Εξαμηνιαίο Project στο μάθημα Κατανεμημένων Συστημάτων

Φοιτητές : Παναγιώτης Ζευγολατάκος 03117804
Δημήτρης Κουνούδης 03117169
Γεώργιος Λαγός 03117034

Εισαγωγή :

Στην εξαμηνιαία αυτή εργασία καλούμαστε να υλοποιήσουμε ένα σύστημα blockchain για το κρυπτονόμισμα Noob Cash Coin (NBC) το οποίο θα χρησιμοποιεί μηχανισμό consensus το proof of work.

Αρχικά, θα δώσουμε ένα overview του συστήματος:

Σε γενικές γραμμές, είναι όπως περιγράφεται στην εκφώνηση: υπάρχουν N κόμβοι, οι οποίοι περιέχουν ένα wallet με τις υπογραφές τους, το δικό τους version του blockchain, τα UTXOs για όλους τους κόμβους και μια ουρά η οποία περιέχει transactions που πρόκειται να πραγματοποιηθούν, δηλαδή να προστεθούν στο blockchain.

Θεωρήσαμε σωστό να μην ορίσουμε κάποια μεταβλητή NBC σε κάθε κόμβο η οποία να αντιπροσωπεύει τον αριθμό των coins ενός κόμβου, αλλά να βρίσκουμε το balance αθροίζοντας τα UTXOs, εφόσον έτσι κι αλλιώς η μεταβλητή NBC θα γίνεται update με βάση αυτά κάθε φορά που προστίθεται ένα block στην αλυσίδα. Αυτό, βέβαια, έχει ως αποτέλεσμα να μην θεωρείται ‘ασφαλές’ το balance **έως ότου να μη γίνεται** mine κάποιο block, το οποίο ισχύει και στον πραγματικό κόσμο· όσο πιο πολλά blocks γίνουν mine από τότε που δημιουργήθηκε ένα transaction, τόσο πιο ασφαλές θεωρείται.

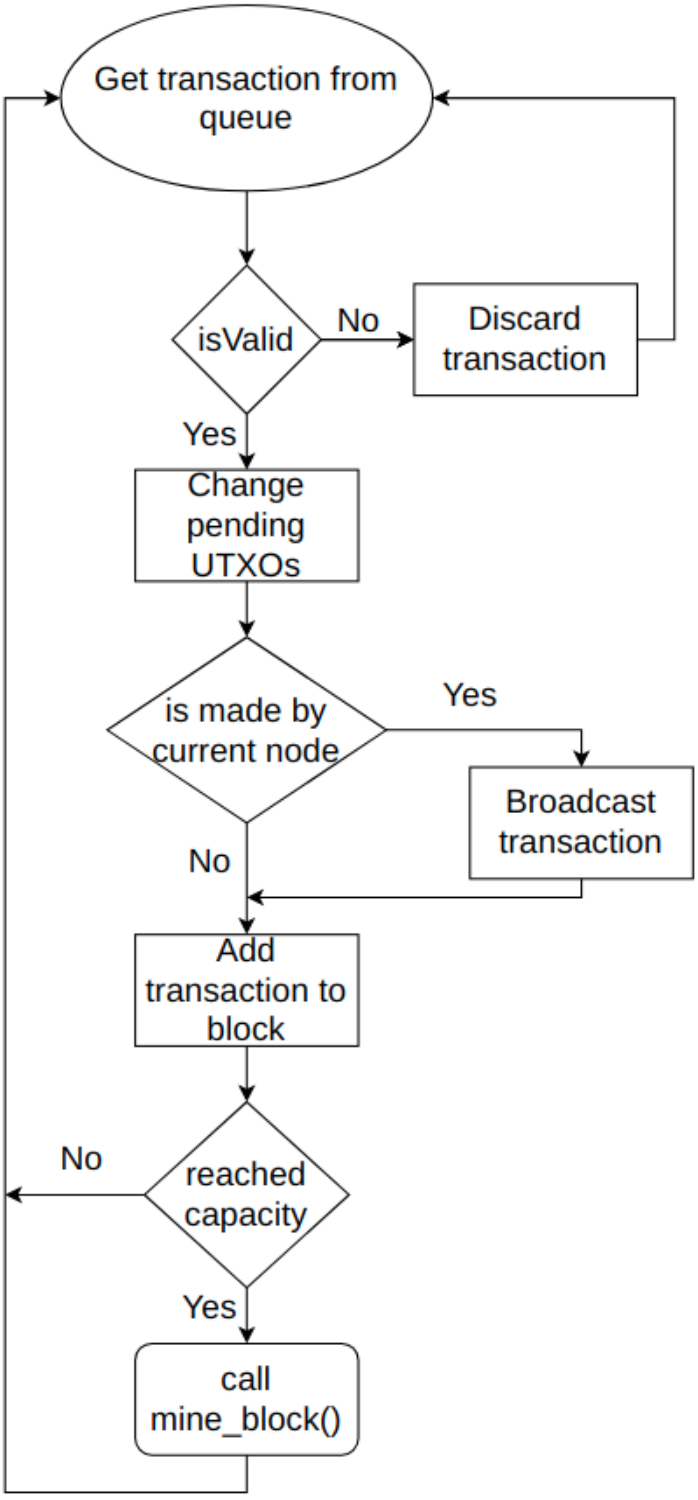
Για να δημιουργηθεί ένα transaction, ένας κόμβος ελέγχει αν έχει τα απαραίτητα UTXOs και εφόσον είναι αρκετά τα χρησιμοποιεί ως inputs, δημιουργεί τα αντίστοιχα outputs για τον εαυτό του και τον παραλήπτη, το υπογράφει με το private key του και το κάνει broadcast. Στη συνέχεια οι υπόλοιποι κόμβοι του συστήματος επαληθεύουν το transaction χρησιμοποιώντας την υπογραφή του και τα UTXOs που χρησιμοποιούνται και εφόσον είναι έγκυρο επιχειρούν να το προσθέσουν στην αλυσίδα.
Να σημειωθεί πως στο template που μας δόθηκε, η συνάρτηση validate_transaction δεν καλεί μόνο τη συνάρτηση verify_signature, αλλά βλέπει και αν ο κόμβος που έκανε το transaction έχει αρκετά NBC coins προκειμένου να το κάνει. Αυτό υλοποιήθηκε έμμεσα, λόγω της προσπάθειας αφαίρεσης των UTXO που ορίζονται ως inputs.
Για παράδειγμα, έστω πως ένας κόμβος έχει ένα UTXO με 100 NBC coins και θέλει πρώτα να στείλει 10 coins και μετά 5 coins. Το πρώτο transaction για 10 coins θα έχει ως input το UTXO με 100 coins, ενώ το δεύτερο transaction για 5 coins θα έχει ως input το UTXO με 90 coins, το οποίο δημιουργήθηκε από το πρώτο transaction ως output. Εάν για κάποιο λόγο ένας κόμβος λάβει το δεύτερο transaction πρώτα, θα προσπαθήσει να αφαιρέσει ένα UTXO με 90 coins, το οποίο δε θα υπάρχει (θα υπάρχει μόνο το αρχικό UTXO με 100 coins). Έτσι, η προσπάθεια αφαίρεσής του θα αποτύχει και το transaction θα θεωρηθεί invalid. Εφόσον ένα transaction

