CS3235 Cource Project: Bitcoin Client in Rust

You are NOT allowed to use code publicly available on the Internet for this assignment. Please make sure that you do not share your solutions, or the background materials for your solution, with other students outside of your group in any private communication. If you wish to share some ideas, inspiration, or questions, please do so on the Canvas -> Discussion for everyone's benefit.

Further clarification (if any) will be in this Clarification Document.

1. Goal

This course project aims to familiarize you with the Rust programming language for writing memory-safe concurrent/parallelized programs.
Each group is expected to independently implement a simplified version of a *Bitcoin client* based on a partially-implemented project template provided **(Part A & Part B**). **Each student** should also independently finish an assignment quiz related to memory safety and Rust (**Part C**). This file mainly describes Part A & Part B.

2. Outline & Important Dates

A team can be of size of 3 or 4. Please fill your team members in this Google spreadsheet by **Mar 15th 6:00 PM**. You can use the Canvas -> Discussion to find teammates. We would assume that you have discussed an equitable work distribution with your teammates (see this section for more). After that, we will randomly group students who don't have a team.

Please download the project template with the name `bitcoin-rust` from Canvas, then set up the environment (Section 3), finish Part A (Section 5) and Part B (Section 6) as described in this document. If you are using GitHub, please make sure your code repository is *private*, because we will check for plagiarism with online repositories and other groups' submissions.

Important dates:

- Confirm your team members by **Mar 15th 6:00 PM**.
- Implement Part A and Part B. Submit your code project as one `tar.gz` file on Canvas by **April 23rd, 11:59 PM**.
- Submit your state dumps `tar.gz` and a demonstration video on Canvas by **April 24th, 11:59 PM**.
- The deadline for Part C will be revealed later.

3. Environment Setup

3.1 Installation

The required environment is a Linux operating system. While not a compulsory requirement, we recommend using `Debian` or `Ubuntu` as the operating system, especially for Part B since Part B requires writing sandboxing policies on Linux. Our test environment is Ubuntu 20.04.

Please install `Rust` and `Cargo` package manager in your OS (rustc 1.60+, cargo 1.60+). Here is the official documentation to install `Rust`.

We recommend `Visual Studio Code` (vscode) for the code editor but you can use any of your choice. After installing `Rust` and `Cargo`, you can install the following two plugins in vscode:

- `rust-analyzer` (by `rust-lang`)
- `Even Better TOML` (by `tamasfe`)

3.2 Testing & Debugging

There are 7 different packages to implement which will be compiled to 3 `bin_*` packages. To build the whole project, you can use the command `cargo build` at the root directory of the project. The compiled binary programs will be in the `target/debug` folder. To run an executable program, you can use one of the following commands:

```
cargo run --bin <binary_name> -- <parameters>
cargo run --bin <binary_name>
./target/debug/<binary_name> <parameters>
```

For example, to run the executable program corresponding to the `bin_wallet` package without parameters, you can use one of the following commands:

```
cargo run --bin bin_wallet
./target/debug/bin_wallet
```

To run unit tests on individual packages (subdirectories under the root directory of the project), there are two ways:

- For a package, you can run either one of the following commands in the package folder:

  ```
  cargo test  # stdout is eaten if passing tests
  cargo test -- --nocapture   # always show stdout
  ```

- For library packages (subdirectories whose name starts with `lib`, e.g., `lib_chain`), there is one additional way to run unit tests in vscode. You can open the `lib.rs` in the package folder. You will see the `Run Test|Debug` buttons after `#[test]` code line in the editor if you installed the `rust-analyzer` plugin. You can click the `Run Test` button to run the following unit test or click the `Debug` button to debug it.

