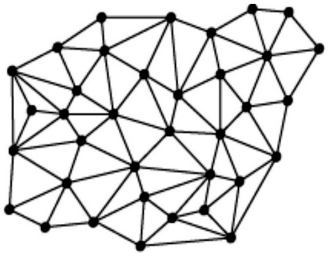


Κατανεμημένα Συστήματα
Εξαμηνιαία Εργασία

Μπέτζελος Χρήστος - 03116067
Βεκράκης Εμμανουήλ - 03116068
Κρανιάς Δημήτριος - 03116030

Εισαγωγή

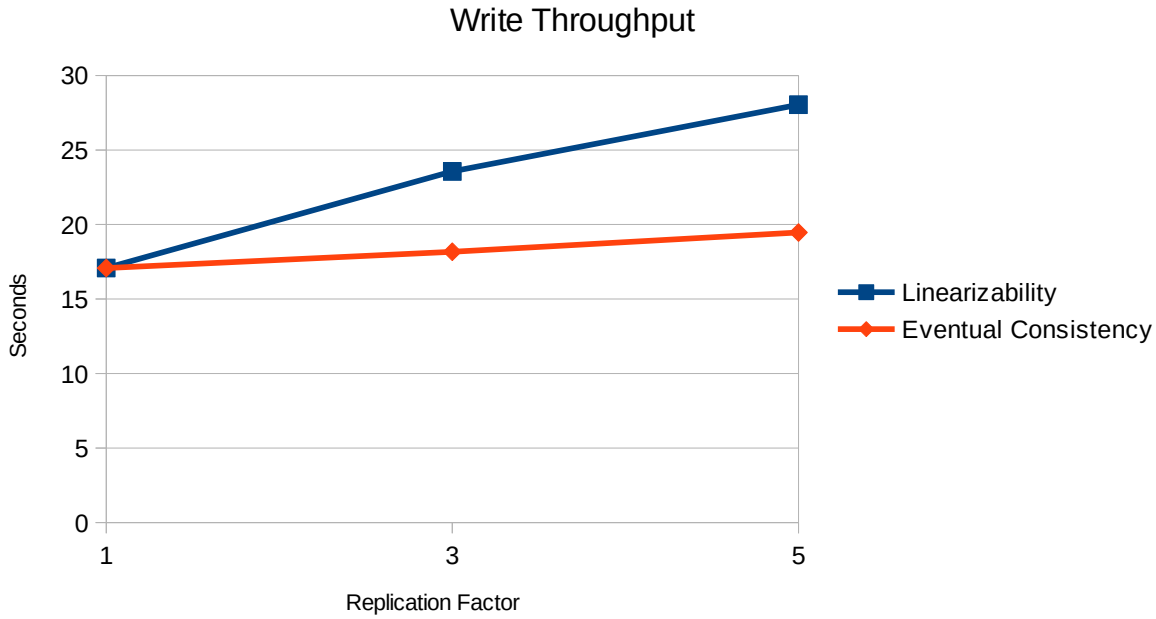
Στην παρούσα εργασία κληθήκαμε να αναπτύξουμε το ToyChord, μία απλοποιημένη έκδοση του peer-to-peer συστήματος Chord. Οι βασικές λειτουργίες που επικεντρωθήκαμε ήταν αυτές της διαχείρισης του δακτυλίου (εισαγωγές και αποχωρήσεις κόμβων), η εισαγωγή δεδομένων σε key-value μορφή και η εκτέλεση ερωτημάτων για τα δεδομένα αυτά. Δεν λάβαμε υπόψη μας περιπτώσεις μη οικειοθελούς αποχώρησης κάποιου κόμβου (πχ λόγω αποτυχίας), περιπτώσεις ταυτόχρονων join/departs σε συνδυασμό με επεξεργασία δεδομένων, καθώς επίσης δεν υλοποιήσαμε finger tables. Παρόλα αυτά, υλοποιήσαμε μία πλήρως λειτουργική εφαρμογή διαμοιρασμού αρχείων, με πολλαπλούς κατανεμημένους κόμβους DHT. Η εφαρμογή μας δοκιμάστηκε με 10 διαφορετικούς χρήστες/κόμβους, οι οποίοι "ζουν" σε 5 εικονικές μηχανές που μας παραχωρήθηκαν από την υπηρεσία του Okeanos. Η ανάπτυξη της εφαρμογής έγινε σε Python με χρήση του Flask framework για την υποστήριξη κλήσεων εντός του δικτύου των μηχανημάτων.

