Blockchain
Documentation

Paolo Bettelini, Gianni Grasso, ~~Giacinto Di Santis~~
Scuola d'Arti e Mestieri di Trevano (SAMT)

Contents

1 Introduction

1.1 Abstract

This project is not meant to be an effectively secure or scalable blockchain.
The goal is to produce a *proof-of-concept* software for educational purposes.

1.2 Information

This is a project of the Scuola Arti e Mestieri di Trevano (SAMT) under the following circumstances

- **Section**: Computer Science
- **Year:** Third
- **Class:** Module 306
- **Supervisor:** Luca Muggiasca
- **Title:** Blockchain
- **Start date**: 2021-27-01
- **Deadline**: 2022-05-05

and the following requirements

- **Documentation**: a full documentation of the work done
- **Diary**: constant changelog for each work session
- **Source code**: working source code of the project

All the source code and documents can be found at https://github.com/LuMug/Blockchain [**gitrepo**].

1.3 Structure

This document is structured as such:

1. **Introduction:** General informations, requirements and scope of the project
2. **Blockchain:** How a blockchain works
3. **Blockchain implementation:** Our blockchain implementation
4. **Programming language:** How to build a programming language
5. **Programming language implementation:** Out programming language implementation

2 Analysis

2.1 Requirements

Req-01	
Name	Proof-of-Work
Priority	1
Version	1.1
Notes	none
Description	The blockchain consensus must be based on the Proof-of-Work algorithm.

Req-02	
Name	Peer Discovery
Priority	1
Version	1.1
Notes	none
Description	Nodes must be able to find and connect to eachother.
Subrequirements	
Req-01_0	Peer discovery based on centralized servers must be minimized.

Req-03	
Name	Smart Contract
Priority	1.1
Version	1.0
Notes	none
Description	Nodes must be able to process smart contracts.

Req-04	
Name	Programming language
Priority	2
Version	1.1
Notes	none
Description	A custom programming language must be developed in order to write smart contracts.

Req-05	
Name	Forger tool
Priority	2
Version	1.1
Notes	none
Description	A utility tool to generate keypairs and sign transactions must be developed.

Req-06	
Name	API
Priority	1
Version	1.1
Notes	none
Description	A node with HTTP POST routes must be developed.
Subrequirements	
Req-06_0	There must be a route to get the latest mined blocks
Req-06_1	There must be a route to get block informations
Req-06_2	There must be a route to get wallet informations
Req-06_3	There must be a route to get all the transaction regarding a wallet
Req-06_4	There must be a route to deploy transactions
Req-06_5	Responses must be in the JSON format

Req-07	
Name	Miner
Priority	1
Version	1.1
Notes	none
Description	A node that will solve Proof-of-works must be developed.
Subrequirements	
Req-06_0	The number of CPUs used must be configurable

Req-08	
Name	Executables
Priority	1
Version	1.1
Notes	none
Description	Every software must be shipped in a single executable.
Subrequirements	
Req-08_0	Executable for the seeder software
Req-08_1	Executable for the full-node software
Req-08_2	Executable for the miner-node software
Req-08_3	Executable for the api-node software
Req-08_4	Executable for the webserver
Req-08_5	Executable for the forger software
Req-08_6	Executable for the programming language compiler
Req-08_7	Every executable must have an help page

Req-09	
Name	Website
Priority	1
Version	1.1
Notes	none
Description	A website about the blockchain must be developed.
Subrequirements	
Req-09__0	The website must contains search form to search blocks, wallets and transaction
Req-09__1	The website must have a section to deploy transactions

3 Blockchain

3.1 Block

Each user owns a pair of private and public key.

All the transactions broadcasted to the network are grouped into blocks, which contain

- Markle tree root hash
- Timestamp
- nBits (PoW)
- Nonce (PoW)
- Previous block hash
- Number of transactions

With each block being confirmed, the blockchain is created.

3.2 Proof of Work

Proof-of-Work (PoW) is a cryptographic proof that a party has spent a certian amount of computational effort.

When a miner solves the puzzle the current block is archived, a new block is generated and all the transactions in the previous block are confirmed. The miner is then rewarded by the system.

3.3 Proof of Stake

3.4 Smart Contracts

Smart contracs are programs associated with an address and run on the blockchain. The nodes run code from the contract program at a relevant event, such as a received transation.

Users can interact with the contract via transactions. Contracts can often interact with other contracts and some of them are Turing-complete.

3.5 Difficulty

If we want the time gap between two mined blocks to be approximately N units of time, we need adjust the difficulty of the mining process every M units of time such that

$$\text{difficulty}_{\text{current}} = \text{difficulty}_{\text{previous}} \cdot \frac{M}{\Delta(\frac{M}{N})}$$

where $\Delta(x)$ is the units of time to mine the last x blocks.

Or we could adjust the difficulty N blocks

$$\text{difficulty}_{\text{current}} = \text{difficulty}_{\text{previous}} \cdot \frac{N \cdot k}{\Delta(k)}$$

where k is the number of last blocks on which you want to base the adjustment on. These two formulas are the same.

3.6 Mempool

The mempool is the place where uncofirmed transactions wait to be confirmed. When a transaction is broadcasted and received by the nodes, if valid, it is put in the mempool. When a new block is created, up to X transactions are removed from the mempool and will be included in the next block.

3.7 Deployment

To deploy a transaction a node has to broadcast it to his peers. To avoid flooding of the network, a node will only broadcast the same transaction once (as long as the transaction is still in the mempool).

4 Blockchain Implementation

4.1 Peer Discovery

4.1.1 Registration

When a node starts it generates a random UUID.

When to peers establish a connection, they change a `RegisterNode` packet. This packet contains information about their service address and UUID.

4.1.2 Seeder

The first step of our peer discovery solution is the *Seeder*. The Seeder is a server which stores information about node's service addresses.
A node might register himself to a seeder by opening a TCP connection and sending a `RegisterNode` packet. Likewise, a node might ask a seeder the service addresses of **N** nodes by sending a `RequestNodes` packet.

When the seeder receives a `RegisterNode` it will store its address in the services pool. The services pool has a fixed capacity of `POOL_CAPACITY`. Nodes are stored along with their registration timestamp, this means that when the pool reaches max capacity the oldest node will be uncached. If an already registered node send a registration packet, the timestamp is updated. Connection is then closed.

When the seeder receives a `RequestNodes` it will try to randomly draw as many nodes as requested (max `MAX_REQUEST`) excluding the requester itself. The `ServeRequestNodes` response packet will be sent back. Connection is then closed.

It is important to only rely on a seeder when 0 peers are known or when all the known peers are unreachable.

