BlockChat - Project Κατανεμημένων Συστημάτων

Πέτρος Αγγελάτος
01308133

Απόστολος Τζίνας
03119109

7 Απριλίου 2024

Abstract

Σε αυτή την αναφορά παρουσιάζουμε τον σχεδιασμό και τα αποτελέσματα επίδοσης της υλοποίησης του Blockchat. Το Blockchat είναι μία πλατφόρμα χρηματικών συναλλαγών και ανταλλαγής μηνυμάτων βασισμένη στη τεχνολογία των proof-of-stake blockchains. Αρχικά περιγράφουμε την γενική αρχιτεκτονική του συστήματος και τις τεχνολογίες που χρησιμοποιήθηκαν για την υλοποίηση. Έπειτα, παρουσιάζουμε την πειραματική μεθοδολογία με την οποία μετράμε την διεκπεραιωτική ικανότητα (throughput) του δικτύου. Τέλος, εξετάζουμε την κλιμακωσιμότητα και την δικαιοσύνη του συστήματος.

1 Σχεδιασμός

Ο Blockchat κόμβος είναι το κύριο μέρος της εφαρμογής. Διατηρεί το blockchain επικοινωνώντας με τους υπόλοιπους κόμβους του δικτύου. Συμπληρωματικό μέρος της εφαρμογής αποτελεί το Blockchat CLI, με το οποίο ο χρήστης μπορεί να αλληλεπιδράσει με έναν Blockchat κόμβο στέλνοντας συναλλαγές και ελέγχοντας τη κατάσταση του blockchain. Το Blockchat CLI συνδέεται στο REST API που διαθέτει ο Blockchat κόμβος. Τα επιμέρους components της εφαρμογής περιγράφονται παρακάτω.

Για την υλοποίηση της εφαρμογής επιλέξαμε την γλώσσα προγραμματισμού Rust που συνδυάζει πολύ καλό performance σε συνδυασμό με memory safety και ένα πολύ εκφραστικό σύστημα τύπων.

1.1 Πορτοφόλια

Στο BlockChat κάθε κόμβος διατειρεί ένα πορτοφόλι από το οποίο μπορεί να στείλει και να λάβει BCC, να στείλει μηνύματα σε άλλους κόμβους, αλλά και να δεσμεύσει κάποιο ποσό για staking. Η δημιουργία ενός πορτοφολιού γίνεται με την δημιουργία ενός τυχαίου RSA public/private key.

Κάθε πορτοφόλι αντιστοιχεί σε μία διεύθυνση η οποία υπολογίζεται ντετερμινιστικά ως το sha256(public_key). Χρησιμοποιώντας το hash του public key αντί για το public key απευθείας επιτυγχάνουμε σημαντικά μικρότερο μήκος διεύθυνσης, κάτι που είναι σημαντικό για την ευχρηστία του συστήματος. Blockchains όπως το bitcoin ακολουθούν παρόμοια τακτική.

Για την αναπαράσταση μίας διεύθυνσης πορτοφολιού χρησιμοποιούμε το base62 encoding. Η επιλογή του base62 έγινε για να έχουμε σύντομη αναπαράσταση μίας διεύθυνσης χωρίς όμως να χρειάζονται ειδικοί χαρακτήρες όπως στο base64. Ένα παράδειγμα διεύθυνσης φαίνεται στο Listing 1.

```
Pmrd9tHIGKv9WZ3cgW3faJR6LTylhqy0uUXFiHG1XXw1
```

