Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Building Practical Longest Chain Protocols

Distributed Systems Laboratory

Marc Widmer

widmmarc@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Protocol Labs
Research

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

February 5, 2024

Acknowledgements

I thank Matej Pavlovic and Yann Vonlanthen for their guidance and assistance with this project. Discussing various aspects of this project and everything around it with you was always a great joy. I am looking forward to continue working with you during my master's thesis.

Abstract

This project presents a carefully designed modular abstraction of a longest-chain consensus protocol and an implementation of it in the Mir framework. It outlines how one can use this implementation as a consensus layer in other applications and provides an implementation of a simple chat app as a reference. Further, it also presents a rudimentary example of a visualization tool for the longest-chain consensus system.

Contents

Chapter 1

Introduction

This project aims to develop a modular abstraction for a longest-chain consensus protocol, taking inspiration from the well-known Bitcoin Blockchain [1]. Our approach focuses on cleanly encapsulating core functionalities into simple modules with carefully designed interfaces. The result is a thoroughly documented implementation of this protocol in the Mir framework.

Mir [2] is a framework designed to implement and debug distributed algorithms written in Go. In Mir, a distributed algorithm consists of multiple nodes that communicate over a network to execute a protocol jointly. Every node is itself built up of one or multiple modules. Each module should encapsulate a distinct set of functionality and let other modules interact with it through a carefully designed interface. These interactions are performed via events that are produced, consumed, and processed by the modules.

Although the result of this project is a fully functional system, it is not intended as a 'production' system; rather, it should serve as the basis for possible future work to extend this system to simulate different network conditions or node behaviors. For example, one could simulate "Selfish Mining[3]", where a set of colluding miners don't share the blocks that they mined with other nodes for a while in order to grow a competing branch.

The following will describe the design of this system and how it is implemented in Mir.

The source code for this project can be found on GitHub[1].

Note: The rest of this report (except for the conclusion) consists of parts taken from the read-me for this project. I recommend reading the read-me directly for a more pleasant experience.

[1] https://github.com/komplexon3/mir/tree/longest-chain-consensus/pkg/blockchain

Chapter 2

High-Level Architecture

We will first outline the different parts that make up the system. How these elements interact with each other will be described in Operation, and a more detailed description of every element can be found in Modules.

Each node has a set of core modules running the blockchain and an application module that runs the business logic and provides certain functionality to the core modules.

The blocks making up the blockchain contain the following:

- **block id:** Identifier of a block, computed by hashing the block with the block id set to 0.
- **previous block id:** Identifier of the predecessor block.
- **payload:** An application dependant payload.
- **timestamp:** Timestamp of when the block was mined.
- **miner id:** Identifier of the node that mined the block.

Every node keeps track of all nodes that it knows of. At the very beginning, this is solely the genesis block. After a couple of blocks have been mined, all nodes together form a tree of blocks. Whichever leaf of this tree is the deepest is called the head, and the chain of blocks from the genesis block to this leaf is the canonical chain. The payloads of all the blocks in the canonical chain together define the current state stored in the blockchain.

The nodes consist of the following core modules:

- **Blockchain Management Module (BCM):** It forms the core of the system and is responsible for managing the blockchain.
- **Miner Module:** Mines new blocks by simulating proof-of-work.

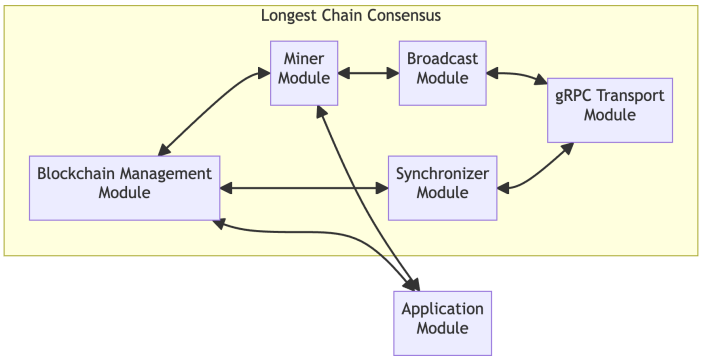

Figure 2.1: High-Level Architecture

- **Synchronizer Module:** Resolves issues when new blocks are to be added to the blockchain, but their parent blocks are unknown to this node's BCM.
- **Broadcast Module:** Broadcasts newly mined blocks to all other nodes. It can also simulate network delay and dropped messages.

