2CSDE93- Blockchain Technology

LAB 4-B.

In this article, we will understand the working of practical byzantine fault tolerance in Blockchain systems, the math behind this algorithm, its significance, write its pseudo code, and then implement it in node.js

Outline

- What is Fault tolerance and Fault tolerant systems
- What is Practical Byzantine Fault Tolerance
- PBFT In Blockchain, States And Messages
- Implementation of PBFT In Node.Js

- WHAT IS FAULT TOLERANCE AND FAULT TOLERANT SYSTEMS?

Fault tolerance and Fault tolerant systems

- Any system that continues to operate under the influence of unknown or known faults, which results in the reduced capacity of the systems, can be called a fault tolerant system and it exhibits fault tolerance properties.

- Fault tolerant systems, unlike other systems, do not breakdown when a fault occurs, instead, the system operates even in the presence of a fault, but at a reduced throughput or high latency.

Byzantine fault tolerance

- Byzantine faults are particularly present in distributed systems. These faults are an outcome of misinformation among the system nodes. The reasons for these faults/misinformation are mostly unknown to the members of the distributed systems.
- Therefore, in this case, a node may be behaving strangely and sending a different response to different nodes in the network, as a result, it is difficult to classify this node as malicious or faulty.
- Therefore, to make a decision regarding the faulty node, the honest nodes of the system arrive at a consensus and a system that can arrive at a conclusion that is not affected by a malicious/faulty node can be considered as Byzantine fault tolerant system.

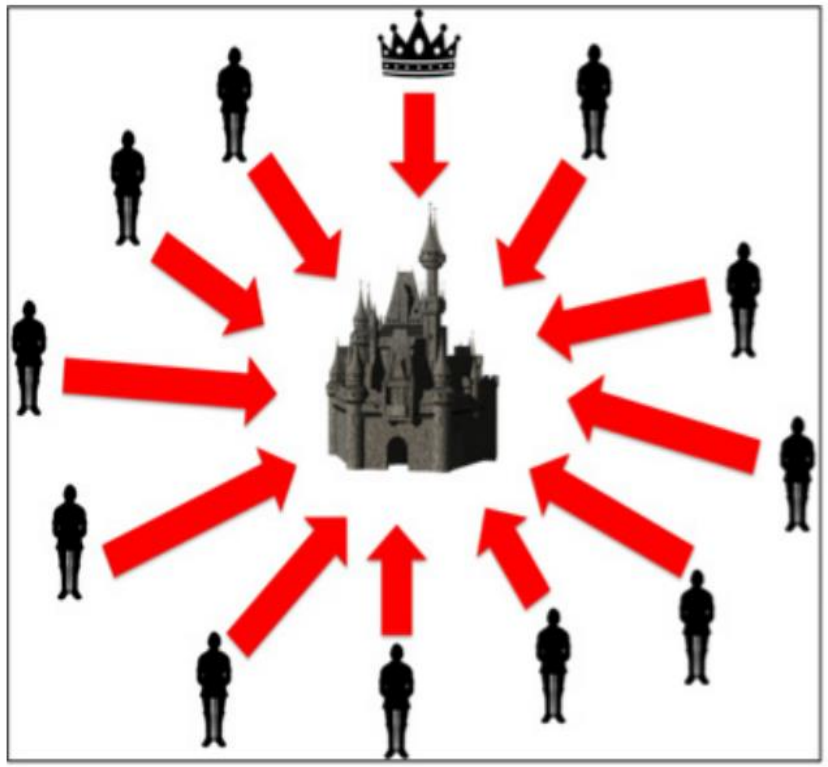
Practical Byzantine Fault Tolerance

- Castro and Liskov developed a novel method to achieve consensus in distributed systems that can tolerate faulty/malicious nodes by replicating nodes/state machines. But PBFT can only tolerate such nodes until the number of faulty nodes is less than one-third of all the nodes.
- Nodes in the network reach consensus about a decision by passing messages among each other about the decision.
- The more the honest nodes, the more secure the system. Since more number of honest nodes will agree on a correct decision than faulty/malicious nodes agreeing on an incorrect decision, false information will be rejected by the majority.