If a node with an already registered address does register with a new UUID, the old entry is removed.

4.1.3 Cache

The nodes will also locally cache some of the peers. To do so we define the *node* table.

```sql
CREATE TABLE IF NOT EXISTS node (
    address VARCHAR(20),
    port INT,
    last_seen_alive DATETIME,
    PRIMARY KEY (address, port)
);
```

When a connection to a peer is established, the node is added to this table. If the node is already in the database, its *last_seen_alive* field is updated.
If the number of nodes in the cache exceeds `MAX_CACHED_NODES`, the node with the oldest *last_seen_alive* field is deleted.

4.1.4 Algorithm

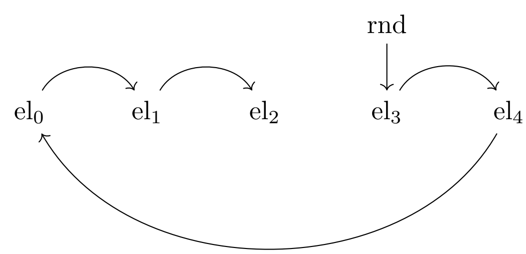
A node will actively try to always have `MIN_CONNECTIONS` established connections. If the number of connections exceeds `MAX_CONNECTIONS`, a random peer is disconnected.

The node will periodically register himself to a random seeder every `REGISTER_INTERVAL` ms. By doing this dead nodes will stop registering and exit the seeder pool.

The node will periodically execute the following update every UPDATE_INTERVAL ms:

1. If peers are less than MIN_CONNECTIONS, update from a random peer up to MAX_TRIES_NEIGHBOUR times
2. If peers are less than MIN_CONNECTIONS, update from local cache
3. If peers are 0, update from a random seeder up to MAX_TRIES_SEEDER times

Picking a random seeder
Seeder addresses are hard-coded in the SEEDERS array. Start with a random index, if that seeder is unreachable traverse the array in a circular manner until one is reachable.



Updating from local cache

1. Read all cached peers from the database
2. Filter the ones without an estblished connection
3. While there is still room, try to establish a connection with them

Updating from seeder

1. Establish a TCP connection with the seeder
2. Send a RequestNodes packet requesting (MIN_CONNECTIONS − CONNECTIONS) peers
3. Filter the ones without an estblished connection
4. Try to connect to each of them

Update from a peer

1. Send a RequestNodes packet requesting (MIN_CONNECTIONS − CONNECTIONS) peers
2. Filter the ones without an estblished connection
3. Try to connect to each of them

When a node receives a RequestNodes it will try to randomly draw as many nodes as requested (max MAX_REQUEST) excluding the requester itself. The ServeRequestNodes response packet will be sent back.

4.2 Wallet

4.2.1 Keypair

A wallet can be create by generating an **EdDSA** (Elliptic Curve Digital Signature Algorithm) keypair on the *ed25519* curve.

The public key can be retrieved given the private key.

4.2.2 Address

The public wallet address is given by the hash of the public key.

$$\text{address} = \text{SHA}_{256}(\text{key}_{\text{priv}})$$

and the human-readable version

$$\text{address}_{\text{UTF-8}} = \text{base64}(\text{SHA}_{256}(\text{key}_{\text{priv}}))$$

4.3 Mining

A miner will try to compute the following:

$$\theta = \text{hash}_{\text{last}} \oplus \text{SHA}_{256}(\text{height}) \oplus \text{SHA}_{256}(\text{nonce}) \oplus \bigoplus_i \text{SHA}_{256}(\text{tx}_i)$$

Where $\oplus$ denotes the XOR operator and $\bigoplus_i$ denotes the XOR over a set, so $0 \leq i < \text{length}(\text{tx})$.

A block is mined if a value for *nonce* such that

$$0 = \sum_{n=0}^{\text{difficulty}} \left(\text{SHA}_{256}(\theta \oplus n) \mod 2\right)$$

is found.

When a peer receives a `SendTransactionPacket` it checks its validity. If the transaction is valid it gets put into the mempool. At the arrival of the next valid `PoWSolvedPacket`, the transactions are drawn from the mempool, inserted into the and inserted into the mining computation. Finally, at the arrival of the next valid `PoWSolvedPacket` the transactions will be effectively confirmed.

Since `SendTransactionPacket` packets contain the sender's public key rather than the address itself, a SQL cache is constructed:

```sql
CREATE TABLE IF NOT EXISTS keyCache (
        pub_key BINARY(32),
        address BINARY(32)
)
```

Here's the SQL table for storing blocks

```sql
CREATE TABLE IF NOT EXISTS block (
        id INT PRIMARY KEY,
        difficulty INT,
        tx_hash BINARY(32),
        nonce BINARY(32),
        miner BINARY(32),
        mined DATETIME
);
```

and for storing transactions

```sql
CREATE TABLE IF NOT EXISTS tx (
        block_id INT,
        sender_pub BINARY(32),
        recipient BINARY(32),
        amount INT,
        timestamp DATETIME,
        last_tx_hash BINARY(32),
        signature BINARY(64)
);
```

UTXO can be calculated given every block and transaction, so here's a table to cache this value.

```sql
CREATE TABLE IF NOT EXISTS wallet (
        address BINARY(32),
        amount INT
)
```

4.4 Nodes

There are three types of nodes:

- **Full Node** Full functional node
- **Mining Node** Node that produces Proof-of-Works
- **API Node** Node with HTTP POST routes

Both miner node and API node extend the full node, so they are both also functional full nodes.

Here is a table of what each node type knows and can do.

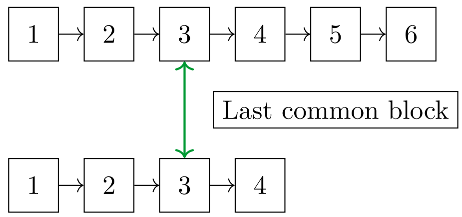
	Makes new blocks	**Deploys transactions**	**knows all transactions**
Full Node	-	-	+
API Node	-	+	+
Mining Node	+	-	+

4.5 Blockchain download

When a node joins the network, either for the first time or after some time of being offline, it needs to download all the transactions and new blocks.

Furthermore, in the event of a branch of the blockchain, the branch that will first reach a higher length than the other must be adopted by every node hosting the other branch.

To accomplish so, each peer periodically asks his neighbours for their blockchain length. If their length is greater than his (and there is consensus) it will try to establish a common block and download the peer's blockchain starting from that block.



Once the common block has been found the other peer will start sending `ServeOldTransactionPacket` and `ServeOldPoWPacket` packets in the correct order. The peer will also send transactions currently in the mempool. Finally, the peer sends a `DownloadDonePacket`.

