

Κατανεμημένα Συστήματα

Project :BlockChat

Όνομα: Γιαννάκη Βίκτωρας

AM: 03119707

Εξάμηνο: 9ο

Ομάδα : 6

GitHub link

<https://github.com/ntua-el19707/DistributedSystems>

Θέμα-Project:

Το θέμα της εργασίας είναι η κατασκευή ενός κατανεμημένου συστήματος Block Chain όπου τα nodes του συστήματος θα κάνουν transactions όπου θα αποθηκεύονται σε ένα Block Chain . Οι κομβοί θα μπορούν να κάνουν είτε αποστολή μηνύματος είτε μεταφορά νομισμάτων .

Παράμετροι του προβλήματος:

- Η κάθε μεταφορά coin θα στοιχίζει στον κόμβο όπου την κάνει 3% όπου το ποσό αυτό θα μεταφέρετε στο validator του block όπου θα αποθηκεύεται το transaction
- Η κάθε μεταφορά μηνύματος θα στοιχίζει όσο είναι το μέγεθος του μηνύματος π.χ. "hello world" κοστίζει 11 BCC(Block Chain Coins)
- Όταν το block γεμίζει θα εφαρμόζετε αλγόριθμος τύπου POS (proof of stake) για την εύρεση του επομένου validator

Επίλυση προβλήματος (Γενική ιδέα):

Η γενική ιδέα πίσω από την λύση μου είναι η εξής :

Ο κάθε κόμβος θα έχει το δικό πορτοφόλι με ένα ποσό όπου θα το χρησιμοποιεί για την έκδοση συναλλαγών. Ακόμη ο κόμβος θα έχει την δυνατότητα να εκδίδει συναλλαγές ταυτόχρονα . Για να πραγματοποιηθεί αυτό ,θα πρέπει ο κόμβος να έχει μια μεταβλητή όπου θα κρατάει την ακάθαρτη χρέωση του κόμβου (στον κώδικα "frozen") αυτά είναι coins όπου τα έχει χρεωθεί αλλά δεν τα έχει ακόμη αποθηκεύσει στο block chain . Όταν επικυρώνει ένα σετ συναλλαγών (συναλλαγή με συναλλαγή φόρου) τότε το δημοσιεύει στους κόμβους , όταν το σετ δημοσιευθεί τότε ο client λαμβάνει μήνυμα επιτυχίας.

Παράδειγμα:

- A έχει σύνολο 15 BCC , A -> B 10BCC
Ο κόμβος A θα αναλάβει την έκδοση του σετ (μεταφοράς και φόρου) , αρχικά θα υπολογίσει το ποσό μεταφοράς 97% 9.7 και 3% 0.3 έπειτα

σπάει σε 2 ρουτινές για την κατασκευή των συναλλαγών (Σημείωση: κατά την ακάθαρτη χρέωση θα πρέπει να **ισχύει** ύπαρξη semaphore κατά την αλλαγή το ποσού frozen). Ας υποθέσουμε ότι η ρουτίνα όπου θα εκδώσει την μεταφορά θα κάνει πρώτη χρέωση . Η ρουτίνα θεωρεί ότι ο χρήστης έχει τα λεφτά και παγώνει 9,7 από 0 -> 9.7 frozen μετα υπολογίζει το υπόλοιπο από την αλυσίδα βρίσκει 15 τότε κάνει $15 - 9.7 = 5.3 < 0 \Rightarrow \text{error else success}$. Με την ίδια λογική θα προχωρήσει και η δεύτερη ρουτίνα όπου στο κομμάτι του ελέγχου θα έχει $15 - 10 < 0$ ($0.3 + 9.7$ από πρώτη ρουτίνα). Σε ένα διαφορετικό παράδειγμα όπου το υπόλοιπο δεν επαρκεί και για τους δυο θα πρέπει να ακυρωθεί το transaction όπου έχει επιτύχει και να αποδεσμευτεί η χρέωση (η συναλλαγή που αποτυγχάνει αυτόματα αποδεσμεύει το frozen της). Αυτό συμβαίνει και στην γενικότερη περίπτωση όπου ο A επιχειρεί ταυτόχρονη έκδοση η μεταφορών όπου δεν επαρκεί για όλες το υπόλοιπο τότε οι ρουτινές που θα αντιληφθούν την υπερχρέωση θα ακυρωθούν και το σύστημα θα χρεωθεί μόνο της μεταφορές όπου επέτυχαν χρέωση.

Αφού γίνει η χρέωση τότε θα επιχειρείται η υπογραφή των συναλλαγών . Εάν αποτύχει τότε με την ίδια λογική θα ακυρώσει το σετ . Σε αυτό σημείο μπορούμε να δημοσιεύσουμε το σετ μεταφοράς .

Η μεταφορά των σετ γίνεται με την χρήση κατανεμημένου συστήματος RabbitMQ

Αυτό μας δίνει τα εξής πλεονεκτήματα

- Αποστολή σε ένα κόμβο του δικτιού όπου αυτός είναι υπεύθυνος για την αποστολή τους
- Το rabbitmq είναι στη ουσία μια ουρά όπου το μηνύματα καταναλώνεται με fifo αρά όλοι κομβοί λαμβάνουν τα μηνύματα με την ίδια σειρά όπου το rabbitmq τα έχει παραλάβει .

Επειδή η μεταφορά γίνεται με rabbitmq κερδίζουμε 1) ότι δεν χρειάζεται να γνωρίζουμε των validator αφού όλα τα transactions καταχωρούνται με τη ίδια σειρά από το block chain service όπου γνωρίζει το validator 2) αφού το σετ αποστάληκε και είναι έγκυρο δεν χρειάζεται να περιμένουμε απάντηση από τους άλλους κόμβους . Εάν ένας κόμβος αποτύχει και το σετ είναι έγκυρο τότε ο κόμβος αυτοκτονεί γιατί βρίσκετε σε μη αποδεκτή κατάσταση .

Για την μεταφορά μηνύματος ακολουθείτε παρόμοια διαδικασία όπου διαφέρει στο σημείο όπου αποθηκεύεται το μήνυμα. Αρχικά υπάρχει διαφορετική αλυσίδα για τα transactions τύπου μηνύματος. Κατά την έκδοση του σετ σε αυτή τη περίπτωση εκδίδεται ένα σετ για την χρέωση με κενούς τον παραλήπτη, validator(block chain coin tax To) και ένα transaction για το μήνυμα. Αφού αποθηκευτεί το μήνυμα εάν ο κόμβος είναι αυτός που το έχει εκδώσει τότε θέτει παραλήπτη των validator του message block, υπογραφεί το σετ και το ξανά δημοσιεύει για transaction-set-coins.

Λύση

Για την λύση της εργασίας έχω επιλέξει ως γλώσσα για την backend υποδομή την golang - v21.3 χωρίς κάποιο framework. Αντί για υλοποίηση cli υλοποιήθηκε frontend σε γλώσσα typescript με το framework Angular v17. Το frontend σερβίρεται από το κάθε node χωρίς να χρειαζόμαστε κάποιο Nginx για να το σερβίρουμε.

Για να τρέξει το σύστημα χρειάζεται ένα rabbirmq server. Στη συγκεκριμένη λύση χρησιμοποιείτε ένα docker container.

Στο repo στο github έχω δυο φακέλους backendService, frontend

Frontend: βρίσκεται ο κώδικας για το user interface,

backendService: backend

Μέσα στο backendService έχουμε 4 φακέλους

1. MessageSystem
2. entitys
3. graphQl
4. services

Τα αρχεία όπου βρίσκονται στη ριζά έχουν να κάνουν με το στήσιμο του http server και τους http controllers(handlersV1.go). Το MessageSystem είναι για την μεταφορά και κατανάλωση των μηνυμάτων. Το entitys περιέχει τα

data structures όπου χρησιμοποιούνται π.χ.transaction-coin , transaction-msg , blockCoin , blockMsg ,block Στα services υπάρχει ο κώδικας όπου υλοποιεί την λογική του συστήματος . Στο graphql υπάρχει ο κώδικας ώστε να χρησιμοποιήσουμε την βιβλιοθήκη graphql για να παίρνουμε τα δεδομένα χρησιμοποιώντας ένα endpoint

Entitys

Στο φάκελο entitys έχουμε ορίσει τη δομή των δεδομένων. Στο σύστημα έχουμε 2 τύπους transaction ο ένας είναι ο TransactionsCoins και ο άλλος ο TransactionMsg

```
6 type Client struct {
7     Address rsa.PublicKey `json:"address"`
8 }
9 type BillingInfo struct {
10     From Client `json:"from"`
11     To Client `json:"to"`
12 }
13 type TransactionDetails struct {
14     Bill BillingInfo `json:"bill"`
15     Nonce int `json:"nonce"`
16     Transaction_id string `json:"transaction_id"`
17     Created_at int64 `json:"created_at"`
18 }
19
20 type TransactionCoins struct{
21     BillDetails TransactionDetails `json:"transactionDetails"`
22     Amount float64 `json:"amount"`
23     Reason string `json:"reason"`
24 }
25 type TransactionMsg struct {
26     BillDetails TransactionDetails `json:"transactionDetails"`
27     Msg string `json:"Msg"`
28 }
29
```

Όπως φαίνεται από τον πιο πάνω ορισμό τα 2 transactions έχουν το ίδιο ορισμό για τα στοιχεία λογαριασμού και διαφέρουν στο τι αποθηκεύουν π.χ. στο TransactionCoins line 22-23 amount :coins , reason:("transfer" or "fee") . Επίσης στους πιο πάνω δεν έχουμε πεδίο για signature για τη συναλλαγή για το λόγο ότι για να υπογράψουμε την συναλλαγή πρέπει πρώτα να κάνουμε json.Stringfy τη συναλλαγή και το πεδίο signiture θα συμπεριλαμβανόταν στα bytes όπου θα υπογραφούν , εάν υπογραφή η συναλλαγή τότε θα πρέπει να θέσουμε την τιμή στο πεδίο καθιστώντας την υπογραφή άκυρη. Θα μπορούσε να οριστεί το signiture εάν το ορίζαμε χωρίς να το θέταμε ως json field τότε η json stringfy θα το αγνοούσε όμως κατά το στάδιο της μεταφοράς σε άλλους κόμβους θα πρέπει να ορίσουμε και άλλους τύπους ώστε να

