Εξαμηνιαία Εργασία Στα Κατανεμημένα Συστήματα

Ακ. έτος 2019-2020, 9ο εξάμηνο, Σχολή ΗΜΜΥ

Ομάδα:

Σπυραντώνης Κουτρούλης **03114864**
Σεραφείμ Παναγιωτίδης **03115131**

Συνοπτική Παρουσίαση

Στην άσκηση αυτή κληθήκαμε να υλοποιήσουμε ένα κρυπτονόμισμα το οποίο λέγεται **noobcash**. Πιο συγκεκριμένα, υλοποιήσαμε ένα δίκτυο ομότιμων κόμβων (**miners**), το οποίο με χρήση της τεχνολογίας **blockchain** επιτρέπει την αποθήκευση δοσοληψιών (**transactions**), που εκτελούνται μεταξύ των χρηστών του συστήματος (**clients**), με κατανεμημένο τρόπο χωρίς να χρειάζεται, δηλαδή, η μεσολάβηση κάποιας κεντρικής αρχής. Παράλληλα φροντίσαμε να εξασφαλίσουμε την ασφάλεια και την ακεραιότητα του συστήματος, καθώς πήραμε μέτρα ώστε να αποφεύγονται επιθέσεις double spending και forgery attacks. Τέλος, θα πρέπει να αναφέρουμε ότι η συμφωνία (**consensus**) και ο συντονισμός μεταξύ των miners εξασφαλίζεται με χρήση proof-of-work. Στην συνέχεια, ακολουθεί αναλυτικότερη παρουσίαση του συστήματος.

Αρχικοποίηση του Συστήματος

Αρχικά ενεργοποιείται ο πρώτος κόμβος του δικτύου (**Bootstrap Node**). Ο Bootstrap Node φροντίζει να φτίαξει το πρώτο block του blockchain (**Genesis Block**), το οποίο περιέχει ένα μόνο transaction το οποίο δεν έχει αποστολέα και έχει ως παραλήπτη τον εαυτό του. Όταν συνδεθούν όλοι οι κόμβοι στο δίκτυο (υποθέτουμε ότι ο αριθμός των κόμβων που συμμετέχουν είναι γνωστός εξ αρχής), το ποσό αυτό θα μοιραστεί ισάξια μεταξύ όλων των miners. Στην συνέχεια, ο Bootstrap Node απλά περιμένει να συνδεθούν όλοι οι υπόλοιποι κόμβοι στο δίκτυο.

Κάθε φορά που ενεργοποιείται ένας άλλος **Node** φροντίζουμε να επικοινωνήσει αμέσως με τον Bootstrap Node (υποθέτουμε ότι η διεύθυνση ip του καθώς και η θύρα στην οποία ακούει είναι από πριν γνωστά). Αυτό γίνεται προκείμενου να κάνει 'εγγραφή' στο σύστημα. Αρχικά, στέλνει στον Bootstrap Node ένα μήνυμα το οποίο περιλαμβάνει **την διεύθυνση ip του, την θύρα στην οποία ακούει καθώς και την διεύθυνση του πορτοφολιού του**. Ο Bootstrap Node αποθηκεύει (για κάθε κόμβο που συνδέεται σε αυτόν) τα στοιχεία αυτά σε μια δομή η οποία λέγεται **ring**. Στην συνέχεια ο Bootstrap Node απαντάει στον απλό Node με ένα **nodeId** (ο Bootstrap Node έχει nodeId ίσο με μηδέν, ενώ κάθε κόμβος που συνδέεται λαμβάνει το αμέσως επόμενο διαθέσιμο nodeId --> autoIncrement). Τέλος, μόλις ο Node λάβει το nodeId το οποίο του στάλθηκε από τον Bootstrap Node, ενημερώνει τον τελευταίο ότι το όλη η διαδικασία κύλισε ομαλά στέλνοντας του ένα μήνυμα επιβεβαίωσης (**ACK**).

Όταν ο Bootstrap Node λάβει ack από όλους τους κόμβους του δικτύου, δηλαδή είναι σίγουρος ότι όλοι έχουν συνδεθεί στο σύστημα, στέλνει σε αυτούς μέσω broadcast τη δομή **ring** και το **blockchain** (το οποίο προς το παρόν περιέχει μόνο το Genesis Block). Πλέον, όλοι οι κόμβοι γνωρίζουν πως να επικοινωνήσουν μεταξύ τους και

έχουν όλοι το ίδιο blockchain. Το σύστημα έχει έρθει σε σύγκλιση και μπορεί να ξεκινήσει η δημιουργία δοσοληψιών μεταξύ των χρηστών.

Τέλος, η αρχικοποίηση του συστήματος περιλαμβάνει τα αρχικά transactions μέσω των οποίων ο Bootstrap Node μοιράζει το αρχικό ποσό που έδωσε στον εαυτό του ισάξια σε όλους τους υπόλοιπους κόμβους του συστήματος.

