TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

PULCHOWK CAMPUS

A

PROJECT REPORT

ON

FRAMEWORK FOR DISTRIBUTED APPLICATION DEVELOPMENT

SUBMITTED BY:

PRANJAL POKHAREL (PUL075BCT061)

SANDESH GHIMIRE (PUL075BCT075)

SANSKAR AMGAIN (PUL075BCT080)

TILAK CHAD (PUL075BCT094)

SUBMITTED TO:

DEPARTMENT OF ELECTRONICS & COMPUTER ENGINEERING

May, 2023

Page of Approval

TRIBHUVAN UNIVERSIY
INSTITUTE OF ENGINEERING
PULCHOWK CAMPUS
DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING

The undersigned certifies that they have read and recommended to the Institute of Engineering for acceptance of a project report entitled **"Framework for Distributed Application Development"** submitted by **Pranjal Pokharel**, **Sandesh Ghimire**, **Sanskar Amgain**, **Tilak Chad** in partial fulfillment of the requirements for the Bachelor's degree in Electronics & Computer Engineering.

.....
Supervisor
Babu R. Dawadi, PhD
Assistant Professor
Department of Electronics and Computer Engineering,
Pulchowk Campus, IOE, TU.

.....
Internal examiner
.....
Assistant Professor
Department of Electronics and Computer Engineering,
Pulchowk Campus, IOE, TU.

.....
External examiner
.....
Assistant Professor
Department of Electronics and Computer Engineering,
Pulchowk Campus, IOE, TU.

Date of approval:

Copyright

Acknowledgments

This project is being undertaken as a course requirement of the Major Project, as a part of the final year curriculum of Bachelor in Computer Engineering (BCT), IOE.

First of all, we are greatly indebted to our project supervisor Asst. Prof. Dr. Babu Ram Dawadi sir for his continuous guidance and supervision throughout the project. Our supervisor has helped us tackle multiple facets of the project including helping us prepare our initial drafts, making sure we were on schedule for the project completion, providing us with research papers and contexts for tackling specific problems, evaluating our work progress in terms of output, presentation, and literature, guiding us in preparing this final report and, most importantly, helping us achieve the task of representing the campus as ideal students. We are eternally grateful.

We'd like to express our deep gratitude to our teachers and professors from the Department of Electronics and Computer Engineering (DoECE) for providing us with this opportunity to express our creativity and inspiring us to come up with efficient solutions to existing technological problems. The department has facilitated many students over the years with excellent infrastructure and curriculum and we are proud to represent our department to the best of our abilities.

We are sincerely thankful to our classmates for their diligent feedback and continuous motivation. Our appreciation goes to our seniors who have helped and inspired us to reach newer heights in programming pedigree. The sincere and respectful culture of Pulchowk Campus is the best environment for fostering collaboration and competition among its students for novel projects and innovations in the years to come.

We would like to express our special gratitude to our families for their continuous support and their effort in making us what we are today.

Without the help and guidance of the aforementioned people, this study would not be possible.

Authors: Pranjal Pokharel, Sandesh Ghimire, Sanskar Amgain, Tilak Chad

Abstract

With the advancements in power, cost, and availability of microprocessors as well as the rapid growth in networking technologies and infrastructure, it is natural that we have developed technologies to allow hundreds of machines to connect with one another over a LAN for resource-sharing. These resources range from hardware devices to databases, files, computational power, and much more. With the benefits of such a system for developers and students in mind, this project, aims to be a framework using which resources of multiple systems within a computer network can be utilized. This framework is an attempt to abstract away the complexities of distributed application development by providing a platform to work upon. It is intended to be a robust, scalable, and safe platform for developing resource-sharing applications. The project itself is a culmination of our four years of engineering study, as we derive our knowledge base from teachings of the curriculum such as operating systems, distributed systems, data structures and algorithms, microprocessors, and computer networks. Thus, the project serves both facets of academia and practical implementation.
Keywords: *Distributed System, Computer Networks, Resource Sharing, Fault Tolerance*

Contents

List of Figures

List of Tables

List of Abbreviations

ABI	Application Binary Interface
API	Application Programming Interface
ARPA	Advanced Research Projects Agency
ARPANET	Advanced Research Projects Agency Network
CPU	Central Processing Unit
DRMAA	Distributed Resource Management Application API
DRMS	Distributed Resource Management System
FFI	Foreign Function Interface
FTP	File Transfer Protocol
GPU	Graphics Processing Unit
HTTP	Hypertext Transfer Protocol
IO	Input and Output
IOT	Internet Of Things
JM	Job Machine
JS	Javascript
LAN	Local Area Network
NS	Name Server
OS	operating System
OTP	Open Telecom Platform
RAM	Random Access Memory
RDD	Resilient Distributed Dataset
RPC	Remote Procedure Call
TCP	Transmission Control Protocol
TELNET	Telecommunications Network protocol
VM	Virtual Machine

1. Introduction

1.1 Background

In recent decades, there has been a staggering amount of development in computer systems with the development of cheaper and more powerful microprocessors than ever before. Likewise, innovations in high-speed computer network technologies have made it possible to communicate between devices within a few microseconds. On the other hand, more data is being produced than ever before thanks to the spread of devices and the internet of things (IoT), and conventional centralized systems are no longer able to handle the volume of data being produced. Hence, the idea of connecting different autonomous computer systems in order to create an easily accessible, scalable, and reliable distributed system is not limited to the concept but is available for all to use for themselves.

Distributed systems are one of the cornerstones of today's interconnected computer systems around the world. They achieve the processing of enormous amounts of data by breaking it up into smaller, manageable chunks and distributing it across multiple processing systems. From cloud-based services providing services seamlessly to millions of users around the world reliably and consistently to large GPU farms used for crypto-currency mining and training deep learning models, distributed computing has been used to solve a number of computational problems that are infeasible to be used on a single host machine.

We have developed a programming framework that facilitates the creation of reliable, safe, and flexible software applications capable of executing on multiple nodes. This framework provides developers with a robust programming model that supports distributed computing and enables the creation of resilient software systems. Our framework extends the concurrency capability of Golang that allows programmers to distribute function execution in multiple nodes and the low-level API/features of OS accessible through C++ to consolidate cross-platform execution and filesystem access. In addition to this, we have implemented various algorithms pertaining to distributed computing, the results of which can contribute to academic research and a better understanding of distributed systems overall.

1.2 Problem Statement

1. **Limited computational power in single-processor systems:** The evaluation of computationally expensive problems can take minutes or hours, and although com-

puting power is steadily increasing as well as parallel processing capabilities, the complexity continues to keep pace. Single-processor systems are being outmatched in this respect.

2. **Under utilization of available resources:** Even within a small campus network, we have multiple interconnected computers whose processing power we can channel together to create a single abstraction unit capable of huge amounts of processing. However, distributed processing is complicated and has not been tested within the campus network yet.

3. **Development of distributed networking applications:** While simple socket programming can be fairly understood and developed, creating a whole distributed ecosystem is something that requires a lot of expertise and planning. There are frameworks available for this but they are mostly tailored toward a particular application (for eg, combined GPU computation for training deep learning models in TensorFlow) and are not useful for general programming structure.

1.3 Objectives

1. To create a generic framework for developing distributed applications over the network.

2. To develop a distributed resource-sharing and computing platform.

3. To support seamless resource sharing between both homogeneous and heterogeneous systems (based on similar or dissimilar OS) i.e. develop the framework to be system-agnostic.

1.4 Scope

The proposed framework can be adopted by programmers, from a small group of people to an entire team to develop distributed applications from multi-user applications (like a collaborative whiteboard, and paint) to distributed resource sharing and computing. Some of the examples of the project can be a collaborative whiteboard, paint, or computing expensive parallelizable tasks such as a Mandelbrot set. For practical purposes, we will limit the presentation scope to a small LAN consisting of a limited number of computers.

1.4.1 Academic study/research

Academic research is a crucial scope of our project. By sharing our framework implementation with other researchers, they can benefit from the study of distributed algorithms,

optimization techniques, networking protocols, mutual exclusion policies, and more. Academic research often requires the use of specialized hardware or software and involves working with smaller datasets that do not require the complexity of established frameworks such as Hadoop or MapReduce. Our customizable framework with an easy-to-use API, thus, has an important scope in academia. Furthermore, frameworks like Hadoop and MapReduce do not support a general model of programming, with no notion of variables.

1.4.2 Comparision of scope with existing frameworks

There are various distributed computing frameworks like Hadoop and MapReduce. However, there are several key differences between these technologies and our framework. Hadoop and MapReduce are designed specifically for processing large data sets in a distributed environment, whereas our framework more general-purpose and can be used for a wider range of distributed computing tasks. Hadoop and MapReduce rely on a batch processing model, where data is processed in discrete batches rather than in real-time. Our framework supports distributed function calls in a more real-time or on-demand fashion.

2. Literature Review

Apache Hadoop[1], a software framework, allows for the distributed processing of large data sets across clusters of computers. It facilitates the processing of big data using the MapReduce programming model. MapReduce program is composed of a mapping procedure, which performs filtering, sorting, and other operations on data and a reduce method, which performs the summary operation (averaging the result of all computations). Hadoop only supports batch processing and doesn't explicitly provide real-time processing for streaming data. It doesn't support caching and imposes higher latency on data processing.

The Dryad[2] distributed engine is a project at Microsoft Research developed as a general-purpose runtime for the execution of data-parallel applications. The basic working concept behind Dryad is the division of execution resources on the basis of a directed, acyclic graph known as the 'job graph' - the vertices representing computation nodes and the edges representing communication channels. The Job Manager (JM) is responsible for monitoring the overall status of executing vertices and the state of computation regarding how much data has been read and written on its channels. A Dryad job is itself a chain of processes with each process piping its output to the next process in the chain. Components like the Name Server (NS) maintain the list of available vertices while each computer runs a Daemon as a proxy for managing shared processes.

