Project Report
On
Fastchain

Submitted by

M.Naveenbhargav (R170732)

S.Ranjith Kumar(R171146)

P.Jagannadham(R170248)

Under the guidance of

A.Mahendra

Assistant professor

Department of Computer Science and Engineering

RAJIV GANDHI UNIVERSITY OF KNOWLEDGE TECHNOLOGIES

RK Valley

Rajiv Gandhi University of Knowledge and Technologies

(RGUKT), R.K.Valley, Kadapa, Andra Pradesh.

Rajiv Gandhi University of Knowledge Technologies

RK Valley, Kadapa (Dist), Andhra Pradesh, 516330

CERTIFICATE

This is to certify that the project work titled “**Fastchain**” is a bonafied project work submitted by M.Naveenbhargav,S.Ranjith Kumar,P.Jagannadham, in the department of COMPUTER SCIENCE AND ENGINEERING in partial fulfillment of requirements for the award of degree of Bachelor of Technology in Computer science and engineering for the year 2021-2022 carried out the work under the supervision.

A.Mahendra
Assistant Professor
Department of CSE
Project Internal Guide
RGUKT R.K. Valley

N.Satyanandaram
Head of the Department
Computer Science and Engineering
RGUKT R.K. Valley

Submitted for the Practical Examination held on……………………….

Internal Examiner

External Examiner

ACKNOWLEDGMENT

The satisfaction that accompanies the successful completion of any task would be incomplete without the mention of the people who made it possible and whose constant guidance and encouragement crown all the efforts success.

I am extremely grateful to our respected Director,Prof. K SANDHYA RANI for fostering an excellent academic climate in our institution.

I also express my sincere gratitude to our respected Head of the Department Mr. N.Satyanandhan for his encouragement, overall guidance in viewing this project a good asset and effort in bringing out this project.

I would like to convey thanks to our guide at college Mr. A Mahendra sir for his guidance, encouragement, co-operation and kindness during the entire duration of the course and academics.

My sincere thanks to all the members who helped me directly and indirectly in the completion of project work. I express my profound gratitude to all our friends and family members for their encouragement.

Index

S. No	INDEX	PAGE NUMBER
1	Abstract	5
2	Introduction	6
3	Purpose	7
4	Scope	7
5	Requirement Specification	8-9
6	Flow Chart	10
7	Methodology	11--15
8	Implementation	16-30
9	Project Output	31
10	Conclusion	32
11	References	33

Abstract

A blockchain is a distributed, decentralized, and immutable digital ledger that records transactions and stores data in a secure and transparent manner. It is composed of a series of blocks, each containing a set of transactions, that are cryptographically linked to the previous block in the chain.

The decentralized nature of blockchain means that there is no single point of failure, and no need for intermediaries such as banks or financial institutions to facilitate transactions. The immutability of blockchain ensures that once a transaction is recorded on the chain, it cannot be altered or deleted.

Introduction

Blockchain is a powerful technology that has the potential to transform many industries, from finance and banking to supply chain management, healthcare, and more. One of the key advantages of blockchain is its ability to create secure, decentralized, and transparent systems that can eliminate the need for intermediaries and increase efficiency.

Golang, also known as Go, is a popular programming language that is well-suited for building blockchain applications. Its simplicity, efficiency, and concurrency features make it ideal for developing distributed systems, such as blockchain networks. Golang also has strong built-in support for cryptography, which is a critical component of blockchain technology.

Our fastchain consists of core features of the typical blockchain like wallet creations, transactions , unspent transactions resolving , persistent blocks storage etc.

Purpose

The purpose of fastchain is to implement the blockchain that is scalable, fast, reliable and maintainable along with core features.

Scope

Fastchain can successfully create and maintain wallets , can able to transact between different addresses , store transactions in persistence database using badgerDB along with that is also have network support for peer to peer connections .

Advantages

- Fast
- Secure
- Horizantally Scaling
- Crypto wallet creation

Disadvantages

- Needs multiple nodes to replicate the data.Energy
- Consumption

Requirement Specification

Hardware Configuration

Client Side:	
Ram	512MB
Hard disk	10GB
Processor	1.0 GHz

Software Requirements

Language	Golang
Frameworks	BadgerDB, Crypto

Golang

- Go, also known as Golang, is an open-source programming language developed by Google in 2007.
- Go is designed to be simple, efficient, and easy to use, with a focus on speed and scalability.
- Go is a statically typed language, meaning that variable types are checked at compile time, making it less prone to errors and bugs.
- Go is particularly well-suited for building distributed systems, such as microservices and blockchain applications, due to its concurrency features and support for network programming.

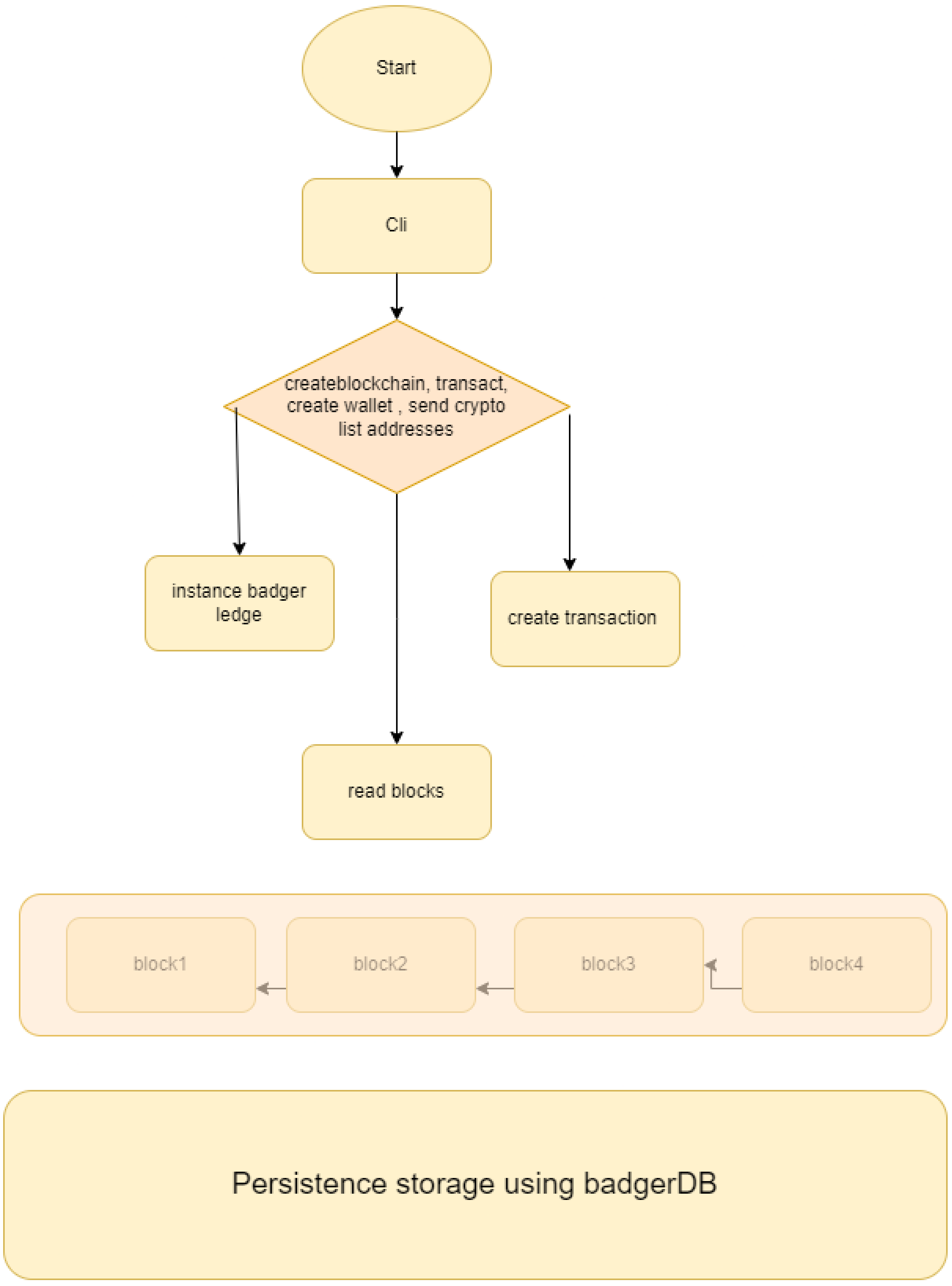
BadgerDB

BadgerDB is an embedded key-value store written in Go. It is designed to be fast, efficient, and easy to use, with a focus on providing a simple API and low-latency performance. BadgerDB is optimized for read-heavy workloads, with support for transactions, compression, and memory-mapped I/O..

