



ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ
ΥΠΟΛΟΓΙΣΤΩΝ

Κατανεμημένα Συστήματα
Εξαμηνιαία Εργασία - BlockChat

Αναστάσιος Στέφανος Αναγνώστου
03119051

10 Απριλίου 2024

Περιεχόμενα

I	Σχεδιασμός Συστήματος	3
1	Back-end	3
1.1	Βασικές Δομές Δεδομένων	3
1.1.1	Wallet	3
1.1.2	Account	4
1.1.3	Block	5
1.1.4	Transaction	7
1.2	Κόμβοι	10
1.2.1	BootstrapNode	10
1.2.2	OrdinaryNode	11
2	Front-end	14
2.1	Συνολική Εικόνα	17
II	Πειράματα	18
3	Πειραματική Διάταξη	18
4	Απόδοση του συστήματος	20
4.1	Χρονοβόρα τμήματα του κώδικα	20
4.2	Συναρτήσεις του συστήματος	22
4.3	Ρυθμαπόδοση και Block time	25
5	Κλιμακωσιμότητα του συστήματος	27
5.1	Χρονοβόρα τμήματα του κώδικα	27
5.2	Συναρτήσεις του συστήματος	29
5.3	Ρυθμαπόδοση και Block time	33
6	Δικαιοσύνη	35

Μέρος I

Σχεδιασμός Συστήματος

Αρχικά, ο κώδικας αποτελείται από τα παρακάτω αρχεία με την δομή που φαίνεται:

```
app
└─ Main.hs
src
├─ Account.hs
├─ Block.hs
├─ BootstrapNode.hs
├─ CLI.hs
├─ Lib.hs
├─ OrdinaryNode.hs
├─ ServiceType.hs
├─ Transaction.hs
├─ Types.hs
├─ Utils.hs
└─ Wallet.hs
```

0 directories, 11 files

Σχήμα 1: Δομή του κώδικα

Το αρχείο `Main.hs` είναι ο οδηγός της εφαρμογής, δηλαδή αυτό που εκτελείται για την εκκίνηση του προγράμματος. Στο αρχείο `CLI.hs` υλοποιείται το front-end της εφαρμογής, δηλαδή η διεπαφή με τον χρήστη. Τα υπόλοιπα αρχεία είναι το back-end της εφαρμογής, δηλαδή οι απαραίτητες συναρτήσεις και η λογική για την λειτουργία του blockchain δικτύου. Εξάιρεση αποτελούν τα αρχεία `Utils.hs` και `Types.hs` τα οποία περιέχουν βοηθητικές συναρτήσεις και δηλώσεις τύπων δεδομένων αντίστοιχα.

1 Back-end

1.1 Βασικές Δομές Δεδομένων

Αρχικά θα αναλυθούν οι βασικές δομές δεδομένων που χρησιμοποιούνται στο σύστημα. Αυτές είναι:

- Wallet
- Account
- Block
- Transaction

1.1.1 Wallet

Το αρχείο `Wallet.hs` περιέχει την υλοποίηση της δομής `Wallet`. Ορίζει για τα υπόλοιπα αρχεία τον τύπο `Wallet`, ως ένα συνώνυμο για ένα ζευγάρι από `PublicKey`, `PrivateKey` και την συνάρτηση `generateWallet`, η οποία δημιουργεί ένα τέτοιο ζευγάρι κλειδιών.

```

1 {-# LANGUAGE PackageImports #-}
2
3 module Wallet
4   ( Wallet,
5     emptyWallet,
6     generateWallet,
7   )
8 where
9
10 import Codec.Crypto.RSA (PrivateKey, PublicKey, generateKeyPair)
11 import "crypto-api" Crypto.Random (SystemRandom, newGenIO)
12
13 type Wallet = (PublicKey, PrivateKey)
14
15 emptyWallet :: Wallet
16 emptyWallet = (undefined, undefined)
17
18 -- these returns an IO pair, so in order to actually use
19 -- the values, we have to be inside of something
20 -- that executes an IO action
21 generateWallet :: Int -> IO (PublicKey, PrivateKey)
22 generateWallet bits = do
23   g <- newGenIO :: IO SystemRandom
24   let (pubKey, privKey, _) = generateKeyPair g bits
25   return (pubKey, privKey)

```

Listing 1: Wallet.hs

1.1.2 Account

Το αρχείο `Account.hs` περιέχει την υλοποίηση της δομής `Account`. Ορίζει για τα υπόλοιπα αρχεία των τύπο, καθώς και πρόσβαση σε όλα τα πεδία του, μία μεταβλητή (βασικά, σταθερή συνάρτηση) `initialAccount` και κάποιες βοηθητικές συναρτήσεις για την ενημέρωση του λογαριασμού και γρήγορη πρόσβαση στο υπόλοιπό του, λαμβάνοντας υπόψιν το staking. Δύο πράγματα είναι αξιοσημείωτα σε αυτό το σημείο:

1. Η σταθερή συνάρτηση / μεταβλητή `initialAccount` χρησιμοποιείται από τους κόμβους για την αρχικοποίηση του λογαριασμού τους. Αυτό σημαίνει ότι ο bootstrap κόμβος δεν χρειάζεται να τους στείλει κάποια αρχική πληροφορία σχετική με τον λογαριασμό τους.
2. Η συνάρτηση `updateStake` πανωγράφει το προηγούμενο ποσό stake. Αυτό επιτρέπει στον χρήστη της εφαρμογής να ενημερώνει το stake του.

```

1 data Account = Account
2   { accountBalance :: Double, -- the balance of the account.
3     accountNonce :: Int,
4     -- a field that is incremented with every
5     -- outgoing transaction, in order to guard
6     -- against replay attacks.
7     accountStake :: Double
8     -- the amount of stake that the account has
9     -- for the PoS protocol.
10   }
11 deriving (Show, Eq)
12
13 -- | The initial account has a balance of 1000 coins, a nonce of 0 and no stake.
14 initialAccount :: Account
15 initialAccount = Account 1000 0 0
16
17 -- | The available balance of an account is the balance minus the stake.
18 availableBalance :: Account -> Double
19 availableBalance Account {accountBalance = bal, accountStake = st} = bal - st
20
21 -- | This function takes an amount and an account as arguments and returns a new account
22 -- with the balance updated by the amount.

```

```

23 updateBalanceBy :: Double -> Account -> Account
24 updateBalanceBy amount acc = acc {accountBalance = accountBalance acc + amount}
25
26 -- | This function takes an account as an argument and returns a new account with the nonce
27 -- incremented by 1.
28 updateNonce :: Account -> Account
29 updateNonce acc = acc {accountNonce = accountNonce acc + 1}
30
31 -- | This function takes an amount and an account as arguments and returns a new account
32 -- with the stake updated by the amount.
33 updateStake :: Double -> Account -> Account
34 updateStake amount acc =
35     acc
36     { accountStake = amount,
37       accountBalance = accountBalance acc + (accountStake acc - amount)
38     }

```

Listing 2: Account.hs

1.1.3 Block

Το αρχείο για τον ορισμό της δομής Block θα αναλυθεί σε τμήματα, καθώς εμπεριέχει διάφορα μέρη τα οποία εξυπηρετούν ανεξάρτητους μεταξύ τους σκοπούς.

Αρχικά, στο 3 φαίνεται, εκτός από κάποιες απαραίτητες εισαγωγές βιβλιοθηκών, η δήλωση της δομής BlockInit. Αυτή η δομή χρησιμοποιείται ως βοηθητική δομή για την κανονική δομή Block. Εμπεριέχει όλα τα πεδία του block εκτός από το hash, το οποίο υπολογίζεται και συμπεριλαμβάνεται κατά την δημιουργία ενός στιγμιότυπου της δομής Block.

```

1 data BlockInit = BlockInit
2   { initIndex   :: Int, -- index number of block
3     initTimestamp :: UnixTime, -- microseconds since 1st Jan 1970
4     initTransactions :: [Transaction], -- list of txs
5     initValidator  :: PublicKey, -- public key of the node that validated the tx
6     initPreviousHash :: ByteString -- the hash of the previous block
7   }

```

Listing 3: Block.hs

Στην συνέχεια, το BlockInit ορίζεται ως στιγμιότυπο της κατηγορίας κλάσεων Binary, ώστε να μπορεί να κωδικοποιηθεί σε bytes προς αποστολή και προς hashάρισμα.

```

1 instance Binary BlockInit where
2   put :: BlockInit -> Put
3   put (BlockInit index time trans val prev) = do
4     put index
5     put time
6     put trans
7     put val
8     put prev
9   get :: Get BlockInit
10  get = do
11    index <- get
12    time <- get
13    trans <- get
14    val <- get
15    BlockInit index time trans val <$> get

```

Listing 4: Block.hs

Στην συνέχεια 5, ορίζεται η δομή Block, η οποία περιέχει ακόμα το πεδίο blockCurrentHash. Ορίζεται και αυτή ως στιγμιότυπο της κατηγορίας κλάσεων Binary για τους ίδιους λόγους με το BlockInit.

```

1 data Block = Block
2   { blockIndex   :: Int, -- index number of block
3     blockTimestamp :: UnixTime, -- microseconds since 1st Jan 1970
4     blockTransactions :: [Transaction], -- list of txs

```

```

5     blockValidator :: PublicKey, -- public key of the node that validated the tx
6     blockPreviousHash :: ByteString, -- the hash of the previous block
7     blockCurrentHash :: ByteString
8 }
9 deriving (Show, Eq)
10
11 instance Binary Block where
12     put :: Block -> Put
13     put (Block index time trans val prev curr) = do
14         put index
15         put time
16         put trans
17         put val
18         put prev
19         put curr
20     get :: Get Block
21     get = do
22         index <- get
23         time <- get
24         trans <- get
25         val <- get
26         prev <- get
27         Block index time trans val prev <$> get

```

Listing 5: Block.hs

Ακόμα, στο 6 ορίζεται η συνάρτηση `computeHash` η οποία συμπληρώνει το πεδίο `blockCurrentHash` του `Block` δεδομένου ενός `BlockInit`, ορίζεται η `finalizeBlock`, η οποία επιστρέφει ένα ολοκληρωμένο `Block` από ένα `BlockInit` και τέλος η `createBlock` η οποία είναι ένα wrapper για την `finalizeBlock` και προσφέρεται στον υπόλοιπο κώδικα ως η μέθοδος για την δημιουργία ενός `Block`.

```

1 finalizeBlock :: BlockInit -> Block
2 finalizeBlock initBlock =
3     Block
4     { blockIndex = initIndex initBlock ,
5       blockTimestamp = initTimestamp initBlock,
6       blockTransactions = initTransactions initBlock ,
7       blockValidator = initValidator initBlock ,
8       blockPreviousHash = initPreviousHash initBlock ,
9       blockCurrentHash = computeBlockHash initBlock
10    }
11
12 emptyBlock :: Block
13 emptyBlock = Block 0 (UnixTime 0 0) [] (PublicKey 0 0 65537) BS.empty BS.empty
14
15 -- the capacity of transactions that the block holds is specified
16 -- by an environmental constant called "capacity".
17 createBlock :: Int -> UnixTime -> [Transaction] -> PublicKey -> ByteString -> Block
18 createBlock ind time list pub prev = finalizeBlock $ BlockInit ind time list pub prev

```

Listing 6: Block.hs

Εδώ ορίζεται ένα συνώνυμο τύπου και μία βοηθητική σταθερά.

```

1 type Blockchain = [Block]
2
3 blockMsgHeader :: ByteString
4 blockMsgHeader = "2"

```

Listing 7: Block.hs

Τέλος, ορίζονται συναρτήσεις για την αποστολή ενός `Block` στο δίκτυο και την επικύρωση ενός `Block`. Ακόμα, ορίζονται κάποιες βοηθητικές συναρτήσεις για τον υπολογισμό του μέσου `block time` από τα `time stamps` των `Block` στην αλυσίδα.

```

1 validateBlock newblock prevblock validator = validatorOK && hashOK
2 where

```

```

3     validatorOK = blockValidator newblock == validator
4     hashOK = blockPreviousHash newblock == blockCurrentHash prevblock
5
6     -- | Helper function to broadcast a block to all peers.
7     broadcastBlock :: [Peer] -> Block -> IO ()
8     broadcastBlock peers block = mapM_ sendBlock peers
9     where
10        msg = BS.append blockMsgHeader $ encodeStrict block
11        sendBlock :: (HostName, ServiceName) -> IO ()
12        sendBlock (host, port) = connect host port $ \(sock, _) -> send sock msg
13
14    -- | This function takes a transaction and a blockchain and returns True if the transaction is unique.
15    -- A transaction is unique if it is not present in any of the blocks in the blockchain.
16    txIsUnique :: Transaction -> Blockchain -> Bool
17    txIsUnique tx = all (notElem tx . blockTransactions)
18
19    -- | This function takes a blockchain and returns the mean time between blocks.
20    meanBlockTime :: Blockchain -> Double
21    meanBlockTime chain = harmonicMean $ map unixdiffToSecs times''
22    where
23        times = map blockTimestamp chain
24        times' = zip times (tail times)
25        times'' = map (uncurry diffUnixTime) times'
26        unixdiffToSecs :: UnixDiffTime -> Double
27        unixdiffToSecs (UnixDiffTime sec usec) = realToFrac sec + realToFrac usec / 1000000
28
29    harmonicMean :: Fractional a => [a] -> a
30    harmonicMean numbers = amount / inverseSum
31    where (inverseSum, amount) = foldr (\num (a, b) -> (a + 1 / num, b + 1)) (0, 0) numbers

```

Listing 8: Block.hs

1.1.4 Transaction

Ο κώδικας για την υλοποίηση της δομής Transaction ακολουθεί τις ίδιες αρχές με τον κώδικα για την υλοποίηση της δομής Block. Για αυτόν τον λόγο δεν θα παρουσιαστεί ολόκληρος αλλά τα πιο σημαντικά μέρη.

Αρχικά, και αυτός ο τύπος ορίζεται ως στιγμιότυπο της κατηγορίας κλάσεων Binary, επειδή πρέπει και αυτός να μπορεί να κωδικοποιηθεί σε bytes για την αποστολή του στο δίκτυο.