Πειράματα

Τελικός στόχος της ανάπτυξης της παραπάνω εφαρμογής ήταν η εκτέλεση μιας σειράς πειραμάτων, που θα οδηγήσουν στην καλύτερη κατανόηση εννοιών των κατανεμημένων συστημάτων, όπως του replication με linearizability και eventual consistency. Συγκεκριμένα, εκτελέσαμε πειράματα για τους δύο αυτούς τύπους replication για την περίπτωση που το replication factor (k) είναι 1, 3 και 5. Στην περίπτωση του k=1, πρακτικά δεν έχουμε replication, οπότε εκτελέσαμε το πείραμα μία φορά καθώς το είδος του replication δεν είχε καμία επίδραση στα αποτελέσματα.

Πείραμα 1

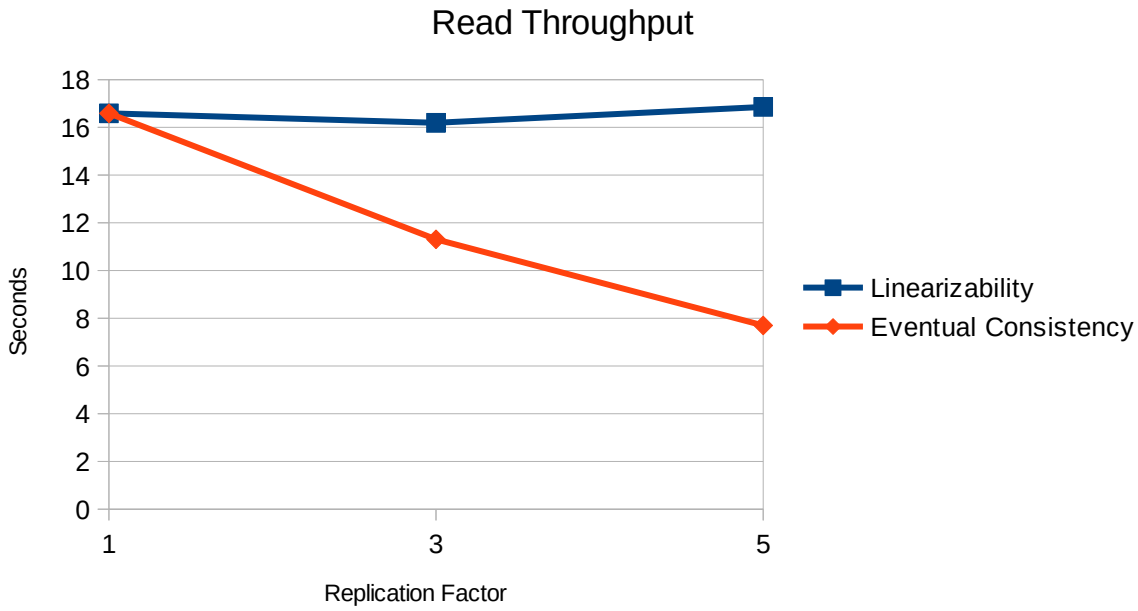
Αρχικά για καθένα συνδυασμό είδους replication και k εισάγαμε τα στοιχεία που μας δόθηκαν στο αρχείο insert.txt ξεκινώντας από τυχαίο κόμβο κάθε φορά και μετρήσαμε τον απαιτούμενο χρόνο. Τα αποτελέσματα φαίνονται στο παρακάτω διάγραμμα.



