ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΚΑΤΑΝΕΜΗΜΕΝΑ ΣΥΣΤΗΜΑΤΑ – 9ο ΕΞΑΜΗΝΟ

Αγγελική Δημητρίου (03117106)

Αφροδίτη Τζομάκα (03117107)

Νικόδημος Μαρκέτος (03117095)

Ομάδα: 42

Αναφορά Εξαμηνιαίας Εργασίας

- NoobCash -

Εισαγωγή

Προκειμένου να εμπεδώσουμε και να εφαρμόσουμε στην πράξη τις βασικές αρχές των κατανεμημένων συστημάτων, κληθήκαμε να σχεδιάσουμε και να υλοποιήσουμε ένα απλό σύστημα blockchain. Πρόκειται ουσιαστικά για μια κατανεμημένη βάση που επιτρέπει στους χρήστες της να κάνουν δοσοληψίες (transactions) μεταξύ τους με ασφάλεια, χωρίς να χρειάζονται κάποια κεντρική αρχή. Το όνομα του συστήματός μας είναι noobcash και σε αυτό θα καταγράφονται δοσοληψίες μεταξύ n κόμβων που θα αποτελούν το υπό εξέταση δίκτυο. Το consensus μεταξύ των κόμβων θα εξασφαλίζεται μέσω Proof-of-Work (PoW). Επιπλέον κάθε κόμβος είναι και miner, υλοποιεί δηλαδή το PoW.

Ο κώδικάς μας αποτελείται από δύο κύρια κομμάτια (modules), το **backend** – όπου βρίσκεται όλη η λογική και το REST API – και το **cli** – όπου βρίσκεται η λειτουργικότητα και η διεπαφή με τον χρήστη. Ακολουθεί η ανάλυση της υλοποίησης των επιμέρους κομματιών.

Σχεδιασμός

►***Backend:***

Όπως ορίζει η εκφώνηση ο κάθε χρήστης του noobcash θα έχει ένα noobcash wallet προκειμένου να προσδιορίζεται μοναδικά και να πραγματοποιεί transactions. Το wallet αυτό υλοποιείται στο ***wallet.py*** αρχείο. Κάθε wallet αποτελείται από ένα private key που του επιτρέπει να συναλλάσσεται με ασφάλεια, το αντίστοιχο public key που απαιτείται για να λάβει ο κάτοχος του wallet χρήματα από άλλον χρήστη (ουσιαστικά αποτελεί τη διεύθυνση του χρήστη για τις συναλλαγές του) και τη διεύθυνση(ip, port) του wallet/χρήστη για το REST API. Εδώ βρίσκονται και οι ζητούμενες συναρτήσεις wallet_balance() και generate_wallet() με την λειτουργικότητα που περιγράφει η εκφώνηση.

Επόμενη λογική οντότητα που υλοποιήθηκε είναι το block, στο αρχείο ***block.py***. Στο αρχείο αυτό περιέχονται όλες οι ζητούμενες συναρτήσεις που αφορούν το block, mine_block() και validate_block(), με την λειτουργικότητα που περιγράφει η εκφώνηση, καθώς και κάποιες ακόμα βοηθητικές συναρτήσεις (validate_currentHash(), validate_previousHash(), to_json() ομώνυμης λειτουργικότητας). Τα πεδία του Object είναι επίσης εκείνα που περιγράφονται στην εκφώνηση. Σημειώνουμε εδώ πως προκειμένου να hashαριστεί το πεδίο listOfTransaction του object επιλέξαμε, μετά από μελέτη της σχετικής βιβλιογραφίας, να οργανώσουμε τα transactions σε ένα Merkle Tree. Με τον τρόπο αυτό, μπορούμε εύκολα, παίρνοντας τη ρίζα του δέντρου, να hashαρουμε όλα τα transactions.

Στη συνέχεια υλοποιήσαμε την δομή και τις βασικές λειτουργίες ενός transaction. Στο ***transaction.py*** αρχείο υπάρχουν οι συναρτήσεις sign_transaction() και verify_signature() από τις ζητούμενες που αφορούν τα transactions και οι βοηθητικές συναρτήσεις create_transaction_id(), to_json(). Όμοια και εδώ τα πεδία του Object είναι εκείνα που περιγράφονται στην εκφώνηση. Σημειώνουμε εδώ πως το transaction_id

έχει προκύψει hasháροντας ολόκληρο το object transaction. Ωστόσο αυτό συμβαίνει στην αρχή της εκτέλεσης όπου τα πεδία transaction_id, signature και transaction_outputs είναι None.