A peer will not send a download request if the lengths are the same, except at startup (so that the mempool is updated and kept up to date).

4.5.1 Finding common node

Start by sending a `RequestIsHashExistsPacket` containing the hash of the last block available. If the response block ID is positive (not -1, this is the last common block. Otherwise, try the same thing with the block before. If the iteration exceeds a depth of 5, set the common ID to be 0.

4.6 Constants

Here is a list of hard-coded values and constants.

Database
Constants related to the database node caching

Name	Value
MAX_CACHED_NODES	50

Blockchain
Constants related to the blockchain system:

Name	Value
BLOCK_REWARD	1000

Node
Constants related to the peer connections:

Name	Value
MAX_CONNECTIONS	150
MIN_CONNECTIONS	100
REGISTER_INTERVAL	180000
UPDATE_INTERVAL	180000
MAX_TRIES_SEEDER	5
MAX_TRIES_NEIGHBOUR	5
SEEDERS	127.0.0.1:4670 127.0.0.1:4671 127.0.0.1:4672

Packets IDs

Name	Value
REQUEST_NODES	0x0
SERVE_NODES	0x1
REGISTER_NODE	0x2
SEND_TRANSACTION	0x3
REQUEST_BLOCKCHAIN_LENGTH	0x4
SERVE_BLOCKCHAIN_LENGTH	0x5
REQUEST_IF_HASH_EXISTS	0x6
SERVE_IF_HASH_EXISTS	0x7
POW_SOLVED	0x8
REQUEST_DOWNLOAD	0x9
SERVE_OLD_TX	0xA
SERVE_OLD_POW	0xB
DOWNLOAD_DONE	0xC

4.7 Packets

Packets are sent over a TCP socket stream. Each packet is preceded by the length of the packet, encoded as a Little-Endian 32-bit integer.

4.7.1 DownloadDonePacket

This package indicates that the requested download has ended.

Name	Type	Descritpion
ID	u8	Packet Identifier

4.7.2 PoWSolvedPacket

A miner will broadcast this packet when he mines a block.

Name	Type	Descritpion
ID	u8	Packet Identifier
nonce	blob	The mined nonce
miner	blob	Address of the miner
timestamp	u64	Timestamp

4.7.3 RegisterNodePacket

Used to register the node service to another peer or a seedeer.

Name	Type	Descritpion
ID	u8	Packet Identifier
port	u16	The port of the service
uuid	UUID	The UUID of the node

4.7.4 RequestBlockchainLengthPacket

Used to request the length of the blockchain to a peer.

Name	Type	Descritpion
ID	u8	Packet Identifier

4.7.5 RequestDownloadPacket

Used to request download of new blocks, transactions and mempool from a peer.

Name	Type	Descritpion
ID	u8	Packet Identifier
port	i32	Starting block ID

4.7.6 RequestIfHashExistsPacket

Used to ask a peer wheter a certian hash exists in his blockchain.

Name	Type	Descritpion
ID	u8	Packet Identifier
hash	blob	Hash

4.7.7 RequestNodesPacket

Used to request peers addresses to a peer or a seeder.

Name	Type	Descritpion
ID	u8	Packet Identifier
amount	u32	Requested amount
exclude	UUID	UUID to exclude

4.7.8 SendTransactionPacket

This packet is broadcasted when a node wants to deploy a transaction.

Name	Type	Descritpion
ID	u8	Packet Identifier
timestamp	u64	The timestamp
recipient	blob	Recipient address
sender_pub	blob	Sender public key
amount	u64	The amount of the transaction
last_tx_hash	blob	The hash of the last transaction
signature	blob	The signature

4.7.9 ServeBlockchainLengthPacket

Used to serve the blockchain length to a peer.

Name	Type	Descritpion
ID	u8	Packet Identifier
length	i32	Blockchain length

4.7.10 ServeIfHashExistsPacket

Used to serve the result of a block hash search. If the hash is found this packet will contain the block ID. The block ID is -1 otherwise.

Name	Type	Descritpion
ID	u8	Packet Identifier
id	i32	Block ID

4.7.11 ServeNodesPacket

This packet is used to serve the requested nodes addresses.

Name	Type	Descritpion
ID	u8	Packet Identifier
amount	i32	Amount of nodes
nodes	node...	Node addresses

4.7.12 ServeOldPoWPacket

This packet is used to serve an old Proof-of-Work packet.

Name	Type	Descritpion
ID	u8	Packet Identifier
nonce	blob	The mined nonce
miner	blob	Address of the miner
timestamp	u64	Timestamp

4.7.13 ServeOldTransactionPacket

This packet is used to serve an old trasaction packet.

Name	Type	Descritpion
ID	u8	Packet Identifier
timestamp	u64	The timestamp
recipient	blob	Recipient address
sender_pub	blob	Sender public key
amount	u64	The amount of the transaction
last_tx_hash	blob	The hash of the last transaction
signature	blob	The signature

4.8 Encodings

Here's a certain types are encoded.
Types from u16 to i64 are encoded in Little-Endian format.

4.8.1 Blob

Encoding of binary data.

Name	Type	Descritpion
length	i32	Data size
data	u32...	Blob data

4.8.2 UUID

Encoding of a UUID value.

Name	Type	Descritpion
left	u64	Left-most bits
right	u64.	Right-most bits

4.8.3 Node

Encoding of a node service address.

Name	Type	Descritpion
address	string	Address string
port	u16.	Port

4.8.4 String

Encoding of a string value.

Name	Type	Descritpion
length	u8	String length
data	u8...	String UTF-8 chars

4.9 Interactions

Interactions between nodes when a packet is sent.

Packet sent	Response
`PoWSolvedPacket`	
`RegisterNodePacket`	
`RequestBlockchainLengthPacket`	`ServeBlockchainLengthPacket`
`RequestDownloadPacket`	(`ServeOldPowPacket` | `ServeOldTransactionPacket`) + `DownloadDonePacket`
`RequestIfHashExistsPacket`	`ServeIfHashExistsPacket`
`RequestNodesPacket`	`ServeNodesPacket`
`SendTransactionPacket`	

4.10 API

The API node has some HTTP POST routes to interact with the blockchain.

