ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Κατανεμημένα Συστήματα

Εξαμηνιαία Εργασία - BlockChat

Αναστάσιος Στέφανος Αναγνώστου
03119051

29 Μαρτίου 2024

Περιεχόμενα

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. Εξαίρεση αποτελούν τα αρχεία `Utils.hs` και `Types.hs` τα οποία περιέχουν βοηθητικές συναρτήσεις και δηλώσεις τύπους 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.

```haskell
{−# LANGUAGE PackageImports #−}

module Wallet
  ( Wallet,
    emptyWallet,
    generateWallet,
  )
where

import Codec.Crypto.RSA (PrivateKey, PublicKey, generateKeyPair)
import "crypto−api" Crypto.Random (SystemRandom, newGenIO)

type Wallet = (PublicKey, PrivateKey)

emptyWallet :: Wallet
emptyWallet = (undefined, undefined)

-- these returns an IO pair, so in order to actually use
-- the values, we have to be inside of something
-- that executes an IO action
generateWallet :: Int -> IO (PublicKey, PrivateKey)
generateWallet bits = do
  g <- newGenIO :: IO SystemRandom
  let (pubKey, privKey, _) = generateKeyPair g bits
  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.

```haskell
data Account = Account
  { accountBalance :: Double, -- the balance of the account.
    accountNonce :: Int,
    -- a field that is incremented with every
    -- outgoing transaction, in order to guard
    -- against replay attacks.
    accountStake :: Double
    -- the amount of stake that the account has
    -- for the PoS protocol.
  }
  deriving (Show, Eq)

-- | The initial account has a balance of 1000 coins, a nonce of 0 and no stake.
initialAccount :: Account
initialAccount = Account 1000 0 0

-- | The available balance of an account is the balance minus the stake.
availableBalance :: Account -> Double
availableBalance Account {accountBalance = bal, accountStake = st} = bal − st

-- | This function takes an amount and an account as arguments and returns a new account
-- with the balance updated by the amount.
```

```haskell
updateBalanceBy :: Double -> Account -> Account
updateBalanceBy amount acc = acc {accountBalance = accountBalance acc + amount}

-- | This function takes an account as an argument and returns a new account with the nonce
-- incremented by 1.
updateNonce :: Account -> Account
updateNonce acc = acc {accountNonce = accountNonce acc + 1}

-- | This function takes an amount and an account as arguments and returns a new account
-- with the stake updated by the amount.
updateStake :: Double -> Account -> Account
updateStake amount acc =
  acc
    { accountStake = amount,
      accountBalance = accountBalance acc + (accountStake acc - amount)
    }
```

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 στιγμιοτύπου της δομής `Block`.

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

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.

```haskell

instance Binary BlockInit where
  put :: BlockInit -> Put
  put (BlockInit index time trans val prev) = do
    put index
    put time
    put trans
    put val
    put prev
  get :: Get BlockInit
  get = do
    index <- get
    time <- get
    trans <- get
    val <- get
    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`.

```haskell

data Block = Block
  { blockIndex :: Int, -- index number of block
```

```
    blockTimestamp :: UnixTime, -- microseconds since 1st Jan 1970
    blockTransactions :: [Transaction], -- list of txs
    blockValidator :: PublicKey, -- public key of the node that validated the tx
    blockPreviousHash :: ByteString, -- the hash of the previous block
    blockCurrentHash :: ByteString
  }
  deriving (Show, Eq)

instance Binary Block where
  put :: Block -> Put
  put (Block index time trans val prev curr) = do
    put index
    put time
    put trans
    put val
    put prev
    put curr
  get :: Get Block
  get = do
    index <- get
    time <- get
    trans <- get
    val <- get
    prev <- get
    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`.

```

computeBlockHash :: BlockInit -> ByteString
computeBlockHash = convert . hashWith SHA256 . B.toStrict . encode

finalizeBlock :: BlockInit -> Block
finalizeBlock initBlock =
  Block
    { blockIndex = initIndex initBlock,
      blockTimestamp = initTimestamp initBlock,
      blockTransactions = initTransactions initBlock,
      blockValidator = initValidator initBlock,
      blockPreviousHash = initPreviousHash initBlock,
      blockCurrentHash = computeBlockHash initBlock
    }

emptyBlock :: Block
emptyBlock = Block 0 (UnixTime 0 0) [] (PublicKey 0 0 65537) BS.empty BS.empty

-- the capacity of transactions that the block holds is specified
-- by an environmental constant called "capacity".
createBlock :: Int -> UnixTime -> [Transaction] -> PublicKey -> ByteString -> Block
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.

```
type Blockchain = [Block]

blockMsgHeader :: ByteString
```

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
2  validateBlock :: Block -> Block -> PublicKey -> Bool
3  validateBlock newblock prevblock validator = validatorOK && hashOK
4    where
5      validatorOK = blockValidator newblock == validator
6      hashOK = blockPreviousHash newblock == blockCurrentHash prevblock
7
8  -- | Helper function to broadcast a block to all peers.
9  broadcastBlock :: [Peer] -> Block -> IO ()
10 broadcastBlock peers block = mapM_ sendBlock peers
11   where
12     msg = BS.append blockMsgHeader $ encodeStrict block
13     sendBlock :: (HostName, ServiceName) -> IO ()
14     sendBlock (host, port) = connect host port $ \(sock, _) -> send sock msg
15
16 -- | This function takes a transaction and a blockchain and returns True if the transaction is unique.
17 -- A transaction is unique if it is not present in any of the blocks in the blockchain.
18 txIsUnique :: Transaction -> Blockchain -> Bool
19 txIsUnique tx = all (notElem tx . blockTransactions)
20
21 -- | This function takes a ClockTime and returns the number of milliseconds since 1st Jan 1970.
22 milliseconds :: ClockTime -> Integer
23 milliseconds (TOD sec pico) = sec * 1000 + pico `div` 1000000000
24
25 -- | This function takes a blockchain and returns the mean time between blocks.
26 meanBlockTime :: Blockchain -> Double
27 meanBlockTime chain = fromIntegral (sum times'') / fromIntegral (length times)
28   where
29     times = map (Data.UnixTime.toClockTime . blockTimestamp) (reverse chain)
30     times' = zip times (tail times)
31     times'' = map (\(a, b) -> milliseconds b - milliseconds a) times'
```

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:

```haskell
-- | Broadcasts a transaction to a list of peers.
broadcastTransaction :: [Peer] -> Transaction -> IO ()
broadcastTransaction peers t = mapM_ sendMsg peers
  where
    msg = append txMsgHeader $ encodeStrict t
    sendMsg :: (HostName, ServiceName) -> IO ()
    sendMsg (host, port) = connect host port $ \(sock, _) -> send sock msg

-- | Verifies the signature of a transaction.
verifySignature :: Transaction -> Bool
verifySignature t = verify from tid sig
  where
    -- pay attention to the order of the arguments
    from = senderAddress t
    tid = B.fromStrict $ hashID t
    sig = B.fromStrict $ signature t

-- | This function takes a transaction and a map of public keys to accounts and returns
-- whether the transaction is valid or not.
validateTransaction :: Transaction -> PubKeyToAcc -> Bool
validateTransaction t m = verifySignature t && maybe False validateSender senderAcc
  where
    validateSender acc = availableBalance acc >= txCost t
    senderAcc = Map.lookup (senderAddress t) m
```

Listing 9: Transaction.hs

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

```haskell
type Peer = (HostName, ServiceName)

type PubKeyToAcc = Map.Map PublicKey Account
```

Listing 10: Types.hs

ενώ η συνάρτηση `encodeStrict` ορίζεται στο αρχείο `Utils.hs`:

```haskell
encodeStrict :: (Binary a) => a -> BS.ByteString
encodeStrict = BS.toStrict . encode

decodeMaybe :: (Binary a) => Maybe BS.ByteString -> a
decodeMaybe = maybe (decode "") (decode . BS.fromStrict)

receiveChunks :: Socket -> Int -> IO BS.ByteString
receiveChunks socket limit = receiveChunks' ""
  where
    receiveChunks' acc | BS.length acc < limit = do
      raw <- recv socket 1024 -- have a byte
      case raw of
        Nothing -> return acc -- can't get no more bytes
        Just msg -> receiveChunks' $ BS.append acc msg -- keep eating
    receiveChunks' acc | BS.length acc == limit = return acc
    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.

```haskell
-- | This function takes a transaction and a state of accounts as arguments and returns
```

```haskell
-- a new state of accounts, as a result of the transaction. It does not update the nonce
updateAccsByTX :: Transaction -> PubKeyToAcc -> PubKeyToAcc
updateAccsByTX t m = case serviceType t of
  Coins c ->
    -- if the receiver is the zeropub, then the transaction is a staking transaction
    if receiverAddress t == zeropub
    then
      Map.adjust (updateBalanceBy (- txFee t) . updateStake c) (senderAddress t) m
    else
      let cost = txCost t
          sender = senderAddress t
          receiver = receiverAddress t
          temp = Map.adjust (updateBalanceBy (- cost)) sender m
       in Map.adjust (updateBalanceBy c) receiver temp
  Message _ ->
    let cost = txCost t
        sender = senderAddress t
     in Map.adjust (updateBalanceBy (- cost)) sender m
  Both (c, _) ->
    let cost = txCost t
        sender = senderAddress t
        receiver = receiverAddress t
        temp = Map.adjust (updateBalanceBy (- cost)) sender m
     in Map.adjust (updateBalanceBy c) receiver temp

-- | This function takes a list of transactions and a state of accounts as arguments and returns
-- a new state of accounts, as a result of the transactions.
updateAccsByTXs :: [Transaction] -> PubKeyToAcc -> PubKeyToAcc
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`.