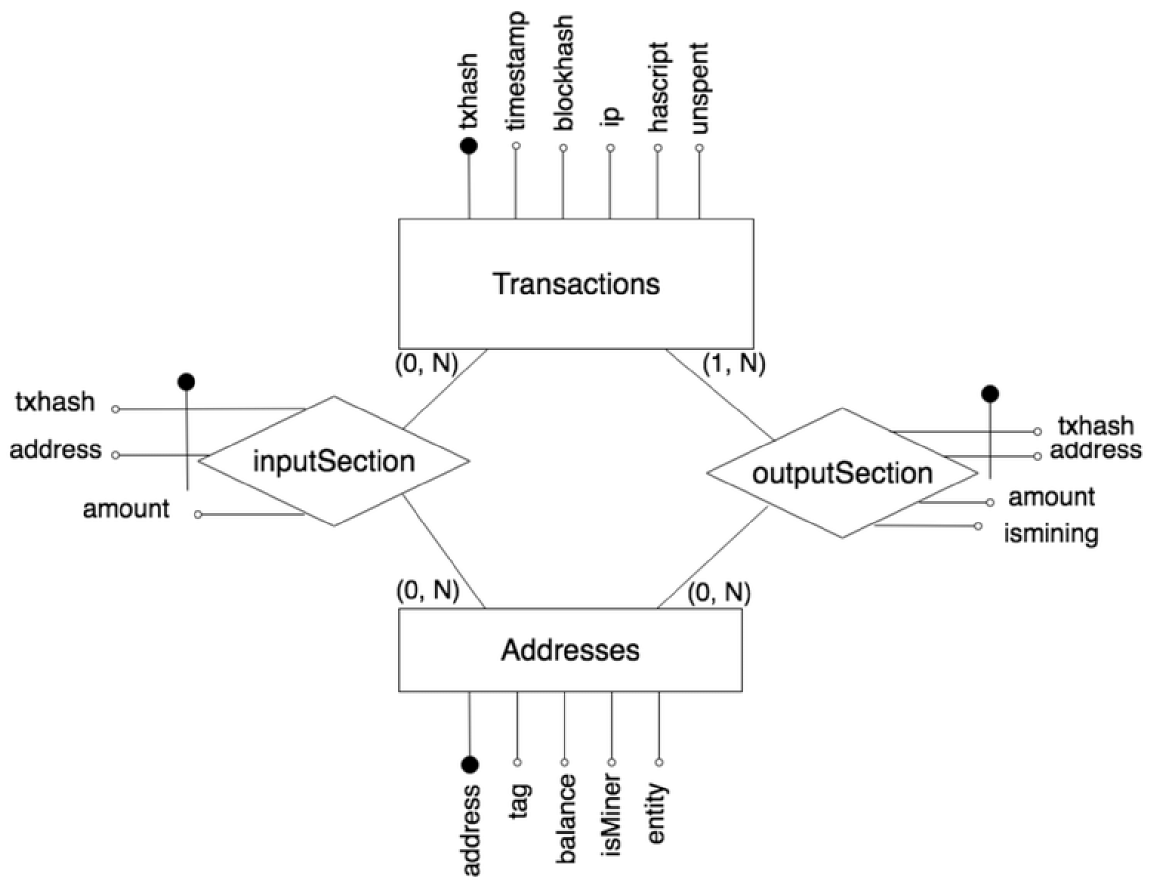
Crypto – Go Framework

Scrytpo" is a standard Go package that provides cryptographic primitives for secure communication, including hash functions, symmetric and asymmetric encryption, digital signatures, and key exchange protocols. It is designed to be fast, efficient, and easy to use, with a simple and consistent API..

Design(Flow Chart)



ER Diagram:


txhash
timestamp
blockhash
ip
hascript
unspent
Transactions
(0, N)
(1, N)
txhash
address
amount
inputSection
outputSection
txhash
address
amount
ismining
(0, N)
(0, N)
Addresses
address
tag
balance
isMiner
entity

Methodology

The architecture of the blockchain implementation consists of the following layers:

- Blockchain Layer: The layer that manages the blockchain structure and handles block creation and validation.
- UTXO Layer: The layer that keeps track of unspent transaction outputs.
- Transaction Layer: The layer that handles transaction creation, signing, and verification.
- Wallet Layer: The layer that manages public and private keys for digital signatures.
- Markle Tree Layer: The layer that creates a hash tree for efficient verification of transactions.
- Network Layer: The layer that enables nodes to communicate and sync the blockchain.

Top down system design

We provide the user with cli to select different options to operate on blockchain like create wallet, create blockchain , list addresses, send value, get balance , reindex utxo,

Start node etc.

They modify or add the transactions to the blocks and add the blocks to the blockchain by providing the proof derivation and store the transactions in the merkle tree in the block using cryptography index hashing etc.

These blocks data will be stored persistently in the badgerDB storage with act as the backbone of the blockchain.

The new instances can be scaled to the new nodes and can be included as the miners and will become the part of our blockchain and used for the mining and adding the blocks to the chain.

