ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

# ΚΑΤΑΝΕΜΗΜΕΝΑ ΣΥΣΤΗΜΑΤΑ: BLOCKCHAT

Αναφορά Εξαμηνιαίας Εργασίας

GitHub Repository: Click here

## Ομάδα: 13

| | |
|---|---|
| Γεώργιος Μυστριώτης | 03119065 |
| Φίλιππος Σεβαστάκης | 03119183 |
| Γεώργιος Καούκης | 03119006 |

Απρίλιος 2024

# Εισαγωγή

Σκοπός της παρούσας εργασίας είναι η υλοποίηση μίας πλατφόρμας ανταλλαγής μηνυμάτων και καταγραφής δοσοληψιών, η οποία στηρίζεται σε ένα blockchain που χρησιμοποιεί για το consensus τον αλγόριθμο proof-of-stake, ο οποίος επιλέγει τον validator ενός μπλοκ, με πιθανότητα ανάλογη με το ποσό των νομισμάτων που κατέχει ο κάθε κόμβος. Για την επικοινωνία των κόμβων μέσα στο δίκτυο, χρησιμοποιούνται TCP sockets, που προσφέρουν έναν αξιόπιστο και ασφαλή τρόπο επικοινωνίας. Τέλος, για την εφαρμογή κρυπτογραφικών αλγορίθμων για την ασφάλεια των επικοινωνιών και των δεδομένων, επιλέχθηκε η βιβλιοθήκη PyCryptodome, η οποία προσφέρει μια ευέλικτη και αξιόπιστη υλοποίηση κρυπτογραφικών λειτουργιών.

# Σχεδιασμός Συστήματος

Η υλοποίηση του συστήματος έχει την εξής δομή:

```
src
|-- block.py
|-- blockchain.py
|-- commands.py
|-- config.py
|-- message.py
|-- node.py
|-- p2p.py
|-- proof_of_stake.py
|-- running_script.py
|-- transaction.py
|-- transaction_pool.py
|-- utils.py
|-- wallet.py
```

Με το αρχείο running_script.py εκκινούμε την εφαρμογή. Τα παρακάτω αρχεία αποτελούν τις βασικές κλάσεις για την υλοποίηση του blockchain:

- block.py
- blockchain.py
- transaction.py
- transaction_pool.py
- wallet.py

Για την επικοινωνία των κόμβων χρησιμοποιούνται τα αρχεία:

- message.py
- node.py
- p2p.py

Μέσα στο αρχείο node.py υλοποιείται επίσης ένα απλό CLI. Τέλος, τα υπόλοιπα αρχεία περιέχουν βοηθητικές συναρτήσεις για τη λειτουργία της εφαρμογής.

## Main Blockchain Structures

**block.py:**

Στο συγκεκριμένο αρχείο υλοποιείται το block, η βασική δομική μονάδα του blockchain. Η συγκεκριμένη κλάση, περιλαμβάνει τα πεδία που ζητώνται από την εκφώνηση, δηλαδή το index του block, τη χρονική στιγμή που δημιουργήθηκε, τα transactions του, τον validator του, το hash του προηγούμενου block και το hash του. Για την εύρεση του hash, χρησιμοποιήθηκε ο αλγόριθμος SHA256, με όρισμα ένα λεξικό που περιέχει τα υπόλοιπα δεδομένα. Το block περιέχει ακόμα συναρτήσεις που εξυπηρετούν τη δημιουργία του genesis block, τον έλεγχο πληρότητας, τον υπολογισμό των χρεώσεαν από τα transactions του και την προβολή του σε αναγνώσιμη μορφή.

**blockchain.py**

Η κλάση blockchain δρα ως container για τα blocks, καθώς είναι μία λιστα στην οποία προστίθενται. Περιλαμβάνει βοηθητικές συναρτήσεις για τη δημιουργία του πρώτου transaction και την ευκολότερη διαχείριση των blocks στο εσωτερικό του.

**transaction.py**

Στο αρχείο transaction.py περιλαμβάνεται η αντίστοιχη κλάση, η οποία αντιπρωσοπεύει τις δοσοληψίες και τα μηνύματα που δημιουργούν οι χρήστες. Το unique id κάθε transaction δημιουργείται με hashing, όπως στο block. Για τα αρχικά transactions από τον bootstrap στους κόμβους, καθώς και για τα stakes, τα οποία προσμετρώνται ως transactions, η χρέωση, δηλαδή τα χρήματα που λαμβάνονται από τον validator που θα προστεθεί το συγκεκριμένο transaction, είναι 0.