```haskell
-- | This function takes a transaction (only the initial fields ) and returns its hash.
computeHashID :: TransactionInit -> ByteString
computeHashID = convert . hashWith SHA256 . B.toStrict . encode

-- | This function takes a private key and a bytestring and returns the signature of the bytestring.
computeSignature :: PrivateKey -> ByteString -> ByteString
computeSignature privkey bytestring = B.toStrict $ sign privkey (B.fromStrict bytestring)

-- | This function takes a transaction and a private key and
-- returns the transaction with the signature field filled.
finalizeTransaction :: TransactionInit -> PrivateKey -> Transaction
finalizeTransaction initTx privKey =
  Transaction
    { senderAddress = initSenderAddress initTx,
      receiverAddress = initReceiverAddress initTx,
      serviceType = initServiceType initTx,
      nonce = initNonce initTx,
      hashID = computeHashID initTx,
      signature = computeSignature privKey (computeHashID initTx)
    }

-- | This function takes a public key (sender), another public key (receiver), a service type,
-- a counter and a private key and returns a transaction.
createTransaction :: PublicKey -> PublicKey -> ServiceType -> Int -> PrivateKey -> Transaction
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

Κατάρχάς, ορίζονται κάποιοι βοηθητικοί τύποι:

```
data BootstrapNode = BootstrapNode HostName ServiceName

-- BootInfo does not change. It is set once, using the arguments
-- passed to the program and then remains constant. The IP and the PORT
-- of the Boot node are known to the other nodes beforehand.
data BootInfo = BootInfo
  { bootNodeID :: Int,
    bootNodeIP :: HostName,
    bootNodePort :: ServiceName,
    bootNodeNumb :: Int -- number of nodes to insert into the network
  }
  deriving (Show, Eq)

data BootState = BootState
  { bootCurrID :: Int,
    bootPublicKeys :: [(Int, PublicKey)],
    bootPeers :: [(HostName, ServiceName)],
    bootBlockchain :: Blockchain
  }
  deriving (Show, Eq)

-- | This is the initial state of the boot node
emptyBootState :: BootState
emptyBootState = BootState 0 [] [] []

-- | This is the function that the driver calls to
-- initiate the bootstrap node logic
bootstrapNode :: BootInfo -> IO BootState
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.

```

-- | This is a helper function that updates the state of the boot node
updateState :: (Int, (PublicKey, HostName, ServiceName)) -> BootState -> (BootState, ())
updateState t s = (s {bootPublicKeys = newKeys, bootPeers = newPeers}, ())
  where
    (num, (pub, ip, port)) = t
    newKeys = (num, pub) : bootPublicKeys s
    newPeers = (ip, port) : bootPeers s

-- | This is a helper function that increments the current ID of the boot state
incrementID :: BootState -> (BootState, Int)
incrementID s = (s {bootCurrID = newID}, newID)
  where
    newID = bootCurrID s + 1

-- | This is the function that handles the server logic
server :: [MVar Int] -> IORef BootState -> (Socket, SockAddr) -> IO ()
server triggers ioref (socket, _) = do
  currID <- atomicModifyIORef' ioref incrementID
  when (currID <= length triggers) $ do
    msg <- receiveChunks socket 4096 -- for public key (2048), an ip and a port
    let keyval = (decode . LBS.fromStrict) msg :: (PublicKey, HostName, ServiceName)
    atomicModifyIORef' ioref $ updateState (currID, keyval)
    send socket $ encodeStrict currID
    putMVar (triggers !! (currID - 1)) 1
```

Listing 15: BootstrapNode.hs

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

```
4
5  -- NodeInfo does not change.
6  -- It is set once, using the arguments passed to the program and then remains constant.
7  data NodeInfo = NodeInfo
8    { nodeIP :: HostName,
9      nodePort :: ServiceName,
10     nodeInfoWallet :: Wallet
11   }
12   deriving (Show, Eq)
```

Listing 17: OrdinaryNode.hs

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

Στην συνέχεια η συνάρτηση `server`:

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

```
    epsilon = 0.0001 :: Double
    epsProbs = map (\(x, y) -> (x + epsilon, y)) probs

-- | Helper function to get a valid block from a queue of blocks
-- given the last block and the public key of the validator.
getValidatorBlockFrom :: BLQueue -> Block -> PublicKey -> IO Block
getValidatorBlockFrom qBlocks lastBlock valKey = do
  block <- atomically $ dequeueTQ qBlocks
  if validateBlock block lastBlock valKey
    then return block
    else getValidatorBlockFrom qBlocks 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 παρουσίαζεται 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`.

```
  let initialAccounts = Map.fromList $ map ((,initialAccount) . snd) keys
      clishared = (blockchainRef, accountRef) :: CLISharedState
      mypeers = filter (/= (myid, mypub)) keys
      cliinfo = CLIInfo mywallet (Map.fromList mypeers) ip port friends
      -- This function processes transactions (keeping track of the counter) and mints
      -- a new block when the counter reaches the capacity.
      processTXs :: IO ()
      processTXs = processTXs' clishared [] queuedTXs (initialAccounts, initialAccounts)
        where
          processTXs' :: CLISharedState -> [Transaction] -> TXQueue -> (PubKeyToAcc, PubKeyToAcc) -
          processTXs' sharedState vTxs qTxs (accmap, fallback) = do
            when (length vTxs /= capacity) $ do
              tx <- atomically $ dequeueTQ qTxs
              blockchain <- (readIORef . fst) sharedState
              if validateTransaction tx accmap && txIsUnique tx blockchain
                then
                  processTXs' sharedState (tx : vTxs) qTxs (updateAccsByTX tx accmap, fallback)
                else
                  processTXs' sharedState vTxs qTxs (accmap, fallback)
            blockchain <- (readIORef . fst) sharedState
            (newblock, newaccs) <- mint (head blockchain) vTxs (accmap, fallback)
            writeIORef (fst sharedState) (newblock : blockchain)
            -- the cli is responsible for updating the account nonce
            atomicModifyIORef' (snd sharedState) (\a -> ((newaccs Map.! mypub) {accountNonce = accountN
            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:

```
mint :: Block -> [Transaction] -> (PubKeyToAcc, PubKeyToAcc) -> IO (Block, PubKeyToAcc)
mint = mint' (filter ((ip, port) /=) friends)

mint' :: [Peer] -> Block -> [Transaction] -> (PubKeyToAcc, PubKeyToAcc) -> IO (Block, PubKeyToAcc)
mint' peers lastBlock vTxs (accountMap, fallback) = do
  let -- the stake has to be taken from the fallback state
      -- because subsequent states are not validated
      accs = Map.elems fallback
      weights = zip (map accountStake accs) [1 ..]

      prevhash = blockPreviousHash lastBlock
      currhash = blockCurrentHash lastBlock
      seed = decode . BS.fromStrict $ prevhash

```

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

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. Ειδάλλως, 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`.

```
-- | Handle the input from the user
handle :: String -> CLISharedState -> ReaderT CLIInfo IO ()
handle input shared = do
  liftIO $ threadDelay 500000
  let tokens = words input
      (blockref, accref) = shared
  case tokens of
    ("t" : numStr : "Coins" : coinsStr : _) ->
      case (reads numStr, reads coinsStr) of
        ([(num, "")], [(coins, "")]) -> do
          liftIO $ atomicModifyIORef' accref (\a -> (a {accountNonce = accountNonce a + 1}, ()))
          acc' <- liftIO $ readIORef accref
          sendTx num (Coins coins) acc'
        _ -> liftIO $ putStrLn "Invalid command. Try entering 'help' for help."
    ("t" : numStr : "Message" : msgParts) ->
      case reads numStr of
        [(num, "")] -> do
          liftIO $ atomicModifyIORef' accref (\a -> (a {accountNonce = accountNonce a + 1}, ()))
          acc' <- liftIO $ readIORef accref
          sendTx num (Message $ unwords msgParts) acc'
        _ -> liftIO $ putStrLn "Invalid command. Try entering 'help' for help."
    ("t" : numStr : "Both" : coinsStr : "," : msgParts) ->
      case (reads numStr, reads coinsStr) of
        ([(num, "")], [(coins, "")]) -> do
          liftIO $ atomicModifyIORef' accref (\a -> (a {accountNonce = accountNonce a + 1}, ()))
          acc' <- liftIO $ readIORef accref
          sendTx num (Both (coins, unwords msgParts)) acc'
        _ -> liftIO $ putStrLn "Invalid command. Try entering 'help' for help."
    ("stake" : coinsStr : _) ->
      case reads coinsStr of
        [(coins, "")] -> do
          liftIO $ atomicModifyIORef' accref (\a -> (a {accountNonce = accountNonce a + 1}, ()))
          acc' <- liftIO $ readIORef accref
          stake coins acc'
        _ -> liftIO $ putStrLn "Invalid command. Try entering 'help' for help."
    ["view"] -> do
      blockchain <- liftIO $ readIORef blockref
      prettyPrintBlock (head blockchain)
    ["blockchain"] -> do
      liftIO $ threadDelay 5000000 -- just for testing
      blockchain <- liftIO $ readIORef blockref
      prettyPrintBlockchain blockchain
      -- show the mean time between blocks but disregard the genesis block
      -- that is the last one in the list
      liftIO $ putStrLn $ "Mean time between blocks: " ++ show (meanBlockTime $ init blockchain)
    ["balance"] -> do
      acc <- liftIO $ readIORef accref
      liftIO $ print (accountBalance acc)
    ["peers"] -> do
      keymap <- asks cliIDtoKey
      liftIO $ prettyPrintPeers keymap
    ["load", filename] -> do
      liftIO $ putStrLn $ "Loading transactions from " ++ filename
      -- execute each line of the file as a command
      contents <- liftIO $ readFile filename
      mapM_ ('CLI.handle' shared) (lines contents)
    ["exit"] -> liftIO $ putStrLn "Exiting .." >> exitSuccess
    ["help"] -> do
      liftIO $ putStrLn "t <recipient id> Coins <coins>       - send coins"
      liftIO $ putStrLn "t <recipient id> Message <msg>       - send message"
      liftIO $ putStrLn "t <recipient id> Both (<coins>, <msg>) - send both"
      liftIO $ putStrLn "stake <coins>                        - stake coins"
      liftIO $ putStrLn "view                                 - view last block"
      liftIO $ putStrLn "blockchain                           - view the entire blockchain"
```

