Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Building Practical Distributed Algorithms

Distributed Systems Lab

Andrea Jiang, Catherine Schmit
jianga@student.ethz.ch, schmitc@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Yann Vonlanthen, Matej Pavlovic
Prof. Dr. Roger Wattenhofer

February 27, 2024

Acknowledgements

We would like to express our sincere gratitude to everyone who contributed to the success of this project. We extend our deepest appreciation to Matej Pavlovic and Yann Vonlanthen, our project supervisors, for their guidance, support, and encouragement throughout the duration of this project.

Abstract

Mir is a framework for implementing, debugging, and analyzing distributed protocols. It is created and maintained by Protocol Labs Research's ConsensusLab. It provides abstractions for an orchestration engine and different distributed system components. The framework aims to be general enough to support a wide range of storage, network transports, and cryptographic implementations. Our project aims to improve the debugging capabilities of Mir, making it a tool that facilitates the study of distributed algorithms by researchers, developers, and students. We enhanced Mir's debugging capabilities with a new debugger module written in Go, enabling easy interaction with Mir nodes through a new simple React interface. This facilitates visualization and manipulation of distributed algorithms on standard user laptops.

Contents

Chapter 1

Introduction

1.1 Mir Framework

Mir is a framework for implementing, debugging, and analyzing distributed protocols. It has the form of a library that provides abstractions representing different components of a distributed system and an engine orchestrating their interaction. It has been developed and currently maintened by the ConsensusLab, part of the Protocol Labs Research, focusing on exploring fundamental problems of coordination, consistency, and scalability in decentralised systems. [1] [2]

Mir aims to be general enough to enable implementing distributed protocols in a way that is agnostic to network transport, storage, and cryptographic primitive implementation. All these (and other) usual components of a distributed protocol implementation are encapsulated in abstractions with well-defined interfaces. While Mir provides some implementations of those abstractions to be used directly "out of the box", the consumer is free to provide their own implementations. [1]

1.2 Our Project

In this project we aim to leverage Mir to build a tool that allows researchers, developers and students alike to gain additional insights into distributed algorithms. We believe that this framework will have the potential to significantly support the development of new algorithms, by allowing faster implementing, debugging and better insights. Once an algorithm is developed, our tool can also help visualize it and explain it to others. Moreover, it could be used in teaching, both during lectures and hands-on exercises. [3]

We enhanced the Mir Framework's debugging capabilities by creating a new debugging module written in Go, the framework's primary language. This module facilitates user interaction with Mir nodes through a simple React interface accessible on common user laptops (by running it locally using a browser), enabling visualization and manipulation of Mir nodes.

Chapter 2

Design and Implementation

2.1 Methodology

During the development of the project, technical decisions were made after discussing ideas in weekly meetings. These meetings ensured that our objectives were aligned and that we tracked our progress. Our main communication tool was Slack, where we could also receive assistance from other Mir developers, in addition to our supervisors.

We chose Go as the programming language for the backend to align with the Mir framework, which is also written in Go. This decision facilitated seamless integration. After multiple discussions and experiments, we settled on React for the frontend due to its robust support and extensive functionality.

2.1.1 Communication Protocol

The choice of communication protocol between the frontend and backend was a pivotal architectural decision. We ultimately decided to use WebSockets instead of other alternatives such as REST API, GraphQL, or gRPC.

WebSockets offer a persistent, full-duplex communication channel over a single TCP connection [4]. In contrast, HTTP-based REST API and GraphQL follow a request-response paradigm, which inherently limits real-time interactions. Although gRPC supports streaming, its complexity and over-the-wire protocol, based on HTTP/2, may not be as straightforward for real-time web applications as WebSockets. WebSockets enable the server to push updates to the frontend without waiting for a request, ensuring immediate data synchronization and a dynamic user experience.

Additionally, WebSockets offer a significant reduction in latency compared to other communication protocols [4]. After the initial handshake, data can be transferred between the client and server with minimal overhead, eliminating the need for repeated HTTP headers. Although efficiency was not our primary objective, this still provides an advantage.

The simplified logic for real-time client-server interaction without multiple HTTP connections or managing HTTP/2 streams also aids in the development and scalability of our framework [5].

