# Software Engineer Test Task

## Problem Statement

Implement a simple decentralised payment system that will resemble existing cryptocurrencies.

The system *shall* be based on a ledger, and it *shall* maintain the state consisting of a mapping from public keys to the amounts of coins belonging to them:

$$\text{ledger} : \text{PublicKey}_u \to \text{Amount}_u$$

The owner of the secret key *shall* be able to transfer some of the coins that belong to them to another user, provided that they know the other user's public key. Coins are indivisible and you *may* enforce an upper limit on the amount of coins available on a single account.

All transactions *must* be authorised, meaning that only the current owner of the coins will be able to move them from their account to someone else's. The system *should not* enforce any trust assumptions on its users.

## Functionality

Nodes *shall* be able to support the following queries:

1. `SUBMIT tx`: each node (including a disconnected one) should be able to validate a new transaction and add it to the local ledger. The response to this query *shall* be `Just txId` if the transaction was accepted (and assigned id `txId`), and `Nothing` otherwise.
2. `QUERY txId`: each node *shall* be able to reply to the query "was transaction with id `txId` added to the local ledger?"
3. `BALANCE pubKey`: each node *shall* be able to reply with actual balance of public key `pubKey` as for local ledger state

where `tx` is a data structure that consists of `from`, `to`, and `amount` fields.

### Restrictions

Node should respond `TRUE` to `QUERY txId` iff the transaction was submitted to network and was considered valid by node.

We don't expand on transaction validation rules here (task performer is welcome to derive them by himself from the notion of ledger).

Nodes should eventually reach consensus. In particular, for any given transaction `tx` submitted more than `stabilityTimeout` ms ago, either *must* hold:

- Each node responds `TRUE` on `QUERY txId`
- Each node responds `FALSE` on `QUERY txId`

This predicate is to be checked among all nodes which weren't in disconnected state at least for the last `resyncTimeout` ms.

## Security model

1. Each node in the system behaves honestly at any given moment (i.e. adversary is not allowed to launch a node or hack into an existing node)
2. Adversary has the ability to disconnect any single node from the network:
    - At each particular moment, at most one node may be disconnected
    - Adversary can't perform the disconnect operation more often than once in `disconnectTimeout` ms
3. Adversary has online access to all conversations between nodes in the network
    - Adversary has no access to conversation between node and client
4. Adversary holds some predefined balance in the system and is able to use the functionality of a node (`SUBMIT`, `QUERY`)

## Technical Requirements

The system *must* be implemented in Haskell. You *may* use any third-party libraries you find appropriate for this task.

You *may* have nodes play different roles in the network if that helps.

You *may* assume that the network connections between any two active nodes are reliable and all the messages will be delivered in a timely manner.

Your code *must* be buildable with `stack build` command.

### Digital signature algorithm

You *must* use the `ed25519` package for digital signatures (see `https://hackage.haskell.org/package/ed25519`).

Both public and secret keys *must* be serialized for network communication as unwrapped byte strings without length prefix.

### Node launching

Every new node *shall* be launched using the following CLI:

```
$ ./node id
         nodeCount socketDir
         disconnectTimeout
```

```
stablityTimeout
resyncTimeout
distributionFile
```

where

- `nodeCount` is the number of nodes in the network
- 0 <= `id` < `nodeCount` is the unique identifier of a new node
- `socketDir` is a path to the directory with Unix sockets
- `disconnectTimeout`, `stabilityTimeout`, and `resyncTimeout` are as described above and are provided in milliseconds
- `distributionFile`: file providing info about how coins are distributed at start of system

Distribution file is a collection of lines in following format:

```
<base-16 encoding of ed25519 PK, 64 ASCII chars> <amount, number of value up to `2^31 - 1
```

Example of Distribution file (two accounts with 120 and 1029 coins):

```
1b05041342cfc016ba6355024ff48d619ee89069f97e196d45cadd37d49232e7 120
4cf139fb27a976991210ec4c4b663b3dcfa6bb43dc09677c74e7898be368bd26 1029
```

## Communication format

Nodes communicate with each other using UNIX sockets.

Each node has a single UNIX socket file in the socket directory, named after its id, e.g. if node 2 wants to connect to node 3, it uses the file `<socket dir>/3.sock`.

Nodes must not communicate in any other form except through these sockets.

Implementation *must* use functions `connectToUnixSocket`, `listenUnixSocket` from `Serokell.Communication.IPC` module for communication via UNIX sockets.

Note that these functions are specially implemented to be suitable for testing submissions. In particular, there are two notions in place to emulate real network communication upon fast and reliable UNIX sockets:

- When `SRK_SOCK_CONC=C` env variable is provided, all outgoing sends (i.e. sends which are made within connections initiated by `connectToUnixSocket`) are executed under semaphore with capacity `C`
- When `SRK_SOCK_DELAY=D` env variable is provided, all outgoing sends are delayed by `D` ms

This module is located in `serokell-test-task-lib` top-level package, in the archive shared with you. Use `serokell-test-task-lib` package as external dependency in your `stack.yaml`.

The commands that a node can accept are encoded as follows:

- `SUBMIT tx`:

```
0 <Ed25519 secret key> <Ed25519 public key> <amount>
  (64 bytes)           (32 bytes)          (4 bytes)
```

(101 bytes total)

If the transaction was added, the response is:

```
1 <transaction id>
  (32 bytes)
```

(33 bytes total)

or otherwise:

```
0
```

(1 byte total)

- `QUERY tx`

  ```
  1 <transaction id>
    (32 bytes)
  ```

  (33 bytes total)

  The response is 1 byte: `0` (False) or `1` (True)

- `BALANCE pubKey`:

  ```
  2 <Ed25519 public key>
    (32 bytes)
  ```

  (33 bytes total)

  The response is 4-byte:

  ```
  <amount>
  (4 bytes)
  ```

## Limits

Limits are bounded by relation:

```
SRK_SOCK_DELAY < disconnectTimeout < resyncTimeout < stabilityTimeout
```

The solution author may claim more accurate bounds on limits his solution works with.

## Sample implementation

In the archive shared with you, we provide a sample implementation which can be used as a basis for your own implementation, although it diverges from the description above in certain ways, namely:

- The communication format differs from the required one.
- Only a mock-up of the functionality is implemented.

# Grading

Task should be submitted as a zip archive or a private GH repository.

Your solution will be:

- Tested on providing required Functionality in a correct way
- Benchmarked on how many transactions per second (tps) it may sustain while still providing required Functionality
- Checked against various attack vectors Adversary may handle (automatically and via code review)
- Reviewed with consideration of the code practices used

During automated testing, various values of node count, limits, `SRK_SOCK_CONC`, `SRK_SOCK_DELAY` will be used.

During task validation, reviewer may address his questions to the solution author, ask for some modifications.

Grading criteria (and the scoring method) will be shared with the solution author together with the review.