DMBLOCK Assignment 1

Lukáš Častven
Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií
xcastven@stuba.sk

17 march 2024

Contents

1 Programming language choice

For assignment we received a set of Java source codes, which are to be used to implement given tasks. Since there is a 5 point bonus for rewritting these source codes and implementing this assignment in different language, I choose to do it in Rust [1], because I wanted to better myself in it.

I rewrote provided source codes for each of the three phases into Rust. My source codes roughly match the provided ones.

Each phase is in it's separate directory with test as listed in the assignment.

1.1 Environment details

This assignment was developed and tested in this environment:

- OS: Arch Linux, Kernel 6.6.21-1-lts,
- Rust compiler: rustc 1.76.0 (07dca489a 2024-02-04),
- Cargo: cargo 1.76.0 (c84b36747 2024-01-18),
- Libraries: each version is listed in corresponging Cargo.toml file.

2 Phase 1 - Simple Coin

2.1 Assignment

Centralized authority FIIT accepts transactions from users. Implement logic for processing transactions into a ledger. FIIT groups transactions into into a pseudo blockchain. In each block, FIIT will receive list of transactions which are validated, applied and subset of valid ones is returned.

These transactions can depend on each other, there may be double-spends, and otherwise invalid transactions.

In this phase, implement a UTXO FIIT chain, which in each epoch takes a list of proposed transactions, validates them, applies valid ones to its internal state and returns a subset of valid ones. The size of the subset isn't defined.

Also implement a version in which, not only validity is checked, but also transactions are processed in order to maximize received fees.

2.2 Implementation

Implementation for this phase is in submodule *fiitcoin*. This submodule is a Rust library crate, so I could easily reuse it in next phases with importing it as a dependency.

Required handlers are in rust file *fiitcoin/src/handler.rs*. Both implement common trait `TxHandler`:

Listing 1: TxHandler

```
pub trait TxHandler<'a> {
    /// Each epoch accepts unordered vector of proposed transactions.
    /// Checks validity of each, internally updates the UTXO pool, and
    /// returns vector of valid ones.
    ///
    /// # Beware
    /// Transactions can be dependent on other ones. Also, multiple
    /// transactions can reference same output.
    fn handle(&mut self, possible_txs: Vec<&'a Tx>) -> Vec<&'a Tx>;

    /// Returns reference to internal pool
    fn pool(&self) -> &UTXOPool;

    /// Returns mutable reference to internal pool
    fn pool_mut(&mut self) -> &mut UTXOPool;

    /// Moves internal pool, while consuming self
    fn move_pool(self) -> UTXOPool;

    /// Checks if:
    ///     1. All UTXO inputs are in pool
    ///     2. Signatures on inputs are valid
    ///     3. No UTXO is used more than once
    ///     4. Sum of outputs is not negative
    ///     5. Sum of inputs >= Sum of outputs
    fn is_tx_valid(&self, tx: &Tx) -> bool;

    /// Filters independent txs from dependent ones,
    /// applies them and returns both sets
    fn handle_independent(
        &mut self,
        txs: Vec<&'a Tx>
    ) -> (Vec<&'a Tx>, Vec<&'a Tx>);

    /// Applies given tx to the internal pool
    fn apply_tx(&mut self, tx: &Tx);


    fn is_input_in_pool(&self, input: &Input) -> bool;
}
```

Methods `is_tx_valid`, `handle_independent` `apply_tx`, `is_input_in_pool` have default implementations in the trait declaration, because all of them are used in both handlers. Methods `handle`, `pool`, `pool_mut`, and `move_pool` are implemented in each handler.

The `is_tx_valid` function is self explanatory, and all checks, which are performed are listed in its doc comment.

The `handle_independent` function is used to separate transactions, which are not dependent on transactions in the currently proposed list and can be applied right away, from invalid and dependent transactions. The valid independent ones are also applied to the internal state.

Listing 2: handle_independent

```rust
/// Filters independent txs from dependent ones,
/// applies them and returns both sets
fn handle_independent(
    &mut self,
    txs: Vec<&'a Tx>
) -> (Vec<&'a Tx>, Vec<&'a Tx>) {
    let mut handled = vec![];
    let mut dependent = vec![];
    let tx_set: HashSet<[u8; 32]> = txs.iter().map(|&tx| tx.hash()).collect();

    for &tx in txs.iter() {
        if tx.inputs().iter().all(|i| self.is_input_in_pool(i)) {
            // tx is only dependent on outputs in pool
            if self.is_tx_valid(tx) {
                self.apply_tx(tx);
                handled.push(tx);
            }
        } else if tx
            .inputs()
            .iter()
            .any(|i| tx_set.contains(&i.output_tx_hash()))
        {
            // tx is dependent on some outputs from this batch
            dependent.push(tx)
        }
    }

    (handled, dependent)
}
```

The `apply_tx` function takes a valid transaction and mutates internal state according to the transaction.