μπορούμε να μεταφέρουμε και την υπογραφή . Για αυτό το λόγο ορίζουμε 2 ακόμη τύπους όπου είναι οι τύποι όπου μεταφέρουμε τα δεδομένα και τα services χρησιμοποιούν για έκδοση και επαλήθευση συναλλαγών. Ακόμη στα blocks αποθηκεύονται τους τύπους δεδομένων TransactionCoins , TransactionMsg αφού δεν χρειάζεται να αποθηκεύσουμε την υπογραφή αφού έχει ήδη επαληθευτεί . Τέλος η συναλλαγή διαβάζεται από From -> To (From:- , To:+).

```
53 type TransactionCoinEntityRoot struct {
54     Transaction TransactionCoins `json:"transaction"`
55     Signiture [] byte `json:"signiture"`
56 }
```

```
61 type TransactionMsgEntityRoot struct {
62     Transaction TransactionMsg `json:"transaction"`
63     Signiture [] byte `json:"signiture"`
64 }
```

Για να ορίσουμε τα blocks πρέπει αρχικά να ορίσουμε ένα γενικό τύπο για το block ώστε να μην χρειάζεται να ορίσουμε 2 φορές genesis , validateBlock και MineBlock .

```
12 // Type Block
13 type Block struct {
14     Index int `json:"index"`
15     CreatedAt int64 `json:"created_at"`
16     Validator rsa.PublicKey `json:"validator"`
17     Capicity int `json:"capicity"`
18     CurrentHash string `json:"current_hash"`
19     ParentHash string `json:"parrent_hash"`
20 }
```

Αυτός είναι ο ορισμός του block . Ορίζουμε το index ως το το index του block στην αλυσίδα , CreatedAt ως τον χρόνο σε unix όπου το block γίνεται mine , Validator το public key του κόμβου (όχι pem type: <https://pkg.go.dev/crypto/rsa#PublicKey>), Capicity το μέγεθος των συναλλαγών που μπορούν να αποθηκευτούν , CurrentHash το hash του block , ParentHash το hash του προηγούμενου block . Για τα 2 αυτά hashes το σύστημα απαιτεί να έχουν μήκος 64 λόγο του mining αλγόριθμου .

Κατά την γένεση του block θέτουμε το ParentHash="11...11" size:64 , CurrentHash ="random hash".

Παράδειγμα για κατανόηση πως δημιουργείται η ακολουθία των hash

Ας θέσουμε ParentHash ως A και CurrentHash ως B .

Αρχικά γίνονται 2 bit operations μεταξύ των A και B

Θέτουμε ως Γ το string που προκύπτει από το A XOR B

Θέτουμε ως Δ το int που προκύπτει από το A AND B

το hash του επομένου block προκύπτει από το sha256(Γ) Δ φορές .

Κατά την διαδικασία mine το capacity καθορίζεται από το capacity του προηγούμενου block.

Το blockCoin ορίζεται ως

```
66 // Type BlockCoin
67 type BlockCoinEntity struct {
68     BlockEntity Block `json:"block"`
69     Transactions []TransactionCoins `json:"transactions"`
70     workers int
71     perNode float64
72 }
73
74 /*
```

Το πεδίο BlockEntity περιέχει τα στοιχεία του block οπού αναφέρθηκαν πιο πάνω. Τα Transactions είναι τα coin transactions δημιουργείται με σταθερό μέγεθος = Block.Capacity συνεπώς δημιουργείται με zero values στη κάθε γραμμή του πίνακα

Go zero values

Int – 0

Float - 0

Bool - false

string – ""

rsa.PublicKey{nil , 0 }

Στην αλυσίδα θα πρέπει να έχει ένα σταθερό ποσό οπού θα ισούται με ένα ποσό perNode στην προκείμενη περίπτωση 1000 bcc * workers(ο αριθμός των κόμβων) . Επειδή κάθε μεταφορά μεταφορά coins απαιτεί 2 transactions fee , transfer μας βολεύει να απαιτήσουμε το μέγεθος του πίνακα να είναι **ζυγό** ώστε να γεμίζει πλήρως χωρίς να μένει κάποια γραμμή κενή η να

χρειάζεται να υπολογίσουμε φόρο για το φόρο . Συνεπώς κατά την γένεση δημιουργούνται 2 transactions όπου το κάθε transaction δίνει $\text{perNode} / 2 * \text{workers}$ στον validator . Εάν επιχειρηθεί αλυσίδα με μόνο μέγεθος πίνακα τότε η Genesis – επιστρέφει error .

το BlockMsg

```
223 type BlockMessage struct {
224     BlockEntity Block `json:"block"`
225     Transactions []TransactionMsg `json:"transactions"`
226 }
```

Το πεδίο BlockEntity περιέχει τα στοιχεία του block όπου αναφέρθηκαν πιο πάνω. Τα Transactions είναι τα msg transactions δημιουργείται με σταθερό μέγεθος = Block.Capacity συνεπώς δημιουργείται με zero values στη κάθε γραμμή του πίνακα . Στην προκείμενη περίπτωση δεν μας πειράζει αν το capacity είναι μόνο ή ζυγό . Τέλος σε αυτή την περίπτωση κατά την γένεση δεν έχουμε bootstrap transactions .

Services

Για την υλοποίηση τις λογικής του συστήματος χρησιμοποιούμε τα services και κάνουμε dependency injection . Δηλαδή π.χ. δημιουργούμε το walletService και το κάνουμε inject σε όλες τις services που το χρειάζονται όπως TransactionManagerService , blockchainService ...

Transaction Manager

```
16 type TransactionManagerService interface {
17     Service.Service // it is indeed a service
18     TransferMoney(to rsa.PublicKey, ammount float64) (entitys.TransactionCoinSet, error)
19     SendMessage(to rsa.PublicKey, msg string) (entitys.TransactionMessageSet, error)
20     unValidService() error
21 }
```

Δημιουργεί τα σετ transactions εάν επιτύχει επιστέφει το σετ για μεταφορά αλλιώς επιστρέφει error και αυτό με την σειρά του στέλνεται στο client

WalletService


```

24 type WalletService interface {
25     Service.Service
26     // will generate a wallet
27     GenerateWallet(size int, method func(io.Reader, int) (*rsa.PrivateKey, error)) error
28     Sign(transaction TransactionService) error
29     GetPub() rsa.PublicKey
30     Freeze(coins float64) error
31     UnFreeze(coins float64) error
32     GetFreeze() float64
33 }
24

```

Δημιουργεί ένα σετ public-private key , έχει την δυνατότητα να υπογράψει , να δεσμεύσει και να αποδεσμεύσει coins.

BlockchainCoinsService

```

79 // Block Coin Service Chain
80 type BlockchainCoinsService interface {
81     Service.Service
82     Genesis(capacity, workers int, perNode float64) error
83     FindBalance(key rsa.PublicKey) float64
84     findAndLock(coins float64) (float64, error)
85     GetTransactions(from, twoWay bool, keys []rsa.PublicKey, times []int64) []entity.TransactionCoins
86     InsertTransaction(t entity.TransactionCoinSet) error
87     RetriveChain() entity.BlockChainCoins
88     InsertNewBlock(block entity.BlockCoinEntity) error
89     SetWorkers(workers []rsa.PublicKey)
90     SetWhoAndQueue(who int, queueAndExchange RabbitMqService.QueueAndExchange)
91     SetStakeCoins(coins float64)
92     GetStakeCoins() float64
93 }
80

```

Με αυτό το service μπορούμε να επεξεργαστούμε την αλυσίδα καθώς και να αντλήσουμε δεδομένα από αυτήν . Κατά την προσθήκη ενός σετ συναλλαγών εάν το block έχει ήδη γεμίσει (index of transaction == capacity) τότε αυτόματα ο κόμβος οπου θα είναι ο miner κάνει broadcast το νέο block και μετά όλοι οι κομβοί συμπεριλαμβανομένου και του miner καταναλώνουν το block το κάνουν validate έπειτα το προσθέτουν , θέτουν το index =0 και συνεχίζουν και προσθέτουν τις συναλλαγές . Ακόμη η συνάρτηση θέτει στην συναλλαγή φόρου τον παραλήπτη των validator του block οπου θα τοποθετήσει την συναλλαγή .Επίσης η συνάρτηση κατά την διαδικασία εξόρυξης δεσμεύει coins(stake) το οποίο καθορίζει την πιθανότητα για να είναι αυτός ο miner εάν δεν έχει αρκετά τότε έχει 0% να είναι αυτός αφού γίνει το lottery spin() έπειτα το ποσό αποδεσμεύεται . Στην υλοποίηση το stake ονομάζεται ScaleFactor για το λόγο ότι είχε προηγηθεί διαφορετική υλοποίηση το ros στην οποία έφτιαχνε κατανομή με βάση το ποσό που έχει στείλει και λάβει ένα κόμβος . Στην προηγούμενη αυτή περίπτωση το ποσοστό αυτό καθόριζε την αξία των νομισμάτων που λάμβανε (0-1(0-100%) , 1>(100%>) αυτό έδινε διαφορετική σημασία στην κατανομή αφού αυξάνει η

μειώνει ανάλογα με τι κάνει ένα κόμβος. Γενικά οπού υπάρχει στον κώδικα `scaleFactor` σημαίνει ότι είναι `coins stake` ή `msg stake`. Για την ορθή λειτουργία θα πρέπει το `service` να τοποθετεί τα `transactions` με την σειρά που τα έχει λάβει για αυτό πριν επεξεργαστεί την αλυσίδα την κλειδώνει χρησιμοποιώντας `semaphore` ώστε στο σύστημα να επικρατεί συνοχή δεδομένων (να μην τοποθετεί συναλλαγές σε διαφορετικές θέσεις, να μην προσθέτει συναλλαγές καθώς διαβάζει το υπόλοιπο). Τέλος η `BlockchainCoins` είναι υπεύθυνη για την δέσμευση και διάβασμα υπολοίπου.

BlockchainMsg

```
385 type BlockchainMsgService interface {
386     Service.Service
387     Genesis(capacity int) error
388     GetTransactions(from, twoWay bool, keys []rsa.PublicKey, times []int64) []entity.TransactionMsg
389     InsertTransaction(t entity.TransactionMessageSet) error
390     RetriveChain() entity.BlockChainMessage
391     InsertNewBlock(block entity.BlockMessage) error
392     SetWorkers(workers []rsa.PublicKey)
393     SetWhoAndQueue(who int, queueAndExchange RabbitMqService.QueueAndExchange)
394 }
```

Το `service BlockchainMsg` δουλεύει με τον ίδιο τρόπο με κάποιες διάφορες. Αρχικά εάν το σεντ προέχεται από το ίδιο κόμβο τότε θέτει παραλήπτη των `validator` του `block` υπογράφει το σεντ και το δημοσιεύει. Άλλη διαφορά είναι ότι δεν είναι δυνατό να δεσμεύσουμε τα νομίσματα στο `stake` γιατί το `service` δεν έχει `walletService`. Στην περίπτωση της αλυσίδας αυτής θα αφήσουμε σταθερό και ίσο το `stake` με αποτέλεσμα να έχουμε ισοπιθανη κατανομή. Θα μελετήσουμε στην περίπτωση τις αλυσίδας νομισμάτων την αλλαγή κατανομής.

