Project: Blockchain

AKSAMENTOV Ivan
DINTE Maria

University of Strasbourg, 2018

Introduction

This project is focused on the simulation and visualization of the evolution of a blockchain. We implement a simple distributed cryptocurrency, similar to Bitcoin that uses blockchain to keep transactions records and relies on a decentralized peer-to-peer distributed protocol.

System tequirements

This software is cross-platform and will run anywhere where there is a Python 3.6 and Node.js 8 available. The code was developed and tested on Ubuntu 14.04 and 16.04.

Directory structure

Nodes

Directory or file	Contents
`.bc/`	Python virtual environment will be installed here
`.conda/`	Python (miniconda) will be installed here
`.data/`	ECDSA private keys will be saved here
`.tmp/`	Temporary files will be stored here
`blockchain/`	Python library package implementing blockchain-related functionality: blocks, transactions, Merkle tree etc.
`blockchain_rpc/`	Python library package implementing RPC-related functionality: initializing gRPC server and client, Node classes etc.
`dns_seeder/`	Python executable package that implements DNS seeder node
`full_node/`	Python executable package that implements full node
`protos/`	Protobuf gRPC service definition (shared between Python and

	Node.js code)
`scripts/`	Helper bash scripts for nodes
`tests_unit/`	Automated unit tests
`.env`	Configuration for the visualization app
`.env.defaults`	Configuration defaults for the visualization app

Visualization web app

Directory or file	Contents
`viewer/`	Visualization app
`viewer/.tmp/`	Temporary files will be stored here
`viewer/app/`	Express.js-based Node.js server application that serves Next.js client app as well as REST API
`viewer/client/`	Next.js-based client application
`viewer/common/`	Code shared between server and client
`viewer/node/`	Node.js will be installed here
`viewer/node_modules/`	Node.js packages will be installed here
`viewer/scripts/`	Helper bash scripts for the viewer
`viewer/.babel`	Configuration for Babel JavaScript transpiler
`viewer/.eslintrc`	Configuration for ESLint static analyzer
`viewer/.env`	Configuration for the visualization app
`viewer/.env.defaults`	Configuration defaults for the visualization app
`viewer/package.json`	Package definition for the visualization app

Installing and running

The general usage scenario is as follows:

- Run DNS seeder node to bootstrap the peer-to-peer network
- Run one or more full nodes and oberve they start communicating and mining blocks
- Run visualization app to see list of nodes, contents of blocks and list of pending transactions

Installing and running nodes

Blockchain nodes are implemented in Python 3. The source code is contained in the directories in of the project root (with the exception of directory `viewer/`),

Installing dependencies

Provided script will install a fresh version of python via conda (miniconda, actually), will create a local python virtual environment in directory `.bc` and will install python packages required for the project (see `requirements.txt`).

```
cd <PROJECT ROOT>

# Optionally, remove the previous artifacts
rm -rf .bc .data .conda .tmp

# Run the installer script
scripts/install
```

`scrypt` package requires OpenSSL headers and library to be installed.
On Ubuntu it can be installed with:

```
sudo apt-get install libssl-dev
```

Configuring the environment

Nodes use the settings read from the file `.env` in the root directory. The default configuration example is provided in `.env.defaults`

Copy the default configuration file:

```
cp .env.defaults .env
```

Modify the default settings in `.env` file if necessary.

Activate virtual environment "bc"

```
source .bc/bin/activate
```

Running unit tests

In order to run all unit tests, run:

```
./scripts/test
```

Running the DNS seeder node:

DNS seeder is a trusted node, which address is hardcoded by software maintainers into all nodes. It provides initial node discovery in order to bootstrap the peer-to-peer network.

```
./scripts/run dns_seeder
```

Running the full node:

Full node maintains the blockchain and mines new blocks

```
./scripts/run full_node <host>:<port>
```

or simply:

```
./scripts/run full_node :<port>
```

to run on localhost

Installing and running visualization app

Blockchain visualization is implemented as a React/Node.js web application. The source code for it is located in subdirectory `viewer/`