Listing 3: apply_tx

```
/// Applies given tx to the internal pool
fn apply_tx(&mut self, tx: &Tx) {
    for input in tx.inputs().iter() {
        self.pool_mut().remove_utxo(&input_to_utxo(input));
    }
    for (i, output) in tx.outputs().iter().enumerate() {
        let utxo = UTXO::new(tx.hash(), i.try_into().unwrap());
        self.pool_mut().add_utxo(utxo, &output)
    }
}
```

Handler

Handler implements the FIITcoin chain logic of validating, applying and returning valid subset of transactions. It's functionality is very simple, loop until there are no dependent transactions, which are retrieved with `handle_independent` function. At the end returns the subset of valid, applied transactions.

Listing 4: Handler::handle

```
fn handle(&mut self, possible_txs: Vec<&'a Tx>) -> Vec<&'a Tx> {
    let mut handled: Vec<&'a Tx> = vec![];
    let mut to_handle = possible_txs;

    loop {
        let (independent, dependent) = self.handle_independent(to_handle);
        handled.extend(independent);
        if dependent.is_empty() {
            break;
        }
        to_handle = dependent;
    }

    handled
}
```

MaxFeeHandler

This implementation of a handler processes transactions in order to maximize collected fees. Since there is no maximum count of processed transactions (if there was the problem would be NP hard and very similiar to Knapsack Problem [2]), I choose very simple heuristic how to achieve this... Process all valid transactions.

Firstly, for each transaction a fee is calculated. The fee is a difference between sum of input values and sum of output values.

$$Fee = \sum_{n=1}^{|inputs|} inputs_n - \sum_{n=1}^{|outputs|} outputs_n$$

Then the initial list of proposed transactions is sorted by their fee, from highest to lowest. And then are processed just like in 2.2.

Listing 5: MaxFeeHandler::handle

```
fn handle(&mut self, possible_txs: Vec<&'a Tx>) -> Vec<&'a Tx> {
    let tx_map: HashMap<[u8; 32], &'a Tx> =
        possible_txs.iter().map(|&tx| (tx.hash(), tx)).collect();

    let mut with_fees: Vec<(u64, &Tx)> = possible_txs
        .iter()
        .filter_map(|&tx| match self.calc_fee(tx, &tx_map) {
            Some(fee) => Some((fee, tx)),
            None => None,
        })
        .collect();
    with_fees.sort_unstable_by(|tx1, tx2| tx1.0.cmp(&tx2.0));
    with_fees.reverse();

    let mut handled: Vec<&'a Tx> = vec![];
    let mut to_handle = with_fees.iter().map(|tx| tx.1).collect();

    loop {
        let (independent, dependent) = self.handle_independent(to_handle);
        handled.extend(independent);
        if dependent.is_empty() {
            break;
        }
        to_handle = dependent;
    }

    handled
}
```

2.3 Tests

Tests for this phase are in *fiitcoin/tests*, and are implemented according to provided test names in assignment.

Test 1 through 7 are in *fiitcoin/tests/is_tx_valid_test.rs*. However test case number 7 isn't implemented. This is because in the provided source codes, Java's `double` is used, which is a signed representation of a fractional number [3]. I used Rust's `u32` [4] to denominate value of an output, thus testing if a sum of all outputs is negative is meaningless. Even if I created a raw bytes representation of a transaction, and instead of unsigned integer encoded a signed one, it would be treated as a large unsigned integer, in which case the check for sum of inputs being less then or equal to sum of outputs would fail, thus creating the same result.

Test 8 through 15 are in *fiitcoin/tests/handler_test.rs*. And max fee tests 1 through 3 are in *fiitcoin/tests/max_fee_handler_test.rs*.

To run these tests, navigate to *fiitcoin* and run this command:

```
$ RUST_LOG=debug cargo test --release
```

3 Phase 2 - Trust and Consensus

3.1 Assignment

Implement algorithm for distributed consensus based on relationalship graph. Network is an oriented graph, in which each edge represents a trust relationalship between two nodes. If there is an edge $A \to B$ it means that node B is a follower of node A (A is a followee of B) and listens to transactions proposed by A.