Websockets are also widely supported by modern web browsers [5], making our debugger accessible without specialized client software.

2.1.2 Messaging Formats

For sending messages between the frontend and the backend, we have decided to use both the ‘protojson‘ [6] and standard ‘json‘ [7] libraries.

JSON (JavaScript Object Notation) is an efficient, text-based format widely utilized for data interchange on the web. Renowned for its human-readable nature and broad compatibility across programming languages, JSON has become a universal standard for APIs and web services. Its compatibility with various frontend technologies further enhances its appeal. Moreover, the Go standard library's json package provides robust support for marshaling and unmarshaling Go data structures to and from JSON format.

Protojson is a component of the Protocol Buffers (Protobuf) ecosystem designed for serializing structured data. In Go, the protojson.Marshal function converts Protobuf messages into their JSON representation. This conversion proves invaluable in our case because, within the Mir system, events are Protobuf messages of type protobuf_oneof, which cannot be handled with the standard json.marshal and json.unmarshal methods. By leveraging protojson, we can preserve the structure of the Protobuf messages already in place.

2.1.3 Enhancing the Event Interceptor

In developing our framework, we utilized an existing codebase of the Mir framework that provided an interceptor capable of capturing events in real-time, both asynchronously and synchronously with the program's execution. The interceptor played a crucial role in our debugger implementation by serving as a point of interaction with the system's event flow. Recognizing its potential, we decided to modify the interceptor and associated functions to extend its functionality beyond event interception. Our enhancements transformed the interceptor into a tool that not only records events as they are executed but also influences their subsequent execution.

To accomplish this, we expanded the capabilities of the interceptor and all related functions. In addition to receiving a list of events as input, these functions were modified to also return a list of events. This returned list is then processed as the definitive set of events scheduled for execution. This modification allows us to observe and modify the sequence and nature of events in real-time by editing

or removing events from the event list. This improved interceptor served as the foundation for our debugger.

2.1.4 Frontend Framework

Given the absence of a previous frontend interface, during the selection of the most suitable technology, we had a wide choice with the only constraint being compatibility with common web browsers.

Our approach to selecting a frontend framework involved conducting thorough research on the most modern frameworks, their pros and cons, their compatibility with the existing backend, and considering preferences from the team.

At first, the easiest frontend framework seemed to be using Python's Flask due to its easily learning curve, ease of use, and as Python was known by all the members of the team. Unfortunately, after a brief trial to set up the websocket connections, we found that they are not supported, as the most common library is 'flask_socketio' which uses a different kind of socket.

Therefore, we chose to go with the language most supported for the frontend, JavaScript, and restricted the choice to its major frameworks: React, Vue.js, and Angular. We consulted multiple sources, with the most extensive being the "State of Frontend 2022" report by TSH [8], which provided insights into current industry trends and frameworks' popularity.

React emerged as the preferred frontend framework due to its popularity, suitability for our requirements, and previous experience of some members of the team. React is a JavaScript library for building user interfaces, known for its declarative and component-based approach. It enables developers to create reusable UI components, simplifying the development process and promoting code reusability. By leveraging a virtual DOM and efficient rendering techniques, React delivers optimal performance, making it suitable for building fast and responsive web applications. Its active community and vast ecosystem of libraries further enhance its appeal, providing solutions for various development needs. Overall, React's combination of simplicity, performance, and flexibility makes it a popular choice for modern web development projects [9].

Our implementation process began with a basic HTML and React setup (using npm and Node.js), gradually incorporating other features such as web components and connection with Mir module. However, challenges arose with web component usage, particularly regarding difficulties in unregistering web components after deletion. Therefore, we decided to replace web components with simpler React components. As we progressed, we also decided to simplify our approach to a straightforward HTML and simple React setup to streamline the user experience as they would no longer need an additional server running using npm and Node.js to support React. This choice brings some limitations in modu-

larity and possible future maintainability with also some issues such as managing dependencies, resolving syntax differences, and addressing CORS issues. But, in the end, it resulted in a better user experience, which was one of the main aspects of the project.