Οι πιο σημαντικές συναρτήσεις είναι οι ακόλουθες:

```

1  -- | Broadcasts a transaction to a list of peers.
2  broadcastTransaction :: [Peer] -> Transaction -> IO ()
3  broadcastTransaction peers t = mapM_ sendMsg peers
4  where
5      msg = append txMsgHeader $ encodeStrict t
6      sendMsg :: (HostName, ServiceName) -> IO ()
7      sendMsg (host, port) = connect host port $ \(sock, _) -> send sock msg
8
9  -- | Verifies the signature of a transaction.
10 verifySignature :: Transaction -> Bool
11 verifySignature t = verify from tid sig
12 where
13     -- pay attention to the order of the arguments
14     from = senderAddress t
15     tid = B.fromStrict $ hashID t
16     sig = B.fromStrict $ signature t
17
18 -- | This function takes a transaction and a map of public keys to accounts and returns
19 -- whether the transaction is valid or not.
20 validateTransaction :: Transaction -> PubKeyToAcc -> Bool
21 validateTransaction t m = verifySignature t && maybe False validateSender senderAcc
22 where
23     validateSender acc = availableBalance acc >= txCost t
24     senderAcc = Map.lookup (senderAddress t) m

```

Listing 9: Transaction.hs

όπου PubKeyToAcc και Peer είναι τύποι που ορίζονται στο αρχείο Types.hs:

```

1
2  type Peer = (HostName, ServiceName)
3
4  type PubKeyToAcc = Map.Map PublicKey Account

```

Listing 10: Types.hs

ενώ η συνάρτηση encodeStrict ορίζεται στο αρχείο Utils.hs:

```

1  encodeStrict :: (Binary a) => a -> BS.ByteString
2  encodeStrict = BS.toStrict . encode
3
4  decodeMaybe :: (Binary a) => Maybe BS.ByteString -> a
5  decodeMaybe = maybe (decode "") (decode . BS.fromStrict)
6
7  receiveChunks :: Socket -> Int -> IO BS.ByteString
8  receiveChunks socket limit = receiveChunks' ""
9  where
10     receiveChunks' acc | BS.length acc < limit = do
11         raw <- recv socket 4096 -- have a byte
12         case raw of
13             Nothing -> return acc -- can't get no more bytes
14             Just msg -> receiveChunks' $ BS.append acc msg -- keep eating
15     receiveChunks' acc | BS.length acc == limit = return acc
16     receiveChunks' acc = return $ fst $ BS.splitAt limit acc

```

Listing 11: Utils.hs

Σημαντική επίσης είναι η ακόλουθη συνάρτηση, που χρησιμοποιείται για την ενημέρωση των λογαριασμών βάσει μίας συναλλαγής, και η επέκτασή της σε λίστα συναλλαγών. Αυτές θα χρησιμοποιηθούν από τους κόμβους για την ενημέρωση του soft state τους. Παρατηρείται ότι δεν ενημερώνουν το nonce του λογαριασμού. Για πρακτικούς λόγους, που μπορούν να εξηγηθούν αναλυτικά αργότερα, στον κώδικα των κόμβων, η ενημέρωση γίνεται από το front-end.


```

1  -- | This function takes a transaction and a state of accounts as arguments and returns
2  -- a new state of accounts, as a result of the transaction. It does not update the nonce
3  updateAccsByTX :: Transaction -> PubKeyToAcc -> PubKeyToAcc
4  updateAccsByTX t m = case serviceType t of
5    Coins c ->
6      -- if the receiver is the zeropub, then the transaction is a staking transaction
7      if receiverAddress t == zeropub
8        then
9          Map.adjust (updateBalanceBy (-txFee t) . updateStake c) (senderAddress t) m
10       else
11         let cost = txCost t
12           sender = senderAddress t
13           receiver = receiverAddress t
14           temp = Map.adjust (updateBalanceBy (-cost)) sender m
15         in Map.adjust (updateBalanceBy c) receiver temp
16 Message _ ->
17   let cost = txCost t
18     sender = senderAddress t
19   in Map.adjust (updateBalanceBy (-cost)) sender m
20 Both (c, _) ->
21   let cost = txCost t
22     sender = senderAddress t
23     receiver = receiverAddress t
24     temp = Map.adjust (updateBalanceBy (-cost)) sender m
25   in Map.adjust (updateBalanceBy c) receiver temp
26
27 -- | This function takes a list of transactions and a state of accounts as arguments and returns
28 -- a new state of accounts, as a result of the transactions.
29 updateAccsByTXs :: [Transaction] -> PubKeyToAcc -> PubKeyToAcc
30 updateAccsByTXs txs initial = foldr updateAccsByTX initial txs

```

Listing 12: Transaction.hs

Τέλος, φαίνεται η συνάρτηση `createTransaction` που θα χρησιμοποιείται για την δημιουργία συναλλαγών και κάποιες βοηθητικές συναρτήσεις, ίδιες στην λογική και στην δομή με τις αντίστοιχες του `Block.hs`.

```

1  -- | This function takes a transaction (only the initial fields) and returns its hash.
2  computeHashID :: TransactionInit -> ByteString
3  computeHashID = convert . hashWith SHA256 . B.toStrict . encode
4
5  -- | This function takes a private key and a bytestring and returns the signature of the bytestring.
6  computeSignature :: PrivateKey -> ByteString -> ByteString
7  computeSignature privkey bytestring = B.toStrict $ sign privkey (B.fromStrict bytestring)
8
9  -- | This function takes a transaction and a private key and
10 -- returns the transaction with the signature field filled.
11 finalizeTransaction :: TransactionInit -> PrivateKey -> Transaction
12 finalizeTransaction initTx privKey =
13   Transaction
14     { senderAddress = initSenderAddress initTx,
15       receiverAddress = initReceiverAddress initTx,
16       serviceType = initServiceType initTx,
17       nonce = initNonce initTx,
18       hashID = computeHashID initTx,
19       signature = computeSignature privKey (computeHashID initTx)
20     }
21
22 -- | This function takes a public key (sender), another public key (receiver), a service type,
23 -- a counter and a private key and returns a transaction.
24 createTransaction :: PublicKey -> PublicKey -> ServiceType -> Int -> PrivateKey -> Transaction
25 createTransaction p1 p2 s n = finalizeTransaction (TransactionInit p1 p2 s n)

```

Listing 13: Transaction.hs

1.2 Κόμβοι

Η λογική των κόμβων διατυπώνεται στα αρχεία `OrdinaryNode.hs` και `BootstrapNode.hs`. Στο πρώτο αρχείο ορίζεται η λογική των κόμβων που συμμετέχουν στο δίκτυο, ενώ στο δεύτερο ορίζεται η λογική του κόμβου που είναι υπεύθυνος για την αρχικοποίηση του δικτύου.

1.2.1 BootstrapNode

Κατ'αρχάς, ορίζονται κάποιοι βοηθητικοί τύποι:

```
1
2  -- | BootInfo does not change. It is set once, using the arguments
3  -- | passed to the program and then remains constant. The IP and the PORT
4  -- | of the Boot node are known to the other nodes beforehand.
5  data BootInfo = BootInfo
6    { bootNodeID :: Int,
7      bootNodeIP :: HostName,
8      bootNodePort :: ServiceName,
9      bootNodeNumb :: Int -- number of nodes to insert into the network
10    }
11    deriving (Show, Eq)
12
13  data BootState = BootState
14    { bootCurrID :: Int,
15      bootPublicKeys :: [(Int, PublicKey)],
16      bootPeers :: [(HostName, ServiceName)],
17      bootBlockchain :: Blockchain
18    }
19    deriving (Show, Eq)
20
21  -- | This is the initial state of the boot node
22  emptyBootState :: BootState
23  emptyBootState = BootState 0 [] [] []
24
25  -- | This is the function that the driver calls to
26  -- | initiate the bootstrap node logic
27  bootstrapNode :: BootInfo -> IO BootState
28  bootstrapNode = runReaderT bootstrapNodeLogic
```

Listing 14: `BootstrapNode.hs`

Στην συνέχεια ορίζονται κάποιες βοηθητικές συναρτήσεις καθώς και η συνάρτηση `server`, η οποία εκφράζει την λογική του κόμβου.

```
1  -- | This is a helper function that updates the state of the boot node
2  updateState :: (Int, (PublicKey, HostName, ServiceName)) -> BootState -> (BootState, ())
3  updateState t s = (s {bootPublicKeys = newKeys, bootPeers = newPeers}, ())
4    where
5      (num, (pub, ip, port)) = t
6      newKeys = (num, pub) : bootPublicKeys s
7      newPeers = (ip, port) : bootPeers s
8
9  -- | This is a helper function that increments the current ID of the boot state
10  incrementID :: BootState -> (BootState, Int)
11  incrementID s = (s {bootCurrID = newID}, newID)
12    where
13      newID = bootCurrID s + 1
14
15  -- | This is the function that handles the server logic
16  server :: [MVar Int] -> IORef BootState -> (Socket, SockAddr) -> IO ()
17  server triggers ioref (socket, _) = do
18    currID <- atomicModifyIORef ioref incrementID
19    when (currID <= length triggers) $ do
20      msg <- recv socket 4096 -- for public key (2048), an ip and a port
21      -- for some reason the startup phase gets stuck
22      -- if 'receiveChunks' is used instead of 'recv'
23      let keyval = decodeMaybe msg :: (PublicKey, HostName, ServiceName)
```

```

24     atomicModifyIORef' ioref $ updateState (currID, keyval)
25     send socket $ encodeStrict currID

```

Listing 15: BootstrapNode.hs

MVar είναι μεταβλητές οι οποίες χρησιμοποιούνται για συγχρονισμό. Εν προκειμένω, παρνώνται στην συνάρτηση `server` ώστε να μην συνεχίσει η εκτέλεση του κυρίου νήματος εκτέλεσης του κόμβου, προτού έχει αποστείλει όλα τα ids στους κόμβους του δικτύου. Το `BootState` είναι `IORef` γιατί η συνάρτηση `server` δεν μπορεί να επιστρέψει τιμή: το επιβάλλει ο τύπος της.

Τέλος, ολόκληρη η συνάρτηση εκτέλεσης του κόμβου, καθώς και κάποιες βοηθητικές συναρτήσεις, φαίνονται παρακάτω:

```

1  -- | This is a helper function that creates the genesis transaction
2  createGenesisTX :: Wallet -> Int -> Transaction
3  createGenesisTX (pub, priv) totalnodes = createTransaction zeropub pub tx 1 priv
4  where
5      tx = Coins $ 1000 * fromIntegral totalnodes
6
7  -- | This is a helper function that creates the genesis block
8  createGenesisBlock :: UnixTime -> Wallet -> Int -> Block
9  createGenesisBlock time wallet totalnodes = createBlock 1 time [genesisTX] zeropub prevHash
10 where
11     prevHash = encodeStrict (1 :: Int)
12     genesisTX = createGenesisTX wallet totalnodes
13
14 bootstrapNodeLogic :: ReaderT BootInfo IO BootState
15 bootstrapNodeLogic = do
16     myip <- asks bootNodeIP
17     myport <- asks bootNodePort
18     mywallet <- liftIO $ generateWallet 2048 -- get a wallet
19     totalNodes <- asks bootNodeNumb
20     -- setup the state and locks
21     state <- liftIO $ newIORef emptyBootState
22     triggers <- liftIO $ mapM (const newEmptyMVar) [1 .. totalNodes]
23     _ <- liftIO $ forkIO $ serve (Host myip) myport $ server triggers state
24     liftIO $ mapM_ takeMVar triggers -- wait for all nodes to connect
25     time <- liftIO getUnixTime
26     fstate <- liftIO $ readIORef state
27     let genesisBI = createGenesisBlock time mywallet totalNodes
28         final = fstate {bootBlockchain = [genesisBI]}
29         keys = bootPublicKeys final :: [(Int, PublicKey)]
30         friends = bootPeers final :: [(HostName, ServiceName)]
31         msg = BS.append "0" (encodeStrict (keys, friends, genesisBI))
32
33     -- reversing the list of friends seems to actually matter, at least when creating nodes
34     -- from the command line. It seems that the bootstrap node tries to connect to the last
35     -- node to enter the network too fast, before the node has time to start the server.
36     liftIO $ mapM_ (\(ip, port) -> connect ip port (\x -> send (fst x) msg)) (reverse friends)
37     return final

```

Listing 16: BootstrapNode.hs

Φαίνεται πως ουσιαστικά το μόνο που κάνει η συνάρτηση είναι να γεννάει ένα νήμα το οποίο τρέχει την συνάρτηση `server`, περιμένει χρησιμοποιώντας τις μεταβλητές `MVar` ώστε να συνδεθούν όλοι οι κόμβοι στο δίκτυο και, τέλος, χρησιμοποιεί το `state` που χτίστηκε σταδιακά για να αποστείλει όλες τις απαραίτητες πληροφορίες στους κόμβους.

1.2.2 OrdinaryNode

Τα σημαντικά σημεία για την λογική των κόμβων που συμμετέχουν στο δίκτυο είναι 2:

- Η συνάρτηση `server` που εκφράζει την λογική αρχικοποίησης καθενός κόμβου επικοινωνώντας με τον bootstrap κόμβο καθώς και την επικοινωνία του με τους υπόλοιπους κόμβους.
- Οι συναρτήσεις `processTXs` και `mint` που εκφράζουν τον τρόπο με τον οποίο οι κόμβοι διαχειρίζονται τις συναλλαγές και την υλοποίηση του Proof of Stake πρωτοκόλλου.

Αρχικά, ορίζονται κάποιοι βοηθητικοί τύποι:

```

1 type TXQueue = TQueue Transaction
2
3 type BLQueue = TQueue Block
4
5 type StartupState = ([Int, PublicKey], [Peer], Block)
6
7 -- NodeInfo does not change.
8 -- It is set once, using the arguments passed to the program and then remains constant.
9 data NodeInfo = NodeInfo
10 { nodeIP :: HostName,
11   nodePort :: ServiceName,
12   nodeInfoWallet :: Wallet

```

Listing 17: OrdinaryNode.hs

Ορίζονται απλά συνώνυμα τύπων. Το σημαντικό είναι ο τύπος TQueue (Transactional Queue), ο οποίος επδέχεται επεξεργασίας από πολλαπλά νήματα.

Στην συνέχεια η συνάρτηση server:

```

1 deriving (Show, Eq)
2
3 enqueueTQ :: TQueue a -> a -> STM ()
4 enqueueTQ = writeTQueue
5
6 dequeueTQ :: TQueue a -> STM a
7 dequeueTQ = readTQueue
8
9 data ServerEnv = ServerEnv (TVar StartupState) (MVar Int) TXQueue BLQueue
10
11 server :: ServerEnv -> (Socket, SockAddr) -> IO ()
12 server (ServerEnv startupState trigger queuedTxes queuedBlocks) (socket, _) = do
13   resp <- receiveChunks socket $ 16 * 4096
14   let (msgtype, msg) = BS.splitAt 1 resp
15   case msgtype of
16     "0" -> handleDecodeStartup startupState trigger msg
17     "1" -> handleDecodeTx queuedTxes msg
18     "2" -> handleDecodeBlock queuedBlocks msg
19     _ -> liftIO $ putStrLn "This should not be seen"
20
21 -- The following are debugging wrappers. The logic could be inlined above
22 -- in the server, but the wrappers are helpful for debugging, in case
23 -- something goes wrong.
24 handleDecodeStartup :: TVar StartupState -> MVar Int -> BS.ByteString -> IO ()
25 handleDecodeStartup startupState trigger msg =
26   case decodeOrFail (LBS.fromStrict msg) of
27     Left (_, _, errmsg) -> putStrLn $ "From startup: " ++ errmsg
28     Right (_, _, result) -> do
29       atomically $ writeTVar startupState result
30       putMVar trigger 1
31
32 handleDecodeTx :: TXQueue -> BS.ByteString -> IO ()
33 handleDecodeTx queue msg =
34   case decodeOrFail (LBS.fromStrict msg) of
35     Left (_, _, errmsg) -> putStrLn $ "From tx: " ++ errmsg
36     Right (_, _, tx) -> atomically $ enqueueTQ queue tx
37
38 handleDecodeBlock :: BLQueue -> BS.ByteString -> IO ()
39 handleDecodeBlock queue msg =
40   case decodeOrFail (LBS.fromStrict msg) of
41     Left (_, _, errmsg) -> putStrLn $ "From block: " ++ errmsg
42     Right (_, _, block) -> atomically $ enqueueTQ queue block

```

Listing 18: OrdinaryNode.hs

Ουσιαστικά, η συνάρτηση server λαμβάνει ένα μήνυμα, ελέγχει την κεφαλίδα του μηνύματος για να διακρίνει αν πρόκειται για συναλλαγή, block ή μήνυμα αρχικοποίησης και τέλος το αποκωδικοποιεί καταλλήλως.