Βασικές Λειτουργίες του Συστήματος

Αρχικά θα περιγράψουμε κάποιες βασικές δομές δεδομένων που έχει ο κάθε miner. Πρώτα από όλα περιέχει τη δομή **ring** η οποία, όπως αναφέρθηκε και παραπάνω, περιέχει για κάθε άλλο κόμβο του δικτύου την ip του, το port του, την wallet address του και το nodeId του. Έτσι, κάποιος κόμβος μπορεί να επικοινωνήσει με όλους του υπόλοιπους. Στην συνέχεια, περιέχει ένα dictionary από dictionaries το οποίο ονομάσαμε **unspentTranasactionsMap.** Σε αυτό φροντίζουμε να κρατάμε για κάθε διεύθυνση πορτοφολιού (για κάθε client) τα transactionOutputs των transactions που αναφέρονται στην εν λόγω διεύθυνση. Έτσι, είμαστε σε θέση ανά πάσα στιγμή να υπολογίσουμε το υπόλοιπο του κάθε πορτοφολιού. Ακόμη, περιέχει μια δομή που λέγεται **tempUsedOutputs** η οποία θα παρουσιαστεί αναλυτικότερα παρακάτω. Σκοπός της είναι η διευκόλυνση της διαδικασίας consensus όταν βρεθεί ένα block και γίνει broadcast σε όλους. Τέλος περιέχει μια δομή που λέγεται **transactionPool** η οποία είναι μια λίστα στην οποία εισάγονται transactions που έχουν δημιουργηθεί. Ένας δαίμονας φροντίζει να τραβάει transactions από το transactionPool και να τα παραδίδει στον miner ο οποίος αφού τα επικυρώσει τα εισάγει στο block του. Ακολουθεί και μια σχηματική παρουσίαση των παραπάνω δομών προκείμενου να γίνουν περισσότερο κατανοητές:

```
transactionPool = [
                'transactionId1': transaction1,
                'transactionId2': transaction2,
                 ...
               ]
```

```
unspentTransactionsMap = {
'address1':   {'transactionOutputId11': transactionOutput11,
        'transactionOutputId12': transactionOutput12, ...},
'address2': {'transactionOutputId21': transactionOutput21,
'transactionOutputId22': transactionOutput22, ...},
                …}
```

```
ring = [
        {'nodeId': nodeId1, 'ip': ipAddr1, 'port': port1, 'address': publicKey1},
        {'nodeId': nodeId2, 'ip': ipAddr2, 'port': port2, 'address': publicKey2},
        …]

tempUsedOutputs = {
                'transactionOutputId11': [transactionOutput11, address1],
                'transactionOutputId12': [transactionOutput12, address2],
                ...}
```

Στην συνέχεια, παρουσιάζονται οι βασικές λειτουργίες του συστήματος:

Δημιουργία transaction

Δημιουργείται ένα transaction με βάση πληροφορίες που έχει αποστείλει ο client και στην συνέχεια υπογράφεται από αυτόν. Αν μπορεί να γίνει το transaction (όλα τα πεδία αυτού δεν παραβιάζουν κάποιον περιορισμό (πχ συναλλαγή ποσού που δεν διαθέτει ο κάτοχος)), τότε τοποθετούμε στο transaction τα transactionInputs που αυτό χρειάζεται προκείμενου να γίνει. Στην συνεχεία, αν όλα έχουν πάει καλά κάνουμε broadcast το transaction σε όλους τους υπόλοιπους κόμβους του δικτύου. Τέλος, προσθέτουμε το transaction στην δομή transactionPool.

Λήψη transaction

Κατά τη λήψη ενός transaction, ο miner το παίρνει όπως είναι και το τοποθετεί κατευθείαν στην δομή transactionPool.

Εισαγωγή transaction σε block

Ο miner, μέσω ενός δαίμονα, τραβάει transactions από την δομή transactionPool, τα επικυρώνει και αν όλα πάνε καλά τα προσθέτει στο τρέχων block του. Αν το transaction που εισάγουμε ‘γεμίσει’ το block τότε ο εν λόγω κόμβος ξεκινάει να κάνει mine το block .

Επικύρωση transaction

Αρχικά ελέγχουμε την υπογραφή του αποστολέα. Για τις ψηφιακές υπογραφές έχουμε χρησιμοποιήσει τον αλγόριθμο RSA και έχουμε φροντίσει οι παράμετροι αυτού να είναι αρκετά μεγάλες (1024 bits), έτσι ώστε να αποφεύγονται επιθέσεις πλαστογράφησης. Στην συνέχεια ελέγχουμε αν όντως υπάρχουν τα transactionInputs του εν λόγω transaction στην δομή unspentTransactionMap, δηλαδή ελέγχουμε αν υπάρχουν τα λεφτά για να γίνει το transaction. Παράλληλα με αυτόν τον τρόπο αποφεύγονται και επιθέσεις τύπου double-spending. Αν υπάρχουν, τα αφαιρούμε από την δομή και τα τοποθετούμε προσωρινά (μέχρι να βρεθεί winning block) στην δομή tempUsedOutputs, ενώ μετά παράγουμε τα transactionOutputs του εν λόγω transaction τα οποία και τοποθετούμε στην δομή unspentTransactionMap. Αν δεν υπάρχουν, θεωρούμε το transaction invalid και το πετάμε.

Block mining

Όταν γεμίσει ένα block, ξεκινάμε να το κάνουμε mine. Το mine εξαρτάται άμεσα από την τιμή της παραμέτρου DIFFICULTY. Ψάχνουμε να βρούμε το nonce εκείνο το οποίο θα κάνει το hash του block να ξεκινάει με DIFFICULTY μηδενικά. Για το hash του block χρησιμοποιείται η συνάρτηση hash sha256. Όποιος miner το βρει πρώτος κάνει broadcast το block του σε όλους τους υπόλοιπους (έχουμε φροντίσει μέσω boolean μεταβλητών να μην γίνεται ο miner να τραβάει transactions από το transactionPool για όσο χρόνο κάνει mine).

Λήψη block

Κατά τη λήψη ενός block, το πρώτο πράγμα που κάνει ο miner είναι να σταματήσει την διαδικασία mining. Στην συνέχεια επικυρώνει το block και αν όλα πάνε καλά το προσθέτει στο blockchain που διατηρεί. Αν το block βγει invalid τότε φροντίζει να προχωρήσει στην διαδικασία επίλυσης συγκρούσεων. Τέλος, σε κάθε περίπτωση, είτε το block είναι valid είτε όχι, εκτελεί μια διαδικασία προσαρμογής η οποία ουσιαστικά επιτυγχάνει το consensus κομμάτι του συστήματος.