Παρατηρούμε ότι και στις δύο περιπτώσεις, το write throughput φαίνεται να αυξάνει γραμμικά με την αύξηση του k. Ωστόσο, στην περίπτωση του linearizability, η κλίση της ευθείας είναι πολύ μεγαλύτερη από την περίπτωση του eventual consistency. Αυτό είναι αναμενόμενο, καθώς στο linearizability ο κόμβος που εκτελεί το insert περιμένει μέχρι να λάβει επιβεβαίωση ότι η πληροφορία γράφτηκε και στον τελευταίο Replica Manager πριν επιστρέψει, ενώ στο eventual consistency ο πρωτεύων κόμβος επιστρέφει μήνυμα επιβεβαίωσης μόλις γράψει την πληροφορία ο ίδιος στη μνήμη του και έπειτα ενημερώνει τους επόμενους Replica Managers. Επομένως ο blocking τρόπος με τον οποίο εγγυάται το linearizability με chain replication ότι όλοι οι κόμβοι έχουν πάντα την ίδια τιμή για κάθε κλειδί είναι αυτός που οδηγεί την αντίστοιχη περίπτωση στο να καθυστερεί περισσότερο κατά την εγγραφή. Στο eventual consistency ωστόσο, παρόλο που θεωρητικά η διάδοση της πληροφορίας γίνεται παράλληλα με τις επόμενες εισόδους, στην πράξη επειδή τα threads είναι περιορισμένα, σε ορισμένες περιπτώσεις υπάρχουν καθυστερήσεις, που οδηγούν σε αυτή τη μικρή αύξηση του χρόνου καθώς αυξάνεται το k.

Πείραμα 2

Στη συνέχεια, έχοντας εισάγει τα παραπάνω δεδομένα, εκτελέσαμε για καθένα συνδυασμό replication και k μία σειρά από ερωτήματα που μας δόθηκαν στο αρχείο query.txt. Παρακάτω φαίνεται ο απαιτούμενος χρόνος για την εκτέλεση των queries για τον αντίστοιχο συνδυασμό είδους replication και replication factor.



Στο πείραμα αυτό παρατηρούμε πολύ μεγάλη διαφορά στην επίδραση της αύξησης του replication factor στο read throughput του συστήματος. Αναλυτικότερα, καθώς αυξάνεται το k, στην περίπτωση του eventual consistency ο απαιτούμενος χρόνος των ερωτημάτων μειώνεται, ενώ στο linearizability εμφανίζει μία ιδιαίτερη συμπεριφορά. Το πρώτο είναι απολύτως λογικό, καθώς στο eventual consistency, ένα query απαντάται από οποιονδήποτε κόμβο έχει αντίγραφο του κλειδιού που αναζητείται. Έτσι, στην περίπτωση που το query ξεκινήσει από οποιονδήποτε Replica Manager υπεύθυνο για το κλειδί αυτό, θα απαντηθεί αμέσως. Αν πάλι ο κόμβος από τον οποίο ξεκινάει το query δεν είναι Replica Manager, τότε αρκεί να φτάσει κυκλικά το query μέχρι τον πρωτεύοντα Replica Manager. Είναι λοιπόν προφανές, ότι όσο αυξάνεται το k, τόσο αυξάνονται οι πιθανότητες να απαντηθεί αμέσως κάποιο query και τόσο μειώνονται αντίστοιχα τα αναμενόμενα βήματα που θα διανύσει αυτό εντός του δακτυλίου.

Στην περίπτωση του linearizability με chain replication, τα queries απαντώνται από τον τελευταίο κόμβο στη σειρά, ανεξαρτήτως από τον κόμβο από τον οποίο ξεκινούν. Είναι λογικό, λοιπόν, ότι τα αναμενόμενα βήματα που θα κάνει το κάθε query να είναι ίδια ανεξάρτητα από την τιμή του k. Αυτό φαίνεται και από την ευθεία του Read Throughput, στην όποια οι μικρές διακυμάνσεις οφείλονται σε τυχαίους παράγοντες, δηλαδή στους κόμβους από τους οποίους ξεκινούν τα queries, που κάθε φορά είναι τυχαίοι.

Πείραμα 3

Τέλος, κληθήκαμε να εκτελέσουμε μία σειρά από εισαγωγές, ανανεώσεις και ερωτήματα δεδομένων σε ένα DHT με 10 κόμβους και k=3 για τις δύο περιπτώσεις replication. Σκοπός του πειράματος αυτού ήταν να διαπιστώσουμε από τα αποτελέσματα των queries ποιο είδος replication δίνει τις πιο fresh τιμές. Από την θεωρία, περιμένουμε στα αποτελέσματα για eventual consistency να υπάρξουν κάποιες stale τιμές, ενώ στο linerizability όλα τα αποτελέσματα να περιέχουν την τελευταία τιμή.