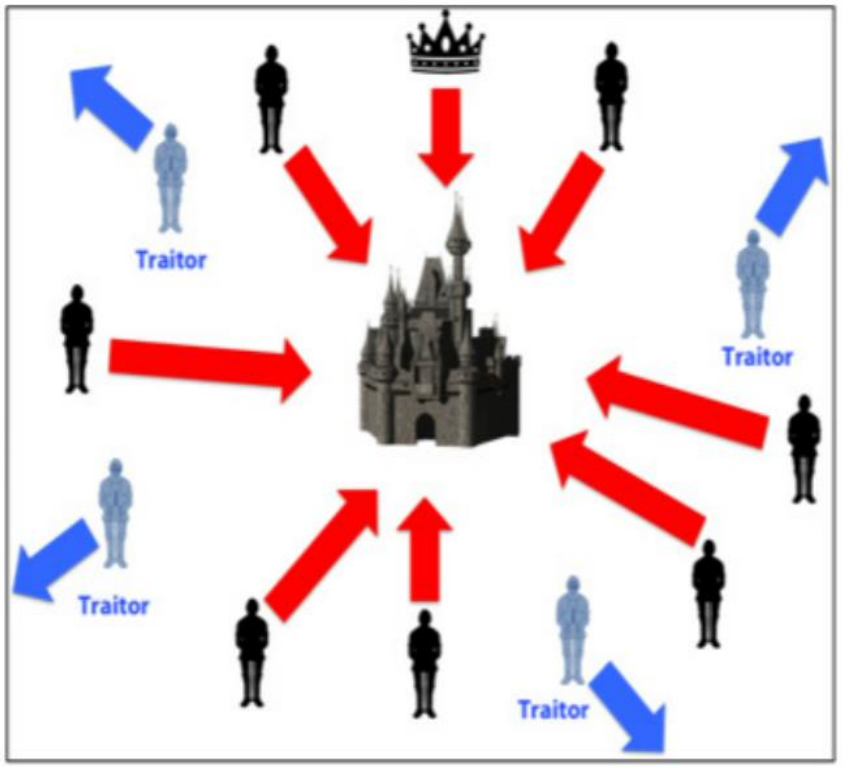
Practical Byzantine Fault Tolerance

- In order to keep the system secure, pbft requires 3f+1 nodes in the system, where f is the maximum number of faulty nodes that the system can tolerate. Therefore, for the group of nodes to make any decision, approval from 2f+1 nodes is required.