Προχωράμε στο αρχείο ***node.py***. Εκεί βρίσκεται ουσιαστικά όλος ο σχεδιασμός του συστήματος. Κάθε Node αντιπροσωπεύει έναν χρήστη του συστήματος. Έτσι λοιπόν, κάθε Node έχει ένα wallet, ένα id (myid), ένα miner_pid που αποτελεί το pid της διεργασίας miner κάθε κόμβου (κάθε χρήστης είναι και miner), τις πληροφορίες για το δίκτυο (network_info), τη δική του έκδοση του blockchain, 2 ειδών utxos, τις πληροφορίες του genesis (block και utxos) και κάποιες βοηθητικές παραμέτρους. Εδώ αξίζει να σημειωθεί η ύπαρξη των 2 ειδών utxos. Πιο συγκεκριμένα κρατάμε τα τελευταία έγκυρα utxos (που υπάρχουν στο τελευταίο επικυρωμένο μπλοκ του blockchain - prev_val_utxos) και τα τωρινά, προς επικύρωση, utxos (curr_utxos). Τα πρώτα χρησιμοποιούνται ώστε να γίνονται validate blocks που προκύπτουν από άλλους miners, ώστε στην περίπτωση που δεν υπάρχει conflict (διακλάδωση) να μην χρειάζεται roll back στο genesis. Τα δεύτερα χρησιμοποιούνται πριν το mining και παίρνουν την θέση των prev_val στο τέλος της διαδικασίας προσθήκης ενός block στο blockchain. Η κλάση επιπλέον περιέχει τις συναρτήσεις που αρχικοποιούν τους χρήστες και το δίκτυο (create_wallet(), genesis(), initialize_network(), update_network_info()), τις συναρτήσεις που αφορούν τις συναλλαγές του χρήστη (create_transaction(), create_and_broadcast_transaction(), validate_transaction(), view_transactions()), τις συναρτήσεις που αφορούν στο consensus (add_block(), check_mining(), start_mining(), validate_chain(), find_longest_chain(), resolve_conflict()) και τα broadcasts (broadcast_block/transaction()). Παρακάτω, στο σενάριο εκτέλεσης θα αναλύσουμε πως το σύστημα μας χρησιμοποιεί τις προαναφερθείσες συναρτήσεις/λειτουργίες και έχει την επιθυμητή συμπεριφορά.

Όπως αναφέρθηκε και παραπάνω κάθε χρήστης είναι και miner. Για να υλοποιήσουμε την απαίτηση αυτή δημιουργούμε ένα ξεχωριστό process που εκτελεί το mining μέσω του subprocess module της Python και συγκεκριμένα της κλάσης Popen. Με αυτόν τον τρόπο αποφεύγουμε το starvation των requests που δέχεται ένας κόμβος σε περίπτωση καταιγισμού δοσοληψιών. Ο κώδικας που ξεκινάει, σταματάει και ελέγχει την διαδικασία του mining βρίσκεται στο αρχείο **miner.py**.

Από την μεριά του REST API έχουμε το αρχείο ***rest.py***. Εδώ υπάρχουν τα endpoints στο οποία «ακούει» κάθε node/χρήστης του δικτύου. Για την ανάπτυξη αυτής της διεπαφής χρησιμοποιήθηκε η βιβλιοθήκη Flask της Python. Τα endpoints είναι οργανωμένα αρχικά σε σχέση με το αν θα κληθούν από το backend ή από το cli και δευτερευόντως ανάλογα με το αντικείμενο στο οποίο αναφέρονται (πχ. bootstrap, node, transaction).

