGER</b> Set timeout in seconds for each test [default: 1]
	<b>--extra</b>	<b>-e</b>	<b>TEXT</b> Give extra args to each go test command
	<b>--help</b>		Show this message and exit.

Figure 3-2: Command line options for ptest



Figure 3-3: Live updates during a Go test run using ptest

ptest extends dstest to improve organization of output files and produce a single log file for each test that can be loaded into DisViz. All the test output files are added to a directory where the tests were run. For each failed test (or all tests if **-archive** is true), ptest will make a directory with the name of the test that failed. The directory

Test	Failed	Total	Time
TestManyElections3A	0	5	18.41 $\pm$ 4.97
TestBasicAgree3B	0	5	10.02 $\pm$ 3.49
TestReliableChurn3C	2	5	22.28 $\pm$ 3.30
TestSnapshotBasic3D	0	5	16.06 $\pm$ 5.78

Figure 3-4: Final ptest output after all tests complete

contains each of the per node files created by GoVector, an overall system combined log file with each node file appended together, and another log file capturing all the data from standard out, which will contain print output during the test. This file structure can be seen in Figure 3-5.

```

v 20250424_021541
  v TestReliableChurn3C_6
    ≡ combined_logs.log
    ≡ raft_0_log_file-zap-Log.txt
    ≡ raft_1_log_file-zap-Log.txt
    ≡ raft_2_log_file-zap-Log.txt
    ≡ raft_3_log_file-zap-Log.txt
    ≡ raft_4_log_file-zap-Log.txt
    ≡ TestReliableChurn3C_6.log
  > TestReliableChurn3C_18

```

Figure 3-5: ptest output files

## 3.2 GoVector

While the tests are running, the source code uses the updated GoVector API to generate valid DisViz log files. The updated GoVector API is inspired by the Zap API and offers the same methods as the Zap API. The previous GoVector API is unchanged for backwards compatibility, but new methods are added. Zap is a widely used Go

logging library that allows for fast, structured logging, with highly customizable formatting [40]. An example of a logging call using the updated GoVector API (same in the Zap API) is:

```
1 goLog.Info("failed to fetch URL",
2     // Structured context as strongly typed Field values.
3     zap.String("url", url),
4     zap.Int("attempt", 3),
5     zap.Duration("backoff", time.Second),
6 )
```

Using a strongly typed API to log data gives more type safety to the programmer, which helps reduce bugs related to logging. The API also allows for the user to forgo this type safety with a more flexible "sugared" logger:

```
1 goLog.Sugar().Infow("failed to fetch URL",
2     // Structured context as loosely typed key-value pairs.
3     "url", url,
4     "attempt", 3,
5     "backoff", time.Second,
6 )
7 goLog.Sugar().Infof("Failed to fetch URL: %s", url)
```

The user can also easily make copies of a GoLog to add information to each log or enable various Zap options:

```
1 // make a new logger that adds 'grpc' and a url to each log,
2 // and a stack trace for logs at or above Info level
3 goLog.Named("grpc").With(zap.String("url", url)).WithOptions(
4     zap.AddStacktrace(zapcore.InfoLevel)
5 )
```

If the system already has an RPC library, the user does not have to wrap their payload inside a GoVector payload. Instead, they can put the GoVector payload that includes the process's vector clock inside their RPC payload:

```
1
2 // new methods
3 func (gv *GoLog) PrepareSendZap(msg string, level zapcore.Level, fields
4     ...zap.Field) (encodedBytes []byte)
5
6 func (gv *GoLog) UnpackReceiveZap(msg string, buf []byte, level zapcore.Level,
7     fields ...zap.Field)
```



### 3.3 DisViz Client Server Model

After the tests finish and ptest zips up the JSON log files written by the Zap API into one combined log file for each test, the DisViz server parses the logs. The server has two functions. First, it acts as a standard web server, hosting all the static content of the DisViz website. This includes the main HTML file, and all the front end JavaScript files that respond to user input and create the visualization. Second, it hosts a websocket that the JavaScript on the browser will connect to once the website is opened on the browser. On the DisViz website, the user supplies a file path to a log file on the server's file system. For now the server and browser are assumed to be running on the same device, but this is not required (see subsection 8.1.6). When a log file path is submitted by the user, the browser sends a `filePathRequest` over the websocket to the server. With this path, the server now does the heavy lifting, reading and parsing the entire file. It constructs an ordering of all the log events using the vector clock and writes a topologically sorted ordering of those events back to the file system. It is from this sorted file that the server sends small sections to the browser. This ensures that valid subsets of events are sent to the browser.

After the browser receives the initial subset of data sent by the server, it parses and displays the logs just like it previously did in ShiViz. The user can interact with the visualization the same as before (with caveats, see subsection 8.1.4). Once the user scrolls to the bottom or top of the visualization, the browser sends a `slideWindowRequest` over the websocket. The server responds by reading and sending a shifted portion of data from the sorted log file, and the browser parses and displays the new visualization.

The logs sent over the websocket are valid JSON, from which the browser parses all the keys and values, displaying each key-value pair separately. With the updated GoVector API, the caller, function, and stacktrace information can easily be added to the logs when desired, as seen in Figure 3-6.

When the browser receives data from the server, it displays the subset of data the user is viewing, and gives controls for the user to manually decide what subset they are viewing. These controls, seen in figure Figure 3-7, are given on the right panel



Figure 3-6: DisViz display of log information

next to the visualization.

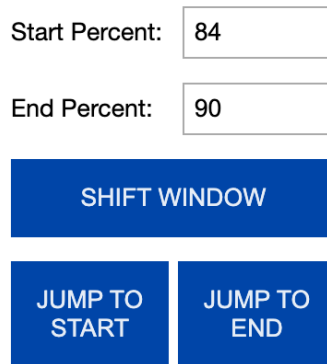


Figure 3-7: DisViz controls for shifting the subset of the log file being displayed

## 3.4 Search in DisViz

When the user enters a query in the search bar, the browser sends the query over the websocket in a `searchRequest`. The server again does the heavy lifting of searching through the entire log file for any matches to the query, then sends the browser the

number of matches, and a subset of the log file that contains the first match. The browser then parses and displays the received data. The search bar now shows how many results there are and what position in the results the user is at, see figure Figure 3-8.



Figure 3-8: DisViz search bar during active search

When the user clicks the navigation buttons in the search, or explicitly enters a result number to view, a `nextResultRequest` is sent to the server, which will return a subset of the data containing the next search result.

### 3.5 Meeting the design goals

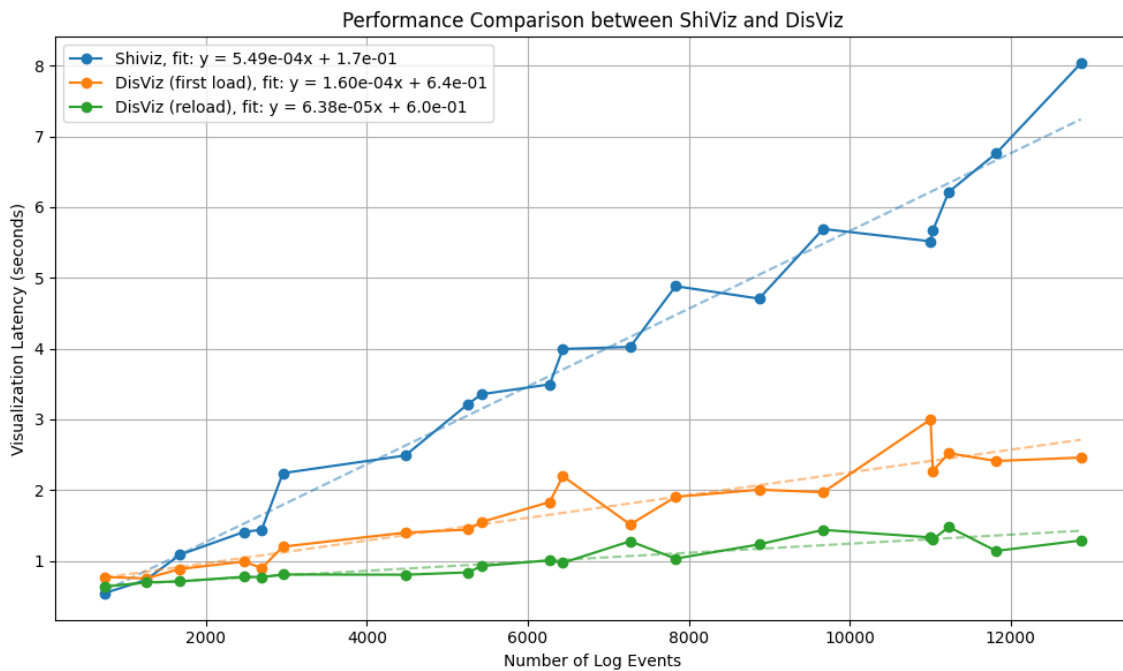


Figure 3-9: Latency to create visualization for Raft test in ShiViz and DisViz

## **Reduce latency in DisViz**

Moving to a client server model in DisViz significantly improves the latency to create and interact with the visualization, as desired. See Figure 3-9 for a comparison between ShiViz and DisViz, visualizing the same logs as described in Figure 2-9. DisViz is around 3.4 times faster than ShiViz in creating the visualization, comparing the slope of the fitted least squares regression lines. There is an added benefit when a file is opened in DisViz that has been visualized before. Since DisViz writes a topologically sorted version of the log file to the file system, visualizing the file after the first time will be faster, around 8.6 times faster than ShiViz, since the sorted file can be immediately read from the file system. Section 4.3.2 discusses the likely reasons for the improvement in latency.

## **Make GoVector more flexible**

With the changes to the `PrepareSend` and `UnpackReceive` methods in GoVector, the user can put the vector clock payload inside their RPC payload, instead of the other way around. This makes it easier to instrument an existing distributed system with GoVector.

## **Add different information in logs**

In GoVector, the user now has a clear and readable way to add unique information to each log. Since DisViz no longer uses regular expressions to parse each log line, instead parsing the log as JSON key value pairs, the information in each log is displayed in a more readable way, as in Figure 3-6.