Apache Spark[3] is another distributed, open-source, distributed processing system used for large data processing. The main component of Apache Spark is RDDs which are collections of objects that are spread across the cluster. The independent parts can be split across the cluster, then the computation is performed in a single node and finally, the overall result is reduced to a single structure. All those details are abstracted out, and the programmer interacts with the data as if they are in a single node.

"A high-level framework for network-based resource sharing"[4] by James E. White proposes an application-independent, resource-independent framework for resource sharing built on the distributed design techniques within ARPA Computer Network and outlines an alternative to the approach used by ARPANET system builders since the 1970s. It introduces current software approaches to resource sharing with Host-Host Protocols and various function-oriented protocols by which processes deliver and receive specific services via IPC. Such hands-on resource-sharing protocols like TELNET have inherent limitations which are addressed by protocols that simplify and standardize the dialogue between user and server

processes. FTP is one such family of protocols that follow a command/response format. On the basis of such principles, the paper formally defines a machine-independent, application-independent request/reply format of communication and the idea of modeling resources as a collection of procedures. The idea behind function dispatching is derived from the original paper for RPC, 'Implementing Remote Procedure Calls'[5].

Gluster [6] implemented RPC by sending the entire Go source file and dynamically linking it. However, it can only be implemented in Unix-based operating systems, and if the linking fails error cannot be propagated back to the programmer, and the node might crash without proper error handling.

2.1 Related theory

2.1.1 Distributed System

A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system. A computing element, which we will generally refer to as a node, can be either a hardware device or a software process.[7]

Distributed Systems are characterized by their properties of concurrency, independent failures, and having no global clock. They are used in a wide range of applications like Web search, Simulation, File Sharing, and many more.

2.1.2 Fault Tolerance

The ability of a system to keep running uninterruptedly when one or more of its components fail is referred to as fault tolerance.

In the context of a distributed system, [8] Fault tolerance is the dynamic method that's used to keep the interconnected systems together and sustain reliability and availability in distributed systems. The hardware and software redundancy methods are the known techniques of fault tolerance in a distributed system. The hardware methods ensure the addition of some hardware components such as CPUs, communication links, memory, and I/O devices while in the software fault tolerance method, specific programs are included to deal with faults. An efficient fault tolerance mechanism helps in detecting faults and if possible recovering from them.

2.1.3 Transmission Control Protocol

Transmission Control protocol, a compliment of the Internet Protocol, is a common protocol that enables applications running on different nodes on the network to effectively and reliably send and receive data from one another. An important feature of this protocol is the fact

that it is connection-oriented and requires a connection to be established between the two parties before data is sent or received. It implements features of flow control, error control as well as congestion control.

2.1.4 Race Condition and Mutual Exclusion

In distributed systems, mutual exclusion and race conditions are two related concepts that are important to understand for building correct and efficient concurrent/parallelly executing inter-node processes.

Mutual exclusion is a technique used to prevent multiple nodes from modifying shared resources at the same time. If multiple nodes attempt to modify a shared resource simultaneously, the program may encounter *race conditions*, a condition in which the system's behavior becomes dependent on the timing of uncontrollable events. In a distributed system, race conditions can be more prevalent since processes may be executed concurrently on different machines and may not have a consistent view of the shared resource. Race conditions can lead to data inconsistencies or errors and can be difficult to detect and resolve.

To prevent this, mutual exclusion is used to ensure that only one thread can access a shared resource at any given time, while all other threads must wait until the resource is released. This can be achieved using synchronization primitives such as locks, semaphores, or monitors.

2.1.5 FIFO Message Queue

In multi processing environment, for two-way communication, the order of the message sent from one node to another is essential. First In First Out (FIFO) message queue retains the order of the message and operates in a first-out fashion. This approach ensures that if a backlog of messages builds up, the oldest messages are processed first. This removes the necessity for maintaining absolute timestamps for individual requests and ensures the processing of requests in the correct order.

3. Methodology

3.1 Description of the Working Principle

The developed framework acts as an abstraction over all the low-level OS system calls and networking aspects needed for distributed application development. Developers will be able to focus on writing programs for this machine while remaining ignorant of the physical network underneath and OS-specific system calls. A task can be submitted in the network using our own protocol for distributed task management. By separating into independent different layers, the framework remains flexible enough for developers to optimize the application for their own purpose.

3.1.1 Low level OS layer

This layer of the framework deals directly with the OS. This core part of the framework is being implemented in 'The C++ Programming Language' and will directly interact with OS for retrieving resource information and managing the filesystem. This layer needs to be implemented for each target platform we wish to support.

Distributed File System (GutFS)

A distributed network file system named Guthi File System (GutFS) has been implemented. It allows seamless syncing and sharing of files and resources across connected nodes. Each connected node maintains three different components on per node basis locally:

1. **Global Root Structure**
   This structure maintains the node's view of the entire network file system.

2. **Local File System**
   This component maintains the node's own file system.

3. **Cache**
   It caches the recent file fetched and updates accordingly.

Each file is maintained as a structure containing three fields :

- Unique file name

- IPv4 address of the owning node
- Timestamp of the latest change

Whenever a file has to be fetched, the timestamp corresponding to the file is fetched from the network. If only the timestamp is greater than the one stored in the cache, the whole content is fetched again and cached with the updated timestamp. Otherwise, the one available in the cache is used.

The Following series of operations occurs when a local file is changed :

- The node's own local file system is updated.
- The node requests all node's local filesystems and syncs with its own view of the global root structure.
- Local filesystem is merged with global root structure.
- Global root structure is broadcasted along with *RootStructureUpdated* message over the network.

Multi-Level Caching

To avoid re-fetching the unchanged contents frequently, a multi-level caching system has been implemented. It operates on two levels :

- If the current RAM usage of the node is less than 80 percent, the content is cached in RAM.
- If the current RAM usage of the node is greater than 80 percent, the content is cached in the disk.
- If current RAM usage is less than 80 percent and there is content cached in the disk, they are moved to RAM

Resource Manager

In a distributed system, different nodes may need to monitor other nodes for the dispatching of functions. The status of nodes is obtained from the operating system and passed to all the connected nodes.

3.1.2 Networking layer

The network stack is implemented in the Golang programming language because of its rich support for concurrency. The networking layer includes client discovery and routing, passing of runtime information of the system communication between the nodes, transferring files, caching, dispatching of tasks, and node failure detection.

Communication Establishment Protocol Between Nodes

In the realm of network communication, the establishment of a connection between two nodes is a critical process that underlies the exchange of information. To initiate this process, one node sends a connection request to the other, prompting the receiving node to respond with an acknowledgment and its relevant node information. Upon receipt of this acknowledgment, a **persistent TCP connection is established between the two nodes**, which can subsequently be leveraged for data transmission without incurring the overhead associated with repeated connection establishment attempts. By minimizing the frequency of connection requests, this approach enables a more efficient and reliable mode of communication between network nodes. It is worth noting that such connections are subject to termination when the exchange of data is complete or due to various other factors that may interfere with the connectivity between nodes.

Network Message Format

A specified message format is used for communication between nodes. First, four bytes of the message include the length of the total message. This allows the receiving node to read a fixed amount of bytes from the TCP connection object. The next 24 bytes of the message are the type of message that is transmitted. Then, the remaining bytes contain all the gob-encoded messages.

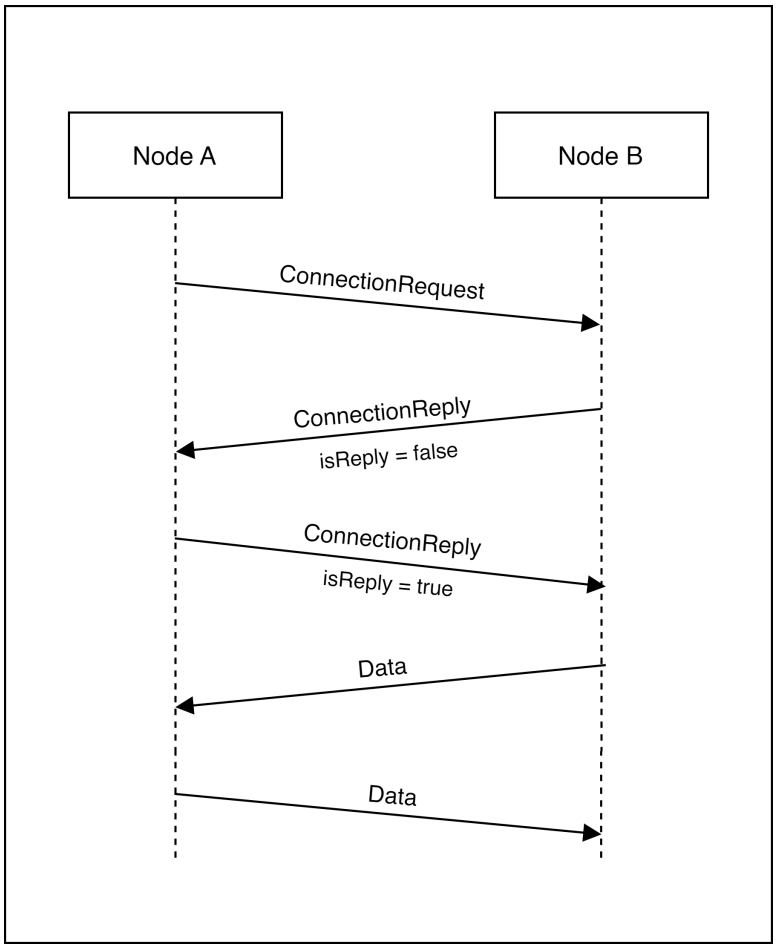


Figure 3.1: **Connection initiation protocol:** This figure provides an overview of connection establishment steps between nodes. The parameter *isReply*s indicates whether it is a connection reply to the *ConnectionReply* should be sent. If *isReply* is *true* then an acknowledgment is not sent by the receiving node.