Τέλος, έχουμε τα βοηθητικά αρχεία ***utils.py*** και ***config.py***. Το πρώτο υλοποιεί τη λογική του broadcast και τη συλλογή των στατιστικών που ζητούνται στην εκφώνηση (average block time, throughput) ενώ το δεύτερο περιέχει τις τιμές που ορίζουν την παραμετροποίηση του συστήματος. Αυτές είναι οι NODES (πλήθος χρηστών), CAPACITY (χωρητικότητα του block), DIFFICULTY (ο αριθμός των μηδενικών με τα οποία ξεκινά το hash κάθε block) και BOOTSTRAP_IP (η IP του κόμβου που θα λειτουργήσει ως bootstrap/coordinator).

►*CLI:*

Για το frontend της εφαρμογής και την υλοποίηση της διεπαφής με τον χρήστη/client αναπτύξαμε ένα Command Line Interface ο κώδικας του οποίου βρίσκεται στο αρχείο **cli.py** του module cli. Ξεκινώντας το cli ουσιαστικά «σηκώνουμε» και τα nodes του δικτύου. Με την επιλογή –n καθορίζουμε τον αριθμό αυτών καθώς και πως ο εν λόγω node είναι ο bootstrap ενώ ταυτόχρονα δίνουμε και το ζεύγος ip/port που τους χαρακτηρίζει. Με τη βοήθεια της διεπαφής αυτής, ο χρήστης έχει την δυνατότητα να εκτελεί transactions προσδιορίζοντας το id του κόμβου/παραλήπτη και το αντίστοιχο ποσό, να ελέγχει το υπόλοιπο του λογαριασμού του, να βλέπει τα τελευταία επικυρωμένα transactions, να τρέχει tests και να λαμβάνει τα στατιστικά της επίδοσης του συστήματος.

► *Σενάριο Εκτέλεσης:*

Αρχικά εκκινούμε τον bootstrap κόμβο προσδιορίζοντας το πλήθος των κόμβων του δικτύου. Στη συνέχεια καλεί τη συνάρτηση create_wallet() με όρισμα την ip/port διεύθυνση του ώστε να πάρει public/private key και εκτελεί το genesis όπως αυτό ορίζεται στην εκφώνηση. Έπειτα εισέρχονται στο σύστημα οι υπόλοιποι κόμβοι δημιουργώντας και εκείνοι το δικό τους wallet και κοινοποιώντας τις πληροφορίες ip/port και public key στον bootstrap. Εκείνος είναι υπεύθυνος να «εγγράφει» τους κόμβους στο σύστημα, να δημιουργεί το ring των πληροφοριών και να αποδίδει ids σε κάθε έναν από τους κόμβους. Μόλις εισέλθει και ο τελευταίος κόμβος, ο bootstrap διαμοιράζεται τις πληροφορίες του συστήματος. γνωστοποιεί σε όλους τους κόμβους το πρώτο block, δηλαδή το genesis block και εκτελεί τα πρώτα transactions που αποδίδουν τα 100 NBCs σε κάθε κόμβο. Από το σημείο αυτό το σύστημα μπορεί να λειτουργήσει κανονικά.

Πιο συγκεκριμένα, αν το CAPACITY είναι μικρότερο ή ίσο των αρχικών transaction το mining ξεκινάει. Αλλιώς περιμένει μέχρι να εκτελεστούν CAPACITY transactions συνολικά. Κάθε φορά που στέλνεται ένα transaction μέσω του cli, το αντίστοιχο endpoint (cli/transaction) καλεί την συνάρτηση create_and_broadcast_transaction() η οποία με τη σειρά της δημιουργεί και κάνει broadcast το transaction. Η φάση της δημιουργίας περιλαμβάνει τους ελέγχους εγκυρότητας της συναλλαγής (πχ όχι αρνητικά amounts) και ύστερα προσπαθεί να βρει τα inputs στα curr_utxos και κατά συνέπεια να πιστοποιήσει ότι ο αποστολέας έχει αρκετά NBCs για να εκτελέσει τη συναλλαγή. Αν το παραπάνω ισχύει δημιουργείται και υπογράφεται το transaction, δημιουργούνται τα outputs της συναλλαγής (ενημερώνοντας και τα curr_utxos), αυτή προστίθεται στις pending συναλλαγές του συγκεκριμένου κόμβου και ελέγχουμε αν πλέον χρειάζεται να γίνει mining. Από την αντίθετη πλευρά όταν ένας κόμβος λάβει μια συναλλαγή που έχει προκύψει από broadcast στο endpoint (/transaction/receive) θα χρειαστεί να την επικυρώσει. Η διαδικασία αυτή περιλαμβάνει ξανά ελέγχους εγκυρότητας, ύπαρξης των απαιτούμενων inputs και χρημάτων και ενημέρωσης των curr_utxos και της λίστας με τα pending transactions. Ξανά χρειάζεται να ελέγξουμε την ανάγκη για mining.