# Chapter 4

## Implementation

### 4.1 Ptest

As mentioned, ptest extends dctest to organize output files better in the context of using DisViz. dctest is implemented in Python, and ptest is a Bash script that executes dctest, passing through the command line arguments to dctest, then does some clean up afterward. An abridged version of ptest is seen below:

```
1 OUTPUT=$(python3 -u $SCRIPT_PATH -o "$RESULT_PATH" "$@" | tee /dev/tty)
2 if [ -d "$RESULT_PATH" ]; then
3     for folder in "$RESULT_PATH"/*; do
4         if [ -d "$folder" ]; then
5             GoVector --log_type zap --log_dir "$folder" --outfile
6                 "$folder/combined_logs.log"
7             subfolder=$(basename "$folder")
8             mv "$RESULT_PATH/$subfolder.log"
9                 "$RESULT_PATH/$subfolder/$subfolder.log"
10        fi
11    done
12 fi
```

While dctest is running, for each test it captures all output meant for standard out and writes it to a file in the output directory with the name and count for that test. dctest was modified to pass that same name as the command line argument `output_dir` to the go test command:

```

1 test_cmd = ["go", "test", f"-run={test}", "-test.v" if verbose else "",
              f"-output_dir={output_dir / f"{test}_{i}"}", *extraargs]

```

The user's Go code can then read this command line argument and construct the appropriate path for where to write the GoVector log file:

```

1 var goLogPrefix = flag.String("output_dir", "", "Directory where output files
   should be written")
2 rf.Logger = govec.InitGoVector(fmt.Sprintf("raft_%v", me),
   fmt.Sprintf("%v/raft_%v", *goLogPrefix, me), config)

```

So, after `dstest` runs, each test will have a log file capturing the standard out, and a folder with the same name containing all the files created by the user's Go code (in this case, created by GoVector). The files cannot be loaded into DisViz directly, so the GoVector command will concatenate all the file content together into one `combined_logs.log` file, and then move the standard out log file for that test into the directory with the other log files, as seen in Figure 3-5.

## 4.2 Changes to GoVector

The GoVector changes largely consist of embedding a Zap Logger inside the GoLog struct, so that all the Zap logger's methods are immediately available to any GoLog object. The main object that user's of the GoVector library interact with and call methods on is the GoLog struct:

```

1 type GoLog struct {
2     ...
3     zapLogPrefix string
4     *zap.Logger
5     goLogWriteSyncer      *GoLogWriteSyncer
6     goLogCore              *GoLogCore
7     initialized           bool
8     preInitializationEntries []*ZapEntryInput
9     SugaredLogger          *zap.SugaredLogger
10    wrappedLogger          *zap.Logger
11    wrappedLoggerTwice      *zap.Logger
12    ...
13 }

```

The fields shown are some of the added fields needed to make GoVector use Zap internally.

### 4.2.1 Zap Primitives

The Zap Logger does not directly write to a file. Instead, it stores one or more Zap Core objects:

```
1 type Core interface {
2     // If the Level is enabled for this core
3     Enabled(Level) bool
4     // Adds structured context to the Core.
5     With([]Field) Core
6     // Determines whether the supplied Entry should be logged.
7     Check(Entry, *CheckedEntry) *CheckedEntry
8     // Write the log entry
9     Write(Entry, []Field) error
10    // Flushes buffered logs
11    Sync() error
12 }
```

The Core objects that Zap provides use a Zap Encoder to take a log entry and produce a byte array that a Zap WriteSyncer writes to the file:

```
1 type Encoder interface {
2     ObjectEncoder
3     Clone() Encoder
4     // EncodeEntry encodes an entry and fields,
5     // along with any accumulated
6     // context, into a byte buffer and returns it.
7     EncodeEntry(Entry, []Field) (*buffer.Buffer,
8         error)
9 }
10
11 type WriteSyncer
12     interface {
13         io.Writer
14         Sync() error
15     }
```

### 4.2.2 Logging with Zap

Using these primitives, we implement our own Core and WriteSyncer objects for a Zap Logger to use:

```

1  type GoLogWriteSyncer struct {
2      mu          sync.RWMutex
3      unbuf       zapcore.WriteSyncer
4      buf         *zapcore.BufferedWriteSyncer
5      manualBuffer bool
6      // always == unbuf or buf, under mu
7      active zapcore.WriteSyncer
8  }

```

```

1  type GoLogCore struct {
2      zapcore.Core
3      gv *GoLog
4  }

```

The `GoLogCore` embeds a Zap Core, so we only need to redefine methods to implement specialized behavior:

```

1  func (c *GoLogCore) Write(entry zapcore.Entry, fields []zapcore.Field) error {
2      c.gv.mutex.Lock()
3      defer c.gv.mutex.Unlock()
4      fields = c.gv.addMetadataFields(entry, fields)
5      if !c.gv.initialized {
6          return nil
7      }
8      return c.Core.Write(entry, fields)
9  }
10 func (c *GoLogCore) Check(ent zapcore.Entry, ce *zapcore.CheckedEntry)
    *zapcore.CheckedEntry {
11     if c.Enabled(ent.Level) {
12         return ce.AddCore(ent, c)
13     }
14     return ce
15 }

```

One important detail is that when a Zap Logger calls `Check()` on a core to see if that core will log the `Entry`, the core adds itself to the Zap `CheckedEntry` object, which contains a slice of all the `Core` objects that will write the `Entry`. Even though the implementation is the same as the library implementation, we must implement the `Check` method shown in order to add the `GoLogCore` object to the `CheckedEntry`, instead of the embedded `Core` object. This way, when the `CheckedEntry.Write` is called and the saved `Core` objects are iterated through, the `Write` method will be called on our `GoLogCore` object, which adds the vector clock metadata to the `Entry`:

```

1  func (gv *GoLog) addMetadataFields(entry zapcore.Entry, fields []zapcore.Field)
    []zap.Field {

```



```

2     if !gv.initialized {
3         gv.preInitializationEntries = append(gv.preInitializationEntries,
            &ZapEntryInput{entry: entry, fields: fields})
4         return fields
5     }
6     gv.tickClock()
7     return append(fields,
8         zap.String("processId", gv.pid),
9         gv.currentVC.ReturnVCStringZap("VCString"),
10    )
11 }

```

addMetadataFields uses the initialized boolean to determine if the InitGoVector method has been called on a GoLog object:

```

func (gv *GoLog) InitGoVector(processid string, config GoLogConfig, logfilenames
...string) {
    ...
    gv.prepareZapLogger(logfilenames)
    gv.initialized = true
    ...
}

```

InitGoVector uses the GoLogConfig to set the GoLog options and open the files that logs should be written to. When instrumenting the etcd repository to use GoVector to make DisViz-compatible logs (see section 5.1), it was necessary to create a GoLog object before knowing what the filename should be called. This allows the GoLog object to be created and passed down to necessary objects for logging before an appropriate filename is known for the log file to write logs to. If filenames are not known when a GoLog needs to be created, users can call UninitializedGoVector:

```

1 func UninitializedGoVector() *GoLog {
2     ...
3     goLogCore := &GoLogCore{Core: NewNopCore(), gv: goLog}
4     goLog.updateLoggers(zap.New(goLogCore, zap.AddCaller(),
        zap.AddStacktrace(goLog.level)))
5     ...
6 }

```

`updateLoggers` updates each of the four stored Zap `Logger` objects at once: the original, a sugared logger, and two loggers that skip a level in the stacktrace for when `GoLog.PrepareSend` or `GoLog.UnpackReceive` are called, to only include user level code in the caller and stacktrace information. Since a `GoLog` can be passed around and called on before files have been configured to write to, `addMetadataFields` will check if the logger has been initialized yet, and store any `Entry` objects to log later in `preInitializationEntries` once the `GoLog.InitGoVector` is called.

After the vector clock and pid is added to the fields of the log entry, the `GoLogCore` will use its embedded Zap `Core` to write to the file, which uses a Zap `Encoder` to prepare a byte string, and calls `GoLogWriteSyncer.Write`. `GoVector` previously supported toggling buffered logging, so we implement a `GoLogWriteSyncer` to keep this behavior. Zap offers a `BufferedWriteSyncer`, but this does not allow for toggling the buffering. `GoLogWriteSyncer` stores a Zap `WriteSyncer`, a Zap `BufferedWriteSyncer` that writes to the same `WriteSyncer`, and uses a read write mutex to keep track of which of those two should be written to.

Figure 4-1 shows a summary of what methods are called in what order when making a call to `GoLog.Info`. The yellow boxes show Zap methods, and the green boxes show `GoVector` methods.

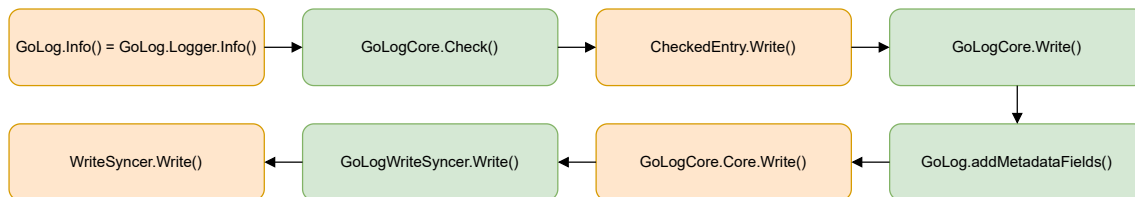


Figure 4-1: An abridged call chain for making a logging call with a `GoLog` object

### 4.2.3 Initialization

Once `GoLog.InitGoVector` is called, the embedded Zap `Logger` will be properly configured with our `GoLogCore` and `GoLogWriteSyncer`. Since a `GoLog` is only created

through `UninitializedGoVector`, the `GoLogCore` and Zap Logger will already be created:

```
1 baseWriteSyncer := zap.Open(filePaths...)
2 goLog.goLogWriteSyncer = &GoLogWriteSyncer{
3     unbuf: baseWriteSyncer, buf: nil, active: baseWriteSyncer,
4 }
5 core := zapcore.NewCore(
6     zapcore.NewJSONEncoder(encoderConfig), goLog.goLogWriteSyncer,
7 )
8 goLog.goLogCore.Core = core
9 // opts has options for adding caller and stacktrace
10 goLog.updateLoggers(goLog.Logger.WithOptions(opts...))
```

## 4.3 Changes to ShiViz

To run the existing code on the server side, we duplicate the JavaScript, use Node to run it, Express to set up a web server hosting the static site content, and ws to host a web socket for the frequent requests between the server and client [19, 26]. The webpage runs the same JavaScript code, but different changes are made to the server and browser JavaScript to properly use the web socket. The overall approach was to break up the monolithic system running entirely in the browser, and split components of it between the server and browser, trying to change the actual implementation details minimally.