ορίζει τόσο τα inputs όσο και τα outputs, αυτό σημαίνει πως κάθε κόμβος πρέπει να έχει συγκεκριμένα UTXOs ενός κόμβου προκειμένου να τα χρησιμοποιήσει ως inputs.
Αυτό έχει ως αποτέλεσμα να γίνονται δεκτά τα transactions με τη σειρά που τα δημιουργεί ο κόμβος που τα έστειλε, ο οποίος είναι και αυτός που ελέγχει εάν έχει αρκετό balance ώστε να εκτελέσει ένα transaction και μόνο τότε δημιουργεί τα σωστά inputs και outputs για ένα transaction. Ακόμα και στη χειρότερη περίπτωση που όλοι οι κόμβοι δεχτούν το δεύτερο transaction πρώτα και το απορρίψουν, δε θα εξαφανιστεί, διότι το έχει κάνει validate ο κόμβος που το έστειλε, οπότε κάποια στιγμή θα μπει στην αλυσίδα.

Επίσης έχει χρησιμοποιηθεί lock για τη διαδικασία επεξεργασίας transaction/mining, την επίλυση conflict και το simulation (αν τρέχει), έτσι ώστε να έχει μόνο ένα από αυτά access στα στοιχεία ενός κόμβου. Ο λόγος που έχει χρησιμοποιηθεί στο simulation (σχεδιαστική απόφαση), είναι έτσι ώστε να προστίθενται τα transactions ενός κόμβου ομοιόμορφα. Αν δε γινόταν αυτό, η ουρά με τα transactions που πρέπει να επαληθευτούν που υπάρχει σε κάθε κόμβο θα γέμιζε πρώτα με τα transactions του κόμβου αυτού και μετά με τα εισερχόμενα transactions, το οποίο φυσικά θα λυνόταν όσο έτρεχαν οι κόμβοι, αλλά υπάρχει η πιθανότητα ένας κόμβος να κάνει τόσες συναλλαγές ώστε να τελειώσει το balance του και να μην εκτελέσει πολλές από τις επόμενες, ενώ αν έτρεχε ομοιόμορφα να λάμβανε NBC από άλλον κόμβο.

Ένα ‘πρόβλημα’ το οποίο ίσως θα ήταν προτιμότερο να υλοποιήσουμε με διαφορετικό τρόπο, είναι η αντιμετώπιση των pending transactions που βρίσκονται σε έναν κόμβο όταν αυτός κάνει mine. Όταν ένας κόμβος κάνει mine, αφαιρεί από τα pending transactions του όλα τα transactions από άλλους κόμβους, και αντίστοιχα οι άλλοι κόμβοι τα αναδημιουργούν. Αυτό είναι απόρροια πολλών ωρών debugging, όπου συνειδητοποιήσαμε πως υπήρχε πρόβλημα με την εγκυρότητα των transactions που είχαν γίνει ενώ είχε ληφθεί mine κάποιο block (αν οι αρχικοί κόμβοι τα ξαναβάλουν στα pending επειδή δεν πραγματοποιήθηκαν και το ξανακάνουν process δημιουργείται πρόβλημα με τα UTXOs και δε θεωρούνται έγκυρα από τον ίδιο τον κόμβο). Για να μην υπάρχει double spending ή να χάνονται transactions, όταν κάποιος κόμβος κάνει mine, αφαιρεί όλα τα pending transactions που δεν έχει δημιουργήσει ο ίδιος. Αντίστοιχα, οι κόμβοι που τα είχαν κάνει broadcast (για να καταλήξουν στον κόμβο που έκανε mine) τα έχουν βάλει και στο δικό τους block το οποίο προσπαθούσαν να κάνουν mine. Σε αυτήν την περίπτωση, ελέγχουν αν τα transactions που έχουν βάλει στο block τους υπάρχουν στο εισερχόμενο block, και αν δεν υπάρχουν πάει να πει πως έχουν γίνει discard από τον κόμβο που έκανε το mine και τα δημιουργούν εκ νέου. Ίσως και να είναι λειτουργικό χωρίς τα παραπάνω, αλλά το working version του κώδικα αυτό είναι και σε σχέση με το χρόνο που χρειάζεται για να γίνει mine ένα block η διαφορά είναι αμελητέα (παρόλο που μπορεί να έχει μια σχετικά μικρή επίπτωση στη ρυθμαπόδοση).

Όσον αφορά την υπογραφή και επιβεβαίωση των transactions, έγινε χρήση της βιβλιοθήκης cryptography (https://cryptography.io/en/latest/hazmat/primitives/asymmetric/rsa/)
αντί της PyCrypto (https://pycryptodome.readthedocs.io/en/latest/src/hash/hash.html),
εφόσον είδαμε πως χρειάζεται περισσότερη ώρα για να γίνει verify μια υπογραφή με τη δεύτερη (https://github.com/nlitsme/pyCryptoBenchmarking).
Στις επόμενες σελίδες βρίσκονται high-level διαγράμματα και εξηγήσεις για τη λειτουργία του συστήματός μας:



Κάθε node έχει μια συνάρτηση worker η οποία δουλεύει σύμφωνα με το παρόν διάγραμμα:

Αρχικά παίρνει ένα transaction από το queue (FIFO), και στη συνέχεια ελέγχει αν είναι έγκυρο, ελέγχοντας την υπογραφή του και ένας κόμβος έχει αρκετό balance έτσι ώστε πραγματοποιήσει το transaction αυτό (στην πραγματικότητα ελέγχει και αλλάζει τα UTXOs*, όπως εξηγήθηκε παραπάνω). Αν δεν είναι έγκυρο προχωράει στο επόμενο transaction.