**transaction_pool.py**

Η συγκεκριμένη κλάση είναι βοηθητική. Χρησιμοποιείται για την αποθήκευση των transactions που δημιουργούνται και τη δημιουργία ενός soft state, μέχρι να εκτελεστούν με την προσθήκη τους σε ένα block. Όταν ένα block μπορεί να δημιουργηθεί, αφαιρούνται από το transaction pool τα αντίστοιχα transactions. Η κλάση αυτή εξασφαλίζει ότι δε θα έχουμε απώλεια κάποιου transaction, καθώς και ότι δε θα εισέλθει σε block κάποιο invalid ή διπλότυπο transaction.

**wallet.py:**

Στο αρχείο wallet.py υπάρχει η αντίστοιχη κλάση, η οποία περά από τη βασική λειτουργία δημιουργίας και αποθήκευσης κλειδιών, εμπεριέχει την πλειονότητα των συναρτήσεων επεξεργασίας transactions και blocks που καθιστούν την εφαρμογή μας λειτουργική. Η κλάση διατηρεί πληροφορίες για το current blockchain, τους peers, καθώς και valid ή temporary αλλαγές. Επιπλέον, διαθέτει ένα instance του transaction_pool, ώστε να διαχειρίζεται ορθά τα transactions. Μέσω του wallet γίνεται το verification και το validation τόσο των transactions, όσο και των blocks. Όταν στείλουμε ένα transaction, μετά από τους απαραίτητους ελέγχους αυτό γινεται broadcast. Αντίστοιχα, όταν λάβουμε ένα transaction, εξετάζεται για το validity του και σε περίπτωση επιτυχίας αλλάζουν οι temporary τιμές και μπαίνει στο transaction pool περιμένοντας τη δημιουργία ενός block. Όταν το transaction pool

αποκτήσει περισσότερα transactions από το capacity του block, μπορεί πλέον να δημιουργηθεί ένα νέο block, Σε αυτή την περίπτωση καλείται η mint_block η οποία εάν είμαι ο validator επιστρέφει ένα έτοιμο block που δίνεται στην broadcast block για να σταλεί στους υπόλοιπους peers. Αν ένας κόμβος δεν είναι ο validator, τότε αναμένει block ώστε να μην γίνει overfloat το δίκτυο αλλά συνεχίζουμε να βάζουμε transactions στο pool. Αντίστοιχα με τα transactions, αν λάβουμε κάποιο block, αυτό γίνεται handled από την handle_block, στην οποία ελέγχεται το validity και αν είναι το block που περιμένουμε το βάζουμε στο blockchain. Η wallet λειτουργεί με temporary και safe balances, ώστε να έχουμε ορθή εποπτεία της τρέχουσα κατάστασης, αλλά η τελική αλλαγή στα balances να γίνεται με την εισαγωγή transactions σε block. Τέλος, σε περίπτωση πολλών stake transactions από ίδιο κόμβο στο ίδιο block, ως valid stake κρατάμε το τελευταίο transaction.

## Proof-of-Stake

**proof_of_stake.py:**

Στο αρχείο αυτό περιέχεται ο αλγόριθμος proof_of_stake. Ο validator επιλέγεται με πιθανότητα ανάλογη των stakes που έχουν δοθεί. Όλοι οι κόμβοι χρησιμοποιούν τον αλγόριθμο για να βρουν τον επόμενο validator, και η συμφωνία τους εξασφαλίζεται με χρήση κοινού seed (το οποίο επιλέγουμε να εξάγεται από το hash του προηγούμενου block).

## Node Communication

**message.py:**

Το message.py περιέχει μία βοηθητική κλάση για τη δημιουργία των μηνυμάτων που θα αποστέλλονται μεταξύ των sockets και χρησιμοποιείται από τις παρακάτω κλάσεις.