### 4.3.1 Server

Using Node to run the JavaScript code originally meant to run in the browser requires the use of global proxy objects to make the browser dependencies not available in Node no-ops when called. The JavaScript code uses jQuery and D3, which are dependencies for manipulating the DOM and easily accessing elements on the webpage. It also uses the `window` and `document` objects available to JavaScript in the browser. Since none of these are available in Node, the best approach would be to refactor the code and remove all the uses of these objects. However, the JavaScript code is more than 14,000

lines over 55 files. For the scope of this thesis, and to reduce the chance of introducing bugs, our approach was to treat the code as modular by trying to make as few changes as possible to the original code, and instead move pieces of it around. With this in mind, we define global Proxy objects for all of the browser specific objects that are called:

```
1  ['$','d3','window','document'].reduce(  
2    (proxy, prop) => globalThis[prop] = proxy, new Proxy(() => {}, {  
3      // Intercepts property access, e.g. $.testing  
4      get(target, prop, receiver) {  
5        return receiver;  
6      },  
7      // Intercepts function calls, e.g. $()  
8      apply() {  
9        return $;  
10     }  
11  }));
```

This proxy will intercept any property accesses or function calls to return itself, so existing calls in the JavaScript like

```
1  $(".input input, .input textarea").on('input propertychange', function(e) {  
2    context.resetView();  
3  });
```

are no-ops and do not need to be edited in the server code.

The entry point to the server code is `disviz/index.js`, and after the global proxies are defined, we need to run all of the server JavaScript files in the global context, instead of how imports are usually handled in Node. The webpage defines all the JavaScript files with `<script>` tags in the HTML file, which will run all of the JavaScript in the shared global browser environment. Node import statements do not run the files in the global context. Instead, they run the file in a separate context and make any explicit export statements in the file available in the local context of the file that imported it. In order to mimick how the code runs in the browser, we use the built in `vm` module to run all 55 files in the same context:

```

1 [fileNames].forEach(filename => {
2   const filePath = path.join(__dirname, filename);
3   const code = fs.readFileSync(filePath, {encoding: 'utf8'} );
4   vm.runInThisContext(code, { filename: filePath });
5 });

```

With the environment properly set up, the web socket can properly respond to each type of message:

```

1 ws.on('message', async (event) => {
2   const message = JSON.parse(event);
3   switch (message.type) {
4     case "filePathRequest":
5       return await handleFilePathRequest(message);
6     case "slideWindowRequest":
7       return await handleSlideWindowRequest(message);
8     case "searchRequest":
9       return await handleSearchRequest(message);
10    case "nextResultRequest":
11      return await handleNextResultRequest(message);
12  }
13 });

```

## filePathRequest

A `filePathRequest` will read the file, use existing ShiViz functions to get a topological sorting of log events, then write that sorting back to a new file, from which it sends sections back to the browser. `AbstractGraph.getNodesTopologicallySorted` already existed in ShiViz, but had to be modified to produce a more accurate topological sort. It uses Khan's algorithm to produce a valid topological sorting, which maintains a list of unprocessed events that have no incoming edges from other unprocessed events, adding an event to the topological sort once all the events with incoming edges have been processed [29]. `getNodesTopologicallySorted` did this with a stack, processing events in a depth first style for a particular node until it reaches an event with a parent from another node. Using this sorting does not work well when viewing a small window of log events, because the server may send events only from one node in the system, when the visualization of all the events would show events from other

nodes at the same horizontal position. To give a more accurate view, we use the list of unprocessed events as a queue instead of a stack. This ensures that every event in the topological sort appears only after all events that would be rendered higher than it in the visualation are added to the sort.

### **slideWindowRequest**

This request is processed similarly to how a `filePathRequest` is processed, it just does not read a new file before sending a section of the log file back to the browser. The message request contains the start and ending offsets request by the browser, and it reads those offsets (adjusted to start and end on newlines) from the current file and sends that string back to the browser, which processes and visualizes only those events.

### **searchRequest**

To handle a `searchRequest`, the browser sends the query string entered by the user, and the server loads it into the `SearchBar` object then calls `SearchBar.query()`, following the same steps that the browser used to do with an event listener. To send an appropriate section of the file back to the browser, each log event has an added offset field containing the offset in the file where the start of that log event is. When the server gets the search results from the `MotifNavigator` object, it will contain the offset in the file where each search result starts. The server then reads a fixed section of the file starting from that offset and sends it to the browser.

### **nextResultRequest**

This request is processed similarly to how a `searchRequest` is processed, it just does not make a new query to the `searchBar` object, instead it reads a search result from the existing search, and sends a section of the file containing that result back to the browser. Once the browser receives the logs and processes them, it runs the same search query over the logs, so that the visualization can emphasize which events match the query.

### 4.3.2 Client

The changes to the JavaScript code running in the browser mostly consisted of making web socket requests where appropriate, but the more technical changes consisted of handling retry logic with sending web socket messages, and using an `IntersectionObserver` to automatically shift the section of the log file being visualized.

#### Web Socket

We define a `sendWithRetry` method on our web socket for the browser to send a message to the server:

```
1 ws.sendWithRetry = function (message) {
2   if (ws.readyState !== WebSocket.OPEN) {
3     return new Promise((resolve, reject) => setTimeout(() =>
4       ws.sendWithRetry(message).then(resolve, reject), reconnectDelay));
5   }
6   let { promise, resolve, reject } = Promise.withResolvers();
7   // display error to user if any occurs
8   promise.catch((reason) => {
9     const exception = new Exception(reason, true);
10    Shiviz.getInstance().handleException(exception);
11  });
12  message.id = generateRequestId();
13  // Save the resolver so we can call it when the response comes back.
14  pendingRequests[message.id] = { resolve, reject };
15  setTimeout(() => {
16    if (pendingRequests[message.id]) {
17      delete pendingRequests[message.id];
18      ws.sendWithRetry(message).then(resolve, reject);
19    }
20  }, requestTimeout);
21  ws.send(JSON.stringify(message));
22  return promise;
23 }
```

The `ws.onmessage` callback function can now read the `id` field from the message received from the server, and resolve the saved promise that `ws.sendWithRetry`

created. This allows the JavaScript in the browser to await the promise returned by `ws.sendWithRetry`, and not have to manually determine if the connection is active or the request times out. The recursive calls in the case of a timeout or a disconnected web socket will call the `resolve` or `reject` mutator of the original promise once they resolve, so the original promise won't be forgotten.

## Infinite Scroll

In order to make the visualization easier to use, we must automatically request more logs from the server once the user scrolls near the top or bottom of the visualization. We use the builtin `IntersectionObserver` API to register callback functions to call once a particular HTML element intersects the user's view-port [42]. We store this object in the `VisualGraph` object, and add sentinel elements in the HTML at the top and bottom of the visualization, calling `IntersectionObserver.observe` on them once the `VisualGraph` is created. We also do the same for the first and last events in the visualization. This allows the user to more easily navigate through the entire log file.

## Highlighting Search Events

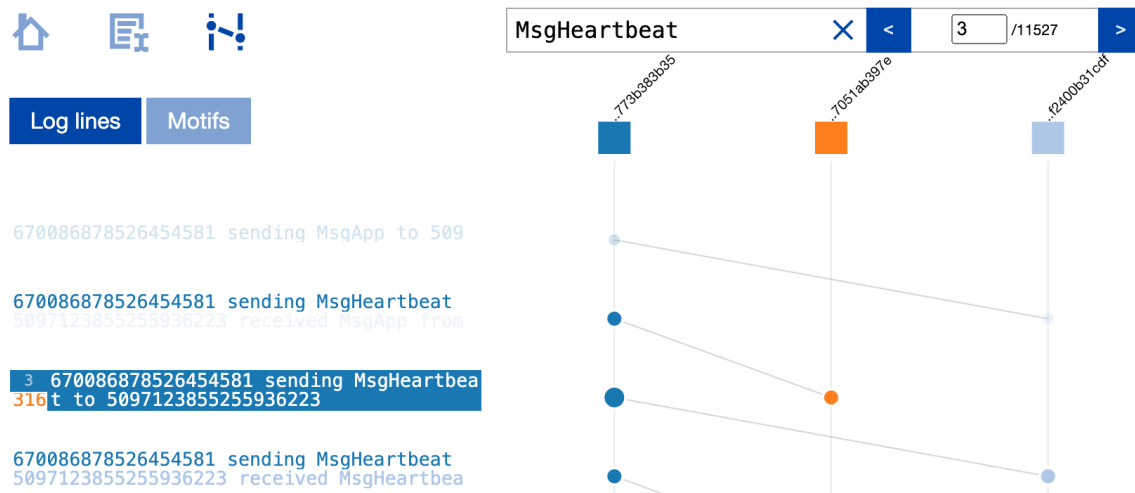


Figure 4-2: A highlighted event during a DisViz search



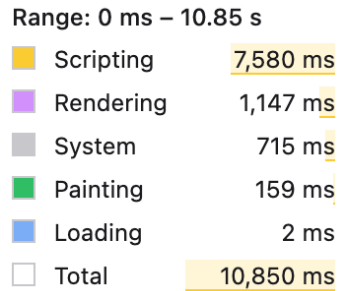
Highlighting search results for the user required fine grained control over the JavaScript event loop, specifically the use of `queueMicrotask`. When the client gets a response to a `searchRequest` or `nextResultRequest` and runs the same query over the logs from the server, it highlights the event that the user is viewing in the search results, see Figure 4-2. This is the same behavior when the user hovers over an event; that event and the corresponding log in the left panel are highlighted. The browser knows which event to highlight by reading the `lineToHighlight` field in the response from the server. Once the browser finds this event during the transformation to emphasize events matching the user's search, it needs to dispatch a mouse over event to the HTML element. However, since the visualization is redrawn during this process, the HTML element has not been created yet, so we cannot dispatch an event to it. Instead of modifying how and when HTML elements in the visualization are created, we can use `queueMicrotask`:

```
1 HighlightMotifTransformation.prototype.transform = function(model) {
2     var nodes = this.motifGroup.getNodes();
3     for (var i = 0; i < nodes.length; i++) {
4         var node = nodes[i];
5         var visualNode = model.getVisualNodeByNode(node);
6         const events = node.getLogEvents();
7         for (let i = 0; i < events.length; i++) {
8             const event = events[i];
9             if (event.getLogLine() === this.lineToHighlight) {
10                 var id = "#node" + visualNode.getId();
11                 queueMicrotask(() => $(id)[0].dispatchEvent(new
12                     MouseEvent("mouseover")));
13                 break;
14             }
15         }
16     }
```

The JavaScript event loop will process the callback function passed to `queueMicrotask` the next time the browser gives up control to the event loop, either by awaiting a promise, or finishing the execution in its call stack [32]. This allows the execution of the transformation to finish and the HTML elements to be created before the callback

runs. At this point we can successfully dispatch a mouse over event to the appropriate HTML element, so that the log event in the visualization is highlighted for the user.