Ορίζονται επίσης δύο βοηθητικές συναρτήσεις για το στάδιο minting του πρωτοκόλλου:

```

1 sampleValidator :: (RandomGen g) => g -> [(Double, Int)] -> Int
2 sampleValidator g probs = evalState (sampleStateRVar (weightedCategorical epsProbs)) g
3 where
4   epsilon = 0.0001 :: Double
5   epsProbs = map \(x, y) -> (x + epsilon, y) probs
6
7   -- | Helper function to get a valid block from a queue of blocks
8   -- given the last block and the public key of the validator.
9   getValidatorBlockFrom :: BLQueue -> Block -> PublicKey -> IO Block
10  getValidatorBlockFrom qBlocks lastBlock valKey = do
11    block <- atomically $ dequeueTQ qBlocks
12    if validateBlock block lastBlock valKey

```

Listing 19: OrdinaryNode.hs

Σημειώνεται ότι η συνάρτηση `getValidatorBlockFrom` ουσιαστικά μπλοκάρει αν ο κόμβος δεν λαμβάνει το block που περιμένει. Αυτό εξασφαλίζει ότι ο κόμβος δεν συνεχίζει το πρωτόκολλο μόνος του.

Μόλις ο κόμβος αρχικοποιηθεί (δεν παρουσιάζεται ο κώδικας γιατί δεν έχει κάποιο ενδιαφέρον) γεννάει ένα νήμα εκτέλεσης το οποίο τρέχει την συνάρτηση `processTXs` και ένα άλλο νήμα εκτέλεσης το οποίο τρέχει το front-end. Ήδη είναι εκκινημένο ένα νήμα εκτέλεσης που τρέχει την συνάρτηση `server`.

```

1 accountRef <- liftIO $ newIORef initialAccount
2 let initialAccounts = Map.fromList $ map ((,initialAccount) . snd) keys
3   clishared = (blockchainRef, accountRef) :: CLISharedState
4   mypeers = filter (/= (myid, mypub)) keys
5   cliinfo = CLIInfo mywallet (Map.fromList mypeers) ip port friends
6   -- This function processes transactions (keeping track of the counter) and mints
7   -- a new block when the counter reaches the capacity.
8   processTXs :: IO ()
9   processTXs = processTXs' clishared [] queuedTXs (initialAccounts, initialAccounts)
10  where
11    processTXs' :: CLISharedState -> [Transaction] -> TXQueue -> (PubKeyToAcc, PubKeyToAcc) -> IO ()
12    processTXs' sharedState vTXs qTXs (accmap, fallback) = do
13      when (length vTXs /= capacity) $ do
14        tx <- atomically $ dequeueTQ qTXs
15        blockchain <- (readIORef . fst) sharedState
16        if validateTransaction tx accmap && txIsUnique tx blockchain
17          then
18            processTXs' sharedState (tx : vTXs) qTXs (updateAccsByTX tx accmap, fallback)
19          else
20            processTXs' sharedState vTXs qTXs (accmap, fallback)
21        blockchain <- (readIORef . fst) sharedState
22        (newblock, newaccs) <- mint (head blockchain) vTXs (accmap, fallback)
23        writeIORef (fst sharedState) (newblock : blockchain)
24        -- the cli is responsible for updating the account nonce
25        atomicModifyIORef' (snd sharedState) (\a -> ((newaccs Map.! mypub) {accountNonce = accountNonce a}, ()))
26        processTXs' sharedState [] qTXs (newaccs, newaccs)

```

Listing 20: OrdinaryNode.hs

Η λογική της συνάρτησης `processTXs` είναι απλή: αν οι επικυρωμένες συναλλαγές είναι λιγότερες από την χωρητικότητα του block, τότε συνέχισε να επικυρώνεις. Μόλις αυτό δεν ισχύει, κάλεσε την `mint`, ενημέρωσε το state με το καινούργιο block και ξεκίνα πάλι την διαδικασία.

Η δε συνάρτηση `mint` είναι επίσης απλή:

```

1
2 mint :: Block -> [Transaction] -> (PubKeyToAcc, PubKeyToAcc) -> IO (Block, PubKeyToAcc)
3 mint = mint' (filter ((ip, port) /=) friends)
4
5 mint' :: [Peer] -> Block -> [Transaction] -> (PubKeyToAcc, PubKeyToAcc) -> IO (Block, PubKeyToAcc)
6 mint' peers lastBlock vTXs (accountMap, fallback) = do
7   let -- the stake has to be taken from the fallback state
8       -- because subsequent states are not validated
9       accs = Map.elems fallback
10      weights = zip (map accountStake accs) [1 ..]
11
12      prevhash = blockPreviousHash lastBlock

```

```

13     currhash = blockCurrentHash lastBlock
14     seed = decode . BS.fromStrict $ prevhash
15
16     validator = sampleValidator (mkStdGen seed) weights
17     valkey = fst $ Map.elemAt (validator - 1) accountMap
18   if valkey == mypub
19   then do
20     currtime <- getUnixTime
21     let newBlock = createBlock (blockIndex lastBlock + 1) currtime (reverse vTxS) valkey currhash
22         fees = txsFee vTxS
23         plusFees acc = acc {accountBalance = accountBalance acc + fees}
24         newAccs = Map.update (Just . plusFees) valkey accountMap
25     broadcastBlock peers newBlock
26     return (newBlock, newAccs)
27   else do
28     -- spin on the queue of blocks until a valid one is found
29     block <- getValidatorBlock lastBlock valkey
30     let finalAccs = updateAccsByTXs (blockTransactions block) fallback

```

Listing 21: OrdinaryNode.hs

Ουσιαστικά, εκτελεί την λοταρία βάσει του τελευταίου block και του stake καθενός κόμβου και ελέγχει το public key που κληρώθηκε. Αν είναι το ίδιο του κόμβου, τότε δημιουργεί ένα νέο block και το αποστέλλει στο δίκτυο. Ειδικά, περιμένει μέχρι να παραλάβει το block από τον validator. Σε αυτήν την περίπτωση, ενημερώνει κατάλληλα το state χρησιμοποιώντας το τελευταίο έγκυρο state που έχει.

2 Front-end

Το front-end της εφαρμογής είναι υλοποιημένο στο αρχείο CLI.hs.

```

1  data CLIInfo = CLIInfo
2    { cliWallet  :: Wallet, -- Wallet of the user
3      cliIDtoKey :: Map.Map Int PublicKey, -- Peers of the user
4      cliNodeIP  :: HostName, -- Node IP of the user
5      cliNodePort :: ServiceName, -- Node Port of the user
6      cliPeers   :: [(HostName, ServiceName)]
7    }
8  deriving (Show)
9
10 type CLISharedState = (IORef Blockchain, IORef Account)
11
12 -- | Send a transaction to the network
13 sendTx :: Int -> ServiceType -> Account -> ReaderT CLIInfo IO ()
14 sendTx recvID service myacc = do
15   peers <- asks cliPeers
16   keymap <- asks cliIDtoKey
17   (pub, priv) <- asks cliWallet
18   let mynonce = accountNonce myacc
19       recvPub = Map.lookup recvID keymap
20   case recvPub of
21     Nothing -> liftIO $ putStrLn "Invalid recipient. Check whether the ID was yours or if it does not exist."
22     Just somekey -> do
23       let tx = createTransaction pub somekey service mynonce priv
24       liftIO $ putStrLn $ "Sending " ++ show service ++ " to " ++ show recvID
25       liftIO $ broadcastTransaction peers tx -- this handles the correct sending
26
27 -- | Send a staking transaction to the network
28 stake :: Double -> Account -> ReaderT CLIInfo IO ()
29 stake coins myacc = do
30   peers <- asks cliPeers
31   (pub, priv) <- asks cliWallet
32   let mynonce = accountNonce myacc
33       tx = createTransaction pub zeropub (Coins coins) mynonce priv
34   liftIO $ putStrLn $ "Staking " ++ show coins
35   liftIO $ broadcastTransaction peers tx

```

Listing 22: CLI.hs

Οι συναρτήσεις 24 καλούνται όταν parseαριστεί επιτυχώς η είσοδος του χρήστη και στέλνουν την αντίστοιχη συναλλαγή στο δίκτυο. Η συνάρτηση για το parse είναι η `handle`.

```

1  -- | Handle the input from the user
2  handle :: String -> CLISharedState -> ReaderT CLInfo IO ()
3  handle input shared = do
4    let tokens = words input
5    (blockref, accref) = shared
6    case tokens of
7      ("t" : numStr : "Coins" : coinsStr : _) ->
8        case (reads numStr, reads coinsStr) of
9          (([num, ""], [(coins, "")]) -> do
10             liftIO $ atomicModifyIORef' accref (\a -> (a {accountNonce = accountNonce a + 1}, ()))
11             acc' <- liftIO $ readIORef accref
12             sendTx num (Coins coins) acc'
13           _ -> liftIO $ putStrLn "Invalid command. Try entering 'help' for help."
14      ("t" : numStr : "Message" : msgParts) ->
15        case reads numStr of
16          ([num, ""]) -> do
17             liftIO $ atomicModifyIORef' accref (\a -> (a {accountNonce = accountNonce a + 1}, ()))
18             acc' <- liftIO $ readIORef accref
19             sendTx num (Message $ unwords msgParts) acc'
20           _ -> liftIO $ putStrLn "Invalid command. Try entering 'help' for help."
21      ("t" : numStr : "Both" : coinsStr : ", " : msgParts) ->
22        case (reads numStr, reads coinsStr) of
23          (([num, ""], [(coins, "")]) -> do
24             liftIO $ atomicModifyIORef' accref (\a -> (a {accountNonce = accountNonce a + 1}, ()))
25             acc' <- liftIO $ readIORef accref
26             sendTx num (Both (coins, unwords msgParts)) acc'
27           _ -> liftIO $ putStrLn "Invalid command. Try entering 'help' for help."
28      ("stake" : coinsStr : _) ->
29        case reads coinsStr of
30          ([coins, ""]) -> do
31             liftIO $ atomicModifyIORef' accref (\a -> (a {accountNonce = accountNonce a + 1}, ()))
32             acc' <- liftIO $ readIORef accref
33             stake coins acc'
34           _ -> liftIO $ putStrLn "Invalid command. Try entering 'help' for help."
35      ["view"] -> do
36         blockchain <- liftIO $ readIORef blockref
37         prettyPrintBlock (head blockchain)
38      ["blockchain"] -> do
39         liftIO $ threadDelay 5000000 -- just for testing
40         blockchain <- liftIO $ readIORef blockref
41         prettyPrintBlockchain blockchain
42         -- show the mean time between blocks but disregard the genesis block
43         -- that is the last one in the list
44         liftIO $ putStrLn $ "Mean time between blocks: " ++ show (meanBlockTime $ init blockchain)
45      ["balance"] -> do
46         acc <- liftIO $ readIORef accref
47         liftIO $ print (accountBalance acc)
48      ["peers"] -> do
49         keymap <- asks cliDtoKey
50         liftIO $ prettyPrintPeers keymap
51      ["load", filename] -> do
52         liftIO $ putStrLn $ "Loading transactions from " ++ filename
53         -- execute each line of the file as a command
54         contents <- liftIO $ readFile filename
55         mapM_ ('CLI.handle' shared) (lines contents)
56      ["exit"] -> liftIO $ putStrLn "Exiting .." >> exitSuccess
57      ["help"] -> do
58         liftIO $ putStrLn "t <recipient id> Coins <coins>      - send coins"
59         liftIO $ putStrLn "t <recipient id> Message <msg>      - send message"
60         liftIO $ putStrLn "t <recipient id> Both (<coins>, <msg>) - send both"
61         liftIO $ putStrLn "stake <coins>      - stake coins"
62         liftIO $ putStrLn "view              - view last block"
63         liftIO $ putStrLn "blockchain        - view the entire blockchain"
64         liftIO $ putStrLn "balance           - view account balance"

```

```

65     liftIO $ putStrLn "peers"                -- show list of peers"
66     liftIO $ putStrLn "load <filename>"      -- load transactions from a file"
67     liftIO $ putStrLn "exit"                 -- exit the shell"
68     liftIO $ putStrLn "help"                 -- show this message"
69     _ -> liftIO $ putStrLn "Invalid command. Try entering 'help' for help."