Αν είναι έγκυρο, ελέγχει αν δημιουργήθηκε από τον κόμβο στον οποίο βρίσκεται, στην οποία περίπτωση το κάνει και broadcast.

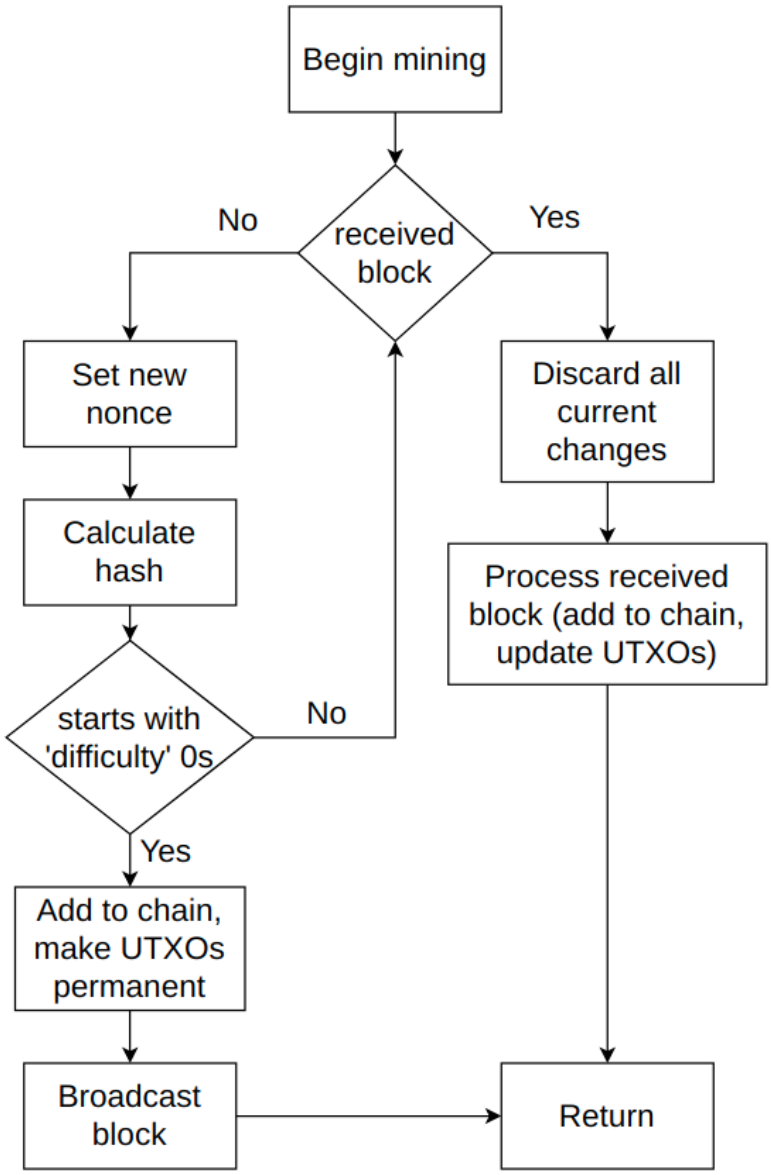
Στη συνέχεια το προσθέτει στο block το οποίο προσπαθεί να δημιουργήσει και όταν το block περιέχει *capacity* transactions, καλείται η συνάρτηση για να ξεκινήσει η διαδικασία mining, η οποία εξηγείται στο επόμενο διάγραμμα.

*μάλιστα αλλάζει τα pending_UTXOs, μια μεταβλητή που έχουμε ορίσει έτσι ώστε να μην πειράζουμε τα πραγματικά UTXOs, παρά μόνο αν προστεθεί ένα καινούριο block στο σύστημα.

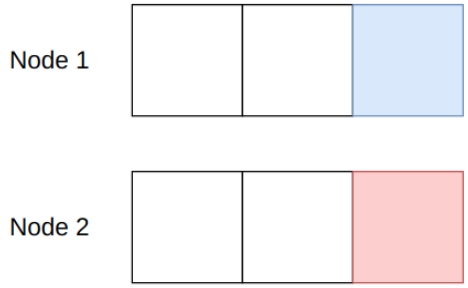
Παρακάτω εξηγείται η διαδικασία mining και συγκεκριμένα η υλοποίηση της συνάρτησης mine_block:

Η διαδικασία mining που ακολουθείται έχει ως εξής:

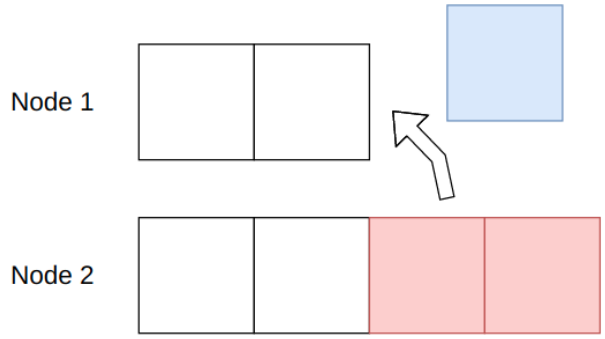
Αρχικά ελέγχεται εάν έχει ολοκληρώσει τη διαδικασία mining ένας άλλος κόμβος και έχει κάνει broadcast το block που βρήκε.
Αν έχει βρεθεί, τότε αφαιρούνται όλες οι προσωρινές αλλαγές που έχουν γίνει (current_block, pending_UTXOs), προστίθεται το block στην αλυσίδα και πραγματοποιούνται όλα τα transactions που βρίσκονται μέσα σε αυτό.
Εάν δεν έχει βρεθεί τότε υπολογίζεται ένα καινούριο nonce (+1 από το προηγούμενο με μηδενισμό όταν φτάσει το 2^64), με αυτό το nonce υπολογίζεται το καινούριο hash του block που προσπαθεί να γίνει mine και αν ξεκινάει με *difficulty* μηδενικά, τότε είναι έγκυρο. Σε αυτήν την περίπτωση προστίθεται στο chain, οι αλλαγές στα UTXOs γίνονται μόνιμες (UTXOs = pending_UTXOs) και το καινούριο block γίνεται broadcast.