In the future, as the project gets more complex, other libraries/tools could be evaluated for integration (e.g., TypeScript, Material UI, Tailwind UI, Bootstrap, SCSS, Next.js), but for now, we deemed that they were not necessary and they would have only slowed down the development while bloating the codebase.

2.2 Implementation

2.2.1 WebSocket Interface

The Websocket Interface used by the backend and frontend to communicate is the following:

2.2.1.1 Connection

To connect the backend and the frontend, the following steps are necesserary

- Each Mir node that is initialized with the interceptor returned from the function NewWebSocketDebugger is running in 'debugger' mode. It starts a websocket server at the port address 'xxxx' given as an argument to NewWebSocketDebugger.
  For an example, see the pingpong application at `mir/samples/` , you can run it using `` `go run ./pinpong -d -port=8080 0` `` which would run the websocket server of the pingpong node 0 on port 8080).
- Before the node starts to run normally, it waits for a websocket client to connect. (The frontend will act as the client.)
- The frontend connects to the specified port using `` `ws://localhost:\${this.port}/ws` ``).
- Upon successful connection, the server sends an "init" event to the frontend.
- In response, the frontend sends a JSON object `{"Type": "start"}` to initialize the connection.
- The connection terminates either when the server stops or when the client disconnects, indicated by sending a message with `{"Type": "close"}`.

2.2.1.2 Event Format (JSON)

Events exchanged between the frontend and backend are formatted as JSON objects with the following structure:

```
{
  "event":     "protojson.Marshal(event)",
  "timestamp": "timestamp"
}
```

Notably, ‘event‘ undergoes double JSON parsing.

2.2.1.3 Frontend Response Format (JSON)

Responses from the frontend follow a specific JSON format:

```
{
  "Type": "",
  "Value": ""
}
```

2.2.1.4 Supported Response Types, Expected Values, and Actions

The table below summarizes the supported response types, their expected values, and the corresponding actions:

Type	Value	Frontend Action	Backend Action
start	null	Initialize the connection	Send the first event
accept	null	Add accepted event to the log and wait for the next event	Update the list of events with the accepted one and send the next event
decline	null	Drop the declined event and wait for the next event	Drop the declined event and send the next event
replace	Modified event (in the format mentioned in section 2.2.1.2)	Add modified event to the log and wait for the next event	Update the list of events with the modified event received and send the next event
close	null	Close the websocket connection	Stop sending data to the client

Table 2.1: Expected Action in Frontend and Backend after JSON Response

2.2.2 Overview of Backend Functions

2.2.2.1 NewWebSocketDebugger

Input Parameters: ownID: t.NodeID, port: string, logger: logging.Logger
Return Elements: *eventlog.Recorder, error
Returns the interceptor required for debugging a given node using the nodes ID, a port number for establishing the WebSocket connection, and a logger. The returned interceptor can be used to initialize the node for debugging.

2.2.2.2 WSWriter

WSWriter is a struct that facilitates WebSocket communication between the backend and frontend by managing the connection, sending events to the frontend, and receiving messages back. It implements methods for flushing, closing the connection, and writing events to the frontend.

- **Flush**
  Input Parameters: wsw: *WSWriter
  Return Elements: error
  This method is currently not implemented, as we had no use for it.
- **Close**
  Input Parameters: wsw: *WSWriter
  Return Elements: error
  This method closes the WebSocket connection if it is open.
- **Write**
  Input Parameters: list: *events.EventList, timestamp: int64
  Return Elements: *events.EventList, error
  This method waites for a websocket connection to be established and then transforms each event from the input list into a json and sends it to the frontend. It then waits for a message from the frontend detailing how to proceed with that event. All accepted events are returned in a list.