**node.py:**

Η κλάση Node αντιπροσωπεύει κάθε κόμβο του συστήματος. Πιο συγκεκριμένα, αρχικοποιεί τον κόμβο με τη διεύθυνση IP και τη θύρα, δημιουργεί ένα νέο πορτοφόλι (Wallet) για τον κόμβο, δημιουργεί ένα νέο αντικείμενο P2P (peer-to-peer) για τον κόμβο, το οποίο χρησιμοποιείται για την επικοινωνία με άλλους κόμβους στο δίκτυο, αρχικοποιεί το blockchain και εκκινεί την εφαρμογή. Μέσα σε αυτή πραγματοποιείται η αρχική διανομή νομισμάτων στο πορτοφόλι του πρώτου κόμβου (αναγνωρίζεται από το id), δημιουργούνται ουρές για την επεξεργασία εισόδου για κάθε νήμα, εκκινείται ένα ξεχωριστό νήμα για την ανάγνωση εισόδου και την αποστολή της στα αντίστοιχα νήματα για επεξεργασία. Στο εσωτερικό της node υλοποιείται ακόμη ένα υποτυπώδες cli, με βοηθητικές συναρτήσεις από το αρχείο commands.py, που διαβάζει τις εντολές του χρήστη και εκτελεί τις διάφορες λειτουργίες. Τέλος, εξυπηρετεί τη διαδικασία αναμονής και αποστολής των blocks στο δίκτυο και τερματίζει την εφαρμογή, όταν δοθεί κατάλληλη εντολή.

**p2p.py:**

Η κλάση P2P υλοποιεί τη λειτουργικότητα ενός peer-to-peer δικτύου για την επικοινωνία και την ανταλλαγή πληροφοριών μεταξύ των κόμβων του συστήματος. Στον constructor, η κλάση αρχικοποιεί τις μεταβλητές που χρειάζεται για τη λειτουργία της, όπως η διεύθυνση IP, η listening-θύρα, το public_key, και δημιουργεί ένα socket στο οποίο ο κάθε κόμβος θα ακούει για συνδέσεις από άλλους κόμβους. Το socket αυτό είναι που γίνεται bound στην listening-θύρα, η οποία και είναι

εκείνη που πρέπει να γίνει γνψστή στους υπόλοιπους κόμβους. Η κλάση περιλαμβάνει την μέθοδο p2p_network_init η οποία καλείται από τον constructor προς αρχικοποίηση του peer-to-peer δικτύου, τόσο από τον bootstrap κόμβο όσο και από τους υπόλοιπους. Η μέθοδος αυτή με την σειρά της καλεί τις επόμενες διαδικασίες: Αρχικά, μία μέθοδο connect_to_bootstrap_node για τη σύνδεση στον bootstrap κόμβο και αποστολή των πληροφοριών που απαιτούνται για αρχικοποίηση από τους άλλους κόμβους. Ο bootstrap κόμβος παράλληλα αναλαμβάνει με τη μέθοδο bootstrap_mode την αποστολή πληροφοριών όπως το id του κάθε νεοεισαχθέντα κόμβου, αλλά αποθήκευση των πληροφοριών αυτου. Όταν συνδεθεί-επικοινωνήσει μαζί του το θεμιτό πλήθος από nodes αναλαμβάνει την αποστολή σε αυτούς των πληροφοριών που σύλλεξε. Στη συνέχεια καλείτε από όλους η συνάρτηση connect_to_all_peers για την δημιουργία connections από και προς όλους τους κόμβους του δικτύου (για ευκολία, χωρίς ιδιαίτερο επιπλέον κόστος δεν έχουμε διατηρήσει την σύνδεση με τον bootstrap, αλλά ξανα-δημιουργούμε μία εδώ). Στο σημείο αυτό γίνεται σύνδεση των κόμβων και δημιουργούνται τα threads για την ακρόαση εισερχόμενων συνδέσεων. Τέλος, τα threads αυτά καλούν τις μεθόδους handle_connection και message_handler που χειρίζονται την επικοινωνία μεταξύ των κόμβων, διαχειρίζονται την αποστολή και λήψη μηνυμάτων σε μορφή pickle και αποσυμπιέζουν τα δεδομένα για επεξεργασία από το πρόγραμμα, την οποία μάλιστα τα ίδια αναλαμβάνουν με την κλήση των αντίστοιχων συναρτήσεων της κλάσης Wallet, handle_transaction, handle_block και handle_blockchain. Η κλάση P2P είναι σαφώς κρίσιμη για τη λειτουργία του συστήματος.