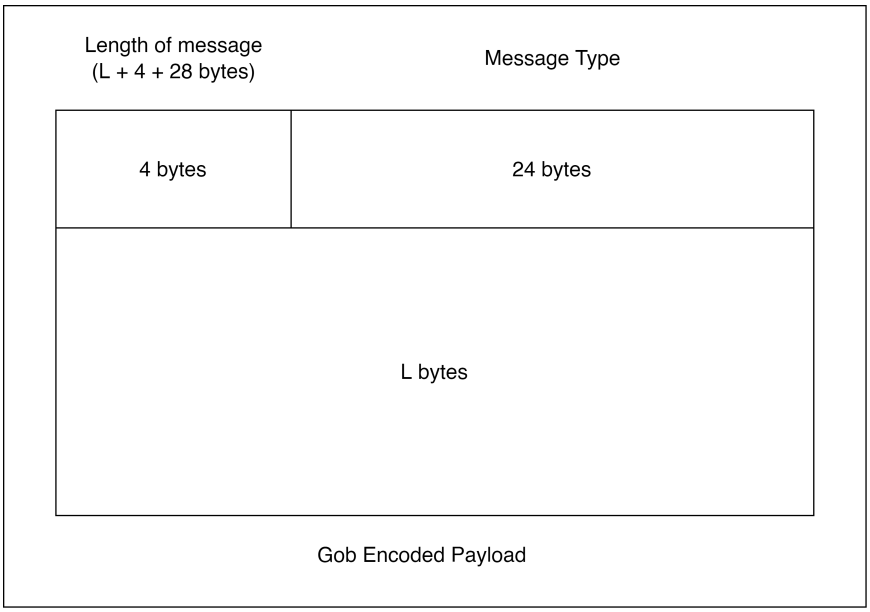


Figure 3.2: **Network Message Format - Header + Payload:** Illustration of the format of the message along with their respective size

Message Type	Content of the Message
getnodes	Request for the node information
node	Node information
echo	Message for pinging node
echo_reply	Reply of ping message
connect	Connection Request
connection_reply	Reply of the connection request
get_mem_info	Request of the Memory Information
get_cpu_info	Request of the CPU information
cpuinfo	CPU Information
meminfo	Memory Information
get_fs	Request of the global filesystem
filesystem	Global Filesystem in the node
variable	Entire variable structure
array	Array of variables
indexed_array	Array of variables with start and final index
symbol_table	All the variables present in the current node
token_request_sk	Request for the token, so that node can execute critical section
get_var	Request for variable
validity_info	Invalidation Signal for the nodes
function_dispatch	Function information and
func_state	State of the function
func_completed	Event Signalling completion of function requested

Table 3.1: Message header: Type fields and their corresponding follow-up content type

Distributed Variable

Our framework is designed to provide programmers with a flexible and efficient method of creating and sharing distributed variables across multiple nodes. By utilizing a **hashing mechanism to generate integer-based storage names (i.e. symbol tables)**, we can optimize the storage of distributed variables without sacrificing functionality or performance. Along with the value, and id, distributed variables also contain the type of variable which can be obtained using the reflection capability of the Go programming language. This allows safety and proper error propagation making the program easier to debug.

Arbitrary Variable Types

To ensure that our distributed variables can be used with any data type, we have implemented them as type *interface* . While this provides programmers with greater flexibility, it also requires that caution be exercised when using these variables, as any attempts to cast them to a different type can result in runtime errors. Although Go generics are a safer and faster alternative to *interface*, they were not suitable for our implementation, as the type of variable cannot be known at compile time when passing it to another node. As a result, we believe that our decision to use *interface* strikes a balance between flexibility and performance, while also ensuring the safety and integrity of our distributed variables

Optimization

One of the main issues of distributed variables is to, sharing of the value between nodes. When a variable is updated in one node, its value should be propagated to all the nodes connected to it. However, sending the value of every variable every time is very expensive. So, **instead of sending a variable every time when it is updated, we send an invalidation message for that particular variable to all the nodes**. The size of the invalidation message is small compared to the message when sending the entire variable. When another node tries to access the invalidated value, it requests the variable from the source node and receives the variables.

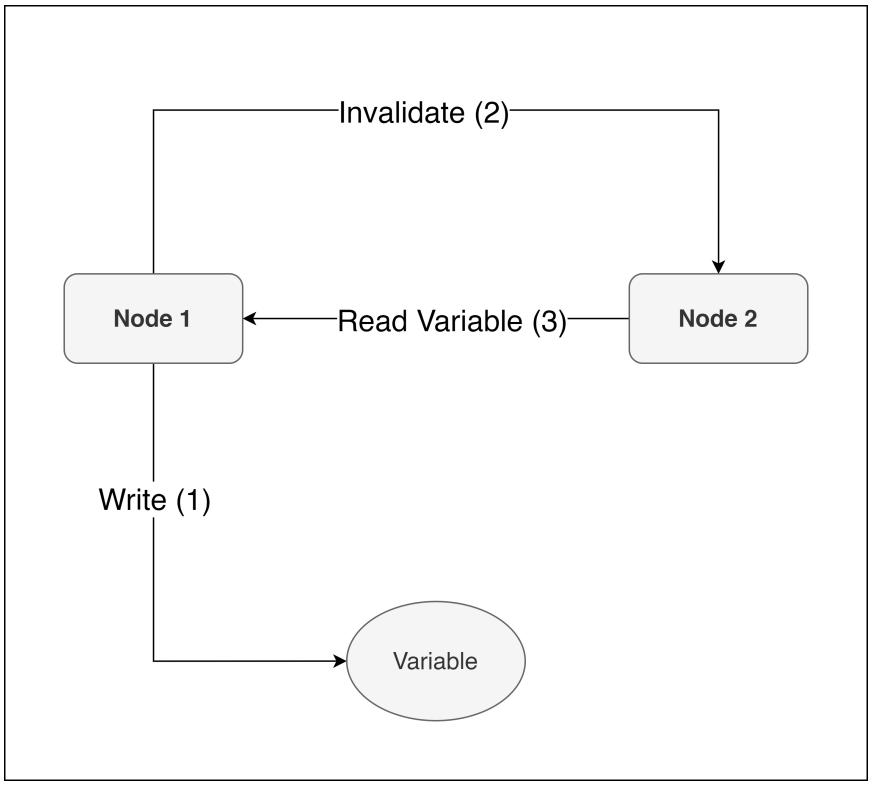


Figure 3.3: **Variable Invalidation Protocol:** When a node changes the value of a variable, invalidation is sent to all the connected nodes. Then, when another node tries to read the updated variable, it requests the value from the source node.

Distributed Mutual Exclusion

In a distributed system environment, concurrent access to shared variables may result in race conditions that can have detrimental effects on system performance and reliability. While traditional mutex and semaphore-based approaches have been used to address this challenge in non-distributed systems, such approaches are not suitable for distributed systems due to the absence of shared memory.

To overcome this limitation, we have implemented the Suzuki Kasami Algorithm [9] for achieving mutual exclusion of algorithms between nodes. This algorithm is particularly well-suited for use in distributed systems where communication delay is unpredictable. Our decision to adopt the Suzuki Kasami Algorithm over the Ricart and Agrawala algorithm [10] was driven by the former's optimal message-passing paradigm, which enables it to deliver superior performance in distributed systems with unpredictable communication delays.

By utilizing the Suzuki-Kasami Algorithm in our distributed system, we have successfully achieved mutual exclusion of algorithms between nodes so they can truly implement algorithms correctly without any race conditions.

Internode Function Call

Our framework allows programmers to make function calls from one node to another node provided that the function signature exists in both nodes. This allows programmers to develop a complete distribution system. The motive is to implement remote procedure calls over the main network without requiring the overhead of using a separate RPC server. Key points considered while implementing the dispatch mechanism include,

1. **Abstraction:** Remote function calls are abstracted enough that the user doesn't realize it is being made remotely i.e. from a user's perspective, the function runs as if it were running in the local machine.

2. **Arbitrary Function Invocation:** Our framework has ensured that functions of any signature can be invoked on any external node provided that the function definition is available on both nodes. Due to the statically typed nature of Go and the complexity constraints in generating functions at runtime, any function that is required for remote dispatch must be provided at compile time.

3. **Network Latency:** The performance of function dispatch is calculated considering the effect of network latency and persistent connections are used to account for the intermittent change in distributed variables.

4. **Error Handling:** Invalid function calls are properly handled and it is ensured that the core process is not halted unless explicitly provided a termination interrupt occurs.

To make sure these requirements are fulfilled, the following approaches were taken,

1. **Reflection:** For working with function definitions, we have used the reflection feature of the Go programming language to work with dispatch at runtime [1]. The Package 'reflect' of the Go standard library implements run-time reflection, allowing a program to manipulate objects with arbitrary types. This feature is used to compare and call arbitrary functions, thus fulfilling one of our key requirements.

2. **Network-Wide Global Function Store:** The reflection features are used to maintain a global store of all remotely 'callable' functions. Every node has a copy of this function store and only the functions added to this store can be invoked dynamically.

3. **Error Handling at Run Time:** Reflection panics are impossible to handle when they occur due to their pure run-time nature. As a result, a lot of type checks are made at different stages of the function calls to ensure the main program loop does not crash.

Serialization of Message

Before sending data over the network, the message needs to be serialized. For serialization, we used Golang's internal Gob encoding [2] protocol. The primary reasons for using Gob encoding for serialization (over traditional serializations such as JSON or XML) include,

1. **Efficiency:** Gob encoding is fast and efficient for Go data types and structures.

2. **Type Safety:** Encoded data contains information about its type and structure within itself, so we don't need to send in data type details. This also ensures type safety as we automatically enforce type compatibility between the encoder and the decoder.

3. **Custom Types:** Since Gob supports the encoding of custom types based on interface details, it is ideal for our frameworks that use many custom structures and types.