Practical Byzantine Fault Tolerance

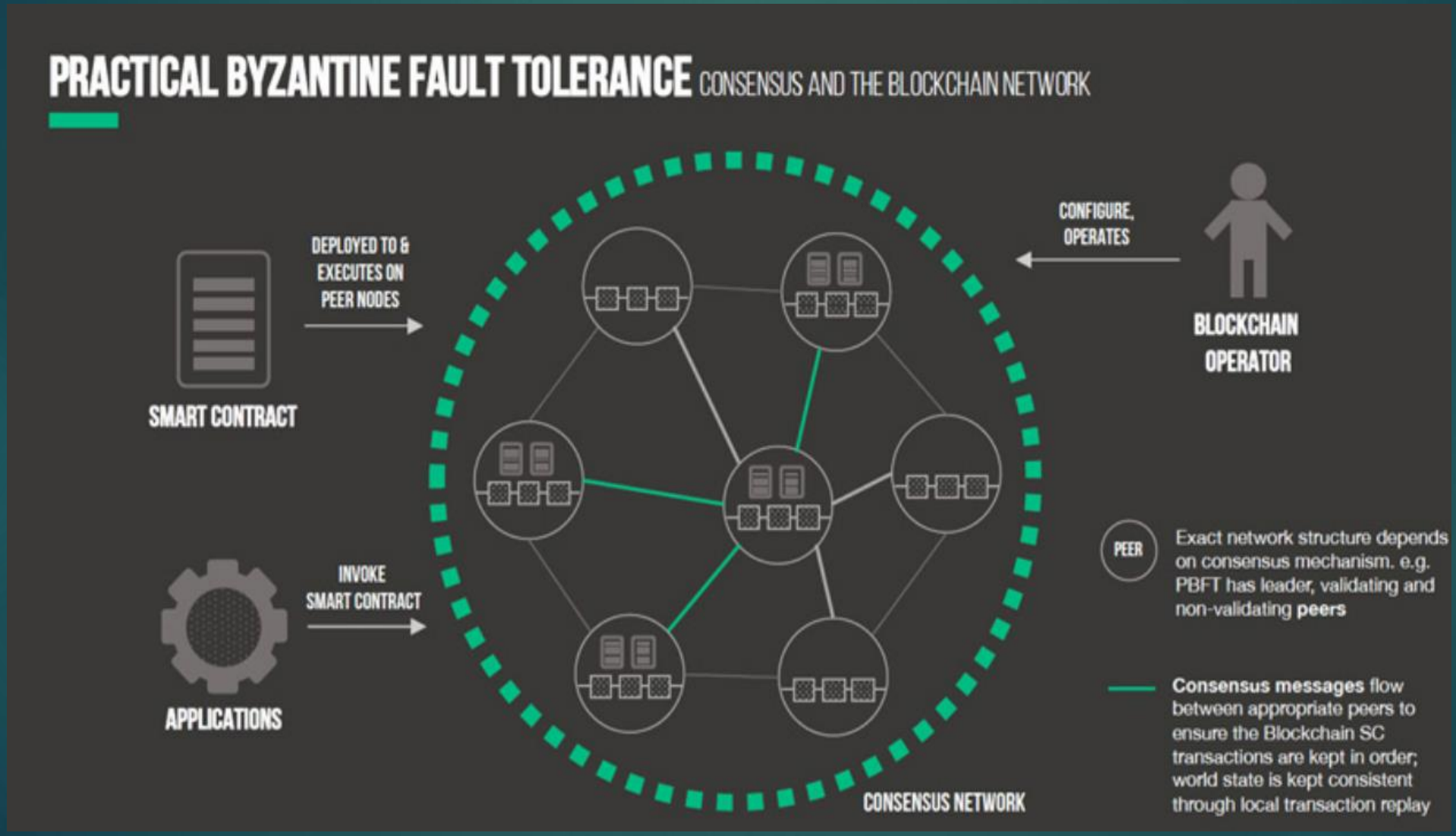


Coordinated Attack Leading to Victory



Uncoordinated Attack Leading to Defeat

Practical Byzantine Fault Tolerance



- PBFT IN BLOCKCHAIN, STATES AND MESSAGES

PBFT in Blockchain

- Nodes in the system share messages among each other to commit a block to the chain. Malicious nodes, in this case, may broadcast tampered blocks, as a result, the block which is considered valid by a maximum number of nodes is considered valid in its entirety by the entire network.

- In bitcoin(proof of work), block proposer is the fastest miner, whereas, in proof of stake, block proposer is the richest miner.

- In PBFT, the block creator may not be any special miner, but the proposed block which is committed to the chain would be the most agreed block. Thereby serving the same purpose that PoW and PoS do, i.e. adding a new block to the chain.

States and Messages

This section described the various states of each node at different sessions and different messages that the nodes pass between each other during any round of block proposal:

- `NEW ROUND`: Proposer to send new block proposal. Validators wait for `PRE-PREPARE` message.
- `PRE-PREPARED`: A validator has received `PRE-PREPARE` message and broadcasts `PREPARE` message. Then it waits for `2F + 1` of `PREPARE` or `COMMIT` messages.
- `PREPARED`: A validator has received `2F + 1` of `PREPARE` messages and broadcasts `COMMIT` messages. Then it waits for `2F + 1` of `COMMIT` messages.
- `COMMITTED`: A validator has received `2F + 1` of `COMMIT` messages and is able to insert the proposed block into the blockchain.
- `FINAL COMMITTED`: A new block is successfully inserted into the blockchain and the validator is ready for the next round.
- `ROUND CHANGE`: A validator is waiting for `2F + 1` of `ROUND CHANGE` messages on the same proposed round number

Algorithm

`NEW ROUND`

- A proposer is elected in a round-robin fashion.
- The proposer collects transactions from the transaction pool.
- Proposer creates a block proposal and broadcasts it to the network. The state of the proposer now changes to `PRE-PERPARED` state.
- Validators receive the `PRE-PREPARE` message and enter the `PRE-PREPARED` state.
- Validators now verify the proposal and then broadcast a `PREPARE` message to the other validators.

`PRE-PREPARED`

- Validators wait for `2F+1` valid `PREPARE` messages, and then enter the `PREPARED` state.
- Validators now broadcast `COMMIT` messages upon entering `PREPAPRED` state.

`PREPARED`

- Validators wait for `2F+1` commit messages and then enter `COMMITTED` state.