Στην περίπτωση που πρέπει να γίνει mining αρχικά δημιουργούμε το block με τα CAPACITY pending transactions και εκκινούμε την διεργασία του miner (αν αυτός δεν τρέχει ήδη). Εκείνος είναι υπεύθυνος για την εκτέλεση της συνάρτησης mine_block() που υλοποιεί το PoW. Όταν η συνάρτηση αυτή επιστρέψει το block με το nonce που προέκυψε ανακοινώνεται στον κόμβο που ξεκίνησε τη συγκεκριμένη διαδικασία mining και εκείνος με τη σειρά του προσπαθεί να το προσθέσει στο blockchain του. Η προσπάθεια αυτή συνοψίζεται στη συνάρτηση add_block() την οποία θα αναλύσουμε αμέσως μετά. Μόλις αυτή επιτύχει ο κόμβος κάνει broadcast το mined block. Όταν ένας node λάβει ένα καινούριο block στο endpoint /block/receive, αρχικά σταματά τον δικό του miner στην περίπτωση που τρέχει (συνήθως αυτή είναι η περίπτωση) και στη συνέχεια επιχειρεί και αυτός την add_block().

Η add_block() ξεκινάει και εκείνη με κάποιους ελέγχους εγκυρότητας και συνεχίζει κοιτώντας τις εξής δύο περιπτώσεις: α) το previous hash του block που ελέγχει ισούται με το current hash του τελευταίου block του blockchain β) το παραπάνω δεν ισχύει και άρα έχουμε conflict. Στην περίπτωση α) πραγματοποιείται επικύρωση όλων των transactions που υπάρχουν στο block χρησιμοποιώντας τα prev_val_utxos κι όχι τα current όπως παλιότερα. Εδώ να σημειωθεί πως στην περίπτωση που η συνάρτηση κλήθηκε από το block/create σίγουρα βρισκόμαστε στην περίπτωση αυτή. Αν επαληθευτούν όλες οι συναλλαγές, μπορούμε να τις αφαιρέσουμε από τη λίστα με τα pending transactions και να προσθέσουμε το block στο blockchain. Ταυτόχρονα θα ενημερωθούν τα έγκυρα utxos και τέλος θα επιχειρήσουμε να ξαναπροσθέσουμε τις δοσοληψίες που υπήρχαν στα pending transactions αλλά όχι στο block που λήφθηκε και θα χρειαστεί να γίνουν αργότερα mine. Στη β) περίπτωση υπάρχει κάποια διακλάδωση στην αλυσίδα του κόμβου και η προκύπτουσα σύγκρουση θα πρέπει να επιλυθεί με τη βοήθεια του αλγορίθμου consensus. Αν πρόκειται για block που ταιριάζει σε προηγούμενο σημείο της αλυσίδας το αγνοούμε γιατί έχουμε επιλέξει να θεωρήσουμε πως δημιουργεί μικρότερη αλυσίδα. Σε αντίθετη περίπτωση η συνάρτηση resolve_confict() θα καλέσει την find_longest_chain() η οποία θα μάθει τα μήκη των αλυσίδων όλων των κόμβων του δικτύου και θα επιστρέψει εκείνους με το μέγιστο μήκος. Αν ο κόμβος που κάλεσε τη resolve_conflict() είναι μέλος της παραπάνω λίστας, θα κρατήσει την αλυσίδα του και θα προχωρήσει. Αντίθετα, θα ζητήσει τις αλυσίδες των κόμβων της λίστας αυτής και το blockchain του πρώτου από αυτούς που μπορεί να επαληθευτεί θα υιοθετηθεί τελικά. Η επαλήθευση του blockchain γίνεται μέσω της συνάρτησης validate_chain() η οποία θα χρειαστεί να διατρέξει όλη την αλυσίδα, ξεκινώντας από το genesis block (και τα genesis utxos) και προσπαθώντας να κάνει διαδοχικά add_block() και validate_transaction(). Προφανώς, αυτό είναι ένα χρονοβόρο σενάριο, το οποίο είναι όμως αναπόφευκτο δεδομένης της παραπάνω σχεδίασης.