Όσον αφορά την περίπτωση του ταυτόχρονου mine από 2 ή περισσότερους κόμβους, οι κόμβοι προσθέτουν στην αλυσίδα τους το block που θα λάβουν πρώτο, και έτσι μπορεί να καταλήξουν με δύο ή περισσότερα διαφορετικά version της αλυσίδας, όπου τα τελευταία τους block είναι διαφορετικά:



Προκειμένου να καταλήξουν όλοι με την ίδια αλυσίδα, τρέχουν τον αλγόριθμο consensus όταν κάποιος κόμβος λάβει ένα block το οποίο δε συμφωνεί με το δικό του version της αλυσίδας (λάθος previous_hash), σύμφωνα με τον οποίο υιοθετείται η αλυσίδα με το μεγαλύτερο μήκος:



Για να το πετύχουμε αυτό, αρχικά κάνουμε undo τα transactions που υπάρχουν στα blocks που πρέπει να αφαιρεθούν, δημιουργώντας τα UTXOs που υπάρχουν ως inputs τους και αφαιρώντας τα outputs τους. Στη συνέχεια, εκτελούμε τα transactions που υπάρχουν στα καινούρια blocks της σωστής αλυσίδας και καταλήγουμε με τα σωστά UTXOs.
Έπειτα, βάζουμε πίσω στο queue τα transactions τα οποία υπήρχαν στα λανθασμένα blocks, αλλά ταυτόχρονα δεν υπάρχουν στα καινούρια blocks, δηλαδή transactions τα οποία (πλέον) δεν έχουν επαληθευτεί.
Τέλος, αφαιρούμε τα τελευταία λανθασμένα block που υπάρχουν στην αλυσίδα με το μικρότερο μήκος και προσθέτουμε τα καινούρια. Έτσι επιλύουμε τυχόν διακλαδώσεις στο blockchain.

Ακολουθεί μια αναλυτική περιγραφή των αρχείων που έχουμε δημιουργήσει:

Στο αρχείο **block.py** ορίζουμε την κλάση Block η οποία θα χρησιμοποιηθεί για να δημιουργήσουμε τα απαραίτητα block της αλυσίδας μας. Περιέχει τα στοιχεία του μπλοκ που αναγράφονται στην εκφώνηση και συγκεκριμένα :

- Το index του μπλοκ
- Μια λίστα από transaction που έχουν πραγματοποιηθεί
- Το υπολογισμένο nonce του μπλοκ
- Το παρόν hash
- Το προηγούμενο hash

Εντός της ίδιας κλάσης ορίζουμε και τις συναρτήσεις **add_transaction** για να προσθέτουμε νέα συναλλαγή στην ήδη υπάρχουσα λίστα και **myHash** η οποία δημιουργεί ένα dictionary με τις πληροφορίες του μπλοκ και χρησιμοποιώντας την hash function SHA-256 καλώντας την αντίστοιχη συνάρτηση της python κωδικοποιεί τις πληροφορίες αυτές.

Στο αρχείο **blockchain.py** ορίζουμε την κλάση Blockchain η οποία θα χρησιμοποιηθεί για να δημιουργήσουμε το blockchain μας και περιέχει τη λίστα από τα blocks που περιέχονται, καθώς και τη χωρητικότητα τους. Εντός της ίδιας κλάσης ορίζουμε και τις συναρτήσεις **add_block** για να προσθέτουμε νέο block στην ήδη υπάρχουσα λίστα των μπλοκ (κάνοντας append σ'αυτή) και συνεπώς στην «αλυσίδα» και **get_transactions** η οποία μας επιστρέφει το σύνολο των συναλλαγών (transaction) που έχουν γίνει σε όλα τα μπλοκ και συνεπώς στο blockchain μας.

Στο αρχείο **node.py** ορίζουμε την κλάση node η οποία θα χρησιμοποιηθεί για να δημιουργήσουμε τους κόμβους της «αλυσίδας» μας. Περιέχει τα στοιχεία του εκάστοτε κόμβου:

- Το Blockchain που έχει ο κόμβος
- Το current_block, το block το οποίο προσπαθεί να φτιάξει για να βάλει στο chain
- Το id του (0 εώς n-1 , όπου n ο αριθμός των κόμβων)
- Το ring που περιέχει τις πληροφορίες κάθε κόμβου
- Τα UTXO του όλων των κόμβων
- Τα pending_UTXOs που είναι τα προσωρινά UTXOs μέχρι να γίνει mine το block
- Το port που του αντιστοιχεί
- Το ip που του αντιστοιχεί
- Το Wallet που του αντιστοιχεί
- Το current_id_count, έναν μετρητή των πόσων κόμβων υπάρχουν
- Το pending_transactions, ένα queue που περιέχει transactions που πρόκειται να επαληθευτούν και να προστεθούν στο current_block

Καθώς και άλλες μεταβλητές οι οποίες βοηθούν με τη λειτουργία του κόμβου (booleans, locks, κλπ…)

Εντός της ίδιας κλάσης ορίζουμε και τις συναρτήσεις :

1. **worker**: επεξεργάζεται τα pending transactions, προσπαθεί να τα κάνει validate και αν είναι valid τότε τα προσθέτει στο block καλώντας την add_transaction_to_block. Επίσης, αν ένα transaction έχει δημιουργηθεί από τον ίδιο κόμβο, καλεί την broadcast_transaction.
2. **get_wallet_balance** : επιστρέφει τον αριθμό των NBC coins που περιέχονται στο wallet συγκεκριμένου κόμβου αθροίζοντας όλα τα UTXOs του. Μάλιστα, προσθέτουμε τα pending_UTXOs, δηλαδή τις προσωρινές αλλαγές που έχουν γίνει. Αυτό βέβαια δεν είναι πλήρως αντιπροσωπευτικό, εφόσον αν υπάρχουν πολλά pending transactions, δεν είναι σίγουρο πως θα γίνουν οι συγκεκριμένες αλλαγές, αλλά το ίδιο 'πρόβλημα' θα υπήρχε αν κοιτούσαμε τα UTXOs που είναι μόνο για transactions που βρίσκονται σε validated block και έχουν προστεθεί στην αλυσίδα.
3. **print_utxos** : για debugging.
4. **view_transactions** : επιστρέφει τη λίστα με τα transaction του μπλοκ που δημιουργείται.