```

Listing 23: CLI.hs

Ο οδηγός όλου του front-end είναι η συνάρτηση `shell`, την οποία καλεί ο κώδικας του `OrdinaryNode.hs`.

```

1  shell shared = do
2    liftIO $ putStrLn "Loading .."
3    liftIO $ putStrLn "Welcome to (the s)hell!"
4    liftIO $ putStrLn "Type 'help' to ask for help."
5  loop
6  where
7    loop :: ReaderT CLIInfo IO ()
8    loop = do
9      liftIO $ putStr "> " >> hFlush stdout
10     input <- liftIO safeGetLine
11     case input of
12       Nothing -> CLI.handle "exit" shared
13       Just line -> CLI.handle line shared >> loop
14
15  -- | Function to safely attempt reading a line, returning Nothing on EOF

```

Listing 24: CLI.hs

Ολόκληρη η εφαρμογή οδηγείται από την `main`.

```

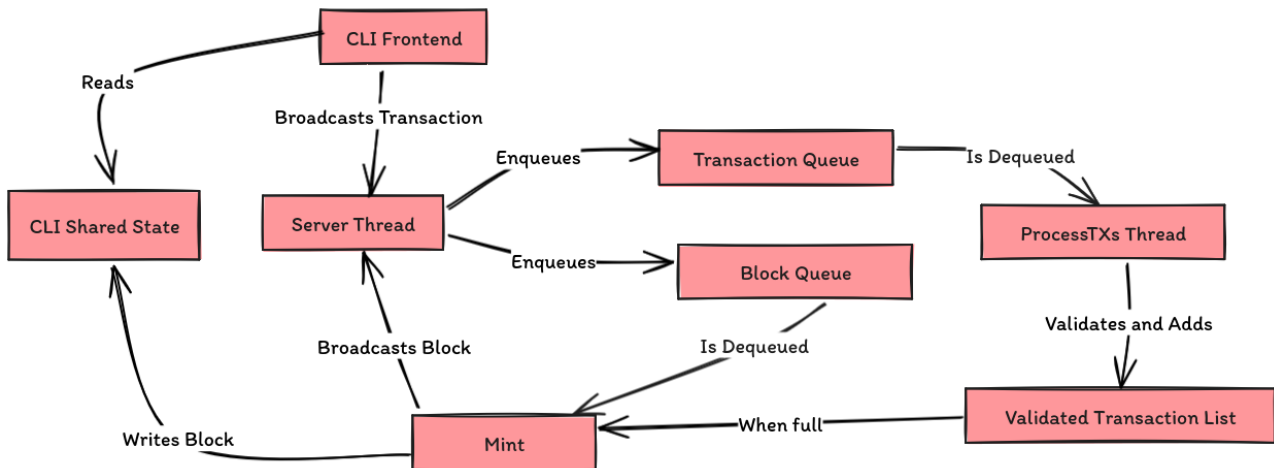
1  main :: IO ()
2  main = getArgs >>= parseArgs >>= startDoingStuff >> exit
3
4  startDoingStuff :: [String] -> IO ()
5  startDoingStuff [host, port, bip, bport, capacity] = void $ do
6    wallet <- generateWallet 2048
7    node (BootstrapNode bip bport) (read capacity) (NodeInfo host port wallet)
8  startDoingStuff [host, port, num] = void $ bootstrapNode (BootInfo 0 host port nodes)
9  where
10    nodes = read num :: Int
11  startDoingStuff _ = usage >> exit
12
13  parseArgs :: [String] -> IO [String]
14  parseArgs ("--node" : restArgs) = help restArgs
15  where
16    help :: [String] -> IO [String]
17    help args | length args == 5 = return args
18    help _ = usage >> exit
19  parseArgs ("--bootstrap" : restArgs) = help restArgs
20  where
21    help :: [String] -> IO [String]
22    help args | length args == 3 = return args
23    help _ = usage >> exit
24  parseArgs _ = usage >> exit
25
26  usage :: IO ()
27  usage =
28    putStrLn "Usage: main --node <ip> <port> <bootstrap ip> <bootstrap port> <capacity>"
29    >> putStrLn "      main --bootstrap <ip> <port> <num nodes>"
30
31  exit :: IO a
32  exit = exitSuccess

```

Listing 25: Main.hs

2.1 Συνολική Εικόνα

Το CLI parseάρει την είσοδο και, καλώντας την `broadcastTransaction`, στέλνει την συναλλαγή σε όλους τους κόμβους του δικτύου. Αυτοί, μέσω της συνάρτησης `server` που τρέχει μονίμως σε ξεχωριστό νήμα, διαχειρίζονται τις συναλλαγές βάζοντάς τις σε μία σειρά αναμονής. Παράλληλα, η `processTXs` αφαιρεί τις συναλλαγές από την ουρά και, αν καταφέρει να τις επικυρώσει, τις προσθέτει σε μία λίστα συναλλαγών που θα περιληφθούν σε ένα block. Όταν αυτή αποκτήσει μήκος `capacity`, τότε καλεί την `mint` για να δημιουργήσει ένα νέο block, το αποστέλλει στο δίκτυο και ενημερώνει το CLI Shared State.



Σχήμα 2: Διάγραμμα Εφαρμογής

Μέρος II

Πειράματα

Ανά πείραμα αξιολογούνται, αφενός τα πιο χρονοβόρα κομμάτια του κώδικα, όπως υποδεικνύει το profiling καθενός κόμβου κατά την εκτέλεση του πειράματος, αφετέρου οι συναρτήσεις `mint`, `validateTransaction` και `processTXs`, οι οποίες συνιστούν την λογική λειτουργίας των κόμβων του συστήματος. Επίσης, εκτιμάται η ρυθμαπόδοση του συστήματος και το μέσο `block time`.

3 Πειραματική Διάταξη

Απουσία ικανοποιητικής υποδομής κατανεμημένων υπολογιστών, τα πειράματα εκτελέστηκαν χρησιμοποιώντας Docker Containers, τα οποία οδηγήθηκαν καταλλήλως από Bash Scripts. Τα containers χτίστηκαν ως εξής:

```
1
2 FROM haskell:9.6.4
3
4 WORKDIR /app
5
6 # Copy the necessary files and directories for the build
7 COPY stack.yaml package.yaml /app/
8
9 # Install dependencies
10 RUN stack setup && stack build --dependencies-only
11
12 # Now copy the rest of the project
13 COPY . /app
14
15 # Reuse .stack-work if possible (consider this based on your CI/CD setup)
16 COPY .stack-work /app/.stack-work
17
18 # Build the project
19 RUN stack build
20
21 ENTRYPOINT ["stack", "exec", "--", "BlockChat-exe"]
22 CMD []
```

Listing 26: Dockerfile

και το script που οδηγεί τα containers:

```
1 #!/bin/bash
2
3 BASE_IP="172.0.0."
4 BASE="0"
5 PORT="36900"
6 IMAGE_NAME="blockchat"
7 PREFIX="experiments/profiled_outputs/docker_harmonic"
8 SUFFIX=""
9 CAPACITY=("5" "10" "20" "1")
10 TESTS=("throughput" "scalability" "fairness")
11 PROFLOG="/app/node"
12
13 for TEST in "${TESTS[@]};
14 do
15     if [ "$TEST" != "scalability" ];
16     then
17         NODES="5"
18     else
19         NODES="10"
20     fi
21     if [ "$TEST" == "fairness" ];
22     then
23         initial_stake ="stake 100"
24     else
```

```

25     initial_stake ="stake 10"
26 fi
27
28 for CAP in "${CAPACITY[@]}";
29 do
30     WORKDIR=$PREFIX/$TEST$SUFFIX/capacity$CAP
31     if [ ! -d "$WORKDIR" ];
32     then
33         mkdir -p "$WORKDIR"
34     fi
35
36     docker network create --subnet=172.0.0.0/16 blockchat-net
37     echo "spawning bootstrap node"
38     docker run \
39         --net blockchat-net \
40         --ip "$BASE_IP$((BASE+2))" \
41         -p $PORT:$PORT \
42         --name bootstrap \
43         $IMAGE_NAME --bootstrap "$BASE_IP$((BASE+2))" $PORT $NODES &
44
45     sleep 2
46
47     docker logs -f bootstrap > "$WORKDIR"/bootstrap.log &
48
49     echo "spawning node 1"
50     input="experiments/input$NODES/trans1.txt"
51     (cat <(echo "$initial_stake") <(echo "load $input") <(echo "blockchain") <(echo "balance"))|\
52     docker run -i \
53         --net blockchat-net \
54         --ip "$BASE_IP$((BASE+2+1))" \
55         -p $((PORT+1)):$PORT \
56         --name node1 \
57         $IMAGE_NAME --node "$BASE_IP$((BASE+2+1))" "$PORT" "$BASE_IP$((BASE+2))" \
58         "$PORT" "$CAP" &
59
60     sleep 2
61     docker logs -f node1 > "$WORKDIR"/node1.log &
62
63     for i in $(seq 2 $NODES);
64     do
65         echo "spawning node ${i}"
66         input="experiments/input$NODES/trans${i}.txt"
67         (cat <(echo "stake 10") <(echo "load $input") <(echo "blockchain") <(echo "balance"))|\
68         docker run -i \
69             --net blockchat-net \
70             --ip "$BASE_IP$((BASE+2+i))" \
71             -p $((PORT+i)):$PORT \
72             --name node"$i" \
73             $IMAGE_NAME --node "$BASE_IP$((BASE+2+i))" "$PORT" "$BASE_IP$((BASE+2))" \
74             "$PORT" "$CAP" &
75
76         sleep 2
77         docker logs -f node"$i" > "$WORKDIR"/node"$i".log &
78     done
79
80     # wait for containers from node1 to node$NODES to finish
81     for i in $(seq 1 $NODES);
82     do
83         echo "waiting for node $i to finish"
84         docker wait node"$i"
85         # get the profiled outputs
86         docker cp node"$i":$PROFLOG.prof "$WORKDIR"/node"$i".prof
87         docker rm node"$i"
88     done
89     docker rm bootstrap
90     docker network rm blockchat-net
91 done

```

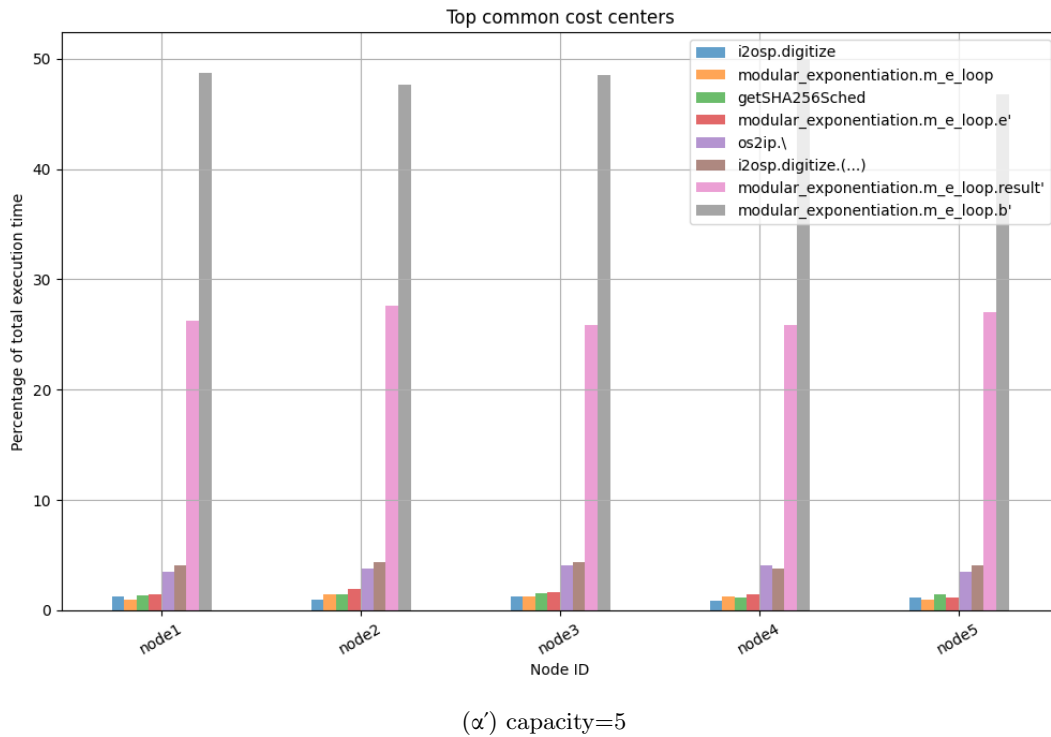
4 Απόδοση του συστήματος

Σημειώνεται ότι το ποσοστό του χρόνου εκτέλεσης των σημείων που υποδεικνύει το profiling δεν είναι κληρονομημένο, δηλαδή δεν εμπεριέχονται στο ποσοστό οι χρόνοι εκτέλεσης των συναρτήσεων που καλούνται από τις συναρτήσεις που εμφανίζονται στο profiling.

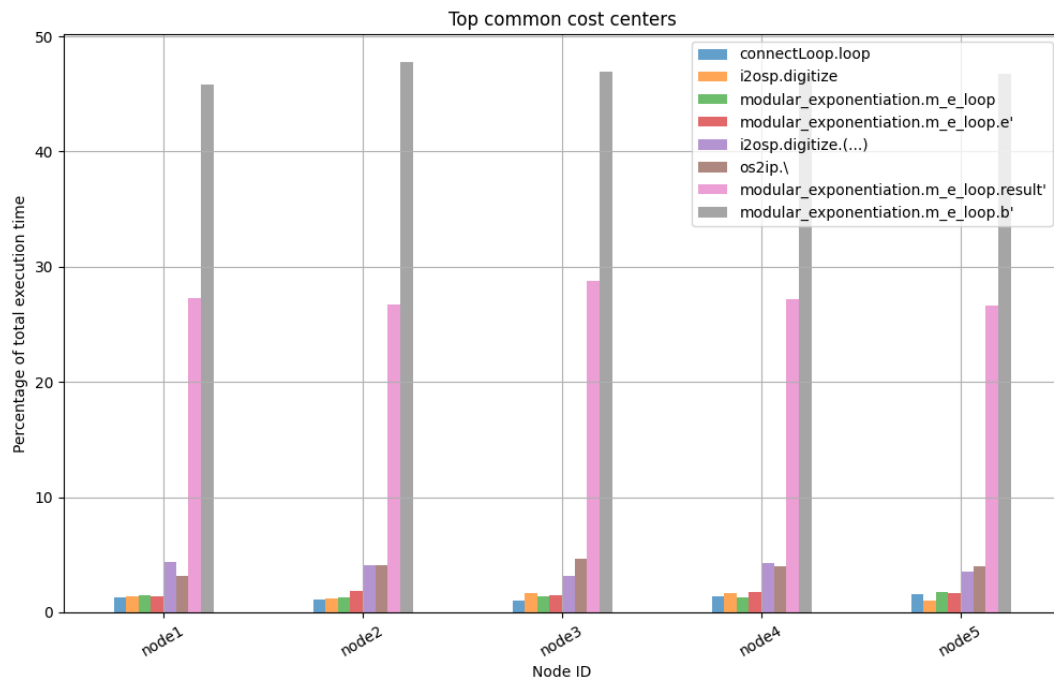
4.1 Χρονοβόρα τμήματα του κώδικα

Στο πείραμα για την αξιολόγηση της ρυθμαπόδοσης του συστήματος, στήνεται ένα δίκτυο 5 κόμβων, καθένας εκ των οποίων εκτελεί 1 staking, με stake 10 BCC συναλλαγή και 50 συναλλαγές (συγκεκριμένα αποστολές μηνυμάτων) προς τους άλλους κόμβους. Η ταχύτητα αποστολής συναλλαγών είναι ίδια μεταξύ των κόμβων, ίση με $2\frac{txs}{s}$ και παραμένει σταθερή μεταξύ όλων των πειραμάτων.