Implementation

```go
package cli

import (
    "fastchainv2/blockchain"
    "fastchainv2/network"
    "fastchainv2/wallet"
    "flag"
    "fmt"
    "log"
    "os"
    "runtime"
    "strconv"
)

// Responsible for processing command line arguments
type CommandLine struct {
}

func (cli *CommandLine) printUsage() {
    fmt.Println("Usage:")
    fmt.Println(" getbalance -address ADDRESS - get the balance for an address")
    fmt.Println(" createblockchain -address ADDRESS creates a blockchain and sends
genesis reward to address")
    fmt.Println(" printchain - Prints the blocks in the chain")
    fmt.Println(" send -from FROM -to TO -amount AMOUNT -mine - Send amount of
coins. Then -mine flag is set, mine off of this node")
    fmt.Println(" createwallet - Creates a new Wallet")
    fmt.Println(" listaddresses - Lists the addresses in our wallet file")
    fmt.Println(" reindexutxo - Rebuilds the UTXO set")
    fmt.Println(" startnode -miner ADDRESS - Start a node with ID specified in
NODE_ID env. var. -miner enables mining")
}

func (cli *CommandLine) validateArgs() {
    if len(os.Args) < 2 {
        cli.printUsage()
        runtime.Goexit()
    }
}

// Start node. If has miner address, start as miner
func (cli *CommandLine) StartNode(nodeID, minerAddress string) {
    fmt.Printf("Starting Node %s\n", nodeID)
    if len(minerAddress) > 0 {
        if wallet.ValidateAddress(minerAddress) {
            fmt.Println("Mining is on. Address to receive rewards: ",
minerAddress)
        } else {
            log.Panic("Wrong miner address!")
        }
    }

    network.StartServer(nodeID, minerAddress)
}

func (cli *CommandLine) reindexUTXO(nodeID string) {
    chain := blockchain.ContinueBlockChain(nodeID)
    defer func() {
        err := chain.Database.DB.Close()
        if err != nil {
            log.Panic(err)
        }
    }()

    UTXOSet := blockchain.UTXOSet{chain}
    UTXOSet.Reindex()

    count := UTXOSet.CountTransactions()
    fmt.Printf("Done! There are %d transactions in the UTXO set.\n", count)
}

func (cli *CommandLine) listAddresses(nodeID string) {
    wallets, _ := wallet.CreateWallets(nodeID)
    addresses := wallets.GetAllAddresses()

    for _, address := range addresses {
        fmt.Println(address)
    }
}

func (cli *CommandLine) createWallet(nodeID string) {
    wallets, _ := wallet.CreateWallets(nodeID)
    address := wallets.AddWallet()
    wallets.SaveFile(nodeID)

    fmt.Printf("New address is: %s\n", address)
}

func (cli *CommandLine) printChain(nodeID string) {
    chain := blockchain.ContinueBlockChain(nodeID)
    defer func() {
        err := chain.Database.DB.Close()
        if err != nil {
            log.Panic(err)
        }
    }()

    iter := chain.Iterator()

    for {
        block := iter.Next()

        fmt.Printf("Hash: %x\n", block.Hash)
        fmt.Printf("Prev. hash: %x\n", block.PrevHash)
        pow := blockchain.NewProof(block)
        fmt.Printf("PoW: %s\n", strconv.FormatBool(pow.Validate()))
        for _, tx := range block.Transactions {
            fmt.Println(tx)
        }
        fmt.Println()

        if len(block.PrevHash) == 0 {
            break
        }
    }
}

func (cli *CommandLine) createBlockChain(address, nodeID string) {
    if !wallet.ValidateAddress(address) {
        log.Panic("Address is not Valid")
    }

    chain := blockchain.InitBlockChain(address, nodeID)
    defer func() {
        err := chain.Database.DB.Close()
        if err != nil {
            log.Panic(err)
        }
    }()

    UTXOSet := blockchain.UTXOSet{chain}
    UTXOSet.Reindex()

    fmt.Println("Finished!")
}

func (cli *CommandLine) getBalance(address, nodeID string) {
    if !wallet.ValidateAddress(address) {
        log.Panic("Address is not Valid")
    }

    chain := blockchain.ContinueBlockChain(nodeID)
    UTXOSet := blockchain.UTXOSet{chain}
    defer func() {
        err := chain.Database.DB.Close()
        if err != nil {
            log.Panic(err)
        }
    }()

    balance := 0
    pubKeyHash := wallet.Base58Decode([]byte(address))
    pubKeyHash = pubKeyHash[1 : len(pubKeyHash)-4]
    UTXOs := UTXOSet.FindUnspentTransactions(pubKeyHash)

    for _, out := range UTXOs {
        balance += out.Value
    }

    fmt.Printf("Balance of %s: %d\n", address, balance)
}

func (cli *CommandLine) send(from, to string, amount int, nodeID string, mineNow
bool) {
    if !wallet.ValidateAddress(to) {
        log.Panic("Address is not Valid")
    }

    if !wallet.ValidateAddress(from) {
        log.Panic("Address is not Valid")
    }

    chain := blockchain.ContinueBlockChain(nodeID)

    UTXOSet := blockchain.UTXOSet{chain}
    defer func() {
        err := chain.Database.DB.Close()
        if err != nil {
            log.Panic(err)
        }
    }()

    wallets, err := wallet.CreateWallets(nodeID)
    if err != nil {
        log.Panic(err)
    }
    wal := wallets.GetWallet(from)

    tx := blockchain.NewTransaction(&wal, to, amount, &UTXOSet)

    if mineNow {
        cbTx := blockchain.CoinbaseTx(from, "")
        txs := []*blockchain.Transaction{cbTx, tx}
        block := chain.MineBlock(txs)
        UTXOSet.Update(block)
    } else {
        network.SendTx(network.KnownNodes[0], tx)
    }

    fmt.Println("Success!")
}

// Parse command line arguments and processes commands
func (cli *CommandLine) Run() {
    cli.validateArgs()

    nodeID := os.Getenv("NODE_ID")
    if nodeID == "" {
        fmt.Printf("NODE_ID env is not set!")
        runtime.Goexit()
    }

    getBalanceCmd := flag.NewFlagSet("getbalance", flag.ExitOnError)
    createBlockchainCmd := flag.NewFlagSet("createblockchain", flag.ExitOnError)
    sendCmd := flag.NewFlagSet("send", flag.ExitOnError)
    printChainCmd := flag.NewFlagSet("printchain", flag.ExitOnError)
    createWalletCmd := flag.NewFlagSet("createwallet", flag.ExitOnError)
    listAddressesCmd := flag.NewFlagSet("listaddresses", flag.ExitOnError)
    reindexUTXOCmd := flag.NewFlagSet("reindexutxo", flag.ExitOnError)
    startNodeCmd := flag.NewFlagSet("startnode", flag.ExitOnError)

    getBalanceAddress := getBalanceCmd.String("address", "", "The address to get
balance for")
    createBlockchainAddress := createBlockchainCmd.String("address", "", "The
address to send genesis block reward to")
    sendFrom := sendCmd.String("from", "", "Source wallet address")
    sendTo := sendCmd.String("to", "", "Destination wallet address")
    sendAmount := sendCmd.Int("amount", 0, "Amount to send")
    sendMine := sendCmd.Bool("mine", false, "Mine immediately on the same node")
    startNodeMiner := startNodeCmd.String("miner", "", "Enable mining mode and
send reward to ADDRESS")

    switch os.Args[1] {
    case "reindexutxo":
```

```go
		err := reindexUTXOCmd.Parse(os.Args[2:])
		if err != nil {
			log.Panic(err)
		}
	case "getbalance":
		err := getBalanceCmd.Parse(os.Args[2:])
		if err != nil {
			log.Panic(err)
		}
	case "createblockchain":
		err := createBlockchainCmd.Parse(os.Args[2:])
		if err != nil {
			log.Panic(err)
		}
	case "startnode":
		err := startNodeCmd.Parse(os.Args[2:])
		if err != nil {
			log.Panic(err)
		}
	case "listaddresses":
		err := listAddressesCmd.Parse(os.Args[2:])
		if err != nil {
			log.Panic(err)
		}
	case "createwallet":
		err := createWalletCmd.Parse(os.Args[2:])
		if err != nil {
			log.Panic(err)
		}
	case "printchain":
		err := printChainCmd.Parse(os.Args[2:])
		if err != nil {
			log.Panic(err)
		}
	case "send":
		err := sendCmd.Parse(os.Args[2:])
		if err != nil {
			log.Panic(err)
		}
	default:
		cli.printUsage()
		runtime.Goexit()
	}

	if getBalanceCmd.Parsed() {
		if *getBalanceAddress == "" {
			getBalanceCmd.Usage()
			runtime.Goexit()
```

```
        }
        cli.getBalance(*getBalanceAddress, nodeID)
    }

    if createBlockchainCmd.Parsed() {
        if *createBlockchainAddress == "" {
            createBlockchainCmd.Usage()
            runtime.Goexit()
        }
        cli.createBlockChain(*createBlockchainAddress, nodeID)
    }

    if printChainCmd.Parsed() {
        cli.printChain(nodeID)
    }

    if createWalletCmd.Parsed() {
        cli.createWallet(nodeID)
    }
    if listAddressesCmd.Parsed() {
        cli.listAddresses(nodeID)
    }
    if reindexUTXOCmd.Parsed() {
        cli.reindexUTXO(nodeID)
    }

    if sendCmd.Parsed() {
        if *sendFrom == "" || *sendTo == "" || *sendAmount <= 0 {
            sendCmd.Usage()
            runtime.Goexit()
        }

        cli.send(*sendFrom, *sendTo, *sendAmount, nodeID, *sendMine)
    }

    if startNodeCmd.Parsed() {
        nodeID := os.Getenv("NODE_ID")
        if nodeID == "" {
            startNodeCmd.Usage()
            runtime.Goexit()
        }
        cli.StartNode(nodeID, *startNodeMiner)
    }
}



package blockchain
```

```go
import (
    "bytes"
    "encoding/gob"
    "log"
    "time"
)

type Block struct {
    Timestamp    int64
    Hash         []byte
    Transactions []*Transaction
    PrevHash     []byte
    Nonce        int
    Height       int
}

func (b *Block) HashTransactions() []byte {
    var txHashes [][]byte

    for _, tx := range b.Transactions {
        txHashes = append(txHashes, tx.Serialize())
    }
    tree := NewMerkleTree(txHashes)

    return tree.RootNode.Data
}

func CreateBlock(txs []*Transaction, prevHash []byte, height int) *Block {
    block := &Block{time.Now().Unix(), []byte{}, txs, prevHash, 0, height}
    pow := NewProof(block)
    nonce, hash := pow.Run()

    block.Hash = hash[:]
    block.Nonce = nonce

    return block
}

func Genesis(coinbase *Transaction) *Block {
    return CreateBlock([]*Transaction{coinbase}, []byte{}, 0)
}

func (b *Block) Serialize() []byte {
    var res bytes.Buffer
    encoder := gob.NewEncoder(&res)

    err := encoder.Encode(b)
    Handle(err)

    return res.Bytes()
}

func Deserialize(data []byte) *Block {
    var block Block

    decoder := gob.NewDecoder(bytes.NewReader(data))

    err := decoder.Decode(&block)

    Handle(err)

    return &block
}

func Handle(err error) {
    if err != nil {
        log.Panic(err)
    }
}
```

```go
package blockchain

import (
    "bytes"
    "crypto/ecdsa"
    "encoding/hex"
    "errors"
    "fmt"
    "log"

    "fastchainv2/database"

    "os"
    "runtime"
)

const (
    dbPath      = "/tmp/blocks_%s"
    genesisData = "First Transaction from Genesis"
)

type BlockChain struct {
    LastHash []byte
    Database *database.Database
}

type ChainIterator struct {
    CurrentHash []byte
    Database    *database.Database
}

func DBExists(path string) bool {
    if _, err := os.Stat(path + "/MANIFEST"); os.IsNotExist(err) {
        return false
    }

    return true
}

func InitBlockChain(address, nodeId string) *BlockChain {
    path := fmt.Sprintf(dbPath, nodeId)

    if DBExists(path) {
        fmt.Println("Blockchain already exists.")
        runtime.Goexit()
    }

    // Get database instance
    db, err := database.GetDatabase(path)
    Handle(err)

    // Create coinbase transaction
    cbtx := CoinbaseTx(address, genesisData)

    // Create genesis block
    genesis := Genesis(cbtx)
    fmt.Println("Genesis created")

    // Store genesis block data
    err = db.Update(genesis.Hash, genesis.Serialize())
    Handle(err)

    // Set last hash as genesis block hash
    err = db.Update([]byte("lh"), genesis.Hash)
    Handle(err)

    // Return chain that has only genesis block
    blockchain := BlockChain{genesis.Hash, db}
    return &blockchain
}
```

```go
func ContinueBlockChain(nodeId string) *BlockChain {
    path := fmt.Sprintf(dbPath, nodeId)

    if DBExists(path) == false {
        fmt.Println("No existing blockchain found, create one!")
        runtime.Goexit()
    }

    db, err := database.GetDatabase(path)
    Handle(err)

    // Get last block hash
    lastHash, err := db.Read([]byte("lh"))
    Handle(err)

    chain := BlockChain{lastHash, db}
    return &chain
}

func (chain *BlockChain) MineBlock(transactions []*Transaction) *Block {
    for _, tx := range transactions {
        if chain.VerifyTransaction(tx) != true {
            log.Panic("Invalid Transaction")
        }
    }

    // Get last block hash
    lastHash, err := chain.Database.Read([]byte("lh"))
    Handle(err)

    // Get serialized last block data
    lastBlockBytes, err := chain.Database.Read(lastHash)
    Handle(err)

    // Deserialize byte data to block
    lastBlock := *Deserialize(lastBlockBytes)

    // Create new block
    newBlock := CreateBlock(transactions, lastHash, lastBlock.Height+1)

    // Store new block
    err = chain.Database.Update(newBlock.Hash, newBlock.Serialize())
    Handle(err)

    // Update last block hash
    err = chain.Database.Update([]byte("lh"), newBlock.Hash)
    Handle(err)
    chain.LastHash = newBlock.Hash

    return newBlock
}

func (chain *BlockChain) AddBlock(block *Block) {

    // If the chain already has this block, cancel the process
    _, err := chain.Database.Read(block.Hash)
    if err == nil {
        return
    }

    // Store new block
    err = chain.Database.Update(block.Hash, block.Serialize())
    Handle(err)

    // Get last block hash
    lastHash, err := chain.Database.Read([]byte("lh"))
    Handle(err)

    // Get serialized last block data
    lastBlockData, err := chain.Database.Read(lastHash)
    Handle(err)

    // Deserialize byte data to block
    lastBlock := Deserialize(lastBlockData)

    // If block height is bigger than last block height, set it as the last block
    if block.Height > lastBlock.Height {
        err := chain.Database.Update([]byte("lh"), block.Hash)
        Handle(err)

        chain.LastHash = block.Hash
    }
}

func (chain *BlockChain) GetBlockHashes() [][]byte {
    var blocks [][]byte

    iter := chain.Iterator()

    for {
        block := iter.Next()
        blocks = append(blocks, block.Hash)

        // Iterate until the first block
        if len(block.PrevHash) == 0 {
            break
        }
    }

    return blocks
}

func (chain *BlockChain) GetBlock(blockHash []byte) (Block, error) {
    blockData, err := chain.Database.Read(blockHash)

    if err != nil {
        log.Panic("block is not found")
    }

    return *Deserialize(blockData), err
}

func (chain *BlockChain) GetBestHeight() int {
    lastHash, err := chain.Database.Read([]byte("lh"))
    Handle(err)

    lastBlockData, err := chain.Database.Read(lastHash)
    Handle(err)

    lastBlock := *Deserialize(lastBlockData)

    return lastBlock.Height
}

func (chain *BlockChain) Iterator() *ChainIterator {
    return &ChainIterator{chain.LastHash, chain.Database}
}

func (iter *ChainIterator) Next() *Block {
    currentBlockData, err := iter.Database.Read(iter.CurrentHash)
    Handle(err)

    block := Deserialize(currentBlockData)
    iter.CurrentHash = block.PrevHash

    return block
}

// Finds unspent transaction outputs
// Unspent means that these outputs were not referenced in any inputs
func (chain *BlockChain) FindUTXO() map[string]TxOutputs {
    UTXO := make(map[string]TxOutputs)
    spentTXOs := make(map[string][]int)

    iter := chain.Iterator()

    for {
        block := iter.Next()

        // Iterate transactions
        for _, tx := range block.Transactions {
            txID := hex.EncodeToString(tx.ID)

        Outputs:
            // Iterate transaction outputs
            for outIdx, out := range tx.Outputs {
                if spentTXOs[txID] != nil {
                    for _, spentOut := range spentTXOs[txID] {
                        if spentOut == outIdx {
                            continue Outputs
                        }
                    }
                }

                outs := UTXO[txID]
                outs.Outputs = append(outs.Outputs, out)
                UTXO[txID] = outs
            }

            if tx.IsCoinbase() == false {
                for _, in := range tx.Inputs {
                    inTxID := hex.EncodeToString(in.ID)
                    spentTXOs[inTxID] = append(spentTXOs[inTxID], in.Out)
                }
            }
        }

        if len(block.PrevHash) == 0 {
            break
        }
    }
    return UTXO
}

func (chain *BlockChain) FindUnspentTransactions(pubKeyHash []byte) []Transaction
{
    var unspentTxs []Transaction

    spentTXOs := make(map[string][]int)
```

```go
    iter := chain.Iterator()

    for {
        block := iter.Next()

        // Iterate transactions
        for _, tx := range block.Transactions {
            txID := hex.EncodeToString(tx.ID)

        Outputs:
            // Iterate transaction outputs
            for outIdx, out := range tx.Outputs {
                if spentTXOs[txID] != nil {
                    for _, spentOut := range spentTXOs[txID] {
                        if spentOut == outIdx {
                            continue Outputs
                        }
                    }
                }

                if out.IsLockedWithKey(pubKeyHash) {
                    unspentTxs = append(unspentTxs, *tx)
                }
            }

            if tx.IsCoinbase() == false {
                for _, in := range tx.Inputs {
                    if in.UsesKey(pubKeyHash) {
                        inTxID := hex.EncodeToString(in.ID)
                        spentTXOs[inTxID] = append(spentTXOs[inTxID], in.Out)
                    }
                }
            }
        }

        if len(block.PrevHash) == 0 {
            break
        }
    }
    return unspentTxs
}

func (chain *BlockChain) FindSpendableOutputs(pubKeyHash []byte, amount int) (int,
map[string][]int) {
    unspentOuts := make(map[string][]int)
    unspentTxs := chain.FindUnspentTransactions(pubKeyHash)
    accumulated := 0
```

```
Work:
    for _, tx := range unspentTxs {
        txID := hex.EncodeToString(tx.ID)

        for outIdx, out := range tx.Outputs {
            if out.IsLockedWithKey(pubKeyHash) && accumulated < amount {
                accumulated += out.Value
                unspentOuts[txID] = append(unspentOuts[txID], outIdx)

                if accumulated >= amount {
                    break Work
                }
            }
        }
    }

    return accumulated, unspentOuts
}

func (chain *BlockChain) FindTransaction(ID []byte) (Transaction, error) {
    iter := chain.Iterator()

    for {
        block := iter.Next()

        for _, tx := range block.Transactions {
            if bytes.Compare(tx.ID, ID) == 0 {
                return *tx, nil
            }
        }

        if len(block.PrevHash) == 0 {
            break
        }
    }

    return Transaction{}, errors.New("transaction does not exist")
}

func (chain *BlockChain) SignTransaction(tx *Transaction, privateKey
ecdsa.PrivateKey) {
    prevTXs := make(map[string]Transaction)

    // Iterate previous transactions
    for _, in := range tx.Inputs {
        prevTX, err := chain.FindTransaction(in.ID)
        Handle(err)
        prevTXs[hex.EncodeToString(prevTX.ID)] = prevTX
    }

    tx.Sign(privateKey, prevTXs)
}

func (chain *BlockChain) VerifyTransaction(tx *Transaction) bool {

    if tx.IsCoinbase() {
        return true
    }

    prevTXs := make(map[string]Transaction)

    for _, in := range tx.Inputs {
        fmt.Println("TX: " + hex.EncodeToString(tx.ID) + "\nInput Prev: " +
hex.EncodeToString(in.ID))

        prevTX, err := chain.FindTransaction(in.ID)
        Handle(err)

        fmt.Println("Tx in DB: " + hex.EncodeToString(prevTX.ID))

        prevTXs[hex.EncodeToString(prevTX.ID)] = prevTX
    }

    return tx.Verify(prevTXs)
}

package blockchain

import (
    "bytes"
    "crypto/ecdsa"
    "crypto/elliptic"
    "crypto/rand"
    "crypto/sha256"
    "encoding/gob"
    "encoding/hex"
    "fastchainv2/wallet"
    "fmt"
    "io"
    "log"
    "math/big"
    "strings"
)

type Transaction struct {
    ID      []byte
    Inputs  []TxInput
    Outputs []TxOutput
}

func (tx *Transaction) Hash() []byte {
    var hash [32]byte

    txCopy := *tx
    txCopy.ID = []byte{}

    salt := make([]byte, 32)
    _, err := io.ReadFull(rand.Reader, salt)
    if err != nil {
        fmt.Printf("ERROR: Creating salt failed: %s\n", err)
    }

    data := txCopy.Serialize()
    data = append(data, salt...)

    // Perform sha256 on serialized transaction
    hash = sha256.Sum256(data)

    return hash[:]
}

func (tx Transaction) Serialize() []byte {
    var encoded bytes.Buffer

    enc := gob.NewEncoder(&encoded)
    err := enc.Encode(tx)
    if err != nil {
        log.Panic(err)
    }

    return encoded.Bytes()
}

func DeserializeTransaction(data []byte) Transaction {
    var transaction Transaction

    decoder := gob.NewDecoder(bytes.NewReader(data))
    err := decoder.Decode(&transaction)
    Handle(err)

    return transaction
}

func CoinbaseTx(to, data string) *Transaction {
```

```go
    if data == "" {
        randData := make([]byte, 24)
        _, err := rand.Read(randData)
        Handle(err)
        data = fmt.Sprintf("%x", randData)
    }

    txin := TxInput{[]byte{}, -1, nil, []byte(data)}
    txout := NewTXOutput(20, to)

    tx := Transaction{nil, []TxInput{txin}, []TxOutput{*txout}}
    tx.ID = tx.Hash()

    return &tx
}

func NewTransaction(w *wallet.Wallet, to string, amount int, UTXO *UTXOSet)
*Transaction {
    var inputs []TxInput
    var outputs []TxOutput

    pubKeyHash := wallet.PublicKeyHash(w.PublicKey)
    acc, validOutputs := UTXO.FindSpendableOutputs(pubKeyHash, amount)

    if acc < amount {
        log.Panic("Error: not enough funds")
    }

    for encodedTxID, outs := range validOutputs {
        txID, err := hex.DecodeString(encodedTxID)
        Handle(err)

        for _, out := range outs {
            input := TxInput{txID, out, nil, w.PublicKey}
            inputs = append(inputs, input)
        }
    }

    from := fmt.Sprintf("%s", w.Address())

    outputs = append(outputs, *NewTXOutput(amount, to))

    if acc > amount {
        outputs = append(outputs, *NewTXOutput(acc-amount, from))
    }

    tx := Transaction{nil, inputs, outputs}
    tx.ID = tx.Hash()
    UTXO.BlockChain.SignTransaction(&tx, w.PrivateKey)

    return &tx
}

func (tx *Transaction) IsCoinbase() bool {
    return len(tx.Inputs) == 1 && len(tx.Inputs[0].ID) == 0 && tx.Inputs[0].Out ==
-1
}

func (tx *Transaction) Sign(privateKey ecdsa.PrivateKey, prevTXs
map[string]Transaction) {
    if tx.IsCoinbase() {
        return
    }

    for _, in := range tx.Inputs {
        if prevTXs[hex.EncodeToString(in.ID)].ID == nil {
            log.Panic("ERROR: Previous transaction is not correct")
        }
    }

    txCopy := tx.TrimmedCopy()

    for inId, in := range txCopy.Inputs {
        prevTX := prevTXs[hex.EncodeToString(in.ID)]
        txCopy.Inputs[inId].Signature = nil
        txCopy.Inputs[inId].PubKey = prevTX.Outputs[in.Out].PubKeyHash

        dataToSign := fmt.Sprintf("%x\n", txCopy)

        r, s, err := ecdsa.Sign(rand.Reader, &privateKey, []byte(dataToSign))
        Handle(err)
        signature := append(r.Bytes(), s.Bytes()...)

        tx.Inputs[inId].Signature = signature
        txCopy.Inputs[inId].PubKey = nil
    }
}

func (tx *Transaction) Verify(prevTXs map[string]Transaction) bool {
    if tx.IsCoinbase() {
        return true
    }

    for _, in := range tx.Inputs {
        if prevTXs[hex.EncodeToString(in.ID)].ID == nil {
            log.Panic("Previous transaction not correct")
        }
    }

    txCopy := tx.TrimmedCopy()
    curve := elliptic.P256()

    for inId, in := range tx.Inputs {
        prevTx := prevTXs[hex.EncodeToString(in.ID)]
        txCopy.Inputs[inId].Signature = nil
        txCopy.Inputs[inId].PubKey = prevTx.Outputs[in.Out].PubKeyHash

        r := big.Int{}
        s := big.Int{}

        sigLen := len(in.Signature)
        r.SetBytes(in.Signature[:(sigLen / 2)])
        s.SetBytes(in.Signature[(sigLen / 2):])

        x := big.Int{}
        y := big.Int{}
        keyLen := len(in.PubKey)
        x.SetBytes(in.PubKey[:(keyLen / 2)])
        y.SetBytes(in.PubKey[(keyLen / 2):])

        dataToVerify := fmt.Sprintf("%x\n", txCopy)

        rawPubKey := ecdsa.PublicKey{Curve: curve, X: &x, Y: &y}
        if ecdsa.Verify(&rawPubKey, []byte(dataToVerify), &r, &s) == false {
            return false
        }
        txCopy.Inputs[inId].PubKey = nil
    }

    return true
}

func (tx *Transaction) TrimmedCopy() Transaction {
    var inputs []TxInput
    var outputs []TxOutput

    for _, in := range tx.Inputs {
        inputs = append(inputs, TxInput{in.ID, in.Out, nil, nil})
    }

    for _, out := range tx.Outputs {
        outputs = append(outputs, TxOutput{out.Value, out.PubKeyHash})
    }
    txCopy := Transaction{tx.ID, inputs, outputs}

    return txCopy
}

func (tx Transaction) String() string {
    var lines []string

    lines = append(lines, fmt.Sprintf("--- Transaction %x:", tx.ID))
    for i, input := range tx.Inputs {
        lines = append(lines, fmt.Sprintf("     Input %d:", i))
        lines = append(lines, fmt.Sprintf("       TXID:     %x", input.ID))
        lines = append(lines, fmt.Sprintf("       Out:       %d", input.Out))
        lines = append(lines, fmt.Sprintf("       Signature: %x",
input.Signature))
        lines = append(lines, fmt.Sprintf("       PubKey:    %x", input.PubKey))
    }

    for i, output := range tx.Outputs {
        lines = append(lines, fmt.Sprintf("     Output %d:", i))
        lines = append(lines, fmt.Sprintf("       Value:  %d", output.Value))
        lines = append(lines, fmt.Sprintf("       Script: %x", output.PubKeyHash))
    }

    return strings.Join(lines, "\n")
}

package database

import (
    "fmt"
    "github.com/dgraph-io/badger"
    "log"
    "os"
    "path/filepath"
    "strings"
)

type Database struct {
    DB *badger.DB
}

func GetDatabase(dir string) (*Database, error) {
    opts := badger.DefaultOptions(dir)
    opts.Logger = nil

    if db, err := badger.Open(opts); err != nil {
        if strings.Contains(err.Error(), "LOCK") {
```

```go
            if db, err := retry(dir, opts); err == nil {
                log.Println("database unlocked, value log truncated")
                return &Database{db}, nil
            }
            log.Println("could not unlock database:", err)
        }
        return nil, err
    } else {
        return &Database{db}, nil
    }
}

func (db *Database) Iterator(prefetchValues bool, fn func(*badger.Iterator) error)
error {
    err := db.DB.View(func(txn *badger.Txn) error {

        opts := badger.DefaultIteratorOptions
        opts.PrefetchValues = prefetchValues
        it := txn.NewIterator(opts)
        defer it.Close()

        return fn(it)
    })

    return err
}

func (db *Database) Read(key []byte) ([]byte, error) {
    var value []byte

    err := db.DB.View(func(txn *badger.Txn) error {
        item, err := txn.Get(key)

        if err != nil {
            return err
        }

        value, err = item.ValueCopy(nil)
        return err
    })

    return value, err
}

func (db *Database) Update(key []byte, value []byte) error {

    fmt.Println("Add Key: " + string(key))
    err := db.DB.Update(func(txn *badger.Txn) error {
        return txn.Set(key, value)
    })

    return err
}

func retry(dir string, originalOpts badger.Options) (*badger.DB, error) {
    lockPath := filepath.Join(dir, "LOCK")
    if err := os.Remove(lockPath); err != nil {
        return nil, fmt.Errorf(`removing "LOCK": %s`, err)
    }
    retryOpts := originalOpts
    retryOpts.Truncate = true
    db, err := badger.Open(retryOpts)
    return db, err
}

package network

import (
    "bytes"
    "encoding/gob"
    "encoding/hex"
    "fastchainv2/blockchain"
    "fmt"
    "io"
    "io/ioutil"
    "log"
    "net"
    "os"
    "runtime"
    "syscall"

    "github.com/vrecan/death/v3"
)

const (
    protocol      = "tcp"
    version       = 1
    commandLength = 12
)

var (
    nodeAddress     string
    mineAddress     string
    KnownNodes      = []string{"localhost:3000"}
    blocksInTransit [][]byte
```

```
	memoryPool      = make(map[string]blockchain.Transaction)
)

func RequestBlocks() {
	for _, node := range KnownNodes {
		SendGetBlocks(node)
	}
}

func SendBlock(addr string, b *blockchain.Block) {
	fmt.Println("Send block command: " + addr)

	data := Block{nodeAddress, b.Serialize()}
	payload := GobEncode(data)
	request := append(CmdToBytes("block"), payload...)

	SendData(addr, request)
}

func SendData(addr string, data []byte) {
	conn, err := net.Dial(protocol, addr)

	if err != nil {
		fmt.Printf("%s is not available\n", addr)
		var updatedNodes []string

		for _, node := range KnownNodes {
			if node != addr {
				updatedNodes = append(updatedNodes, node)
			}
		}

		KnownNodes = updatedNodes

		return
	}

	defer func() {
		err := conn.Close()
		if err != nil {
			log.Panic(err)
		}
	}()

	_, err = io.Copy(conn, bytes.NewReader(data))
	if err != nil {
		log.Panic(err)
	}
```

```go
}

func SendInv(address, kind string, items [][]byte) {
    fmt.Println("Send get Inv command: " + address + " kind: " + kind)

    inventory := Inv{nodeAddress, kind, items}
    payload := GobEncode(inventory)
    request := append(CmdToBytes("inv"), payload...)

    SendData(address, request)
}

func SendGetBlocks(address string) {
    fmt.Println("Send get blocks command: " + address)

    payload := GobEncode(GetBlocks{nodeAddress})
    request := append(CmdToBytes("getblocks"), payload...)

    SendData(address, request)
}

func SendGetData(address, kind string, id []byte) {
    fmt.Println("Send get data command: " + address + " kind:" + kind)

    payload := GobEncode(GetData{nodeAddress, kind, id})
    request := append(CmdToBytes("getdata"), payload...)

    SendData(address, request)
}

func SendTx(addr string, tnx *blockchain.Transaction) {
    fmt.Println("Send Tx command: " + addr + " Tx: " + hex.EncodeToString(tnx.ID))

    data := Tx{nodeAddress, tnx.Serialize()}
    payload := GobEncode(data)
    request := append(CmdToBytes("tx"), payload...)

    SendData(addr, request)
}

func SendVersion(addr string, chain *blockchain.BlockChain) {
    fmt.Println("Send version command: " + addr)

    bestHeight := chain.GetBestHeight()
    payload := GobEncode(Version{version, bestHeight, nodeAddress})

    request := append(CmdToBytes("version"), payload...)
    SendData(addr, request)
}

func HandleAddr(request []byte) {
    var buff bytes.Buffer
    var payload Addr

    buff.Write(request[commandLength:])
    dec := gob.NewDecoder(&buff)

    err := dec.Decode(&payload)
    if err != nil {
        log.Panic(err)

    }

    KnownNodes = append(KnownNodes, payload.AddrList...)
    fmt.Printf("there are %d known nodes\n", len(KnownNodes))
    RequestBlocks()
}

func HandleBlock(request []byte, chain *blockchain.BlockChain) {
    var buff bytes.Buffer
    var payload Block

    buff.Write(request[commandLength:])
    dec := gob.NewDecoder(&buff)
    err := dec.Decode(&payload)
    if err != nil {
        log.Panic(err)
    }

    blockData := payload.Block
    block := blockchain.Deserialize(blockData)

    fmt.Println("Received a new block!")
    chain.AddBlock(block)

    fmt.Printf("Added block %x\n", block.Hash)

    if len(blocksInTransit) > 0 {
        blockHash := blocksInTransit[0]
        SendGetData(payload.AddrFrom, "block", blockHash)

        blocksInTransit = blocksInTransit[1:]
    } else {
        UTXOSet := blockchain.UTXOSet{chain}
        UTXOSet.Reindex()
    }
}

func HandleInv(request []byte) {
    var buff bytes.Buffer
    var payload Inv

    buff.Write(request[commandLength:])
    dec := gob.NewDecoder(&buff)

    err := dec.Decode(&payload)
    if err != nil {
        log.Panic(err)
    }

    fmt.Printf("[Handle Inv] With %d, Type: %s\n", len(payload.Items),
payload.Type)

    if payload.Type == "block" {
        blocksInTransit = payload.Items

        blockHash := payload.Items[0]
        SendGetData(payload.AddrFrom, "block", blockHash)

        var newInTransit [][]byte
        for _, b := range blocksInTransit {
            if bytes.Compare(b, blockHash) != 0 {
                newInTransit = append(newInTransit, b)
            }
        }

        blocksInTransit = newInTransit
    }

    if payload.Type == "tx" {
        txID := payload.Items[0]

        if memoryPool[hex.EncodeToString(txID)].ID == nil {
            SendGetData(payload.AddrFrom, "tx", txID)
        }
    }
}

func HandleGetBlocks(request []byte, chain *blockchain.BlockChain) {
    fmt.Println("Handling Get Blocks.")

    var buff bytes.Buffer
    var payload GetBlocks
```

```go
    buff.Write(request[commandLength:])
    dec := gob.NewDecoder(&buff)

    err := dec.Decode(&payload)
    if err != nil {
        log.Panic(err)
    }

    blocks := chain.GetBlockHashes()
    SendInv(payload.AddrFrom, "block", blocks)
}

func HandleGetData(request []byte, chain *blockchain.BlockChain) {
    var buff bytes.Buffer
    var payload GetData

    buff.Write(request[commandLength:])
    dec := gob.NewDecoder(&buff)
    err := dec.Decode(&payload)
    if err != nil {
        log.Panic(err)
    }

    fmt.Printf("Handle Get Data: %s, Type: %s\n", payload.AddrFrom, payload.Type)

    if payload.Type == "block" {
        block, err := chain.GetBlock([]byte(payload.ID))
        if err != nil {
            return
        }

        SendBlock(payload.AddrFrom, &block)
    }

    if payload.Type == "tx" {
        txID := hex.EncodeToString(payload.ID)
        tx := memoryPool[txID]

        SendTx(payload.AddrFrom, &tx)
    }
}

func HandleTx(request []byte, chain *blockchain.BlockChain) {
    var buff bytes.Buffer
    var payload Tx

    buff.Write(request[commandLength:])
    dec := gob.NewDecoder(&buff)

    err := dec.Decode(&payload)
    if err != nil {
        log.Panic(err)
    }

    txData := payload.Transaction
    tx := blockchain.DeserializeTransaction(txData)
    memoryPool[hex.EncodeToString(tx.ID)] = tx

    fmt.Printf("[Handle Tx] From: %s, MemoryPool Size: %d\n", payload.AddrFrom,
len(memoryPool))

    if nodeAddress == KnownNodes[0] {
        for _, node := range KnownNodes {
            if node != nodeAddress && node != payload.AddrFrom {
                SendInv(node, "tx", [][]byte{tx.ID})
            }
        }
    } else {
        fmt.Println("Waiting more transactions to mine.")
        if len(memoryPool) >= 2 && len(mineAddress) > 0 {
            fmt.Println("Starting mining..")
            MineTx(chain)
        }
    }
}

func MineTx(chain *blockchain.BlockChain) {
    var txs []*blockchain.Transaction

    for id := range memoryPool {
        tx := memoryPool[id]

        if chain.VerifyTransaction(&tx) {
            txs = append(txs, &tx)
        }
    }

    if len(txs) == 0 {
        fmt.Println("All Transactions are invalid")
        return
    }

    cbTx := blockchain.CoinbaseTx(mineAddress, "")
    txs = append(txs, cbTx)
    newBlock := chain.MineBlock(txs)
    UTXOSet := blockchain.UTXOSet{chain}
    UTXOSet.Reindex()

    fmt.Println("New Block mined")

    for _, tx := range txs {
        txID := hex.EncodeToString(tx.ID)
        delete(memoryPool, txID)
    }

    for _, node := range KnownNodes {
        if node != nodeAddress {
            SendInv(node, "block", [][]byte{newBlock.Hash})
        }
    }

    if len(memoryPool) > 0 {
        MineTx(chain)
    }
}

func HandleVersion(request []byte, chain *blockchain.BlockChain) {
    var buff bytes.Buffer
    var payload Version

    buff.Write(request[commandLength:])
    dec := gob.NewDecoder(&buff)

    err := dec.Decode(&payload)
    if err != nil {
        log.Panic(err)
    }

    bestHeight := chain.GetBestHeight()
    otherHeight := payload.BestHeight

    fmt.Printf("Handling Version from: %s, Version: %d, BestHeight: %d,
OtherBestHeight: %d\n",
        payload.AddrFrom, payload.Version, bestHeight, otherHeight)

    if bestHeight < otherHeight {
        SendGetBlocks(payload.AddrFrom)
    } else if bestHeight > otherHeight {
        SendVersion(payload.AddrFrom, chain)
    }

    // Add new node to known nodes
```

```go
	if !NodeIsKnown(payload.AddrFrom) {
		KnownNodes = append(KnownNodes, payload.AddrFrom)
	}
}

func HandleConnection(conn net.Conn, chain *blockchain.BlockChain) {
	req, err := ioutil.ReadAll(conn)

	if err != nil {
		log.Panic(err)
	}

	command := BytesToCmd(req[:commandLength])
	fmt.Printf("Received %s command\n", command)

	switch command {
	case "addr":
		HandleAddr(req)
	case "block":
		HandleBlock(req, chain)
	case "inv":
		HandleInv(req)
	case "getblocks":
		HandleGetBlocks(req, chain)
	case "getdata":
		HandleGetData(req, chain)
	case "tx":
		HandleTx(req, chain)
	case "version":
		HandleVersion(req, chain)
	default:
		fmt.Println("Unknown command")
	}

}

func StartServer(nodeID, minerAddress string) {
	nodeAddress = fmt.Sprintf("localhost:%s", nodeID)
	mineAddress = minerAddress

	ln, err := net.Listen(protocol, nodeAddress)

	if err != nil {
		log.Panic(err)
	}

	defer func() {
		err := ln.Close()
```

```
        if err != nil {
            log.Panic(err)
        }
    }()

    chain := blockchain.ContinueBlockChain(nodeID)
    defer func() {
        err := chain.Database.DB.Close()
        if err != nil {
            log.Panic(err)
        }
    }()

    go CloseDB(chain)

    if nodeAddress != KnownNodes[0] {
        SendVersion(KnownNodes[0], chain)
    }

    for {
        conn, err := ln.Accept()
        if err != nil {
            log.Panic(err)
        }
        go HandleConnection(conn, chain)

    }
}

func NodeIsKnown(addr string) bool {
    for _, node := range KnownNodes {
        if node == addr {
            return true
        }
    }

    return false
}

func CloseDB(chain *blockchain.BlockChain) {
    d := death.NewDeath(syscall.SIGINT, syscall.SIGTERM, os.Interrupt)

    d.WaitForDeathWithFunc(func() {
        defer os.Exit(1)
        defer runtime.Goexit()

        err := chain.Database.DB.Close()
        if err != nil {
```

```go
            log.Panic(err)
        }
    })
}

package blockchain

import (
    "bytes"
    "crypto/sha256"
    "encoding/binary"
    "fmt"
    "log"
    "math"
    "math/big"
)

// PoW difficulty
const Difficulty = 12

// Represents proof of work
type ProofOfWork struct {
    Block  *Block
    Target *big.Int
}

// Create new PoW
func NewProof(b *Block) *ProofOfWork {
    target := big.NewInt(1)
    target.Lsh(target, uint(256-Difficulty))

    pow := &ProofOfWork{b, target}

    return pow
}

func (pow *ProofOfWork) InitData(nonce int) []byte {
    data := bytes.Join(
        [][]byte{
            pow.Block.PrevHash,
            pow.Block.HashTransactions(),
            ToHex(int64(nonce)),
            ToHex(int64(Difficulty)),
        },
        []byte{},
    )

    return data
```