Async Load Service

```
11
12 type AsyncLoadService interface {
13     Service.Service
14     Consumer()
15 }
```

Αυτό το `service` είναι υπεύθυνο για την κατανάλωση και προσθήκη των συναλλαγών στην σωστή αλυσίδα. Δηλαδή καταναλώνει τα σεντ και ανάλογα με το τι καταναλώνει το προσθέτει στη σωστή αλυσίδα.

RabbitMqService

```

11 type RabbitMqService interface {
12     Service.Service
13     ConsumerTransactionsCoins()
14     ConsumerTransactionsMsg()
15     GetChannelTransactionCoin() chan entity.TransactionCoinSet
16     GetChannelTransactionMsg() chan entity.TransactionMessageSet
17     PublishBlockCoin(block entity.BlockCoinMessageRabbitMq) error
18     PublishBlockMsg(block entity.BlockMessageMessageRabbitMq) error
19     ConsumeNextBlockCoin() entity.BlockCoinMessageRabbitMq
20     ConsumeNextBlockMsg() entity.BlockMessageMessageRabbitMq
21     PublishTractioncoinSet(t entity.TransactionCoinSet) error
22     PublishTractionMsgSet(t entity.TransactionMessageSet) error
23     BroadCastSystemInfo(p entity.RabbitMqSystemInfoPack) error
24     ConsumeNextSystemInfo() entity.RabbitMqSystemInfoPack
25     PublishStake(stake entity.StakePack, Dst QueueAndExchange) error
26     ConsumeStake(Dst QueueAndExchange) entity.StakePack
27 }

```

Αυτό το service είναι υπεύθυνο για την κατανάλωση και δημοσίευση μηνυμάτων. Τα μηνύματα παραλαμβάνονται με την ίδια σειρά από όλους τους κόμβους. Αυτό συμβαίνει διότι το rabbitmq δίνει τα μηνύματα με την ίδια σειρά όπου τα έχει παραλάβει. Για να επιτύχουμε το broadcast θα πρέπει να δημοσιεύομαι ένα μήνυμα όχι σε ουρά αλλά δίνοντας του ένα exchange topic ο κάθε κόμβος δίνει μια ουρά δική του στο exchange topic έτσι το rabbitmq κάνει forward το μήνυμα στις αντίστοιχες ουρές αρά τα μηνύματα με το ίδιο topic τοποθετούνται με την ίδια σειρά σε όλες τις ουρές που έχουν δεσμευμένο το topic. Συνολικά το σύστημα υποστηρίζει 7 είδη μηνυμάτων, transaction-message-set, transaction-coin-set, block-coin, block-message, system-info, stake-coin-pack, stake-msg-pack. Αρά για την ορθή λειτουργία πρέπει να δώσω στο service τα 7 αυτά queue και τα topics όπου ο κόμβος θα καταναλώνει μηνύματα.

```

54 TransactionCoinSetQueueExchange := genSet("transactionCoins", "TCOINS", node)
55 TransactionMsgSetQueueExchange := genSet("transactionMsg", "TMSG", node)
56 BlockMsgQueueExchange := genSet("BlockCoins", "BCOIN", node)
57 BlockCoinQueueExchange := genSet("BlockMsg", "BMSG", node)
58 SystemInfoQueue := genSet("SystemInfo", "SINFO", node)
59 stakeCoinQueue = genSet("StakeCoins", "STCOIN", node)
60 stakeMsgQueue = genSet("StakeMsg", "STMMSG", node)

```

Η πρώτη παράμετρος είναι το όνομα της ουράς η δεύτερη το topic και η 3 το id του κόμβου ώστε να έχει μοναδικό όνομα για να καταναλώνει μόνο ο κόμβος που πρέπει τα μηνύματα. Η genSet δημιουργεί το σετ

π.χ. ;

id="1"

QueueAndExchange={queue:"transactionsCoins-1", exchange:"TCOINS"}

Μια διαφορά με τρόπο οπου καταναλώνονται τα μηνύματα είναι ότι για τα σετ συναλλαγών(transactionMsg , TransactionCoins) τα μηνύματα καταναλώνονται ασύγχρονα ενώ για τα άλλα είδη καταναλώνονται οπότε το σύστημα ζητήσει το μήνυμα.

System Info Service

```
type SystemInfoService interface {
    Service.Service
    AddWorker(body entitys.ClientRequestBody) error
    IsFull() bool
    BroadCast(sFm, sFc float64) error
    Consume() (error, float64, float64)
    IsOk() bool
    GetWorkers() []rsa.PublicKey
    NodeDetails(key rsa.PublicKey) (entitys.ClientInfo, int)
    Who(index int) (rsa.PublicKey, error)
    ClientList(key rsa.PublicKey) ([]entitys.ClientInfo, int)
    Nodes() []entitys.ClientInfo
}
```

Περιέχει δεδομένα για ούλους τους κόμβους. Τα δεδομένα που έχει είναι όλα τα public keys , τα id , τα private ip τους καθώς και την public ip μπορεί κάποιος να βρει των κόμβο . Από αυτό το service ο coordinator κάνει broadcast τα δεδομένα τους συστήματος.

Τα δεδομένα ενός κόμβου είναι τα εξής .

```
26 type ClientInfo struct {
27     Id          string `json:"nodeId"`
28     IndexId     int    `json:"indexId"`
29     Uri         string `json:"uri"`
30     UriPublic   string `json:"uriPublic"`
31 }
38 type ClientRequestBody struct {
39     Client      ClientInfo `json:"clientInfo"`
40     PublicKey   rsa.PublicKey `json:"key"`
41 }
```

Από αυτά η System Info Service δημιουργεί 2 μεταβλητές ένα table όπου για το κάθε IndexId έχει το publicKey του αντίστοιχου κόμβου και ένα map[string] όπου το κλειδί είναι το hash του public key και έχει ως τιμές τα στοιχεία ClientInfo.

Ο coordinator κάνει publish τα στοιχεία αυτά και αντίστοιχα οι υπόλοιποι τα καταναλώνουν με αποτέλεσμα όλοι οι κομβοί να έχουν τα στοιχεία όλων των κόμβων και να τα χρησιμοποιούν όπως επιθυμούν.

Η συνάρτηση IsOK() εμποδίζει να εξυπηρετα το rest api εάν δεν έχει αρχικοποιηθεί το σύστημα.

Η συνάρτηση IsFull() απαντά αν έχει πληροφορίες για όλους τους κόμβους

Register Service

```
14
15 type RegisterService interface {
16     Service.Service
17     Register()
18 }
19
```

Αυτό το service χρησιμοποιείτε κατά την εκκίνηση του κόμβου (Όχι coordinator) για να ενημερώσει τον coordinator την ύπαρξη του. Με τη σειρά του coordinator τον αποθηκεύει δίνοντας ένα indexId. Όταν ο coordinator λάβει μηνύματα για όσους κόμβους περιμένει τότε σταματά να δέχεται καινούργιους κόμβους στο σύστημα, δημοσιεύει τις πληροφορίες των κόμβων και αρχίζει να τους στέλνει το αρχικό ποσό π.χ. 1000 BCC επειδή το ποσό θα αποθηκευτεί στο block chain θα κοπεί 3% και θα δοθεί στον validator δηλαδή οι περισσότεροι κομβοί θα αρχίσουν με 970 και οι κομβοί όπου είναι validators θα επωφεληθούν τους φόρους. Εάν ο συνολικός αριθμός των workers * 2 > Capacity περισσότεροι από ένα validator αλλιώς validator ο coordinator.

BalanceService & Inbox Service

```
14
15 type BalanceService interface {
16     Service.Service
17     FindBalance(sender rsa.PublicKey) float64
18     FindAndLock(amount float64) (float64, error)
19     GetTransactions(keys []rsa.PublicKey, times []int64)
20     TransactionListCoin
21     GetChain() ChainCoinDTO
22     SetStake(coins float64)
23     GetStake() float64
24 }
25
26 type InboxService interface {
27     Service.Service
28     Send(keys []rsa.PublicKey, times []int64) (error, Inbox)
29     All(times []int64) (error, Inbox)
30     Receive(keys []rsa.PublicKey, times []int64) (error, Inbox)
31     SendAndReceived(keys []rsa.PublicKey, times []int64) (error, Inbox)
32     GetBlockChain() (ChainMsgDTO, error)
33 }
34
35 type InboxInn struct {
36     ...
37 }
38
```

Τα 2 αυτά services χρησιμοποιούνται κυρίως για την εξαγωγή δεδομένων από τις αλυσίδες. Όμως το Balance Service μπορεί να χρησιμοποιηθεί για να αλλάξει το stake του κόμβου και να δεσμεύσει coins. Ο λόγος για τον οποίο υπάρχουν τα 2 αυτά service είναι ότι τα blockchainService είναι private και οι controllers και resolvers δεν έχουν πρόσβαση σε αυτές.

Stake Service & SpinService


```

s/S/stake-service.go ~/D/v/D/D/b/s/L/lottery-service.go buffers
13 type StakeService interface {
14     Service.Service
15     distributionOfStake(weight float64) (map[rsa.PublicKey]float64, float64)
16     MapOfDistributesRoundUp(weight float64) (map[rsa.PublicKey]int, int)
17     GetCurrentHash() string
18     GetWorkers() []rsa.PublicKey
19 }

```

```

14 type LotteryService interface {
15     Service.Service
16     Spin(scaleFactor float64) (rsa.PublicKey, error)
17     LoadStakeService(stakeService Stake.StakeService)
18 }

```

Από αυτές της 2 services φτιάχνουμε τον αλγόριθμο για επιλογή του validator . Αρχικά κατασκευάζεται μια stake service έπειτα φορτώνεται μέσω LoadStakeService στην lottery-service και blockchainservice θα καλέσει την Spin() με το ποσό όπου δεσμευτικές και θα επιστραφεί το public key που επιλέχθηκε .