, and the following supporting modules (provided by Mir):

- **gRPC Transport Module:** Used for communication between nodes.
- **Event Mangler Module:** Used by the broadcast module to simulate network delay and dropped messages.
- **Timer Module:** Used by the Miner to simulate proof-of-work.

Lastly, a user-implemented **Application Module** handles all business logic. In particular, it needs to compute the state of the blockchain, verify payloads, and provide payloads to the miner.

Next to the modules, it also includes an interceptor that intercepts all communication between different modules and allows for visualization/debugging tools to consume this communication via a websocket connection. An example of how to use the information provided by the interception can be found on GitHub[1].

Note: The event mangler module and the timer module were omitted from High-Level Architecture figure for simplicity.

[1] https://github.com/komplexon3/longest-chain-project/tree/main/chain-visualizer

Chapter 3

Operation

We will now walk through how the different modules interact with each other.

Note: In the sequence diagrams accompanying the following discussion (3.1, 3.2, 3.3), the boxes with a dotted outline have a different meaning depending on the label in the top left corner:

- loop: The sequence in the box repeats indefinitely or until the condition in the brackets holds.
- alt: The two boxes making up this box describe two alternative sequences.
- opt: The sequence in the box is optional. It is performed if the condition in the boxes holds.
- par: The two boxes making up this box describe two sequences that are performed in parallel.

At the start, the application module must initialize the BCM by sending it an `InitBlockchain` event, which contains the initial state. The BCM creates a genesis block with an empty payload and stores it together with the initial state. It then instructs the miner module to start mining via a `NewHead` event. To start mining, the miner module requests a payload (`PayloadRequest`) from the application module. The miner now starts to mine, and the initialization sequence of the node is completed.

After this, the node will remain inactive (other than mining a block) until a new block has been mined. This block was either mined by this node's miner or by another node's miner. In the first case, the miner would send the new block to its BCM and broadcast the block to all other nodes via the broadcast module (`NewBlock` event). In the second case, the broadcast module would have received the block from the other node's broadcast module and sent it to the BCM as a `NewBlock` event.

The BCM will then check whether or not it can connect this block to its tree of blocks. If it cannot connect the block, we call it an orphan block. Such cases