4. **Efficient Pointer Encoding:** Gob encoding flattens pointers automatically i.e. it can encode and decode pointers to values without changing the code structure.

[1] Go reflection: `https://pkg.go.dev/reflect`

[2] Gob Encode: https://go.dev/blog/gob

Node Failure Detection and Handling

Distributed systems face a significant challenge in detecting node failures due to the absence of a dedicated link between nodes. Furthermore, there is also the question of how to exactly handle a node failure after it has been detected.

Failure Detection

First, to address the challenge of failure detection, our framework employs a **continuous echo message transmission mechanism** to all connected nodes, followed by a wait for a reply from each node. In the event that a reply is not received within a 10-second interval, the corresponding node is deemed to be disconnected. This approach provides a reliable and low-overhead means of detecting node failures in a distributed system environment.

Failure Handling

Notably, different distributed systems may require distinct approaches for handling node failures. To cater to this requirement, our framework offers flexibility to programmers by enabling the binding of a **callback function** that will be invoked in the event of a node failure. By providing this functionality, our framework empowers programmers to design custom node failure handling mechanisms that align with their specific distributed system needs.

Function Call and Node Failure

We have exposed a function that allows programmers to **bind the state to the progress of a remotely executing function**. The state is then sent to every other node in the system to ensure that it is available in the event of a node failure. This approach ensures that, in the event of a node failure, the next dispatch of the function can resume from the state where the failed node left off. We tested the effectiveness of this technique in a variety of scenarios and found that it is a reliable way to handle node failures in a distributed system. However, we acknowledge that this technique may not be necessary or appropriate for every use case. As a result, we have made the state-binding mechanism an extension case for our distributed resource-sharing framework rather than an internal feature. This allows programmers to choose whether or not to use this technique based on the specific needs of their application.

The combination of our continuous echo message transmission mechanism and the ability to bind a callback function, along with state management for function call resumption from point of execution failure, represents a robust approach to addressing node failure detection

and handling in distributed systems.

Time synchronization

Time synchronization is one of the most critical factors that determine the pipeline of execution in distributed systems. For any sequence of events that needs to happen sequentially, we must establish an order based on a clock synchronized across all system nodes. Since we do not use a dedicated time server for our implementation in order to adhere to peer-to-peer system design as closely as possible, we **use an external NTP clock** to synchronize events between systems.

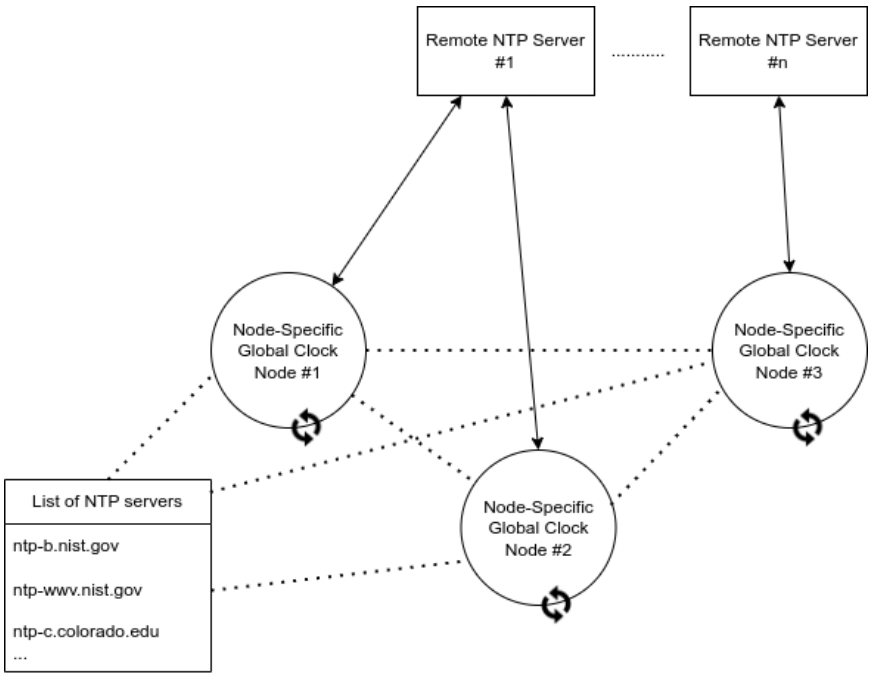


Figure 3.4: **Time Synchronization Mechanism using NTP:** Each node syncs its local clock with a remote NTP server, thus ensuring there is eventual consistency in local network time.

The implementation details for time synchronization can be enumerated as,

1. **Fetch Time From Remote NTP Server:** The framework maintains a list of remote NTP servers that return accurate time. The basic approach is to use time from the first server that is up and running. The time fetching is done every 10-15 seconds (arbitrarily decided) since servers do not respond well to continuous polling.

2. **Node-Specific Global Clock:** Since we are using an external clock, the nodes each have their own global clock state which is updated every second depending on one of two cases - (a) fetch time from NTP server at time t or, (b) locally increment the time every second.

The global clock time is used for any time-ordered events for distributed processing. The NTP client was written in Go, the implementation details for which is derived from the RFC 5905 document specifying the NTP Protocol[11].

4. System design

4.1 Requirement Specification

4.1.1 Functional Requirements

- A node should be able to connect to the network by providing its address and port
- A node should be able to access the shared resources of another node
- A user of the framework should be able to use it to execute a computational task on multiple nodes
- A user of the framework should be able to utilize the distributed file system in their application
- A user should be able to see the current network information and the status of all nodes connected to it.
- A user should be able to see the resource information provided by each node in the resource manager.

4.1.2 Non-Functional Requirements

- The low-level abstractions should be hidden from the user
- The framework should be as generic as far as possible to the user application
- The framework should be scalable in terms of the addition of nodes

A component-based approach has been followed to develop the proposed framework i.e. parts of the system that will perform the related tasks are grouped into a single component. Components involved include caching system, resource manager, node discovery and connection component, task schedular and dispatcher, memory pool, thread pool, and a component wrapping OS-specific functionalities.

The lower level layer of the framework is written in C++ while the upper layer is implemented in Go. Communication between two layers is managed in a strict client-server fashion where communication is non-blocking and bi-directional.

4.2 Components and Interactions

4.2.1 Interaction between low-level OS layer and Networking Layer

The communication between the Go runtime that handles the networking stuff and the C++ runtime that handles low-level OS-specific features is governed by a client-server architecture. The C++ runtime manifests itself as a daemon listening to the events from the OS and client (Go runtime) and responds to the message accordingly. The communication between the daemon and the client is bi-directional and non-blocking.

4.2.2 Daemon-Client Interaction

Daemon implemented in C++ and Go runtime strictly communicates in a client-server fashion. The communication is non-blocking and bi-directional.

The client and servers (daemon, in this case) communicate by following a specific protocol. The protocol is named as Guthi Protocol and is specified as follows:

- The first 5 bytes of any message between client and server should be followed by 5 magic bytes: *0x47 0x55 0x54 0x48 0x49*
- The next byte contains the type of message that follows. It is defined by the enumeration:

Command	Value
GetFile	0
CheckIfInCache	1
RequestFileMetadata	2
NoSuchResourceExists	3
TrackedFileChanged	4
TrackThisFile	5
EchoMessage	6
Continuation	7
InvalidRequest	8
GetCachedFile	9
ResetConnection	10
MessageNone	Other

Table 4.1: EMessage enum values and description

- The next 2 byte contains the length of the message in little-endian order (least significant byte followed by most significant byte).
- The remaining n bytes (where n is the length received in the previous step) contain message-specific metadata.

  For demonstration, an echo message request containing a "Hello World" message is transferred as the following stream of bytes:

  0x47 0x55 0x54 0x48 0x49 0x06 0x0B 0x00 'H' 'e' 'l' 'l' 'o' ' ' 'W' 'o' 'r' 'l' 'd'