Όλες αυτές κατασκευάζονται στο αρχείο services.go και γίνονται load στο σύστημα , με το αρχείο main.go με την συνάρτηση BootOrDie()

Notes:

- Όταν ένας κόμβος εκκινεί εάν δεν είναι ο coordinator κάνει register και περιμένει εντολή από των coordinator για να συνεχίσει να αρχικοποιηθεί.
- Εάν υπάρχουν μηνύματα από προηγούμενες φορές το σύστημα θα καταρρεύσει γιατί θα λάβει μηνύματα από προηγούμενες φορές και θα έχει λάθος κατάσταση γιαυτο καλό είναι να γίνεται εκκαθάριση με τη χρήση script
- Εάν το σύστημα θα τρέξει πρώτη φορά καλό είναι να ξεκινήσουν πρώτα οι μη coordinator ώστε να δημιουργήσουν τις ουρές και έπειτα να αρχίσει ο coordinator (ο coordinator κάνει publish στην αρχικοποίησή του με αποτέλεσμα εάν δεν υπάρχουν οι ουρές οι workers να χάσουν τα αρχικά πακέτα εάν συμβεί αυτό χρήση script για εκκαθάριση και επανεκκίνηση).

ENV παράμετροι είναι οι παράμετροι που αποθηκεύουμε στο .env ώστε να δώσουμε τις απαραίτητες πληροφορίες που χρειάζεται ο κόμβος.

Env-coordinator

```
serverPort=severport  
workers=howmanyworkers  
coordinator=true  
nodeId="string-id"  
hostCoordinator=node-0  
myNetwork=node-0  
coordinatorPort=8000  
scaleFactorCoin=stakeCoins  
scaleFactorMsg=stakeMsg  
rabbitMQ=amqp://user:psassword@host:hostPort/  
CapacityBlockMsg=5  
CapacityBlockCoin=6  
publicUri=http://public-ip:port  
perNode=howMuchPerNode
```

Env-worker

```
serverPort=8000  
coordinator=false  
nodeId="string-id"  
hostCoordinator=node-0  
myNetwork=node-3  
coordinatorPort=8000  
publicUri= http://public-ip:port  
rabbitMQ= amqp://user:psassword@host:hostPort/
```

Post End Points :

Για να εισάγουμε δεδομένα στο σύστημα έχουμε 3 endpoints .

1. Transfer Coins : `http://public-ip:port/api/v1/transfer` =>
Request Body{

to //index id of receiver node

amount //coins to transfer

}
}
2. Transfer Msg : `http://public-ip:port/api/v1/send` =>
Request Body{

to //index id of receiver node

msg //message to send

}
}
3. Stake : `http://public-ip:port/api/v1/stake` =>
Request Body{

stake // new stake

}
}

Get Data:

Για την εξαγωγή δεδομένων χρησιμοποιούμε το graphql από το endpoint <http://public-ip:port/graphql?query=<graphql-query>>

Το σύστημα υποστηρίζει τα εξής ερωτήματα

Ερώτημα	Παράμετροι	Πεδία
self		client.nodeId , client.indexId , client.uri, client.uriPublic , total , stakeCoins
clients		nodeId , indexId , uri, uriPublic
allNodes		nodeId , indexId , uri, uriPublic
getTransactionsCoins //all transactions		Transactions.From , Transactions.To ,

		Transactions.Coins , Transactions.Reason , Transactions.Time , Transactions.Nonce , Transactions.TransactionId
nodeTransactions //node transactions		Transactions.From , Transactions.To , Transactions.Coins , Transactions.Reason , Transactions.Time , Transactions.Nonce , Transactions.TransactionId
inbox	Optional(From:number , TimeBefore:number , TimeAfter:number)	transactions.From , transactions.To , transactions.Msg , transactions.Time , transactions.Nonce , transactions.TransactionId
send	Optional(To:number , TimeBefore:number , TimeAfter:number)	transactions.From , transactions.To , transactions.Msg , transactions.Time , transactions.Nonce , transactions.TransactionId
allTransactionMsg	Optional(From:number , To:number , TimeBefore:number , TimeAfter:number)	transactions.From , transactions.To , transactions.Msg , transactions.Time , transactions.Nonce , transactions.TransactionId
blockChainCoins		block.index , block.validator , block.created_at , block.current_hash , block.parrent_hash , block.capacity, transactions.From , transactions.To , transactions.Coins , transactions.Reason , transactions.Nonce , transactions.Time , transactions.TransactionId
blockChainMsg		block.index , block.validator , block.created_at , block.current_hash , block.parrent_hash , block.capacity, transactions.From , transactions.To , transactions.Msg , transactions.Nonce , transactions.Time , transactions.TransactionId
balance	Optional(Node:number)	availableBalance

Παράδειγμα

Για να πάρουμε δεδομένα ποιος κόμβος ? , σε πιο ιδιωτικό δίκτυο ? , πόσοι συνολικά κομβοί? , ποσό υπόλοιπο έχει ? με ποιους μπορεί να κάνει συναλλαγές ?

Κατασκευάζεται το εξής: {

```
self{
  client{
    indexId , uri
  },
  total
},
clients{
  indexId , uri ,nodeId , uriPublic
},
balance{ availableBalance }
```

}

Ένα άλλο πιο απλό παράδειγμα ποσό έχει υπόλοιπο έχει ο κόμβος 0

```
{
  balance(Node:0){ availableBalance }
}
```

Frontend

Κάθε ένας κόμβος σερβίρει ένα web-api το οποίο μας δίνει την δυνατότητα να εκτελεί τα endpoints του συστήματος χρησιμοποιώντας ui .

- Σελίδα Home , endpoint /#/

Menu

Home

Transactions

Message Overview

Transaction Coins Overview

Forms


Fetch data

Nodes

BLOCK CHAT

Node vZs6wPCb

Index 10: 0 <http://node0.pocor>

1261 810000000000 

Node Transaction coins

Select From Client

1

Select To Client

Coins Min

Coins Max

Select To Reason


foo

Date Picker

MM/DD/YYYY - MM/DD/YYYY

Start Time

End Time

ID	From	To	Reason	Coins	Reason	Send Time
sBFRPaQ2WA	1	0	IN26	 0.00	Yes	21/03/2024, 14:28:59

Από αυτή την σελίδα μπορούμε να δούμε τα στοιχεία του κόμβου , το υπόλοιπο το καθώς και τα transactions coins οπου των αφορούν .Καθώς δίνεται και η δυνατότητα εφαρμογής φίλτρου ώστε να μπορούμε να φιλτράρουμε τις συναλλαγές ανάλογος με τα στοιχεία που μας ενδιαφέρουν .

- Σελίδα Inbox , endpoint#/inbox

Menu

Home

Transaction Message Overview

Inbox

Send

All Messages

Transaction Coins Overview

Forms

Fetch data

Nodes

BLOCK CHAT

Node vZsδwPCh INBOX

Select From Client

Message

Date Picker

Start Time

End Time

MM/DD/YYYY - MM/DD/YYYY

ID	From	Nonce	To	Message	Send Time
0YWwUK59fZ	4	335	0	Yes	21/03/2024, 19:33:25
K8GaaZ619k	3	366	0	There really is one	21/03/2024, 19:33:24
1ug8AgiZ6R	3	369	0	To Everything?	21/03/2024, 19:33:24
f574qVDxu1	2	286	0	do you have...er...	21/03/2024, 19:33:23
iAVz2loOKV	2	287	0	This has made a lot of people very angry.	21/03/2024, 19:33:23
99tBv4ZAaq	2	273	0	Er..good morning, O Deep Thought	21/03/2024, 19:33:23

Από αυτή την σελίδα μπορούμε να δούμε transaction msg οπου έχουν ως παραλήπτη των κόμβο .Καθώς δίνεται και η δυνατότητα εφαρμογής φίλτρου ώστε να μπορούμε να φιλτράρουμε τις συναλλαγές ανάλογος με τα στοιχεία που μας ενδιαφέρουν.

- Σελίδα Send , endpoint /#/send

Menu

Home

Transaction Message Overview

Inbox

Send

All Messages

Transaction Coins Overview

Forms

Fetch data

Nodes

BLOCK CHAT

Node vZsδwPCh SEND

Select To Client

Message

Date Picker

Start Time

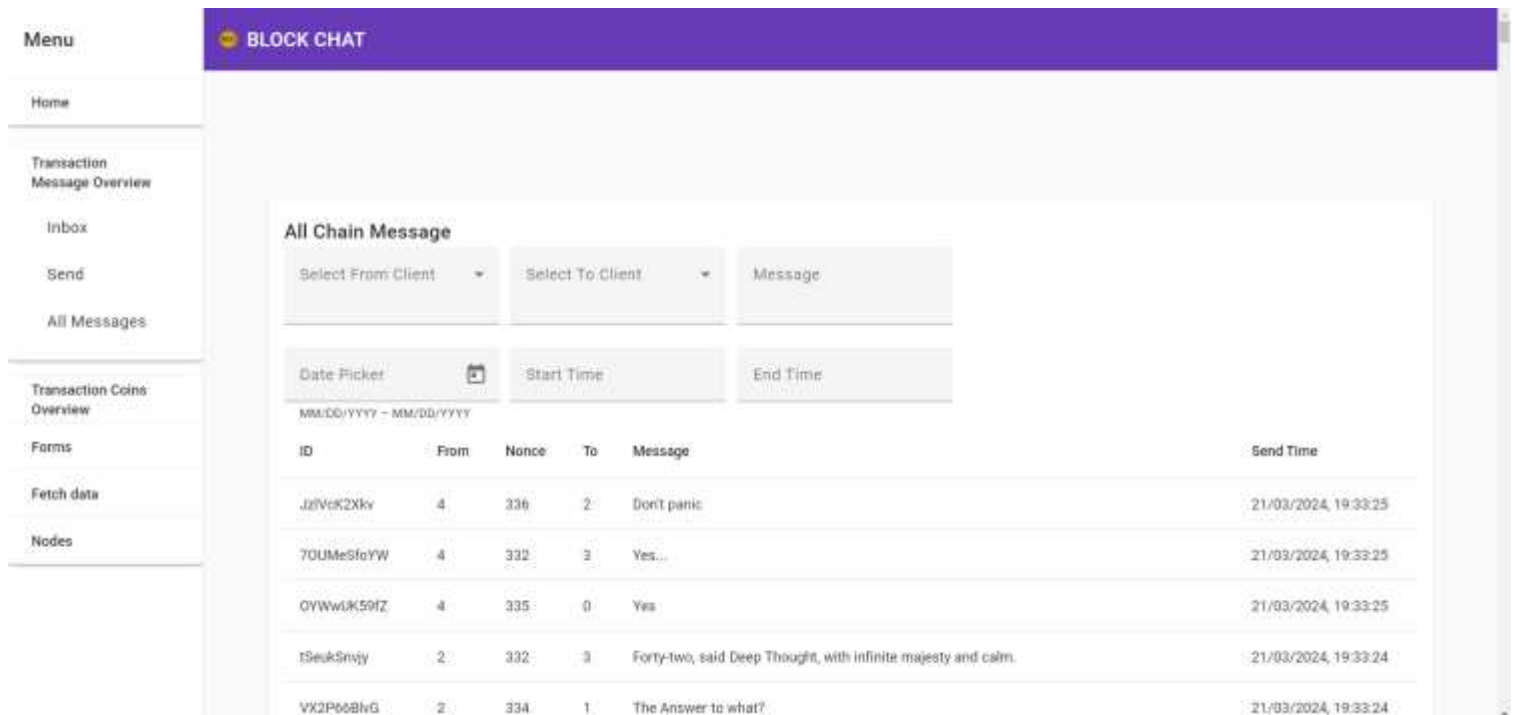
End Time

MM/DD/YYYY - MM/DD/YYYY

ID	From	Nonce	To	Message	Send Time
xIP6HClU9	0	237	2	This has made a lot of people very angry.	21/03/2024, 19:33:22
aWjQN7sCgT	0	229	1	yes I did	21/03/2024, 19:33:22

Από αυτή την σελίδα μπορούμε να δούμε transaction msg οπού έχουν ως αποστολέα των κόμβο .Καθώς δίνεται και η δυνατότητα εφαρμογής φίλτρου ώστε να μπορούμε να φιλτράρουμε τις συναλλαγές ανάλογος με τα στοιχεία που μας ενδιαφέρουν.