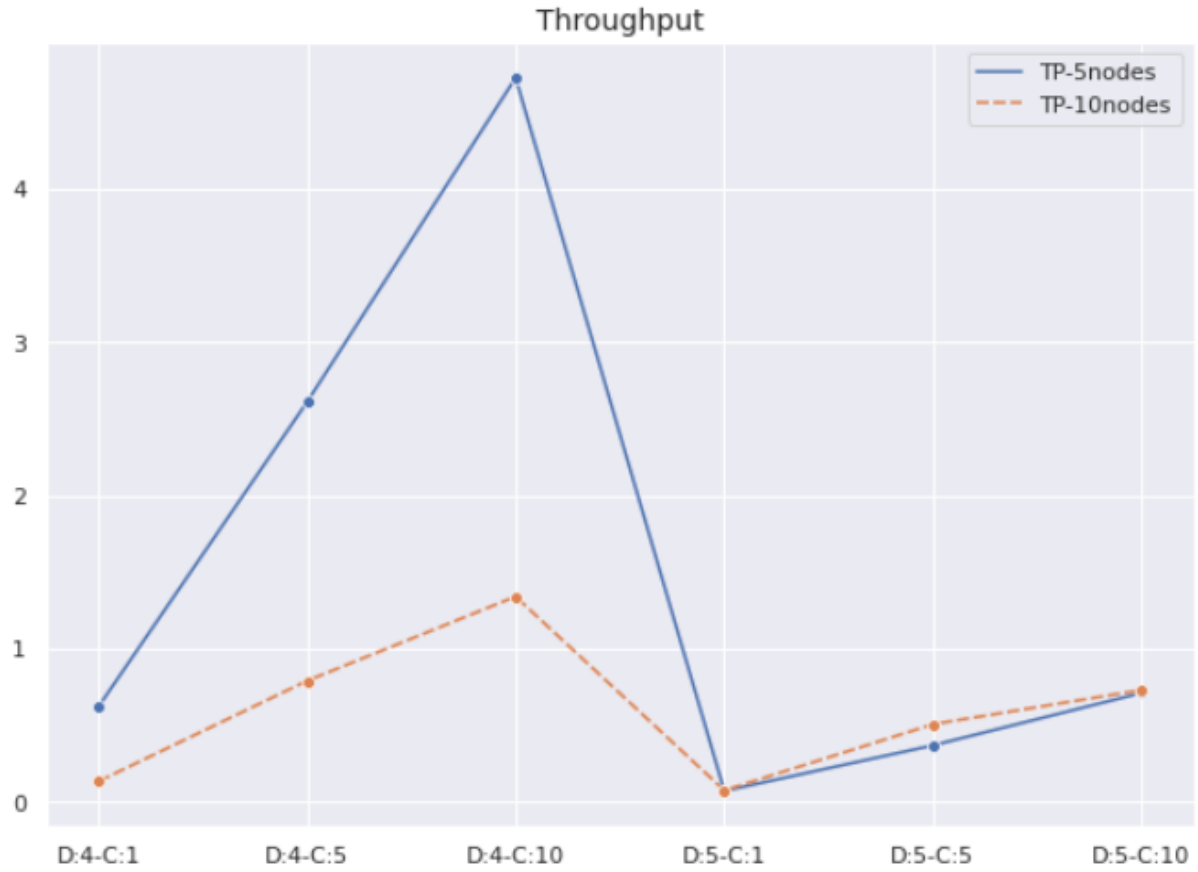
Μετά από την προσπάθεια εισαγωγής του καινούριου block στην αλυσίδα, ο κόμβος που το έλαβε θα χρειαστεί να ελέγξει αν η λίστα των pending transactions του έχει μήκος μεγαλύτερο ή ίσο της χωρητικότητας ώστε να ξεκινήσει και πάλι mining. Αν η παραπάνω συνθήκη δεν ισχύει αυτό δε σημαίνει υποχρεωτικά πως δεν υπάρχουν transactions στο δίκτυο που πρέπει να γίνουν mine, αλλά πως ίσως λόγω των προηγούμενων συγκρούσεων στο chain απορρίφθηκαν δοσοληψίες διότι μέχρι