Επικύρωση block

Κάθε block πέραν του hash του, περιέχει πεδίο το οποίο κρατάει το hash του προηγούμενου block του blockchain. Κατά την επικύρωση ενός block που έχει λάβει, ο miner αρχικά ελέγχει εάν το hash του block αυτού είναι όντως αυτό που δηλώνεται. Στην συνέχεια, ελέγχει αν το πεδίο previousHash του block που έλαβε είναι ίδιο με το πεδίο currentHash του τελευταίου block που είχε προστεθεί στο blockchain. Εάν ισχύουν και οι δύο συνθήκες τότε το block θεωρείται valid, αλλιώς θεωρείται invalid.

Διαδικασία επίλυσης συγκρούσεων

Στην περίπτωση κατά την οποία ένας miner λάβει ένα invalid block προχωράει σε αυτή την διαδικασία. Ο miner ζητάει από όλους τους άλλους κομβους του δικτύου το blockchain που αυτοί έχουν σχηματίσει. Επιλέγει και κρατάει αυτό με το μεγαλύτερο μήκος. Σε περίπτωση που υπάρχουν πολλαπλές επιλογές κρατάει την πρώτη. Στην συνέχεια επικυρώνει το blockchain που έλαβε και ενημερώνει την δομή unspentTransactionsMap με βάση τα transacations των block του blockchain που επέλεξε (έχουμε φροντίσει μέσω boolean μεταβλητών να μην γίνεται ο miner να τραβάει transactions από το transactionPool για όσο χρόνο εκτελεί την διαδικάσια επίλυσης συγκρούσεων).

Επικύρωση blockchain

Η διαδικασία αυτή αποτελείται από την επικύρωση κάθε block του εν λόγω blockchain.

Διαδικασία προσαρμογής – Consensus

Κάθε φορά ποιος κάποιος miner λαμβάνει ένα block (προφανώς το winning block κάποιου round) εκτελεί μια αλληλουχία ενεργειών, οι οποίες ως σκοπό έχουν να υπάρχει συνέπεια στο σύστημα. Αυτό σημαίνει ότι όλοι οι κόμβοι του δικτύου θα πρέπει να έχουν την ίδια εικόνα για την κατάσταση του συστήματος. Η λειτουργικότητα αυτή μπορεί να εξηγηθεί καλύτερα μέσω ενός παραδείγματος. Έστω ότι έχουμε δύο miners M1 και M2. Αυτοί τραβάνε transactions από το transaction pool και τα τοποθετούν στα block τους, έστω B1 και B2 αντίστοιχα. Ας υποθέσουμε ότι ο M1 γεμίζει το block του και λύνει το puzzle proof-of-work πριν από τον M2. Συνεπώς, ο M1 θα στείλει το B1 (winning block) στον M2, ο οποίος θα σταματήσει την διαδικασία mining με το που το λάβει. Έστω TRANSACTIONS1 τα transactions που περιέχονται στο B1 και TRANSACTIONS2 τα transactions που περιέχονται στο B2. Προφανώς, τα δύο αυτά σύνολα είναι διάφορα μεταξύ τους με πολύ μεγάλη πιθανότητα. Στην συνέχεια περιγράφεται η διαδικασία που πρέπει να ακολουθήσουμε (ως M2), έτσι ώστε οι M1 και M2 να έχουν την ίδια εικόνα για την κατάσταση του συστήματος. Το πρώτο πράγμα που κάνουμε είναι να σχηματίσουμε τα σύνολα T1 και T2, όπου T1 είναι τα transactions που ανήκουν στο σύνολο TRANSACTIONS1 και δεν ανήκουν στο σύνολο TRANSACTIONS2, ενώ T2 είναι τα transactions που ανήκουν στο σύνολο TRANSACTIONS2 και δεν ανήκουν στο σύνολο TRANSACTIONS1. Μετά, ορίζουμε το σύνολο T1_INPUTS, το οποίο περιέχει όλα τα transactionInputs του συνόλου T1. Με βάση το T1_INPUTS ορίζουμε τα σύνολα :

CLEAN_TRANSACTIONS,
DIRTY_TRANSACTIONS και
TO_BE_USED_AGAIN_TRANSACTIONS_INPUTS

ως εξής:

Το CLEAN_TRANSACTIONS περιέχει όλα τα transactions εκείνα του συνόλου T2 τα οποία δεν είχαν ούτε ένα transactionInput το οποίο ανήκει στο σύνολο T1_INPUTS. Το DIRTY_TRANSACTIONS περιέχει όλα τα transactions εκείνα του συνόλου T2 τα οποία είχαν τουλάχιστον ένα transactionInput το οποίο ανήκει στο σύνολο T1_INPUTS. Το TO_BE_USED_AGAIN_TRANSACTIONS_INPUTS περιέχει όλα τα transactionInputs που προέρχονται από τα transactions του συνόλου T2, τα οποία δεν ανήκουν στο σύνολο T1_INPUTS (προφανώς τα εν λόγω transaction inputs μπορεί να προέρχονται και από dirtyTransaction). Στην συνέχεια, προσθέτουμε στην δομή unspentTransactionMap όλα τα transactionInputs του συνόλου TO_BE_USED_AGAIN_TRANSACTIONS_INPUTS, δηλαδή όλα τα transactionInputs εκείνα τα οποία δεν έχουν ξοδευτεί ουσιαστικά και μπορούν να χρησιμοποιηθούν εκ νέου. Αυτό επιτυγχάνεται με τη βοήθεια της δομής tempUsedOutputs στην οποία φροντίζουμε να κρατάμε προσωρινά (για κάθε γύρο) τα transactionInputs τα οποία αφαιρούμε από το unspentTransactionMap. Μετά, αφαιρούμε από το unspentTransactionMap όλα τα transactionInputs του συνόλου T1_INPUTS τα οποία υπάρχουν σε αυτό (μπορεί προφανώς κάποιο transactionInput που ανήκει στο T1_INPUTS να μην ανήκει στο unspentTransactionMap). Στην συνέχεια, για κάθε transaction του συνόλου T1, προσθέτουμε τα transactionOutputs αυτού στο unspentTransactionMap. Ακολούθως, για κάθε transaction το οποίο ανήκει στο σύνολο DIRTY_TRANSACTIONS φροντίζουμε να αφαιρέσουμε τα transactionOutputs που αυτό είχε παράγει από το unspentTransactionMap. Παράλληλα ελέγχουμε αν μπορούμε να ξαναδημιουργήσουμε κάποιο dirtyTransaction με νέα transactionInputs. Αν γίνεται το δημιουργούμε και το τοποθετούμε στο σύνολο CLEAN_TRANSACTIONS. Σε αυτό το σημείο αξίζει να αναφέρουμε ότι κάθε transaction περιέχει έναν counter, ο οποίος είναι αρχικοποιημένος στο 3, που υποδηλώνει πόσες φορές ένα transaction μπορεί, αφού έχει δημιουργηθεί, να επανατοποθετηθεί στο transactionPool. Κάθε φορά που τραβάμε ένα transaction από το transactionPool ελέγχουμε αν αυτός ο δείκτης είναι μηδέν. Αν είναι τότε πετάμε το transaction, αλλίως συνεχίζουμε κανονικά. Έτσι, δεν υπάρχει περίπτωση transactions τα οποία για οποιοδήποτε λόγο (πιθανή απόπειρα double spending πχ) είναι invalid, να κυκλοφορούν στο δίκτυο μας για πάντα. Τέλος, με βάση τα όσα ειπώθηκαν παραπάνω, η τελευταία ενέργεια με την οποία εξασφαλίζεται το consensus κομμάτι του συστήματος είναι να προσθέσουμε στην αρχή του transactionPool

τα transactions του συνόλου CLEAN_TRANSACTIONS (το ότι τα βάζουμε στην αρχή του transactionPool σημαίνει ότι τους δίνουμε προτεραιότητα, αφού θα είναι

τα πρώτα που θα τραβηχτούν) , αφού για κάθε ένα από αυτά έχουμε μειώσει κατά ένα τον δείκτη που αναφέρθηκε παραπάνω.

Συνοπτική Παρουσίαση Κώδικα

Στην συνέχεια θα παρουσιαστούν πολύ συνοπτικά οι κλάσεις και τα αρχεία που χρησιμοποιήσαμε για την υλοποίηση μας.

- **Κλάση Wallet**
  Ένα αντικείμενο αυτής της κλάσης ορίζει ένα πορτοφόλι. Παράγεται ένα ζεύγος κλειδιών RSA από το οποίο το **public key** ορίζει την διεύθυνση (**address**) του πορτοφολιού, ενώ το **private key** είναι γνωστό μόνο στον κάτοχο του πορτοφολιού και χρησιμοποιείται για τις ψηφιακές υπογραφές.

Κλάση TransactionOutput

Ένα αντικείμενο αυτής της κλάσης ορίζει ένα transactionOutput . Οι πληροφορίες που περιέχονται σε αυτό είναι η διεύθυνση στην οποία αναφέρεται (**address**), το πόσο (**amount**) που σχετίζεται με αυτό, το id του transaction από το οποίο προήλθε (**transactionId**) καθώς και ένα μοναδικό αναγνωριστικό αυτού (**id**).

Κλάση Transaction

Ένα αντικείμενο αυτής της κλάσης ορίζει ένα transaction. Οι πληροφορίες που περιέχονται σε αυτό είναι η διεύθυνση του αποστολέα (**senderAddress**), η διεύθυνση του παραλήπτη (**receiverAddress**), το ποσό της δοσοληψίας (**amount**), τα **transactionInputs** (πρόκειται για λίστα από transactionOutput.id, όπου transactionOutput αντικείμενο της κλάσης TransactionOutput) του transaction, τα **transactionOutputs** (πρόκειται για λίστα από transactionOutput, όπου transactionOutput αντικείμενο της κλάσης TransactionOutput) του transaction, την υπογραφή του αποστολέα (**signature**), τον counter που αναφέρθηκε στην λειτουργικότητα 10 (**remainingTries**) καθώς και ένα μοναδικό αναγνωριστικό του transaction (**transactionId**). Θα πρέπει να αναφέρουμε ότι το signature προκύπτει με χρήση της μεθόδου **signTransaction(privateKey)**, ενώ το transactionId με χρήση της μεθόδου **hash()**.

Κλάση Block

Ένα αντικείμενο αυτής της κλάσης ορίζει ένα block. Οι πληροφορίες που περιέχονται σε αυτό είναι το hash του προηγούμενου block (**previousHash**), ένα **timestamp** που σηματοδοτεί τη στιγμή δημιουργίας του, το hash του block (**currentHash**), το **nonce** που χρησιμοποιήθηκε για το proof-of-work puzzle, έναν **index** που υποδηλώνει την θέση του στο blockchain, καθώς και μια λίστα από αντικείμενα της κλάσης Transaction που αυτό περιέχει (**listOfTransactions**). Η μέθοδος **hash()** υπολογίζει το digest του block, ενώ η μέθοδος **pow()** χρησιμοποιείται για να ελέγξουμε αν το digest αυτό ξεκινάει με DIFFICULTY μηδενικά.

Κλάση Blockchain

Ένα αντικείμενο αυτής της κλάσης ορίζει ένα blockchain. Αυτό ουσιαστικά είναι μια λίστα από αντικείμενα τη κλάσης Block (**blocks**).