Blockchain Height

Returns the height of the blockchain.
Route: `/getBlockchainHeight`
Example:

```json
{
    "status": "Ok",
    "height": 3
}
```

4.10.1 Get Block

Returns the block data given its ID.
Route: `/getBlock/<id>`
Example:

```json
{
    "status": "Ok",
    "nonce": "iKU71ffO++lmUKvJNwxyM3x/gOgjoiqlyvRxIcTKmCk=",
    "miner": "ZZ1PjoBwOZQGP0AYPmwxHsa5Z9YhRDaDbE1XIKyCUsA=",
    "timestamp": 1651363707165,
    "last_hash": "AvM6wrrEZL29GGMigfkCzflUN54yrX/jQZXImkFpcAI=",
    "hash": "6TrC8QKdegOvnonbaFQJhG6DZddkMaXwoOdst9tC+8w=",
    "nTx": 0
}
```

4.10.2 Get UTXO

Returns the UTXO of a wallet given its address.
Route: `/getUTXO/<address>`
Example:

```json
{
    "status": "Ok",
    "utxo": 1500
}
```

4.10.3 Get Transaction

Returns the transaction data given its hash.
Route: `/getTx/<hash>`
Example:

```
{
    "status": "Ok",
    "blockId": 3,
    "sender": "ZZ1PjoBw0ZQGP0AYPmwxHsa5Z9YhRDaDbE1XIKyCUsA=",
    "recipient": "SQAf8paS6Rr5nnd5dC8Bx5h6YKzdAd0Y0g9UMrTa378=",
    "amount": 500,
    "timestamp": 1651363700693,
    "lastTxHash": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
    "signature": "Z0n8TA1zenp+U0ONzSAJLot3JZ5GUedPvViW8vCN8Q6h
                  vmqB81lHsTFg4g7Gb9DFMcnB8KK6TcknAbIAfZTrBg==",
    "hash": "MmVCpq21pR/ZLYZwjaMF0AdyRVImq2YZ4Mq09mstnzw="
}
```

4.10.4 Deploy Transaction

Broadcasts a transaction packet to the network.
Route: `/deploy`
The packet data is in the HTTP body.

4.10.5 Get Transactions

Returns the list of all transactions received or spent by a given address.
Route: `/getTxs/<address>`
Example:

```
{
    "status": "Ok",
    "txs":[
        {
            "blockId": 3,
            "sender": "ZZ1PjoBw0ZQGP0AYPmwxHsa5Z9YhRDaDbE1XIKyCUsA=",
            "recipient": "SQAf8paS6Rr5nnd5dC8Bx5h6YKzdAd0Y0g9UMrTa378=",
            "amount": 500,
            "timestamp": 1651363700693,
            "lastTxHash": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
            "signature": "Z0n8TA1zenp+U0ONzSAJLot3JZ5GUedPvViW8vCN8Q6hv
                          mqB81lHsTFg4g7Gb9DFMcnB8KK6TcknAbIAfZTrBg==",
            "hash": "MmVCpq21pR/ZLYZwjaMF0AdyRVImq2YZ4Mq09mstnzw="
        }
    ]
}
```

4.10.6 Get last Transaction

Returns the last spent transaction from a given wallet.
Route: `/getLastTx/<address>`
Example:

```
{
    "status": "Ok",
    "blockId": 4,
    "sender": "ZZ1PjoBwOZQGP0AYPmwxHsa5Z9YhRDaDbE1XIKyCUsA=",
    "recipient": "SQAf8paS6Rr5nnd5dC8Bx5h6YKzdAd0Y0g9UMrTa378=",
    "amount": 500,
    "timestamp": 1651363715045,
    "lastTxHash": "Bd+Wlarl0kNjxX8ZpvFKaNA0MNjwY2RSUzKrk/iksxQ=",
    "signature": "VEgzDvWkS+/T4Tt2RBdq/4TZv5Wv6L7dxf7Xb3WqcxV0
                 svYdL/fxde3N34x04KoilvqifQ0MfqP4Nt+nntYeBw==",
    "hash": "reAXKECxKhTfpEE/60nrfD+UgYURGka5Wl97XjlrSUg="
}
```

4.10.7 Error responses

When the ID/Address/Hash is not found

```
{
    "status": "Not Found"
}
```

When the address is invalid

```
{
    "status": "Invalid Address"
}
```

When the hash is invalid

```
{
    "status": "Invalid Hash"
}
```

When the ID is invalid

```
{
    "status": "Invalid ID"
}
```

4.10.8 Other

The slash character `/` must be replaced with `%2F` in the URL.

4.11 Website

The website interact with an API node to display informations about the blockchain and deploy transactions.