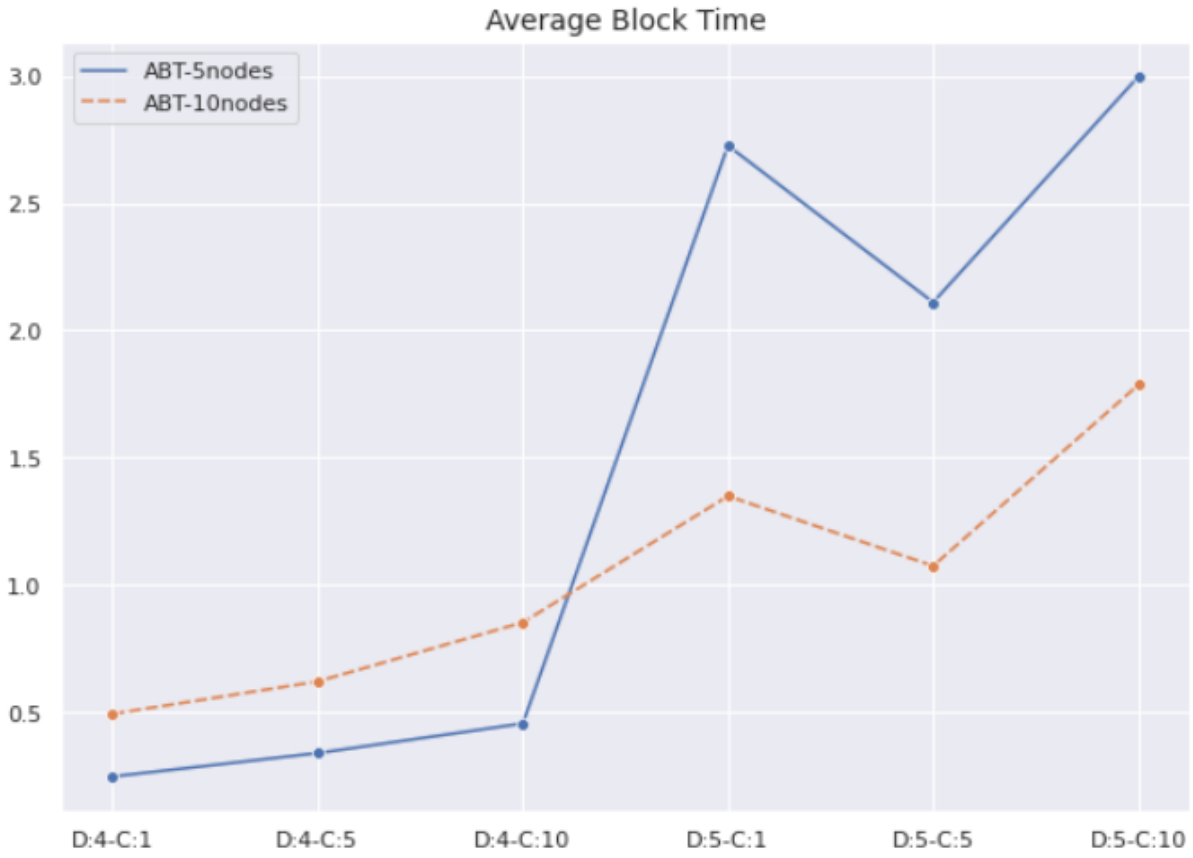
εκείνη τη στιγμή δεν υπήρχαν τα αντίστοιχα inputs που να τις δικαιολογούν. Τα τελευταία έχουν πλέον δημιουργηθεί με την προσθήκη του καινούριου block και άρα για να διορθώσουμε αυτή την ασυνέπεια θα ρωτήσουμε όλους τους κόμβους του δικτύου για πιθανά pending transactions. Αν καταφέρουμε να προσθέσουμε transactions που φτάνουν το CAPACITY (και άρα θα κάνουμε mine) θα σταματήσουμε να ρωτάμε αλλιώς αν δε βρούμε αρκετά pending transactions απλά θα προχωρήσουμε παρά κάτω.