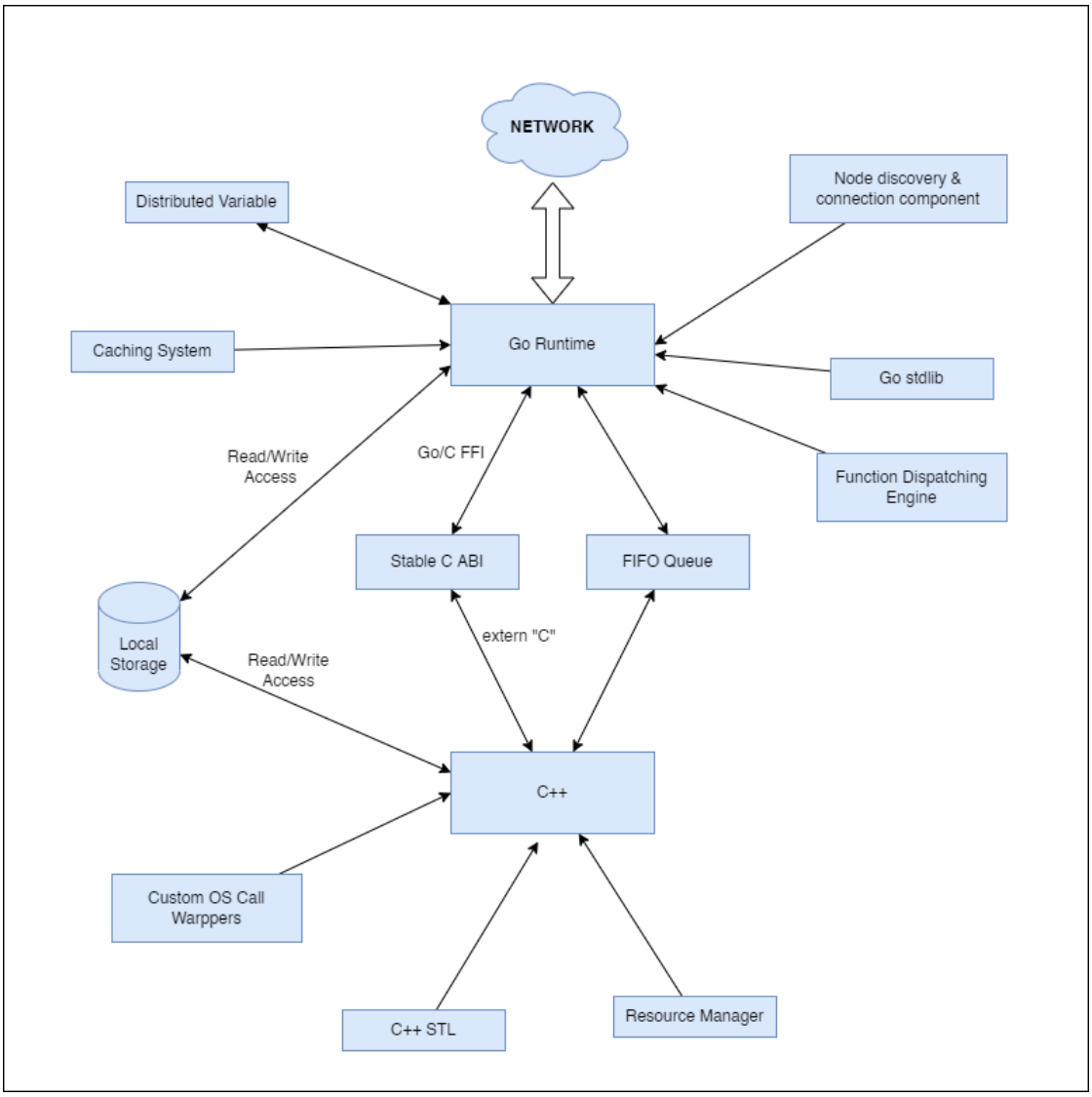


Figure 4.1: **Component Interaction (single node view):** Overview of the interaction of different components of the system. Multiple nodes communicate with each other through the network and communication is handled by Go runtime. C++ Runtime is responsible for resource various OS system calls, queries of resource usage of the system, and management of the filesystem. Go and C++ Runtime communicate through foreign FFI calls provided by the stable C ABI from the Go runtime and FIFO Queue.

4.3 UML Diagrams

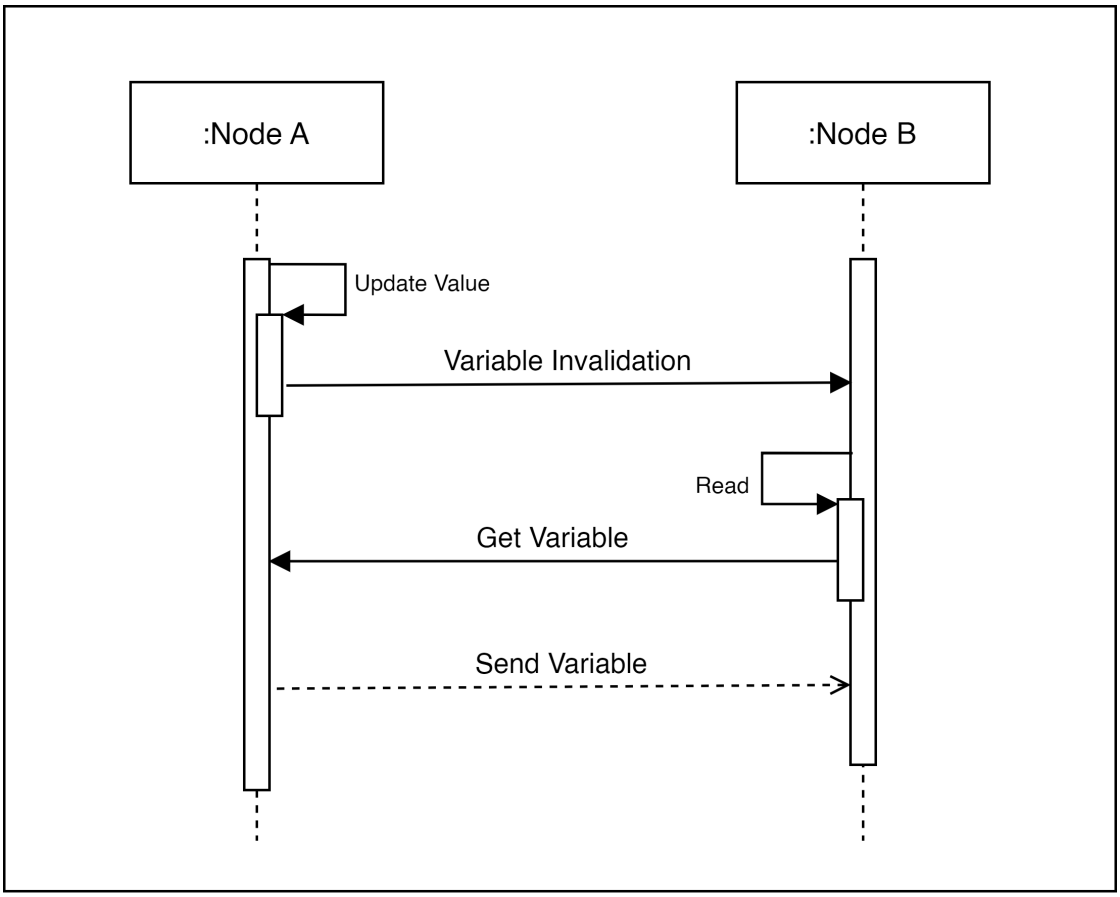


Figure 4.2: **Sequence Diagram for Distributed Variable Read and Write Operation.** When a node changes the value of a variable, invalidation is sent to all the connected nodes. Then, when another node tries to read the updated variable, it requests the value from the source node

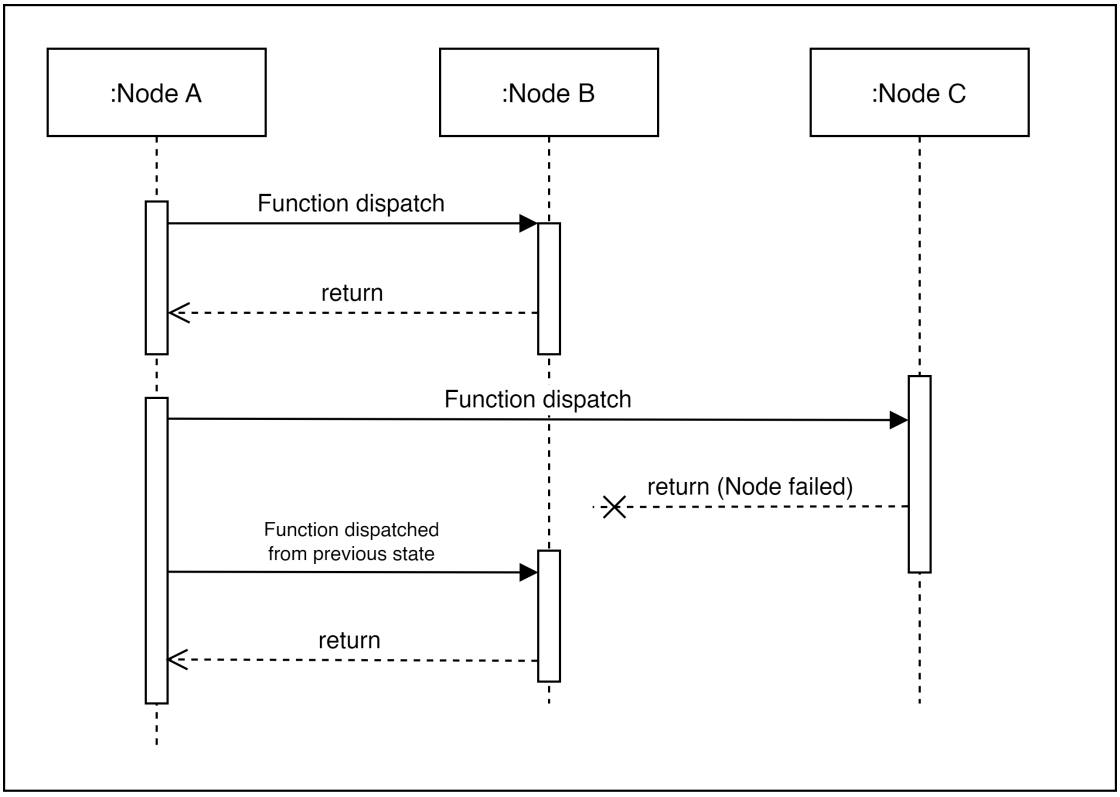


Figure 4.3: **Sequence Diagram for Inter-node Function Call.** If the function call fails due to node failure, the execution is resumed from the state right before failure on any other node which is available.