Installing dependencies

The installer script will install the LTS version of Node.js locally and will install the required packages via NPM (see `package.json`)

```
cd <PROJECT ROOT>/viewer

# Optionally, remove the previous artifacts
rm -rf .build .tmp node_modules

# Run the installer script
scripts/install
```

Configuring the environment

Nodes use the settings read from the file `.env` in the root directory. The default configuration example is provided in `.env.defaults`

Copy the default configuration file:

```
cp .env.defaults .env
```

Modify the default settings in `.env` file if necessary.

Running the app in production (release) mode

Production build of the app (both client and server) can be triggered by the included script:

```
./scripts/build
```

The result will be generated in directory `.build/` .

After that, the production app can be started with:

```
./scripts/start
```

By default, the app will be served at `http://localhost:3000` (
this can be changed in `viewer/.env`)

Running the app in development (debug) mode

Development version of the app, with hot reloading and additinal debugging capabilities, can be started with:

```
./scripts/dev
```

General description

Nodes

The peer-to-peer network consists of nodes. Every node is operated using provided software. All nodes are identical, compatible and are identified only by currently used ECDSA key pair (see below).

In functional capabilities there are few types of nodes:

- *Full nodes* maintain blockchain, create (mine) new blocks and may (or may not)

generate transactions. They periodically send transaction and block discovery and sharing requests to all known peers, in order to stay updated on the state of the blockchain and to receive as many new transactions as possible. They broadcast the newly produced blocks as well.
- *Partial nodes* keep track of blockchain and generate transactions. They do not participate in mining. This is what typically called a wallet: it allows users to send payments without bothering with the complexity of mining.
- *Viewer node* passively observes the blockchain and does not generate transactions nor blocks. For demonstartion purposes.
- *DNS seeder nodes* do not participate in blockchain operations at all and only provide peer discovery service for other nodes, which is useful on initial bootstrapping of the peer-to-peer network (see below).

We don't specifically distiguish "block nodes" or "participant nodes". The operation mode can be changed in parameters of a node. It is up to user to choose whether to run mining, generate transactions or to forward peer information at any given time.

Identities

The notwork is fully anonymous. Any user of the system can creat any number of nodes and change keys and addresses at any time. No real identity verification is required.

Peer-to-peer network, peer discovery and advertisement

On initial launch nodes don't know about existence of other nodes. However, every node has at least one hardcoded address of a trusted DNS seeder node. If this seeder node is online, on launch nodes try to connect to it and to get peer information from it. DNS seeder simply forwards addresses of all peers it ever heard about to every other node.

Nodes then proceed to connection to these received nodes and, this way, bootstrap a peer-to-peer communicaion without DNS seeder. Nodes further explore the network by sharing and listening for the peer information messages.

In the end, network tends to organize itself into a dense mesh, where every node is connected to each other. This is beneficial for all types of nodes: for example, this way, partial nodes can broadcast their transactions to many nodes, *reducing delay* of inclusion into the blockchain, while full nodes receive many transactions and are free to chose the best set of transactions to collect *transaction fees* from.

Blocks

Block is an elementary unit of the blockchain. Due to hash pointers and cryptographic signatures, blocks has an important property: once added, block cannot be removed or changed without breaking the chain (which will be detected and modifications will be ignored by other nodes).

In our cryptocurrency, the purpose of blocks is to store the financial transaction history. Every transaction is a record of transferring coins from one address to one or multiple other addresses (see below).

Every block contains (see `protos/blockchain.proto`) its own *hash*, a *hash to previous block* in the chain a *nonce* integer field (that is to be found during mining, see below) a set of transactions and a *root of the Merkle tree*.

The hash is being calculated via a pre-defined hash function (see puzzles section).