```
65        liftIO $ putStrLn "balance                                    -- view account balance"
66        liftIO $ putStrLn "peers                                      -- show list of peers"
67        liftIO $ putStrLn "load <filename>                            -- load transactions from a file"
68        liftIO $ putStrLn "exit                                       -- exit the shell"
69        liftIO $ putStrLn "help                                       -- show this message"
70      _ -> 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 :: CLISharedState -> ReaderT CLIInfo IO ()
2  shell shared = do
3    liftIO $ putStrLn "Loading .."
4    liftIO $ threadDelay 2000000
5    liftIO $ putStrLn "Welcome to (the s)hell!"
6    liftIO $ putStrLn "Type 'help' to ask for help."
7    loop
8    where
9      loop :: ReaderT CLIInfo IO ()
10     loop = do
11       liftIO $ putStr "> " >> hFlush stdout
12       input <- liftIO safeGetLine
13       case input of
14         Nothing -> CLI.handle "exit" shared
15         Just line -> CLI.handle line shared >> loop
```

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 2056
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

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:

```
FROM haskell:9.6.4

WORKDIR /app
COPY . /app

# Now using the specified Stack version for setup and build
RUN stack setup
RUN stack build --profile

# more arguments will be passed to 'stack exec'
ENTRYPOINT ["stack", "exec", "--profile", "--", "BlockChat-exe"]
# Optional: Default arguments that can be overridden
CMD []
```

Listing 26: Dockerfile

και το script που οδηγεί τα containers:

```
#!/bin/bash

# execute this script from within the experiments directory

BASE_IP="172.0.0."
BASE="0"
PORT="35900"
IMAGE_NAME="blockchat"
PREFIX="experiments/profiled_outputs/docker"
SUFFIX=""
CAPACITY=("5" "10" "20")
PROFLOG="/app/node"

for TEST in "throughput" "scalability" "fairness";
do
    if [ $TEST != "scalability" ];
    then
        NODES="5"
    else
        NODES="10"
    fi
    for CAP in "${CAPACITY[@]}";
    do
        WORKDIR=$PREFIX/$TEST$SUFFIX/capacity$CAP
        if [ ! -d "$WORKDIR" ];
        then
            mkdir -p "$WORKDIR"
        fi