Nodes in network are either trusted or byzantine. Each trusted node should reach consensus with other peers in the network. Implement trusted node, which defines how each trusted node in network behaves. Then test this network in a simulation with different relationalship graphs and with different parameters. At the end of the simulation, each node should return the same subset of transactions, upon which consensus was reached. Assume all transactions are valid. Different simulation parameters:

- Probability of an edge existing = $p\_graph \in \{0.1, 0.2, 0.3\}$,
- Probability of an node being byzantine = $p\_byzantine \in \{0.15, 0.3, 0.45\}$,
- Probability of a transaction distribution to node = $p\_tx\_dist \in \{0.01, 0.05, 0.1\}$,
- Number of rounds in simulation = $rounds \in \{10, 20\}$.

Also implement a byzantine node. This node acts like an adversary in the network, and is trying to distrupt the network. There can up to 45% of byzantine nodes in the network. Their behaviour is:

- Dead - don't resend any transactions,
- Selfish - only resend their transactions,
- Mix - switch between previous two behaviours.

At the end of the simulation, all trusted nodes must return the same subset of transactions. The size of this subset should be maximal, and time to reach consensus should be reasonable.

3.2 Implementation

Implementation for this phase is in submodule *consensus*. This submodule is also a Rust library crate, because I didn't know if I will reuse it later or not.

Implementations of trusted and byzantine node are in *consensus/src/node.rs*, and both of them share common trait `Node`:

Listing 6: Node trait

```
pub trait Node<const N: usize> {
    /// If ‘ith‘ entry is ‘true‘ then this Node follows the ‘ith‘ Node
    fn followees_set(&mut self, followees: [bool; N]);

    /// Initializes proposed set of txs
    fn pending_txs_set(&mut self, pending_txs: HashSet<Tx>);

    /// Returns proposed txs, which shall be send to this Node’s followers.
    /// After final round, the behaviour changes and it will return txs,
    /// on which consensus was reached.
    fn followers_send(&self) -> &HashSet<Tx>;

    /// Candites from different Nodes
    fn followees_receive(&mut self, candidates: &Vec<Candidate>);

    fn is_byzantine(&self) -> bool;
}
```

TrustedNode

Trusted node starts with transmitting its transactions, and after receiving a set of proposed transactions from followees it starts to track how many distinct peers proposed each transaction, and adds each proposed transaction to the ones it will transmit in next round.

A trusted node will assume a consensus was reached upon a transaction, if certain number of peers propose it back to it. The number is called `consensus_threshold` in code and is calculated with this formula:

$$CT = min(1, Probablefollowers - Probablebyzantinenodes)$$

Each node has access to simulation's parameters so it can compute how many followers it should have and how many byzantine nodes there are in network.

At the end of a simulation, each node returns the set of transactions upon which consensus was reached on.

ByzantineNode

Byzantine node is not interesting. It either doesn't resend any transactions, only resends its transactions and no other, or switches between these two behaviours.

3.3 Tests

There is only one test, and that is in *consensus/tests/simulation.rs*, in which for all permutations of the simulation parameters are parallelly tested. Each simulation has a result, and this result is written to */tmp/sim-result.txt* file (in *consensus*, there is a file from this test that I ran).

Since randomness is used to initialize each simulation, if it fails it is rerun, but not more than 3 times, in order to make sure that it failed due to a bug, not due to an edge case.

Each line in the *consensus/sim-result.txt* contains what parameters were used, how long the initialization of the simulation took, what seeds were used for randomness, how long the simulation took and what was the size of the transaction subset upon which consensus was reached on. If the simulation fails more than 3 times, there would an additional entry in the line, which mentions this, but I didn't encounter any.

Test cases in assignment are covered by this one simulation which uses all possible permutations of parameters.

To run these tests, navigate to *consensus* and run this command:

```
RUST_LOG=info cargo test --release
```

4 Phase 3 - Blockchain

4.1 Assignment

In this phase, create a node which is part of a distributed consensus protocol based on UTXO blockchain. Implemented code will receive incoming transactions and blocks, and maintain and update them. Reuse code from Phase 1(2).

Blockchain is responsible for maintaining a chain of blocks. Because whole chain would be big, only keep few newest blocks.

In this chain, forks can occur, so it is needed to maintain corresponding UTXO pool.

New genesis block will not be mined, ignore any new block which is trying to be a genesis block. When a fork occurs, at the max height of the chain is the newest block. If a reorg happens after a fork, don't put transactions from the shorter branch into the mempool. Also fees are neglected.

Bonus for implementing a multisig transactions.

4.2 Implementation

Implementation for this phase is in submodule. The main code is in *blockchain/src/blockchain.rs*. In there is a struct called `Blockchain` which contains a ring buffer holding the most recent blocks, and a mempool for unprocessed transactions. For manipulation with the `Blockchain` struct, `BlockHandler` from *blockchain/src/handler.rs* should be used.