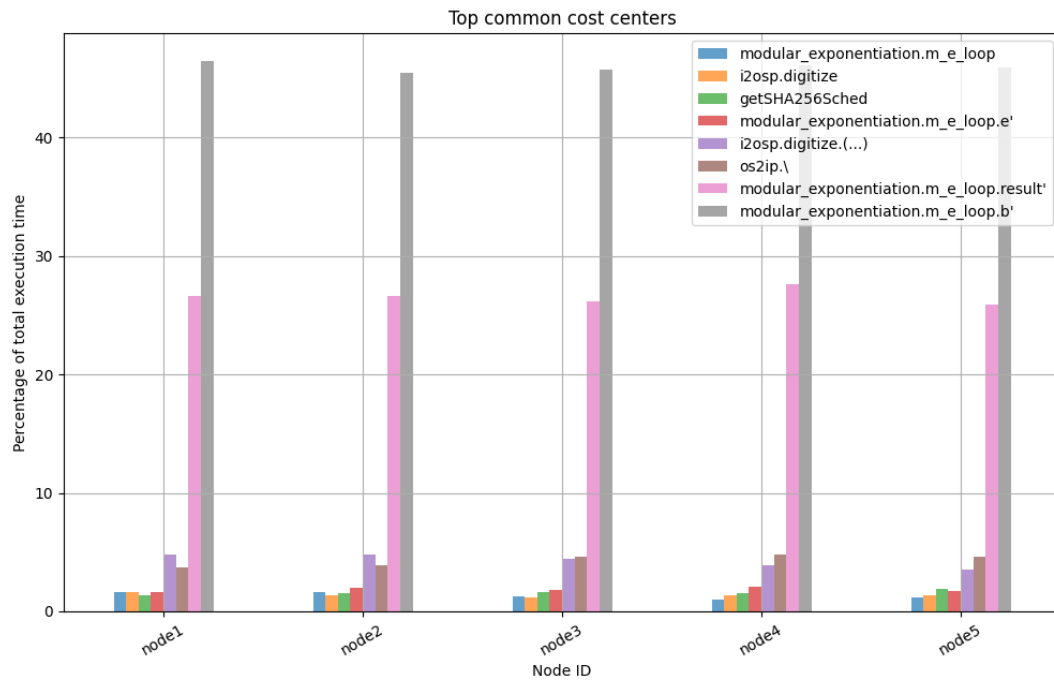
Το πρώτο πράγμα που φαίνεται στο σχήμα 3 είναι ότι το μακράν πιο χρονοβόρο μέρος του κώδικα είναι η συνάρτηση `modular_exponentiation` που χρησιμοποιείται γενικά για την κρυπτογράφηση / αποκρυπτογράφηση και υπογραφή / επαλήθευση μηνυμάτων. Συγκεκριμένα, φαίνεται να λαμβάνει περίπου το 45% του συνολικού χρόνου υπολογισμού¹ του προγράμματος.



¹Ο profiler της Haskell μετράει *CPU time* όχι *blocking time*



(β') capacity=10

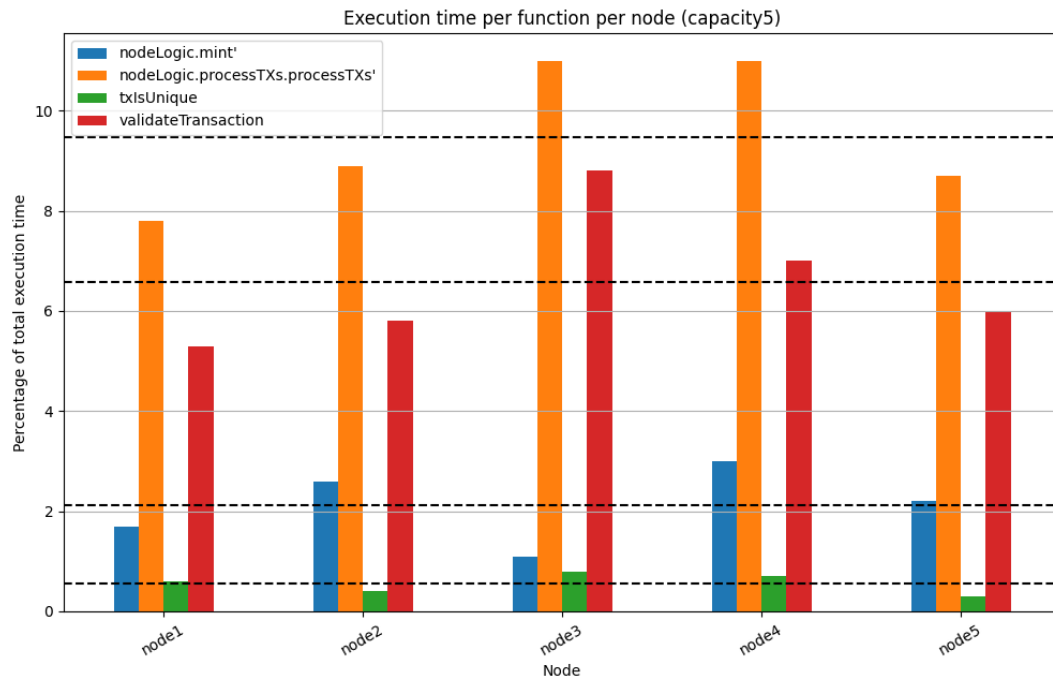


(γ') capacity=20

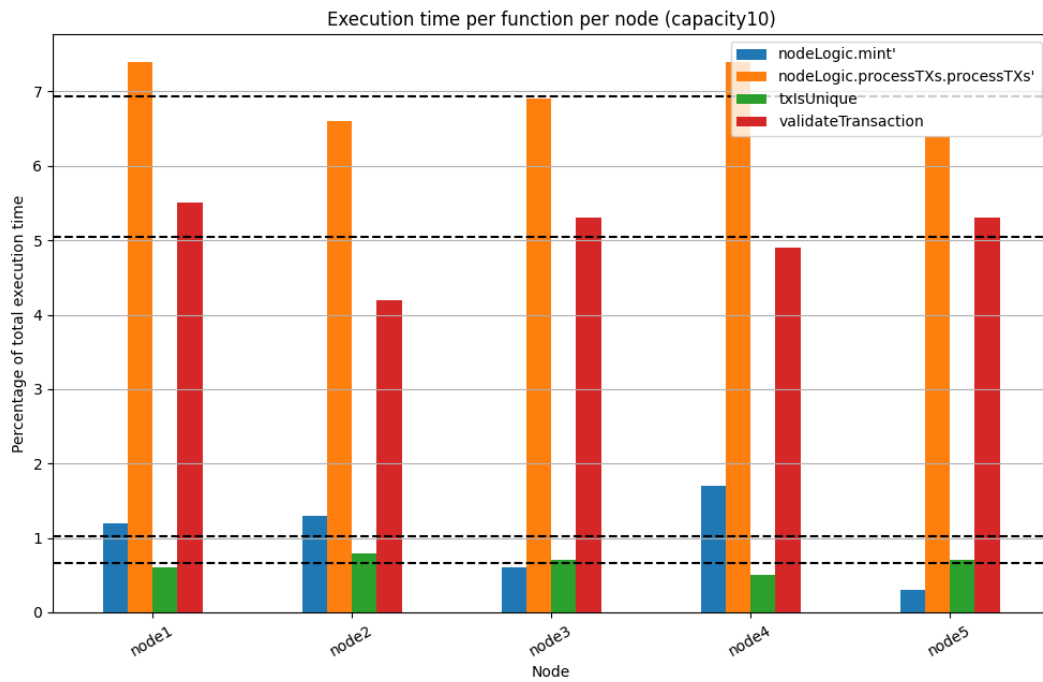
Σχήμα 3: Τα πιο χρονοβόρα κομμάτια του κώδικα

4.2 Συναρτήσεις του συστήματος

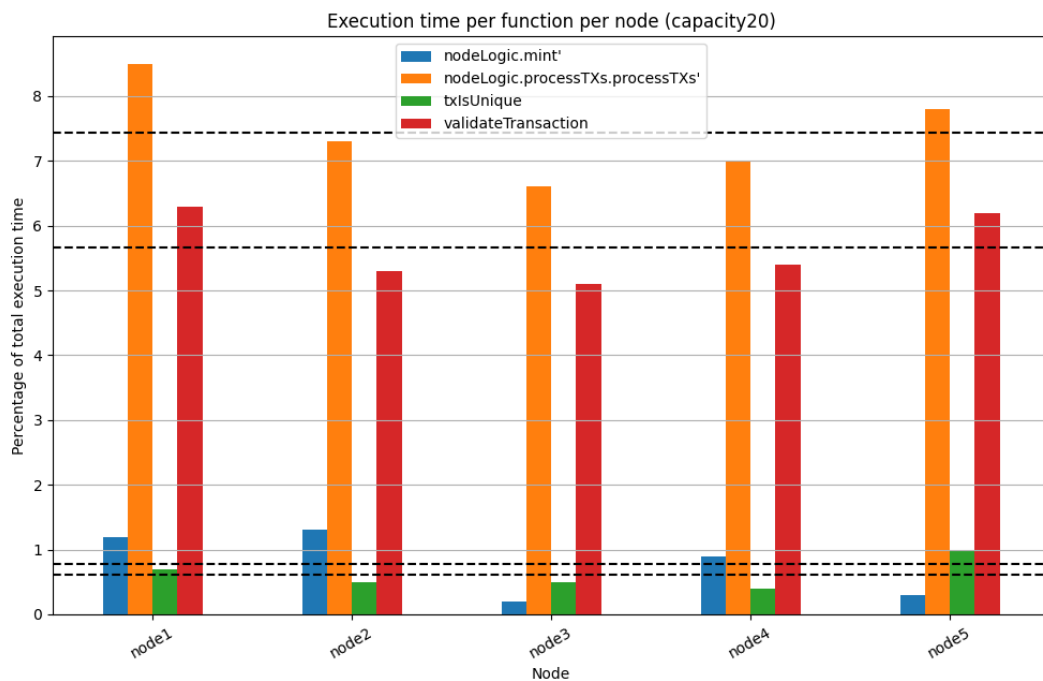
Σχετικά με τις top level συναρτήσεις του συστήματος, παρατηρείται ότι, με κάποιες μικρές διακυμάνσεις, η processTXs καταναλώνει 9-10% του συνολικού CPU time, η mint 1-3% και η validateTransaction 5-9%, με μέσο όρο περίπου 6.5%.



(α') Ρυθμαπόδοση capacity=5



(β') Ρυθμαπόδοση capacity=10



(γ') Ρυθμαπόδοση capacity=20

Σχήμα 4: Ποσοστό χρόνου επί του συνολικού χρόνου εκτέλεσης που λαμβάνει η κάθε συνάρτηση

Στον πίνακα 1 παρουσιάζονται ορισμένα στατιστικά σχετικά με τις συναρτήσεις `processTXs`, `validateTransaction`, `txIsUnique` και `mint`. Το πιο σημαντικό να παρατηρηθεί είναι ότι, για κάθε κόμβο, οι κλήσεις στην συνάρτηση `validateTransaction` είναι ακριβώς τόσες όσες και οι συναλλαγές που αποστέλλονται από όλους τους κόμβους ($5 + 5 \times 50 = 255$). Επίσης, $\#mint + \#validateTransaction = \#processTXs$ ². Παρότι φαίνεται σαν να επικυρώθηκαν όλες οι συναλλαγές, αυτό δεν ισχύει. Στην πραγματικότητα, επειδή οι κόμβοι δεν παραλαμβάνουν κατ' ανάγκην τις συναλλαγές με την σειρά αποστολή τους, είναι πιθανό κάποιος validator να ακυρώσει κάποια συναλλαγή η οποία με διαφορετική σειρά θα είχε επιβεβαιωθεί. Για αυτόν τον λόγο φαίνεται ότι ένα υποσύνολο των συναλλαγών εξετάζεται για την μοναδικότητά τους από την συνάρτηση `txIsUnique`. Έτσι αιτιολογείται και το γεγονός ότι το blockchain έχει μήκος μικρότερο από το μέγιστο δυνατό του δεδομένων των συναλλαγών. Παραδείγματος χάριν, για capacity 5 έχει μήκος $41 = \frac{207}{5} < \frac{255}{5} = 51$.

Πίνακας 1: Στατιστικά συναρτήσεων ανά κόμβο

(α') capacity=5

Node	Function	Entries	TimeInh
node1.prof:	nodeLogic.processTXs.processTXs'	301	7.3
node1.prof:	validateTransaction	255	5.0
node1.prof:	txIsUnique	228	0.5
node1.prof:	nodeLogic.mint'	45	1.7
node2.prof:	nodeLogic.processTXs.processTXs'	301	6.3
node2.prof:	validateTransaction	255	5.0
node2.prof:	txIsUnique	229	0.4
node2.prof:	nodeLogic.mint'	45	0.9
node3.prof:	nodeLogic.processTXs.processTXs'	301	7.6
node3.prof:	validateTransaction	255	5.5
node3.prof:	txIsUnique	225	0.6
node3.prof:	nodeLogic.mint'	45	1.4
node4.prof:	nodeLogic.processTXs.processTXs'	301	6.7
node4.prof:	validateTransaction	255	4.9
node4.prof:	txIsUnique	226	0.6
node4.prof:	nodeLogic.mint'	45	1.1
node5.prof:	nodeLogic.processTXs.processTXs'	301	8.7
node5.prof:	validateTransaction	255	6.2
node5.prof:	txIsUnique	228	0.6
node5.prof:	nodeLogic.mint'	45	1.8

Όπως φαίνεται από τον πίνακα, λοιπόν, για τον υπολογισμό του block time και της ρυθμαπόδοσης λαμβάνονται υπόψιν τόσα μπλοκς όσα και οι κλήσεις στην συνάρτηση `mint` και τόσες συναλλαγές όσες τα μπλοκς επί την εκάστοτε χωρητικότητα.

$$\text{Χωρητικότητα} = 5 \Rightarrow \text{Μπλοκς} = 45 \text{ και Συναλλαγές} = 225$$

$$\text{Χωρητικότητα} = 10 \Rightarrow \text{Μπλοκς} = 22 \text{ και Συναλλαγές} = 220$$

$$\text{Χωρητικότητα} = 20 \Rightarrow \text{Μπλοκς} = 11 \text{ και Συναλλαγές} = 220$$

(1)

²H -1 διαφορά είναι επειδή έγινε η τελευταία κλήση και τα προγράμματα έλαβαν σήμα τερματισμού

(β') capacity=10				(γ') capacity=20	
Node	Function	Entries	TimeInh	Entries	TimeInh
node1.prof:	nodeLogic.processTXs.processTXs'	278	7.4	267	8.5
node1.prof:	validateTransaction	255	5.5	255	6.3
node1.prof:	txIsUnique	228	0.6	229	0.7
node1.prof:	nodeLogic.mint'	22	1.2	11	1.2
node2.prof:	nodeLogic.processTXs.processTXs'	278	6.6	267	7.3
node2.prof:	validateTransaction	255	4.2	255	5.3
node2.prof:	txIsUnique	229	0.8	229	0.5
node2.prof:	nodeLogic.mint'	22	1.3	11	1.3
node3.prof:	nodeLogic.processTXs.processTXs'	278	6.9	267	6.6
node3.prof:	validateTransaction	255	5.3	255	5.1
node3.prof:	txIsUnique	227	0.7	223	0.5
node3.prof:	nodeLogic.mint'	22	0.6	11	0.2
node4.prof:	nodeLogic.processTXs.processTXs'	278	7.4	267	7.0
node4.prof:	validateTransaction	255	4.9	255	5.4
node4.prof:	txIsUnique	226	0.5	227	0.4
node4.prof:	nodeLogic.mint'	22	1.7	11	0.9
node5.prof:	nodeLogic.processTXs.processTXs'	278	6.4	267	7.8
node5.prof:	validateTransaction	255	5.3	255	6.2
node5.prof:	txIsUnique	227	0.7	229	1.0
node5.prof:	nodeLogic.mint'	22	0.3	11	0.3