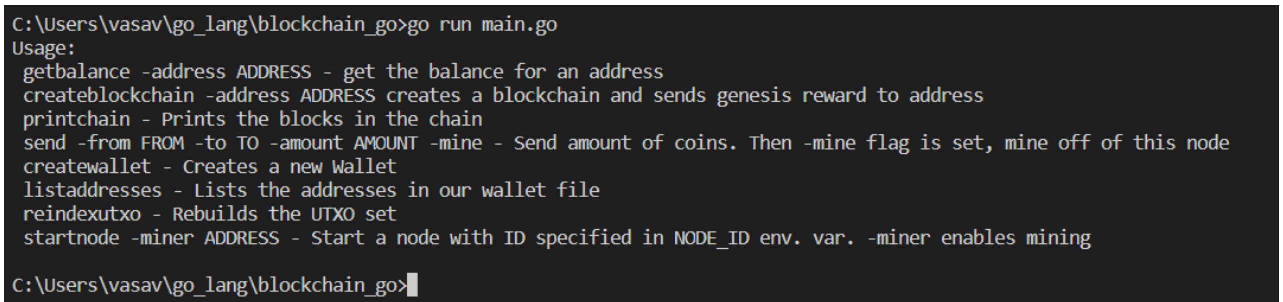
```go
}

func (pow *ProofOfWork) Run() (int, []byte) {
    var intHash big.Int
    var hash [32]byte

    nonce := 0

    for nonce < math.MaxInt64 {
        data := pow.InitData(nonce)
        hash = sha256.Sum256(data)

        fmt.Printf("\r%x", hash)
        intHash.SetBytes(hash[:])

        if intHash.Cmp(pow.Target) == -1 {
            break
        } else {
            nonce++
        }

    }

    fmt.Println()

    return nonce, hash[:]
}

func (pow *ProofOfWork) Validate() bool {
    var intHash big.Int

    data := pow.InitData(pow.Block.Nonce)

    hash := sha256.Sum256(data)
    intHash.SetBytes(hash[:])

    return intHash.Cmp(pow.Target) == -1
}

func ToHex(num int64) []byte {
    buff := new(bytes.Buffer)
    err := binary.Write(buff, binary.BigEndian, num)
    if err != nil {
        log.Panic(err)

    }

    return buff.Bytes()
```

```go
}

package blockchain

import (
    "bytes"
    "encoding/hex"
    "github.com/dgraph-io/badger"
    "log"
)

var (
    utxoPrefix = []byte("utxo-")
)

type UTXOSet struct {
    BlockChain *BlockChain
}

func (u UTXOSet) FindSpendableOutputs(pubKeyHash []byte, amount int) (int,
map[string][]int) {
    unspentOuts := make(map[string][]int)
    accumulated := 0

    err := u.BlockChain.Database.DB.View(func(txn *badger.Txn) error {
        it := txn.NewIterator(badger.DefaultIteratorOptions)
        defer it.Close()

        for it.Seek(utxoPrefix); it.ValidForPrefix(utxoPrefix); it.Next() {
            item := it.Item()
            k := item.Key()

            k = bytes.TrimPrefix(k, utxoPrefix)
            txID := hex.EncodeToString(k)

            err := item.Value(func(val []byte) error {
                outs := DeserializeOutputs(val)

                for outIdx, out := range outs.Outputs {
                    if out.IsLockedWithKey(pubKeyHash) && accumulated < amount {
                        accumulated += out.Value
                        unspentOuts[txID] = append(unspentOuts[txID], outIdx)
                    }
                }

                return nil
            })
            Handle(err)
        }
        return nil
    })
    Handle(err)
    return accumulated, unspentOuts
}

func (u UTXOSet) FindUnspentTransactions(pubKeyHash []byte) []TxOutput {
    var UTXOs []TxOutput

    err := u.BlockChain.Database.DB.View(func(txn *badger.Txn) error {
        opts := badger.DefaultIteratorOptions

        it := txn.NewIterator(opts)
        defer it.Close()

        for it.Seek(utxoPrefix); it.ValidForPrefix(utxoPrefix); it.Next() {
            item := it.Item()
            var v []byte

            err := item.Value(func(val []byte) error {
                v = val
                return nil
            })
            Handle(err)

            outs := DeserializeOutputs(v)
            for _, out := range outs.Outputs {
                if out.IsLockedWithKey(pubKeyHash) {
                    UTXOs = append(UTXOs, out)
                }
            }

        }
        return nil
    })
    Handle(err)

    return UTXOs
}

func (u UTXOSet) FindUTXO(pubKeyHash []byte) []TxOutput {
    var UTXOs []TxOutput

    db := u.BlockChain.Database

    err := db.DB.View(func(txn *badger.Txn) error {
```