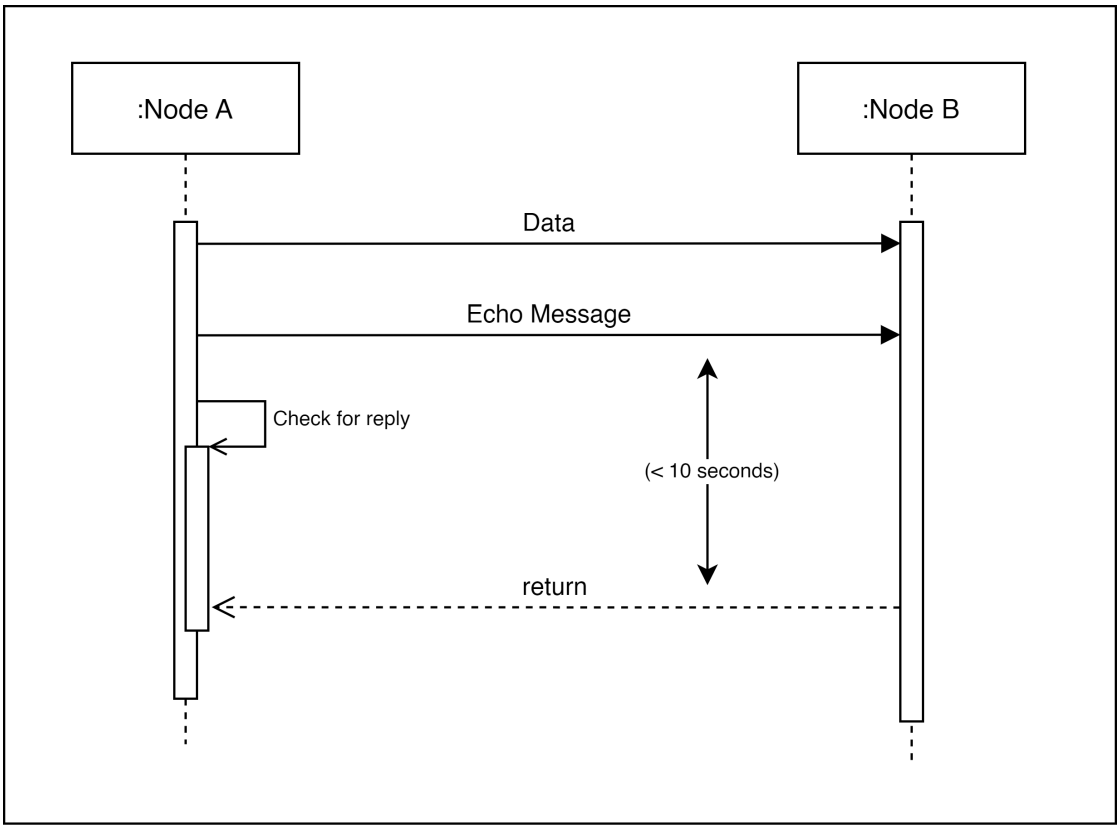


Figure 4.4: **Sequence Diagram for Node Failure Detection**. An echo message is sent to all the nodes, and if the response is not received in the next 10 seconds, then the node is considered failed.

4.4 Languages and Tools

C++

C++ is a general-purpose, compiled language developed by Bjarne Stroustrup and was initially released in 1985 as an addition to the C language. One of the appealing features of C++ is the fact that it provides the programmer access to low-level system resources. The low-level components like resource management, File System, Caching, and OS call wrappers are implemented in C++.

Go

Go [1] is a high-level programming language developed by Google. Go was chosen as the programming language for implementing the Network platform because of its rich support for concurrency and reflection. Upper-layer networking components like client discovery, routing, runtime information sharing, and file transfer are written in Golang.

CMake

CMake [2] is an open-source, cross-platform, meta-build system to build software. CMake is not a build system itself; it generates another system's build files. It supports directory hierarchies and applications that depend on multiple libraries. It provides a proper abstraction for building projects with different configurations.

Javascript

JavaScript (JS) [3] is a lightweight, interpreted, or just-in-time compiled programming language with first-class functions. While it is most well-known as the scripting language for Web pages, many non-browser environments also use it, such as Node.js, Apache CouchDB and Adobe Acrobat. JavaScript is a prototype-based, multi-paradigm, single-threaded, dynamic language, supporting object-oriented, imperative, and declarative styles. The interface to view the network information provided to the user, the interface to connect a new node to the network, and the resource monitoring graphs in this project are written in Javascript.

[1] Go: https://go.dev/

[2] CMake: https://cmake.org/

[3] JS MDN: https://developer.mozilla.org/en-US/docs/Web/JavaScript

5. Results & Discussion

5.1 Analysis of Different Algorithm

Effect of Persistent Connection

With the initial implementation of our project, we established connections between nodes every time we needed to pass some data at that very instant (non-persistent connection). We later developed a version that cached connections for future use, thus effectively establishing a persistent connection. We compared the result of performance between the persistent connection and non-persistent connection for the rendering of mandelbrot in two different nodes.

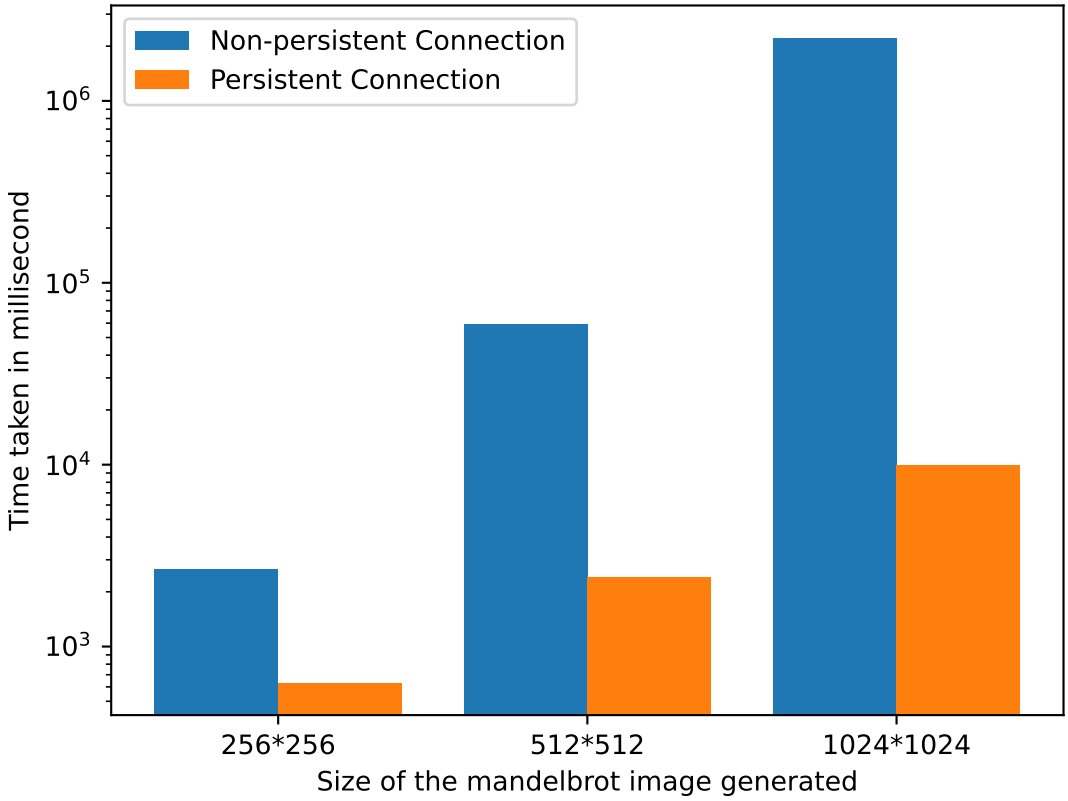


Figure 5.1: Plot of time taken to render Mandelbrot set of different sizes with persistent and non-persistent connection

The performance difference between the two cases is very much drastic. The difference in performance is even more when the complexity of the application increases.

Effect of Number of nodes in the speed of computation

We tested the effect on computation time as the number of nodes changes. The results demonstrated that there is a decrease in computation time with an increase in the number of nodes. However, the findings also suggest that this relationship is not linear, as further increasing the number of nodes leads to an increase in total time. This outcome could be attributed to the potential overhead associated with network communication as the number of nodes increases. Overall, these findings highlight the importance of carefully considering the number of nodes in a distributed computing system to optimize performance.

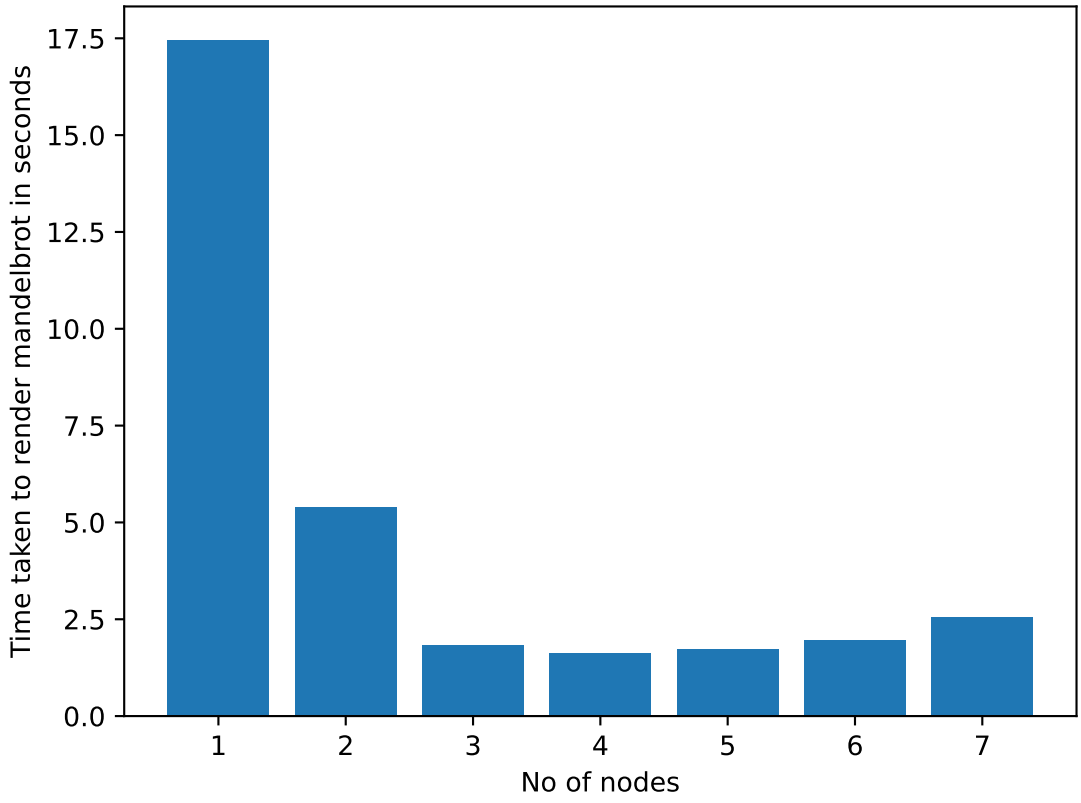


Figure 5.2: Effect of number of nodes on the speed of Mandelbrot rendering

5.2 Network Visualization Interface

An Interface to visualize the Network has been created as a web application. It enables each node to view the current information of the network. The interface also enables a new node

to connect to the network by providing its address and port.