        docker network create --subnet=172.0.0.0/16 blockchat-net

        docker run --rm \
            --net blockchat-net \
```

```
34              --ip "$BASE_IP$((BASE+2))"\
35              -p $PORT:$PORT \
36              --name bootstrap\
37              $IMAGE_NAME --bootstrap "$BASE_IP$((BASE+2))" $PORT $NODES +RTS -p -RTS &
38
39          sleep 2
40
41          docker logs -f bootstrap > $WORKDIR/bootstrap.log &
42
43          if [ $TEST == "fairness" ];
44          then
45              initial_stake ="stake 100"
46          else
47              initial_stake ="stake 10"
48          fi
49
50          input="experiments/input$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" +RTS -p -po$PROFLOG -RTS &
59          sleep 2
60          docker logs -f node1 > $WORKDIR/node1.log &
61
62          for i in $(seq 2 $NODES);
63          do
64              input="experiments/input$NODES/trans$i.txt"
65              (cat <(echo "stake 10") <(echo "load $input") <(echo "blockchain") <(echo "balance"))|\
66                  docker run -i \
67                      --net blockchat-net \
68                      --ip "$BASE_IP$((BASE+2+i))"\
69                      -p $((PORT+i)):$PORT \
70                      --name node"$i"\
71                      $IMAGE_NAME --node "$BASE_IP$((BASE+2+i))" "$PORT" "$BASE_IP$((BASE+2))"\
72                      "$PORT" "$CAP" +RTS -p -po$PROFLOG -RTS &
73              sleep 2
74              docker logs -f node"$i" > $WORKDIR/node"$i".log &
75          done
76
77          # wait for containers from node1 to node$NODES to finish
78          for i in $(seq 1 $NODES);
79          do
80              docker wait node"$i"
81              # get the profiled outputs
82              docker cp node"$i":$PROFLOG.prof $WORKDIR/node"$i".prof
83              docker rm node"$i"
84          done
85
86          docker network rm blockchat-net
87      done
88  done
```

Listing 27: run-docker-experiments.sh

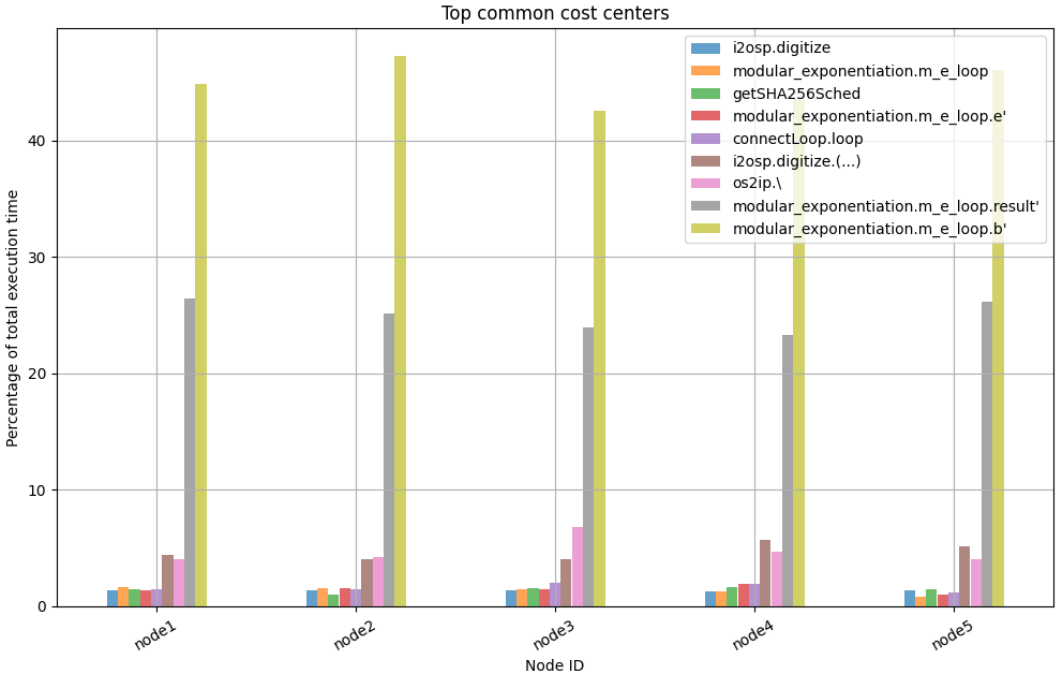
4 Απόδοση του συστήματος

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

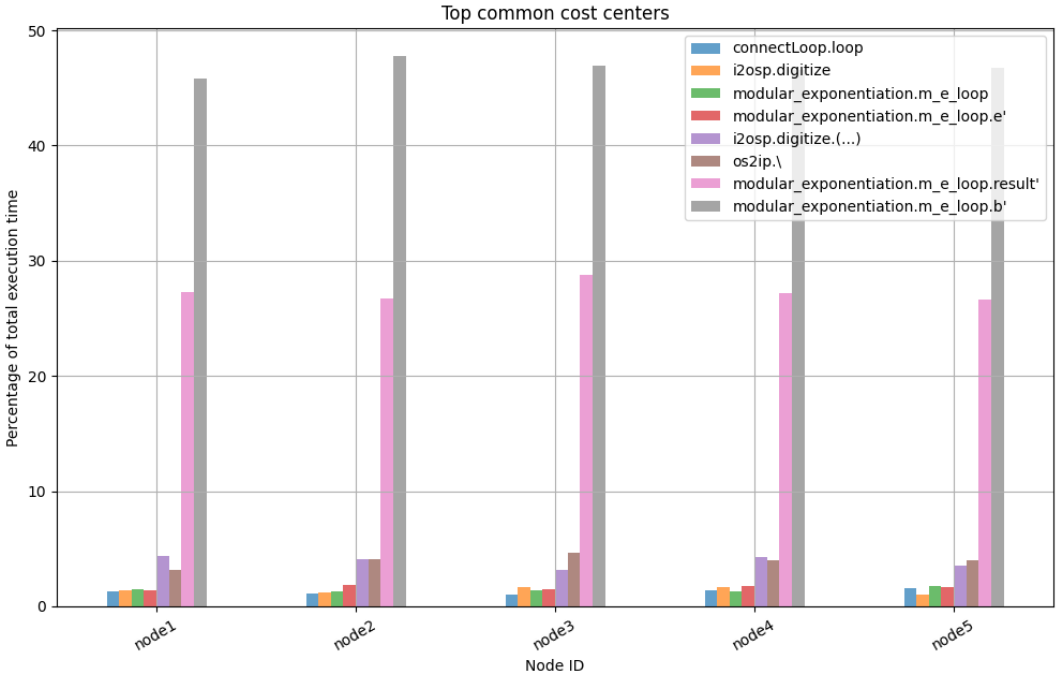
Στο πείραμα για την αξιολόγηση της ρυθμαπόδοσης του συστήματος, στήνεται ένα δίκτυο 5 κόμβων, καθένας εκ των οποίων εκτελεί 1 staking, με stake 10 BCC συναλλαγή και 50 συναλλαγές (συγκεκριμένα αποστολές μηνυμάτων) προς τους άλλους κόμβους. Η ταχύτητα αποστολής συναλλαγών είναι ίδια μεταξύ των κόμβων,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