The address of the API node is in `website-frontend/js/post.js`

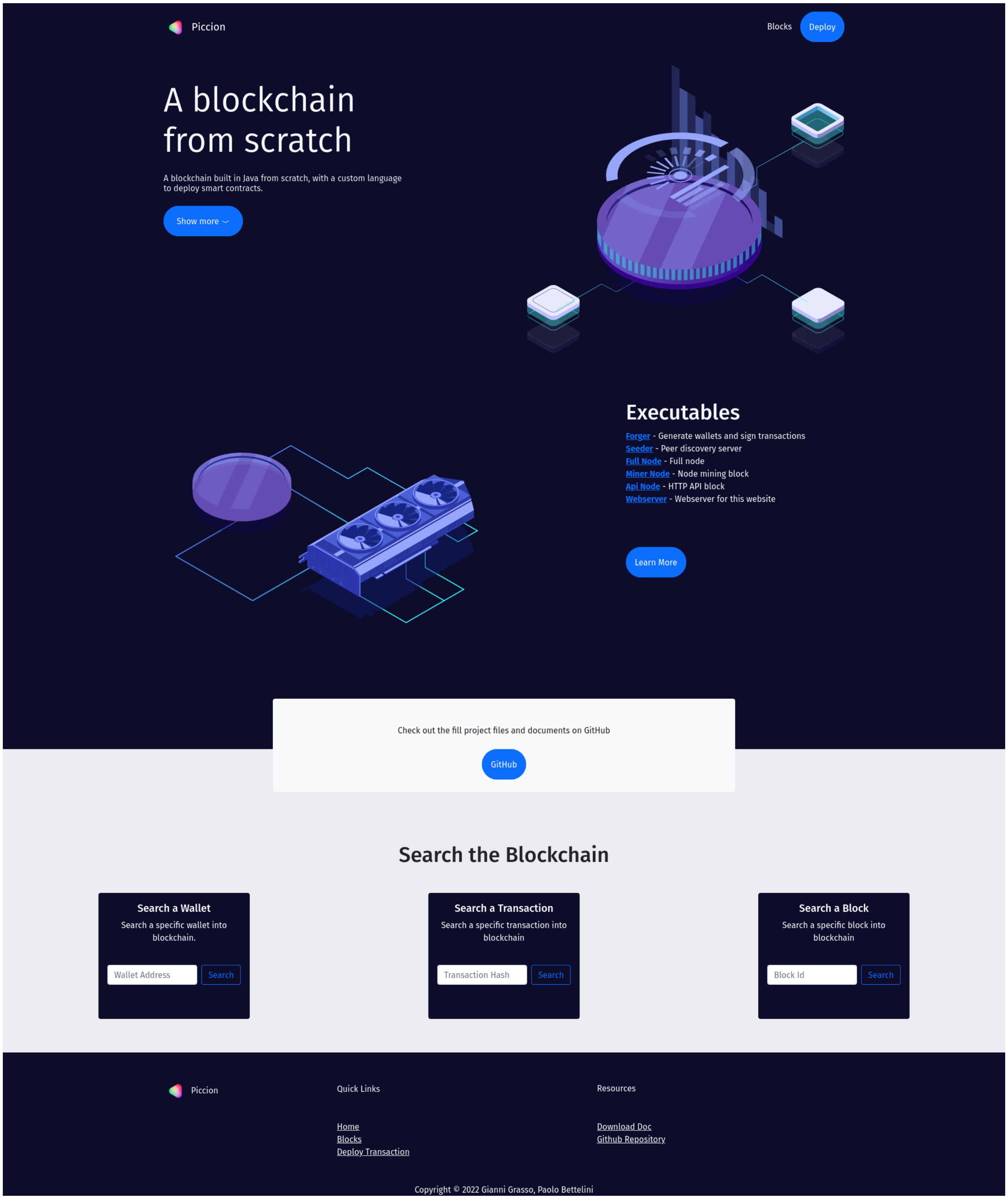
4.11.1 Dependency table

The website relies on various libraries, some of which are not stored locally. This means that the user will query third-party servers, thus the website will not work locally if you do not have a free internet connection.

Dependency table			
Name	**Description**	**Stored**	**Version**
Bootstrap (CSS)	Styling framework	Locally	5.1.3
Bootstrap (JS)	Styling framework	Remotely	4.6.1
Dropzone.js	File drop	Remotely	1.0