5. **get_transactions_number** : χρήσιμο για να μην πειραχτούν τα πρώτα UTXOs που δίνουν τα αρχικά ποσά σε όλους τους κόμβους.
6. **transaction_exists** : χρήσιμο για να μην μπει στο blockchain μια συνάρτηση που έχει γίνει ήδη (πιθανά από κάποιον άλλον, πχ. μετά από resolve_conflicts). Έτσι αποφεύγεται το double spending.
7. **get_transaction_inputs** : λαμβάνει τη λίστα από τα UTXOs που χρειάζονται για να γίνει ένα transaction. Ελέγχει αρχικά αν το υπόλοιπο του wallet επαρκεί για τη συναλλαγή καλώντας τη συνάρτηση get_wallet_balance και συγκρίνοντας το αποτέλεσμα με το ζητούμενο για τη συναλλαγή ποσό (amount). Επειτα συγκεντρώνει επαρκή αριθμό από unspent transaction output μέσα από τη λίστα UTXOs και επιστρέφει μια πλειάδα από τα input αυτά και το ποσό στο οποίο αθροίζονται . Αν το υπόλοιπο δεν επαρκεί τότε απλά επιστρέφει None και αποτυγχάνει η δημιουργία των transactions.
8. **create_new_block** : δημιουργεί νέο μπλόκ σύμφωνα με το προηγούμενο hash και το index του.
9. **generate_wallet** : Δημιουργεί ένα Wallet object για το συγκεκριμένο node όπως ορίζεται στον constructor της κλάσης wallet όπου παράγεται ένα ζευγάρι private & public key χρησιμοποιώντας τον αλγόριθμο RSA.
10. **register_node_to_ring** : μέσω αυτής της συνάρτησης ο αρχικός κόμβος του blockchain (bootstrap node) προσθέτει ένα καινούριο κόμβο στο ring θέτοντάς του τα id, ip, port, public_key που έλαβε.
11. **initialize_nodes** : Αφού δημιουργηθούν τα nodes και προστεθούν στο ring, μέσω της συνάρτησης αυτής μεταδίδεται το ring σε όλα τα nodes και στη συνέχεια δημιουργεί τα αρχικά transactions για να δοθούν τα αρχικά 100 NBC στους κόμβους.
12. **create_transaction** : Μέσω της συνάρτησης αυτής δημιουργείται ένα transaction από έναν κόμβο, υπογράφεται με την υπογραφή του και προστίθενται τα απαραίτητα UTXO inputs. Στη συνέχεια δημιουργούνται τα 2 outputs για τον αποστολέα και τον παραλήπτη και προστίθενται στα outputs του transaction.
13. **broadcast_transaction** : γνωστοποιείται μια συνάρτηση και στους άλλους κόμβους.
14. **validate_transaction** : Καλείται όταν πρέπει να γίνει process ένα transaction. Αρχικά, κάνει verify την υπογραφή αυτού που το έστειλε. Έπειτα προσπαθεί να αφαιρέσει τα UTXOs που έχουν δοθεί ως inputs και να προσθέσει αυτά που έχουν οριστεί ως outputs. Αν αποτύχει η αφαίρεση των inputs (επειδή δεν υπάρχουν), αυτό σημαίνει πως το transaction δεν είναι valid και η συνάρτηση επιστρέφει, ελέγχοντας έτσι έμμεσα και το balance ενός wallet.
15. **add_transaction_to_block** : Προστίθεται ένα transaction στο current_block, και αν το block έχει φτάσει capacity transactions, τότε καλείται η mine_block.
16. **process_received_block** : Καλείται όταν ληφθεί ένα σωστό block. Γίνονται discard το current_block και τα pending_UTXOs, εφόσον ο κόμβος δεν κατάφερε να κάνει mine. Στη συνέχεια εκτελεί τα transactions του εισερχόμενου block και το βάζει στην αλυσίδα. Επίσης, επαναφέρει τα transactions του κόμβου που βρίσκονταν στο current_block για να ξαναδημιουργηθουν, εφόσον τελικά δε χρησιμοποιήθηκαν και υπάρχει περίπτωση να απορριφθούν από άλλους κόμβους.
17. **mine_block :** Η συνάρτηση αυτή καλείται όταν πρέπει να γίνει mine ένα block. Διαλέγεται τυχαία ένα nonce από το 0 έως το 2^{64}. Έπειτα ανάλογα με τη δυσκολία που

έχουμε ορίσει (config.difficulty) υπαγορεύουμε πόσα μηδενικά θέλουμε να βρούμε στην αρχή του hex string του block που θα υπολογίσουμε. Μόλις καταλήξουμε σε μια σειρά ψηφίων που να αρχίζουν με τον απαιτούμενο αριθμό μηδενικών ('0' * config.difficulty) σταματάμε το mining του συγκεκριμένου block, το προσθέτουμε στο blockchain , δημιουργούμε το επόμενο block με previous_hash το hash του block που μόλις βρέθηκε και καλώντας τη συνάρτηση broadcast_block γνωστοποιούμε το block και στους άλλους κόμβους του δικτύου. Εάν κατά τη διάρκεια του mining ληφθεί ένα (σωστό) block από άλλον κόμβο, σταματάει η διαδικασία mining και καλείται η process_received_block.

18. **broadcast_block :** Γνωστοποιούμε ένα μπλοκ σε ολόκληρο το blockchain.
19. **validate_block** : Ελέγχουμε αν ένα block έχει ως previous_hash το current_hash του τελευταίου block της αλυσίδας και αν ξεκινάει με difficulty μηδενικά.
20. **validate_chain** : καλείται όταν λάβουν οι κόμβοι την αλυσίδα από τον bootstrap και κάθε block της αλυσίδας γίνεται validate, με εξαίρεση το genesis.
21. **add_UTXOs** : καλείται από την process_received_block και εκτελεί τα σωστά transactions που υπάρχουν σε ένα block που λήφθηκε.
22. **reinsert_transactions** : καλείται από την process_received_block και επαναφέρει τα transactions του κόμβου που δεν κατάφεραν να μπουν στο block και να γίνουν mine, αλλά ταυτόχρονα δεν υπάρχουν στο εισερχόμενο block.
23. **recreate_node_transaction** : ουσιαστικά επαναδημιουργεί τα transactions που έχουν δημιουργηθεί από τον ίδιο τον κόμβο. Αυτό χρειάζεται για να οριστούν καινούρια inputs σε περίπτωση προηγούμενης αποτυχίας προσθήκης στο block. Για να αποφευχθεί το double spending, του δίνεται το transaction_id του προηγούμενου transaction.
24. **undo_UTXOs** : καλείται από τη resolve_conflicts για να γίνουν undo τα transactions που υπάρχουν στα τελευταία blocks της αλυσίδας που έχουν επιβεβαιωθεί πως είναι λανθασμένα. Ο λόγος που ονομάστηκε έτσι είναι επειδή το focus είναι στο να έρθουν τα UTXOs σε μια προηγούμενη σωστή μορφή.
25. **resolve_conflicts** : καλείται όταν ένας κόμβος λάβει ένα block το οποίο δεν είναι valid. Ζητάει από όλους τους κόμβους το δικό τους version της αλυσίδας και επιλέγει να υιοθετήσει το μεγαλύτερο. Συγκεκριμένα ψάχνει να βρει σε ποιο σημείο δημιουργήθηκε παρακλάδι της αλυσίδας, αφαιρεί τα δικά του λανθασμένα blocks και προσθέτει τα καινούρια σωστά. Ταυτόχρονα, κάνει undo τα transactions που βρίσκονται στα λανθασμένα block για να φέρει τα UTXOs σε μια σωστή προηγούμενη μορφή και στη συνέχεια εκτελεί τα transactions που βρίσκονται στα καινούρια σωστά blocks. Τέλος, επαναφέρει τα transactions που υπήρχαν στα λανθασμένα blocks ενώ ταυτόχρονα δεν υπάρχουν στα καινούρια σωστά blocks.