Το πρώτο πράγμα που φαίνεται στο σχήμα 2 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 υπολογισμού[1] του προγράμματος.

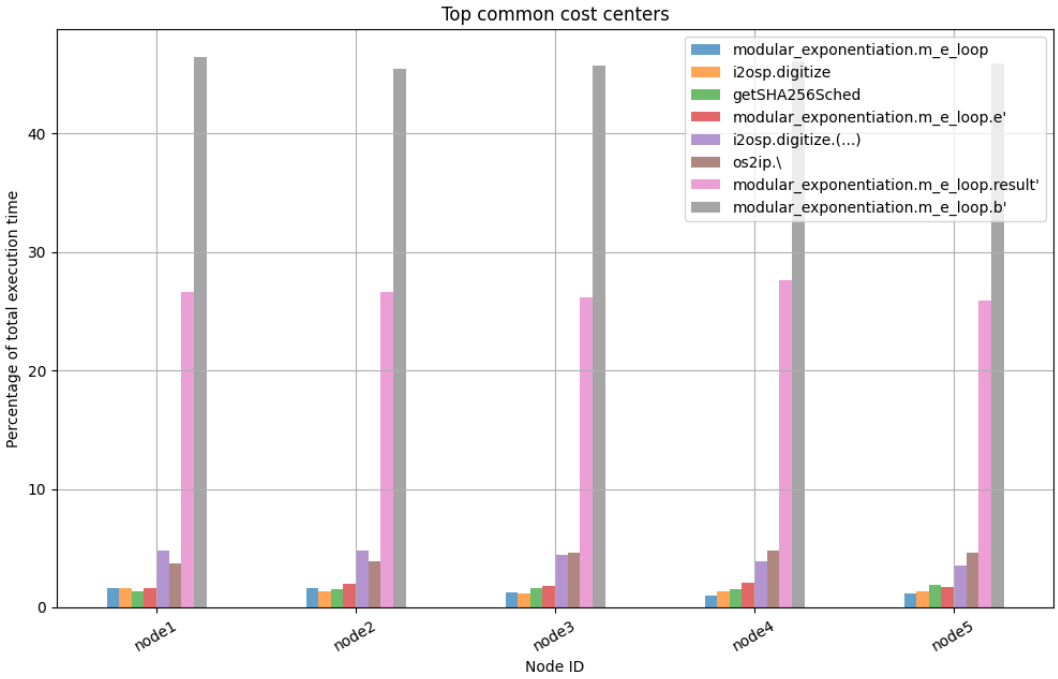


(α′) capacity=5

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



(β′) capacity=10

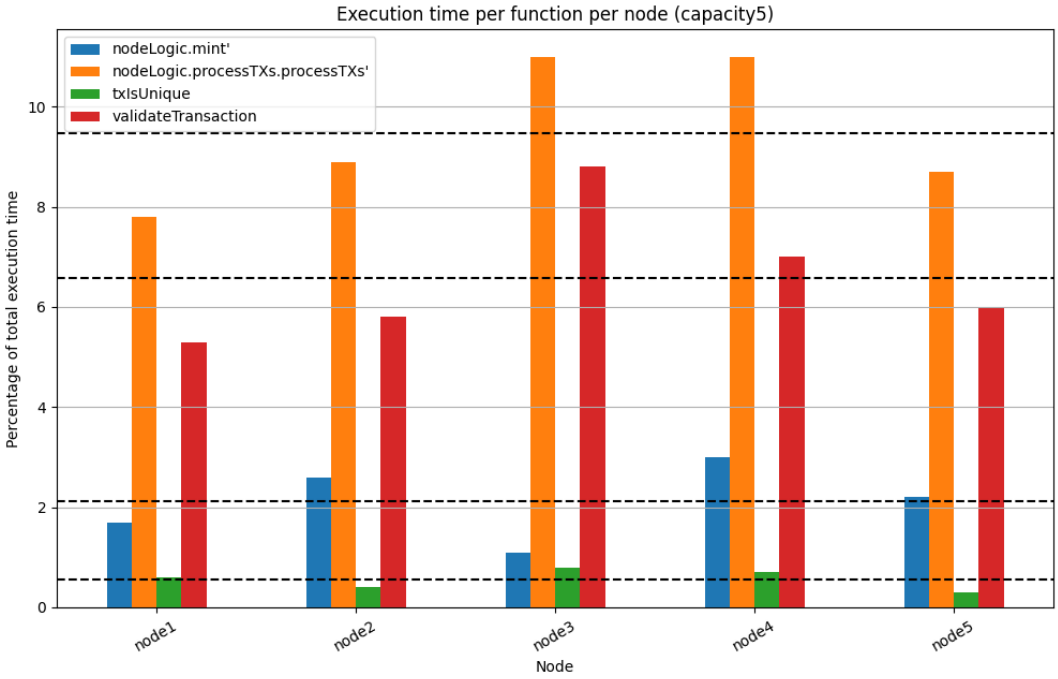


(γ′) capacity=20

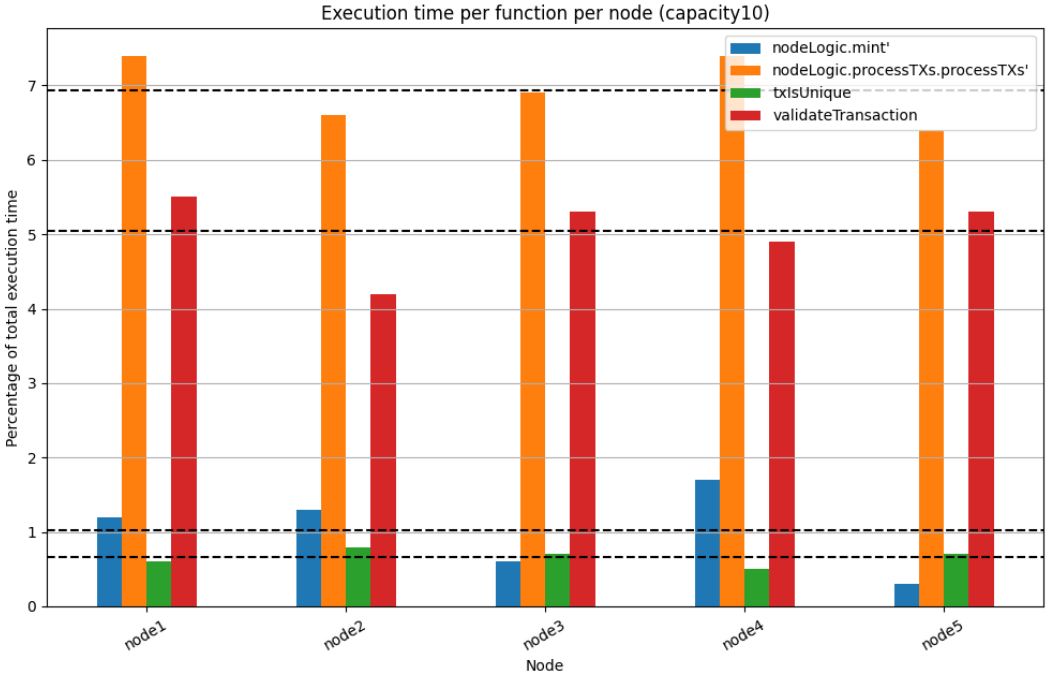
Σχήμα 2: Τα πιο χρονοβόρα κομμάτια του κώδικα

4.2 Συναρτήσεις του συστήματος

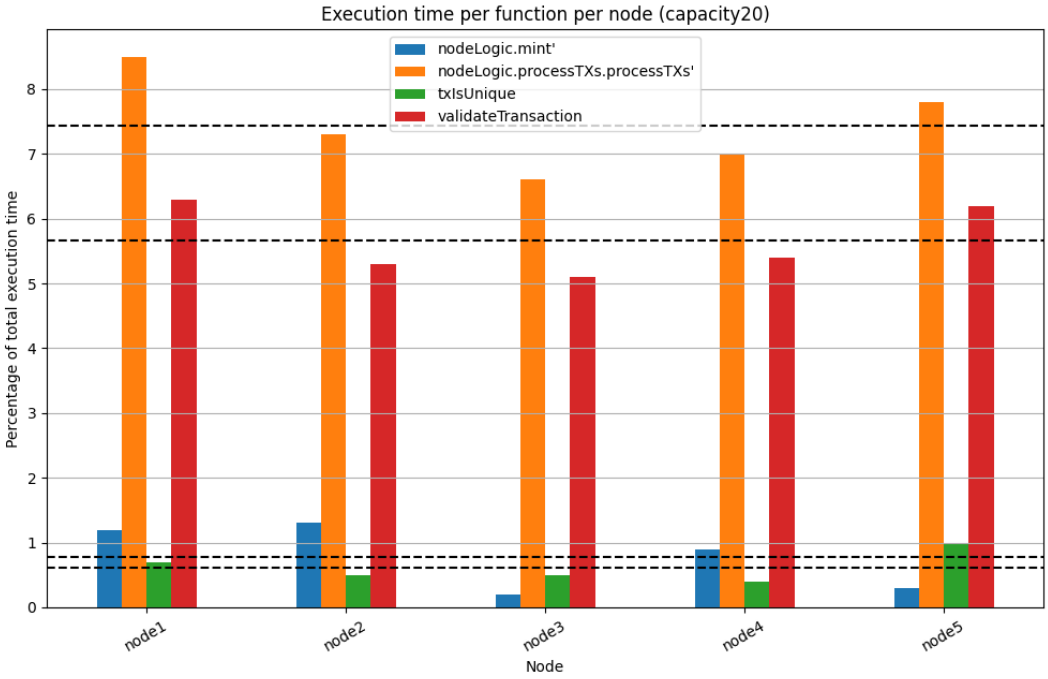
Σχετικά με τις top level συναρτήσεις του συστήματος, παρατηρείται ότι, με κάποιες μικρές διακυμάνσεις, η processTXs καταναλώνει 9-10% του συνολικού CPU time, η mint 1-3% και η validateTransaction 5-9%, με μέσο όρο περίπου 6.5%.



(α′) Ρυθμαπόδοση capacity=5



(β′) Ρυθμαπόδοση capacity=10



(γ′) Ρυθμαπόδοση capacity=20

Σχήμα 3: 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]. Παρ΄ότι 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'	264	7.8
node1.prof:	validateTransaction	223	5.3
node1.prof:	txIsUnique	207	0.6
node1.prof:	nodeLogic.mint'	40	1.7
node2.prof:	nodeLogic.processTXs.processTXs'	265	8.9
node2.prof:	validateTransaction	223	5.8
node2.prof:	txIsUnique	207	0.4
node2.prof:	nodeLogic.mint'	41	2.6
node3.prof:	nodeLogic.processTXs.processTXs'	263	11.0
node3.prof:	validateTransaction	223	8.8
node3.prof:	txIsUnique	201	0.8
node3.prof:	nodeLogic.mint'	39	1.1
node4.prof:	nodeLogic.processTXs.processTXs'	263	11.0
node4.prof:	validateTransaction	223	7.0
node4.prof:	txIsUnique	201	0.7
node4.prof:	nodeLogic.mint'	39	3.0
node5.prof:	nodeLogic.processTXs.processTXs'	265	8.7
node5.prof:	validateTransaction	223	6.0
node5.prof:	txIsUnique	209	0.3
node5.prof:	nodeLogic.mint'	41	2.2

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.

$$\begin{aligned} &\text{Χωρητικότητα} = 5 \Rightarrow \text{Μπλοκς} = 41 \text{ και } \Sigma\text{υναλλαγές} = 205 \\ &\text{Χωρητικότητα} = 10 \Rightarrow \text{Μπλοκς} = 22 \text{ και } \Sigma\text{υναλλαγές} = 220 \\ &\text{Χωρητικότητα} = 20 \Rightarrow \text{Μπλοκς} = 11 \text{ και } \Sigma\text{υναλλαγές} = 220 \end{aligned} \tag{1}$$

[2] Η -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 4 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 κατάνάγκην όλοι οι κόμβοι στο ίδιο σημείο της αλυσίδας και για αυτόν τον λόγο διαφοροποιείται η μέτρησή τους. Εδώ λαμβάνεται υπόψιν το μέγιστο μήκος της αλυσίδας μεταξύ των κόμβων.