Κλάση Node

Ένα αντικείμενο αυτής της κλάσης ορίζει ένα κόμβο του δικτύου. Τα **πεδία** αυτής της κλάσης είναι τα εξής:

firstTime, lastTime : χρησιμοποιούνται για την εξαγωγή στατιστικών σχετικά με τη ρυθμαπόδοση του συστήματος.

blockTimes : μια λίστα που κρατάει για κάθε block που προστίθεται στο blockchain τον χρόνο που μεσολάβησε από την δημιουργία του μέχρι να προστεθεί σε αυτό.

legitTransactions : μια λίστα που περιέχει τα transactionId όλων των transactions που έχουν προστεθεί στο blockchain.

tempUsedOutputs : η δομή όπως έχει περιγραφεί παραπάνω

isMining, winningBlockFound, existsConflict και isStillAdjusting: boolean μεταβλητές που δείχνουν αν ο miner κάνει mines, αν έχει βρεθεί το winning block του round, αν υπάρχει σύγκρουση με το ληφθέν block και αν ο miner εκτελεί ακόμη την διαδικασία προσαρμογής, αντίστοιχα.
transactionPool : η δομή όπως έχει περιγραφεί παραπάνω.

myIp και myPort : η διεύθυνση ip και η θύρα που ακούει ο miner.

chain : Το blockchain, όπως ο miner το έχει σχηματίσει (αντικείμενο της κλάσης Blockchain).

wallet : το πορτοφόλι (αντικείμενο της κλάσης Wallet) του miner (ο οποίος λειτουργεί και ως client).

nodeId : το id του κόμβου (το έλαβε από τον Bootstrap).

workingBlock : το current block του miner (αντικείμενο της κλάσης Block).

ring : η δομή όπως έχει περιγραφεί παραπάνω.

unspentTransactionMap : η δομή όπως έχει περιγραφεί παραπάνω.

Οι βασικές **μέθοδοι** της κλάσης είναι οι εξής:

unicast(address, body) : αποστολή unicast μηνύματος με περιεχόμενο body στην διεύθυνση address.

broadcast(address, body) : αποστολή broadcast μηνύματος σε όλους τους κόμβους του ring.

registerNodeToRing(ip, port) : εγγραφή του κόμβου στο δίκτυο (μέσω του Bootstrap) και λήψη αναγνωριστικού nodeId.

walletBalance(address) : επιστρέφει το υπολειπόμενο πόσο του πορτοφολιού με διεύθυνση address.

myWalletBalance() : επιστρέφει το δικό μου υπολειπόμενο ποσό

createTransaction(senderAddress, amount, receiverAddress) : Δημιουργία ενός transaction (αντικείμενου της κλάσης Transaction) με βάση τα στοιχεία που δίνονται εφόσον αυτό μπορεί να δημιουργηθεί (δεν παραβιάζεται κάποιος περιορισμός).

verifySignature(transaction) : έλεγχος της υπογραφής του transaction.

validateTransaction(transaction) : επικύρωση του transaction.

broadcastTransaction(transaction) : αποστολή του transaction μέσω broadcast σε όλους του άλλους κόμβους.

transact(senderAddress, amount, receiverAddress) : κλήση των μεθόδων createTransaction, verifySignature και broadcastTransaction, και τοποθέτηση του παραγόμενου αντικείμενου της κλάσης Transaction στο transactionPool.

receive(jsonTransaction): λήψη ενός transaction και τοποθέτηση αυτού στο transactionPool.

pullTransactionFromPool() : μέθοδος-δαίμονας που τραβάει transactions από το transactionPool. Με κλήση της validate ελέγχει αν το transaction είναι valid. Αν είναι τοποθετείται στο block. Αν γεμίσει το block καλεί την μέθοδο mineBlock() (βλέπε παρακάτω).

validateBlock(block) : επικύρωση του block.

mineBlock() : διαδικασία mining του working block.

receiveBlock() : λήψη ένος block.

resolveConflict() : διαδικασία επίλυσης συγκρούσεων.

validateChain(chain) : επικύρωση του chain.

adjust(): διαδικασία προσαρμογής – consensus.

- **Κλάση BootstrapNode**

Πρόκειται για κλάση που κληρονομεί από την κλάση Node. Ένα αντικείμενο αυτής ορίζει τον Bootstrap κόμβο του συστήματος. Περιέχει τα επιπλέον **πεδία numOfNodes** (το πλήθος των κόμβων του δικτύου), **idToGive** (autoIncrement δείκτης που χρησιμοποιείται για την αποστολή αναγνωριστικού nodeId σε όποιον κόμβο κάνει εγγραφή) και μια **acks** (δομή που κρατάει τους κόμβους που έκαναν εγγραφή και από τους οποίους λάβαμε acknowledgement). Επιπλέον, περιέχει και κάποιες **μεθόδους** που σχετίζονται με την αρχικοποίηση του συστήματος:

createGenesisTransaction(): δημιουργία του transaction με το οποίο ο Bootstrap Node αποκτάει κεφάλαιο.
createGenesisBlock(): δημιουργία του genesis block και τοποθέτηση του παραπάνω transaction σε αυτό.
createInitialTransactions(): διαμοιρασμός του διαθέσιμου υπολοίπου του Bootstrap σε όλους τους υπόλοιπους κόμβος με ισάξιο τρόπο.

registerNodeToRing(): μέθοδος που έχει γίνει override και χρησιμοποιείται με σκοπό την τοποθέτηση του εγγραφόμενου κόμβου στο ring και την αποστολή σε αυτόν ενός nodeId.
broadcastInfo(): αποστολή του blockchain και του ring σε όλους τους κόμβους του δικτύου.