### Improvement in latency



Range: 0 ms – 10.85 s		
Scripting		7,580 ms
Rendering		1,147 ms
System		715 ms
Painting		159 ms
Loading		2 ms
Total		10,850 ms

Figure 4-3: Time spent in the browser when rendering all log events for an execution

As shown in Figure 3-9, moving to the client server model provided a significant reduction in latency, a factor of about 3.4. However, since the original ShiViz JavaScript code was duplicated to run on the server, all the same internal objects and algorithms are still used to analyze a log file and construct a partial ordering of the events in the log. The speedup does not inherently come from running the code in Node instead of in the browser. The difference is that the code doesn't need to create any HTML elements like the JavaScript code does in the browser or call any of the jQuery methods. Every time a search is made or a node is collapsed by the user in the browser, the entire visualization is redrawn and all the HTML elements used in the visualization are recreated. This is likely what causes the significant latency as the number of log events increases. As shown in Figure 4-3, when viewing a log file with 12,000 events in DisViz and shifting the window to be the entire file, the time is mostly spent running the browser's JavaScript code, rather than rendering or painting frames. The main difference between the JavaScript running in the browser and in the server is that calls to jQuery or the document are not no-ops and can actually change or create HTML elements. Since the server does not need to create or modify any HTML elements, it can produce a partial ordering of log events much faster and send a small window back to the client. Now the large cost of creating the entire visualization is broken up

into smaller delays for the user as they view small sections of the log. Having two processes running (server and client) also helps decrease latency by sharing the total work that needs to be done.

# Chapter 5

## Case Studies

In order to evaluate how the changes to GoVector make it easier to use in an existing distributed system, we describe the changes required to use GoVector in etcd and a Raft implementation from 6.5840. Making these changes with the old GoVector would have been feasible with the Raft implementation and would require a similar number of changes, showing the changes do not worsen GoVector’s compatibility. Using the old GoVector in etcd would be infeasible due to how their RPC library works, showing the changes can improve GoVector’s compatibility.

### 5.1 etcd

Despite the more flexible GoVector API, instrumenting an existing system to produce vector clock logs is nontrivial. In etcd, there are many layers of objects that are created and configurations that are passed down through these layers during a test, sharing loggers at each step. etcd already uses Zap for its logging, which makes it easier to capture existing logs with GoVector [21].

etcd operations are submitted through an etcd `Client` struct and handled by an `EtcdServer` struct:

```

1  type Client struct {
2      Cluster
3      KV
4      Lease
5      Watcher
6      Auth
7      Maintenance
8      DisvizLogger *govec.GoLog
9      ...
10 }

1  type EtcdServer struct {
2      DisvizLogger *govec.GoLog
3      ...
4  }

```

Each of the embedded structs in the `Client` are interfaces providing the API users can call on. The embedded objects contain methods that make RPC requests through the auto-generated protobuf code. These RPC requests are sent to an `EtcdServer` object listening for requests. We add a `GoLog` object to the `Client` and `EtcdServer` structs and pass it down to the embedded objects during initialization. In each of the methods to send or receive an RPC, `GoLog.PrepareSend` and `GoLog.UnpackReceive` are called. While processing an RPC request, the `EtcdServer` needs to send RPCs between Raft nodes. This is done in `etcd/server/etcdserver/api/rafthttp/peer.go` by `peer.send`, where we now call `GoLog.PrepareSend` and `GoLog.UnpackReceive` before and after sending the RPC to the other raft nodes.

In order to get the `GoLog` object in each of these locations to properly log sending and receiving messages, we must carefully configure it at the beginning of a test, and pass it down as each of the `etcd` structs are defined. A cluster is created with:

```

var zapLogPrefix = flag.String("output_dir", "", "Directory where output files
    should be written")
clus := integration2.NewCluster(t, &integration2.ClusterConfig{Size: 3,
    ZapLogPrefix: os.Getwd() + "/" + *zapLogPrefix})

```

This cluster is created in `etcd/tests/framework/integration/cluster.go`, where a `Member` object contains the `Client` and `EtcdServer` for each node in the cluster. The client is created with a config containing the `GoLog` to use, and this config is passed to a factory function to initialize a `Client`, and pass that `GoLog` to each of the embedded interfaces in the `Client`.

Configuring the `EtcdServer GoLog` is more difficult, which requires using an

uninitialized GoLog. The GoLog objects write to files named with `Member.Id()` to identify the node, but this id is configured during the initialization of an `EtcServer`, where we must pass a GoLog into in order for it to make its way into the `peer` struct. This makes using an uninitialized GoLog useful:

```

1 func (m *Member) Launch() error {
2     ...
3     m.DisvizServerLogger = govec.UninitializedGoVector()
4     m.Server = etcdserver.NewServer(m.ServerConfig)
5     config := govec.GetDefaultZapConfig()
6     config.ZapLogPrefix = m.ZapLogPrefix
7     m.DisvizServerLogger.InitGoVector(fmt.Sprintf("etcd_server_%v", m.ID()),
8         config, fmt.Sprintf("etcd_server_%v", m.ID()))
9     ...
10 }

```

The GoLog in the `Member` struct is passed through: `Member -> EtcServer -> Transport -> peer` in the call chain `integration2.NewCluster -> Member.Launch -> etcdserver.NewServer -> Transport.AddPeer -> rafthttp.startPeer`.

This completes logging when messages are sent between nodes, but we must also capture when local events are logged. Since Zap is already used for this in `etcd`, and we have implemented our own Zap Core object, we define `WrapBaseLogger` in `GoVector`:

```

1 func (gv *GoLog) WrapBaseZapLogger(baseLogger *zap.Logger, opts ...zap.Option)
2     *zap.Logger {
3     opts = append(opts, zap.WrapCore(func(core zapcore.Core) zapcore.Core {
4         return zapcore.NewTee(core, gv.Logger.Core())
5     }))
6     return baseLogger.WithOptions(opts...)
7 }

```

`gv.Logger.Core()` returns our `GoLogCore`, which adds our metadata fields. Now given an existing Zap Logger, we can return a new Logger that uses `zapcore.NewTee` to write logs to the existing core as normal, but also write logs to our `GoLogCore`. With one carefully placed call early in initialization:

```

1 func newClient(cfg *Config) (*Client, error) {
2     ...

```

```

3     client.lg = client.DisvizLogger.WrapBaseZapLogger(client.lg, zap.AddCaller(),
        zap.AddStacktrace(client.lg.Level()))
4     ...
5 }

```

We automatically capture all of the local log events, and similarly for the `EtcdServer`. This saves a significant amount of time doing this manually throughout the repository, and reduces opportunities for bugs. In total, making these changes in etcd amounted to around 5800 lines changed in auto-generated code from protobuf, and around 900 lines changed manually over 31 files.

## 5.2 6.5840 Raft

The implementation effort required to instrument an existing system with GoVector is directly correlated with how large the system is. An implementation of Raft that I wrote for the MIT class 6.5840 is significantly less intricate than the etcd repository, and using GoLog is more straightforward. All tests are performed with locally running Raft nodes, so a channel-based RPC abstraction is used that does not actually use the network to send messages. All Raft nodes have a list of peer `ClientEnd` structs they can send messages to on a stored channel with a `Call` method. Since a relatively smaller number of RPCs are possible, it is feasible to define wrapper methods and call these methods everywhere that RPCs are sent in the Raft implementation. We can also use the old GoVector pattern of wrapping the payload to send in the GoVector payload, since the smaller number of RPCs makes this a more straightforward approach:

```

1 func (rf *Raft) CallWrapper(peer int, svcMeth string, args interface{}, reply
    interface{}) bool {
2     result := rf.peers[peer].Call(svcMeth, rf.PrepareSend(svcMeth, peer, args),
        args, rf.me)
3     if result.Ok {
4         rf.UnpackReceiveResponse(svcMeth, peer, result.Reply, reply)
5     }
6     return result.Ok
7 }

```

In the RPC implementation when a `Call` is being processed, we can call these Raft methods which call the corresponding GoLog methods:

```
1 // Service uses reflect to dynamically call any defined RPCs in the raft
   implementation
2 func (svc *Service) dispatch(methname string, req reqMsg) replyMsg {
3     method := svc.methods[methname]
4     args := reflect.New(req.argsType)
5     svc.methods["UnpackReceive"].Func.Call([]reflect.Value{svc.rcvr,
6         reflect.ValueOf(methname), reflect.ValueOf(req.from),
7         reflect.ValueOf(req.args), args})
8     replyv := reflect.New(method.Type.In(2).Elem())
9     method.Func.Call([]reflect.Value{svc.rcvr, args.Elem(), replyv})
10    out := svc.methods["PrepareSendResponse"].Func.Call([]reflect.Value{svc.rcvr,
11        reflect.ValueOf(methname), reflect.ValueOf(req.from), replyv})
12 }
```

Now the call chain for every RPC sent will be:

`Raft.CallWrapper -> Raft.PrepareSend -> ClientEnd.Call ->`

`Raft.UnpackReceive -> Raft.PrepareSendResponse -> Raft.UnpackReceiveResponse,`

and the request/response for each node in an RPC will be logged. For capturing local logs, the code calls `Raft.Debug`, and we can refactor it to call our GoLog:

```
1 func (rf *Raft) Debug(topic logTopic, format string, a ...interface{}) {
2     time := time.Since(debugStart).Microseconds() / 1000
3     prefix := fmt.Sprintf("%d %v ", int64(time), string(topic))
4     rf.Logger.SugaredLogger.Logf(zapcore.InfoLevel, prefix+format, a...)
5     if topic == DError {
6         panic("got an error")
7     }
8 }
```

The tests for the Raft implementation frequently kill a Raft node and bring it back online by initializing a new `Raft` struct with the same configuration. This introduces a problem when the GoLog object has already been writing vector clock logs to the file. When the new `Raft` struct initializes its GoLog object with the same filename, it will begin appending to the file with a reset vector clock. In order for the logs to be valid, the vector clock for node must increase monotonically by 1, which this would



violate. When initializing a `GoLog`, if the filename already exists, we check to read the last line and try to parse the vector clock from it. If it succeeds, then we initialize the `GoLog` with the vector clock read from the existing file, so we can continue increasing the clock by 1 on each line.