## Αποτελέσματα των πειραμάτων

Στο σημείο αυτό διεξήγαμε πειράματα προκειμένου να αξιολογήσουμε τη λειτουργία του συστήματος σε πραγματικές συνθήκες. Τα πειράματα αυτά εστίασαν σε τρεις κύριους τομείς: την απόδοση του συστήματος, την κλιμακωσιμότητά του και τη δικαιοσύνη του. Για τον σκοπό αυτόν, μας δόθηκαν δύο sets, από 5 και 10 αρχεία αντίστοιχα (ένα για κάθε κόμβο), τα οποία περιέχουν μηνύματα προς τους υπόλοιπους κόμβους στη μορφή:

< recipient_node_id> <message string>

Έτσι, τροποποιήσαμε την παραπάνω υλοποίηση, ώστε κάθε κόμβος να μπορεί να δέχεται τα μηνύματα από το αρχείο του (από το αντίστοιχο set, αναλόγως του αριθμού των κόμβων), να τα επεξεργαστεί και να τα στείλει ως (message) transactions στους υπόλοιπους κόμβους του δικτύου.

```
def file_parsing(id):
    file_name = "trans" + id[2] + ".txt"
    folder_name = "input_" + str(N)
    file_path = folder_name + "/" + file_name
    queue = Queue()
    with open(file_path, 'r') as file:
        for line in file:
            command = "m " + line.strip()
            queue.put(command)
    return queue
```

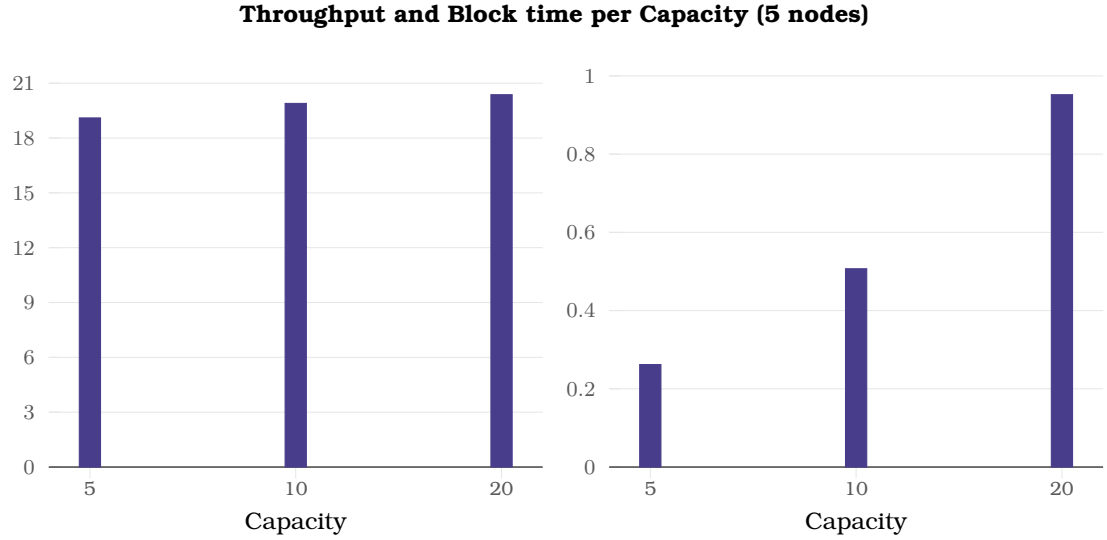
Listing 1: Function that returns a queue of transactions in the form: m <receiver_id> <Message>, from an id-dependent txt file

```
def blockchaining(self, stop_event):
    if self.p2p.id == 'id0':
        self.wallet.initial_distribution()
    else:
        while self.wallet.my_balance()[0] == 0:
            pass
    time.sleep(5)
    self.starting_time = time.time()
    input_queue = file_parsing(self.p2p.id)
    self.command_reading(input_queue, stop_event)
    print("File read!")
```

Listing 2: Function call, before command_reading

## Απόδοση:

Αρχικά, στήσαμε ένα BlockChat με 5 κόμβους και με διαφορετικά block capacities (5, 10 και 20), όπου κάθε κόμβος είχε σταθερό staking 10 BCC. Παίρνοντας τις κατάλληλες μετρήσεις κατά την εκτέλεση, υπολογίσαμε για κάθε διαφορετικό block capacity το throughput, δηλαδή πόσα transactions εξυπηρετούνται στην μονάδα του χρόνου, καθώς και το block time, δηλαδή τον μέσο χρόνο που απαιτείται για να προστεθεί ένα νέο block στο blockchain. Παρακάτω, παρατίθενται τα αντίστοιχα διαγράμματα.