4.11.2 Design

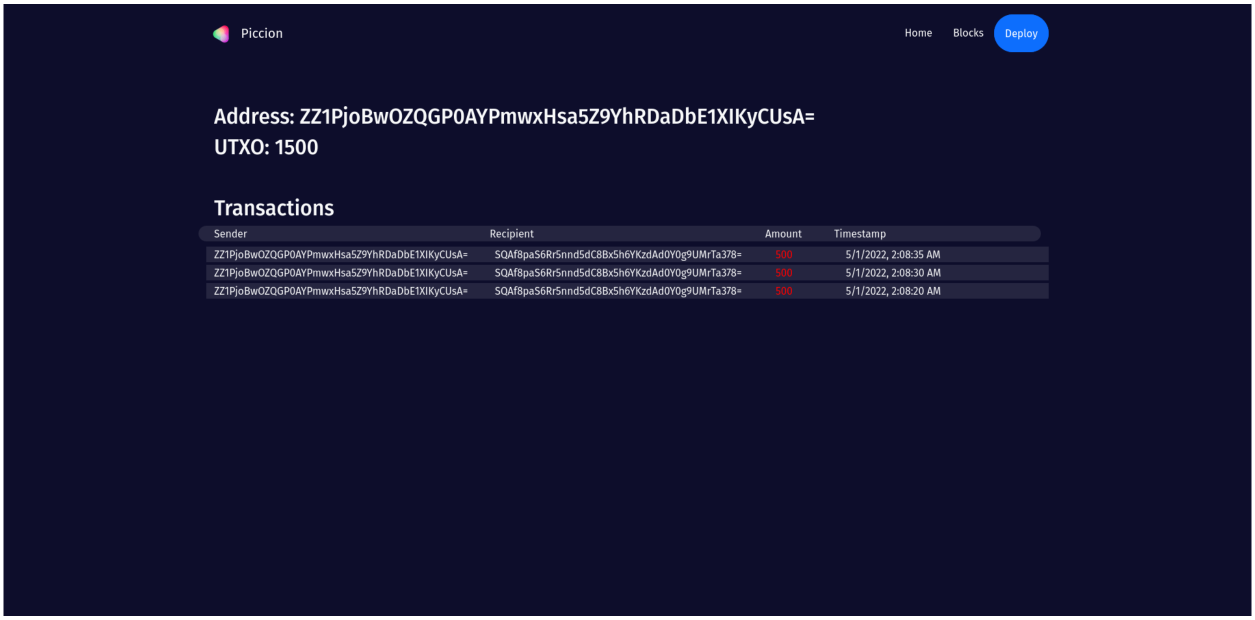
Homepage



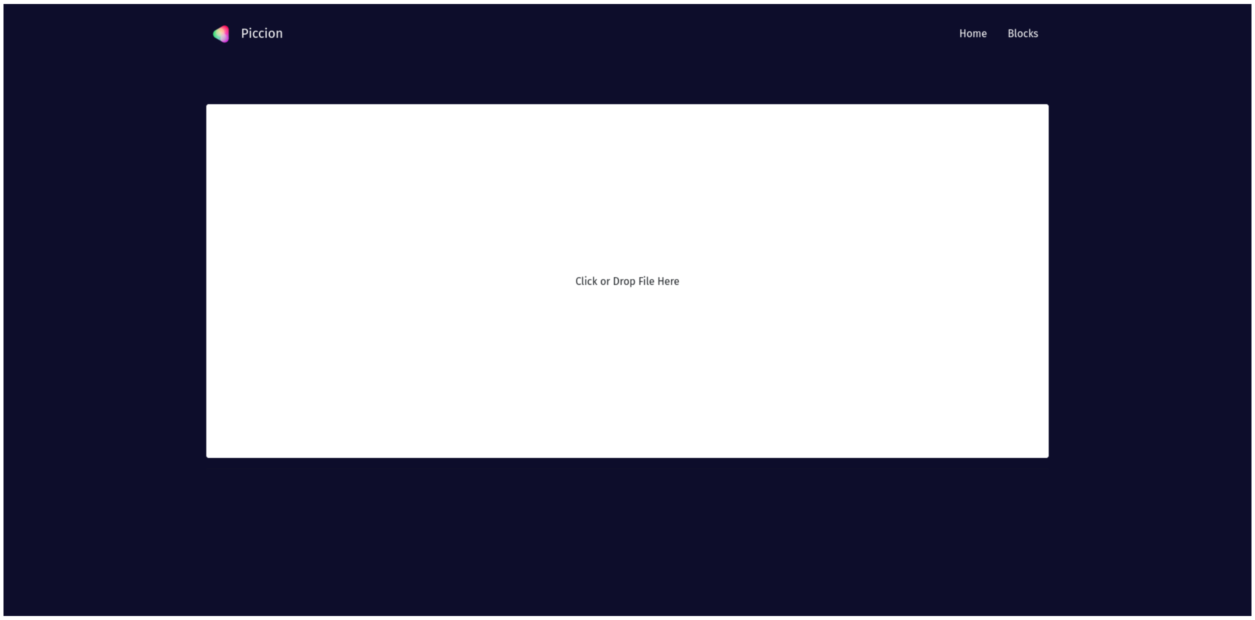
Blocks

Block

Wallet


Piccion
Home
Blocks
Deploy
Address: ZZ1PjoBwOZQGP0AYPmwxHsa5Z9YhRDaDbE1XIKyCUsA=
UTXO: 1500
Transactions
Sender
Recipient
Amount
Timestamp
ZZ1PjoBwOZQGP0AYPmwxHsa5Z9YhRDaDbE1XIKyCUsA= SQAf8paS6Rr5nnd5dC8Bx5h6YKzdAd0Y0g9UMrTa378= 500 5/1/2022, 2:08:35 AM
ZZ1PjoBwOZQGP0AYPmwxHsa5Z9YhRDaDbE1XIKyCUsA= SQAf8paS6Rr5nnd5dC8Bx5h6YKzdAd0Y0g9UMrTa378= 500 5/1/2022, 2:08:30 AM
ZZ1PjoBwOZQGP0AYPmwxHsa5Z9YhRDaDbE1XIKyCUsA= SQAf8paS6Rr5nnd5dC8Bx5h6YKzdAd0Y0g9UMrTa378= 500 5/1/2022, 2:08:20 AM

Deploy page


Piccion
Home
Blocks
Click or Drop File Here

5 Programming Language

5.1 Introduction

We have developed a custom programming language to design smart contracts, *piccions*.
The following sections contain a generic approach to designing a bytecode and the final implementation itself.

5.2 Bytecode

The bytecode is a set of low-level 1-byte long instructions (opcodes) in which a program is compiled. This set of instruction is run on a Virtual Machine (VM).

The life of a program looks as follows:

Source code → compiler → bytecode → VM

When the program is run on the Virtual Machine it is given a finite amount of memory (RAM), this memory is divided into stack and heap.

Programs run in a virtual machine are generally slower, but the advantage is that the compiled program is not CPU-specific and the program can be safely virtualed within the VM itself.

5.3 Stack

The stack is a piece of memory in which values are stacked on top of eachother. The purpose of the stack is to keep in memory temporary values for evaluating nested expressions and variables of the local scope.

5.4 Heap

The heap is the space in memory where all the objects are stored. The objects are (usually) indexed by 32-bit pointers which point to the beginning of the object within the heap. These pointers are the actual values stored in the stack. This means that for a given variable a, its pointers will be stored on the stack, and the pointer will point to a in the heap memory. The Java programming language however directly stored primitive values (numbers and such) on the stack, without a pointer to the heap.

5.5 Instructions

The two fundamental stack operations are **PUSH** and **POP**:
PUSH → pushes a value on top of the stack
POP → removes the topmost value from the stack

A simple bytecode for working with 8-bit values:

Code	**Stack before**	**Stack after**	**Description**
PSH		value	pushes the next value in the bytecode onto the stack
ADD	v1, v2	result	pop the two topmost values, adds them and pushes the result onto the stack
SUB	v1, v2	result	pop the two topmost values, subtracts them and pushes the result onto the stack
DIV	v1, v2	result	pop the two topmost values, divides them and pushes the result onto the stack
MUL	v1, v2	result	pop the two topmost values, multiplies them and pushes the result onto the stack

All of these operator work on and change the state of the stack.

Here is the code to compute $4 + 3$ and place the result on top of the stack

```
00   PSH    // push 4 onto the stack
01    4
02   PSH    // push 3 onto the stack
03    3
04   ADD
```

The stack now only contains the result of the addition, 8, which we could now print.
When the Virtual Machine will execute this code, it will do the following operations:

```
push(next())      // push next value
push(next())      // push next value
push(add(pop(), pop())) // pop two values, add them, push
```

Here's a simple pseudo-code for executing bytecode

```
while (!done) {
    var code = next()

    if (code == PSH)
        var value = next()
        push(value)

    if (code == ADD)
        var v1 = pop()
        var v2 = pop()
        var result = v1 + v2
        push(result)

    if (code == MUL)
        var v1 = pop()
        var v2 = pop()
        var result = v1 * v2
        push(result)

    if (code == SUB)
        var v1 = pop()
        var v2 = pop()
        var result = v2 - v1
        push(result)

    if (code == DIV)
        var v1 = pop()
        var v2 = pop()
        var result = v2 / v1
        push(result)
}
```

You might notice that in the DIV and SUB code, we compute v2 / v1 or v2 - v1 instead of using v1 and then v2. This is because pushing elements on top of the stack means that they will be popped in the reverse order, for MUL and ADD this is not a problem since they are commutative operations.

5.6 Nested expressions

You might think that we need some extra steps, however our stack already supports all kinds of complex nested computations.
Let's consider the following nested expression: **(2 + 2) * 4 / ((3 - 1) * 2)**.

- Start with 2+2
- Multiply it by 4

At this point we must "pause' the current value to compute to other side of the expression

- Compute 3-1
- Multiply by 2

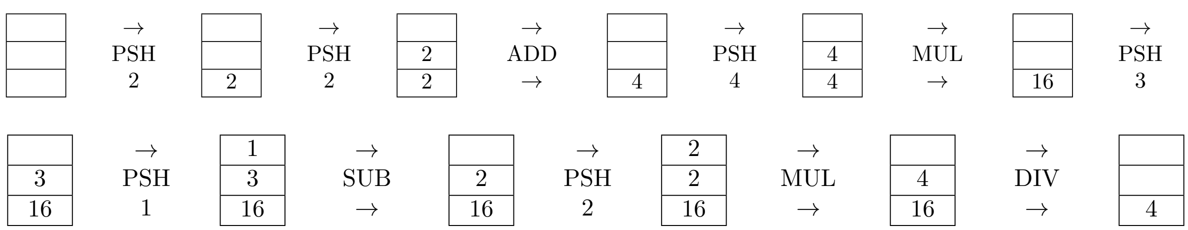
And now we will have both values on top of the stack

- Divide

The bytecode looks as follows

```
PSH 2
PSH 2
ADD
PSH 4
MUL
PSH 3
PSH 1
SUB
PSH 2
MUL
DIV
```

Here's how the stack evolves whilst computing the expression



The beauty of the stack is that it will keep stacking elements on top of eachother, but eventually everything will simplify and the result will be on top of the stack.