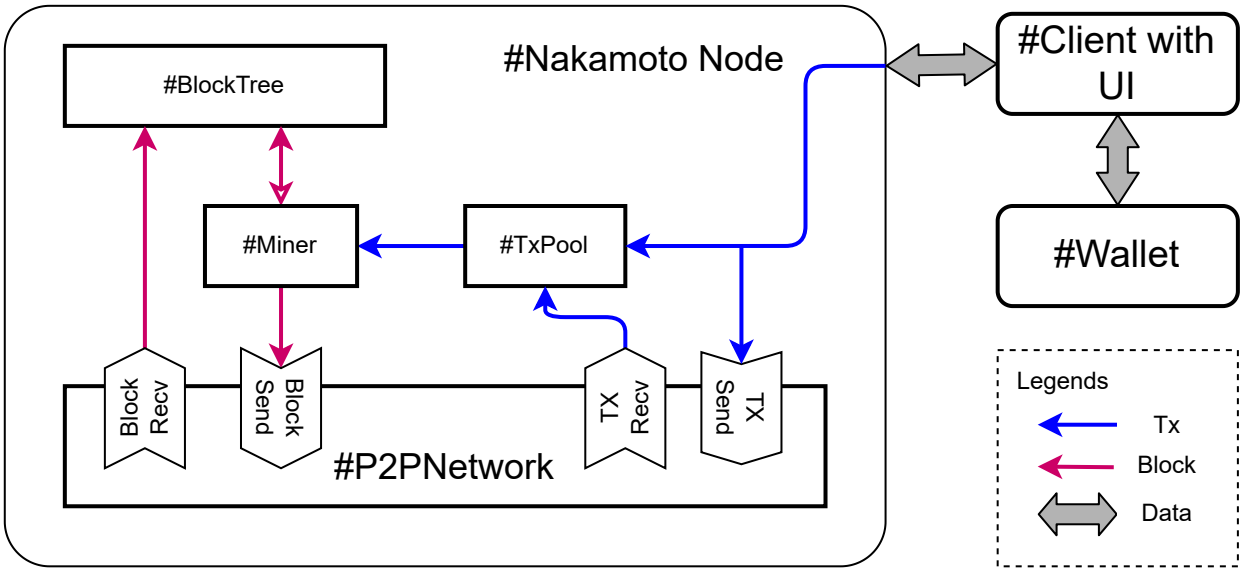
4. Overview of Part A and Part B

In this project, we will implement a simplified, privilege-separated, and sandboxed version of a *Bitcoin Client*, including a node following the Nakamoto Consensus, a wallet, and a client with UI to communicate between them. Please refer to the tutorial on March 6 to learn the basics of Bitcoin/Nakamoto Consensus and the tutorial on March 13 to quickly understand the course project. Tutorial recordings are available on Canvas.

Part A: Implement Bitcoin Client (31pt)

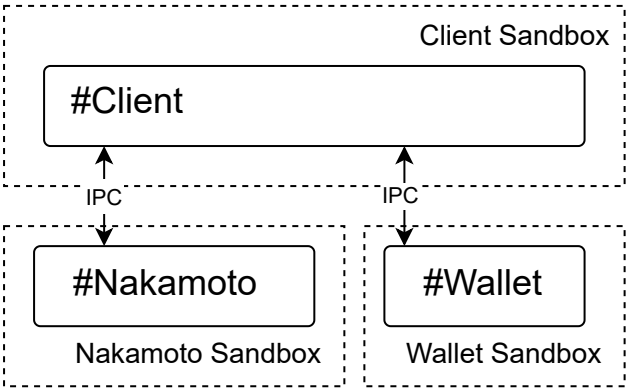
For Part A, please implement the Bitcoin Client: a runnable Nakamoto node (`bin_nakamoto` package), a Wallet (`bin_wallet` package) that has access to user's private key, and a simple client with UI (`bin_client` package). The internal structure of the Bitcoin client is shown in the figure below.

1. `bin_nakamoto` package.
   This package has 4 dependencies (`lib_chain`, `lib_tx_pool`, `lib_network`, and `lib_miner` packages). The BlockTree (`lib_chain` package) stores and updates the whole blockchain, which is actually a block tree. The TxPool(`lib_tx_pool` package) implements a transaction pool, which bookkeeps transactions (TXs for short) on the network that hasn't been finalized. The Miner (`lib_miner` package) creates puzzles based on TXs and the blocktree and tries to solve it to find a new block. The P2PNetwork (`lib_network` package) communicates TXs and blocks with peers on the network. Finally, the `bin_nakamoto` package connects all the above packages and starts a Nakamoto node. All packages should run in the same process with multiple threads. See further instructions in Section 5.
2. `bin_wallet` package.
   This package can create a new pair of public and private keys, sign TXs, verify signatures of TXs, and get the user name and user ID.
3. `bin_client` package.
   This package is a simple client with UI that communicates between the `bin_wallet` package and `bin_nakamoto` package, e.g., create a new TX, sign it using `bin_wallet` package, and send it to `bin_nakamoto` package.
   We have provided you with the UI part of the client.



Part B: Privilege Separation and Sandbox (9pt)

For Part B, please augment the `bin_wallet` and `bin_client` packages with sandboxing to separate their privileges. Each of these packages should be run as a separate process and sandboxed using `seccomp` filters as shown in the figure below. Each process should be given the minimum privileges (i.e., system calls) that they need. See futher instructions in Section 6.



5. Part A: Implement Bitcoin Client

The simplified Nakamoto consensus

We simplify the Bitcoin blockchain but keep the main idea. There are some Nakamoto nodes in the same P2P network. Each node keeps a local view of the blocktree. They can create transactions and broadcast them to other nodes. They would put received transactions into their TX pool, from where some transactions are extracted to create a puzzle with the reference of the last valid block in the longest chain. Once there is a new valid block, Nakamoto nodes would accept it, extend the blocktree, and start building another block. The details are described below.