2.2.2.3 newWSWriter

Input Parameters: port: string, logger: logging.Logger
Return Elements: *WSWriter
Initializes a new instance of WSWriter by configuring the WebSocket Upgrader, initializing an eventSignal channel for handling messages from the frontend asynchronously, and assigning a logger for logging purposes.
Additionally, an HTTP handler is registered for the ‘currentURL/ws‘ endpoint. This handler is responsible for upgrading incoming HTTP requests to WebSocket

connections using the Upgrader configured in the WSWriter object.
For each received message, the function attempts to unmarshal it into a signal map. Specific actions are taken based on the signal's contents, such as handling client commands or terminating the connection upon receiving a 'close' command.
To ensure that the WebSocket server is non-blocking and can handle connections asynchronously, the function spawns a goroutine that starts an HTTP server on the specified port. This allows the backend to continue executing other tasks while listening for incoming WebSocket connections.

2.2.2.4 HandleClientSignal

Input Parameters: wsw: *WSWriter
Return Elements: signal map[string]string
Handles signals received from the frontend and passes them to the event signal channel for processing.

2.2.2.5 EventAction

Input Parameters: actionType: string, value: string,
acceptedEvents: *events.EventList, currentEvent: *eventpb.Event
Return Elements: *events.EventList, error
Determines the next action to take based on the input action type and value. If the action type is to accept the current event, the event is pushed onto the list of accepted events. If the action type is to replace the current event, the new event stored in value is unmarshalled and pushed to the list of accepted events instead of the original event. If the action type is to decline the current event, the event is dropped from the returned list.

2.2.3 Frontend Components and Functions

The frontend is, for ease of final user's use, composed by only 2 files: index.html and index.css . The index.html file cointains the HTML and JavaScript for establishing WebSocket connections, managing incoming and outgoing messages and how the frontend evolves dynamically. While the index.css contains the CSS attributes to improve the user interface.

We will focus on the index.html file as the it contains the main logic. It is made by 2 React components: 2.2.3.1WebSocketConsole and 2.2.3.2App.

2.2.3.1 WebSocketConsole Component

Functionality

This component represents a WebSocket console for a specified port. It displays incoming messages and allows the user to accept, replace, or decline them. In additional there is the possibility to close the connection, and to automatically accepts the incoming messages.

Input Parameters

- **port**: `number` - The WebSocket's server port number to connect to as a client.

Return Elements

The component renders a WebSocket console interface in HTML, allowing users to interact with incoming messages and the WebSocket connection itself.

States and References

These states are used within the WebSocketConsole component to manage the WebSocket connection, incoming messages, errors, and logged messages.

- **ws**: Represents the WebSocket connection object. It is created using `new WebSocket(\`ws://localhost:\${port}/ws\`)`
- **incomingMessage**: Represents the incoming message received from the WebSocket server.
- **editableText**: Represents the incoming message once it is parsed, to be editable by the user. It is also the "value" showed to the user.
- **errors**: Represents the array of potential error messages displayed to the user.
- **loggedMessages**: Represents an array of messages that have been accepted.
- **incomingMessageJSON**: Represents a reference to the parsed JSON object of the incoming message. Used to ease the parsing and deparsing of the messages.
- **syncConnection**: Indicates whether the current connection is Synchronous or Asynchronous. Indicating whether to automatically accept the incoming messages without the user needing to accept or decline.

useEffect Hooks

In the following section, we explain the useEffect hooks used in the WebSocket-Console component:

- **Connecting to WebSocket**:
  - **Functionality**: Connects to the given WebSocket port after the component rendering.
  - **Dependencies**: None
  - **Effect**: Establishes a WebSocket connection and sets up event listeners for open, message, close, and error events.
  - **Cleanup**: Closes the WebSocket connection and removes event listeners when the component unmounts.
- **Parsing Incoming Message**:
  - **Functionality**: Parses the incoming message to make it editable whenever there is a new incoming message.
  - **Dependencies**: `incomingMessage`
  - **Effect**: If the incoming message is not null or empty, parses it into JSON format and converts it to editable text using the `jsonToEditableText` function.
- **Syncronization**:
  - **Functionality**: Manage the messages path when moving between Syncronous and Asyncronous connection.
  - **Dependencies**: `syncConnection`
  - **Effect**: Update the listener on the 'message' event websocket by calling again `handleMessageWebSocket` that will change the incoming messages path according to the syncConnection state.

Functions

In the following section, we explain the functions used in the WebSocketConsole component:

- **handleOpenWebSocket**
  - **Input Parameters**: None
  - **Return Elements**: None
  - **Description**: Updates logged messages and clears errors messages.