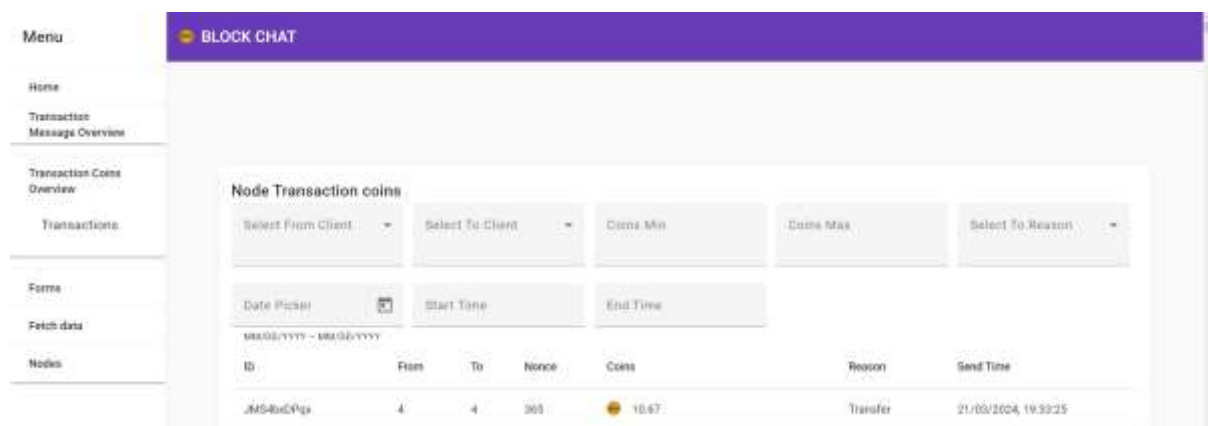
- Σελίδα All Messages, endpoint /#/allMessages



ID	From	Nonce	To	Message	Send Time
JzIVcK2Xkv	4	336	2	Don't panic	21/03/2024, 19:33:25
70UMeSfoYW	4	332	3	Yes...	21/03/2024, 19:33:25
OYWwUk59fZ	4	335	0	Yes	21/03/2024, 19:33:25
tSeukSnvyj	2	332	3	Forty-two, said Deep Thought, with infinite majesty and calm.	21/03/2024, 19:33:24
VX2P66BivG	2	334	1	The Answer to what?	21/03/2024, 19:33:24

Από αυτή την σελίδα μπορούμε να δούμε όλα τα transaction msg . Καθώς δίνεται και η δυνατότητα εφαρμογής φίλτρου ώστε να μπορούμε να φιλτράρουμε τις συναλλαγές ανάλογος με τα στοιχεία που μας ενδιαφέρουν.

- Σελίδα Transactions , endpoint /#/transactions



ID	From	To	Nonce	Coins	Reason	Send Time
JMS4u6Pqz	4	4	365	10.67	Transfer	21/03/2024, 19:33:25

Από αυτή την σελίδα μπορούμε να δούμε όλα τα transactions coins . Καθώς δίνεται και η δυνατότητα εφαρμογής φίλτρου ώστε να μπορούμε να φιλτράρουμε τις συναλλαγές ανάλογος με τα στοιχεία που μας ενδιαφέρουν .

- Σελίδα Transfer Coins , endpoint /#/transfer

The screenshot shows a web application interface with a purple header bar containing a hamburger menu icon, a yellow coin icon, and the text 'BLOCK CHAT'. The main content area is light gray and features a white rounded rectangle titled 'Transfer Coins'. Inside this rectangle, there is a dropdown menu labeled 'Select Client*' with a downward arrow, a text input field labeled 'Coins*', and a gray button labeled 'Transfer'.

Από αυτή την σελίδα μπορούμε να κάνουμε μεταφορά νομίσματος προς κάποιο κόμβο.

- Σελίδα Send Message , endpoint `/#/sendMessage`

The screenshot shows a web application interface with a purple header bar containing a hamburger menu icon, a yellow coin icon, and the text 'BLOCK CHAT'. The main content area is light gray and features a white rounded rectangle titled 'Send a message'. Inside this rectangle, there is a dropdown menu labeled 'Select Client*' with a downward arrow, a text input field labeled 'Msg*' with a small blue icon on the right, and a gray button labeled 'send'.

Από αυτή την σελίδα μπορούμε να κάνουμε αποστολή μηνύματος προς κάποιο κόμβο.

- Σελίδα Change Stake , endpoint `/#/changeStake`

Change Stake

Coins*

Set Stake

Από αυτή την σελίδα μπορούμε να θέσουμε διαφορετική τιμή για το stake coins του κόμβου.

- Σελίδα PlayGround , endpoint `/#/playground`

BLOCK CHAT

Ask GraphQL

GraphQL Query*

```

{
  self {
    client {
      indexId ,
      nodeId ,
      uriPublic ,
    },
    total ,
    stakeCoins
  },
  clients {
    uriPublic
  },
  balance(Node:2){
    availableBalance
  }
}

```

Ask

```

{
  "data": {
    "balance": {
      "availableBalance": 1175.85000000000006
    },
    "clients": [
      {
        "uriPublic": "http://83.212.73.244:8004"
      },
      {
        "uriPublic": "http://83.212.73.244:8006"
      },
      {
        "uriPublic": "http://83.212.73.244:8002"
      },
      {
        "uriPublic": "http://83.212.73.244:8008"
      }
    ],
    "self": {
      "client": {
        "indexId": 0,
        "nodeId": "vZs6wPCh",
        "uriPublic": "http://83.212.73.244:8000"
      },
      "stakeCoins": 10,
      "total": 5
    }
  }
}

```

Από αυτή την σελίδα μπορούμε να εκτελούμαι graphql queries ώστε να παίρνουμε δεδομένα από τον κόμβο.

Πειραματική διαδικασία

Λόγο ότι το σύστημα έχει 2 αλυσίδες για την πειραματική διαδικασία θα μεταβάλλω το capacity block Message . Θα αρχίσω να στέλνω από 50 transactions Message παράλληλα και θα παίρνω το χρόνο που χρειάστηκε για να πάρω ότι για όλα πήρα απάντησή αγνοώντας θετική η αρνητική απάντηση. Ο λόγος που επιλέγεται transactions Message για το πείραμα είναι ότι η έκδοση του προκαλεί και έκδοση transactions coins αρά σε ένα σύστημα 5 κόμβων με 5 μέγεθος block για 50 transactions το RabbitMq θα λάβει το ελάχιστο αγνοώντας το mine της block coin $(50 \times 5 * 2(\text{chains}) + (50 / 5 (\text{mines})) * 5(\text{message}) * 5 (\text{workers}) + (50 / 5) \text{ blocks}) = 760$,αυτό σημαίνει ότι συνολικά στο δίκτυο έχω κίνηση 1010 οπου ο κόμβος του rabbitMq λαμβάνει το 75% . Σε κάθε επανάληψη θα αυξάνεται ο αριθμός το transactions Message . Έπειτα θα φορτωθούν 10 φορές οι συναλλαγές οι οποίες μας δοθήκαν από την εκφώνηση και θα καταγραφεί ο χρόνος ο οποίος χρειάστηκε για να εκδοθούν οι συναλλαγές , ο μέσος ορός αυτού το χρόνου θα χρησιμοποιηθεί για την κατασκευή των γραφημάτων . Μετα θα επαναλαμβάνω τη διαδικασία 2 φορές μια με capacity 10 και μια με 20 μειώνοντας έτσι το πλήθος μηνυμάτων stake . Τέλος θα επαναλάβουμε την διαδικασία για 10 κόμβους οπου για 50 transactions $(50 * 10 * 2 + 50 / 10 * 10 * 10 + 50 / 10)$ 1510 από 2010 75% publish στο RabbitMq. Το capacity to block coin θα διατηρηθεί σταθερο στο 22 .