With the stack, you can compute any nested expression no matter how complex. The only limitation is the size of the stack, if an element is pushes to the stack but there's no more space for it, the program crashes. This is called a stack overflow.

5.7 Data types

So far we've only worked with 8-bit (1 byte) values. This means that every value pushes or popped from the stack is very limited in size [-127;128] or [0;255] depending on how you do the math on the Virtual Machine. Eventually you might want to define other data types, such as 32-bit integers (4 bytes). When pushing an i32 onto the stack, you will need to decompose it into its 4 bytes components, and push each one separately. The action of popping the i32 from the stack, is actually 4 different pops. Each byte will then be recomposed by the VM into an i32 so that you can work on it.

Another data type could be 32-bit floating-point numbers (f32), boolean values, strings and so on.

5.8 Storing variables

The stack is also used to store local variables (variables usable on the current scope of the code). This is done by using a stack frame, which is a frame od data that gets pushes onto the stack.
Whenever a function is called (we'll cover functions later, for now we only have 1 big function which is the whole program) a stackframe is allocated. When the functions ends the stackframe is deallocated.

let's execute the collowing code

```
variable = 54
54 + 46
```

When compiling the code the compiler will decide how big the stack frame for each function will be, depending on the number of variables used. In this case we only need 1 variables (1 byte)

```
ALLOC   // Allocate stack frame of 1 byte
1
PSH     // Push 54
54
STORE   // Store 'variable' into the stack frame
-1
PSH     // Push 46
46
LOAD    // Load 'variable' from the stack frame
-2
ADD     // Add
```

STORE -1 This instruction can be decomposed as:

- pop a value from the stack
- store it at an offset (-1)

After the value is popped, the space for our variable within the stack frame is 1 byte before the current point, hence we use -1 as an offset.

When we call **LOAD -2**, the variable is 2 bytes from us since we previously pushes 46 onto the stack. The compiler automatically fills these values in

5.9 If statement

The first step to implement conditions is to define the bool (boolean) type. We only need 1-bit of information to store a boolean value, however we can only allocate a multiple of 8-bits.

We define the bool type with a byte:

$$\begin{aligned} 00000000 &\rightarrow \text{FALSE} \\ \text{else} &\rightarrow \text{TRUE} \end{aligned}$$

We still lack of a fundamental operation in the bytecode: **GOTO**
The **GOTO** operation tells the code to jump to another instruction.

To define the if structure, we could say that if the condition is FALSE, then we need to skip the body of the statement. For simplify sake, I'll define the **GOTO_IF_NOT** instruction, which pops a boolean value and goes to the instruction given by the next value in the bytecode if the bool value is false.
The condition boolean will be pushed on top of the stack before the if execution. It doesn't matter if it is hardcoded on the bytecode, read from the stack frame or is the return type of a function.

```
if (true) {
    PSH
    2
}
PSH
4
```

There are many ways to implement if statements, here's one:

```
PSH          // Push condition
TRUE
GOTO_IT_NOT  // Go to <checkpoint> if value is false
<checkpoint>
PSH          // If body
2
PSH          // Outer code (<checkpoint> instruction)
4
```

5.10 If-Else Statements

For the if-else statement we can just expand the if logic.
If the condition is not satisfied we will jump to the else body instructions.
If the condition is satisfied, we will execute the if body instructions. At the end of the if body we will jump after the else body.

```
-- Condition on top of stack
-- GOTO_IF_NOT <checkpoint1>
-- ... (if body)
-- GOTO <checkpoint2>
-- ... (else body) <checkpoint1>
-- ... (outer program) <checkpoint2>
```

Consider the following program:

```
if (true) {
    PSH
    2
} else {
    PSH
    4
}
PSH 6
ADD
```

The bytecode looks as follows:

```
PSH
TRUE
GOTO_IT_NOT
<checkpoint1>
PSH
2
GOTO
<checkpoint2>
PSH                 (<checkpoin1>)
4
PSH                 (<checkpoin2>)
6
```

5.11 While loop

LOGIC:
Push condition
If the condition is not satisfied, jump to the end of the body
If the condition is satisfied, execute the body. At the end of the body jump to the beginning.

For this example a new instruction is needed: **EQUALS**

Code	**Stack before**	**Stack after**	**Description**
EQUALS	v1, v2	bool	Pops the two topmost values from the stack, compares them and pushes the (boolean) result onto the stack.

```
00 PSH 2
02 PSH 3
04 EQUALS
```

will result in FALSE in top of the stack

I am also going to use the **PRINT** (v1 -> _), which pops a values and "prints" it to some standard output.

Pushing a condition (from the bytecode) will result in either an infinite loop or no iterations at all. We will read and write to a variable in the stackframe, this is the pseudo-code:

```
variable = 10
while (variable / 10 == 2) {
    print(variable)
    variable = variable + 1
}
```

Here's the bytecode

```
ALLOC 1        // Store variable=10
PSH 10
STORE −1

LOAD −2        // Push condition <checkpoint1>
PSH 10
DIV
PSH 1
EQUALS

GOTO_IF_NOT <checkpoint2>

LOAD −1        // While body
PRINT
LOAD −1
PUSH 1
ADD
STORE −1

GOTO <checkpoint1>
 ... (outer program) <checkpoint2>
```

This will produce the following output (since integer division is rounded the loop will stop at *var = 20*):

10 11 12 13 14 15 16 17 18 19

5.12 Functions

The simpliest form of 'function' is a macro: the compiler places the same instructions multiple times throughout the program. This is not good memory-wise. We need a jump-system.

The first problem is that the function must be defined within the program but not executed (unless it has been called) A simple solution could be

```
 ... code

GOTO <checkpoint1>
 ... (function body)

 ... code <checkpoint1>
```

When we want to call the function we can just jump to the beginning of its instruction. However, when the function has ended we must continue doing what we were previously executing. To solve this problem, before jumping to a function, we push the current position. When the function has finished, it pops the index and jumps to it.

```
 ... code

GOTO <checkpoint1>
 ... (function body)      <function>
GOTO <checkpoint2>
```

```
... code <checkpoint1>
GOTO <function>
... code <checkpoint2>
```

5.13 Function with parameters

Before jumping to the function, push the parameters values onto the stack. The function will pop them and use them.

5.14 Function with return value

Before the function finishes, it pushes the return value onto the stack. However, remember that the function must then jump back to the call checkpoint, but the topmost value is the return value rather than the index to jump to. To solve this we could create an instruction **SWAP**, which swaps the last two values on top of the stack. Using the **SWAP** operation, we can then pop the index to jump to. The following instructions will be able to pop the result value from the stack.