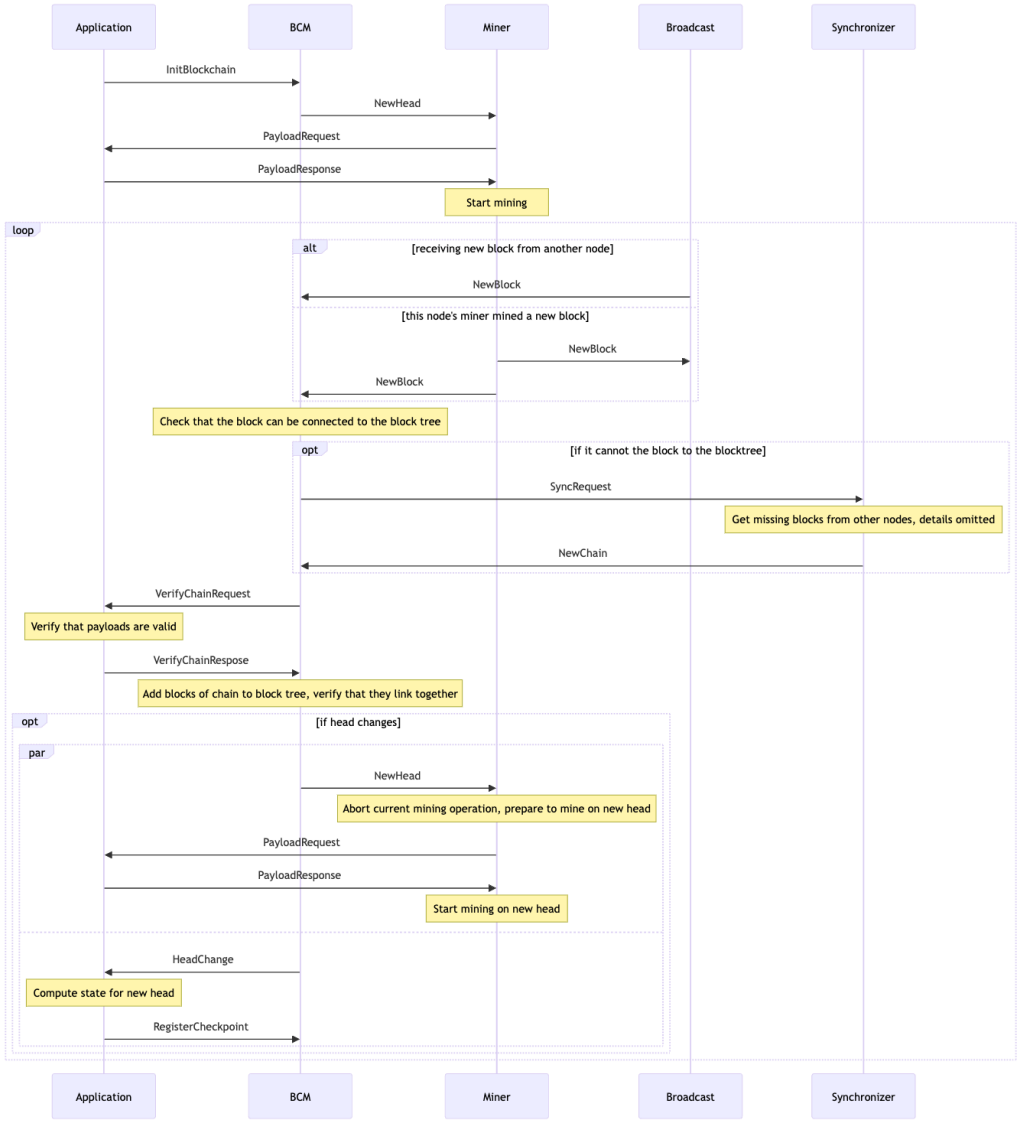

Figure 3.1: System Operation

can occur if the block's parent was mined by another node and the broadcast message for this block has not (yet) reached this node. The BCM will try to resolve these problems with the help of the synchronizer, which will coordinate with other nodes to get the missing blocks. This procedure will be described in more detail below.

The BCM now potentially has multiple blocks to add, which are the new block and possibly a chain of additional blocks from the synchronizer. The blocks are not trusted by the BCM as they might have been mined by another node and must, therefore, be verified. This happens in two ways:

1. The BCM sends all the blocks together with some additional information to the application module (`VerifyChainRequest`). The application then verifies that the payloads are valid. This logic is application-specific. 2. The BCM verifies that the nodes link together correctly.

The BCM can now add all new blocks to the block tree. If the canonical chain changes, i.e., there is a new head, it instructs the miner to start mining on the new head (`NewHead` event). Also, it informs the application about the new head (`HeadChange` event). This event contains some additional information about how the canonical chain changes. For example, if a different branch of the tree is now the canonical chain, it also includes which blocks are no longer part of the canonical chain. This allows for the application to resubmit these payloads if desired. In any case, the application will compute the state corresponding to the new head and register it in the BCM (`RegisterCheckpoint`).