Αποτελέσματα Πειραμάτων

Εκτελούμε πειράματα με 5 και 10 κόμβους στο δίκτυο, οι οποίοι πραγματοποιούν ταυτόχρονα τα αντίστοιχα transactions που δίνονται. Προκειμένου να εξετάσουμε την ορθότητα των αποτελεσμάτων μας καταγράφουμε τα υπόλοιπα όλων των nodes και τα προσθέτουμε για να πιστοποιήσουμε πως αθροίζουν στο ποσό που έλαβε ο bootstrap κατά το genesis. Ταυτόχρονα, ελέγχουμε και τα logs. Ενδεικτικά screenshots φαίνονται στο τέλος της εργασίας, στο Παράρτημα.

Παρακάτω παρουσιάζονται τα συγκριτικά διαγράμματα για τις μετρικές average block time και throughput:





► *Απόδοση Συστήματος*

Αρχικά, θα σχολιαστεί η απόδοση του συστήματος για τις παραμετροποιήσεις που αφορούν τους 5 κόμβους (μπλε γραμμή στα παραπάνω διαγράμματα). Όπως είναι αναμενόμενο, όσον αφορά το throughput βλέπουμε μεγάλη αύξηση όσο αυξάνεται η χωρητικότητα με σταθερή τη δυσκολία. Με άλλα λόγια, το σύστημα εξυπηρετεί αποδοτικότερα τις συναλλαγές στη μονάδα του χρόνου μιας και δεν υπάρχει η συμφόρηση που προκαλεί το συχνό mining. Ωστόσο, για difficulty 5 οι τιμές του throughput είναι σταθερά μικρότερες από εκείνες για difficulty 4. Η μεγάλη διαφορά που παρατηρείται καθιστά φανερό τον αντίκτυπο της δυσκολίας του PoW στη συνολική απόδοση.

Όσον αφορά τη μετρική average block time, δηλαδή το μέσο χρόνο για να γίνει mine ένα block, όπως θα περιμέναμε, με την αύξηση του difficulty έχουμε μεγάλη άνοδο στην τιμή του. Κι αυτό γιατί εξ ορισμού είναι δυσκολότερο να ικανοποιηθεί η συνθήκη του PoW κι άρα πιο χρονοβόρο. Όσον αφορά την αύξηση της χωρητικότητας με σταθερή τη δυσκολία, παρατηρείται διαφορετική συμπεριφορά για διαφορετικά difficulties. Συγκεκριμένα, για difficulty 4 φαίνεται να υπάρχει αυξητική τάση, γεγονός που δικαιολογείται από τον πολλαπλάσιο χρόνο που θα πάρει για να γίνουν hash από το merkle tree τα πολλαπλάσια transactions του block. Αυτή η παρατήρηση, ταιριάζει και στην περίπτωση του difficulty 5 με εξαίρεση το συνδυασμό d=5, c=5 που οδηγεί σε καλύτερο block time από το αναμενόμενο. Μιας και αυτή η συμπεριφορά εμμένει και

στα 10 nodes μπορούμε να θεωρήσουμε πως πρόκειται για μια καθολικά βέλτιστη παραμετροποίηση για τη συγκεκριμένη τιμή δυσκολίας.

Σε γενικές γραμμές, τα configurations με τη μικρότερη δυσκολία έχουν καλύτερη απόδοση. Παρόλα αυτά, σε ένα πραγματικό σενάριο το μικρό average block time δεν είναι επιθυμητό οπότε η αύξηση της δυσκολίας είναι αναπόφευκτη και κατά συνέπεια υπάρχει ένα trade-off μεταξύ των 2 μετρικών.

► *Κλιμακωσιμότητα Συστήματος*

Επαναλαμβάνοντας τα παραπάνω πειράματα για 10 κόμβους (πορτοκαλί γραμμή παραπάνω) παρατηρούμε πως ποιοτικά τουλάχιστον υπάρχει κάποια συνέπεια. Από ποσοτική άποψη, μπορούμε να διακρίνουμε δύο περιπτώσεις. Για τη μικρότερη τιμή δυσκολίας παρατηρείται μικρότερο throughput και μεγαλύτερο average block time. Αντίθετα, για difficulty 5 γίνεται η αντίστροφη παρατήρηση.