In total, my original Raft implementation has around 1800 lines of go code, and the changes described here required around 80 lines of code over 3 files, around 10 times less than what was required in `etcd`, as seen in Table 5.1. This takes significantly less time and is less bug prone than making the same changes on a larger system like `etcd`.

<b>System</b>	<b>Lines Changed</b>	<b>Files Modified</b>
Raft Implementation	80	3
<code>etcd</code>	900	31

Table 5.1: Code changes required to integrate `GoVector`

# Chapter 6

## Evaluation

In order to evaluate how DisViz helps with debugging, we use my Raft implementation from 6.5840 to debug correctness and liveness bugs. For Leader Election, and Committing Logs, we introduce bugs and step through the process a user would take to find the bug. For Snapshotting, we look at an existing bug where the cause is unknown. We show that the logging flexibility provided by GoVector allows for easily adding new information to logs and viewing it in DisViz. The reduced latency provided by the client server model makes the visualizations faster to interact with and provides the same features as ShiViz (e.g. search).

The tests we run are provided by the 6.5840 course staff and are the same used when I first implemented this system.

### 6.1 Leader Election

#### Correctness

We introduce a bug in the `Raft.RequestVote` RPC handler, when a Raft node is preparing a response after receiving a request for a vote from another node. The bug allows for granting a vote more than once in a term:

```
1 func (rf *Raft) RequestVote(args *RequestVoteArgs, reply *RequestVoteReply) {  
2     ...
```

```

3      // Correct code: if args.Term < rf.currentTerm || (rf.votedFor != -1 &&
      rf.votedFor != args.CandidateId) {
4      if args.Term < rf.currentTerm {
5          reply.VoteGranted = false
6      } else {
7          ...
8      }

```

We first run each test for leader election to see what might be failing with:

```
ptest -n 10 -p 10 -v TestInitialElection3A TestReElection3A TestManyElections3A
```

TestReElection3A and TestManyElections3A each fail 4 times due to a panic, so we focus on TestManyElections3A. This test makes a cluster with 7 servers and waits for a leader to be elected. For 10 iterations, three random servers are disconnected, the test makes sure there is still a leader, and then those three are reconnected.

In the captured output for one of the failures, we see the test fails due to logging an error, which causes a panic:

```

1 func (rf *Raft) AppendEntries(args *AppendEntriesArgs, reply *AppendEntriesReply)
   {
2     ...
3     // mark heartbeat
4     if rf.state != follower {
5         rf.Debug(DError, "S%v %v, APPEND ENTRIES FROM UP TO DATE LEADER BUT WE'RE
           NOT A FOLLOWER. Current term: %v\n", rf.me, rf.state, rf.currentTerm)
6     }
7     ...
8 }

```

This cluster is in a split brain situation when this panic occurs. The panicking node thinks it is leader, but it was sent logs from a node that thinks it is the leader with the same term number, which should never happen during normal operation. When viewing the logs for this failure in DisViz, we jump to the end to see the log entry for this error in Figure 6-1, which belongs to the raft\_1 node.

The visualization ends with many logs from this node and no connections to other nodes, so it seems this node became disconnected before being reconnected and

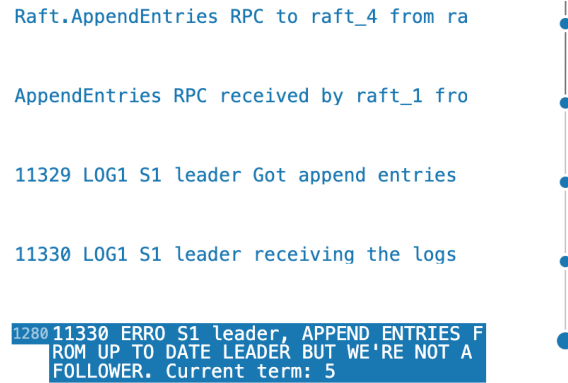


Figure 6-1: Last log from TestManyElections3A failure

receiving an append entries from another node. To find which node sent an append entries to this node, we search for the last occurrence of "Raft.AppendEntries RPC to raft\_1". The node with name raft\_3 sent this log. To find when raft\_3 became a leader we can search for this node's log in the visualization (see Figure 6-2):

```

1 func (rf *Raft) RequestVote(args *RequestVoteArgs, reply *RequestVoteReply) {
2     ...
3     rf.Debug(DVote, "S%v %v, granting for %v in term %v\n", rf.me, rf.state,
4         args.CandidateId, rf.currentTerm)
5     ...
6 }

```

The search functionality supports regular expressions, so searching for `event=/granting for . in term 5/`, we find seven occurrences, where nodes raft\_5, raft\_0, and raft\_4 all grant votes for raft\_1 and then raft\_3 in term 5. Looking at the code for handling a RequestVote RPC, we have checks for when the term in the RPC is less than or greater than the current term:

```

1 func (rf *Raft) RequestVote(args *RequestVoteArgs, reply *RequestVoteReply) {
2     ...
3     if args.Term > rf.currentTerm {
4         rf.convertToFollower(args.Term)
5     }
6     if args.Term < rf.currentTerm {
7         reply.VoteGranted = false
8     } else {

```

```

9         // grant vote. We must be a follower now
10     ...
11 }

```

Before the handler grants a vote, the handler must make sure that it has not voted for anyone in this term already, which identifies the bug. Using the visualization made it quicker to gain context about the communication between the nodes in the cluster, which helped track down the bug.

## Liveness

For a liveness bug relating to leader election, we flip a boolean to never send a yes vote back in a `RequestVote` RPC:

```

1 func (rf *Raft) RequestVote(args *RequestVoteArgs, reply *RequestVoteReply) {
2     ...
3     // Correct code: reply.VoteGranted = true
4     reply.VoteGranted = false
5     ...
6 }

```

Running the same `pptest` command as before, we see that no tests are passing. We will focus on the simplest of these tests `TestInitialElection3A`, which starts a cluster of three nodes and checks that a leader is elected and all the terms match for the nodes. The captured output shows that the test fails with the error message "expected one leader, got none." From Figure 6-2, we see that the node `raft_2` requests a vote and gets responses from `raft_0` and `raft_1`, and both nodes have the logs "granting for 2 in term 3". However, the responses received by `raft_2` have `VoteGranted` set to false, which identifies the bug in `RequestVote`. These log files are short, which shows that using `DisViz` can be helpful regardless of the log file size.

```

81 4887 VOTE S0 follower, granting for 2
4887 in term 3

4887 PERS S0 follower Successfully persi
4887 VOTE S1 follower, granting for 2 in

RequestVote RPC Response to raft_2 from
4887 PERS S1 follower Successfully persi

RequestVote RPC Response to raft_2 from
Raft.RequestVote RPC Response received b

4887 VOTE S2 candidate Got requestVote r

Raft.RequestVote RPC Response received b

```

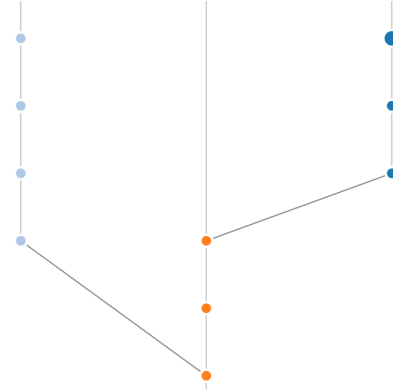


Figure 6-2: Logs from TestInitialElection3A failure

## 6.2 Committing Logs

### Correctness

In order to grant a vote for a peer, that peer's log must be at least as up to date as the node that receive the `RequestVote` RPC. We introduce a bug that comments out this check:

```

1 func (rf *Raft) RequestVote(args *RequestVoteArgs, reply *RequestVoteReply) {
2     // if latestTerm > args.LastLogTerm || (latestTerm == args.LastLogTerm &&
3         rf.getLogIndex(len(rf.log)-1) > args.LastLogIndex) {
4         //     reply.VoteGranted = false
5         //     return
6         // }
7     }

```

This causes every run of `TestRejoin3B` to fail, which makes a cluster of 3 nodes, and tests the rejoin of a partitioned leader. The test fails because after the partitioned leader rejoins the cluster, the cluster fails to reach agreement and commit a newly submitted log. Since it will be helpful to know when a node becomes disconnected and reconnected, we can add logging statements in the testing code with:

```

1 func (cfg *config) disconnect(i int) {
2     ...
3     raft := cfg.rafts[i]
4     raft.Logger.Log(zapcore.InfoLevel, "Disconnecting this node",
5         zap.Object("self", raft))

```

```

5     ...
6 }

```

The Zap API makes it easy to define how to log arbitrary objects using `zap.Object`. The object to log just needs to implement this method:

```

1 func (rf *Raft) MarshalLogObject(enc zapcore.ObjectEncoder) error {
2     enc.AddInt("me", rf.me)
3     enc.AddString("state", string(rf.state))
4     enc.AddInt("logLength", len(rf.log))
5     enc.AddInt("logStart", rf.getLogIndex(0))
6     return nil
7 }

```

And Zap will use this method when encoding the log event. Figure 6-3 shows how this is displayed in DisViz.