```
capacity5/node1.log:Mean time between blocks: 625.0
capacity5/node2.log:Mean time between blocks: 658.5853658536586
capacity5/node3.log:Mean time between blocks: 641.025641025641
capacity5/node4.log:Mean time between blocks: 641.025641025641
capacity5/node5.log:Mean time between blocks: 658.5853658536586

capacity10/node1.log:Mean time between blocks: 1045.5
capacity10/node2.log:Mean time between blocks: 1045.5
capacity10/node3.log:Mean time between blocks: 1045.5
capacity10/node4.log:Mean time between blocks: 1045.5
capacity10/node5.log:Mean time between blocks: 1045.5

capacity20/node1.log:Mean time between blocks: 2000.3636363636363
capacity20/node2.log:Mean time between blocks: 2000.3636363636363
capacity20/node3.log:Mean time between blocks: 2000.3636363636363
capacity20/node4.log:Mean time between blocks: 2000.3636363636363
capacity20/node5.log:Mean time between blocks: 2000.3636363636363
```

Σχήμα 4: 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:

$$\text{Ρυθμαπόδοση} = \frac{\text{Συναλλαγές}}{\text{Χρόνος}} = \frac{\text{Συναλλαγές}}{\text{Μπλοκ}} \cdot \frac{\text{Μπλοκ}}{\text{Χρόνος}} \Leftrightarrow$$

$$\text{Ρυθμαπόδοση} = \frac{\frac{\text{Συναλλαγές}}{\text{Μπλοκ}}}{\text{Μέσος χρόνος δημιουργίας μπλοκ}} = \frac{\text{Χωρητικότητα}}{\text{Μέσος χρόνος δημιουργίας μπλοκ}} \Rightarrow$$

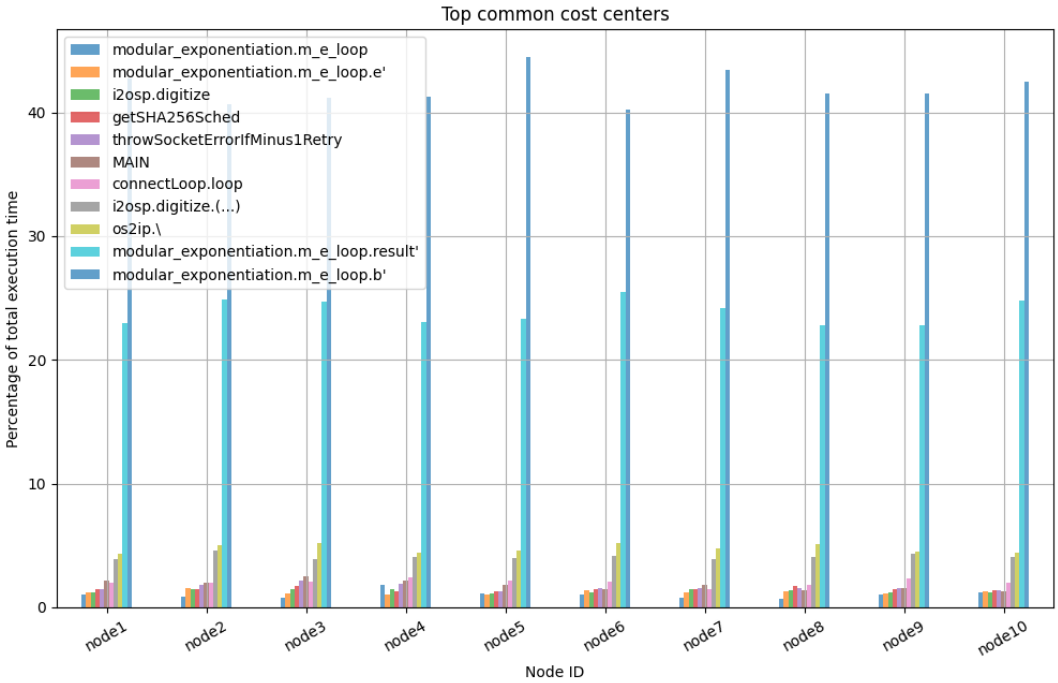
$$\boxed{\begin{aligned} \text{Ρυθμαπόδοση}_{capacity=5} &= \frac{5}{0,658} = 7,598\frac{txs}{s} \\ \text{Ρυθμαπόδοση}_{capacity=10} &= \frac{10}{1,045} = 9,569\frac{txs}{s} \\ \text{Ρυθμαπόδοση}_{capacity=20} &= \frac{20}{2} = 10\frac{txs}{s} \end{aligned}} \tag{2}$$

5 Κλιμακωσιμότητα του συστήματος

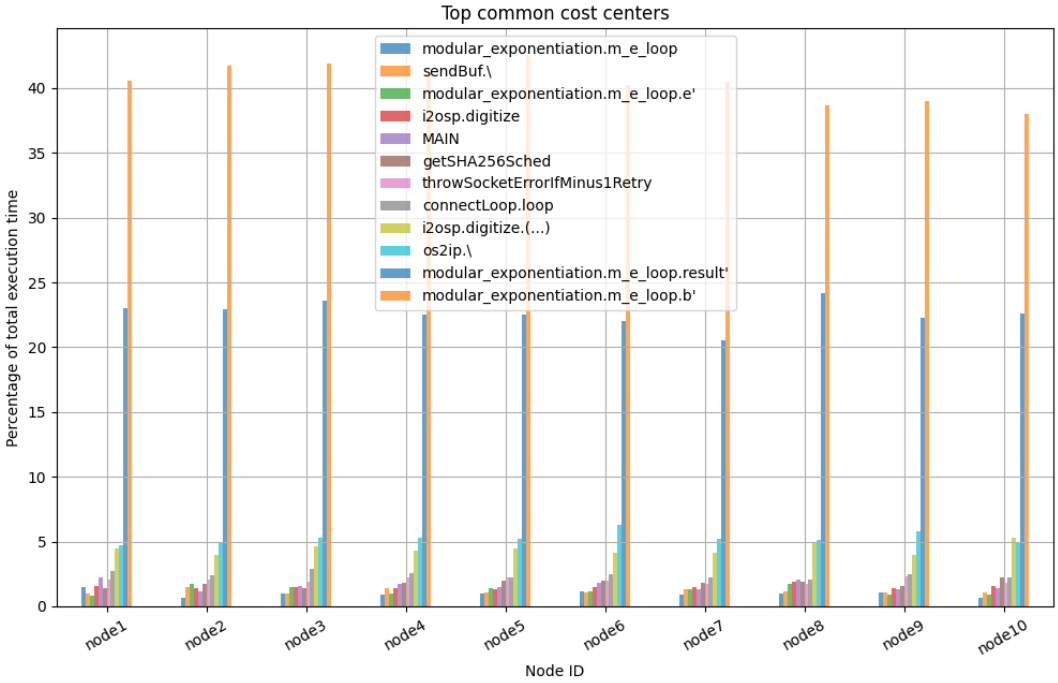