Listing 1: Ενδεικτική διεύθυνση

1.2 Block minting

Κάθε instantiation του δικτύου BlockChat πρέπει να συμφωνήσει σε ένα κοινό block capacity το οποίο ορίζει τον μέγιστο αριθμό από συναλλαγές που μπορεί να υπάρχουν μέσα σε ένα block. Στα αρχικά στάδια της υλοποίησης παρατηρήσαμε ότι όταν τα blocks γίνονται mint μόνο όταν ο επόμενος validator έχει `CAPACITY` συναλλαγές υπάρχει πρόβλημα liveness. Αν το δίκτυο είναι αδρανές και ένας χρήστης κάνει μία συναλλαγή θα θέλαμε μετά από κάποιο χρόνο αυτή να μπει στο blockchain ανεξάρτητα με το αν θα υπάρχουν `CAPACITY - 1` ακόμα συναλλαγές. Για αυτόν τον λόγο ο κάθε validator φτιάχνει καινούριο block είτε όταν έχει `CAPACITY` συναλλαγές είτε όταν παρέλθει κάποιος συγκεκριμένος χρόνος. Στην υλοποίησή μας διαλέξαμε το ένα δευτερόλεπτο ως τον μέγιστο χρόνο αναμονής πριν παραχθεί ένα block.

1.3 Αρχιτεκτονική κόμβου

Για την υλοποίηση του κόμβου χωρίσαμε την λειτουργικότητα σε δύο ανεξάρτητα μέρη. Το πρώτο μέρος αφορά το state machine του κόμβου με όλους τους κανόνες που χρειάζεται να ακολουθεί και το δεύτερο μέρος αφορά όλες τις λεπτομέρειες του δικτύου. Η επιλογή αυτή μας επέτρεψε να αναπτύξουμε και να τεστάρουμε τα δύο μέρη ξεχωριστά.

1.3.1 Δίκτυο

Το interface του δικτύου που φαίνεται στο Listing 2 περιγράφει ένα broadcast δίκτυο όπου η συνάρτηση `recv` παραδίδει στην εφαρμογή ένα μήνυμα που προήλθε από κάποιον άλλο κόμβο αν αυτό υπάρχει και συνάρτηση `send` στέλνει ένα μήνυμα σε όλους τους κόμβους. Το interface αφήνει την εκάστοτε υλοποίηση να αποφασίσει με ποιο τρόπο θα κάνει serialize τα μηνύματα και με ποιο τρόπο θα τα παραδώσει στα υπόλοιπα nodes. Αυτό δίνει τη δυνατότητα να υλοποιηθούν πιο εξελιγμένοι τρόποι broadcast (πχ gossip networking) χωρίς καμία αλλαγή στην υπόλοιπη εφαρμογή.

```rust
pub trait Network<T> {
    fn await_events(&mut self, timeout: Option<Duration>);

    fn recv(&mut self) -> Option<T>;

    fn send(&mut self, msg: &T);
}
```

Listing 2: Το interface του δικτύου

Στην εφαρμογή χρησιμοποιήσαμε δύο υλοποιήσεις του interface. Η πρώτη υλοποιεί ένα εικονικό δίκτυο όπου όλα τα μηνύματα μεταφέρονται από τον αποστολέα στον παραλήπτη μέσω in-memory channels. Αυτή η υλοποίηση χρησιμοποιήθηκε για testing.

Η δεύτερη υλοποίηση του interface αφορά το broadcast δίκτυο όπως αυτό περιγράφεται στην εκφώνηση. Η σύνδεση των κόμβων μεταξύ τους γίνεται με TCP sockets και κάθε μήνυμα μεταφέρεται χρησιμοποιώντας την αναπαράσταση `bincode`.

1.3.2 State machine

Η υλοποίηση του κόμβου έγινε ως ένα state machine το οποίο μεταβαίνει από το ένα state στο άλλο με διακριτά βήματα. Κατά τη διάρκεια μίας μετάβασης ο κόμβος μπορεί να ανταλλάξει μηνύματα με τους υπόλοιπους κόμβους χρησιμοποιώντας μία τυχούσα υλοποίηση του interface του δικτύου όπως φαίνεται στο Listing 3. Η επιλογή του πόσο συχνά καλείται η συνάρτηση `step` αφήνεται στο κομμάτι κώδικα που θα συνδυάσει το state machine με κάποιο συγκεκριμένο network implementation. Η τιμή που επιστρέφει η συνάρτηση `step` είναι ένα hint για το πότε θα ήθελε να ξανακληθεί.

```
pub fn step<N: Network<Message>>(&mut self, network: &mut N) -> Option<Duration>
```

Listing 3: Το interface του δικτύου

Υλοποιώντας το state machine με αυτόν τον τρόπο καταφέραμε να έχουμε έναν πολύ ξεκάθαρο διαχωρισμό δικτύου και λογικής και μας επέτρεψε να έχουμε πολύ καλό έλεγχο του state machine σε unit tests.