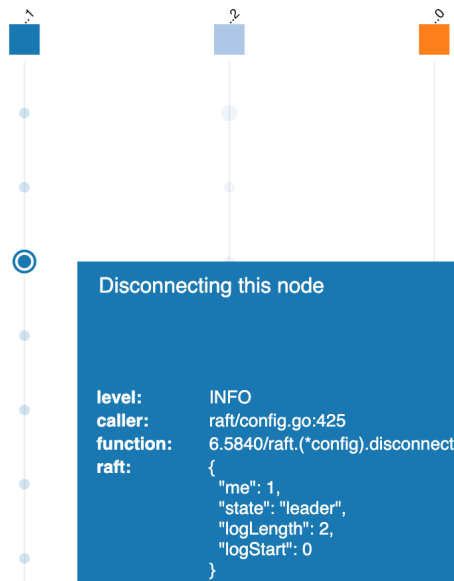


Figure 6-3: Displaying a raft object in DisViz

Searching for when `raft_1` becomes reconnected, and viewing the logs after that, we see that `raft_2` was the leader and committed an event with `raft_0` before also being disconnected. Between the disconnection of `raft_2` and the reconnection of `raft_1`, we see many logs from both nodes that think they are the leader (in different terms) and are trying to send RPCs to other nodes, but no lines connect those events

to other nodes, so no communication is happening. After raft\_1 reconnects, it sends an **AppendEntries** to raft\_0, who responds with a failure because the raft\_1 term is out of date (raft\_2 became leader in a new term after raft\_1 is disconnected). Figure 6-4 shows how DisViz makes this communication easy to see.

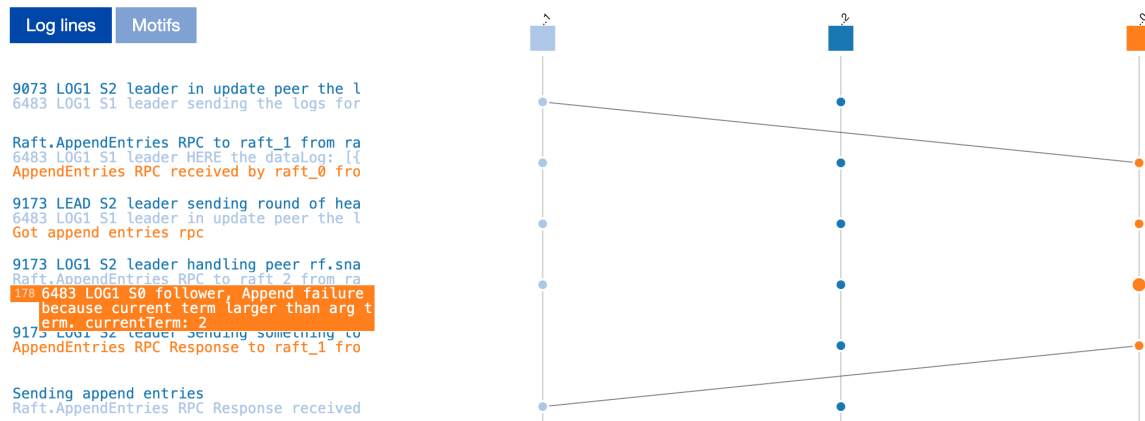


Figure 6-4: AppendEntries failure from raft\_1 to raft\_0

Following the events down, raft\_1 begins an election and gets a vote from raft\_0, completing a majority and becoming a leader. raft\_1 begin sending **AppendEntries** RPCs to raft\_0, but they all fail because raft\_0 has an event committed that raft\_1 does not have, so it never accepts new logs from raft\_1. This tells us that raft\_0 should not have granted a vote to raft\_1, which leads us to add the check for up to date logs for an election candidate.

## Liveness

To introduce a liveness bug, we make a small change to the code that applies newly committed messages to a node's state machine once a leader receives an **AppendEntries** RPC response:

```

1
2 // Correct code: rf.commitIndex > rf.lastApplied {
3 for rf.commitIndex < rf.lastApplied {
4     // send log to apply channel
5 }

```



The diagram illustrates a Raft log replication and conflict resolution scenario between two nodes, S (blue) and F (orange).

**Log Entries:**

- Node S (blue):**
  - 1737 LOG1 S0 leader in update peer the l
  - 1737 LOG1 S1 follower crafting reply. pr
  - 1737 LOG1 S1 follower successfully updat
  - 1737 LOG1 S2 follower crafting reply. pr
  - 1737 TERM S1 follower updating commit
  - 1737 TERM S1 follower and sending on app
  - 1737 TERM S2 follower updating commit in
  - 1738 LOG1 S0 leader got append entries r
  - 1738 CONFLICT S0 leader append resp. fro
- Node F (orange):**
  - 1737 LOG1 F0 leader in update peer the l
  - 1737 LOG1 F1 follower crafting reply. pr
  - 1737 LOG1 F1 follower successfully updat
  - 1737 LOG1 F2 follower crafting reply. pr
  - 1737 TERM F1 follower updating commit
  - 1737 TERM F1 follower and sending on app
  - 1737 TERM F2 follower updating commit in
  - 1738 LOG1 F0 leader got append entries r
  - 1738 CONFLICT F0 leader append resp. fro

**Sequence of Events:**

- Both nodes S and F start with log entry 1737 LOG1 S0/F0 (leader in update peer the l).
- Node S sends **Raft.AppendEntries RPC** to raft\_2 from ra.
- Node F receives **AppendEntries RPC** received by raft\_1 fro.
- Node S receives **Got append entries rpc**.
- Node F receives **AppendEntries RPC** received by raft\_2 fro.
- Node S sends **1737 LOG1 S1 follower crafting reply. pr**.
- Node F receives **Got append entries rpc**.
- Node S sends **1737 LOG1 S1 follower successfully updat**.
- Node F receives **1737 LOG1 S2 follower crafting reply. pr**.
- Node S sends **1737 TERM S1 follower updating commit**.
- Node F receives **1737 TERM S1 follower updating commit**.
- Node S sends **1737 TERM S1 follower and sending on app**.
- Node F receives **1737 TERM S2 follower updating commit in**.
- Node S sends **AppendEntries RPC Response to raft\_0 fro**.
- Node F receives **1737 TERM S2 follower and sending on app**.
- Node S sends **Raft.AppendEntries RPC Response received**.
- Node F receives **AppendEntries RPC Response to raft\_0 fro**.
- Node S sends **1738 LOG1 S0 leader got append entries r**.
- Node F receives **1738 LOG1 F0 leader got append entries r**.
- Node S sends **1738 CONFLICT S0 leader append resp. fro**.
- Node F receives **1738 CONFLICT F0 leader append resp. fro**.

65

## 6.3 Snapshotting

The previous sections introduced bugs and showed how DisViz can be used. This section tracks down an unknown bug that is present in my Raft implementation. Of the seven snapshotting tests, the unknown bug causes `TestSnapshotInstall3D` and `TestSnapshotInstallUnreliable3D` to exhibit failures. Both tests perform similar actions, disconnecting and crashing nodes in a 3 node cluster over a number of iterations, waiting for the cluster to commit logs and making sure they do snapshots so their logs do not grow too large.

Focusing on `TestSnapshotInstall3D`, it fails because after reconnecting a node, the cluster cannot reach an agreement for a new log entry. Since we are focusing on snapshots, the hypothesis is the reconnected node incorrectly applies a snapshot to catch up, giving it an incorrect log history that does not match the other nodes. This bug happens infrequently; using `pctest`, we observe 2 failures during a run of 100 tests. In DisViz, we see that 6 leaders are elected during the test ending with `raft_2` in term 7. Searching for the logs we added in section 6.2, a node is disconnected then reconnected 8 times.

The test ends with many `AppendEntries` RPCs from `raft_2` to `raft_0` and `raft_1`. `raft_0` responds with success each time because their logs are up to date with the leader's logs, see Figure 6-6.

`raft_1` responds with a failure each time because

```
rf.log[args.PrevLogIndex-rf.startingIndex].Term != args.PrevLogTerm
```

So the term of the log preceding the new logs does not match between `raft_2` and `raft_1`. Since we suspect the bug is related to a reconnected node catching up, we search for the last node to be reconnected, `raft_2`. After it is reconnected, it receives an `InstallSnapshot` RPC, then starts an election and becomes leader. All of the `AppendEntries` failures from `raft_1` start after `raft_2` processes the `InstallSnapshot` RPC, so there is more evidence that the bug is related to processing a snapshot.

The next thing we do is add logging statements to show the node's logs and fields related to snapshotting. Utilizing `zap.Array` similar to `zap.Object`, we write:

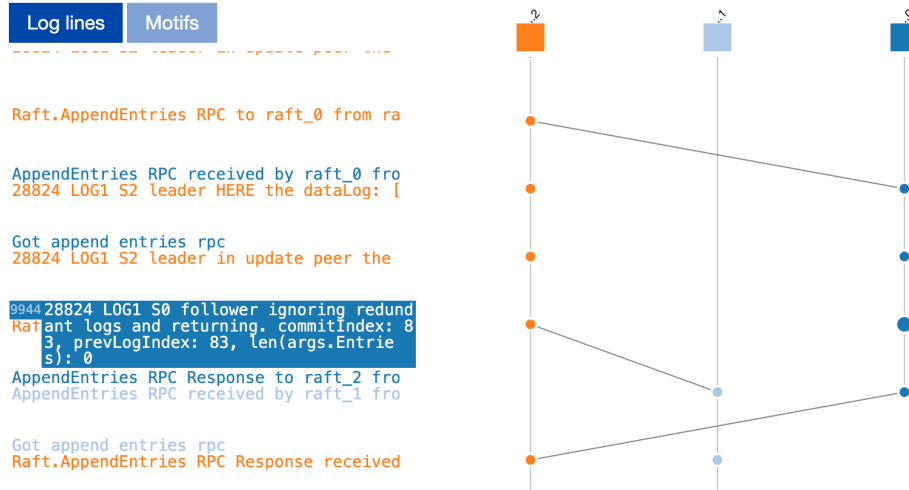


Figure 6-6: Raft follower ignoring redundant logs from leader

```

1  type RaftLogs []*RaftLog
2  func (logs RaftLogs) MarshalLogArray(enc zapcore.ArrayEncoder) error {
3      for _, entry := range logs {
4          enc.AppendObject(entry)
5      }
6      return nil
7  }
8  func (r *RaftLog) MarshalLogObject(enc zapcore.ObjectEncoder) error {
9      enc.AddInt("term", r.Term)
10     enc.AddReflected("command", r.Command)
11     return nil
12 }
13 rf.Logger.Info("sending install snapshot", zap.Array("logs", rf.log),
    zap.Object("me", rf))

```

We add the logging call before sending and after receiving an `InstallSnapshot` or failed `AppendEntries` response. Running 50 more tests and viewing a failure in `DisViz`, we find that the new execution ends with `raft_0` as a leader, getting failed `AppendEntries` responses from `raft_1` for the same reason as before.