- **handleMessageWebSocket**
  - **Input Parameters**: `event` (WebSocket event object)
  - **Return Elements**: None
  - **Description**: Set the incoming message and triggering further processing. Also decides how to manage the message depending on the syncConnection state value.
- **handleErrorWebSocket**
  - **Input Parameters**: `event` (WebSocket event object)
  - **Return Elements**: None
  - **Description**: Logs the error, and updates the errors' array state.
- **handleCloseWebSocket**
  - **Input Parameters**: None
  - **Return Elements**: None
  - **Description**: Adds a message indicating the closure of the WebSocket connection to the errors array.
- **addLoggedMessage**
  - **Input Parameters**: `newMessage` (string)
  - **Return Elements**: None
  - **Description**: Push the new message at the start of the logged messages
- **decomposeJSON**
  - **Input Parameters**: `message` (string)
  - **Return Elements**: Decomposed JSON message (string)
  - **Description**: Parses the incoming JSON message and returns the decomposed message.
- **clearIncomingLogMessages**
  - **Input Parameters**: None
  - **Return Elements**: None
  - **Description**: Clears the incoming message, editable text, and resets the incoming message JSON reference.
- **acceptIncomingLog**
  - **Input Parameters**: None

  - **Return Elements**: None
  - **Description**: Accepts the original event message, sends an ‘accept‘ response on the WebSocket Server, and clears the input.

- **acceptAsyncIncomingLog**
  - **Input Parameters**: `message` (string)
  - **Return Elements**: None
  - **Description**: Accepts the original event message, sends an ‘accept‘ response on the WebSocket Server, and clears the input. Used when in Async mode.

- **replaceIncomingLog**
  - **Input Parameters**: None
  - **Return Elements**: None
  - **Description**: Accepts the replaced log message, sends a ‘replace‘ response on the WebSocket Server, and clears the input.

- **declineIncomingLog**
  - **Input Parameters**: None
  - **Return Elements**: None
  - **Description**: Sends a ‘decline‘ response on the WebSocket and clears the input.

- **closeConnection**
  - **Input Parameters**: None
  - **Return Elements**: None
  - **Description**: Closes the current WebSocket connection, ‘decline‘ the current pending event and sends a ‘close‘ message.

- **changeSyncronization**
  - **Input Parameters**: None
  - **Return Elements**: None
  - **Description**: Toggles synchronization mode and ‘accepts‘ incoming logs if switching to asynchronous mode.

- **EditableText**
  - **Input Parameters**: `text` (string), `textType` (string, indicating the JSON type of the value inside 'text'), `jsonReference` (object, parent JSON object), `jsonReferenceKey` (string, key to access value from parent JSON object), `onUpdate` (function, indicating what to do onUpdate event)

  - **Return Elements**: JSX element
  - **Description**: Modifiable text component. Renders a text component that can be edited. Handles 'onClick', 'onBlur', and 'onChange' events to update its text.

- **jsonToEditableText**
  - **Input Parameters**: `json` (object), `parentJson` (object, parent JSON object), `parentKey` (string, key to access value from parent JSON object), `depth` (number, level of the tree for the indentation)
  - **Return Elements**: JSX element
  - **Description**: Recursively converts a JSON object to another JSON object with leafs converted into editable text. It also maintains the different JSON types after modification (expect for 'null' as once it is modified it will become a 'string' without possibility to change it back to 'null')

2.2.3.2 App Component

Functionality

This component represents the default main application interface. It allows users to connect to WebSocket consoles for different ports by inserting the port number.

Input Parameters

None

Return Elements

The component renders an interface for entering port numbers and displaying WebSocket consoles:

States and References

These states are used to manage the WebSocket Consoles.

- **webSocketConsoles**: State variable to store the array of WebSocket consoles.
- **port**: State variable to store the inputed port number.

Functions

In the following section, we explain the functions used in the WebSocketConsole component:

- **connectWebSocket**
  - **Input Parameters**: None
  - **Return Elements**: None
  - **Description**: Connects to a WebSocket using the provided port number. Checks whether the number in the state variable 'port' is empty or we are already connected to such a port. If not, it creates a new WebSocket console with the provided port number, updates the webSocketConsoles state with the new console, and clears the port state.