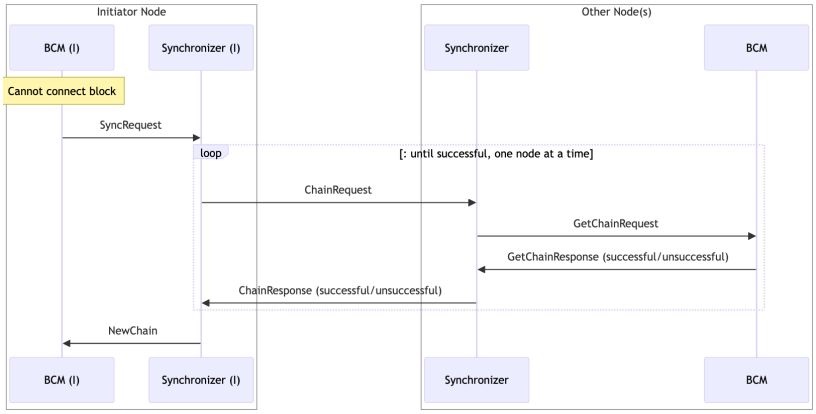

Figure 3.2: Operation Synchronizer

The functionality of the synchronizer was already outlined above. This part will go into more detail on how the synchronizer resolves orphan blocks. Every `SyncRequest` from the BCM contains the id of the orphan block and a collection of id of the block that the BCM has in its tree. With this information, the synchronizer asks one node after another to give it a chain that connects the orphan block to one of the known blocks (`ChainRequest`). The other nodes' synchronizers then query their BCM via a `GetChainRequest` for such a segment and forward the answer back to the initiator's synchronizer. The responses can be unsuccessful. In that case, the synchronizer asks the next node. When a successful response is received, the synchronizer instructs the BCM to add the new chain to the block tree (`NewChain` event).

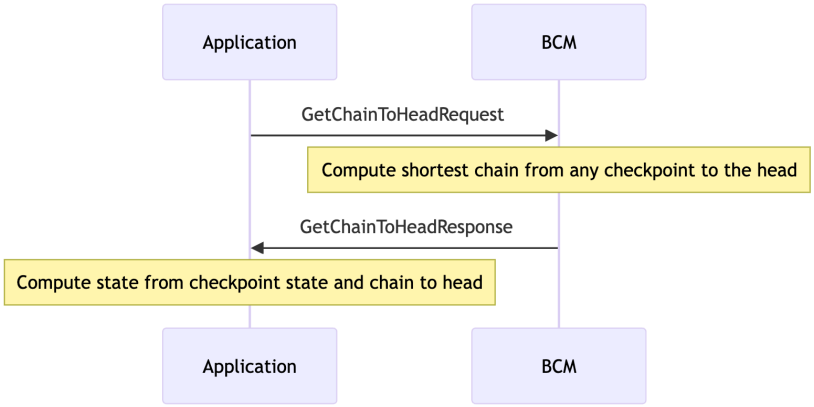

Figure 3.3: Operation State Query

At any point in time, the application can get the current state at the head of the blockchain by sending a `GetChainToHeadRequest` to the BCM. The response to this will include a chain of blocks from a checkpoint to the current head and the state associated with the checkpoint. Using this information, the application can compute the current state.

Chapter 4

Modules

After outlining the modules that make up a node and describing how they interact, the following will describe the modules' functionality in a bit more detail.

4.1 Blockchain Management Module (BCM)

The blockchain manager module is responsible for managing the blockchain. It keeps track of all blocks and links them together to form a tree. In particular, it keeps track of the genesis block, the head of the blockchain, all leaves, and so-called checkpoints. A checkpoint is a block stored by the BCM that has a state stored with it. Technically, checkpoints are not necessary as the state can be computed from the blocks. However, it is convenient not to have to recompute the state from the genesis block every time it is needed.

The BCM must perform the following tasks:

1. Initialize the blockchain by receiving an `InitBlockchain` event from the application module, which contains the initial state that is associated with the genesis block.
2. Add new blocks to the blockchain. If a block with a parent that is not in the blockchain is added, the BCM requests the missing block from the synchronizer. Blocks that are missing their parent are called orphans. All blocks added to the blockchain are verified in two steps:
   - It has the application module verify that the payloads are valid given the chain that the block is part of.
   - The BCM must verify that the blocks link together correctly.

   Additionally, it emits a `TreeUpdate` event. This is solely for debugging/visualization purposes and is not necessary for the operation of the blockchain.
3. Register checkpoints when receiving a `RegisterCheckpoint` event from the application module.

4. Provide the synchronizer with chains when requested. This is to resolve orphan blocks in other nodes.

5. When the head changes, it sends a `HeadChange` event to the application module. This event contains all information necessary for the application to compute the state at the new head as well as information about which payloads are now part of the canonical (i.e., longest) and which ones are no longer part of the canonical chain. Also, it instructs the miner to start mining on the new head (`NewHead` event).

6. Provide a chain of blocks from a checkpoint to the current head and the state associated with the checkpoint when receiving a `GetChainToHeadRequest`. This is used by the application to query the current state.

4.2 Miner Module

The miner module is responsible for mining new blocks. It simulates the process of mining a block by waiting for a random amount of time and then broadcasting the mined block. This random amount of time is sampled from an exponential distribution with a mean of `expMinuteFactor` minutes. The mining is orchestrated by a separate goroutine (`mineWorkerManager`) such that the miner module can continue to receive and process events.