5.2.1 Home

The main index provides general information on the functionalities of the interface.

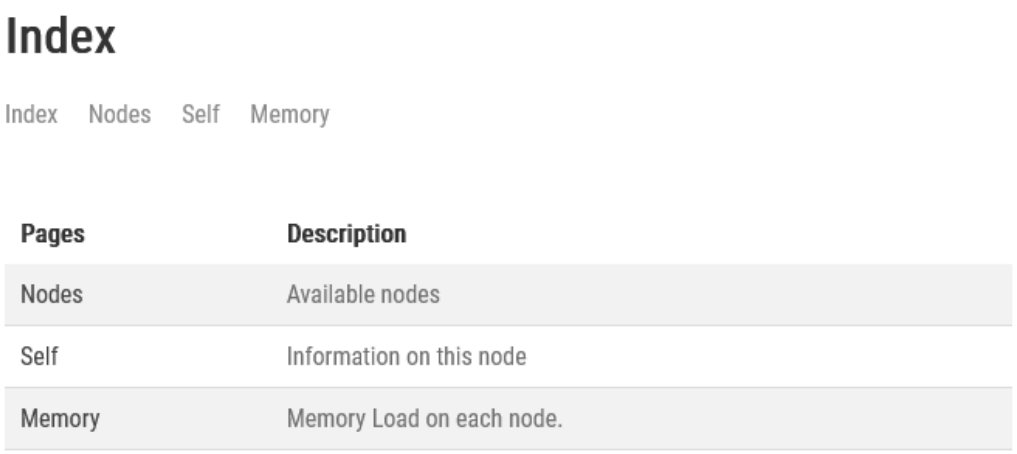


Figure 5.3: **Network Visualization Home**: User Interface of the home page for the network visualization page.

5.2.2 Self Node Information

The Self page provides information about the user's own node like name, id, address, and port.

5.2.3 Nodes

The Nodes page provides information about the nodes connected to the network like their name, id, address and port.

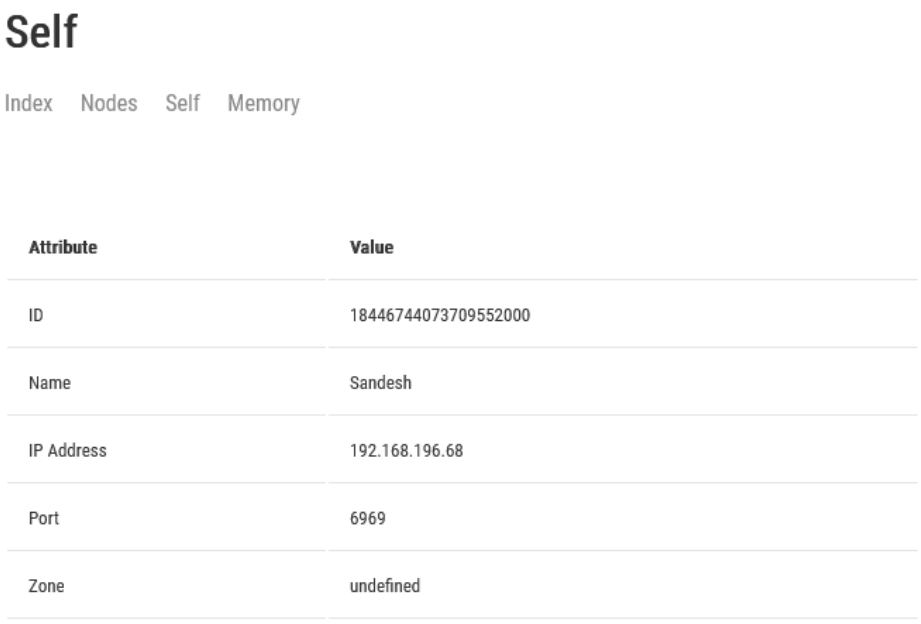


Figure 5.4: **Network Visualization Self**: The Self page provides information of name, id, address, and port of the user's own node

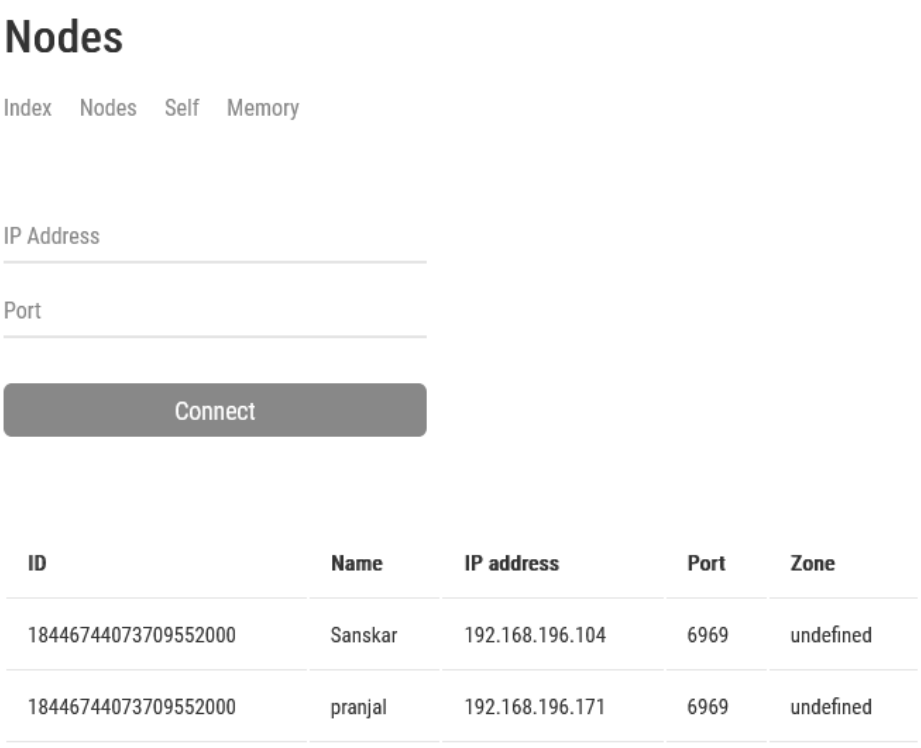


Figure 5.5: **Network Visualization Nodes**: Nodes tab enables a new node to connect to the network as well as view general information about other nodes

5.2.4 Memory

The Memory page provides information about the memory load and CPU usage of each node connected to the network. This information is provided as a resource monitor graph.

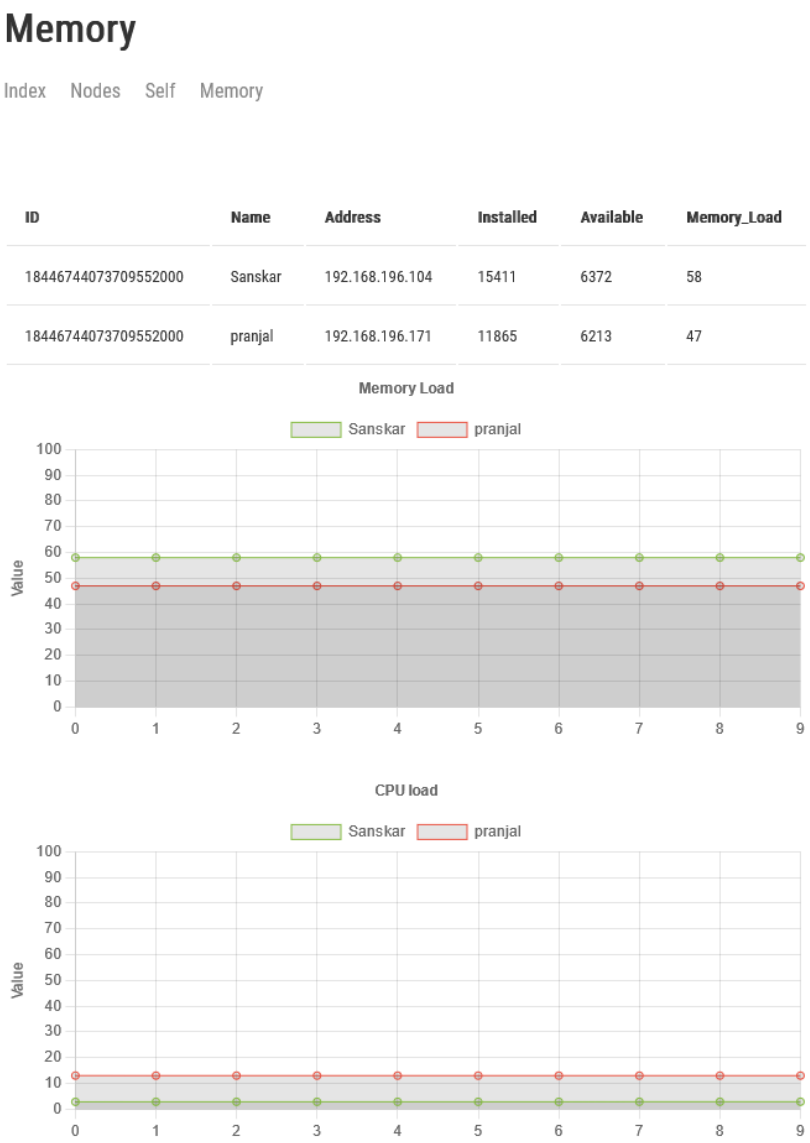
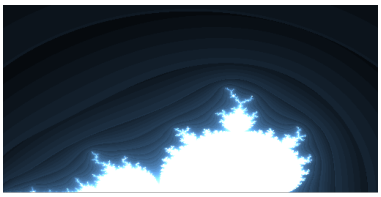


Figure 5.6: **Network Visualization Memory and CPU**: This tab provides information about the memory load and CPU load of each of the nodes connected.