`COMMITTED`

- Validators append the `2F+1` commit messages received into the block, and add the block into the blockchain.
- Validators now move a `FINAL COMMITED` state when the block is inserted in the chain

`FINAL COMMITTED`

- A new round is initiated with a new proposer election.

Pseudocode

```
// NEW_ROUND:

State = NEW_ROUND
proposer = get_proposers_address( blockchain )
if ( current_validator == proposer )
    block = create_block( transaction_pool )
    broadcast_block( block )
    State = PRE_PREPARED

// PRE_PREPARED:

ON message.type == PRE_PREPARE
    verify_block( message.block )
    verify_validator( message.block )
    broadcast_prepare( message.block )
    State = PREPARED

// PREPARED:

ON message.type == PREPARE
    verify_prepare( message.prepare )
    verify_validator( message.prepare )
    prepare_pool.add( message.prepare )

if ( prepare_pool.length > 2F+1 )
        broadcast_commit( message.prepare )

    State = COMMITTED

// COMMITTED:
```

Pseudocode

```
ON message.type == COMMIT
    verify_commit( message.commit )
    verify_validator( message.commit )
    commit_pool.add( message.commit )


if ( commit_pool.length > 2F+1 )
        commit_list = commit_pool.get_commits()
        block.append( commit_list )
        blockchain.append( block )

    State = FINAL_COMMITTED


// FINAL_COMMITTED:


new_round()
```

- IMPLEMENTATION OF PBFT IN NODE.JS

PREREQUISITES FOR IMPLEMENTATION

1. Node.js
2. Postman
3. VS code
4. Knowledge about blockchain concepts

Architecture and Design: To implement PBFT we will develop the following components:

1. Wallet class — for public key and signing data.
2. Transaction class — to create transactions and verify them.
3. Block class — to create blocks, verify blocks and verify proposer of the block.
4. Blockchain class — to create a chain, add blocks, calculate proposer, validate blocks, update blocks.
5. P2p Server class — to broadcast and receive data among peers.
6. Validators — to generate and verify validators
7. Transaction pool, block pool, commit pool, prepare pool and message pool — to store transactions, blocks, commits, prepare and new round messages respectively.
8. App — Express API to interact with the blockchain
9. Config — to store global variables
10. Chain Utilities — to store common functions such as hashing and verifying signatures.

Code

Create a root directory `pbft` and cd into it. All the files in this project are present in the root directory.

```
mkdir pbft && cd pbft
```

The ChainUtil Class

We will start off by creating a `chain-util.js` file which will be used multiple times in this project. This file will be used to create key pair for signing data, generating id's for transactions, hashing data and verifying signatures.

We need three modules to perform these functions. Therefore we need to install them.

```
npm i --save elliptic uuid crypto-js
```

Create a class `ChainUtil` and export it.

```
1   // EDDSA allows us to create keypairs
2   // It is collection of cryptographic algorithms that are used to create keypairs
3   const EDDSA = require("elliptic").eddsa;
4
5   // ed25519 allows us to create key pair from secret
6   const eddsa = new EDDSA("ed25519");
7
8   // uuid/v1 creates timestamp dependent ids
9   const uuidV1 = require("uuid/v1");
10
11  // used for hashing data to 256 bits string
12  const SHA256 = require("crypto-js/sha256");
13
14  class ChainUtil {
15    // a static function to return keypair generated using a secret phrase
16    static genKeyPair(secret) {
17      return eddsa.keyFromSecret(secret);
18    }
19
20    // returns ids used in transactions
```

```
21      static id() {
22        return uuidV1();
23      }
24
25      // hashes any data using SHA256
26      static hash(data) {
27        return SHA256(JSON.stringify(data)).toString();
28      }
29
30      // verifies the signed hash by decrypting it with public key
31      // and matching it with the hash
32      static verifySignature(publicKey, signature, dataHash) {
33        return eddsa.keyFromPublic(publicKey).verify(dataHash, signature);
34      }
35    }
36
37    module.exports = ChainUtil;
```

The Transaction Class

Next, we'll make a transaction class. Create file `transaction.js` in the project folder. Transactions in this project will contain the following properties:

1. id — for identification
2. from — the senders address
3. input — an object that further contains data to be stored and timestamp
4. hash — the SHA256 of input
5. signature — the hash signed by the sender

Create a class `Transaction` in the file and export it.