The operation of the miner module is as follows:

1. When it is notified of a new head (`NewHead` event), it prepares to mine the next block by sending a `PayloadRequest` event to the application module. If it is already mining a block, it aborts the ongoing mining operation.

2. When it receives the `PayloadResponse` containing a payload for the next block, it starts mining a new block with the received payload.

3. When it mines a new block, it broadcasts it to all other modules by sending a `NewBlock` message to the broadcast module. It also shares the block with the blockchain manager module (BCM) by sending a `NewBlock` event to it.

4.3 Broadcast Module

The broadcast module is responsible for broadcasting new blocks to all other nodes. It either does this directly via the transport module or the mangler (parameter `mangle`). If the mangler is used, messages might be dropped and delayed. How many messages should be dropped can be configured by the parameter `dropRate` and the delay can be configured by the parameters `minDelay` and `maxDelay`.

4.4 Synchronizer Module

The synchronizer module assists the BCM in resolving cases when BCM receives an orphan block. An orphan block is a block that cannot be linked to the blockchain because the blockchain does not contain the block that the orphan block is linked to. To do this, the synchronizer module communicates with other nodes to get the missing blocks.

Terminology:

- **internal sync request:** a request to synchronize a chain segment that was initiated by this node
- **external sync request:** a request to synchronize a chain segment that was initiated by another node

The synchronizer module performs the following tasks:

For internal sync requests:

1. When it receives a `SyncRequest` event, it must register the request and send a `ChainRequest` message to one of the another nodes.
2. When it receives a successful `ChainResponse` message, it sends the BCM the chain fixing the missing bit with a `NewChain` event. It then deletes the request.
3. When it receives an unsuccessful `ChainResponse` message, it sends a `ChainRequest` message to the next node. If there are no more nodes to ask, it deletes the request.

For external sync requests:

1. When it receives a `ChainRequest` message, it must register the request and send a `GetChainRequest` event to the BCM.
2. When the BCM responds with a `GetChainResponse` event, the synchronizer responds to the node that sent the `ChainRequest` message with a `ChainResponse` message.

Note: This module assumes that all other nodes respond to requests. For this reason, the messages sent from the synchronizer do not go through the mangler.

4.5 Application Module

The application module is responsible for performing the actual application logic and interacting with users. It does not hold any persistent state but instead relies on the BCM to store the state. However, the application is responsible for computing the state given a chain of blocks and a state associated with the first block in the chain. Also, the application module manages payloads and must provide payloads for new blocks to the miner.

The application module must perform the following tasks:

1. Initialize the blockchain by sending the initial state to the BCM in an `InitBlockchain` event.
2. When it receives a `PayloadRequest` event, it must provide a payload for the next block. Even if no payloads are available, a payload must be provided; however, this payload can be empty.
3. When it receives a `HeadChange` event, it must compute the state at the new head of the blockchain. This state is then registered with the BCM by sending it a `RegisterCheckpoint` event. A checkpoint is a block stored by the BCM that has a state stored with it.
4. When it receives a `VerifyBlocksRequest` event, it must verify that the given chain is valid at an application level and respond with a `VerifyBlocks Response` event.

The websocket interceptor intercepts all events and sends them to a websocket server. Any connected client can then receive these events by subscribing to the websocket server. The interceptor proto file defines events that are specifically intended for the interceptor and not used by the actual blockchain. Since these events technically don't have a destination module, they are sent to the "null" module (ignores all incoming events). However, all events are intercepted and sent to the websocket server.

For this implementation, there are two events intended for the interceptor:

- `TreeUpdate`: This event is sent by the blockchain manager (BCM) when the blockchain is updated. It contains all blocks in the blockchain and the id of the new head.
- **StateUpdate**: This event is sent by the application when it computes the state for the newest head of the blockchain.

Chapter 5

Creating your own application

To create your own application based on this blockchain system, you must first define your application's state and payloads in `statepb.proto` and `payloadpb.proto` files.