4.3 Ρυθμαπόδοση και Block time

Το block time μπορεί να υπολογιστεί λαμβάνοντας τον μέσο όρο των διαφορών των time stamps διαδοχικών blocks. Στο σχήμα 5 φαίνονται οι χρόνοι δημιουργίας block όπως υπολογίστηκαν από κάθε κόμβο. Παρατηρείται ότι δεν είναι πάντοτε ίσοι μεταξύ των κόμβων. Αυτό συμβαίνει γιατί, κατά τον τερματισμό του πειράματος, δεν έχουν φτάσει κατ'ανάγκη όλοι οι κόμβοι στο ίδιο σημείο της αλυσίδας και για αυτόν τον λόγο διαφοροποιείται η μέτρησή τους. Εδώ λαμβάνεται υπόψη το μέγιστο μήκος της αλυσίδας μεταξύ των κόμβων.

```
capacity5/node1.log:Mean time between blocks: 2.418347838929835e-2
capacity5/node2.log:Mean time between blocks: 2.418347838929835e-2
capacity5/node3.log:Mean time between blocks: 2.418347838929835e-2
capacity5/node4.log:Mean time between blocks: 2.418347838929835e-2
capacity5/node5.log:Mean time between blocks: 2.418347838929835e-2

capacity10/node1.log:Mean time between blocks: 6.256585902294834e-2
capacity10/node2.log:Mean time between blocks: 6.256585902294834e-2
capacity10/node3.log:Mean time between blocks: 6.256585902294834e-2
capacity10/node4.log:Mean time between blocks: 6.256585902294834e-2
capacity10/node5.log:Mean time between blocks: 6.256585902294834e-2

capacity20/node1.log:Mean time between blocks: 0.12148699044672626
capacity20/node2.log:Mean time between blocks: 0.12148699044672626
capacity20/node3.log:Mean time between blocks: 0.12148699044672626
capacity20/node4.log:Mean time between blocks: 0.12148699044672626
capacity20/node5.log:Mean time between blocks: 0.12148699044672626
```

Σχήμα 5: Μέσος χρόνος δημιουργίας block (ms)

Από τον χρόνο αυτόν μπορούν να μετρηθούν και οι εξυπηρετούμενες συναλλαγές ανά δευτερόλεπτο. Συγκεκριμένα, μία συναλλαγή εξυπηρετείται όταν επικυρωθεί, δηλαδή όταν καταγραφεί στην αλυσίδα. Άρα, για κάθε capacity έχουμε:

$$\begin{aligned}
\text{Ρυθμαπόδοση} &= \frac{\text{Συναλλαγές}}{\text{Χρόνος}} = \frac{\text{Συναλλαγές}}{\text{Μπλοκ}} \cdot \frac{\text{Μπλοκ}}{\text{Χρόνος}} \Leftrightarrow \\
\text{Ρυθμαπόδοση} &= \frac{\frac{\text{Συναλλαγές}}{\text{Μπλοκ}}}{\text{Μέσος χρόνος δημιουργίας μπλοκ}} = \frac{\text{Χωρητικότητα}}{\text{Μέσος χρόνος δημιουργίας μπλοκ}} \Rightarrow
\end{aligned}$$

$$\begin{aligned}
\text{Ρυθμαπόδοση}_{\text{capacity}=5} &= \frac{5}{0,024} = 208,333 \frac{txs}{s} \\
\text{Ρυθμαπόδοση}_{\text{capacity}=10} &= \frac{10}{0,0626} = 159,744 \frac{txs}{s} \\
\text{Ρυθμαπόδοση}_{\text{capacity}=20} &= \frac{20}{0,121} = 165,289 \frac{txs}{s}
\end{aligned}$$

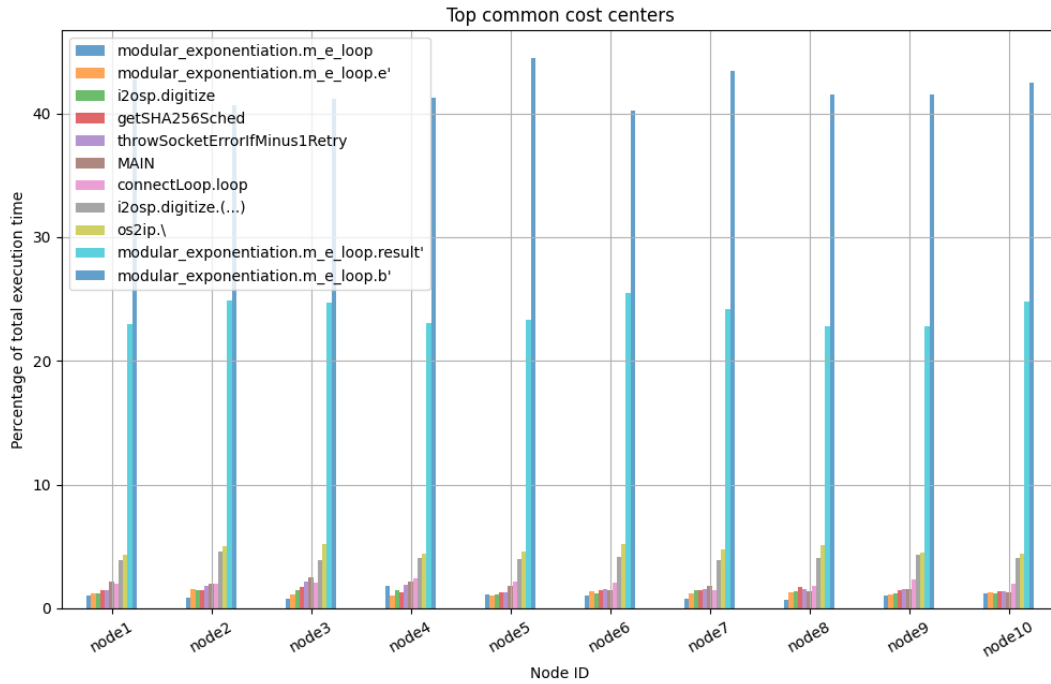
(2)

5 Κλιμακωσιμότητα του συστήματος

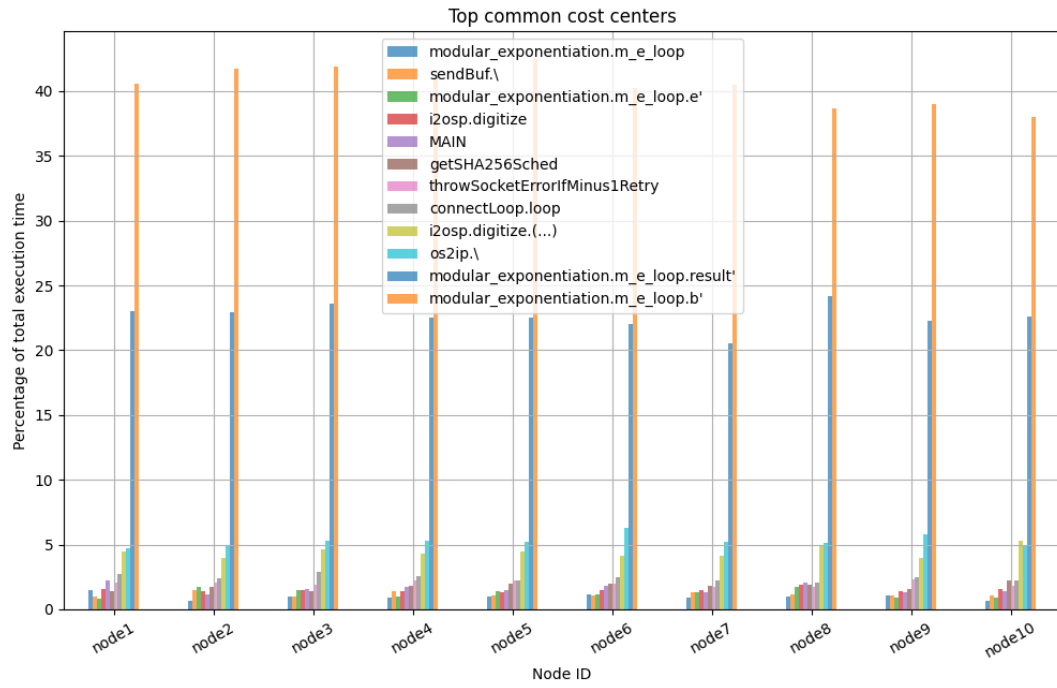
Στο πείραμα κλιμακωσιμότητας, το δίκτυο εκκινείται με 10 κόμβους, καθένας εκ των οποίων εκτελεί 1 staking συναλλαγή, με stake 10 BCC και 100 συναλλαγές (συγκεκριμένα αποστολές μηνυμάτων) προς τους άλλους κόμβους. Σκοπός είναι να εξεταστεί η κλιμάκωση του συστήματος ως προς το πλήθος των συμμετεχόντων κόμβων.

5.1 Χρονοβόρα τμήματα του κώδικα

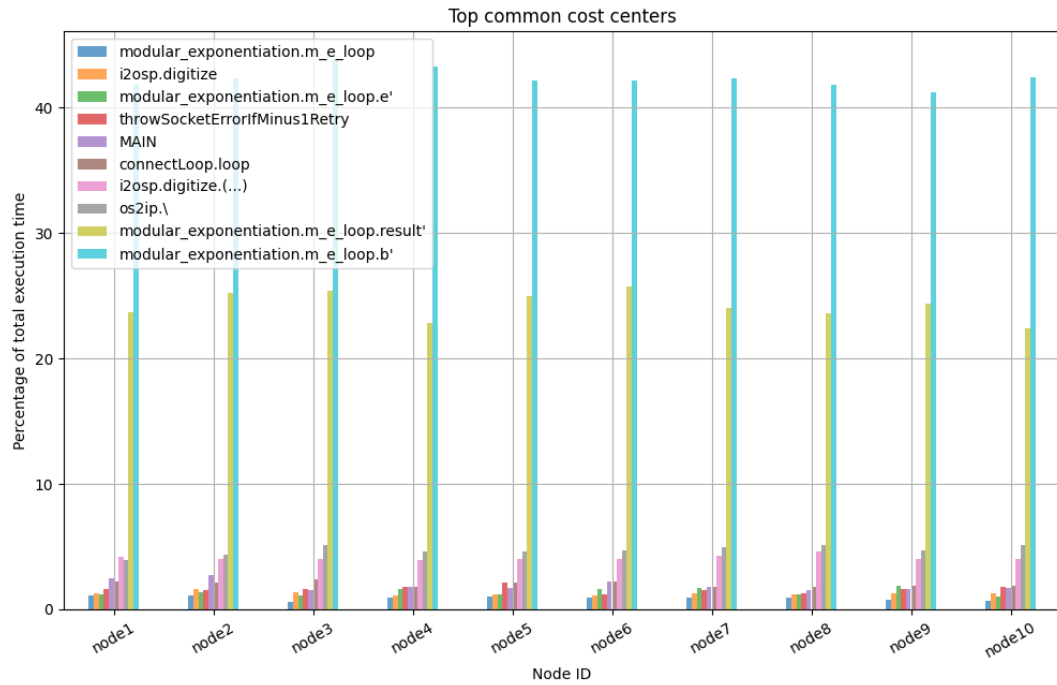
Στα γραφήματα 6 φαίνονται τα πιο χρονοβόρα κομμάτια του κώδικα για κάθε πείραμα κλιμακωσιμότητας. Φαίνεται ότι αυτά είναι τα ίδια με τα πιο χρονοβόρα κομμάτια του κώδικα για το πείραμα ρυθμαπόδοσης, με την συνάρτηση modular_exponentiation να καταλαμβάνει πάλι περίπου το 45% του συνολικού CPU time του προγράμματος.