- **Αρχείο rest.py**

  Μέσω αυτού του κώδικα σηκώνουμε έναν server flask (miner). Αξίζει να αναφέρουμε τα paths που χρησιμοποιούνται:

  /bootstrap/register: εγγραφή κάποιου κόμβου στο σύστημα (μόνο ο Bootstrap ακούει εδώ).
  /bootstrap/registerAck: αποστολή acknowledgement στον Bootstrap (μετά την εγγραφή).
  /bootstrap/info: αποστολή του ring και του blockchain (όλοι ο κόμβοι εκτός του Bootstrap ακούνε εδώ).
  /transactions/broadcast: λήψη κάποιου transaction που έγινε broadcast.
  /transactions/get: δημιουργία κάποιου transaction ως client.
  /block/broadcast: λήψη κάποιου block που έγινε broadcast.
  /chain/conflict/length: request για το μήκος της blockchain του miner.
  /chain/conflict/blockchain: request για το blockchain του miner.
  /ring: request για το ring του miner.
  /wallet: request για το υπόλοιπο μου.
  /lastBlock: request για το το τελευταίο block του blockchain του miner.

- **CLI**

  Κληθήκαμε να πραγματοποιήσουμε ένα cli για την εισαγωγή κάποιων εντολών που θα προσφέρουν συγκεκριμένες δυνατότητες στον client. Δημιουργήσαμε ενα python shell. Κατά την εκκίνηση, το πρόγραμμα διαβάζει την ip του client και ζητείται το port επικοινωνίας από το πληκτρολόγιο. Η πληροφορία αυτή μας βοηθάει στο να γνωρίζει το cli “ποιός client είναι”. Έτσι π.χ. θα ξέρουμε όταν γίνεται μια μεταφορά ποσού, από ποιόν είναι. Οι εντολές που εισάγαμε είναι οι ζητούμενες, δηλαδή **t <address> amount, view, balance, help,** αλλά προσθέσαμε και κάποιες που πιστεύαμε πως είναι αρκετά βοηθητικές. Αυτές είναι οι **ring** και **exit.**

  Για την υλοποίηση του cli χρησιμοποιήσαμε τη βιβλιοθήκη cmd της python. Με ένα mainloop το shell μας λειτουργεί μέχρι να δοθεί συγκεκριμένη εντολή τερματισμού ή “ctrl + c” (interrupt).

Υλοποίηση της εντολής "t"

Αφού ελέγξουμε πως τα ορίσματα είναι σωστά σε αριθμό, δημιουργούμε ένα μήνυμα τύπου json το οποίο περιέχει τη receiver_address και το amount, και αποστέλνεται στο url : ~/transactions/get του αρχείου restpy. Εκεί καλείται η create_transaction() μέσα στην οποία πλέον "προσπαθεί" να δημιουργήσει το transaction. Είτε ολοκληρωθεί σωστά είτε όχι θα λάβουμε μήνυμα για το τι έγινε ακριβώς(Ολοκλήρωση συναλλαγής, λάθος address, χαμηλό υπόλοιπο).

Υλοποίηση της εντολής "view"

Αντιστοίχως εδώ λαμβάνουμε ένα json response από το url : ~/lastBlock. Αυτό περιέχει όλα τα transactions τα οποία έχει το τελευταίο block της αλυσίδας. Τα τυπώνουμε για μια πιο όμορφη εμφάνιση.

Υλοποίηση της εντολής "balance"

Αντιστοίχως εδώ λαμβάνουμε ένα json response από το url : ~/wallet. Με μία πολύ απλή υλοποίηση απλά λαμβάνουμε και τυπώνουμε το υπόλοιπο του client.

Υλοποίηση της εντολής "help"

Υπήρχε σαν λειτουργία ήδη μέσα στη συγκεκριμένη βιβλιοθήκη της python, απλά χρειάστηκε να δημιουργήσουμε το documentation της κάθε εντολής.

Υλοποίηση της εντολής "ring"

Σκεφτήκαμε πως καλό θα ήταν ο κάθε client να γνωρίζει τις διευθύνσεις των άλλων clients ούτως ώστε να διευκολύνεται στις συναλλαγές του. Έτσι δημιουργήσαμε την εντολή "ring", η οποία λαμβάνει ένα json response από το url : ~/ring. Αυτό περιέχει όλες τις διευθύνσεις, εκτός του client φυσικά, και με κατάλληλο τρόπο τις τυπώνουμε όπως και σε ποιό node ανήκει η κάθε μία.

Υλοποίηση της εντολής "exit"

Μια απλή εντολή εξόδου αντί για την χρήση interrupt(ctrl+c)

Πειράματα

Αφού στήσαμε το δίκτυο, τρέξαμε κάποιες προσομοιώσεις με βάση τα αρχεία που μας είχαν δοθεί προκείμενου να εκτιμήσουμε:

1. **Την απόδοση του συστήματος:** Αυτό περιλαμβάνει την **ρυθμαπόδοση (throughput)** του συστήματος, δηλαδή το ρυθμό με τον οποίο εξυπηρετούνται τα transactions (μετρήθηκε σε second), και τον **μέσο χρόνο block time,** δηλαδή τον μέσο χρόνο που απαιτείται για να προστεθεί ένα νέο block στο blockchain (μετρήθηκε σε second).
1. **Την κλιμακωσιμότητα του συστήματος:** Σύγκριση της συμπεριφοράς του συστήματος για πλήθος κόμβων ίσο με 5 και πλήθος κόμβων ίσο με 10.