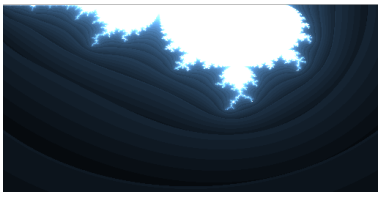
5.3 Rendering of Mandelbrot Set

We implemented the rendering process of the Mandelbrot set using our framework in multiple nodes. An array of distributed variables is created for each pixel of the image. Along, with that, a node failure handler callback function was added, which stores the state of the nodes

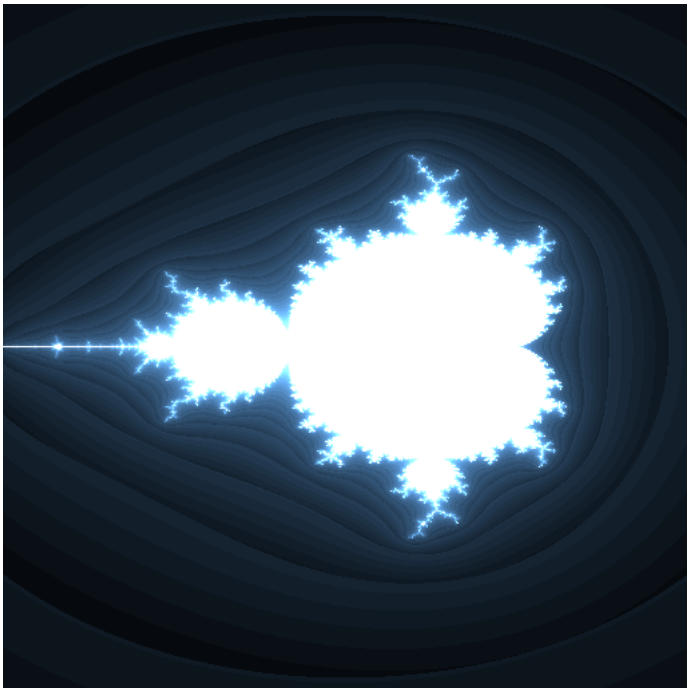
that has been already computed by the failed node, so that it can be dispatched once another node has been connected.



(a)



(b)



(c)

Figure 5.7: (a) Mandelbrot Set Computed in First Node (b) Mandelbrot Set Computed in Second Node (c) Complete Image of mandelbrot set obtained from both nodes

5.4 Distributed Whiteboard

The distributed whiteboard will be our second application of the framework besides Mandlebrot rendering. While the Mandelbrot was a demonstration of the distributed function call, the whiteboard is a presentation of the filesystem portion (GutFS) of our framework. The basic workflow of the whiteboard can be summarised as follows,

1. **Whiteboard - ElectronJS App:** The whiteboard is locally an app written using ElectronJS that features a simple canvas supporting draw feature using different colors. This can be run locally without connecting to the network.

2. **Making Changes (Locally):** Any changes made to the canvas are propagated to a locally stored file in base64 format, where any new changes overwrite the old changes.

3. **Propagating Changes (Across Nodes):** The file, to which the canvas changes are saved, is tracked by our framework's filesystem daemon. If any changes are made to the file by any node, then the changed file is sent to all nodes. Overwrites are made based on the newest made changes.

4. **Displaying Changes - Node Listener:** A separate listener is implemented to listen for changes to the changed file and reload it to the app. We have written this functionality in NodeJS which integrates seamlessly with the ElectronJS App, although the implementation detail is left up to the programmer. Any HTTP server that can re-render the changed file can be used for this portion.

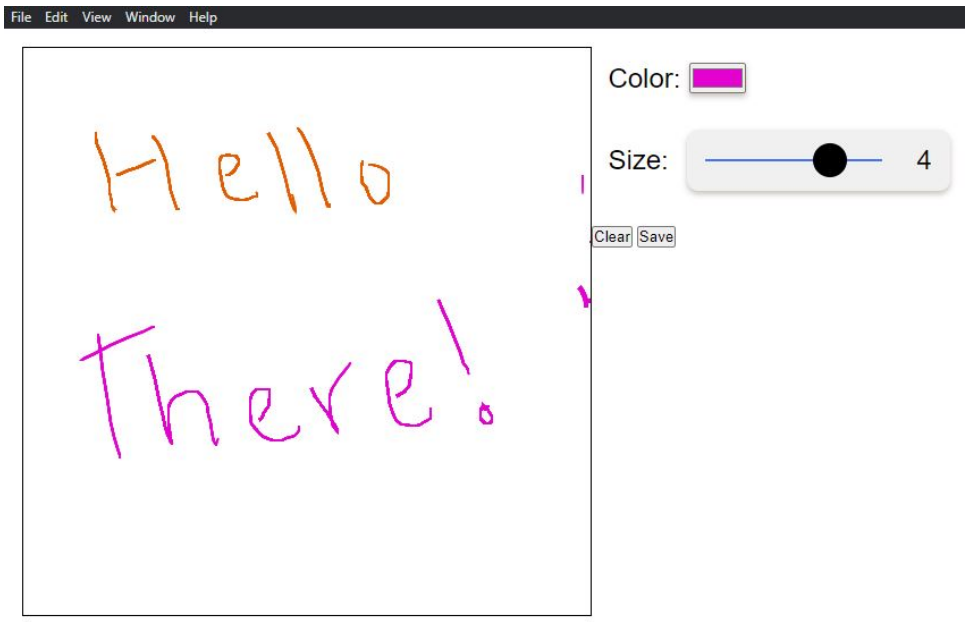


Figure 5.8: White Board Basic User Interface (Single Node Screenshot)

6. Conclusion

To summarize our report, the concept of our project has been derived from the need to utilize the computation power of systems connected in a network and we have been able to achieve a desirable output through our distributed framework. The background knowledge and theory required to establish a knowledge base have been provided, along with the methodology and mindset that is required to put that knowledge into practice for our framework. The implementation provides a fast, secure, and reliable way of communicating with nodes in a network, exchanging files, invoking functions, synchronizing timestamps, and more. The applications developed over our framework have been developed with an academic purpose in mind, with both the Mandlebrot rendering and Distributed Whiteboard intended to serve as examples for further research into mathematical computing and graphics rendering/file sharing respectively in distributed systems. Naturally, some rigorous analysis remains, along with changes proposed in our future enhancements. Overall, research can benefit from the algorithms and techniques implemented for this project, which can lead to new discoveries and advancements in the field of distributed computing study.

7. Limitations and Future enhancement

7.1 Limitations

Although a working framework was created, it still has some limitations. Some of the drawbacks and limitations of this framework are:

- **Multiple Mutual Exclusion Algorithm:** Currently, only one algorithm for the platform-level distributed mutual exclusion is provided. However, it can be inefficient for multiple use cases.
- **Absence of internal logs:** The framework does not provide logs of all internal events that could have been useful for programmers to debug their applications.
- **Presence of Bindings in multiple languages:** Currently, our framework exists for only the Go programming language and that can inherently come with limitations of the Go programming language.
- **Conflict resolution for distributed filesystem:** There is no concrete algorithm in place for conflict resolution to mutual edits to a distributed file other than prioritizing the latest timestamp. Operational transformation [12], a popular algorithm for supporting functionalities in advanced collaborative software systems, is currently being looked at as an alternative for this particular use case.
- **Scalability:** The framework has been developed with academic research in mind, so it doesn't scale well to the industry requirements of data processing.

7.2 Enhancements

- **API Exposure:** Exposing API for cross-language development is a crucial future enhancement that can open up the framework's usage to multiple research projects. The filesystem portion of the project is already language-independent due to its abstract nature, however, functionalities like remote function dispatch can be developed for use through multiple languages and tools.
- **Callback Functions:** Currently callback functions are used in the case of node failure handling, however, they can be extended to be executed in case of other events as well (node connection, distributed variable modification, and others).

- **Task Scheduling:** Functions can be set up to dispatch at a later time from invocation. Furthermore, a regularly scheduled dispatch can be integrated as a functionality of the system.
- **Variable Access Optimizations:** Variable access analysis can be done to minimize overhead to the system. For example, variable changes may not be propagated to nodes which do not actually require it's particular usage at the moment, thus saving network bandwidth.

References

[1] Apache Software Foundation. Apache™ hadoop® project, 2023.

[2] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2007 Eurosys Conference*. Association for Computing Machinery, Inc., March 2007.

[3] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, oct 2016.

[4] James E White. A high-level framework for network-based resource sharing. In *Proceedings of the June 7-10, 1976, national computer conference and exposition*, pages 561–570, 1976.

[5] Andrew D Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59, 1984.

[6] James Whitney, Chandler Gifford, and Maria Pantoja. Distributed execution of communicating sequential process-style concurrency: Golang case study. *The Journal of Supercomputing*, 75(3):1396–1409, 2019.

[7] Maarten Van Steen and Andrew S Tanenbaum. *Distributed systems*. Maarten van Steen Leiden, The Netherlands, 2017.

[8] Arif Sari, Murat Akkaya, et al. Fault tolerance mechanisms in distributed systems. *International Journal of Communications, Network and System Sciences*, 8(12):471, 2015.

[9] Ichiro Suzuki and Tadao Kasami. A distributed mutual exclusion algorithm. *ACM Transactions on Computer Systems (TOCS)*, 3(4):344–349, 1985.

[10] Glenn Ricart and Ashok K Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, 1981.

[11] Rfc 5905: Network time protocol version 4: Protocol and algorithms specification.

[12] Clarence A. Ellis and Simon J. Gibbs. Concurrency control in groupware systems. In *ACM SIGMOD Conference*, 1989.