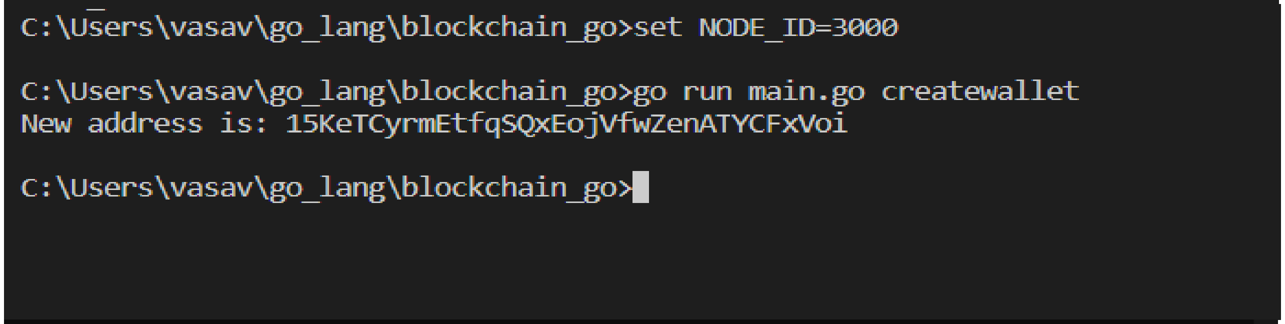
```go
        opts := badger.DefaultIteratorOptions

        it := txn.NewIterator(opts)
        defer it.Close()

        for it.Seek(utxoPrefix); it.ValidForPrefix(utxoPrefix); it.Next() {
            item := it.Item()

            err := item.Value(func(val []byte) error {
                outs := DeserializeOutputs(val)

                for _, out := range outs.Outputs {
                    if out.IsLockedWithKey(pubKeyHash) {
                        UTXOs = append(UTXOs, out)
                    }
                }

                return nil
            })

            Handle(err)
        }

        return nil
    })
    Handle(err)

    return UTXOs
}

func (u UTXOSet) CountTransactions() int {
    db := u.BlockChain.Database
    counter := 0

    err := db.Iterator(true, func(it *badger.Iterator) error {
        for it.Seek(utxoPrefix); it.ValidForPrefix(utxoPrefix); it.Next() {
            counter++
        }

        return nil
    })

    Handle(err)

    return counter
}

func (u UTXOSet) Reindex() {
    db := u.BlockChain.Database

    u.DeleteByPrefix(utxoPrefix)

    UTXO := u.BlockChain.FindUTXO()

    err := db.DB.Update(func(txn *badger.Txn) error {
        for txId, outs := range UTXO {
            key, err := hex.DecodeString(txId)
            if err != nil {
                return err
            }
            key = append(utxoPrefix, key...)

            err = txn.Set(key, outs.Serialize())
            Handle(err)
        }

        return nil
    })
    Handle(err)
}

func (u *UTXOSet) Update(block *Block) {
    db := u.BlockChain.Database

    err := db.DB.Update(func(txn *badger.Txn) error {
        for _, tx := range block.Transactions {
            if tx.IsCoinbase() == false {
                for _, in := range tx.Inputs {
                    updatedOuts := TxOutputs{}
                    inID := append(utxoPrefix, in.ID...)
                    item, err := txn.Get(inID)
                    Handle(err)

                    err = item.Value(func(val []byte) error {
                        outs := DeserializeOutputs(val)

                        for outIdx, out := range outs.Outputs {
                            if outIdx != in.Out {
                                updatedOuts.Outputs = append(updatedOuts.Outputs,
out)
                            }
                        }

                        if len(updatedOuts.Outputs) == 0 {
                            if err := txn.Delete(inID); err != nil {
                                log.Panic(err)
                            }

                        } else {
                            if err := txn.Set(inID, updatedOuts.Serialize()); err
!= nil {
                                log.Panic(err)
                            }
                        }

                        return nil
                    })

                    Handle(err)
                }
            }

            newOutputs := TxOutputs{}
            for _, out := range tx.Outputs {
                newOutputs.Outputs = append(newOutputs.Outputs, out)
            }

            txID := append(utxoPrefix, tx.ID...)
            if err := txn.Set(txID, newOutputs.Serialize()); err != nil {
                log.Panic(err)
            }
        }

        return nil
    })
    Handle(err)
}

func (u *UTXOSet) DeleteByPrefix(prefix []byte) {
    deleteKeys := func(keysForDelete [][]byte) error {
        if err := u.BlockChain.Database.DB.Update(func(txn *badger.Txn) error {
            for _, key := range keysForDelete {
                if err := txn.Delete(key); err != nil {
                    return err
                }
            }
            return nil
        }); err != nil {
            return err
        }

        return nil
    }
```

```
	collectSize := 100000

	err := u.BlockChain.Database.Iterator(false, func(it *badger.Iterator) error {
		keysForDelete := make([][]byte, 0, collectSize)
		keysCollected := 0

		for it.Seek(prefix); it.ValidForPrefix(prefix); it.Next() {
			key := it.Item().KeyCopy(nil)
			keysForDelete = append(keysForDelete, key)
			keysCollected++
			if keysCollected == collectSize {
				if err := deleteKeys(keysForDelete); err != nil {
					log.Panic(err)
				}
				keysForDelete = make([][]byte, 0, collectSize)
				keysCollected = 0
			}
		}

		if keysCollected > 0 {
			if err := deleteKeys(keysForDelete); err != nil {
				log.Panic(err)
			}
		}

		return nil
	})

	Handle(err)
}
```