Η διαδικασία που ακολουθήσαμε ήταν η εξής: Για τις διάφορες τιμές των παραμέτρων DIFFICULTY και CAPACITY, όπως αυτές ορίζονται από την εκφώνηση, τρέξαμε την προσομοίωση. Κατά την εκτέλεση της κάθε μίας προσομοίωσης κρατάγαμε για κάθε miner την στιγμή που λαμβάνει το πρώτο transaction και την στιγμή που τοποθετεί το τελευταίο transaction στο block. Από την διαφορά αυτών των χρονικών στιγμών μπορούμε να υπολογίσουμε πόσα transactions εξυπηρετούνται ανά δευτερόλεπτο (για κάθε miner ξεχωριστά). Επίσης κρατάγαμε για κάθε block τον χρόνο που χρειάζεται για να γίνει το mine, και στην παίρνοντας την μέση τιμή αυτών των χρόνων υπολογίσαμε τον μέσο χρόνο block time (για κάθε miner ξεχωριστά). Έτσι, στο τέλος μιας προσομοίωσης ο κάθε miner δίνει δύο τιμές σχετικά τη ρυθμαπόδοση και τον μέσο χρόνο block time. Παίρνοντας την μέση τιμή αυτών των μετρικών από τις τιμές που έβγαλα όλοι οι miners, υπολογίσαμε την απόδοση του συστήματος για τις συγκεκριμένες παραμέτρους. Στην συνέχεια παρουσιάζονται τα αποτελέσματα που λάβαμε για τις διάφορες προσομοιώσεις.

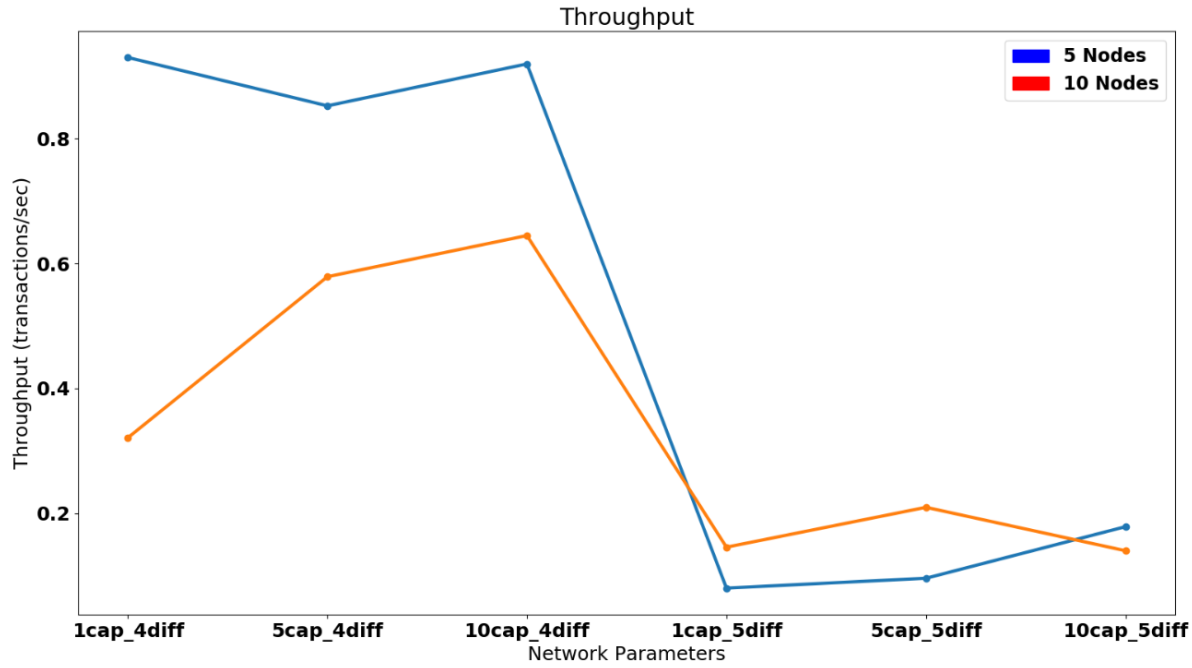
Αποτελέσματα για δίκτυο 5 κόμβων

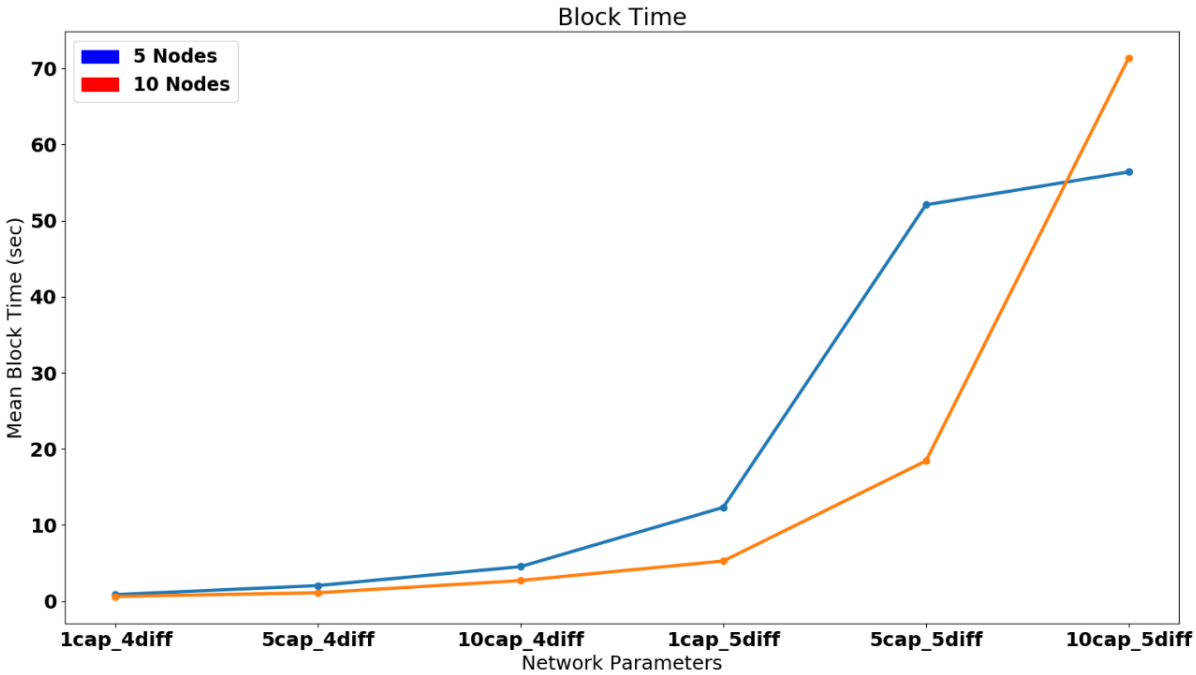
	Throughput (transactions/second)	**Mean Block Time (second)**
CAPACITY 1, DIFFICULTY 4	0.92953302035145961	0.8562253806707563
CAPACITY 5, DIFFICULTY 4	0.852200166886788	2.0514802794967935
CAPACITY 10, DIFFICULTY 4	0.9192254894957242	4.542168665381685
CAPACITY 1, DIFFICULTY 5	0.08050513911178381	12.325318466473089
CAPACITY 5, DIFFICULTY 5	0.0962367786334258	52.080340476399286
CAPACITY 10, DIFFICULTY 5	0.17876745760274296	56.39374468859751