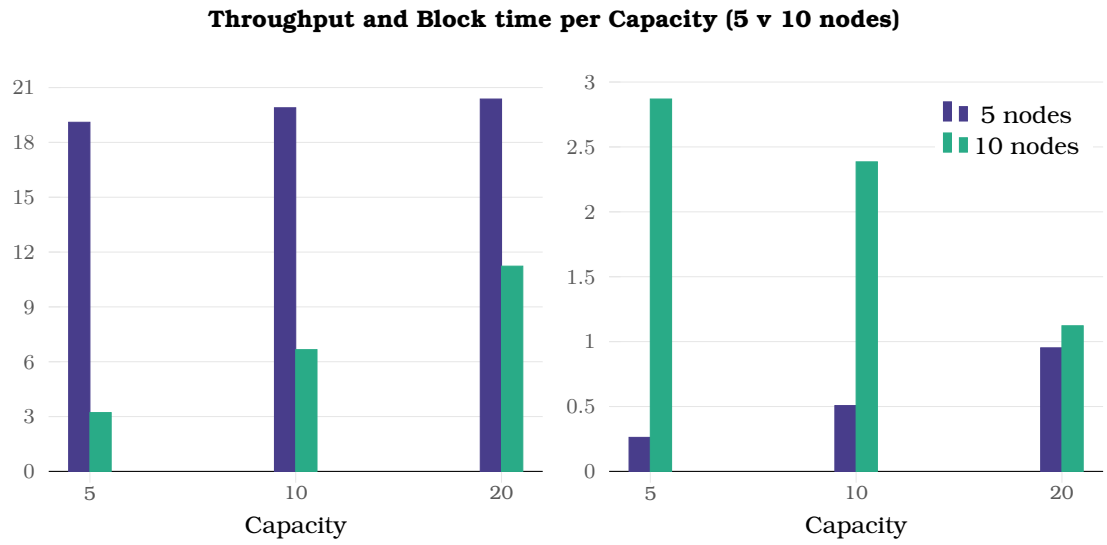

(a) Transactions per second (Throughput)

(b) Mean time (sec) for a Block to be added to the Chain

Όπως διακρίνεται, αρχικά, στα παραπάνω διαγράμματα, το throughput του συστήματός μας είναι αυξάνεται σε κάποιο βαθμό με την αύξηση του block capacity. Η δημιουργία/λήψη των transactions με αυτή των blocks είναι μεν παραλληλοποιημένες, ωστόσο όχι η αποστολή των blocks, οπότε όσο μεγαλύτερο το block capacity τόσο αποφεύγεται το "κόστος" αυτό. Όσον αφορά το block time διάγραμμα, παρατηρούμε ότι ο μέσος χρόνος εξυπηρέτησης ενος block αυξάνει (αναλογικά) με την αύξηση του capacity. Αυτό είναι επίσης αναμενόμενο, δεδομένου ότι πλέον κάθε block χρειάζεται περισσότερα transactions για να "γεμίσει" και κατ' επέκταση εξυπηρετηθεί. Τέλος, αναφορικά με τους χρόνους εκτέλεσης να επισημάνουμε ότι αυτοί παραμένουν σταθεροί για κάθε block capacity (και μάλιστα ικανοποιητικοί - περίπου 26 δευτερόλεπτα για κάθε node για την εκτέλεση των transactions των text files), λογικό αφού έχουμε σταθερό throughput (ή αλλιώς, ενώ αυξάνει το block time με το capacity, έχουμε κατ' αναλογία λιγότερα blocks που θα εξυπηρετηθούν).

### Κλιμακωσιμότητα:

Επαναλάβαμε το πείραμα με 10 clients λαμβάνοντας τις αντίστοιχες μετρήσεις και, παρακάτω, παρουσιάζουμε τις μετρικές του throughput και του block time συναρτήσει του block capacity, μαζί με αυτές του προηγούμενου πειράματος.




(c) Throughput (Transactions per second)

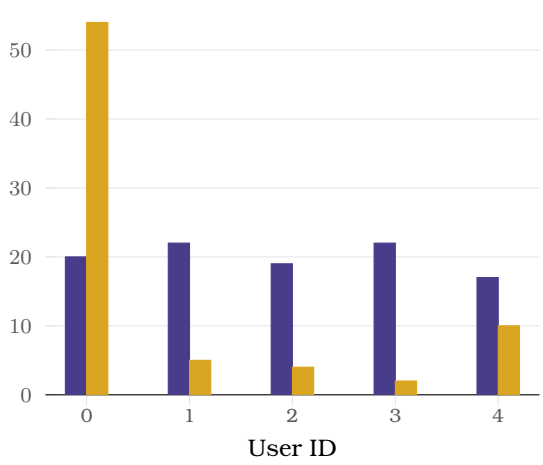
(d) Block Time (Mean time, in sec, per block)