Θα μπορούσαμε, κατά συνέπεια, να πούμε πως το σύστημα μας είναι κλιμακώσιμο για μεγαλύτερες τιμές difficulty, αλλά όταν το mining διαρκεί πολύ λίγο χρόνο (d=4) οι συγκρούσεις που προκύπτουν μεταξύ των κόμβων επισκιάζουν την καλύτερη απόδοση του. Για παράδειγμα, στο configuration d=4, c=1, 10 nodes παρατηρήθηκε ότι τα conflicts οδηγούσαν σε απανωτές αλλαγές αλυσίδας ενός κόμβου, καθυστερώντας ακόμα και το ρυθμό αποδοχής transactions από το CLI.

ΠΑΡΑΡΤΗΜΑ

5 NODES

```
balance
You have 145 NBCs left.
(noob-cli)> stats
stats
Average Block time: 0.21715669542829566, Throughput: 0.6154895791232105.
```

```
balance
You have 66 NBCs left.
(noob-cli)> stats
stats
Average Block time: 0.28857028595755035, Throughput: 0.6340113365936387.
```

```
balance
You have 101 NBCs left.
(noob-cli)> stats
stats
Average Block time: 0.21791898870022497, Throughput: 0.6091463481928415.
```

```
balance
You have 62 NBCs left.
(noob-cli)> stats
stats
Average Block time: 0.2537177076963621, Throughput: 0.6283427904932731.
```

```
balance
You have 126 NBCs left.
(noob-cli)> stats
stats
Average Block time: 0.2591208141540813, Throughput: 0.6118572339036.
```

10 NODES

```
balance
You have 152 NBCs left.
(noob-cli)> stats
stats
Average Block time: 1.066910423551287, Throughput: 0.7405283854927718.
```

```
balance
You have 132 NBCs left.
(noob-cli)> stats
stats
Average Block time: 1.7483165945325578, Throughput: 0.7391085427386952.
```

```
balance
You have 135 NBCs left.
(noob-cli)> stats
stats
Average Block time: 1.2890983990260534, Throughput: 0.7386164773846422.
```

```
balance
You have 102 NBCs left.
(noob-cli)> stats
stats
Average Block time: 3.4674131189073836, Throughput: 0.7191733922908979.
```

```
balance
You have 57 NBCs left.
(noob-cli)> stats
stats
Average Block time: 1.7503805569240025, Throughput: 0.7284309292051685.
```

```
balance
You have 37 NBCs left.
(noob-cli)> stats
stats
Average Block time: 0.8723609924316407, Throughput: 0.7230675733811102.
```

```
balance
You have 98 NBCs left.
(noob-cli)> stats
stats
Average Block time: 3.402358000619071, Throughput: 0.7285700957888469.
```

```
balance
You have 69 NBCs left.
(noob-cli)> stats
stats
Average Block time: 1.2900726454598563, Throughput: 0.7259840577780775.
```

```
balance
You have 143 NBCs left.
(noob-cli)> stats
stats
Average Block time: 2.1702154432024274, Throughput: 0.7365389614923987.
```

```
balance
You have 75 NBCs left.
(noob-cli)> stats
stats
Average Block time: 0.7970124244689941, Throughput: 0.7289357621362406.
```

"VIEW" Consistency

```
(noob-cli)> view
view
60ae8426a8:  Sender: 0, Recipient: 8, Amount: 8 NBCs.
b62cf26b52:  Sender: 0, Recipient: 5, Amount: 6 NBCs.
2b33a5dc42:  Sender: 0, Recipient: 9, Amount: 2 NBCs.
6591084ee9:  Sender: 0, Recipient: 8, Amount: 1 NBCs.
4730feb6bb:  Sender: 0, Recipient: 9, Amount: 8 NBCs.
```

```
(noob-cli)> view
view
60ae8426a8:  Sender: 0, Recipient: 8, Amount: 8 NBCs.
b62cf26b52:  Sender: 0, Recipient: 5, Amount: 6 NBCs.
2b33a5dc42:  Sender: 0, Recipient: 9, Amount: 2 NBCs.
6591084ee9:  Sender: 0, Recipient: 8, Amount: 1 NBCs.
4730feb6bb:  Sender: 0, Recipient: 9, Amount: 8 NBCs.
```