Create a class `Transaction` in the file and export it.

```
1  // Import the ChainUtil class used for hashing and verification
2  const ChainUtil = require("./chain-util");
3
4  class Transaction {
5    // the wallet instance will be passed as a parameter to the constructor
6    // along with the data to be stored.
7    constructor(data, wallet) {
8      this.id = ChainUtil.id();
9      this.from = wallet.publicKey;
10     this.input = { data: data, timestamp: Date.now() };
11     this.hash = ChainUtil.hash(this.input);
12     this.signature = wallet.sign(this.hash);
13   }
14
15   // this method verifies wether the transaction is valid
16   static verifyTransaction(transaction) {
17     return ChainUtil.verifySignature(
18       transaction.from,
19       transaction.signature,
20       ChainUtil.hash(transaction.input)
21     );
22   }
23 }
24
25 module.exports = Transaction;
```

The Wallet Class

Up next is wallet. The wallet holds the public key and key pair. It is also responsible for signing data hashes and creating signed transactions.

Create a file `wallet.js` in the project directory. Add a class `Wallet` and export it.

```
1   // Import the ChainUtil class used for hashing and verification
2   const ChainUtil = require("./chain-util");
3   // Import transaction class  used for creating transactions
4   const Transaction = require("./transaction");
5
6   class Wallet {
7     // The secret phase is passed an argument when creating a wallet
8     // The keypair generated for a secret phrase is always the same
9     constructor(secret) {
10      this.keyPair = ChainUtil.genKeyPair(secret);
11      this.publicKey = this.keyPair.getPublic("hex");
12    }
```

```
14      // Used for prining the wallet details
15      toString() {
16        return `Wallet -
17                publicKey: ${this.publicKey.toString()}`;
18      }
19
20      // Used for signing data hashes
21      sign(dataHash) {
22        return this.keyPair.sign(dataHash).toHex();
23      }
24
25      // Creates and returns transactions
26      createTransaction(data) {
27        return new Transaction(data, this);
28      }
29
30      // Return public key
31      getPublicKey() {
32        return this.publicKey;
33      }
34    }
35
36    module.exports = Wallet;
```

The Validator Class

Since PBFT is a permissioned blockchain consensus algorithm we need to store the address of all the nodes in each nodes system. We can do this manually by choosing a secret, creating a wallet, getting its public key and storing this key into a file and when we run our project it reads this file for keys.

But instead of doing this manually we can automate this by creating a class and adding a function that can return a list of public keys of N number of nodes.

We will create a `Validator` class that will generate a list of public keys known to every node. In this project, we have used the secret phrase for each node as

```
NODE1, NODE2, NODE3.....
```

This way it would be easier for us to make a list of public keys and create the nodes from the command line with the same public key.

```
1   // Import the wallet class
2   const Wallet = require("./wallet");
3
4   class Validators {
5     // constructor will take an argument which is the number of nodes in the network
6     constructor(numberOfValidators) {
7       this.list = this.generateAddresses(numberOfValidators);
8     }
9
10    // This function generates wallets and their public key
11    // The secret key has been known for demonstration purposes
12    // Secret will be passed from command line to generate the same wallet again
13    // As a result the same public key will be generatedd
14    generateAddresses(numberOfValidators) {
15      let list = [];
16      for (let i = 0; i < numberOfValidators; i++) {
17        list.push(new Wallet("NODE" + i).getPublicKey());
18      }
19      return list;
20    }
21
22    // This function verfies if the passed public key is a known validator or not
23    isValidValidator(validator) {
24      return this.list.includes(validator);
25    }
26  }
27  module.exports = Validators;
```

Create a file `config.js` and create three variables `NUMBER_OF_NODES` , `MIN_APPROVALS` and `TRANSACTION_THRESHOLD`

```
1   // Maximum number of transactions that can be present in a block and transaction pool
2   const TRANSACTION_THRESHOLD = 5;
3
4   // total number of nodes in the network
5   const NUMBER_OF_NODES = 3;
6
7   // Minmu number of positive votes required for the message/block to be valid
8   const MIN_APPROVALS = 2 * (NUMBER_OF_NODES / 3) + 1;
9
10  module.exports = {
11    TRANSACTION_THRESHOLD,
12    NUMBER_OF_NODES,
13    MIN_APPROVALS
14  };
```

The Block Class

Next we will create the block class. In the project directory, create a file `block.js` and make a class `Block` in it. Block will have the following properties:

1. timestamp — the time at which the block was made
2. lastHash — hash value of the last block
3. hash — hash value of the current block
4. data — the transactions that the block holds
5. proposer — the public key of the creator of the block
6. signature — the signed hash of the block
7. sequenceNo — the sequence number of block

```
// Import SHA256 used for hashing and ChainUtil for verifying signature
const SHA256 = require("crypto-js/sha256");
const ChainUtil = require("./chain-util");

class Block {
  constructor(
    timestamp,
    lastHash,
    hash,
    data,
    proposer,
    signature,
    sequenceNo
  ) {
    this.timestamp = timestamp;
    this.lastHash = lastHash;
    this.hash = hash;
    this.data = data;
    this.proposer = proposer;
    this.signature = signature;
    this.sequenceNo = sequenceNo;
  }

  // A function to print the block
  toString() {
    return `Block -
        Timestamp    : ${this.timestamp}
        Last Hash    : ${this.lastHash}
        Hash         : ${this.hash}
        Data         : ${this.data}
        proposer     : ${this.proposer}
        Signature    : ${this.signature}
        Sequence No  : ${this.sequenceNo}`;
  }

  // The first block by default will the genesis block
  // this function generates the genesis block with random values
  static genesis() {
    return new this(
      `genesis time`,
      "----",
      "genesis-hash",
      [],
      "P4@P@53R",
```

In the same way, we need to code for the remaining classes

The TransactionPool Class

We need a place to store transactions received from other nodes. Therefore we will make a `TransactionPool` class to store all transactions. Create a file called `transaction-pool.js`

In the same way, we need to code for the remaining classes

The BlockPool Class

To store blocks temporarily we will also make block pool. Create a file `block-pool.js` in which the BlockPool class holds the blocks until it is added to the chain. A block is added to the block pool when a `PRE-PREPARE` message is received.

There are many other data objects received from the nodes that need to be stored. `PREPARE` , `COMMIT` and `NEW_ROUND` messages

Therefore will create three more pools namely, `PreparePool` , `CommitPool` and `MessagePool` . MessagePool will hold the `NEW_ROUND` messages.

In the same way, we need to code for the remaining classes

The PreparePool Class

The CommitPool Class

Commit messages are added after 2f+1 prepare messages are received, therefore we use the prepare message to get the block hash instead of passing the entire block.

The MessagePool Class

MessagePool will work similarly to the other two pools. The only difference would be the extra message it takes with it.

In the same way, we need to code for the remaining classes

The Blockchain Class

We have all of the classes required to make our blockchain class. We can now create a file `blockchain.js` The `Blockchain` class will have the following properties:

1. chain — list of blocks that are confirmed
2. validatorsList — validators list for the given network

In the same way, we need to code for the remaining classes

The P2pserver Class

How do we send messages to other nodes? We will make a P2p Server. Create a class `P2pserver` in a file `p2p-server.js`

To create a p2p server we will make use of sockets. In order to use sockets we will install a 'ws' module. This module makes it utterly easy to work with sockets.

```
npm i --save ws
```

P2pserver class is where the consensus algorithm is implemented. It is the core of the project. This class is responsible for handling messages and broadcasting them.

The App

Now we will connect all the files using an application created using express module. Install the express module using npm.

```
npm i express --save
```

Our application will instantiate the pools, wallet, blockchain, p2pserver and declare some endpoints to interact with our blockchain.

Following are the minimum endpoints required, (you may add more):

1. POST: '/transact' — creates transactions, request object consists of data to be stored in the transaction
2. GET: '/transactions' — sends the transactions in the transaction pool as a response
3. GET: '/blocks' — sends the chain of the blockchain as a response

This completes our coding. You test this application by running the following in the separate terminals:

First node:

```
SECRET="NODE0" P2P_PORT=5000 HTTP_PORT=3000 node app
```

Second node:

```
SECRET="NODE1" P2P_PORT=5001 HTTP_PORT=3001 PEERS=ws://localhost:5000
node app
```

Third node:

```
SECRET="NODE2" P2P_PORT=5002 HTTP_PORT=3002
PEERS=ws://localhost:5001,ws://localhost:5000 node app
```

You can make as many as you want. ***Do update the total number of nodes config file before creating more nodes.***

Hit the endpoint for any node until the pool fills up and check the chain by hitting the `/blocks` endpoint.

Also, hit the endpoint `/transactions` to check if the transaction pool is empty.

You can also make more such endpoints for commit, prepare and message pool.

Thank You