1.3.3 Testing

Λόγω του αυστηρού διαχωρισμού του state machine και του πολύ εύκολου τρόπου να γράψει κανείς unit tests στη Rust καλύψαμε ένα πολύ μεγάλο μέρος της εφαρμογής. Επιπλέον, επειδή κατά τη διάρκεια των test είχαμε στη διάθεσή μας την εικονική υλοποίηση του δικτύου μπορέσαμε να προσομοιώσουμε διάφορα byzantine failures που δεν συμβαίνουν με έναν honest node.

```
#[test]
fn test_message_insufficient_funds() {
    let (mut sender_wallet, _, sender_key) = setup_test_wallet(23);
    let (receiver_wallet, _, _receiver_key) = setup_default_test_wallet();

    let message = String::from("These are 24 characters.");
    let tx = sender_wallet.create_message_tx(receiver_wallet.address.clone(),
message);
    let signed_tx = sender_key.sign(tx.clone());

    let result = sender_wallet.apply_tx(signed_tx.clone());
    assert!(matches!(result, Err(Error::InsufficientFunds)));
    assert_eq!(sender_wallet.nonce, 0);
}
```

Listing 4: Ενδεικτικό test για μη επαρκή BCC

1.3.4 Logging

Στην εφαρμογή κάναμε επίσης χρήση ενός logging framework το οποίο υποστηρίζει πολλά διαφορετικά επίπεδα σημαντικότητας, ξεκινώντας από το υψηλότερο επίπεδο `ERROR` για σφάλματα που ο administrator του node πρέπει να τα χειριστεί άμεσα έως το χαμηλότερο επίπεδο `TRACE` για λεπτομερή περιγραφή των βημάτων που κάνει η εφαρμογή.

Αυτό μας επέτρεψε να μελετήσουμε την εξέλιξη της εφαρμογής και την εύκολο αποσφαλμάτωσή της όταν κάτι δεν δούλευε όπως έπρεπε. Παραθέτουμε ένα απόκομμα των log στο Listing 5.

```
2024-04-07T17:47:07.485024Z TRACE blockchat::node: node-1: accepted valid tx
fa53ae56193134a9d48e952a1e62b3855de3099181ee19fc5f6662f1bb44854d
2024-04-07T17:47:07.485035Z  INFO blockchat::node: node-1: accepted valid block
04822c2e9f123bbdbb8d910abe54c40524eeac10a6bb55b7a737af052131b93f
```

Listing 5: Ενδεικτικά logs της εφαρμογής

1.3.5 Performance tuning

Για την βελτιστοποίηση της ταχύτητας της εφαρμογής χρησιμοποιήσαμε flamegraph traces τα οποία δουλεύουν καταγράφοντας δείγματα του stack frame του προγράμματος πολλές φορές το δευτερόλεπτο και αναλύοντας τι ποσοστό του χρόνου αφιερώνει το πρόγραμμα σε κάθε συνάρτηση. Στο Figure 1 φαίνεται η ανάλυση του step function του state machine κατά τη διάρκεια εκτέλεσης των πειραμάτων που περιγράφονται παρακάτω.

Παρατηρούμε ότι η CPU αφιερώνεται σχεδόν εξολοκλήρου στον υπολογισμό του verification function του RSA. Στα αρχικά στάδια της ανάπτυξης είχαμε επίσης παρατηρήσει ότι ένα μη αμελητέο ποσοστό του χρόνου αφιερονώταν στην παραγωγή και αναγνώριση μηνυμάτων JSON. Η παρατήρηση αυτή μας οδήγησε στην τελική μορφή της εφαρμογής όπου τα μηνύματα ανταλλάσσονται με μορφή bincode, που είναι πολύ πιο γρήγορη.