Στο πείραμα κλιμακωσιμότητας, το δίκτυο εκκινείται με 10 κόμβους, καθένας εκ των οποίων εκτελεί 1 staking συναλλαγή, με stake 10 BCC και 100 συναλλαγές (συγκεκριμένα αποστολές μηνυμάτων) προς τους άλλους κόμβους. Σκοπός είναι να εξεταστεί η κλιμάκωση του συστήματος ως προς το πλήθος των συμμετεχόντων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

5.1 Χρονοβόρα τμήματα του κώδικα

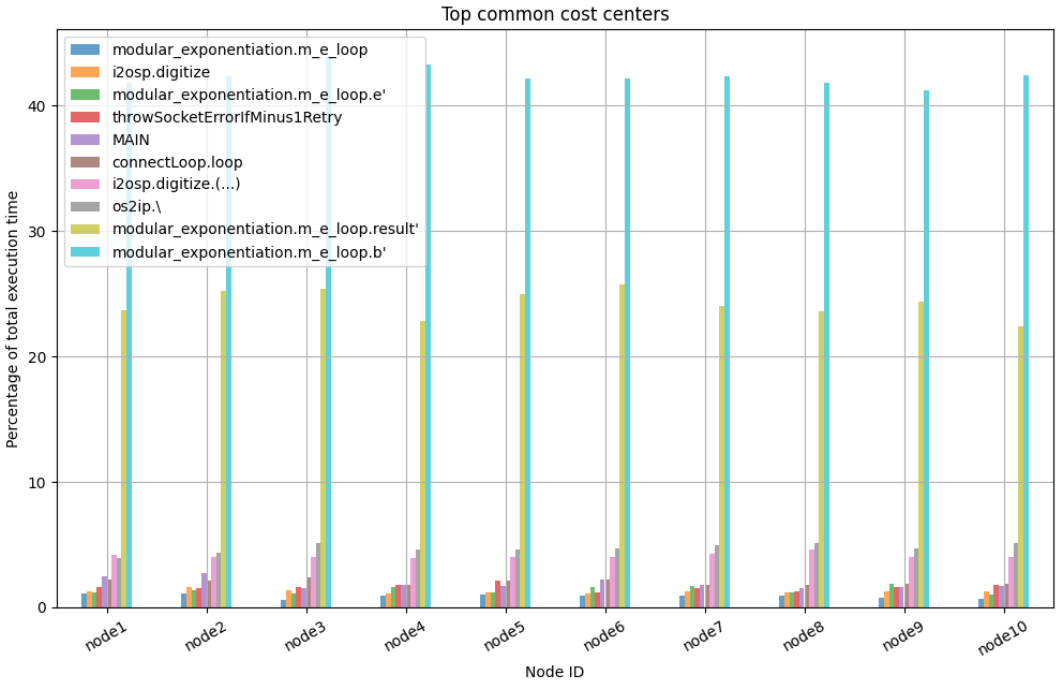
Στα γραφήματα 5 φαίνονται τα πιο χρονοβόρα κομμάτια του κώδικα για κάθε πείραμα κλιμακωσιμότητας. Φαίνεται ότι αυτά είναι τα ίδια με τα πιο χρονοβόρα κομμάτια του κώδικα για το πείραμα ρυθμαπόδοσης, με την συνάρτηση modular_exponentiation να καταλαμβάνει πάλι περίπου το 45% του συνολικού CPU time του προγράμματος.



(α′) capacity=5



(β′) capacity=10

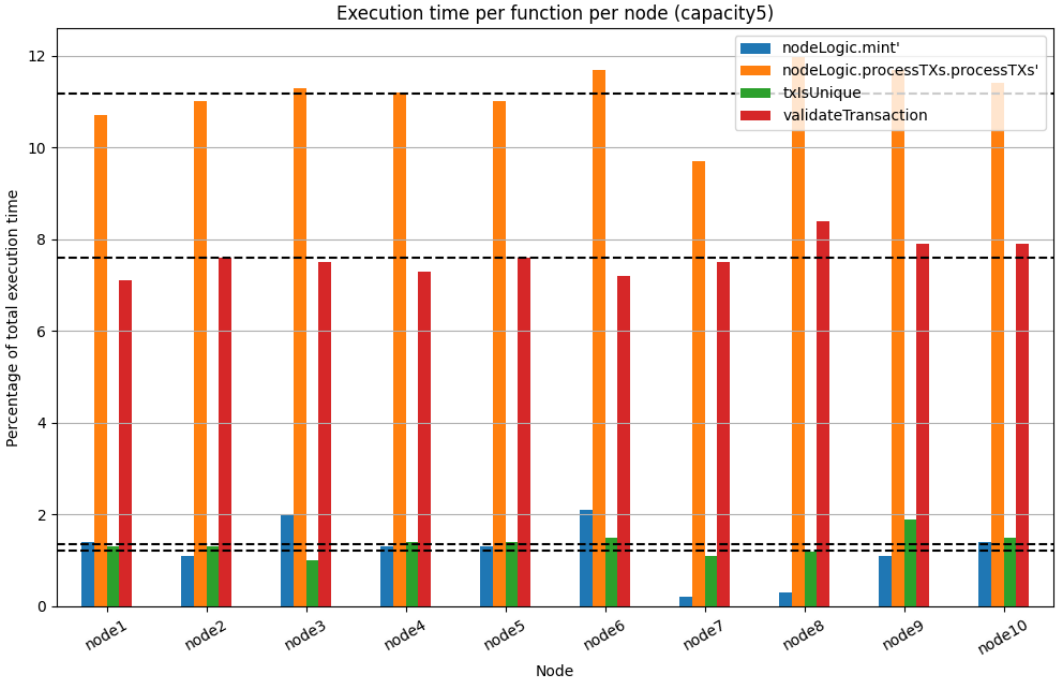


(γ′) capacity=20

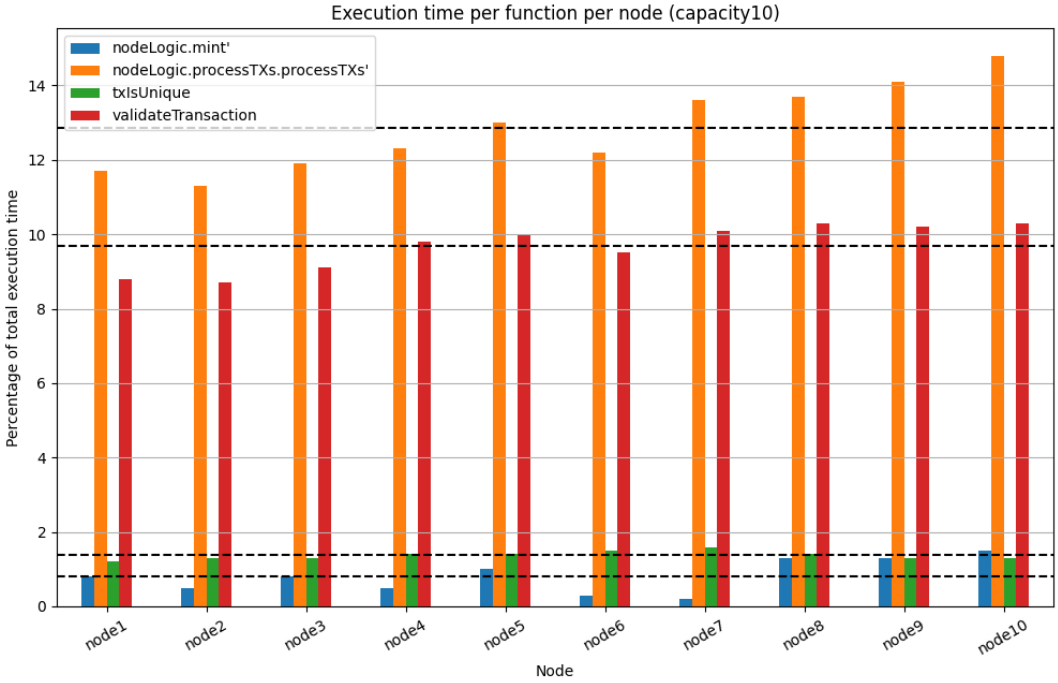
Σχήμα 5: Τα πιο χρονοβόρα κομμάτια του κώδικα

5.2 Συναρτήσεις του συστήματος

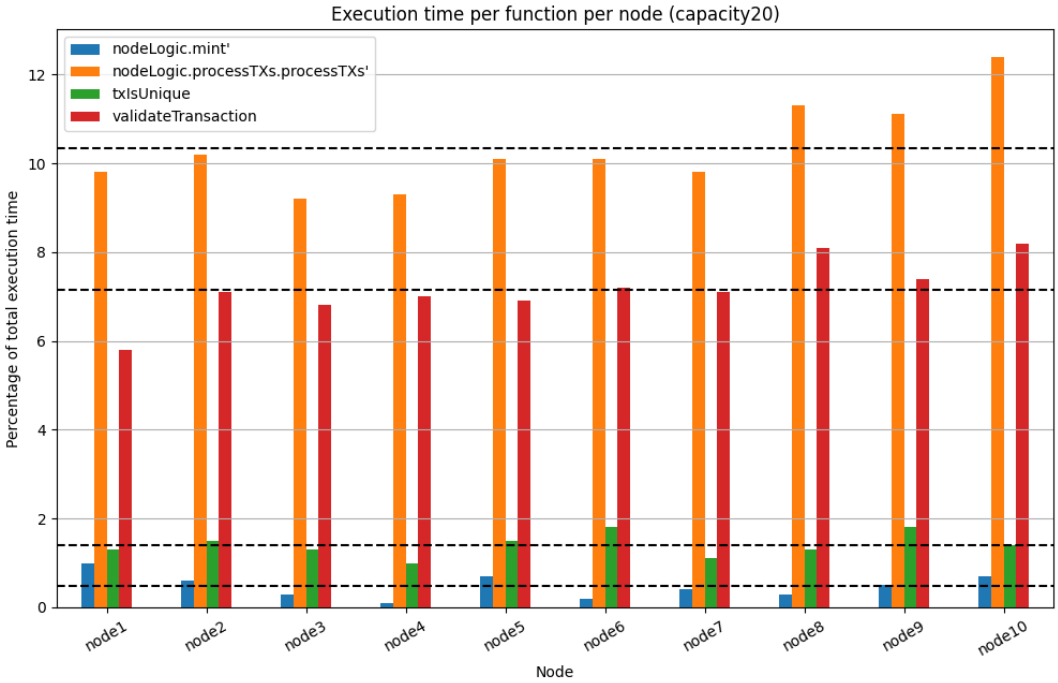
Στα γραφήματα 6 παρατηρείται ότι οι συναρτήσεις καταλαμβάνουν περίπου το ίδιο ποσοστό χρόνου εκτέλεσης με το προηγούμενο πείραμα.



(α΄) capacity=5



(β′) capacity=10



(γ′) capacity=20

Σχήμα 6: 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
node4.prof:	nodeLogic.mint’	100	1.3
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 μπλοκς επί την εκάστοτε χωρητικότητα.

$$\begin{aligned} &\text{Χωρητικότητα} = 5 \Rightarrow \text{Μπλοκς} = 100 \text{ και } \Sigma\text{υναλλαγές} = 500 \\ &\text{Χωρητικότητα} = 10 \Rightarrow \text{Μπλοκς} = 50 \text{ και } \Sigma\text{υναλλαγές} = 500 \\ &\text{Χωρητικότητα} = 20 \Rightarrow \text{Μπλοκς} = 25 \text{ και } \Sigma\text{υναλλαγές} = 500 \end{aligned} \tag{3}$$

5.3 Ρυθμαπόδοση και Block time

Στο σχήμα 7 φαίνονται οι μέσοι χρόνοι δημιουργίας block όπως υπολογίστηκαν από κάθε κόμβο.