Στο αρχείο **transaction.py** ορίζουμε την κλάση Transaction η οποία θα χρησιμοποιηθεί για να δημιουργήσουμε ένα transaction με τις απαραίτητες πληροφορίες. Αποτελείται από μια συνάρτηση αρχικοποίησης η οποία ορίζει τα στοιχεία του εκάστοτε transaction. Εντός της ίδιας κλάσης ορίζουμε και τις συναρτήσεις :

1. **get_hash**: Δημιουργεί ένα json string μέσω της json.dumps το οποίο περιέχει τις πληροφορίες της συναλλαγής (διεύθυνση του αποστολέα και του παραλήπτη, το ποσό που θα σταλεί, τη λίστα των input καθώς και την ώρα που λαμβάνει χώρα) και μέσω του SHA256 επιστρέφει το hash ως κωδικοποιημένο string μέσω της SHA256.

2. **to_dict**: Επιστρέφει ένα dictionary με τα στοιχεία της συναλλαγής

3. **sign_transaction**: υπογράφει τη συναλλαγή ο αποστολέας με το private key του.

4. **verify_signature**: Η συνάρτηση αυτή επιβεβαιώνει την υπογραφή μιας συναλλαγής που έλαβε από άλλον κόμβο.

Για τις παραπάνω συναρτήσεις χρησιμοποιείται η βιβλιοθήκη cryptography: **https://cryptography.io/en/latest/hazmat/primitives/asymmetric/rsa/**

Στο αρχείο **transaction_io.py** ορίζουμε τις κλάσεις Transaction_Input και Transacation_Output, οι οποίες περιέχουν τις πληροφορίες που πρέπει σύμφωνα με την εκφώνηση.

Στο αρχείο **bootstrap.py** δημιουργούμε το genesis block του blockchain και το πρώτο transaction. Χρησιμοποιώντας το Flask framework δημιουργούμε το endpoint:

1. **/node/register (POST) :** Χρησιμοποιείται όταν ένας κόμβος θέλει να προστεθεί στο δίκτυο. Ο bootstrap κόμβος χρησιμοποιεί τις πληροφορίες που έλαβε καλώντας τη register_node_to_ring. 'Επειτα, αν έχουν προστεθεί N κόμβοι στο δίκτυο, καλείται η συνάρτηση initialize_nodes που γνωστοποιεί σε όλους τους κόμβους αρχικές πληροφορίες. Ταυτόχρονα ξεκινάει ο worker που επεξεργάζεται τα transactions και είτε ο client είτε το simulation.

Το αρχείο **config.py** δημιουργήθηκε για να μας διευκολύνει την εκτέλεση καθώς περιλαμβάνει μεταβλητές όπως τον αριθμό των nodes, τη χωρητικότητα του block, τη δυσκολία (αριθμό των αρχικών μηδενικών που πρέπει να υπολογίσουμε) το ip του αρχικού κόμβου και ορισμένες μεταβλητές που μας βοηθάνε στο τρέξιμο των προσομοιώσεων.

Στο αρχείο **wallet.py** ορίζουμε την κλάση Wallet η οποία θα χρησιμοποιηθεί για να δημιουργουμε wallets (objects) με τις απαραίτητες πληροφορίες. Αρχικοποιείται χρησιμοποιώντας το κρυπτογραφικό αλγόριθμο RSA (από τη βιβλιοθήκη cryptography) και παράγει ένα public/private key pair.

Στο αρχείο **rest.py** αρχικοποιούνται όλοι οι κόμβοι πλην του bootstrap και γνωστοποιούν στον bootstrap το ip και το port τους. Επίσης ορίζονται τα παρακάτω endpoints:

1. **/node/initialize (POST)** : όταν όλοι οι κόμβοι έχουν εισαχθεί τότε ο bootstrap κόμβος θα χρησιμοποιήσει αυτό το endpoint για να μεταδώσει το ring στον κάθε κόμβο. Εδώ ελέγχεται αν είναι έγκυρη η αλυσίδα καλώντας την validate_chain και αν είναι, προσθέτει στον node το chain και τα UTXOs που έλαβε. Ταυτόχρονα εκκινεί τον worker για να μπορεί να χειρίζεται εισερχόμενα transactions και είτε τον client είτε το simulation.

2. **/begin (POST)** : Χρησιμοποιείται από τον bootstrap κόμβο, αφού έχει μεταδώσει τα αρχικά transactions και έχουν όλοι οι κόμβοι 100 NBC και ουσιαστικά σηματοδοτεί στο simulation/client πως μπορεί να αρχίσει να δημιουργήσει transactions

Στο αρχείο **common_functions.py** περιέχονται οι συναρτήσεις και τα endpoints που είναι κοινά τόσο στον bootstrap κόμβο όσο και στους υπόλοιπους. Αρχικά ορίζονται τα παρακάτω endpoints:

1. **/block/add (POST)** : Χρησιμοποιείται όταν ένας κόμβος λαμβάνει ένα block που έχει γίνει mine από κάποιον άλλον. Το block γίνεται validate, και εφόσον είναι έγκυρο προστίθεται στο chain, αλλιώς καλείται η συνάρτηση resolve_conflicts.

2. **/transaction/receive (POST)** : Χρησιμοποιείται όταν ένας κόμβος λαμβάνει ένα transaction από έναν άλλον κόμβο, απλά το προσθέτει στο queue των pending transactions.

3. **/transactions/get (GET)** : Χρησιμοποιείται όταν θέλουμε να λάβουμε όλα τα transactions που έχουν γίνει (χρησιμοποιήθηκε για testing όταν στήναμε το σύστημα).