5.15 Storing objects and pointers

The solution is pointers and heap memory. The heap memory is another frame of memory (bigger than the stack) used to allocated every objects. Pointers are values (usually 32-bit integers) that point to an object within the heap memory. Instead of storing the *actual* values in the stackframe, we will only store pointers. Those pointers will point to the *actual* values we care about in the heap memory.

As stated at the beginning, Java doesn't use pointers for primitive values. Integers and such are directly stored in the stack frame (since they are as small as a pointer).

To explain how the heap memory works we can use the C language. The C programming language provides the malloc(size) function, which returns a pointer to a chunk of <size> in the heap that you can use. It also provides the free(pointer) method, to free memory from the heap.

```c
void* chunk = malloc(100);
free(chunk);
```

Other languages don't require you to manually free memory:
Java implements a "Garbage Collector", which frees all the unused memory from the heap.
Rust automatically frees unused memory, however this features comes with some extra retrictions.

Helpful resources: https://gameprogrammingpatterns.com/bytecode.html [**gameprogrammingpatterns**].

5.16 Compiler

The life of a program looks as follows:

Source code → compiler → bytecode → VM

However the compiler is not straightforward

$$\textbf{Source code} \rightarrow \overbrace{\boxed{\textbf{Lexer} \rightarrow \textbf{Parser} \rightarrow \textbf{Validator} \rightarrow \textbf{Assembler}}}^{\text{Compiler}} \rightarrow \textbf{Bytecode} \rightarrow \textbf{VM}$$

5.16.1 Lexer

The lexer takes the sources code and produces a list of tokens.

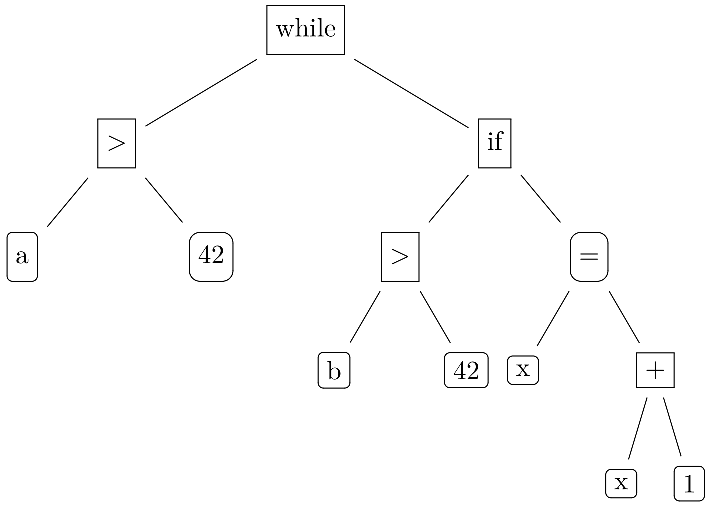
Here's an example of token design.

```
[
    {
        "type": "identifier",
        "value": "function"
    },
    {
        "type": "identifier",
        "value": "if"
    },
    {
        "type": "identifier",
        "value": "while"
    },
    {
        "type": "literal",
        "value": 42
    },
    {
        "type": "operator",
        "value": "{"
    },
    {
        "type": "operator",
        "value": "}"
    },
    {
        "type": "operator",
        "value": ";"
    }
]
```

5.16.2 Parser

The parser takes as input the tokens generated by the lexer. It produces an **AST** (Abstract Syntax Tree).

Here's an example of an abstract syntax tree.



5.16.3 Validator

The validator checks for any error in the AST. It checks if every variable exists, function calls are valid and so on.

5.16.4 Assembler

The assembler takes the AST and produces the final compiled code.

6 Programming Language Implementation

7 Structure

7.1 common

`common/` is a gradle module containing utils and the blockchain protocols.

7.1.1 Protocol

The `protocol/` folder contains all the packets and constants.

7.1.2 Utils

The `utils/` contains various utils.

Byteutils (`byteutils/`)

Utils for serializing data into byte streams.

Crypto (`crypto/`)

Utils for hashing, generating keypairs and signing (EdDSA), base64 conversion.

ParamHandler (`paramhandler/`)

Library for handling CLI parameters and flags.

Stream (`stream/`)

Utils for sending and receiving packets over the network.

7.2 mandate

The `protocol/` folder contains all the documents regarding the project (documentation + diary).

7.3 node-full

7.4 node-api

7.5 node-miner

7.6 piccions

7.7 scripts

7.8 seeder

7.9 website-backend

7.10 website-frontend

7.11 forger

7.11.1 Usage

Generate wallet

```
java -jar forger.jar -gen -out ./key.priv
```

Create transaction

```
java -jar forger.jar -priv ./key.priv -amount 10000
     -out transaction.tx -to <address>
```

Create transaction (no HTTP request for lastHash)

```
java -jar forger.jar -priv ./key.priv -last ./last.tx -amount 10000
     -out transaction.tx -to <address>

java -jar forger.jar -priv ./key.priv -first -amount 10000
     -out transaction.tx -to <address>
```

Dump transaction file content

```
java -jar forger.jar -dump ./transaction.tx
```

Examples of transactions dump

`tx1.tx`:

```
     Amount:          500
     Recipient:       SQAf8paS6Rr5nnd5dC8Bx5h6YKzdAd0Y0g9UMrTa378=
     Sender:          ZZ1PjoBwOZQGP0AYPmwxHsa5Z9YhRDaDbE1XIKyCUsA=
     Signature:       Z0n8TA1zenp+U0ONzSAJLot3JZ5GUedPvViW8vCN8Q6h
                      vmqB81lHsTFg4g7Gb9DFMcnB8KK6TcknAbIAfZTrBg==
     LastHash:        AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=
```

`tx2.tx`:

```
     Amount:          500
     Recipient:       SQAf8paS6Rr5nnd5dC8Bx5h6YKzdAd0Y0g9UMrTa378=
     Sender:          ZZ1PjoBwOZQGP0AYPmwxHsa5Z9YhRDaDbE1XIKyCUsA=
     Signature:       cwMQC5JtBsUMzjEIVNVt/n+zbzm7k//4eRqav+BiYL3P
                      LAqcEaK0QyboXl+FfoV2PpqCZqNHNUzi91Z4CsynAQ==
     LastHash:        MmVCpq21pR/ZLYZwjaMF0AdyRVImq2YZ4MqO9mstnzw=
```

8 Conclusion

Paolo Bettelini

Gianni Grasso

9 References