Next, you must implement an application module that performs the following actions:

1. Initialize the blockchain by sending an initial state as an `InitBlockchain` event to the BCM.
2. When it receives a `PayloadRequest` event, it must provide a payload for the next block. Even if no payloads are available, a payload **must** be provided. However, this payload can be empty.
3. When it receives a `HeadChange` event, it must compute the state at the new head of the blockchain. This state must then be registered with the BCM by sending it a `RegisterCheckpoint` event. Additionally, the information provided in the `HeadChange` event might be useful for payload management.
4. When it receives a `VerifyBlocksRequest` event, it must verify that the given chain is valid at an application level and respond with a `VerifyBlocks Response` event.

See the example blockchain chat app as a reference.

You can now set up all the modules using `system.New(...)` where you can configure the system and provide it with your application module. For the node initialization, you can get all modules of the system by calling `system.Modules()`.

Chapter 6

Example: Chat App

An example of how to use the longest-chain consensus system is the [Blockchain Chat App](../../samples/blockchain-chat/). Users can enter input through standard input line by line and the system replicates all messages in the same order across all nodes. It enforces that all messages sent from the same sender appear in the history in a monotonically increasing order of submission time.

6.1 Specification of the Chat App

The following lists the key specifications of the chat app:

Payload
The payloads consist of a message, the sender id (id of the node from which the message was sent), and a submission ("sent") timestamp.

State
The state consists of the message history and a collection of "last sent" timestamps, one for each sender, where the timestamp corresponds to the submission time of the last message of this sender.

Applying blocks to compute state
When applying a block to a state, the message in the payload is appended to the message history and the "last sent" timestamp corresponding to the sender is updated.

Verifying blocks
To verify a block, the application verifies that the "sent" timestamp of the payload is not before the "last sent" timestamp of the sender stored in the state associated with the block's parent block.

Providing payloads
Each node's application keeps track of all messages that were submitted to it. Additionally, if a fork happens and the branch changes, it re-adds payloads that

were previously part of the old canonical chain to its payload pool and removes the ones that are now part of the canonical chain. If no payloads are available, it simply provides an empty payload.

6.2 Running the Chat App

To run the chat app, you can run the following in multiple terminals from the example's directory, once for each node

```
go run . -numberOfNodes <number of nodes in the network>
-nodeID <id of this node [0, numberOfNodes-1]>
```

Further, you can add the following options to modify the characteristics of the system(s). If they are not set, default values will be used

- `-disableMangle`: Disables all mangling of the messages between the nodes.
- `-dropRate`: The rate at which to drop messages between nodes. (Ignored if `disableMangle` is set.)
- `-minDelay`: The minimum delay by which to delay messages between nodes. (Ignored if `disableMangle` is set.)
- `-maxDelay`: The maximum delay by which to delay messages between nodes. (Ignored if `disableMangle` is set.)
- `-expMiningFactor`: Factor for exponential distribution for random mining duration.

If `tmux` is installed, you can run `./run.sh` to start a network of 4 nodes with reasonable options set.

6.3 Visualization

For this chat application, there exists an accompanying browser-based visualization tool that utilizes the aforementioned websocket interceptor[1]. The visualizer provides insight into the state of the different nodes. In particular, it displays the block tree stored in every node, the current and the current chat/message history (i.e., state corresponding to the current head).

The following information is shown once for each node. In the box at the top, you see the chat/message history. Right below it, you can see the id of

[1] https://github.com/komplexon3/longest-chain-project/tree/main/chain-visualizer

the current head (truncated for readability) and the message part of the head's payload. The biggest part of the window is filled with the block tree that is stored by the node's BCM. It shows how the blocks are connected and the current head is marked with a red border. To compare the trees more easily, each block's background color is derived from its id.