Searching for the logging statement we added at the end of processing an `InstallSnapshot`, we see that earlier in the log, `raft_0` is leader, disconnects, `raft_1` becomes leader, then `raft_0` reconnects and gets an `InstallSnapshot` from `raft_1`. Figure 6-7 shows the `raft_0` state after processing the snapshot, as well as the argument and reply to

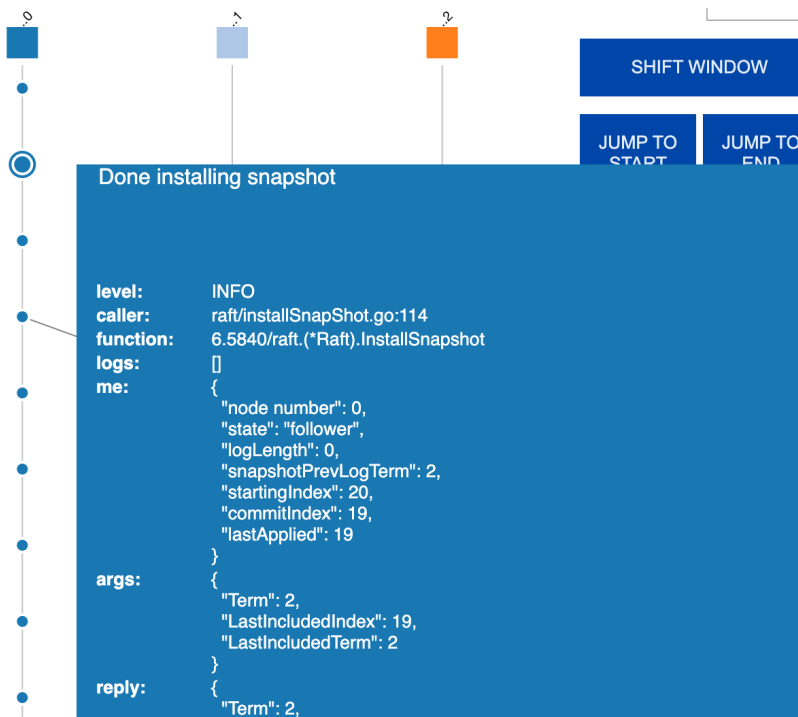


Figure 6-7: Raft follower after processing an InstallSnapshot RPC

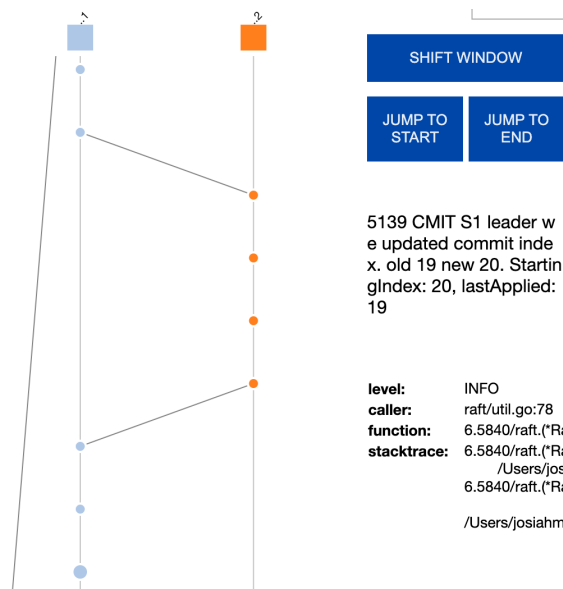


Figure 6-8: Raft leader updating its commit index

the RPC. It seems that the snapshot was processed correctly and raft\_0 is up to date. However, if we look at the log events when raft\_1 sends the InstallSnapshot (see

Figure 6-8), it receives an `AppendEntries` from `raft_2` and updates its commit index. The line exiting the left side of the figure is the `InstallSnapshot` sent to `raft_0`. But the log events from `raft_2` show that it treated the log entry as redundant, thinking it already had the entry, and did not bother to process it. Viewing the preceding events from `raft_2`, it does a snapshot up to index 19, leaving its log length as 0 and starting index as 20. When `raft_1` sends `raft_2` another log entry, it should not be treating that entry as redundant as it does not have it. Looking at the code, we find the culprit:

```
1 func (rf *Raft) AppendEntries(args *AppendEntriesArgs, reply *AppendEntriesReply)
2 {
3     ...
4     if rf.commitIndex >= lastNewIndex || rf.startingIndex >= lastNewIndex {
5         reply.Term = -1
6         rf.Debug(DLog, "%v %v ignoring redundant logs and returning", rf.me,
7             rf.state)
8         return
9     }
10 }
```

The check for the starting index is unnecessary and incorrect. It would be correct to write `rf.startingIndex > lastNewIndex`, but the starting index should be at most 1 greater than the commit index, when the length of the log is 0. Therefore,  $\text{rf.commitIndex} \geq \text{lastNewIndex} \implies \text{rf.startingIndex} > \text{lastNewIndex}$ , and we can remove the second condition entirely, fixing the bug.

This was difficult to find because the bug trickled down for a while without being caught. The "ignoring redundant logs" log event is 23% of the way from the start of the log file, so trying to piece together what happened from the error at the end would be quite difficult from only reading the log file. Using the original ShiViz and GoVector would be more feasible than reading the log file, but the flexibility offered by the new GoVector API with Zap was helpful for adding more logging information, and the reduced latency in DisViz made it less cumbersome to interact with the visualization. The log files for these tests failures have around 4000 – 9000 logs, which

gives a latency in ShiViz around  $2.5 - 5$  seconds to load the visualization and to perform searches, compared to around  $1.5 - 2$  seconds in DisViz. Due to the large number of event listeners and HTML elements in ShiViz when viewing the whole log file, user interactions like clicking on a log event have a noticeable delay as well, on the order of hundreds of milliseconds.

# Chapter 7

## Related Work

There is a wide array of existing work in taking the trace or logs from a distributed system execution and helping the programmer glean useful information from it. These tools vary in their goals (e.g. debugging, performance optimization), presentation, and whether they can operate on a live system, but share a commonality of helping the programmer visualize the complex distributed system.

GridMapper takes logs from a live system and uses ip information to place nodes on a map of the world, providing a clear depiction of how the distributed system is physically laid out in the world [2]. Borysenkov et. al used clustering techniques on the logs and a Time Curve visualization in an offline program to give a high level representation of how the logs change throughout the execution [17, 11]. The ViSiDiA tool can support a live debugging session and visualization of a distributed system with a node-edge graph, however it is platform specific and requires the implementation to be in Java to use their API [1]. OverView is an Eclipse IDE plugin that allows for live or offline visualization of a distributed system, but it is also platform specific to Java [20].

Oddity is another debugging tool that helps the programmer debug a live, running system [41]. It requires a specific system architecture of deterministic event handlers to process incoming messages, update node state, and construct replies. It also requires the user to implement a shim around the communication between nodes, so that Oddity can control how the system progresses. A server uses this shim to control the

system and send information to the client webpage. Through this, Oddity allows the user to decide when events are fired and messages are received by nodes, as well as allows the user to time travel backwards by replaying all the events that happened up until the chosen point. The interface is designed for 10 or fewer nodes, and having to manually choose the order of events that are processed on each node would be tedious for long running interactions. Oddity provides useful functionality to the user, but the architecture requirements and nontrivial shim implementation present barriers to its use in a real-world distributed system like etcd.

Porcupine is a linearizability checker for distributed systems written in Go, used by etcd as well as 6.5840 at MIT [4]. The program takes a model specification for initialization and valid step updates to the state, and checks whether a submitted history is linearizable with respect to the model. It can visualize the history with a partial linearization, showing linearization points and which operations violated linearizability. Jepsen does safety research for distributed system, and has a similar tool called Knossos that verifies linearizability and generates a visualization for debugging [27]. They also have a tool called maelstrom for writing toy implementations of distributed systems which generates timeline visualizations, timeseries graphs, and lamport diagrams.

Each of these systems is useful for their defined goal and lessons can be taken from their approach to digesting the log information, but we desire a tool that is primarily geared for fine-grained analysis of the execution of a distributed system, in an offline manner. While a program like Oddity would be useful, we have chosen to focus on the offline processing of log information because of its potential to be easily integrated into existing distributed system implementations and low overhead.

For improving log visualization in general, not specifically in the context of distributed systems, the Log-it tool gives the programmer granular control over the presentation of the logs [28]. Building a tool specifically for distributed systems is needed due to the unique nature of these types of programs, but it should be a goal of the tool to mimic the control Log-it gives to the programmer, allowing them to change and direct how the log information is collected and presented to them.



# Chapter 8

## Conclusion

In conclusion, this thesis presents DisViz as an extension of ShiViz, GoVector, and dstest to help with debugging distributed systems. ptest provides a way to run parallel tests and collect all testing output. GoVector was refactored to use Zap and provide a more flexible API for logging key value pairs. ShiViz was moved from a monolithic client side design to a server client model. The server does the heavy lifting of processing the entire log file of a given distributed system execution, and it sends small sections of the log file to the client to display. Using a time space diagram makes it easier to understand how nodes in the system are interacting, and a server client model allows the tool to work well with log files on the order of tens of thousands of events.

### 8.1 Future Work

In developing DisViz, several avenues for improvement became apparent. These ideas are discussed below.

#### 8.1.1 Change the Server Side Language

JavaScript was used on the server side so the original ShiViz code that ran in the browser could be reused. Using a language such as Go or C++ would likely lead to

better performance. This is because JavaScript is a JIT compiled language, generally runs in a single thread, and does not have static typing to enable more optimizations. However, this change would require converting around 14,000 lines of JavaScript. This may be feasible using LLMs, but would need thorough testing.

### **8.1.2 TypeScript**

Using TypeScript instead of JavaScript on the server and client would lead to better code quality and increased safety from bugs. TypeScript is a superset of JavaScript that adds static typing. It is compiled down to JavaScript before execution and the type information is discarded. This would not require many line changes to achieve, and would make it easier to identify bugs related to static types.