2.3 User interface

The user interface below has been created to support the functionalities described above, facilitating communication through websockets with the backend. To achieve this, the following components have been implemented:

(It is worth noting that the current interface prioritizes delivering functionalities and does not yet fully implement best UI/UX practices. However, due to its use of React and simple HTML, introducing a better interface around these functionalities should be straightforward. For example, the Sync/Async toggle has been created from a simple checkbox using only simple CSS.)

With reference to Figure 2.1, the following sections can be observed:

1. **Websocket Port Input**: Here, users can input the desired websocket port to connect to, by clicking 'Connect'.
2. **Websocket Console Grid**: This section displays all connected websocket consoles in a grid with 2 columns.
3. **Websocket Console**: Each console interface is displayed here, including various tools for interaction.
4. **Incoming Log**: New Event Logs are displayed here, with editable values highlighted by a surrounding box.
5. **Button Area**: This area contains buttons to determine the fate of the new Event Log.
6. **Accept Button**: Clicking this button accepts the **original** Event Log, even if modified, it will "drop" the modifications.

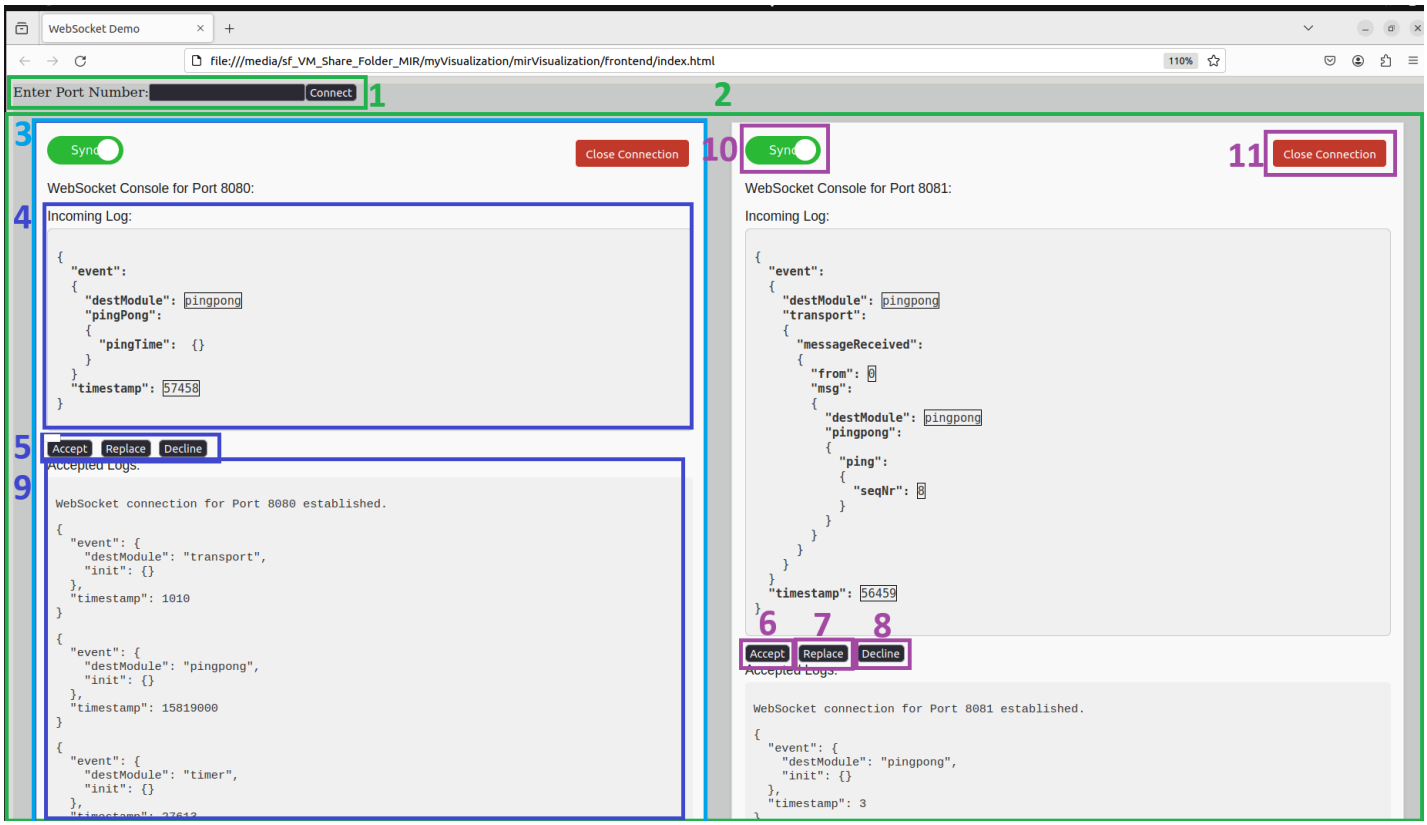

Figure 2.1: Frontend of the debugger

7. **Replace Button**: This button accepts the **modified** Event Log displayed on screen. The modification can be made in the 'Incoming Log' section by clicking inside the highlighted boxes, which activate an input box around the desired value, allowing the user to edit the content. To commit the changes, simply click outside the box. Once all changes are made, clicking 'Replace' accepts the modified Log.

8. **Decline Button**: Clicking this button declines Event Log.

9. **Accepted Log**: This section displays old Accepted and Replaced Event Logs in a stack order, with the newest at the top and the oldest at the bottom.

10. **Sync/Async Toggle**: This toggle changes the mode of Acceptance. In 'Sync' mode, users can interact with the incoming Event Log. In 'Async' mode, the incoming message is automatically accepted, as if the system was not in debugging mode.

11. **Close Connection Button**: Clicking this button closes the connection with the websocket server and stop the node running.