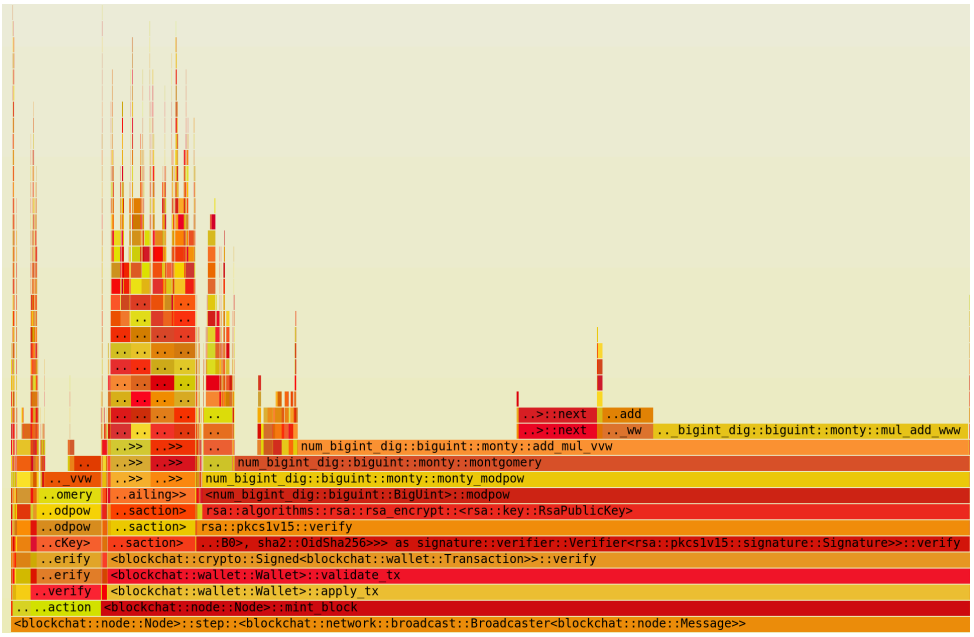

Figure 1: Ανάλυση CPU time για το step function

1.4 Command Line Interface

Το Command Line Interface προσφέρει στον χρήστη πληροφορίες για την κατάσταση του blockchain και του επιτρέπει να στείλει εντολές σε εναν blockchat κόμβο. Κάθε blockchat κόμβος διαθέτει ένα REST API το οποίο είναι προσβάσιμο μόνο από το τοπικό δίκτυο του ίδιου υπολογιστή (localhost). Αυτό το API χρησιμοποιείται από την CLI εφαρμογή για την επικοινωνία με τον κόμβο. Οι εντολές του CLI έχουν εκφραστεί ως τύποι Rust, όπως φαίνεται στο Listing 6. Αυτό επιτρέπει ευκολη επαλήθευση των δεδομένων εισόδου από τον χρήστη.

```rust
#[derive(Serialize, Deserialize)]
pub struct SetStakeRequest {
    pub amount: u64,
}

#[derive(Serialize, Deserialize)]
#[serde(untagged)]
pub enum CreateTransactionRequest {
    Coin { recipient: Address, amount: u64 },
    Message { recipient: Address, message: String },
}
```

Listing 6: Οι τύποι Rust των REST endpoint.

2 Πειραματική Μεθοδολογία

Η απόδοση αυτής της υλοποίησης του Blockchat υπολογίστηκε με μία σειρά πειραμάτων. Αρχικά, μας ενδιαφέρει η διεκπεραιωτική ικανότητα (throughput) του συστήματος υπό διαφορετικές παραμετροποιήσεις. Δηλαδή πόσες συναλλαγές μπορούν να εξυπηρετηθούν ανά την μονάδα του χρόνου. Στα πειράματά μας την υπολογίζουμε σε συναλλαγές ανά δευτερόλεπτο (tx/s). Η διεκπεραιωτική ικανότητα συνδέεται ανάλογα με το block time, δηλαδή τα blocks που παράγονται ανά την μονάδα του χρόνου (blocks/s) και την χωρητικότητα του κάθε block (block capacity). Μετρώντας την διεκπεραιωτική ικανότητα για διαφορετικούς αριθμούς κόμβων και χωρητικότητας block εξετάζουμε την κλιμακωσιμότητα (scaling) του συστήματος. Τέλος, παρατηρούμε την δικαιοσύνη του συστήματος. Δηλαδή, την αναλογία του αριθμού blocks που δημιουργούνται από έναν κόμβο σε ποσοστό με το stake του.