(α') capacity=5



(β') capacity=10

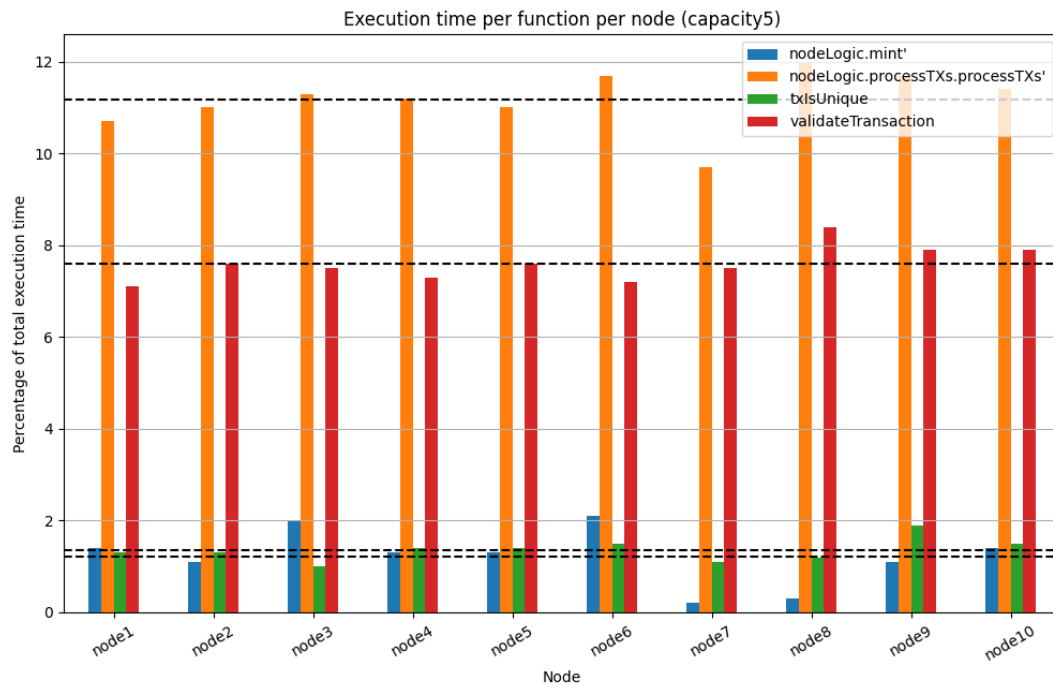


(γ') capacity=20

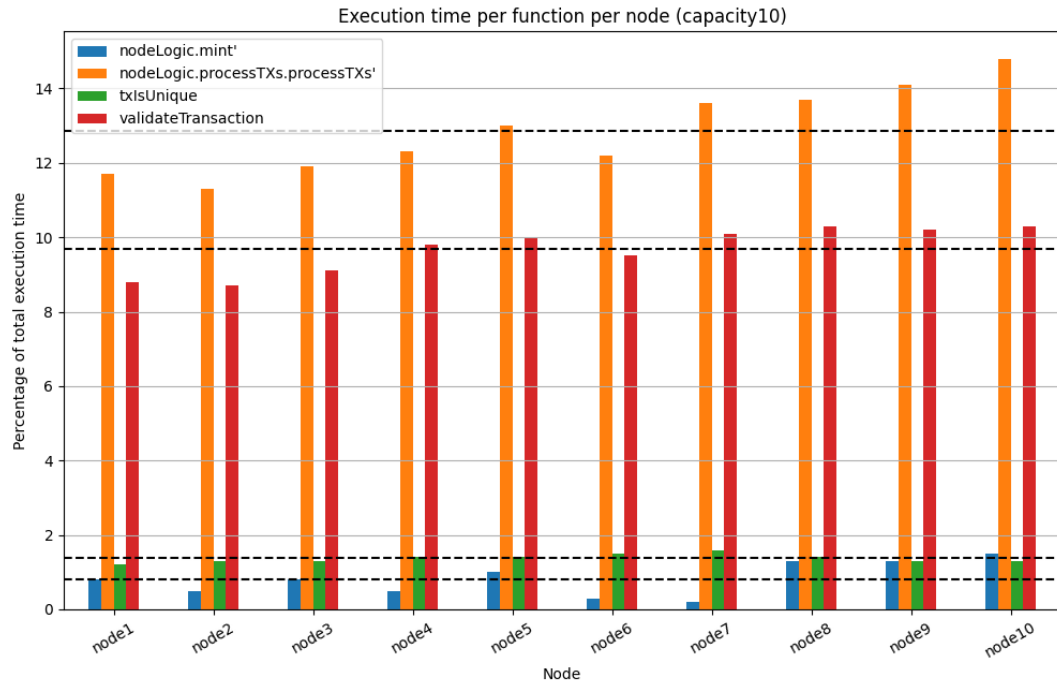
Σχήμα 6: Τα πιο χρονοβόρα κομμάτια του κώδικα

5.2 Συναρτήσεις του συστήματος

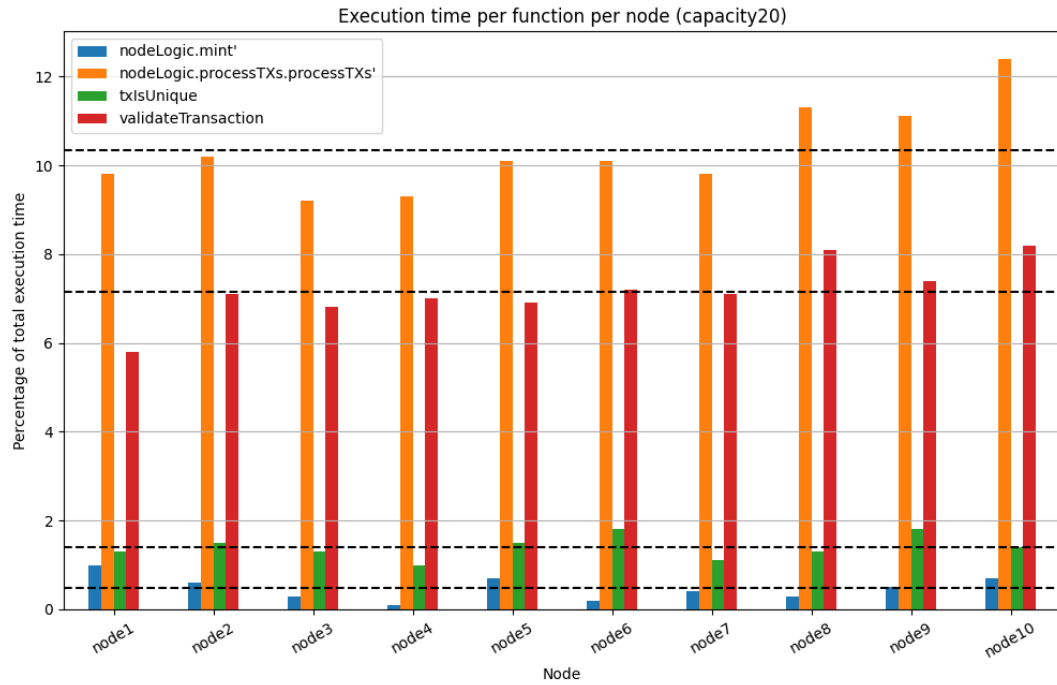
Στα γραφήματα 7 παρατηρείται ότι οι συναρτήσεις καταλαμβάνουν περίπου το ίδιο ποσοστό χρόνου εκτέλεσης με το προηγούμενο πείραμα.



(α') capacity=5



(β') capacity=10



(γ') capacity=20

Σχήμα 7: Ποσοστό χρόνου επί του συνολικού χρόνου εκτέλεσης που λαμβάνει η κάθε συνάρτηση

Στον πίνακα 2α' φαίνονται οι κλήσεις ενδιαφέροντος των κόμβων. Παρατηρώντας το πλήθος των κλήσεων ανά συνάρτηση, διαπιστώνεται ότι οι κόμβοι δεν προλαβαίνουν να επικυρώσουν όλες τις συναλλαγές που λαμβάνουν. Αυτό οφείλεται, αφενός στον μεγαλύτερο όγκο συναλλαγών $10 + 10 \times 100 = 1010$ ο οποίος είναι ≈ 4 φορές μεγαλύτερος από προηγουμένως, και αφετέρου στην μικρή χωρητικότητα του block, το οποίο σημαίνει ότι οι κόμβοι πρέπει συχνά να καλούν την χρονοβόρα συνάρτηση mint και να διακόπτουν την διαδικασία επικύρωσης.

Επίσης, παρατηρείται ότι οι κόμβοι 7-8 έχουν μείνει πολύ πίσω σε σχέση με τους υπόλοιπους κόμβους. Αυτό είναι μάλλον συνέπεια της πειραματικής διάταξης, αφού όλοι οι κόμβοι τρέχουν στο ίδιο μηχάνημα.

Πίνακας 2: Στατιστικά συναρτήσεων ανά κόμβο

(α') capacity=5

Node	Function	Entries	TimeInh
node10.prof:	nodeLogic.processTXs.processTXs'	1111	11.4
node10.prof:	validateTransaction	1010	7.9
node10.prof:	txIsUnique	545	1.5
node10.prof:	nodeLogic.mint'	100	1.4
node1.prof:	nodeLogic.processTXs.processTXs'	1111	10.7
node1.prof:	validateTransaction	1010	7.1
node1.prof:	txIsUnique	544	1.3
node1.prof:	nodeLogic.mint'	100	1.4
node2.prof:	nodeLogic.processTXs.processTXs'	1111	11.0
node2.prof:	validateTransaction	1010	7.6
node2.prof:	txIsUnique	542	1.3
node2.prof:	nodeLogic.mint'	100	1.1
node3.prof:	nodeLogic.processTXs.processTXs'	1111	11.3
node3.prof:	validateTransaction	1010	7.5
node3.prof:	txIsUnique	544	1.0
node3.prof:	nodeLogic.mint'	100	2.0
node4.prof:	nodeLogic.processTXs.processTXs'	1111	11.2
node4.prof:	validateTransaction	1010	7.3
node4.prof:	txIsUnique	544	1.4
node5.prof:	nodeLogic.processTXs.processTXs'	1111	11.0
node5.prof:	validateTransaction	1010	7.6
node5.prof:	txIsUnique	543	1.4
node5.prof:	nodeLogic.mint'	100	1.3
node6.prof:	nodeLogic.processTXs.processTXs'	1111	11.7
node6.prof:	validateTransaction	1010	7.2
node6.prof:	txIsUnique	542	1.5
node6.prof:	nodeLogic.mint'	100	2.1
node7.prof:	nodeLogic.processTXs.processTXs'	1101	9.7
node7.prof:	validateTransaction	1010	7.5
node7.prof:	txIsUnique	495	1.1
node7.prof:	nodeLogic.mint'	90	0.2
node8.prof:	nodeLogic.processTXs.processTXs'	1101	12.0
node8.prof:	validateTransaction	1010	8.4
node8.prof:	txIsUnique	496	1.2
node8.prof:	nodeLogic.mint'	90	0.3
node9.prof:	nodeLogic.processTXs.processTXs'	1111	11.7
node9.prof:	validateTransaction	1010	7.9
node9.prof:	txIsUnique	546	1.9
node9.prof:	nodeLogic.mint'	100	1.1

Αντιθέτως, στους πίνακες 2β' και 2γ' φαίνεται από τις κλήσεις των συναρτήσεων ότι έχουν επικυρωθεί όλες οι συναλλαγές και έχουν παραχθεί τα αντίστοιχα blocks. Η μεγαλύτερη χωρητικότητα των blocks επιτρέπει στους κόμβους μεγαλύτερα χρονικά παράθυρα για την επικύρωση των συναλλαγών και η διακοπή για την παραγωγή των blocks δεν καθυστερεί την εξέλιξη του δικτύου.

(β') capacity=10		(γ') capacity=20			
Node	Function	Entries	TimeInh	Entries	TimeInh
node10.prof:	nodeLogic.processTXs.processTXs'	1061	14.8	1024	12.4
node10.prof:	validateTransaction	1010	10.3	999	8.2
node10.prof:	txIsUnique	503	1.3	501	1.4
node10.prof:	nodeLogic.mint'	50	1.5	25	0.7
node1.prof:	nodeLogic.processTXs.processTXs'	1061	11.7	986	9.8
node1.prof:	validateTransaction	1010	8.8	961	5.8
node1.prof:	txIsUnique	503	1.2	500	1.3
node1.prof:	nodeLogic.mint'	50	0.8	25	1.0
node2.prof:	nodeLogic.processTXs.processTXs'	1061	11.3	992	10.2
node2.prof:	validateTransaction	1010	8.7	967	7.1
node2.prof:	txIsUnique	504	1.3	500	1.5
node2.prof:	nodeLogic.mint'	50	0.5	25	0.6
node3.prof:	nodeLogic.processTXs.processTXs'	1061	11.9	968	9.2
node3.prof:	validateTransaction	1010	9.1	943	6.8
node3.prof:	txIsUnique	505	1.3	500	1.3
node3.prof:	nodeLogic.mint'	50	0.8	25	0.3
node4.prof:	nodeLogic.processTXs.processTXs'	1061	12.3	1033	9.3
node4.prof:	validateTransaction	1010	9.8	1010	7.0
node4.prof:	txIsUnique	506	1.4	454	1.0
node4.prof:	nodeLogic.mint'	50	0.5	22	0.1
node5.prof:	nodeLogic.processTXs.processTXs'	1061	13.0	990	10.1
node5.prof:	validateTransaction	1010	10.0	965	6.9
node5.prof:	txIsUnique	505	1.4	500	1.5
node5.prof:	nodeLogic.mint'	50	1.0	25	0.7
node6.prof:	nodeLogic.processTXs.processTXs'	1061	12.2	1035	10.1
node6.prof:	validateTransaction	1010	9.5	1010	7.2
node6.prof:	txIsUnique	504	1.5	497	1.8
node6.prof:	nodeLogic.mint'	50	0.3	24	0.2
node7.prof:	nodeLogic.processTXs.processTXs'	1061	13.6	997	9.8
node7.prof:	validateTransaction	1010	10.1	972	7.1
node7.prof:	txIsUnique	500	1.6	500	1.1
node7.prof:	nodeLogic.mint'	50	0.2	25	0.4
node8.prof:	nodeLogic.processTXs.processTXs'	1061	13.7	1035	11.3
node8.prof:	validateTransaction	1010	10.3	1010	8.1
node8.prof:	txIsUnique	505	1.4	498	1.3
node8.prof:	nodeLogic.mint'	50	1.3	24	0.3
node9.prof:	nodeLogic.processTXs.processTXs'	1061	14.1	986	11.1
node9.prof:	validateTransaction	1010	10.2	961	7.4
node9.prof:	txIsUnique	505	1.3	501	1.8
node9.prof:	nodeLogic.mint'	50	1.3	25	0.5

Για την μέτρηση του block time και της ρυθμαπόδοσης του συστήματος, λαμβάνονται υπόψιν τόσα μπλοκ όσα και οι κλήσεις στην συνάρτηση mint και τόσες συναλλαγές όσα τα μπλοκ επί την εκάστοτε χωρητικότητα.

$$\text{Χωρητικότητα} = 5 \Rightarrow \text{Μπλοκ} = 100 \text{ και } \text{Συναλλαγές} = 500$$

$$\text{Χωρητικότητα} = 10 \Rightarrow \text{Μπλοκ} = 50 \text{ και } \text{Συναλλαγές} = 500$$

$$\text{Χωρητικότητα} = 20 \Rightarrow \text{Μπλοκ} = 25 \text{ και } \text{Συναλλαγές} = 500$$

(3)

5.3 Ρυθμαπόδοση και Block time

Στο σχήμα 8 φαίνονται οι μέσοι χρόνοι³ δημιουργίας block όπως υπολογίστηκαν από κάθε κόμβο⁴.

```
capacity5/node10.log:Mean time between blocks: 4.044146914918912e-2
capacity5/node1.log:Mean time between blocks: 3.496434225324504e-2
capacity5/node2.log:Mean time between blocks: 3.496434225324504e-2
capacity5/node3.log:Mean time between blocks: 3.496434225324504e-2
capacity5/node4.log:Mean time between blocks: 4.044146914918912e-2
capacity5/node5.log:Mean time between blocks: 4.044146914918912e-2
capacity5/node6.log:Mean time between blocks: 4.044146914918912e-2
capacity5/node7.log:Mean time between blocks: 4.044146914918912e-2
capacity5/node8.log:Mean time between blocks: 4.044146914918912e-2
capacity5/node9.log:Mean time between blocks: 4.044146914918912e-2

capacity10/node10.log:Mean time between blocks: 0.1097684269536267
capacity10/node1.log:Mean time between blocks: 0.10784336631643372
capacity10/node2.log:Mean time between blocks: 0.1097684269536267
capacity10/node3.log:Mean time between blocks: 0.1097684269536267
capacity10/node4.log:Mean time between blocks: 0.1097684269536267
capacity10/node5.log:Mean time between blocks: 0.10784336631643372
capacity10/node6.log:Mean time between blocks: 0.10784336631643372
capacity10/node7.log:Mean time between blocks: 0.1097684269536267
capacity10/node8.log:Mean time between blocks: 0.1097684269536267
capacity10/node9.log:Mean time between blocks: 0.1097684269536267

capacity20/node10.log:Mean time between blocks: 0.10954314829501743
capacity20/node1.log:Mean time between blocks: 0.10954314829501743
capacity20/node2.log:Mean time between blocks: 0.10584314117266758
capacity20/node3.log:Mean time between blocks: 0.10954314829501743
capacity20/node4.log:Mean time between blocks: 0.10954314829501743
capacity20/node5.log:Mean time between blocks: 0.10954314829501743
capacity20/node6.log:Mean time between blocks: 0.10954314829501743
capacity20/node7.log:Mean time between blocks: 0.10954314829501743
capacity20/node8.log:Mean time between blocks: 9.8183030541365e-2
capacity20/node9.log:Mean time between blocks: 0.10954314829501743
```