Chapter 3

User guide

Using the new debugger is quite simple. There are two aspects to use it: the backend and the frontend. In the backend, you need to specify that you will be running in debug mode and initialize any nodes that you want to analyze with the right interceptor. The debugger frontend is quite intuitive and you will need to simply open it by double-clicking the respective file.

3.1 Use The Debugger

1. **Start debugger on the Mir Node(s) to debug**: Initialize the Mir Node with the interceptor returned by NewWebSocketDebugger in `pkg/debugger/debugger.go`. Then you can simply run the node as usual. You have to provide the NewWebSocketDebugger function with the nodes ID, a logger and a port number over which the websocket connection will be established. Each Mir module should run on a **different** port. For simplicity, we suggest using ports 8080+node_identification, therefore 8080, 8081, 8082, etc.

   You can look at the pingpong sample found in `` `mir/samples/` `` for an example. For a step-by-step guide on how to run it, please see section 3.2.

2. **Open the frontend**: To open the user interface, you simply need a browser (e.g., Chrome, Firefox). Navigate to the folder `` `mir/frontend/` `` and double-click the file `index.html`. This action will open a new page in your browser, which will be the interface for interacting with the Mir modules. For further details on its functionality, please refer to Section 2.3.

3. **Connect to the Mir module**: Once you have opened the frontend and started the Mir node in debugging mode, wait until the Mir node starts (it will begin printing `"Waiting interface connection to proceed"` in the terminal). If this is not the case and it prints other information, double-check that you started the node in debugging mode as described above.

Once the node has started, connect to it by inserting its port (`xxxx` (e.g. 8080), the one you indicated when initializing the interceptor for this node). This will open a new console on the browser page.

4. **Interact with the console**: Please refer to Section 2.3 for details.

5. **Close the connection**: You can close the connection by either terminating the Mir node through the terminal or clicking the 'Close Connection' button in the console. This action will drop the connection and stop the node in question.