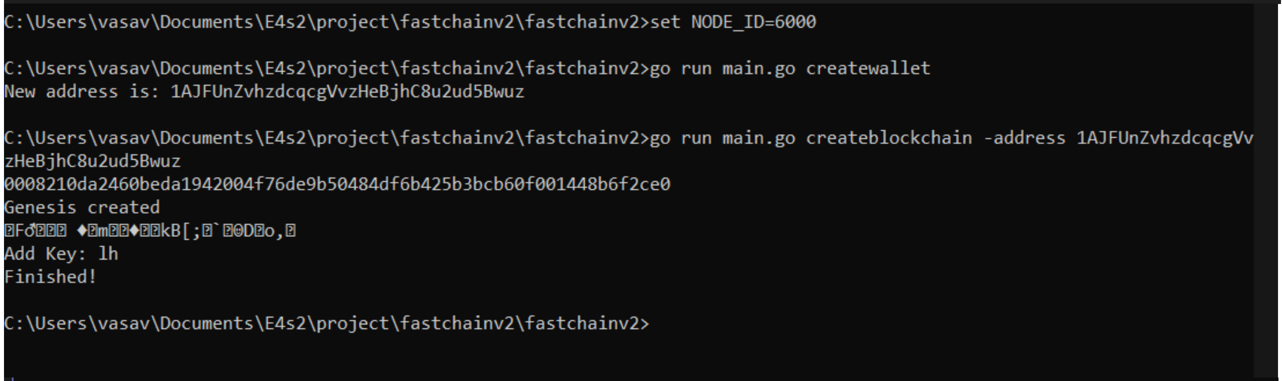
Outputs:



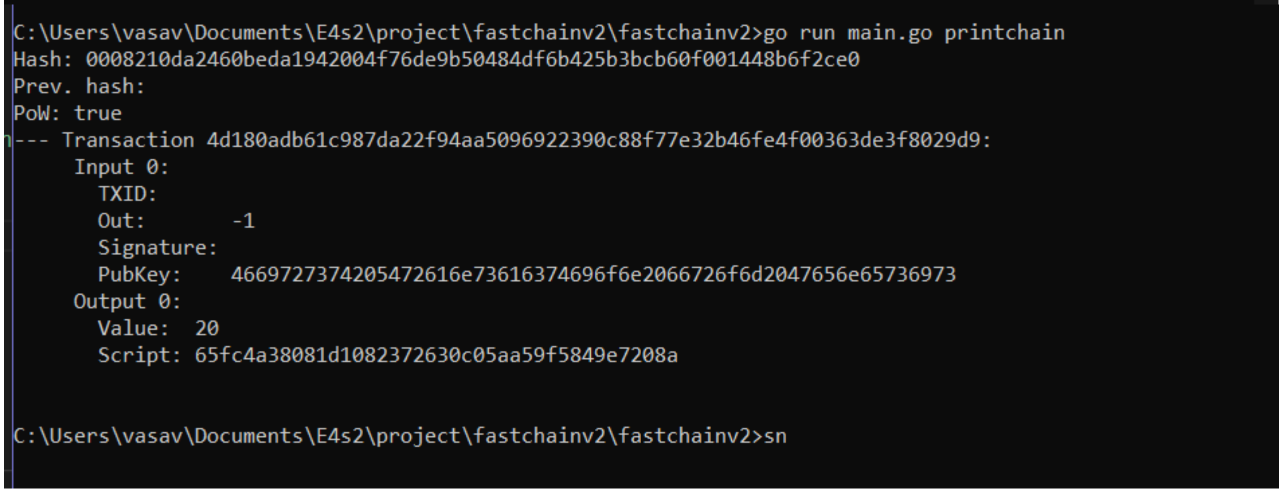
```
C:\Users\vasav\go_lang\blockchain_go>go run main.go
Usage:
 getbalance -address ADDRESS - get the balance for an address
 createblockchain -address ADDRESS creates a blockchain and sends genesis reward to address
 printchain - Prints the blocks in the chain
 send -from FROM -to TO -amount AMOUNT -mine - Send amount of coins. Then -mine flag is set, mine off of this node
 createwallet - Creates a new Wallet
 listaddresses - Lists the addresses in our wallet file
 reindexutxo - Rebuilds the UTXO set
 startnode -miner ADDRESS - Start a node with ID specified in NODE_ID env. var. -miner enables mining

C:\Users\vasav\go_lang\blockchain_go>
```