Proof of work, mining and puzzles

Mining is a process of finding new blocks by solving a computational puzzle. When creating a new block, miner node needs to find a hash that satisfy a particular requirements of the difficulty. In our case we require hashes to be less that a pre-defined target integer value. This significantly narrows the window on valid hashes, and requires significant computations to be performed in order to find a valid hash. Hash functions in use are thought to be irreversible, so the only way to find a valid hash is to try many of different hashes by varying the *nonce* field.

In our application, there is a choice among 2 puzzles that differ in hash function used:

- *SHA-256* puzzle, computationally-intensive puzzle, similarly to that of Bitcoin
- *scrypt* puzzle, memory-intensive, ASIC-resistant puzzle, similarly to that of Litecoin

The reason of using memory-intensive puzzles is that they allow to reduce the speed of growth hashing power of the network (speeds on memory access grow slower that arithmetic processing speeds).

Miners have an *incentive to act honestly* and follow the described protocol, because after inclusion of a new block and verification of it by other nodes, miner receive a significant benefit from *coinbase transaction* as well as gathered *transaction fees* (see section about transactions).

Merkle Tree

Merkle Tree is a data structure, a variant of hash tree, that is used to store the transactions in the blocks. Merkle tree allows to quickly verify that a given set of transactions belongs to a given block, without storing or transferring the transactions themselves. It summarizes the data, allowing for significant bandwith savings and disconnects the validation from the data itself by using hash and reduction functions.

Transactions

Transaction is a record of transferring coins from one address to one or multiple other addresses. Transactions are being store in blocks. Every transaction contains a set of *inputs* and a set of *outputs*.

Every *input* of a transaction refer to an *outputs* of a transaction located previously in the blockchain., using hash of that transaction and an index of that output. Inputs are signed using ECDSA signatures (see below). This allows to verify that coins that the node tries to spend in this transaction are belong to it.

Every *output* provides an amount of coins being transferred and the ECDSA public key of the node that will receive the payment.

There can be zero to multiple inputs and at least one output. The sum of inputs should be spent in the sum of outputs entirely: spending node adds another output to send the change (excess of coins) back to itself. Users typically output slightly less than input and the difference becomes a *transaction fee*. Miner node that includes the transaction into the new blocks is free to keep the fee to himself. Miner typically prefer to process the transactions with higher fees and include them faster into the chain.

The node that creates the block should include *coinbase transaction* as its first transaction. The coinbase transaction has no inputs and has one or more outputs. The node is free to chose the receiver of the coins in the output (nodes typically choose to sent the coinse to themselves). This is the main incentive to perform the mining.

Elliptic Curve Digital Signature Algorithm (ECDSA) and signatures

ECDSA signatures are used for signing transactions. Every node generates a public-private key pair as well as an address. Address or public key uniquely identify the node and allow it to claim the coins it produced from mining or received from transactions.

Visualization

Visualization is implemented as a Node.js web app.

Note that, as the system is peer-to-peer and there are no explicit synchronizations mechanisms, there is no unified blockchain state possible. Every node posesses it's own view into the blockchain and distributed consensus protocol dictates which parts of any of these views of are valid. Even the amount of coins a node owns is a subject to consensus. The absence of a ground truth version of blockchain has direct consequences for the visualization of the system.

Bugs and limitations

Due to limited time we were unable to fully implement or test the following features:

- Transaction generation according to all the rules of the protocol and transaction

verification according to all the rules of the protocol. Both features require a fast algorithm for backwards search in the blockchain and we haven't done it fully.
- We tried to implement the real-time updates in the visualization app by pushing new data through websockets. It worked, however there was a bug causing multple websocket subscription of a same client, so that number of updates grew quickly and made browser to hang. It is probably due to the bug in how we setup the redux-saga. For now page
refresh or button click is required for updates (via REST API).

Conclusion

In this project we have implemented a simple blockchain-based cryptocurrency. The cryptocurrencies change the standards of financial institutes and open new horisons for developing distributed protocols. They are the playground for new research in algorithms, economy and game theory.