LINFO2345 Project report

Maxime Wets, 6331-16-00

December 2023

PART 1

To test the project, just the following commands:

```
bash generate_transactions.sh 400
make -C part1
md5sum part1/log/*.log | cut -d' ' -f1 | sort | uniq -c
```

It will generate 400 transactions, compile all the erlang source files and start the network with 199 normal nodes and 1 builder. The last command will compute the MD5 hash of all the output files for each node to verify their integrity.

Design Choices

Transactions

Transactions can be generated by the `generate_transactions.sh` bash script. It takes the form of a CSV file, where each line represents a different transaction as the string: `sender,receiver,amount`. At the beginning of the algorithm, the builder will read the CSV file before starting to compute blocks.

Blocks

Blocks are represented by the `(tree, {hash, left, right, tx})` record. - Leaf nodes store the original transaction (i.e. the line from the CSV) in the `tree.tx` record, and have no `tree.left` or `tree.right` records; the `tree.hash` value is equal to the MD5 hash of the `tree.tx` value. - Tree nodes have an empty `tree.tx` record, and their `tree.hash` value is equal to the MD5 hash of the concatenation of its `tree.right` and `tree.left` child nodes.

It was chosen to use MD5 as the hashing function for this project, because it is easy to calculate and is only 16 bytes long, making it easier to read when debugging the program. The MD5 function is susceptible to collision attacks, so an attacker could try to forge collisions on purpose in order to push invalid blocks. However, since this project is only a proof-of-concept implementation of a Proof-of-Stake blockchain, I did not take this attack vector into consideration.

In a real-world scenario, the hashing function should be changed to a more collision-resistant function such as SHA256. It would also be a good idea to include more hash values in the record, to make collisions even less likely: For instance, if each node of the Merkle Tree includes the MD5, SHA256 and SHA512 hashes, an attacker would have to find a collision that works for all the hashing algorithms, making it very unlikely.

Directory structure

```
$ tree part1/
    part1
    |-- network.erl
    |-- builder_node.erl
    |-- normal_node.erl
    |-- merkle_tree.erl
    `-- utils.erl
```

Merkle Tree

The merkle tree is implemented in `./part1/merkle_tree.erl`. It exports 2 functions:

- `init/1`: requires a list of transactions as input, and will calculate the merkle tree;
- `root/1`: requires a merkle tree as input, andwill return the hash of the top of the merkle tree.

To compute the Merkle Tree of a list of transactions, the algorithm first calculates the MD5 hash of each transaction, then, transforms each element of that list into a tree record definded as:

```
-record(tree, {hash, left, right, tx})
```

After the first layer has been computed (i.e. all transactions are transformed into leaf nodes), the algorithm will build the tree layer by layer, until one node is left (this will be the root node).

The hash represents the MD5 hash of the transaction (if the node is a leaf) or the hash of its left and right children (if the node is a tree).

Node implementation

The nodes are implemented in two separate modules: `normal_node` and `builder_node`. For this first task, I renamed `Validator` and `Nonvalidator` nodes to `Normal`. Every node will start as a normal node in this waiting loop, waiting to receive an `{init}` message from the network.

```
waiting_loop() ->
    receive
        {init, Type, PID_List} ->
            if
                Type == 1 -> builder_loop(PID_List);
                Type == 0 -> normal_loop(PID_List)
        end
    end.
```

Once a node receives an `init` message, it will run the corresponding init function (`builder_node:init` or `normal_node:init`). This function will then be responsible to prepare the function parameters and then call the corresponding loop (`builder_loop` or `normal_loop`).

Builder node The builder node will first read the remaining transactions from a file and parse it to a list. After that, it will create blocks of size `10 % (remaining tasks)`. This ensures that no block is ever going to exceed the maximum size, and that the last block can be smaller if necessary. The builder node will then broadcast the blocks to the normal nodes until the last block was computed. After that, the builder broadcasts a `{stop}`, indicating that the nodes should print their blocks.

Normal nodes The `normal_init` function simply prints a message indicating that the normal node has started, and then runs a loop waiting for incoming messages: - `{block, NewBlock}`: this indicates that a new block has been pushed, the node will append the block to its internal list and recursively run the loop; - `{stop}`: this indicates that the builder has finished, the node will then create a CSV file and write its version of the blockchain to that file.

PART 2

To test the project, just the following commands:

```
bash generate_transactions.sh 400
make -C part2
md5sum part2/log/*.log | cut -d' ' -f1 | sort | uniq -c
```

It will generate 400 transactions, compile all the erlang source files and start the network with 30 validator nodes, 169 normal nodes and 1 builder. The last command will compute the MD5 hash of all the output files for each node to verify their integrity.

