Highly Dependable Systems

Sistemas de Elevada Confiabilidade

2024-2025

DepChain

Stage 2

Goals

This project aims to develop a simplified permissioned (closed membership) blockchain system with high dependability guarantees, called Dependable Chain (DepChain). In the second stage, we will build upon the work developed during the first stage and refine the application semantics and the dependability guarantees of DepChain.

The second stage, has the following goals:

- Allow clients (represented by accounts in the system) to perform transactions between them (i.e., native cryptocurrency transfers and trigger smart contract execution).
- Strengthen the Byzantine fault tolerance guarantees of the system, including the ability to tolerate Byzantine clients.

Design requirements

We retain the simplifying assumptions from the first stage that the participants (both clients and servers) are static and known before-hand, together with the associated public key pairs and IPs. Moreover, and even though clients can be Byzantine, they cannot subvert the keys of honest clients.

Students must reason about the potential attacks that a Byzantine server or client can make, including collusion between clients and servers, and discuss and implement the appropriate countermeasures. It is expected that students systematically demonstrate the ability of their system to withstand a variety of Byzantine attacks.

The system implements a native cryptocurrency called DepCoin. A client of the system can perform DepCoin transfers between a pair of accounts provided it owns the corresponding private key of the crediting account (the one from which money is withdrawn). The set of all accounts and the associated balance should satisfy the following dependability and security guarantees:

- the balance of each account should be non-negative
- the state of the accounts cannot be modified by unauthorized users
- the system should guarantee the non-repudiation of all operations issued on an account

The system should start with a pre-built block (i.e., genesis block or block 0) that includes two deployed smart contracts, where both contracts should be written in Solidity and compiled to EVM bytecode:

- One contract that implements an ERC-20 [2] fungible token. The ERC-20 token shall be named “IST Coin”, its symbol shall be “IST”, contain 2 decimals, and a total supply of 100 million units. When performing a *transfer* or *transferFrom* the ERC-20 contract shall call the access control contract to check whether the client account address is allowed to transfer. You may use the ERC-20 implementation of OpenZeppelin as a reference [3].

- One contract that implements a blacklist-based access control list. This smart contract must provide the interface below. Note that only authorized parties must be able to modify the blacklist.

    - *addToBlacklist(address) returns (bool)*
    - *removeFromBlacklist(address) returns (bool)*
    - *isBlacklisted(address) returns (bool)*

Furthermore, students must create a genesis file, for instance in JSON format, that represents the world state of the first block (i.e., block 0) and includes an initialization with all account addresses, their balances, and smart contract addresses including their bytecode and initial key value storage values. An example could look like this:

```
{
        “block_hash”: <HASH_OF_LIST_OF_TRANSACTIONS>,
        “previous_block_hash”: null,
        “transactions”: []
        “state”: {
                “<EOA account address>”: {
                        “balance”: “100000”
                },
                “<Contract account address>”: {
                        “balance”: 0,
                        “code”: “0x60016002…”
                        “storage”: {
                                “key1”: “value1”,
                                “key2”: “value2”,
                                …
                        }
                }
                …
}
```

Transactions should be grouped into blocks and each block should point to its previous block and represent the world state after executing the transactions included within the block as compared to the previous block. Blocks should be persisted using the same format as the genesis block.

Implementation requirements

The project should be implemented in Java using Hyperledger Besu for the execution of Solidity smart contracts. Hyperledger Besu [1] is an open source Ethereum client developed under the Apache 2.0 license and written in Java and supported by the Linux Foundation. It includes an EVM implemented in Java that will be used in the project.

The account model should be based on the model of Ethereum, meaning that there are two types of accounts, Externally Owned Accounts (EOA i.e., public/private key pairs) and Contract Accounts (smart contracts). Each account (independent of the type) will be associated to an address and have a balance (i.e., amount of native cryptocurrency owned by the account). Accounts of type Contract Account also have code (EVM bytecode) and storage (key value store) associated.

The concept of transactions does not have to be equivalent to the model of Ethereum. You are free to design your own transaction object, however, it must enable the execution of smart contracts functions and the transfer of native cryptocurrency.

Implementation Steps

To help in the design and implementation task, we suggest that students break up the project into a series of steps, and thoroughly test each step before moving to the next one. Having an automated build and testing process (e.g., JUnit) will help students progress faster. Here is a suggested sequence of steps:

- Step 1: Ensure that all the requirements from the first stage are satisfied.
- Step 2: Implement the smart contracts in Solidity and test their execution using the provided EVM Hyperledger Besu boilerplate code from the second lab assignment (i.e., lab2).
- Step 3: Implement the concept of a genesis block, transactions, transaction execution (e.g., smart contract execution), as well as appending and persisting blocks.
- Step 4: Connect the consensus algorithm developed in the first stage to the components developed in the first stage. Consensus should be responsible for deciding on the order of transactions to be executed and appended to the blockchain.
- Step 5: Implement a thorough set of tests that demonstrate that the implementation is robust against Byzantine behavior.

Submission

Submission will be done through Fénix. The submission shall include:

- a self-contained zip archive containing the source code of the project and any additional libraries required for its compilation and execution. The archive shall also include a set of demo applications/tests that demonstrate the mechanisms integrated in the project to tackle security and dependability threats (e.g., detection of Byzantine behavior from blockchain members). A README file explaining how to run the demos/tests is mandatory.
- a concise report of up to 8 pages in LNCS format [4] with:
    - explanation and justification of the design, including an explicit analysis of the possible threats and corresponding protection mechanisms.
    - explanation of the dependability guarantees provided by the system.

The deadline is **April 2 at 23:59**. More instructions on the submission will be posted in the course page.

Ethics

The work is intended to be done exclusively by the students in the group and the submitted work should be new and original. It is fine, and encouraged, to discuss ideas with colleagues – that is how good ideas and science happens - but looking and/or including code or report material from external sources (other groups, past editions of this course, other similar courses, code available on the Internet, ChatGPT, etc) is forbidden and considered fraud. We will strictly follow IST policies and any fraud suspicion will be promptly reported.

References

[1] Besu Ethereum Client. https://github.com/hyperledger/besu/tree/main

[2] ERC-20 Token Standard
https://ethereum.org/en/developers/docs/standards/tokens/erc-20

[3] OpenZeppelin – ERC20 Implementation
https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20

[4] Information for Authors of Springer Computer Science Proceedings.
https://www.springer.com/gp/computer-science/lncs/conference-proceedings-guidelines