`BlockHandler` has two methods for creating blocks:

Listing 7: BlockHandler

```
impl BlockHandler {
    pub fn create_block(&self, address: &VerifyingKey<Sha256>) -> Block {
        let parent = self.chain.block_at_max_height();
        let mut new_b = IncompleteBlock::new(parent.hash(), address);

        let utxo_pool = self.chain.utxo_pool_at_max_height();
        let mut handler = Handler::new(utxo_pool.clone());

        let tx_pool = self.chain.tx_pool_at_max_height();
        let txs = tx_pool.txs();
        let handled = handler.handle(txs);

        for &tx in handled.iter() {
            new_b.add_tx(tx.clone());
        }
        new_b.finalize()
    }

    pub fn create_fork(
        &self,
        parent_hash: [u8; 32],
        address: &VerifyingKey<Sha256>,
    ) -> Option<Block> {
        let (parent, utxo_pool) = self.chain.at_block_hash(parent_hash)?;
        let mut new_b = IncompleteBlock::new(parent.hash(), address);
        let mut handler = Handler::new(utxo_pool.clone());

        let tx_pool = self.chain.tx_pool_at_max_height();
        let txs = tx_pool.txs();
        let handled = handler.handle(txs);

        for &tx in handled.iter() {
            new_b.add_tx(tx.clone());
        }
        Some(new_b.finalize())
    }
}
```

`create_block` is used when a new block is appended after linearly to the chain, and `create_fork` is used when a fork in chain is created. Each of these two functions reuses the handler from Phase 1(2), and only bundles valid transactions into new block.

Then after creating a block, it can be added with `BlockHandler::process_block`, which calls method `Blockchain::add_block`:

Listing 8: Blockchain::add_block

```
impl Blockchain {
    pub fn add_block(&mut self, block: Block) -> bool {
        let node = match self.at_block_hash(block.prev()) {
            Some(parent) => parent,
            None => return false,
        };

        let mut handler = fiitcoin::handler::Handler::new(node.1.clone());
        let txs: Vec<&fiitcoin::tx::Tx> = block.txs().iter().map(|tx| tx).collect();

        if handler.handle(txs).len() != block.txs().len() {
            log::warn!("Block contained invalid txs!");
            return false;
        };

        for tx in block.txs().iter() {
            self.mempool.remove(tx.hash());
        }
        self.chain.push((block, handler.move_pool()));

        true
    }
}
```

This method validates that the parent of the proposed block exists in the buffer, if yes, once again validates all transactions, updates state of the mempool, and appends the block to the chain.

4.3 Tests

Tests for this phase are in *blockchain/tests/tests.rs*. Some tests cover more than one test case from assignment, they have a comment above them mentioning which tests they cover. To run them navigate to *blockchain* and run this command:

```
RUST_LOG=debug cargo test --release
```

4.4 Multisig bonus

The multisig implementation is in *multisig*. I copied all needed files from previous phases there, because I didn't want to break any existing tests.

I changed transaction outputs from having only one verifier, to holding a list of verifiers and having a threshold indicating how many valid signatures are needed to unlock given output. Also each transaction input has a list of signatures which are used to unlock corresponding output.

Tests are in *multisig/tests*, in *multisig/tests/multisig_tx.rs* are tests just testing individual transactions and in *multisig/tests/multisig_blockchain.rs* are tests for using these multisig transactions in blockchain.

5 Conclusion

Thanks to the phase 1, I now understand how Bitcoin's UTXO model works. The 2nd phase was more challenging on my brain, but when I finally found a formula that works I was happy. The 3rd phase was my favourite. I somewhat understood what and how to implement it yet it was challenging enough.

The idea to rewrite the assignment to Rust was not the best in few situations, hence the code isn't the best Rust code. There are `clone()`s that I would like to get rid off, but couldn't figure out how, or now that I have rewrote it, I would do few things differently. However I really liked working with Rust.

I'm happy how this assignment turned out. I had few rough times, when nothing was working as I wanted, but I managed to implement it pretty well I would say so.

References

[1] "Rust Programming Language — rust-lang.org." https://www.rust-lang.org/. [Accessed 17-03-2024].

[2] Wikipedia, "Knapsack problem — Wikipedia, the free encyclopedia." http://en.wikipedia.org/w/index.php?title=Knapsack%20problem&oldid=1212701319, 2024. [Online; accessed 17-March-2024].

[3] "DOUBLE PRECISION data type — docs.oracle.com." https://docs.oracle.com/javadb/10.10.1.2/ref/rrefsqljdoubleprecision.html. [Accessed 17-03-2024].

[4] "u32 - Rust — doc.rust-lang.org." https://doc.rust-lang.org/std/primitive.u32.html. [Accessed 17-03-2024].