```
capacity5/node10.log:Mean time between blocks: 550.0
capacity5/node1.log:Mean time between blocks: 550.0
capacity5/node2.log:Mean time between blocks: 550.0
capacity5/node3.log:Mean time between blocks: 550.0
capacity5/node4.log:Mean time between blocks: 550.0
capacity5/node5.log:Mean time between blocks: 550.0
capacity5/node6.log:Mean time between blocks: 550.0
capacity5/node7.log:Mean time between blocks: 322.2111111111111
capacity5/node8.log:Mean time between blocks: 322.2111111111111
capacity5/node9.log:Mean time between blocks: 550.0

capacity10/node10.log:Mean time between blocks: 1059.98
capacity10/node1.log:Mean time between blocks: 1059.98
capacity10/node2.log:Mean time between blocks: 1059.98
capacity10/node3.log:Mean time between blocks: 1059.98
capacity10/node4.log:Mean time between blocks: 1059.98
capacity10/node5.log:Mean time between blocks: 1059.98
capacity10/node6.log:Mean time between blocks: 1059.98
capacity10/node7.log:Mean time between blocks: 1059.98
capacity10/node8.log:Mean time between blocks: 1059.98
capacity10/node9.log:Mean time between blocks: 1059.98

capacity20/node10.log:Mean time between blocks: 1708.375
capacity20/node1.log:Mean time between blocks: 1708.375
capacity20/node2.log:Mean time between blocks: 1708.375
capacity20/node3.log:Mean time between blocks: 1708.375
capacity20/node4.log:Mean time between blocks: 1136.4545454545455
capacity20/node5.log:Mean time between blocks: 1708.375
capacity20/node6.log:Mean time between blocks: 1708.375
capacity20/node7.log:Mean time between blocks: 1708.375
capacity20/node8.log:Mean time between blocks: 1708.375
capacity20/node9.log:Mean time between blocks: 1708.375
```

Σχήμα 7: Μέσος χρόνος δημιουργίας block (ms)

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

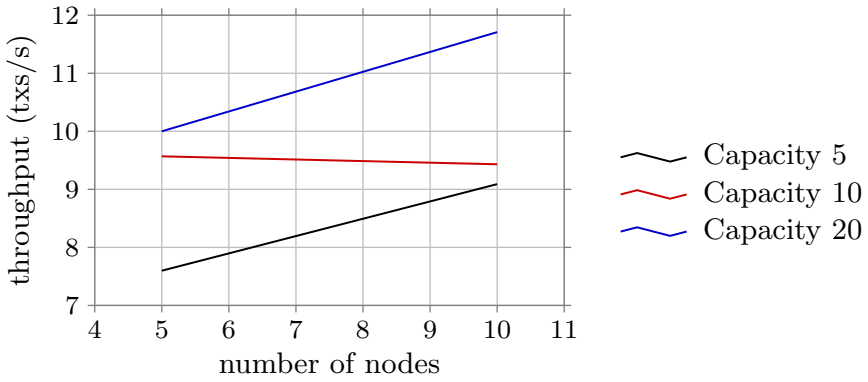
$$\text{Ρυθμαπόδοση} = \frac{\text{Συναλλαγές}}{\text{Χρόνος}} = \frac{\text{Συναλλαγές}}{\text{Μπλοκ}} \cdot \frac{\text{Μπλοκ}}{\text{Χρόνος}} \Leftrightarrow$$

$$\text{Ρυθμαπόδοση} = \frac{\frac{\text{Συναλλαγές}}{\text{Μπλοκ}}}{\text{Μέσος χρόνος δημιουργίας μπλοκ}} = \frac{\text{Χωρητικότητα}}{\text{Μέσος χρόνος δημιουργίας μπλοκ}} \Rightarrow \tag{4}$$

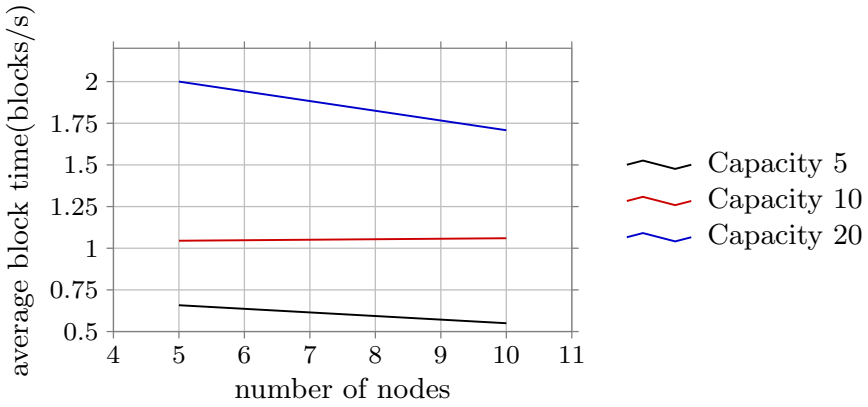
$$\boxed{\begin{aligned} \text{Ρυθμαπόδοση}_{capacity=5} &= \frac{5}{0,550} = 9,090\frac{txs}{s} \\ \text{Ρυθμαπόδοση}_{capacity=10} &= \frac{10}{1,060} = 9,4323\frac{txs}{s} \\ \text{Ρυθμαπόδοση}_{capacity=20} &= \frac{20}{1.708} = 11,709\frac{txs}{s} \end{aligned}}$$

Οι μετρήσεις από τα πειράματα συνοψίζονται στα γραφήματα 8.

(α′) Throughput του συστήματος



(β′) Μέσο block time του συστήματος

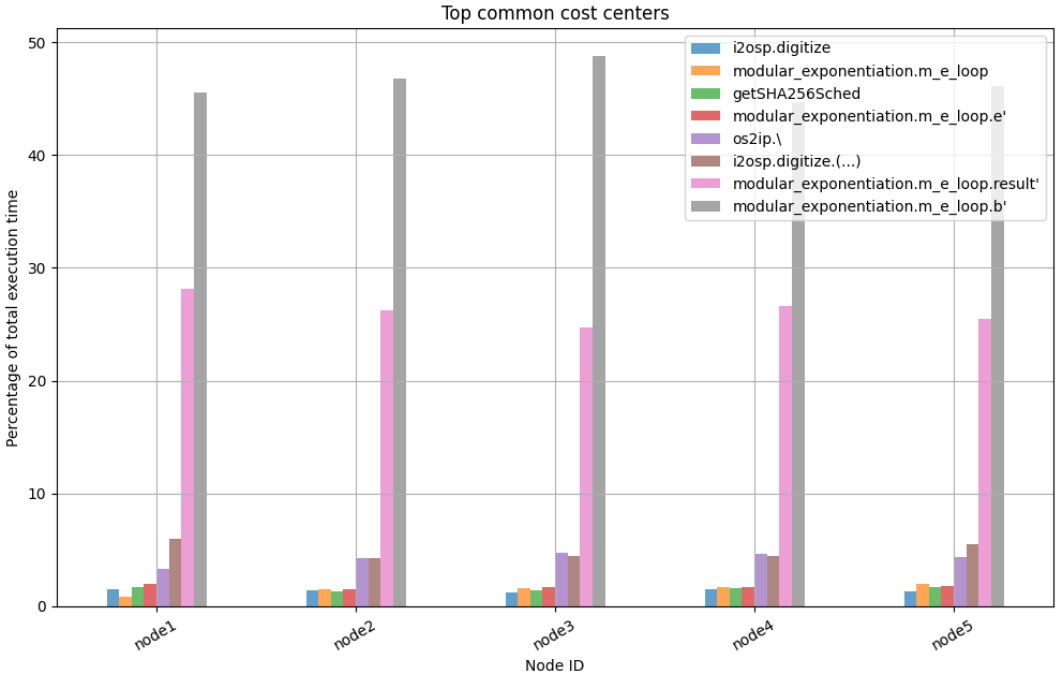


Σχήμα 8: Ρυθμαπόδοση και μέσος block time του συστήματος

Στα γραφήματα 8α′ και 8β′ 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 τις χωρητικότητες 5 και 20, αλλά για την 10 μένει οριακά σταθερός, με ελαφρώς φθίνουσα τάση. Ο δε μέσος χρόνος δημιουργίας block μειώνεται με τον αριθμό των κόμβων για τις χωρητικότητες 5 και 20, αλλά για την 10 μένει οριακά σταθερός, με ελαφρώς αύξουσα τάση. Αυτό δείχνει ότι το δίκτυο μπορεί να κλιμακώνει με τον αριθμό των κόμβων.

6 Δικαιοσύνη

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

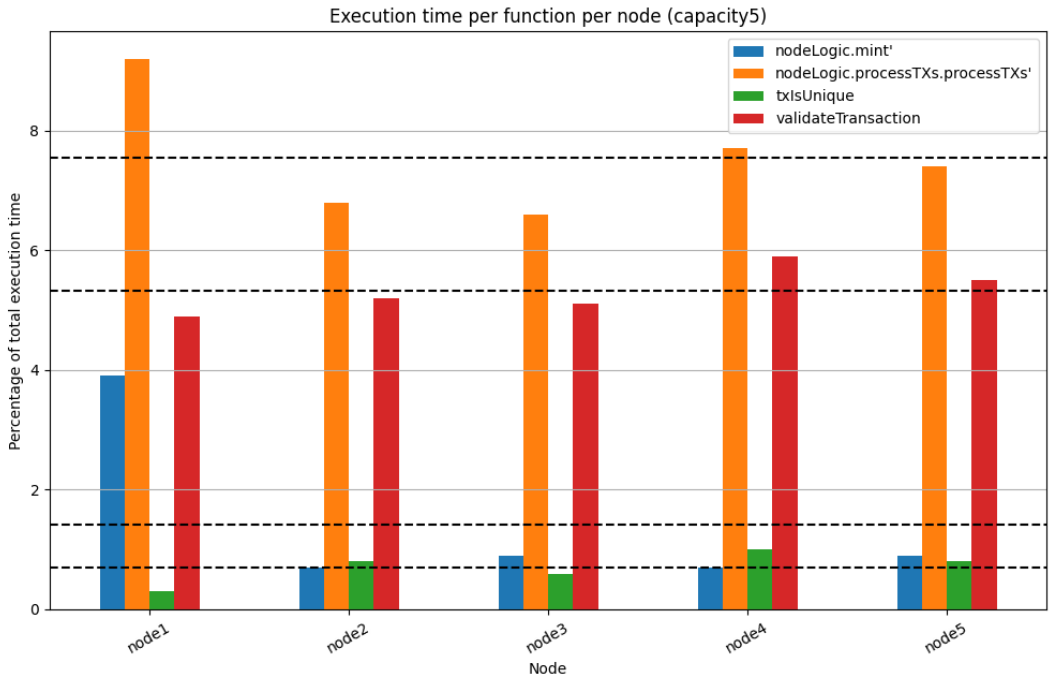


Σχήμα 9: Τα πιο χρονοβόρα κομμάτια του κώδικα capacity=5

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. Παρ'όλα 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 10.

Πίνακας 3: Στατιστικά συναρτήσεων ανά κόμβο capacity=5

Node	Function	Entries	TimeInh
node1.prof:	nodeLogic.processTXs.processTXs’	301	9.2
node1.prof:	validateTransaction	255	4.9
node1.prof:	txIsUnique	230	0.3
node1.prof:	nodeLogic.mint’	45	3.9
node2.prof:	nodeLogic.processTXs.processTXs’	301	6.8
node2.prof:	validateTransaction	255	5.2
node2.prof:	txIsUnique	226	0.8
node2.prof:	nodeLogic.mint’	45	0.7
node3.prof:	nodeLogic.processTXs.processTXs’	300	6.6
node3.prof:	validateTransaction	255	5.1
node3.prof:	txIsUnique	222	0.6
node3.prof:	nodeLogic.mint’	44	0.9
node4.prof:	nodeLogic.processTXs.processTXs’	300	7.7
node4.prof:	validateTransaction	255	5.9
node4.prof:	txIsUnique	223	1.0
node4.prof:	nodeLogic.mint’	44	0.7
node5.prof:	nodeLogic.processTXs.processTXs’	300	7.4
node5.prof:	validateTransaction	255	5.5
node5.prof:	txIsUnique	224	0.8
node5.prof:	nodeLogic.mint’	44	0.9



Σχήμα 10: Ποσοστό χρόνου επί του συνολικού χρόνου εκτέλεσης που λαμβάνει η κάθε συνάρτηση capacity=5

Στο σχήμα 10 φαίνεται, πράγματι, ότι ο κόμβος με το μεγαλύτερο stake καταναλώνει πολύ περισσότερο χρόνο στην συνάρτηση mint' σε σχέση με τους υπόλοιπους κόμβους, ενδεικτικό του γεγονός ότι πράγματι αυτός αναλαμβάνει συχνότερα την δημιουργία των νέων blocks. Μάλιστα, επισκοπώντας τα υπόλοιπα των λογαριασμών των κόμβων στο σχήμα 11, 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 11: Υπόλοιπα λογαριασμών κόμβων capacity=5