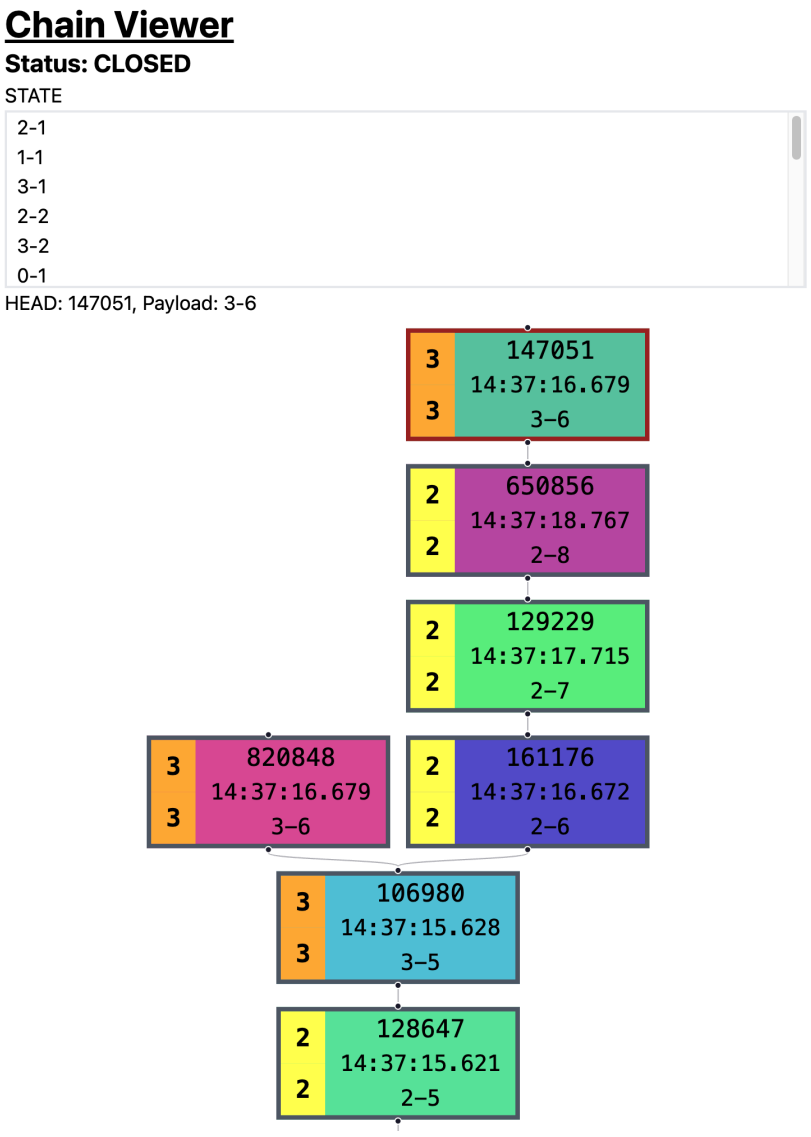

Figure 6.1: Visualizer

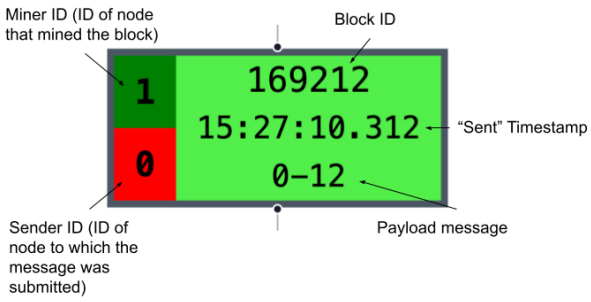

Figure 6.2: Block in Visualizer

Every single block displays the following information:

- **Block ID:** The id of the block (truncated for readability).
- **Miner ID:** The id of the node that mined the block. The background color is unique per id.
- **Sent Timestamp:** The time at which the message was sent. This information is part of the payload. Not to be confused with the time at which the block was mined.
- **Payload Message:** The message that is part of the payload.
- **Sender ID:** The id of the node from which the message was sent. Again, the background is unique per id. Note that this information is part of the payload.

Chapter 7

Conclusion

In this report, we demonstrated how a longest-chain consensus protocol can be abstracted into a set of different modules. We also show how easy it is to implement such a complex system in Mir. The implemented system is easily extensible and could serve as a platform to simulate different network conditions or attack scenarios (e.g., "Selfish Mining [3]").

In addition to the implementation of the longest-chain consensus protocol, we also demonstrate a simple way to create web-based visualization tools for the Mir framework.

Bibliography

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," https://bitcoin.org/bitcoin.pdf.

[2] "Mir - the distributed protocol implementation framework," https://github.com/consensus-shipyard/mir.

[3] I. Eyal and E. G. Sirer, "Majority is not enough: Bitcoin mining is vulnerable," 2013.