Implementation of the Streamlet Consensus Algorithm in Blockchain

Luís Viana
`fc62516`

Guilherme Santos
`fc62533`

November 19, 2023

1 Introduction

The project consists in the implementation of the Streamlet consensus algorithm. The programming language used for this project was C#, supported with the use of Protocol buffers (Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data).

The configuration of the project followed a 5 node architecture, being easily upscaled by the addition of new lines of `ip:port` to the `ips.txt` file.

For the selection of the epoch's leader we instantiated a random with the same seed in all nodes and generated a new value every epoch for an arbitrary but certain decision shared by all nodes.

For the implementation of the data structures defined in the `proto_file.proto` file, we used the following composition, as mainly suggested in the project description, with the exception of the simplification of the Transactions representation.

- Block: Hash (bytes), Epoch (integer), Length (integer), Transactions (string).
- Message: Type (Propose, Vote, Echo), Content (Message or Block), Sender (integer)

The connection between nodes is done via instantiation of a TCPClient for each node, including itself (for the implementation of the URB-Broadcast). The exchange of messages between nodes is done via NetworkStreams.

Our implementation follows the requested basic protocol without fault tolerance or desynchronization of nodes. The code output is represented in the image below.

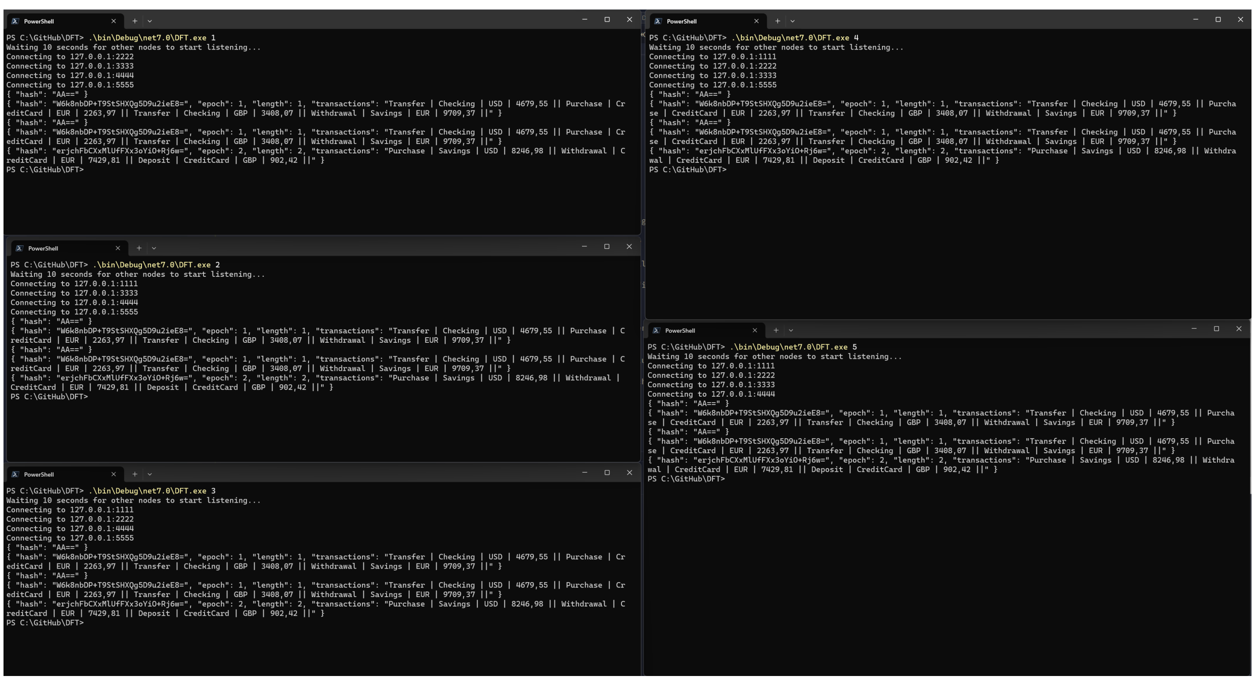

Figure 1: 5 nodes running the Streamlet consensus algorithm.

2 Implementation

2.1 Uniform Reliable Broadcast Protocol:

For the broadcast used in the propose and vote actions we implemented the Uniform Reliable Broadcast Protocol, where the node only sends the message to itself, and upon its reception, it sends to every other node excluding the sender of the message, via a call to the Echo function, the code of which is provided below.

```
2 references
public static void URB_Broadcast(Message message) {
    // Serialize the message to bytes
    byte[] serializedMessage = SerializeMessage(message);
    // Send the serialized message to myself
    MyselfStream?.Write(serializedMessage, 0, serializedMessage.Length);
}

2 references
public static void Echo(Message message) {
    Message echoMessage = new() {
        MessageType = Type.Echo,
        Content = new Content {
            Message = message
        },
        Sender = IDNode
    };
    // Serialize the message to bytes
    byte[] serializedMessage = SerializeMessage(echoMessage);
    foreach (TcpClient otherClient in OtherAddresses) {
        NetworkStream otherStream = otherClient.GetStream();
        // Send only the relevant bytes received, not the entire buffer
        otherStream.Write(serializedMessage, 0, serializedMessage.Length);
    }
}
```

Figure 2: URB-Broadcast and Echo functions.

A Uniform Reliable Broadcast Protocol

operation URB_broadcast (m) **is**
(1) send MSG(m) to p_i.

when MSG (m) **is received from** p_k **do**
(2) **if** (first reception of m) **then**
(3) **for each** $j \in \{1, \ldots, n\} \setminus \{i, k\}$ **do** send MSG (m) to p_j **end for**;
(4) URB_deliver (m) % deliver m to the upper application layer %
(5) **end if**.

Figure 3: URB-Broadcast algorithm.

2.2 Clearing List at Start of Epoch:

To improve performance, we use an approach of clearing the list at the start of each period. This optimization assures that we start from scratch, avoiding redundant data processing from earlier epochs and optimizing memory use.

2.3 Saving of Echo Contents:

We save the contents of the echo to avoid receiving duplicate echo content across epochs.

2.4 Restricting Voting to Nodes Receiving Echo:

To improve the voting process, only nodes that receive the echo of the proposed block are allowed to vote. This decision is especially important for avoiding the leader vote because it already has the real proposed block via URB broadcast, therefore it will not receive the echo of the proposed as stated in 2.3.

2.5 Implementation of `CheckFinalizationCriteria()`:

This function is designed to identify the last finalized block, serving as a reference point for subsequent additions to the blockchain. Blocks that do not meet or exceed the length of the last finalized block are deemed invalid.