4. **/balance (GET)** : Χρησιμοποιείται όταν θέλουμε να λάβουμε το balance ενός κόμβου (χρησιμοποιήθηκε για testing όταν στήναμε το σύστημα).

5. **/chain/get (GET)** : Χρησιμοποιείται όταν ένας κόμβος προσπαθεί να κάνει resolve conflicts. Επιστρέφει την αλυσίδα που υπάρχει στον παρόν κόμβο.

Επίσης ορίζονται οι παρακάτω συναρτήσεις:

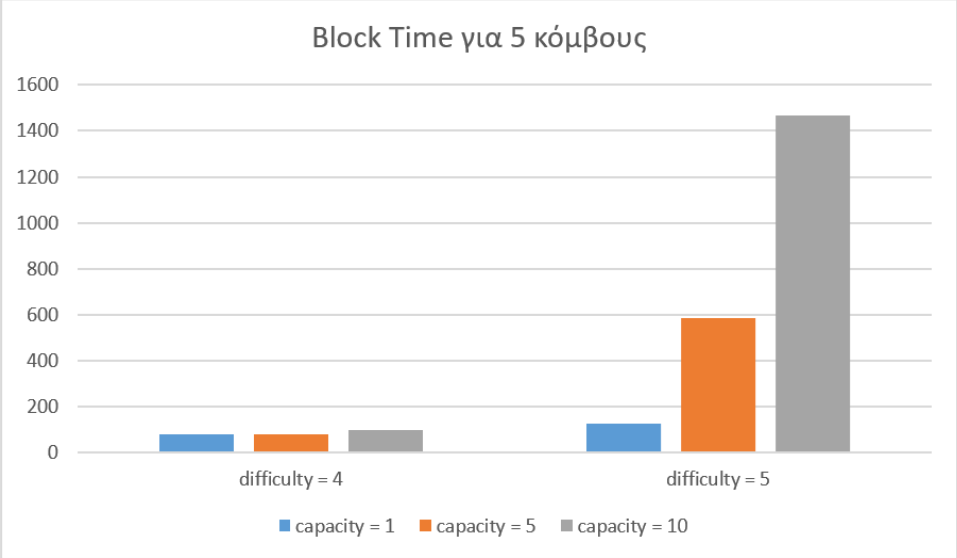
1. **client** : Αυτή η συνάρτηση τρέχει τον client ο οποίος δέχεται inputs από τον χρήστη και υλοποιεί μια από τις ζητούμενες λειτουργίες:
2. **simulation** : Αυτή η συνάρτηση διαβάζει transactions από ένα αρχείο και τα προσθέτει στο queue με τα pending transactions ενός κόμβου. Μάλιστα έχει γίνει χρήση lock, για να εγγυηθούμε πως όλα τα transactions του συστήματος θα προστεθούν ομοιόμορφα στο σύστημα. Μετά την προσθήκη όλων των transaction καλείται η συνάρτηση client για να μπορεί ο κάθε κόμβος να δει τι έχει γίνει.

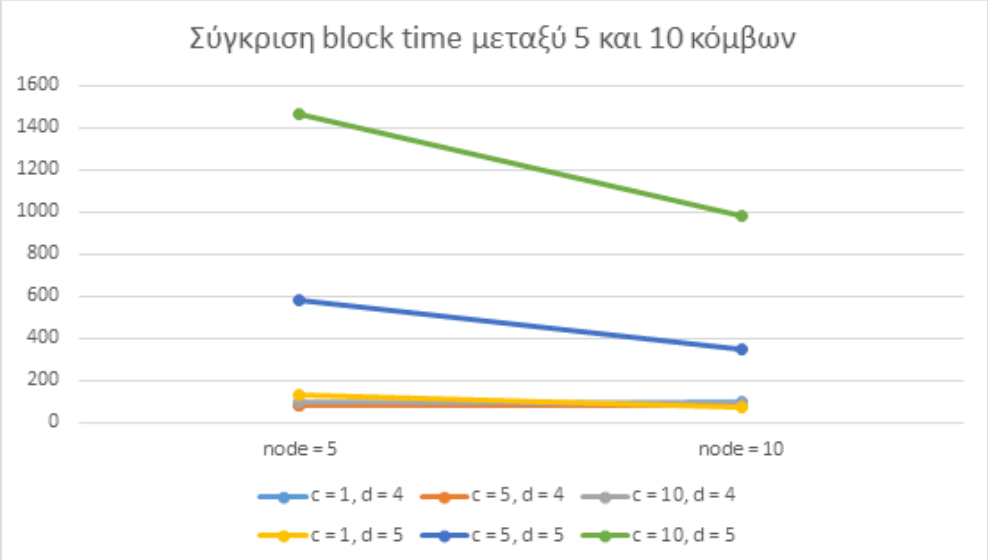
Πειράματα

Για το πειραματικό μέρος θα αξιολογήσουμε την απόδοση και την κλιμακωσιμότητα του συστήματος που δημιουργήσαμε. Σε Virtual Machines που μας δόθηκαν στην υπηρεσία Okeanos θα αναπαραστήσουμε τους κόμβους του blockchain μας και θα τρέξουμε τα εξής πειράματα. Θα τρέξουμε πρώτα μια σειρά απο transaction που μας δόθηκαν με 100 transactions για κάθε κόμβο. Θα μεταβάλλουμε το block size στις τιμές 1,5,10 , τον αριθμό των μηδενικών που απαιτούνται για το mine (difficulty) στις τιμές 4 και 5 και θα τρέξουμε τα παραπάνω για να ελέγξουμε την κλιμακωσιμότητα του συστήματος για 5 και 10 κόμβους . Θα σχεδιάσουμε τις γραφικές για το throughput καθώς και το block size του συστήματος. Θα παρουσιάσουμε παρακάτω τις γραφικές υπολογίζοντας και τοποθετώντας στη γραφική το μέσο όρο του block_time του κάθε μπλοκ όπως και το μέσο όρο του throughput τους.