Design choices

Directory structure

```
$ tree part2/
    part2
    |-- network.erl
    |-- normal_node.erl
    |-- builder_node.erl
    |-- validator_node.erl
    |-- proposer_node.erl
    |-- merkle_tree.erl
    `-- utils.erl
```

Node implementation

The logic for starting and initializing nodes is the same as for the first part. The difference is that the network will now choose arbitrarily which nodes are validators, and which of them are in the proposer group for the first epoch.

Each node keeps the following internal record to keep track of the group status of all nodes. This allows every node to authenticate the sender of a message upon receipt: when a node receives a message, it will first verify in its internal record that the sender is correct before accepting it.

```
-record(nodes, {builder, proposers, validators, normal})
```

Normal nodes No changes were made to the normal nodes: the nodes are waiting for `{block}` messages from the builder and simply add them to their internal blockchain. The election process is completely transparent to the normal nodes.

The normal nodes expect only two messages:

- `{block, NewBlock, From}`: indicates that a new block was forged by the builder, the recepient verifies that the sender is the builder and then adds it to its blockchain;
- `{stop, From}`: indicates that the builder finished computing all the blocks. After veryfing that the sender is the builder, the node will print the blocks.

Builder node Since the builder node should only broadcast block during epoch (and not in the election process), the builder waits for a `{continue}` message from the main validator node before sending the next block. When the builder receives an `{epoch_end}` message, it will stop sending more blocks and wait for the main proposer to announce the next proposer group.

The builder has two loops:

1. **Main loop**: in this loop, the builder waits for permission from the main proposer to send a new block, if the builder gets an `{epoch_end}` message, it will go to the epoch change loop;
2. **Epoch change loop**: the builder waits for the election process to finish before going back to the main loop.

The builder expects the following messages:

- `{continue, From}`: indicates that the main proposer allows further blocks to be broadcast. Upon receipt, the builder will calculate the next block and broadcast it;
- `{epoch_end, From}`: indicates that 10 blocks were sent by the builder. Upon receipt, the builder will wait in a separate loop, waiting for a `{proposers}` message before continuing;
- `{proposers, From}`: indicates that a new proposer group has been elected. Upon receipt, the builder will go back to the main loop and wait for a `{continue}` message.

Validator nodes The validator nodes behave like normal nodes, except that they are "election-aware": the nodes have two loops: one for when the builder is broadcasting blocks, and one for when the epoch is changing.

Validator nodes expect the following messages:

- `{block, NewBlock, From}`: same as for normal nodes;
- `{epoch_end, From}`: indicates that the node should go in the election loop, waiting for a `{validators}` or `{proposers}` message;
- `{validators, List, From}`: a shuffled list of nodes has been received, the recipient will re-shuffle it, send it to the next node and stay in the election loop;
- `{proposers, List, From}`: the next proposers group has been elected by the previous main proposer. After verifying the sender, the node updates its internal nodes record and go to the corresponding loop.

Proposer nodes Proposer nodes have the same implementation as validator nodes, except for the main proposer.

Main proposer node The main proposer (= the first node in the proposer group), is responsible for counting the blocks in each epoch and send `{continue}` messages to the builder when the builder is allowed to broadcast a new block, or broadcast `{epoch_end}` when a new election should start.

The main proposer node expects the following messages in its main loop (=until the epoch counter reaches 10). When entering the loop, the main proposer will check if the epoch count is lower then 10, if yes, it will send `{continue}` to the builder, else it will broadcast `{epoch_end}` to the builder and to all validators, indicating that a new election process is starting.

The main proposer expects the following messages during its main loop:

- `{block, Block, From}`: indicates that the builder has forged a new block, the main proposer will increment the epoch counter and restart the loop;
- `{stop, From}`: indicates that the builder finished processing all the blocks. The node will then print its blockchain to CSV.

When in the election loop, the main proposer only expects one message:

- `{validators, List, From}`: this indicates that the election process is ending, the main proposer will then take the first 10% nodes of the list and broadcast the new list of proposers to all validators and the builder.

Election steps logging

The instructions for this project asked for all the validators to write a log file with the election steps. However, I saw this requirement after finishing to write the code for the second part, so I did not implement it; instead, all the validator nodes (and the builder) are logging their current status of the election process to stdout in the following format:

```
[ID] (type) <Message>
```

where ID is the process ID (unique ID) of the node, type is the current type of the node (`proposer`, `validator`, `main_proposer` or `builder`).

This gives a better overview of what is happening in the backend during the election process.

Analysis of the election process

During the election process, the first node of the proposer group will shuffle the list of all validators and send it to the first node of this shuffled list. Then, each node that receives a shuffled list will shuffle it again and forward it to the next node, until the list finally reaches the original sender. Since each list is shuffled randomly, the more nodes we add in the validator group, the more time it will take for the election to end, because the list is shuffled at every step. This can become a limitation of this leader election system when the number of validators grows, as there is no time bound for the election process (the list has to pass to a minimum of two nodes but there is no guarantee on the time complexity).

PART 3

Unfortunately, I did not have enough time to finish the third part of this project.