### **8.1.3 JavaScript Runtime**

One simple change to achieve better performance on the server side is to experiment with different JavaScript runtimes instead of Node. Bun and Deno are two possible replacements for Node [37, 18]. They are relatively recent JavaScript runtimes that have different performance characteristics, so one of them may provide a performance improvement without having to change any of the implementation code.

### **8.1.4 Support Clustering**

During the evaluation of DisViz, the feature of clustering was not needed to debug distributed system executions, so support was not added for it. The case studies discussed in chapter 5 did not need to include multiple executions within one log file. Clustering would use these multiple executions and display them side by side in the visualization. In order to fully support all the features of ShiViz, the server would need to be changed to detect all the delimiters, and send corresponding portions of each execution to the user.

### 8.1.5 File API

All the log file processing on the server side is still done entirely in memory, reading the entire file and constructing a topological sorting in memory before writing that sorting back to disk. So, the maximum size of the log file DisViz can process is limited by the memory available to the Node process running. A related issue arises when the file the server wants to process is not located on the file system of the device running the server. To solve both issues, a file API could be designed to provide functions for the server to retrieve and store log events. Different implementations of the API could then allow for the file to be accessed from another device, or for external memory techniques to be used to handle files that do not fit in the device's memory.

### 8.1.6 Remote Hosting

Since the server hosts the static site content as well as the web socket, the user opening DisViz on the browser no longer has to store anything related to DisViz on their device. The server and client devices can be different, so a CI/CD pipeline could automatically detect test failures after using ptest, start the DisViz server, and provide a link for users to view the logs from the test failure. For example, the etcd repository is hosted on GitHub, so implementing this with GitHub Actions can help speed up the debugging of test failures. This also allows teams of developers to view the same log file and collaborate on debugging.

As described in subsection 4.3.1, running the ShiViz code in the server required running all the JavaScript files in the same context, so they can all share and mutate their state. This would be an issue for multiple clients connecting to the same server, so one solution would be to create a new fresh context for each client that reruns all the JavaScript files. Another solution is to change the JavaScript code to not use singleton objects for the active visualization, search, etc, so the same context can be shared when the server responds to client requests.

# Bibliography

- [1] Cédric Aguerre, Thomas Morsellino, and Mohamed Mosbah. “Fully-Distributed Debugging and Visualization of Distributed Systems in Anonymous Networks”. In: *7th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications*. Ed. by P. Richard et al. Rome, Italy: INSTICC, Feb. 2012, pp. 764–767. URL: <https://hal.science/hal-00697093>.
- [2] William Allcock et al. “GridMapper: A Tool for Visualizing the Behavior of Large-Scale Distributed Systems”. In: *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*. HPDC '02. USA: IEEE Computer Society, 2002, p. 179. ISBN: 0769516866.
- [3] Joe Armstrong. “Making Reliable Distributed Systems in the Presence of Software Errors”. PhD thesis. Stockholm, Sweden: Royal Institute of Technology (KTH), Dec. 2003. ISBN: ISRN KTH/IMIT/LECS/AVH-03/09-SE. URL: [https://erlang.org/download/armstrong\\_thesis\\_2003.pdf](https://erlang.org/download/armstrong_thesis_2003.pdf).
- [4] Anish Athalye. *Porcupine: A fast linearizability checker in Go*. <https://github.com/anishathalye/porcupine>. 2017.
- [5] Docker/Moby Authors. *Docker: Open Platform for Building, Shipping, and Running Applications*. 2025. URL: <https://github.com/moby/moby>.
- [6] Go Authors. *The Go Programming Language*. 2025. URL: <https://go.dev/>.
- [7] Jaeger Authors. *Jaeger: End-to-End Distributed Tracing*. 2025. URL: <https://github.com/jaegertracing/jaeger>.
- [8] Kubernetes Authors. *Kubernetes: Production-Grade Container Scheduling and Management*. 2025. URL: <https://github.com/kubernetes/kubernetes>.
- [9] SeaweedFS Authors. *SeaweedFS: Fast, Simple and Scalable Distributed File System*. 2025. URL: <https://github.com/seaweedfs/seaweedfs>.
- [10] The Go Authors. *Package testing*. The Go Project. 2025. URL: <https://pkg.go.dev/testing>.
- [11] Benjamin Bach et al. “Time Curves: Folding Time to Visualize Patterns of Temporal Evolution in Data”. In: *IEEE Transactions on Visualization and Computer Graphics* 22.1 (Jan. 2016), pp. 559–568. ISSN: 1077-2626. DOI: 10.1109/TVCG.2015.2467851. URL: <https://doi.org/10.1109/TVCG.2015.2467851>.

- [12] Ivan Beschastnikh, Shayan Hosseini, and Finn Hackett. *Distributed Clocks*. 2024. URL: <https://github.com/DistributedClocks>.
- [13] Ivan Beschastnikh et al. *Visualizing Distributed System Executions*. New York, NY, USA, Mar. 2020. DOI: 10.1145/3375633. URL: <https://doi.org/10.1145/3375633>.
- [14] Ivan Beschastnikh et al. “Visualizing Distributed System Executions”. In: *ACM Transactions on Software Engineering and Methodology* 29.2 (Mar. 2020), 9:1–9:38. DOI: 10.1145/3375633. URL: <https://bestchai.bitbucket.io/shiviz/>.
- [15] Ivan et. al Beschastnikh. *GoVector: Vector Clock Logging for Go*. <https://github.com/DistributedClocks/GoVector>. 2020.
- [16] Ivan et. al Beschastnikh. *ShiViz: A Tool to Visualize Distributed System Logs*. <https://github.com/DistributedClocks/shiviz>. 2025.
- [17] Dmytro Borysenkov et al. *Analyzing Logs of Large-Scale Software Systems using Time Curves Visualization*. 2024. arXiv: 2411.05533 [cs.SE]. URL: <https://arxiv.org/abs/2411.05533>.
- [18] Ryan Dahl and the Deno Contributors. *Deno: A Modern Runtime for JavaScript and TypeScript*. 2020. URL: <https://deno.com/>.
- [19] Ryan Dahl and the Node.js Contributors. *Node.js: JavaScript Runtime Built on Chrome’s V8 Engine*. 2025. URL: <https://nodejs.org/en/>.
- [20] Travis Desell et al. “OverView: A Framework for Generic Online Visualization of Distributed Systems”. In: 107 (2004), pp. 87–101. ISSN: 1571-0661. DOI: <https://doi.org/10.1016/j.entcs.2004.02.050>. URL: <https://www.sciencedirect.com/science/article/pii/S157106610405193X>.
- [21] etcd Authors. *Logging conventions*. etcd, 2023. URL: <https://etcd.io/docs/v3.5/dev-internal/logging/> (visited on 04/23/2025).
- [22] etcd-io. *Adopters of etcd*. 2024. URL: <https://github.com/etcd-io/etcd/blob/main/ADOPTERS.md>.
- [23] etcd-io. *etcd*. 2024. URL: <https://github.com/etcd-io/etcd>.
- [24] Google. *Protocol Buffers: Google’s Data Interchange Format*. <https://protobuf.dev/>. 2008.
- [25] gRPC. *gRPC: A high performance, open source universal RPC framework*. 2024. URL: <https://grpc.io/>.
- [26] TJ Holowaychuk and the Express Contributors. *Express: Fast, Unopinionated, Minimalist Web Framework for Node.js*. 2010. URL: <https://expressjs.com/>.
- [27] jepsen-io. *Jepsen: Distributed systems testing framework*. <https://github.com/jepsen-io>. 2025.

- [28] Peiling Jiang, Fuling Sun, and Haijun Xia. “Log-it: Supporting Programming with Interactive, Contextual, Structured, and Visual Logs”. In: *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. CHI ’23. Hamburg, Germany: Association for Computing Machinery, 2023. ISBN: 9781450394215. DOI: 10.1145/3544548.3581403. URL: <https://doi.org/10.1145/3544548.3581403>.
- [29] A. B. Kahn. “Topological sorting of large networks”. In: *Commun. ACM* 5.11 (Nov. 1962), pp. 558–562. ISSN: 0001-0782. DOI: 10.1145/368996.369025. URL: <https://doi.org/10.1145/368996.369025>.
- [30] Cockroach Labs. *CockroachDB: The Resilient, Distributed SQL Database*. 2025. URL: <https://github.com/cockroachdb/cockroach>.
- [31] Vladimir Mihailenco. *MessagePack for Go*. 2025. URL: <https://msgpack.uptrace.dev/>.
- [32] MIT 6.102 course staff. *Reading 17 addendum: More on the JavaScript Event Loop*. <https://web.mit.edu/6.102/www/sp25/classes/17-callbacks-guis/addendum.html>. 2025.
- [33] Philip O’Toole. *rqlite: Lightweight, Distributed SQLite Database*. 2025. URL: <https://github.com/rqlite/rqlite>.
- [34] Diego Ongaro and John Ousterhout. “In search of an understandable consensus algorithm”. In: *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*. USENIX ATC’14. Philadelphia, PA: USENIX Association, 2014, pp. 305–320. ISBN: 9781931971102.
- [35] Jose Javier Gonzalez Ortiz. *Debugging by Pretty Printing*. Blog post, <https://blog.josejg.com/debugging-pretty/>. Mar. 2021.
- [36] raft.github.io. *The Raft Consensus Algorithm*. 2024. URL: <https://raft.github.io/>.
- [37] Jarred Sumner and the Bun Contributors. *Bun: A Modern JavaScript Runtime*. 2024. URL: <https://bun.sh/>.
- [38] Erlang/OTP Team. *Erlang/OTP Official Website*. 2025. URL: <https://www.erlang.org/>.
- [39] Temporal Technologies. *Temporal: Durable Execution for Microservices*. 2025. URL: <https://github.com/temporalio/temporal>.
- [40] Uber Technologies, Inc. *zap: Blazing fast, structured, leveled logging in Go*. <https://github.com/uber-go/zap>. 2025.
- [41] Doug Woos et al. *A Graphical Interactive Debugger for Distributed Systems*. 2018. arXiv: 1806.05300 [cs.DC]. URL: <https://arxiv.org/abs/1806.05300>.
- [42] Stefan Zager et al. *Intersection Observer API*. <https://www.w3.org/TR/intersection-observer/>. 2023.