1. Nakamoto nodes keep a local copy of their view of the blocktree.
2. Nakamoto nodes choose the longest path of the blocktree and build on it, i.e., create a new computational Proof-of-Work (PoW) puzzle based on it and solve it.
    1. We use Bitcoin's PoW puzzle here but with a lower difficulty (`d`). The mining process is to find a nonce that makes the leading-`d` bits of `this_block_hash = SHA256(nonce || puzzle)` are zero. Here we check the number of `0` (string) in the hex string format of `this_block_hash`. For example, let's say one hex string is `000e0m93x73h0f10` whose first 3 bits are `0` then it meets `d=3`. Note that `puzzle` is the serialized string of structure `Puzzle` in `./lib_chain/src/block.rs`.
    2. Bitcoin uses `double-SHA256` to create the Merkle tree but we use `SHA256` here.
3. If there is a newly mined block and it is valid, Nakamoto nodes would accept it and broadcast it. There are two different cases based on how the Nakamoto node found this new block:
    1. If the Nakamoto node solves the puzzle, this node would add it to the local blocktree, broadcast it, create a new puzzle, and solve the new puzzle.
    2. If it receives a new block, it would first check if it is valid, including checking whether the block has enough prefix `0` for `SHA256(nonce || puzzle)`. If not valid, drop it. If valid, it would add it to the blocktree, and check the current longest path. If the longest path and the last block on this path do not change, it continues its mining. If not, it switches to the

new longest path, creates a puzzle, and starts solving it. If two paths have the same length, here we consider the one whose last block has the larger hash number as the longest path.

3. For every successfully mined block, the block miner (`reward_receiver`) would receive a reward of 10 dollars.

4. Nakamoto nodes put the received transactions (TXs) to their TX pool, extract TXs from the pool to create the puzzle, and remove TXs from the pool if they are finalized.
    1. Here we consider the rule to extract TXs from the pool to be FIFO (First-In-First-Out). You don't need to implement the transaction fee and transaction priority.
    2. If one transaction is included in a valid block on the main chain, it becomes confirmed.
    3. For blocks on the longest chain, if they have a depth larger than k, they are finalized. TXs in these blocks are finalized too. Here we consider removing these finalized TXs from the TX pool.
5. Here in our P2P network, the Nakamoto node establish TCP connections with other Nakamoto node and it can receive or send transactions or blocks using these connections.

Note that every time we use "here" or "we consider" it means it's our simplified protocol, not the standard Bitcoin. If you are interested in Bitcoin, please refer to this online free book *Mastering Bitcoin* for a better understanding.

For fine-grained details of each function, please refer to the comments in the template project. You can search for "Please fill in" to identify the best place to place your code.

What is given?

You are given the partially-implemented project template and some unit test samples.

In the project template, the package names, existing data type definitions, and function type signatures are given with documents.
Please implement the missing parts of the code to realize the desired functionalities shown in the previous subsection and the Part A of overview section. However, **please do not change the given package name, existing data type definitions, and function type signatures, which are used to run unit tests.** Your program would fail the evaluation if they are changed. But you can define additional new data structures and methods as you need.

The given unit test samples (code begins with `mod tests {`) are to help you check if your program can run correctly. Initially, these test cases will fail on the given template project. Please do not modify these unit tests. But these unit tests are not comprehensive, so you would need to write additional unit tests to make sure your code is correct.

When you write and test your code, please note the following:

- These folders whose name starts with `nakamoto_config` under the directory `./tests` are pre-defined configuration files for running a demonstration successfully. Please **do not modify** these files. You can add your own configuration files under the directory `./tests` and use them for your own testing.

- In the config file (aka `Config` structure), we define two different difficulties d for puzzles: `difficulty_leading_zero_len` is for this node's block while `difficulty_leading_zero_len_acc` is for other nodes' block. Two difficulty requirements are for easier unit tests. Your final demonstration should set these two difficulties as the same value. You can roughly choose the value such that a new block can be mined in 5-10 seconds.
- Whenever you see any `timestamp` field in any data structure, please use any positive integer for unit tests.

We have imported some libraries that might be sufficient for you in this project template. The whitelist of third-party libraries you can use will be listed in the clarification document. If you'd like to use additional libraries, please comment on this document.

Implementation & Grading Details

Based on the description in the Nakamoto consensus subsection and the Part A of overview section, please implement the *Bitcoin client*: a Nakamoto node (with 4 dependencies), a wallet, and a client that uses our provided UI. In the end, you should be able to run multiple instances of your *Bitcoin client* on the same machine and they should be able to communicate with each other. Each of the instances can create transactions, mine blocks, and sync with other instances.
The *Bitcoin client* you are going to implement (the outcome of Part A & Part B) is something similar to the figure below:

Specially, you need to implement the following 7 packages. Credits will be given based on evaluating if each package passes **our given unit tests and command line tests**. We will also manually check if your implementation satisfies the provided code comments.