3.2 Debugger Usage Example

To run our ping-pong example already supporting the debugger mode, you can follow these steps:

1. **Clone the git repository:** To clone the repository, open a terminal and run the following command:

   ```
   git clone https://github.com/consensus-shipyard/mir.git
   ```

   To ensure that you are using the exact version of Mir intended for this guide, checkout the specified commit hash with:

   ```
   cd <your-repository-directory>
   git checkout 1054bff
   ```

2. **Compiling and running Mir:** For details on how to install dependencies and generate the necessary files, follow the instructions provided in `` `README.md` ``, located in the root of the repository, specifically the section 'Compiling and running tests'.

3. **Run pingpong:** The pingpong example can be found in `` `mir/samples/` `` You can run the pingpong application on two modes: with or without the debugger active. In a normal run you would run both nodes with:

   ```
   go run ./samples/pingpong 0
   go run ./samples/pingpong 1
   ```

   where 0 and 1 indicate the respective node ID. It will become:

   ```
   go run ./samples/pingpong -d -port=8080 0
   go run ./samples/pingpong -d -port=8081 1
   ```

   where we have specified to run them in debug mode using port 8080 for node 0 and port 8081 for node 1.

- `-d` to indicate it will run in debugging mode.
- `-port=xxxx` to indicate on which port `xxxx` the websocket server (used to communicate with the user interface) will run.

When running in debug mode, the node is simply initialized with the interceptor of the debugger.

4. **Open and interact with the frontend:** To use and interact with the frontend, simply follow steps 2 - 5 of the user guide.

Chapter 4

Conclusion

Our project successfully completed the development of a new debugger module for the Mir framework, along with its user interface. This module enhances Mir's debugging capabilities, allowing users to interact with Mir nodes through a simple and intuitive interface built with React. By providing visualization and manipulation tools, our debugger module assists researchers, developers, and students in studying and understanding distributed algorithms more effectively.

For example, a real-life application of the 'drop' and 'replace' functionality within the debugger module is to experiment with how a distributed system responds to unexpected messages or network failures. This experimentation can lead to the creation of a more resilient and robust system.

Looking ahead, there are several potential functionalities that could be incorporated into the debugger module to further enhance its utility and usability. One such addition could be advanced visualization tools, such as data flow diagrams, to offer users a comprehensive overview of the distributed system's structure and communication patterns. Additionally, we could implement a default mode for certain type of events, allowing users to accept or deny them, or another filter functionality that would display only a subset of events to the user. Different additional functionalities could also be added based on user needs, and given the open-source nature of the project, we hope users will contribute to enriching it.

Outside the project itself, our involvement in working on an open-source project like Mir significantly enriched our knowledge of Git version control and collaboration practices. Through contributing to a real-world project, we gained hands-on experience in managing code repositories, collaborating with team members, and adhering to best practices in software development. This exposure not only improved our technical skills but also fostered a deeper understanding of the collaborative nature of modern software development processes. Overall, our project not only achieved its technical objectives but also provided valuable learning experiences.

Bibliography

[1] ConsensusLab. Mir - the distributed protocol implementation framework. [Online]. Available: https://github.com/consensus-shipyard/mir

[2] ConsensusLab. Consensuslab. [Online]. Available: https://consensuslab.world/

[3] R. Wattenhofer, M. Pavlovic, and Y. Vonlanthen. Building practical distributed algorithms. [Online]. Available: https://tik-db.ee.ethz.ch/file/678ffea862b2b577d30b2bde175634c4/_Bait__Building_Practical_Distributed_Systems.pdf

[4] Alex Diaconu, "Websockets: Pros and cons," 2021, accessed on February 16, 2024. [Online]. Available: https://ably.com/topic/websockets-pros-cons

[5] "What are websockets?" accessed on February 16, 2024. [Online]. Available: https://www.pubnub.com/guides/websockets/#h-12

[6] "Package protojson," 2023, accessed on February 16, 2024. [Online]. Available: https://pkg.go.dev/google.golang.org/protobuf/encoding/protojson

[7] "Package json," 2024, accessed on February 16, 2024. [Online]. Available: https://pkg.go.dev/encoding/json

[8] T. S. House, "The state of frontend 2022," 2022. [Online]. Available: https://tsh.io/state-of-frontend/#report

[9] R. Team, "React - a javascript library for building user interfaces," 2024, accessed on February 14, 2024. [Online]. Available: https://react.dev/