Αποτελέσματα για δίκτυο 10 κόμβων

	Throughput (transactions/second)	**Mean Block Time (second)**
CAPACITY 1, DIFFICULTY 4	0.321062966341539	0.6143374097702494
CAPACITY 5, DIFFICULTY 4	0.5787478896310084	1.1051228825118717
CAPACITY 10, DIFFICULTY 4	0.6445918517979635	2.7091309197829103
CAPACITY 1, DIFFICULTY 5	0.14597613519440078	5.2773895398018436
CAPACITY 5, DIFFICULTY 5	0.20975069048191947	18.431706782061763
CAPACITY 10, DIFFICULTY 5	0.1402746072562804	71.34078271038939

Στην συνέχεια παρουσιάζονται τα γραφήματα στα οποία απεικονίζονται τα αποτελέσματά μας.


Throughput
5 Nodes
10 Nodes
Throughput (transactions/sec)
0.8
0.6
0.4
0.2
1cap_4diff
5cap_4diff
10cap_4diff
1cap_5diff
5cap_5diff
10cap_5diff
Network Parameters


Block Time
5 Nodes
10 Nodes
Mean Block Time (sec)
70
60
50
40
30
20
10
0
1cap_4diff
5cap_4diff
10cap_4diff
1cap_5diff
5cap_5diff
10cap_5diff
Network Parameters

Σχολιασμός Αποτελεσμάτων

Ρυθμαπόδοση: Παρατηρούμε ότι η ρυθμαπόδοση του δικτύου 5 κόμβων είναι, σε γενικές γραμμές, υψηλότερη σε σχέση με αυτή του δικτύου 10 κόμβων. Αυτό είναι πλήρως λογικό καθώς στην περίπτωση που έχουμε 10 κόμβους το σύστημα μας καλείται να εξυπηρετήσει μεγαλύτερη πληθώρα δοσοληψιών. Επιπλέον, παρατηρούμε ότι η παράμετρος CAPACITY του συστήματος επηρεάζει την ρυθμαπόδοση σε πολύ μικρό βαθμό, σχεδόν αμελητέο. Τέλος, βλέπουμε ότι και στις δύο περιπτώσεις η αλλαγή της παραμέτρου DIFFICULTY επηρεάζει άμεσα το ρυθμό με τον οποίο το σύστημα εξυπηρετεί τις δοσοληψίες. Πιο συγκεκριμένα όσο μεγαλύτερο είναι το DIFFICULTY τόσο χαμηλότερη είναι η ρυθμαπόδοση. Αυτό συμβαίνει διότι υλοποιήσαμε το σύστημα με τέτοιο τρόπο ώστε όταν ένας miner κάνει mine να μην εξυπηρετεί δοσοληψίες. Έτσι, όσο αυξάνει το DIFFICULTY αυξάνει και ο μέσος χρόνος mining, με αποτέλεσμα να αυξάνονται τα διαστήματα στα οποία οι miners δεν εξυπηρετούν δοσοληψίες, γεγονός που οδηγεί σε χαμηλότερη ρυθμαπόδοση.

Μέσος χρόνος Block Time: Παρατηρούμε ότι ο μέσος χρόνος Block Time του δικτύου 10 κόμβων είναι, σε γενικές γραμμές, χαμηλότερος σε σχέση με αυτόν του δικτύου 5 κόμβων. Αυτό είναι πλήρως λογικό καθώς στην περίπτωση που έχουμε 10 κόμβους, έχουμε ουσιαστικά τον διπλάσιο αριθμό από miners να ψάχνει την λύση για το puzzle proof-of-work. Συνεπώς, αυξάνονται δραματικά οι πιθανότητες να βρεθεί πιο γρήγορα. Επιπλέον, παρατηρούμε ότι για σταθερή τιμή της παραμέτρου DIFFICULTY όσο αυξάνει η παράμετρος CAPACITY τόσο αυξάνει και ο μέσος χρόνος Block. Τέλος, βλέπουμε ότι και στις δύο περιπτώσεις η αλλαγή της παραμέτρου DIFFICULTY επηρεάζει άμεσα τον μέσο χρόνο Block Time. Πιο συγκεκριμένα όσο μεγαλύτερο είναι το DIFFICULTY τόσο αυξάνει ο χρόνος. Αυτό συμβαίνει διότι μεγαλύτερη τιμής της παραμέτρου DIFFICULTY συνεπάγεται υψηλότερη δυσκολία εύρεσης nonce τέτοιου ώστε το το hash του block να ξεκινάει με DIFFICULTY μηδενικά, γεγονός που οδηγεί σε μεγαλύτερο χρόνο mining.