Για την εύρεση του throughput υπολογίσαμε μέσω το πλήθος των transaction διαιρεμένο με τον τελικό χρόνο της προσομοίωσης. Παρακάτω παρατίθενται οι γραφικές μαζί με σχολιασμό για την κάθε περίπτωση:

Block time:



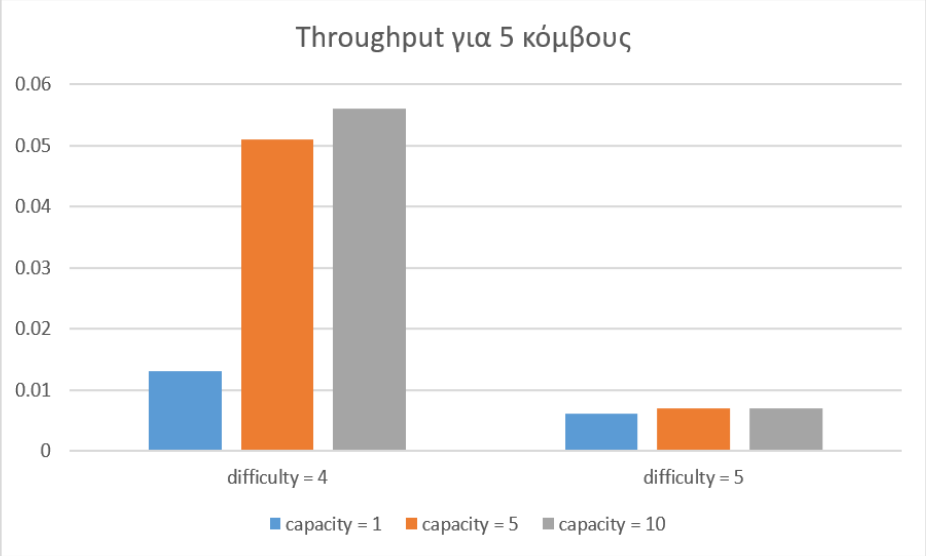


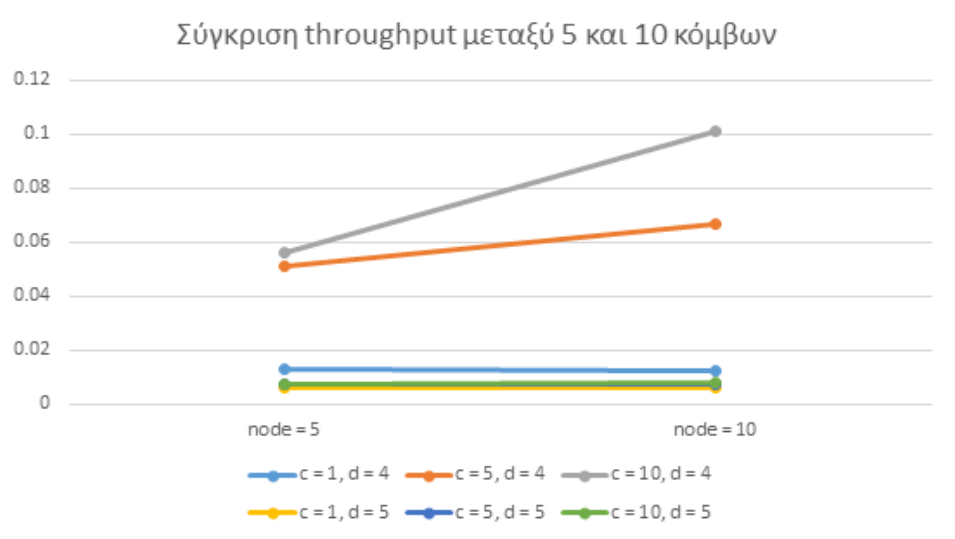
Σχολιασμός :

Παρατηρούμε πως η **αύξηση της χωρητικότητας** του κάθε μπλοκ προκαλεί και την **αύξηση του μέσου χρόνου** που απαιτείται για να γίνει mine ένα μπλοκ . Το γεγονός αυτό είναι αναμενόμενο καθώς θα αναμένει ο κάθε κόμβος περισσότερα transactions για να ξεκινήσει η εξόρυξη του block αυξάνοντας έτσι το χρόνο που απαιτείται.Επίσης παρατηρούμε πως η **αύξηση της δυσκολίας εξόρυξης** (περισσότερα μηδενικά απαιτούνται) προκαλεί και την

αύξηση του μέσου χρόνου που απαιτείται για να γίνει mine ένα μπλοκ. Όπως και πριν είναι αναμενόμενο καθώς όσο πιο συγκεκριμένο είναι το hash που πρέπει να βρεθεί τόσο πιο πολύ χρόνος θα απαιτείται για την εύρεση του (καθώς γίνεται τυχαία). Οι παραπάνω διαπιστώσεις εφαρμόζουν και για τις δύο προσομοιώσεις **ανεξαρτήτως του αριθμού των κόμβων** του συστήματος μας.
Κατά τη σύγκριση του συστήματος των 5 κόμβων με το σύστημα των 10 κόμβων παρατηρούμε πως και στην περίπτωση της δυσκολίας “βαθμού 4” και στην περίπτωση της δυσκολίας “βαθμού 5” **ο μικρότερος αριθμός κόμβων** συνεπάγεται την **πιο αργή εξόρυξη** ενός νέου μπλοκ. Μια τέτοια διαπίστωση είναι προφανής καθώς όσο περισσότερους κόμβους έχουμε να δουλεύουν πάνω στην εξόρυξη ενός μπλοκ (συνεπώς μεγαλύτερη υπολογιστική δύναμη) τόσο πιο πιθανό θα είναι κάποιος να βρει το hash που πληροί τις συνθήκες που απαιτούμε (#μηδενικών).

Throughput:





Σχολιασμός :

Όσον αφορά το throughput του συστήματος η **αύξηση του mining difficulty** μειώνει σημαντικά το **throughput** του συστήματος, το οποίο είναι αναμενόμενο καθώς αυξάνεται ο χρόνος που απαιτείται για το mining του κάθε block (για τους προαναφερθέντες λόγους) και συνεπώς λιγότερα transactions θα “επικυρώνονται” στη μονάδα του χρόνου. Η **αύξηση του capacity** προκαλεί όπως είναι λογικό **αύξηση του throughput** καθώς περισσότερα transactions θα επικυρώνονται σε ένα block και θα απαιτούνται πολύ λιγότερα block για να γίνουν mine. Τα παραπάνω ισχύουν και για σύστημα με 5 αλλά και με 10 κόμβους.
Κατά της **αύξηση του αριθμού των κόμβων** παρατηρούμε **μεγαλύτερο throughput** καθώς μειώνεται όπως είδαμε προηγουμένως ο χρόνος που απαιτείται για την εξόρυξη ενός μπλοκ λόγω μεγαλύτερης υπολογιστικής δύναμης με αποτέλεσμα να αυξάνεται το throughput.

Τέλος εργασίας