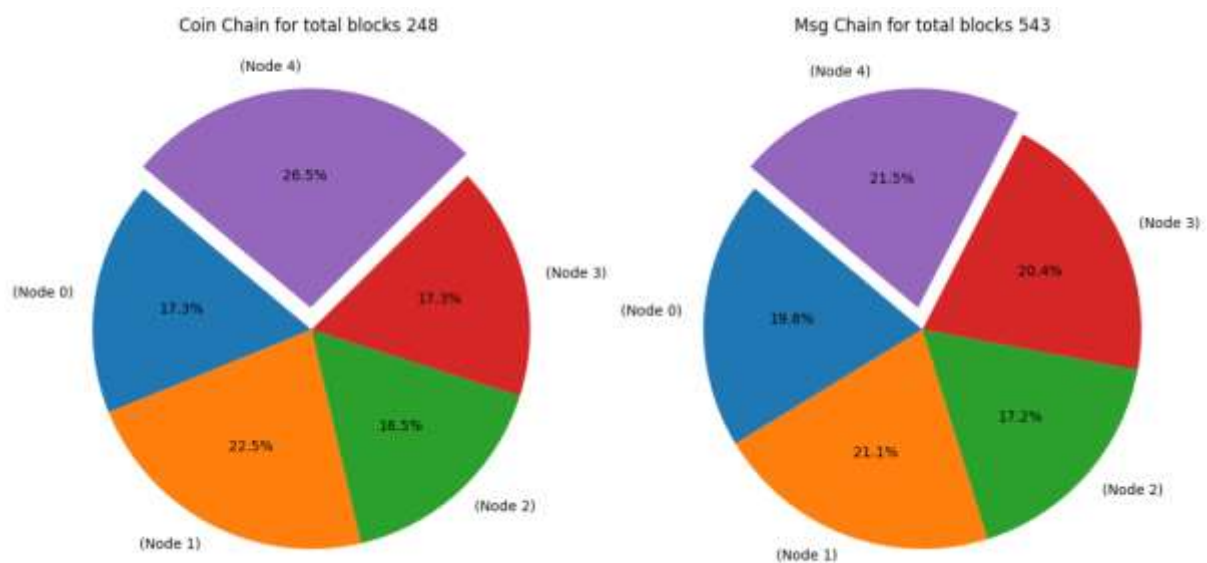
Κομβόι 5 , Capacity: 5 , Stake : StakeV3BCC (10 coins each)

Total Request Per Node	Total Time For All to Respond	Comments
50	1.3555642s	Μετρήσεις από το δίκτυο στο πολυτεχνείο(19/3/2024)
100	2.3647985s	
200	3.9583834s	
300	4.9954515s	
400	6.3105972s	
500	9.0804973s	
600	9.6170755s	
700	13.4329311s	
800	17.8031663s	
1000	17.4633777s	

Από τα transactions txt καταχωρήθηκαν συνολικά 240 transactions σε χρόνο 7.6165375s seconds send(request on anode after node respond). Όταν στάλθηκαν ταυτόχρονα χρειάστηκε 1.2911481s.

A/A	secs
1	1.2911481s.
2	1.2792486s
3	1.3272697s
4	2.0209575s
5	1.3342684s
6	1.4265926s
7	1.4056749s
8	1.4711083s
9	1.9462221s
10	1.648118s

Average Time:1.51506082



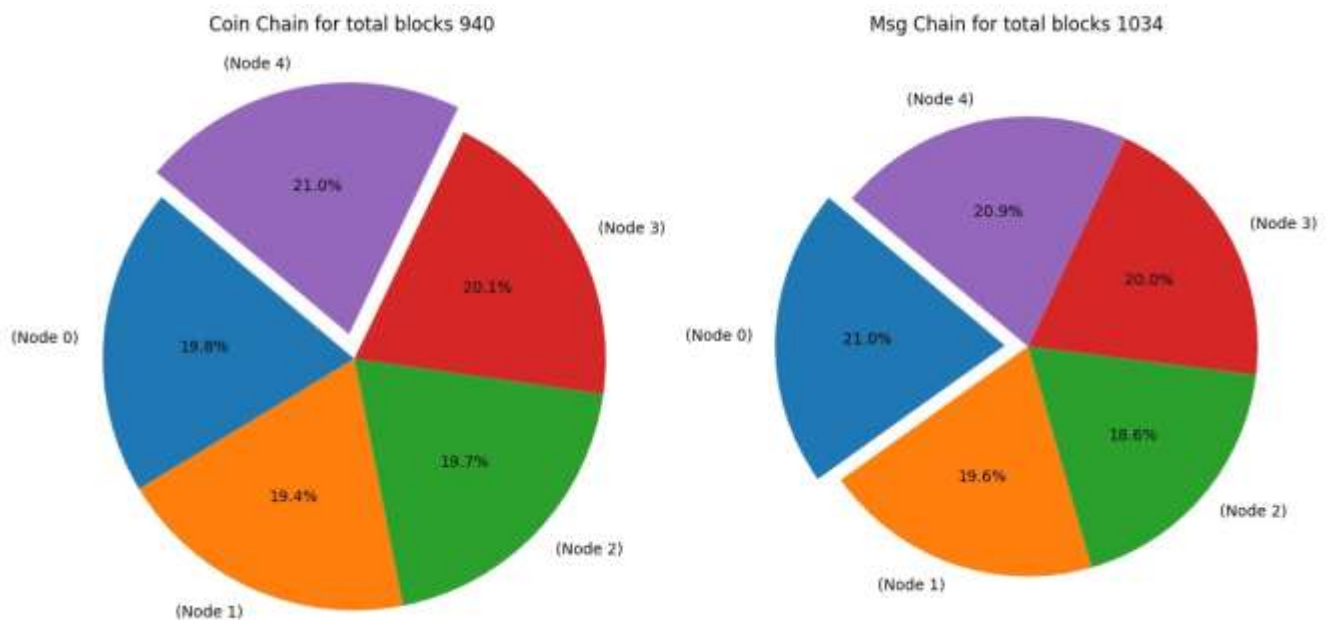
Αύξηση Block σε 10 capacity

50	1.5095963s	Μετρήσεις από το δίκτυο στο πολυτεχνείο(19/3/2024)
100	2.2475396s	
200	4.3143429s	
300	6.2049585s	
400	6.6287295s	
500	8.0631714s	
600	10.4905511s	
700	12.4364393s	
800	16.5826531s	
1000	22.7426737s	

Από τα transactions txt καταχωρήθηκαν συνολικά 240 transactions σε χρόνο 12.1221386s seconds send(request on anode after node respond). Όταν στάλθηκαν ταυτόχρονα χρειάστηκε 1.3548015s

A/A	secs
1	1.3548015s
2	2.0008336s
3	1.504822s
4	2.1517866s
5	1.5679828s
6	1.9097008s
7	1.6523225s
8	1.8142697s
9	1.7325364s
10	1.8880636s

Average time : 1.7577119499999998s



Αύξηση Block σε 20 capacity

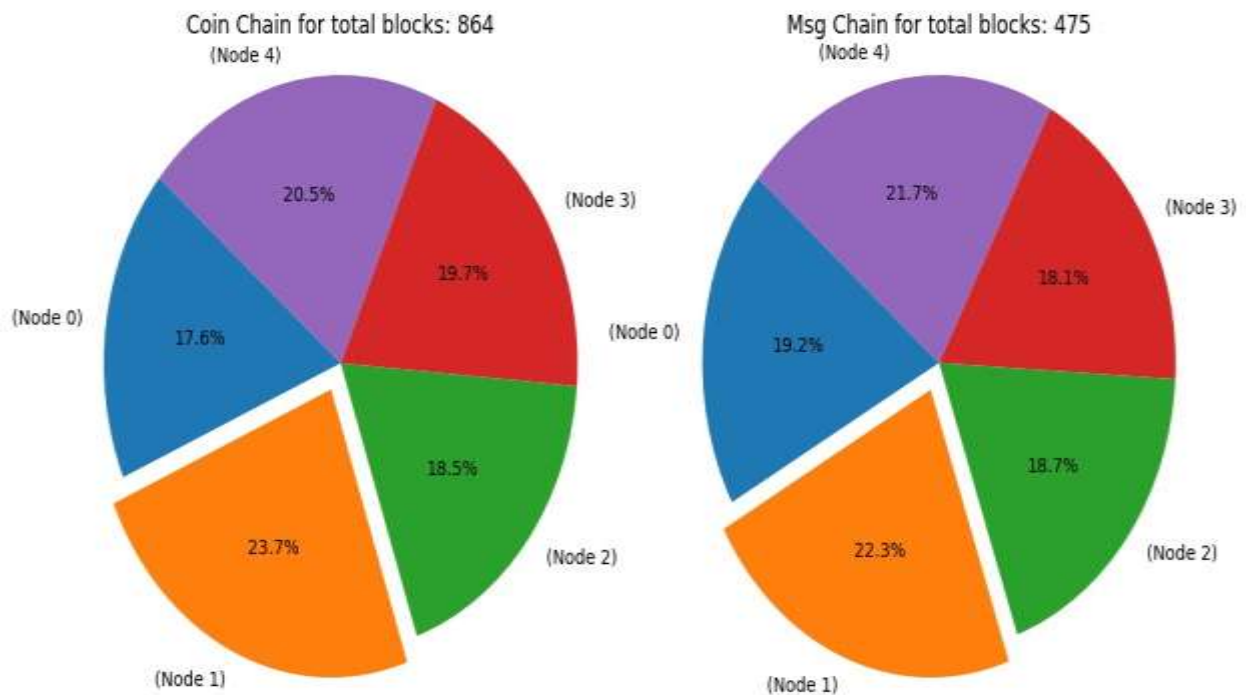
50	1.3742543s	Μετρήσεις από το δίκτυο στο πολυτεχνείο(20/3/2024)
----	------------	--

100	2.7175712s	
200	4.1300687s	
300	7.1155622s	
400	8.7472438s	
500	10.6061439s	
600	13.9300306s	
700	21.4376652s	
800	13.4797851s	
1000	809.8398ms	
2000	1m44.8468295s	
3000	2m57.8302764s	

Από τα transactions txt καταχωρήθηκαν συνολικά 240 transactions σε χρόνο 6.6064368s seconds send(request on anode after node respond). Όταν στάλθηκαν ταυτόχρονα χρειάστηκε 1.252184s

A/A	secs
1	1.252184s
2	704.2058ms
3	849.9416ms
4	742.8611ms
5	1.0128017s
6	665.5472ms
7	807.9661ms
8	661.6467ms
9	1.018863s
10	902.3557ms