Τα πειράματα εκτελέστηκαν τοπικά τρέχοντας τον κάθε Blockchat κόμβο ως διαφορετική διεργασία στον ίδιο υπολογιστή. Το μηχάνημα είχε i9-10885H CPU @ 2.40GHz με 16 πυρήνες και 64GB RAM. Στην αρχή κάθε πειράματος, όλοι οι Blockchat κόμβοι συνδέονται μεταξύ τους με την βοήθεια του bootstrap κόμβου. Έπειτα, ταυτόχρονα, ο κάθε κόμβος στέλνει στο δίκτυο έναν σταθερό αριθμό συναλλαγών. Σε όλες τις εκτελέσεις πειραμάτων, ένας κόμβος στέλνει πάντα τις ίδιες συναλλαγές με την ίδια σειρά. Μόλις συμπεριληφθούν όλες οι εκκρεμείς συναλλαγές στο blockchain, η εκτέλεση του προγράμματος τερματίζεται.

Κατά την διάρκεια της εκτέλεσης, χρονομετρούμε από την στιγμή που ολοκληρώνεται το bootstrapping μέχρι την στιγμή που συμπεριλαμβάνονται όλα τα transaction στο blockchain και το πρόγραμμα τερμάτιζει. Γνωρίζουμε τον συνολικό αριθμό συναλλαγών και blocks σε μία εκτέλεση. Επομένως, μπορούμε να υπολογίσουμε την διεκπεραιωτική ικανότητα και το block time του Blockchat δικτύου (tx/s και blocks/s). Για την δικαιοσύνη του συστήματος, καταγράφουμε τον αριθμό blocks που δημιούργησε ο κάθε κόμβος.

Όλα τα πειράματα εκτελέστηκαν τρεις φορές για την κάθε παραμετροποίηση, και ως τελικό αποτέλεσμα λάβαμε τον μέσο όρο των τριών εκτελέσεων. Οι ακατέργαστες πειραματικές μετρήσεις παραθέτονται στο παράρτημα στο Section A.1.

3 Αποτελέσματα

3.1 Διεκπεραιωτική Ικανότητα και Κλιμακωσιμότητα

Στα παρακάτω διαγράμματα φαίνεται η διεκπεραιωτική ικανότητα του συστήματος για 5/10 κόμβους και για 5/10/20 χωρητικότητα block.

Στο διάγραμμα Figure 2 παρατηρούμε πως για τον ίδιο αριθμό κόμβων, όταν αυξάνεται η χωρητικότητα των blocks, η διεκπεραιωτική ικανότητα (tx/sec) του Blockchat δικτύου επίσης αυξάνεται. Αυτό φαίνεται και στο διάγραμμα Figure 3, αφού όταν διπλασιάζεται η χωρητικότητα των blocks, το block time (blocks/sec) μειώνεται λιγότερο από 50%, επομένως αυξάνεται το throughput. Η αύξηση της διεκπεραιωτικής ικανότητας μαζί με την αύξηση της χωρητικότητας των blocks οφείλεται στον μικρότερο αριθμό blocks που υπογράφονται και επιβεβαιώνονται κατά την διάρκεια της εκτέλεσης. Ο μικρότερος αριθμός κρυπτογραφικών πράξεων κάνει αισθητή διαφορά στην απόδοση καθώς καταναλώνουν πάνω από το 70% του χρόνου εκτέλεσης.