Από τα διαγράμματα αυτά διαπιστώνουμε τα εξής. Όπως και πριν, η αποστολή των blocks καθυστερεί το σύστημα, κάτι που με την αύξηση του block capacity συμβαίνει λιγότερες φορές (λιγότερα blocks). Ωστόσο, συγκριτικά με τα αποτελέσματα του προηγούμενου συστήματος, με την αύξηση των κόμβων το σύστημα έχει περισσότερα listening threads (N-1), τα οποία δρουν ανταγωνιστικά για την απόκτηση των locks. Έτσι, το throughput του συστήματος για την εκτέλεση με 10 κόμβους και 5 block capacity είναι αρκετά χαμηλό, αλλά με σημαντική αύξηση, καθώς μεγαλώνουμε το block capacity. Αντίστοιχα, ενώ έχουμε αρκετά υψηλότερο block time για block capacity 5 (συγκριτικά με τις μετρήσεις του συστήματος με 5 κόμβους) με την αύξηση του capacity διαπιστώνουμε ελλάττωση αυτού. Έτσι, προκύπτει ελλάττωση και στον συνολικό χρόνο εκτέλεσης, από τα 300 δευτερόλεπτα για 5 block capacity, σε 150 και 50 δευτερόλεπτα για 10 και 20 block capacities, αντίστοιχα.

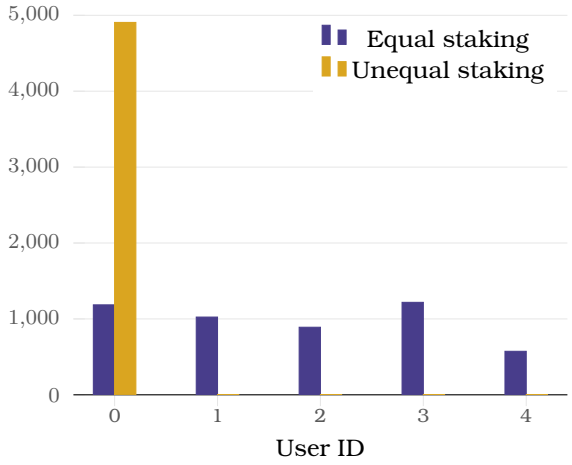
### Δικαιοσύνη:

Τέλος, επαναλάβαμε το πείραμα με 5 κόμβους και 5 block capacity, αυτή τη φορά, ωστόσο, με έναν κόμβο (στην προκειμένη αυτόν με id: id0) να έχει staking 100 BCC. Εξέτασαμε και παραθέτουμε παρακάτω το πλήθος των φορών που ο κάθε κόμβος έγινε validator, καθώς και τα χρήματα που είχε με το πέρας του πειράματος, συγκρίνοντας τα με τις τιμές που προκύπτουν για ίσο staking των 10 BCC.

**Validator times and End-balance for equal (10x5) vs unequal (100-10-10-10-10) staking**




(e) Times each user became validator




(f) End-balance

Όπως ήταν αναμενόμενο, δεδομένης της φύσης του Proof of Stake, ενώ στην περίπτωση των ίσων stakes οι κόμβοι είχαν παραπλήσιο πλήθος φορών που έγιναν validators και κατά συνέπεια "τερμάτισαν" με παραπλήσιο αριθμό BCCs. Αντιθέτως, στην περίπτωση του άνισου staking ο χρήστης που είχε τα περισσότερα, 10πλάσια, stakes έγινε κατ' αναλογία πιο πολλές φορές validator, κάτι που οδήγησε στο να "τερματίσει" με όλα τα BCCs (λόγω των fees). Μάλιστα, επειδή τα BCCs των μηνυμάτα (fees) που έχει να στείλει κάθε κόμβος είναι περισσότερα από αυτά του αρχικού distribution (1000), οι κόμβοι των 10 stakes ξεμένουν χωρίς να τα έχουν στείλει όλα και το σύστημα καταγράφει μόλις 75 blocks (75 validations όπως διακρίνεται και από το πρώτο γράφημα). Είναι, συνεπώς, σαφές ότι για άνισο staking (κατ' επέκταση η λογική του Proof of Stake) καθιστά το σύστημά μας "άδικο" (αλλά και δυσλειτουργικό ως chat).