Average time : 0.8618372899999999s



Μετρήσεις για 10 κόμβους

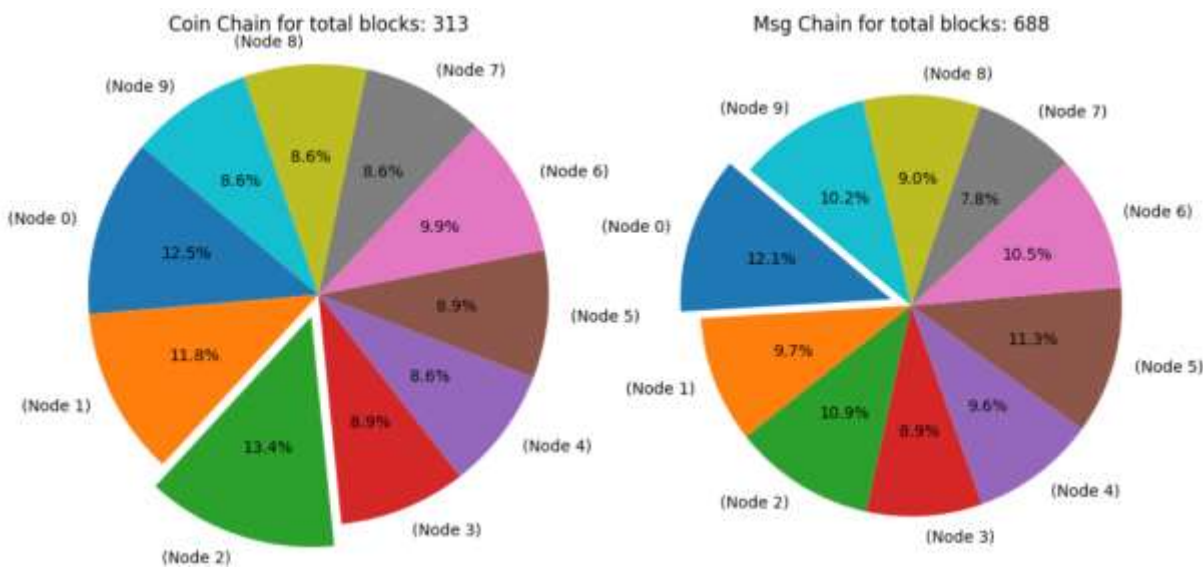
Block σε 5 capacity

50	2.813094s	Μετρήσεις από το δίκτυο στο πολυτεχνείο(20/3/2024)
100	5.2053686s	
200	7.5821856s	
300	9.6926081s	
400	17.3349435s	
500	28.5831521s	
600	26.3868513s	
700	38.0326742s	
800	38.547221s	Δε κάποιες περιπτώσεις ακυρώνονται συναλλαγές γτ στη στιγμή της έκδοσης ο κόμβος δεν έχει επαρκή υπόλοιπο
1000	51.7526421s	

Από τα transactions txt καταχωρήθηκαν συνολικά 1000 transactions σε χρόνο 48.5324582sseconds send(request on anode after node responed). Όταν στάλθηκαν ταυτόχρονα χρειάστηκε 5.2228672s

A/A	secs
1	5.2228672s
2	4.4216227s
3	3.9119728s
4	4.4522896s
5	4.870004s
6	4.1634199s
7	5.2830908s
8	4.4957376s
9	4.6695088s
10	5.9538913s

Average Time: 4.74444047sec



Μετρήσεις για 10 κόμβους

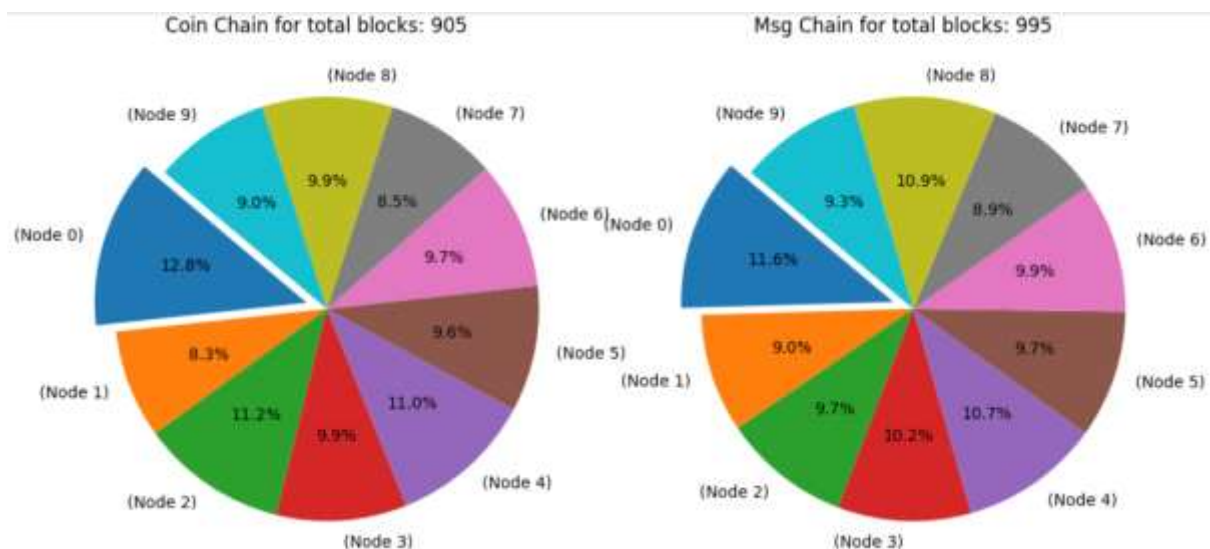
Block σε 10 capacity

50	2.8388638s	Μετρήσεις από το δίκτυο στο πολυτεχνείο(21/3/2024)
100	4.5228348s	
200	7.3108034s	
300	8.6021595s	
400	13.7619658s	
500	13.4186632s	
600	26.3613173s	
700	27.543964s	
800	36.075579s	
1000	44.636559s	

Από τα transactions txt καταχωρήθηκαν συνολικά 1000 transactions σε χρόνο 1m7.9359457s send(request on anode after node respond). Όταν στάλθηκαν ταυτόχρονα χρειάστηκε 5.7308939s

A/A	secs
1	5.7308939s
2	4.3468165s
3	4.8279493s
4	4.829785s
5	5.295001s
6	5.2586533s
7	4.9869522s
8	4.5355374s
9	4.9609742s
10	4.5475242s

Average = 4.9320087s



Μετρήσεις για 10 κόμβους

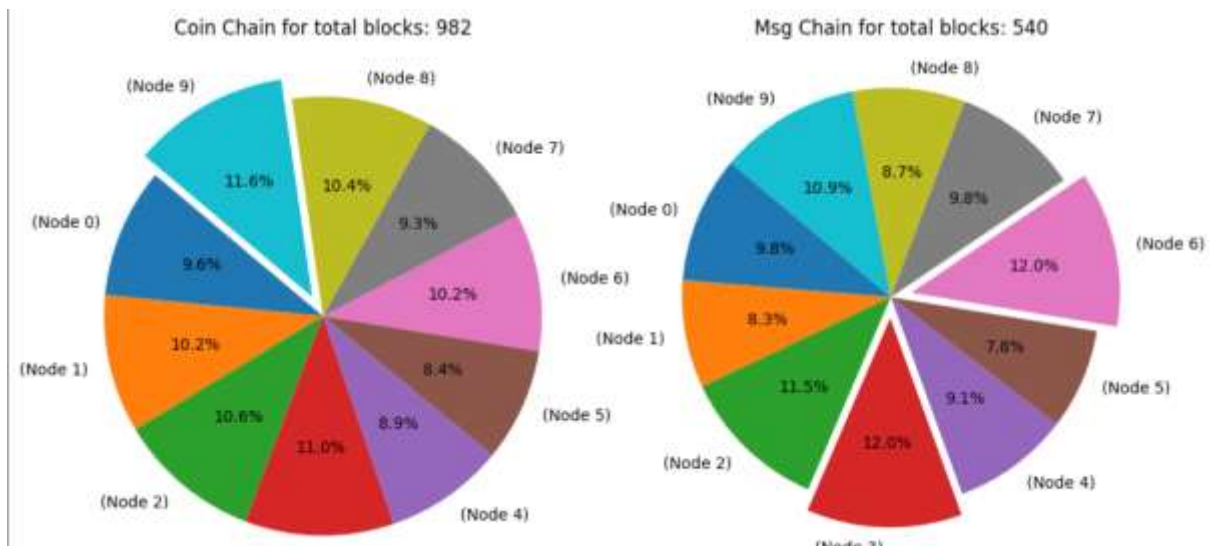
Block σε 20 capacity

50	3.2553084s	Μετρήσεις από το δίκτυο στο πολυτεχνείο(21/3/2024)
100	4.9917777s	
200	6.9559562s	
300	9.0720574s	
400	10.6651745s	
500	16.4560017s	
600	21.2427723s	
700	27.9160378s	
800	36.947591s	
1000	53.247156s	

Από τα transactions txt καταχωρήθηκαν συνολικά 1000 transactions σε χρόνο 59.800975s send(request on anode after node respond). Όταν στάλθηκαν ταυτόχρονα χρειάστηκε 5.8271585s

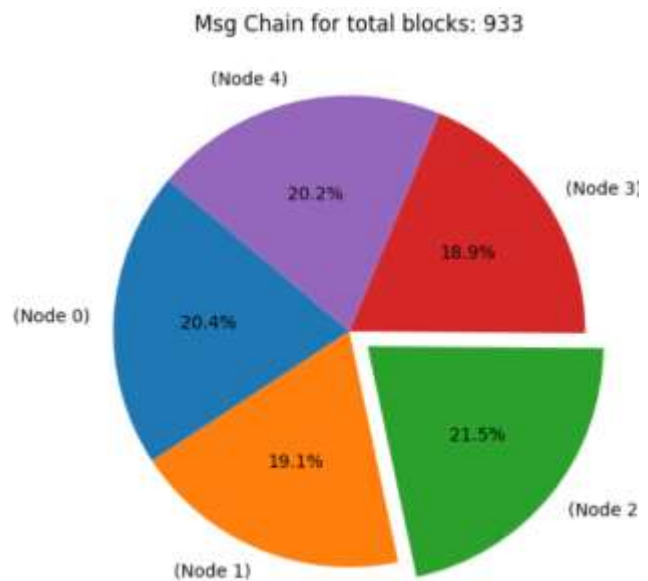
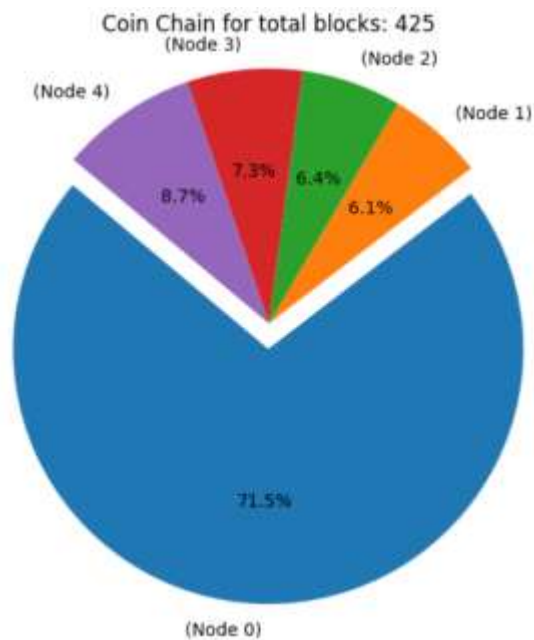
A/A	secs
1	5.8271585s
2	6.700013s
3	6.8778903s
4	7.2525955s
5	7.4505687s
6	4.9871886s
7	7.5938343s
8	6.2772469s
9	5.9096706s
10	7.2822921s

Average Time: 6.6158458499999995



Αύξηση το Stake at node 0 Από 10 σε 100

Block Capacity 5 Nodes-5



Παρατηρήσεις

Στο συγκεκριμένο σύστημα ένα block προσθέτετε ασύγχρονα από το σύστημα αυτό μας το καθιστά αδύνατο να το μετρήσουμε χρόνο για mining σε αντίθεση με την έκδοση συναλλαγών όπου μπορούμε να τις προκαλέσουμε καλώντας ένα endpoint. Για αυτό το λόγο δεν πήραμε μετρήσεις για block mining, όμως κατά τις πιο πάνω μετρήσεις παρατηρήθηκε πως στο σύστημα η διαδικασία mining καθυστερεί την διαδικασία αποθήκευσης των συναλλαγών και ότι το σύστημα με την αύξηση το capacity το σύστημα φαίνεται να έχει βελτιώσει και να επιταχύνει την διαδικασία αποθήκευσης των συναλλαγών στις αλυσίδες.