Στο διάγραμμα Figure 2 επίσης παρατηρούμε πως για όλες τις χωρητικότητες block που μετρήθηκαν (5/10/20), έχουμε μικρή μείωση της διεκπεραιωτικής ικανότητας του συστήματος καθώς ο αριθμός κόμβων αυξάνεται. Στο διάγραμμα Figure 3 φαίνεται η μείωση του block time, το οποίο για ίδιες χωρητικότητες blocks σημαίνει και μείωση της διεκπεραιωτικής ικανότητας. Αυτή η μείωση οφείλεται στον μεγαλύτερο όγκο συναλλαγών στο δίκτυο που λόγω της πειραματικής μεθολογίας πρέπει να επαληθευτούν πριν υπολογιστούν τα πρώτα blocks, κάτι που καταναλώνει τον πολύτιμο υπολογιστικό χρόνο ενος κόμβου. Αν το δίκτυο ήταν λιγότερο συνωστισμένο, αυτός ο χρόνος υπολογισμού θα χρησιμοποιούνταν για την παραγωγή και επιβεβαίωση των blocks.

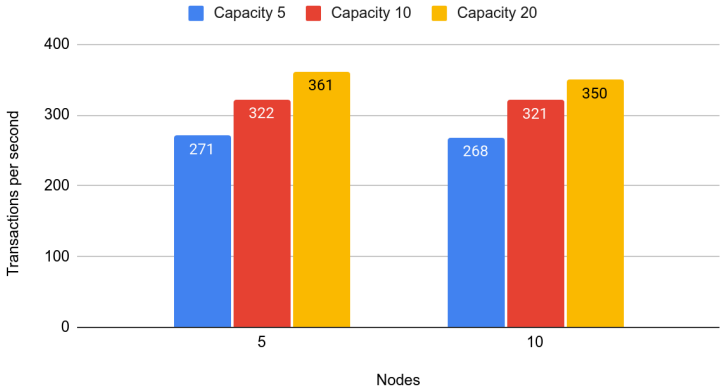

Figure 2: Ρυθμαπόδοση του συστήματος

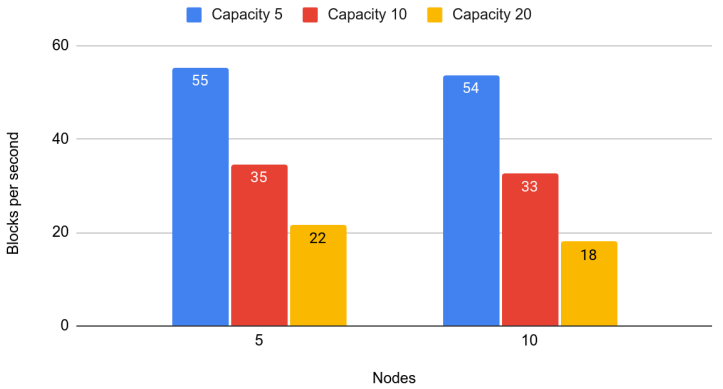

Figure 3: Block throughput

3.2 Δικαιοσύνη

Τέλος, στον πίνακα Table 1 φαίνεται η δικαιοσύνη του συστήματος. Παρατηρούμε πως το ποσοστό των blocks που δημιουργούνται από έναν κόμβο είναι ανάλογο του ποσοστού του stake του.

Node	Stake (BCC)	Stake (%)	Blocks Minted (count)	Blocks Minted (%)
1	100	71.43	36	72.00
2	10	7.14	4	8.00
3	10	7.14	3	6.00
4	10	7.14	4	8.00
5	10	7.14	3	6.00

Table 1: Δικαιοσύνη του συστήματος

APPENDIX A

A.1 Πειραματικά δεδομένα

Node Count	Block Capacity	Throughput (tx/s)	Block time (block/s)
5	5	277	57
5	5	275	56
5	5	260	53
5	10	325	35
5	10	314	34
5	10	326	35
5	20	352	21
5	20	369	22
5	20	363	22
10	5	270	54
10	5	269	54
10	5	264	53
10	10	322	33
10	10	321	33
10	10	320	32
10	20	344	18
10	20	354	18
10	20	351	18

Table 2: Πειραματικές μετρήσεις