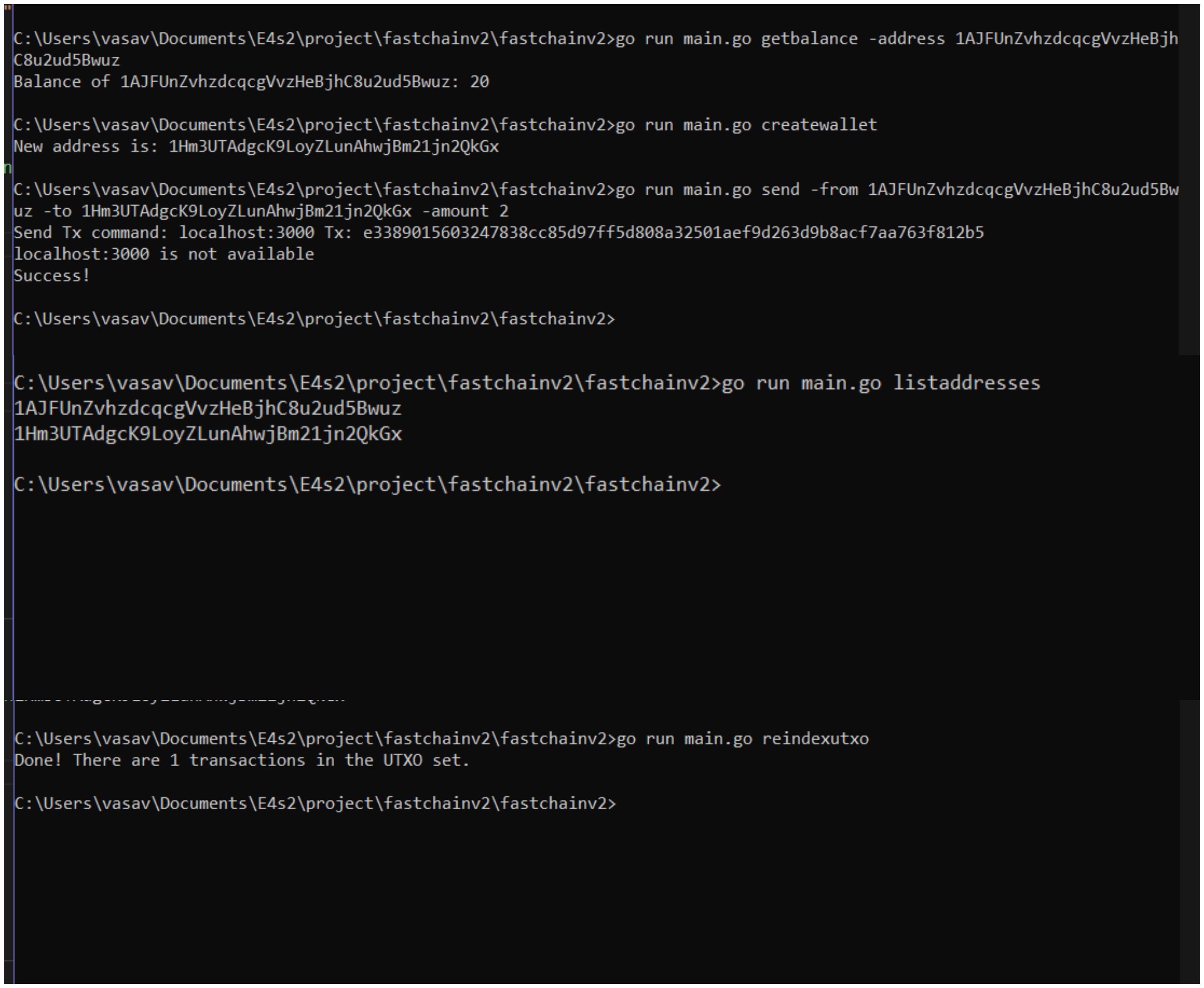
```
C:\Users\vasav\go_lang\blockchain_go>set NODE_ID=3000

C:\Users\vasav\go_lang\blockchain_go>go run main.go createwallet
New address is: 15KeTCyrmEtfqSQxEojVfwZenATYCFxVoi

C:\Users\vasav\go_lang\blockchain_go>
```



```
C:\Users\vasav\Documents\E4s2\project\fastchainv2\fastchainv2>set NODE_ID=6000

C:\Users\vasav\Documents\E4s2\project\fastchainv2\fastchainv2>go run main.go createwallet
New address is: 1AJFUnZvhzdcqcgVvzHeBjhC8u2ud5Bwuz

C:\Users\vasav\Documents\E4s2\project\fastchainv2\fastchainv2>go run main.go createblockchain -address 1AJFUnZvhzdcqcgVv
zHeBjhC8u2ud5Bwuz
0008210da2460beda1942004f76de9b50484df6b425b3bcb60f001448b6f2ce0
Genesis created
�Fσ��� ♦�m��♦��kB[;�`�0D�o,�
Add Key: lh
Finished!

C:\Users\vasav\Documents\E4s2\project\fastchainv2\fastchainv2>
```



```
C:\Users\vasav\Documents\E4s2\project\fastchainv2\fastchainv2>go run main.go printchain
Hash: 0008210da2460beda1942004f76de9b50484df6b425b3bcb60f001448b6f2ce0
Prev. hash:
PoW: true
--- Transaction 4d180adb61c987da22f94aa5096922390c88f77e32b46fe4f00363de3f8029d9:
     Input 0:
       TXID:
       Out:       -1
       Signature:
       PubKey:    4669727374205472616e73616374696f6e2066726f6d2047656e65736973
     Output 0:
       Value:  20
       Script: 65fc4a38081d1082372630c05aa59f5849e7208a


C:\Users\vasav\Documents\E4s2\project\fastchainv2\fastchainv2>sn
```



```
C:\Users\vasav\Documents\E4s2\project\fastchainv2\fastchainv2>go run main.go getbalance -address 1AJFUnZvhzdcqcgVvzHeBjh
C8u2ud5Bwuz
Balance of 1AJFUnZvhzdcqcgVvzHeBjhC8u2ud5Bwuz: 20

C:\Users\vasav\Documents\E4s2\project\fastchainv2\fastchainv2>go run main.go createwallet
New address is: 1Hm3UTAdgcK9LoyZLunAhwjBm21jn2QkGx

C:\Users\vasav\Documents\E4s2\project\fastchainv2\fastchainv2>go run main.go send -from 1AJFUnZvhzdcqcgVvzHeBjhC8u2ud5Bw
uz -to 1Hm3UTAdgcK9LoyZLunAhwjBm21jn2QkGx -amount 2
Send Tx command: localhost:3000 Tx: e3389015603247838cc85d97ff5d808a32501aef9d263d9b8acf7aa763f812b5
localhost:3000 is not available
Success!

C:\Users\vasav\Documents\E4s2\project\fastchainv2\fastchainv2>
```

```
C:\Users\vasav\Documents\E4s2\project\fastchainv2\fastchainv2>go run main.go listaddresses
1AJFUnZvhzdcqcgVvzHeBjhC8u2ud5Bwuz
1Hm3UTAdgcK9LoyZLunAhwjBm21jn2QkGx

C:\Users\vasav\Documents\E4s2\project\fastchainv2\fastchainv2>
```

```
C:\Users\vasav\Documents\E4s2\project\fastchainv2\fastchainv2>go run main.go reindexutxo
Done! There are 1 transactions in the UTXO set.

C:\Users\vasav\Documents\E4s2\project\fastchainv2\fastchainv2>
```

Conclusion:

Here we implemented a core features of a blockchain using Golang, now we can transact , store the blocks , mine in the node, create wallets , list the created wallets , check the balances in the wallets etc. All these has been implemented by following the best practices and minimum standards of the cryptography and security issues along with the scalability in the mind.

Bibilography:

https://golang.dev

https://github.com/dgraph-io/badger

https://pkg.go.dev/crypto

https://gopkg.in/vrecan/death.v3

https://github.com/mr-tron/base58

https://www.blockchain.com/

********************* THANK YOU **********************