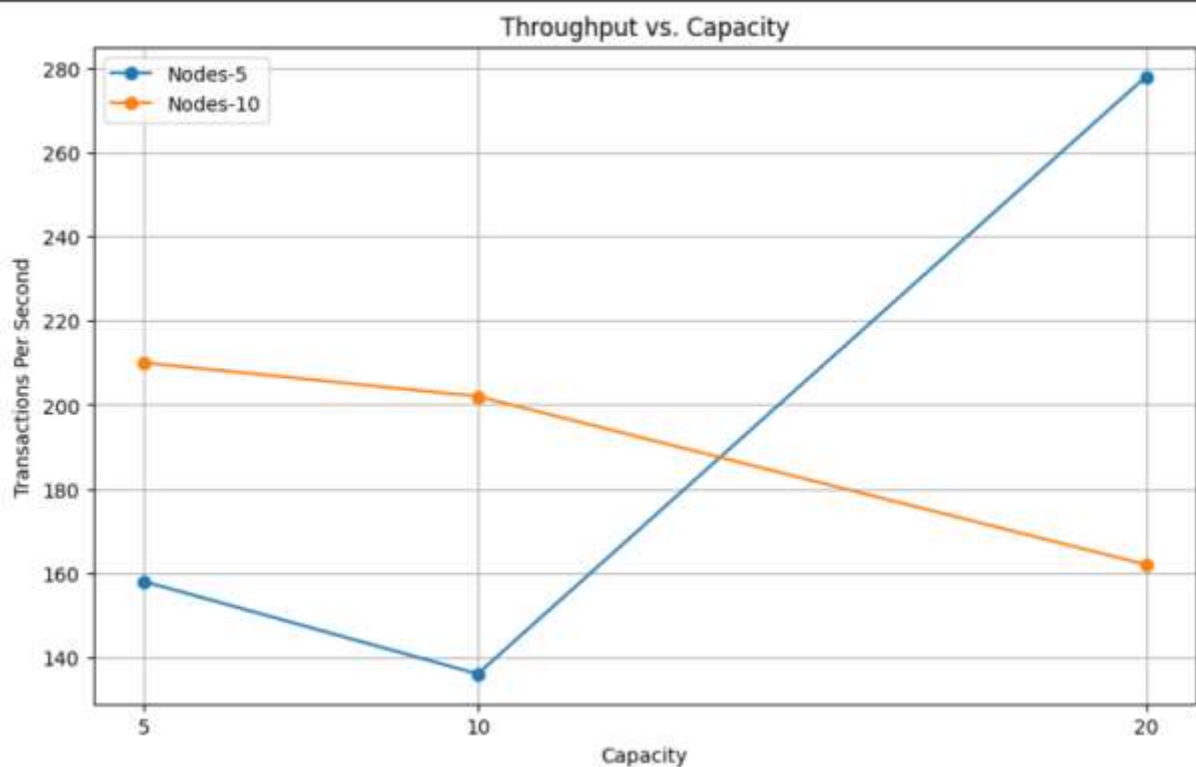
Από τις πιο πάνω μετρήσεις (transaction.txt) προκύπτει ότι

Workers 5

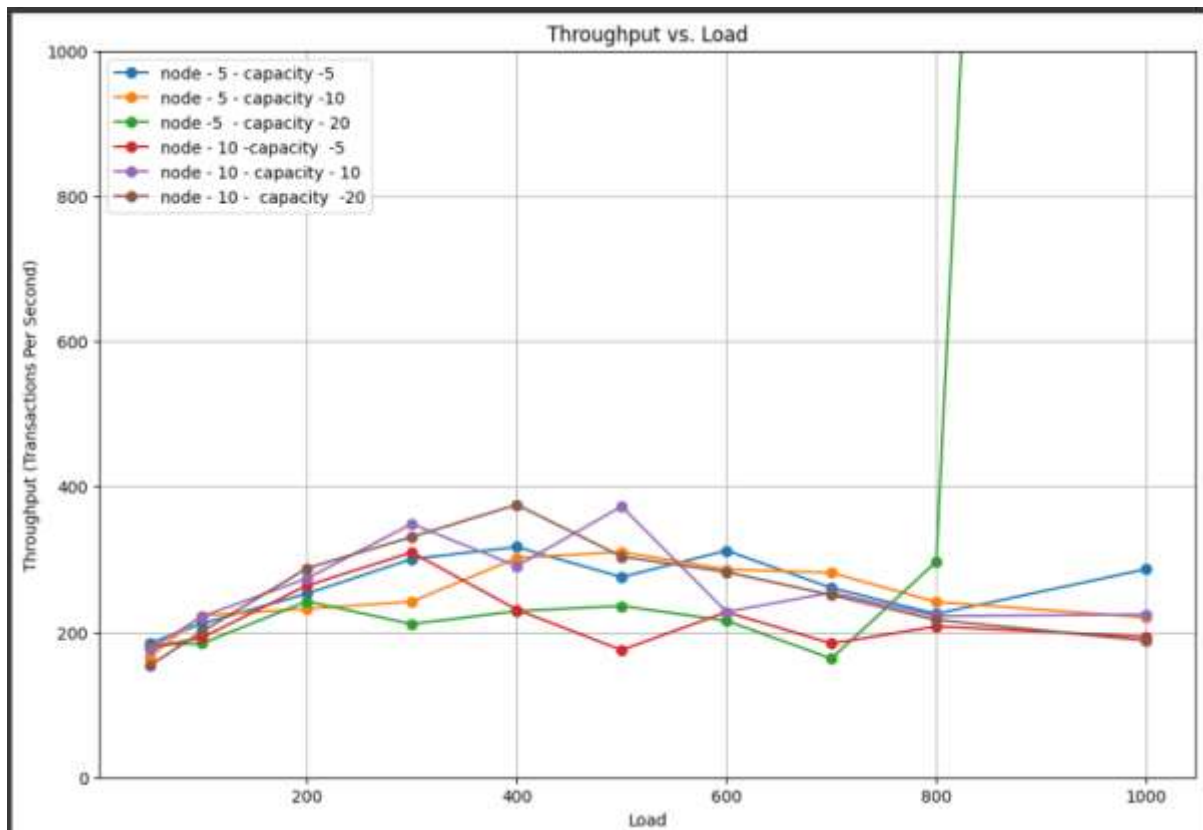
Capacity	5	10	20
Transactions Per Second(240 average time)	158	136	278

Workers 10

Capacity	5	10	20
Transactions Per Second(1000 average time)	210	202	162



Από το πιο πάνω διάγραμμα φαίνεται πως το throughput δεν επηρεάζεται από το capacity, αλλά πιθανότατα από την ταχύτητα δικτιού και όγκο αιτημάτων. Αυτό αναμενόταν αφού το capacity επηρεάζει το stake algorithm το οποίο επηρεάζει την async load όπου αποθηκεύει τις συναλλαγές και όχι τη έκδοση συναλλαγών. Για αυτό το λόγο θα κατασκευάσουμε ένα διάγραμμα throughput - load



Σημείωση: Το load είναι ανά κόμβο δηλαδή σε node-10 το 400(10 * 400 4000 συναλλαγές) όπου αντιστοιχεί σε node-5 800 .

Από την πιο πανό γραφική φαίνεται αύξηση των κόμβων αυξάνουμε το throughput του συστήματος . Ακόμη φαίνεται ότι το σύστημα για 10 κόμβους φαίνεται να λειτουργεί πολύ καλά στο διάστημα $200 < \text{transactions} < 500$. Έπειτα φαίνεται να πέφτει η απόδοση του συστήματος . Οι λόγοι που συμβαίνει αυτό είναι ότι 1) πολλές συνδέσεις με RabbitMQ με αποτέλεσμα καθυστέρηση δημιουργία σύνδεσης 2) ο ένας κόμβος απαιτεί περίπου το 56% τις cpu του VM σε φόρτο εργασίας με αποτέλεσμα με αποτέλεσμα σε 10 κόμβους να κάνει usage 98% δημιουργώντας έτσι καθυστερήσεις λόγω έλλειψης πόρων . Σε κάποιες περιπτώσεις το σύστημα απαντούσε πολύ γρηγορά π.χ. node-5-capacity-20 load 1000 ,αυτό συνέβη για το λόγο ότι στο σύστημα εκδοθήκαν πολλές συναλλαγές και από προηγούμενα load οι οποίες δεν καταχωρήθηκαν ακόμη στη αλυσίδα , κατά συνέπεια ο transaction manager απόρριπτε τις συναλλαγές λόγω μη επαρκούς υπολοίπου .

Τέλος παρατηρούμαι από τα pie charts ότι σε κάθε σερ μετρήσεων ο καταμερισμός πιθανότητας δουλεύει ορθά αφού ο κάθε κόμβος έχει stake 10 bcc σημαίνει ότι η πιθανότητα για 5 κόμβους θα είναι 20 % και για 10 % κόμβους . Από τις μετρήσεις φαίνεται ότι τα προγραμματικά ποσοστά τείνουν προς τα επιθυμητά καθώς αυξάνεται ο αριθμός των block π.χ. στα 248 blocks ο κόμβος 0 έχει 26.5% , +6.5% και σε μια μέτρηση 864 ο κόμβος 23.7% , 3.7% . Γενικά παρατηρούμαι ότι με την αύξηση blocks και nodes έχουμε

καλύτερο καταμερισμό . Στην περίπτωση όπου αλλάξαμε το stake coins σε 100 τότε οι πιθανότητα του κόμβο 0 θα πρέπει να είναι 71.4% και για τους υπολοίπους 7.1% όπου μετρήθηκε 71.5% και για τους υπολοίπους από 6.1% - 8.7%.