- **(3pt)** `lib_chain` is self-contained.
- **(3pt)** `lib_miner` is self-contained. Needs *Parallelization*.
    - Solve the input puzzle on multiple threads as specified by the `thread_count` parameter. Threads can be interrupted anytime.

- **(3pt)** `lib_tx_pool` depends on `lib_chain`.
- **(3pt)** `lib_network` depends on `lib_chain`. Needs *Concurrency*.
    - Handling TCP connections to all neighbors and sending/receiving data.
- **(3pt)** `bin_nakamoto` depends on all above.
    - Set up all the threads and data structures. Feel free to use any number of threads beyond one.
- **(3pt)** `bin_wallet` is self-contained.
- **(3pt)** `bin_client` creates `bin_wallet` and `bin_nakamoto` subprocesses and communicates between them.

(10pt) Please record a video of the process of evaluating each package on tests and the communication between 4 Bitcoin clients. Full points will be given if the recorded demonstration is complete and successful, according to the Clarification Document. Submit the video on Canvas **Project Video Demo Submission** folder.

We suggest you to first implement the `lib_*` packages and then implement the `bin_*` packages.

6. Part B: Privilege Separation and Sandboxing

Implementation

Write sandboxing policies and seccomp filtering code for the three processes: `bin_wallet`, `bin_client` and `bin_nakamoto`. Please make sure that your implementation is privilege separated: from the point that `seccomp` policy is applied, `bin_client` shouldn't deal with networking and `bin_wallet` should not access the file system or the network. Specifically, you need to do the following:

- Using `strace` to trace the process to be executed (`bin_wallet`, `bin_nakamoto` and `bin_client` itself) to collect syscall log and analyze it.
- Write `seccomp-bpf` syscall whitelist filter for those processes.
    - You can add further constraints on arguments of those syscalls to avoid misuse.
- Apply `seccomp` filter after those processes are started. You can apply `seccomp` at the `main` function of `bin_client`, `bin_wallet` and `bin_nakamoto` at the specified location.

Grading Details

- **(3pt)** For each package, 1pt would be given if its seccomp filtering code and its sandboxing policy format are correct.
- **(6pt)** In the same video as Part A, please show the outcome of Part B as well. Speficially, show that the Bitcoin clients can run with your sandboxing policies but cannot run after removing some system calls. We have provided you a script to generate new policy files in which some system calls are removed, based on your policy files. This script at `./random_policy_gen.py` takes the `sha256` hash of your submitted `tar.gz` code on Canvas (`sha256sum ./bitcoin-rust-G01.tar.gz`) as the random seed and generates 6 new policy JSON files (`./bin_client/policies/seccomp_*_DROP*.json`) and a CSV file (`./part_B_results.csv`) for

you to record the running results. Please rerun the program with each of the 6 policy files; all these runs should crash the program. Due to the time limit of the demo, at most 3 minutes will be given for each run of a policy. Please record whether your program crashes in the CSV file during the video demonstration.

7. Final Submission Instructions (For Part A and Part B)

- Submit code: Please finish Part A and Part B in one code project and submit its `bitcoin-client-GXX.tar.gz` to Canvas **Project Code Submission** folder. GXX is your group ID in this Google spreadsheet. Before the submission, please run `./submission_cleanup.sh` beforehand to remove all the state dumps files and the compiled binaries.
- Submit state dumps: After the submission of the code, please record a demonstration video according to the Clarification Document. At the end of the video, you need to submit the state dumps to Canvas **Project State Dumps Submission** folder *in your recording*. Please copy the `./tests/nakamoto_config*` folders into a new folder. If you finished Part B, please also put the CSV file `part_B_results.csv` with your experiment results into the same folder. Then pack this folder into its `Logs-GXX.tar.gz` and submit this tarball. Please do the previous steps in your recording.
- Submit demo: Please submit the recorded demo with name `Video-GXX` to Canvas **Project Video Demo Submission** folder.
- For code, demo, and the state dumps file, 1 submission per group is enough.

Equitable Work Policy and Team Setup

We expect every team member to responsibly own the implementation of at least 1 out of the 7 packages in part A independently. Please split the workload equally and commit to your team members in advance. The team is jointly responsible for the entire submission and will be given the same score for the project.

We assume that all team members have contributed roughly equally. If there is a disagreement among team members about who is expected to finish what, this should be internally resolved, and if not possible, inform the TAs by **Mar 20th**.