Σχήμα 8: Μέσος χρόνος δημιουργίας block (ms)

Πάλι, από τους χρόνους αυτούς μπορούν να μετρηθούν και οι εξυπηρετούμενες συναλλαγές ανά δευτερόλεπτο. Για κάθε capacity έχουμε:

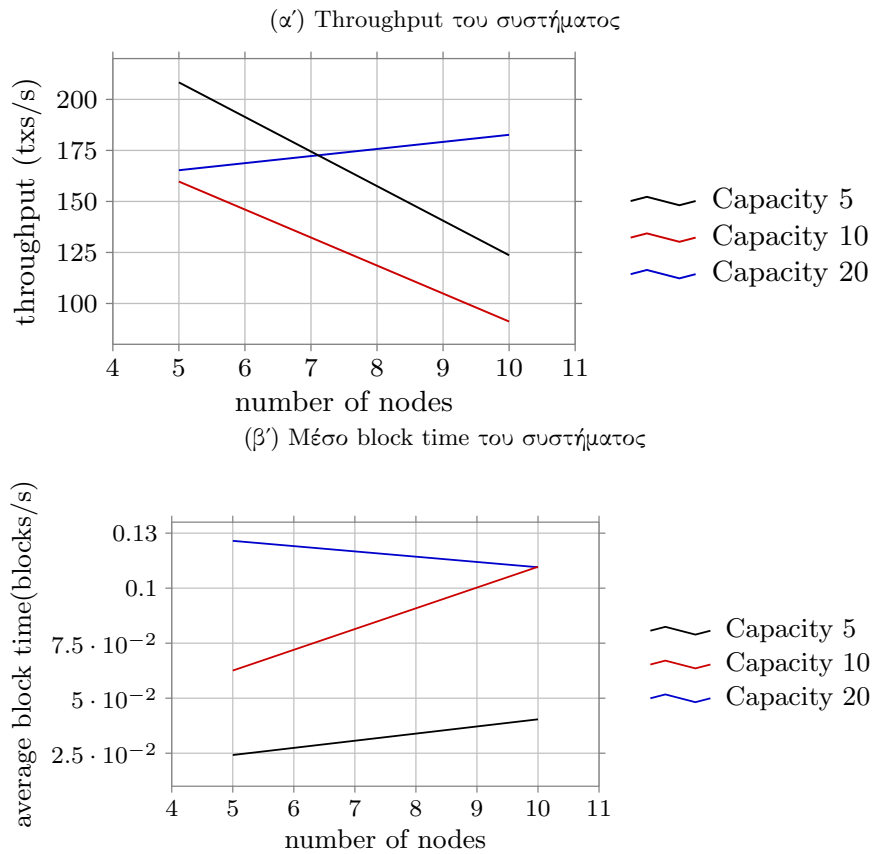
$$\begin{aligned} \text{Ρυθμαπόδοση} &= \frac{\text{Συναλλαγές}}{\text{Χρόνος}} = \frac{\text{Συναλλαγές}}{\text{Μπλοκ}} \cdot \frac{\text{Μπλοκ}}{\text{Χρόνος}} \Leftrightarrow \\ \text{Ρυθμαπόδοση} &= \frac{\frac{\text{Συναλλαγές}}{\text{Μπλοκ}}}{\text{Μέσος χρόνος δημιουργίας μπλοκ}} = \frac{\text{Χωρητικότητα}}{\text{Μέσος χρόνος δημιουργίας μπλοκ}} \Rightarrow \end{aligned} \quad (4)$$

$\text{Ρυθμαπόδοση}_{\text{capacity}=5} = \frac{5}{0,0404} = 123,639 \frac{txs}{s}$ $\text{Ρυθμαπόδοση}_{\text{capacity}=10} = \frac{10}{0.1097} = 91,157 \frac{txs}{s}$ $\text{Ρυθμαπόδοση}_{\text{capacity}=20} = \frac{20}{0.1095} = 182,648 \frac{txs}{s}$
--

Οι μετρήσεις από τα πειράματα συνοψίζονται στα γραφήματα 9.

³ Αρμονικός μέσος.

⁴ Οι χρόνοι αυτοί λήφθηκαν χωρίς profiling ενεργοποιημένο, ώστε να μην υπάρχουν παρεμβολές. Τα προηγούμενα στοιχεία profiling, δηλαδή το πλήθος κλήσεων σε διάφορες συναρτήσεις, παρουσιάστηκαν ως ενδεικτικά για την συμπεριφορά των κόμβων.

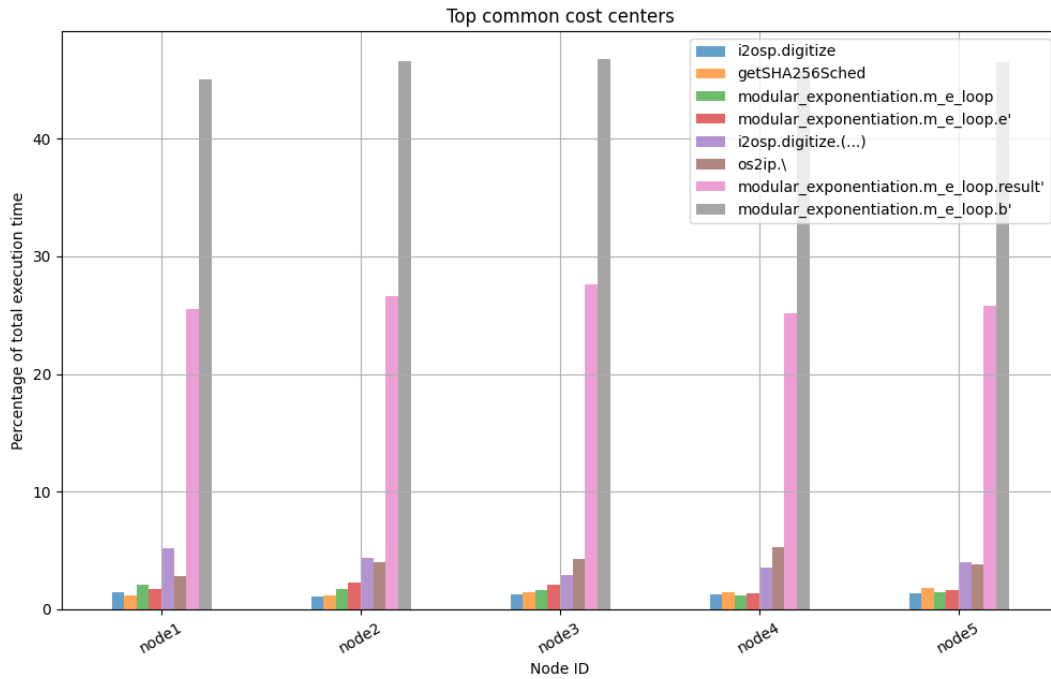


Σχήμα 9: Ρυθμαπόδοση και μέσος block time του συστήματος

Στα γραφήματα 9α' και 9β' φαίνεται η ρυθμαπόδοση και το μέσο block time του συστήματος μεταξύ των πειραμάτων, από 5 έως 10 κόμβους, για κάθε χωρητικότητα. Φαίνεται, ότι το πλήθος των εξυπηρετούμενων συναλλαγών ανά μονάδα χρόνου αυξάνεται με τον αριθμό των κόμβων για την χωρητικότητα 20, αλλά για την 5 και 10 ελαττώνεται. Ο δε μέσος χρόνος δημιουργίας block μειώνεται με τον αριθμό των κόμβων για την χωρητικότητα 20, αλλά για την 5 και 10 έχει αύξουσα τάση. Αυτό δείχνει ότι το δίκτυο μπορεί να κλιμακώνει με τον αριθμό των κόμβων για χωρητικότητα 20.

6 Δικαιοσύνη

Στο πείραμα δικαιοσύνης, το δίκτυο εκκινείται με 5 κόμβους και ο υπάριθμὸν 1 από αυτούς κάνει stake 100 BCC, ενώ οι υπόλοιποι κάνουν stake 10 BCC.

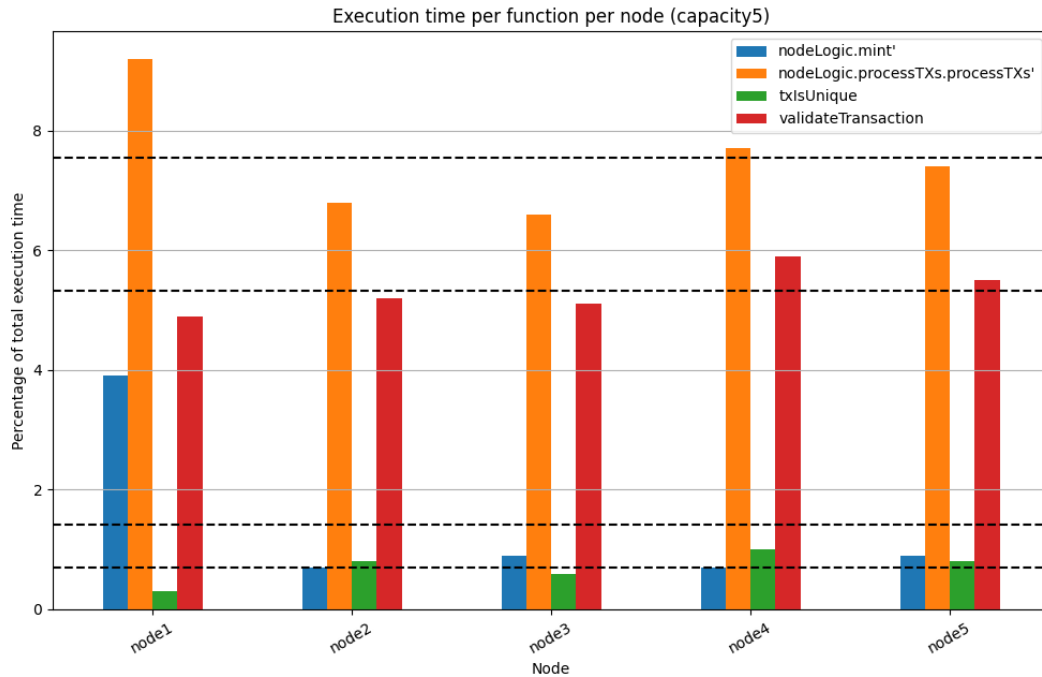


Σχήμα 10: Τα πιο χρονοβόρα κομμάτια του κώδικα capacity=5

Στον πίνακα 3 φαίνονται οι κλήσεις μερικών συναρτήσεων ενδιαφέροντος. Φαίνεται ότι τα πλήθη όλων των κλήσεων είναι ίδια ανά κόμβο, πράγμα που σημαίνει ότι οι κόμβοι εκτελούν τις ίδιες λειτουργίες με την ίδια συχνότητα. Παρόλα αυτά, ο κόμβος με το μεγαλύτερο stake καταναλώνει πολύ περισσότερο χρόνο στην συνάρτηση mint' σε σχέση με τους υπόλοιπους κόμβους, όπως φαίνεται και στο σχήμα 11.

Πίνακας 3: Στατιστικά συναρτήσεων ανά κόμβο capacity=5

Node	Function	Entries	TimeInh
node1.prof:	nodeLogic.processTXs.processTXs'	301	10.8
node1.prof:	validateTransaction	255	5.3
node1.prof:	txIsUnique	231	0.6
node1.prof:	nodeLogic.mint'	45	4.6
node2.prof:	nodeLogic.processTXs.processTXs'	301	7.0
node2.prof:	validateTransaction	255	5.4
node2.prof:	txIsUnique	229	0.5
node2.prof:	nodeLogic.mint'	45	0.9
node3.prof:	nodeLogic.processTXs.processTXs'	301	6.4
node3.prof:	validateTransaction	255	5.0
node3.prof:	txIsUnique	225	0.9
node3.prof:	nodeLogic.mint'	45	0.3
node4.prof:	nodeLogic.processTXs.processTXs'	300	8.1
node4.prof:	validateTransaction	255	6.5
node4.prof:	txIsUnique	223	1.0
node4.prof:	nodeLogic.mint'	44	0.5
node5.prof:	nodeLogic.processTXs.processTXs'	301	8.6
node5.prof:	validateTransaction	255	6.2
node5.prof:	txIsUnique	228	0.6
node5.prof:	nodeLogic.mint'	45	1.6



Σχήμα 11: Ποσοστό χρόνου επί του συνολικού χρόνου εκτέλεσης που λαμβάνει η κάθε συνάρτηση capacity=5

Στο σχήμα 11 φαίνεται, πράγματι, ότι ο κόμβος με το μεγαλύτερο stake καταναλώνει πολύ περισσότερο χρόνο στην συνάρτηση `mint` σε σχέση με τους υπόλοιπους κόμβους, ενδεικτικό του γεγονότος ότι πράγματι αυτός αναλαμβάνει συχνότερα την δημιουργία των νέων blocks. Μάλιστα, επισκοπώντας τα υπόλοιπα των λογαριασμών των κόμβων στο σχήμα 12, παρατηρεί κανείς ότι όντως τα περισσότερα νομίσματα συσσωρεύονται στον κόμβο με το μεγαλύτερο stake. Συμπεραίνεται, λοιπόν, ότι σε βάθος χρόνου συσσωρεύονται νομίσματα στον κόμβο με το μεγαλύτερο stake. Θεωρητικά, αυτό σημαίνει ότι ένας κακόβουλος κόμβος θα μπορούσε να εκμεταλλευτεί το φαινόμενο αυτό και να χειραγωγεί το δίκτυο κατά την θέλησή του. Επομένως, υπάρχει ανάγκη για έναν μηχανισμό που θα εξασφαλίζει ότι, παρά την ανισότητα των stakes, οι κόμβοι θα έχουν ίσες ευκαιρίες στην επικύρωση των συναλλαγών και δεν θα επαφίεται η ασφάλεια του δικτύου σε έναν μόνο κόμβο.

```
==> node1.log <==  
> 3497.2  
  
==> node2.log <==  
> 280.70000000000005  
  
==> node3.log <==  
> 232.70000000000005  
  
==> node4.log <==  
> 191.70000000000005  
  
==> node5.log <==  
> 697.7
```

Σχήμα 12: Υπόλοιπα λογαριασμών κόμβων `capacity=5`