Η παρατήρηση αυτή βασίζεται σε μεγάλο βαθμό στην τύχη, ώστε ο τυχαίος κόμβος στον οποίο κάνουμε το query να είναι Replication Manager για το κλειδί που αναζητούμε και η ενημέρωση της τιμής του να μην έχει προλάβει να φτάσει λόγω μεγάλης ταχύτητας διάδοσης των αλλαγών. Για το λόγο αυτό, τις περισσότερες από τις φορές που εκτελούσαμε το πείραμα δεν παρατηρούσαμε κάποια διαφορά στα αποτελέσματα των queries και όλα είχαν τις σωστές τιμές. Για τους σκοπούς του πειράματος, λοιπόν, αποφασίσαμε να αφαιρέσουμε κάποιον από τους τυχαίους παράγοντες που αναφέραμε προηγουμένως. Ο πρώτος που έχει να κάνει με την πιθανότητα ο κόμβος στον οποίο κάνουμε το query να είναι Replica Manager του κλειδιού που αναζητούμε, μπορεί να βελτιωθεί με την αύξηση των Replica Managers. Ο δεύτερος, που έχει να κάνει με το αν μία αλλαγή έχει προλάβει να διαδοθεί στους Replica Managers, μπορεί να παρακαμφθεί μέσω της προσθήκης μίας μικρής καθυστέρησης από τον πρωτεύοντα Replica Manager πριν ενημερώσει τους γείτονες του για την αλλαγή. Από τις δύο λύσεις, επιλέξαμε την δεύτερη, καθώς διατηρεί τις αρχικές συνθήκες του πειράματος (k=3) και ανταποκρίνεται καλύτερα σε ένα ρεαλιστικό σενάριο, όπου οι κόμβοι του δικτύου βρίσκονται σε μακρινά δίκτυα και επομένως η επικοινωνία μεταξύ τους δεν είναι τόσο γρήγορη όσο στην περίπτωση μας.

Προσθέτοντας, λοιπόν, για eventual consistency μία καθυστέρηση των 0.1msec μεταξύ της εγγραφής ενός κλειδιού στη μνήμη του πρωτεύοντος Replica Manager και της αποστολής της ενημέρωσης στους γείτονες του, μπορέσαμε και παρατηρήσαμε καλύτερα το επιθυμητό φαινόμενο. Παρατηρήσαμε δηλαδή στα αποτελέσματα των queries ότι μερικές φορές επιστρέφεται μία stale τιμή, καθώς η ενημέρωση δεν έχει προλάβει να φτάσει στον κόμβο στον οποίο φτάνει το query. Αντιθέτως, στην περίπτωση του linearizability παρατηρούμε ότι όλα τα αποτελέσματα των queries έχουν τις πιο φρέσκιες τιμές. Ένα παράδειγμα το παραπάνω φαινομένου παρουσιάζεται στα επόμενα screenshots, στο οποίο ενημερώνεται η τιμή του κλειδιού "What's Going On" και μετά αμέσως εκτελείται query για το κλειδί αυτό. Στην περίπτωση του linearizability η τιμή που παίρνουμε είναι η πιο φρέσκια, ενώ στην περίπτωση του Eventual Consistency παίρνουμε μία παλιότερη τιμή. Αυτό είναι και το τίμημα που καλούμαστε να αποφασίσουμε αν θα πληρώσουμε, προκειμένου να χαρούμε τις πολύ γρηγορότερες ταχύτητες που παρατηρήθηκαν στα προηγούμενα δύο πειράματα.

Linearizability:

```
94 insert (key, value): (What's Going On->905303958512502305679465375635696178429610485544, 517)
95 query (key, value): (What's Going On->905303958512502305679465375635696178429610485544, 517)
```

Eventual Consistency:

```
94 insert (key, value): (What's Going On->905303958512502305679465375635696178429610485544, 517
95 query (key, value): (What's Going On->905303958512502305679465375635696178429610485544, 515)
```

Πίνακες Αποτελεσμάτων

Πείραμα 1 - Write Throughput

Replication Factor	Linerizability	Eventual Consistency
1	17.08 sec	17.08 sec
3	23.56 sec	18.17 sec
5	28.04 sec	19.46 sec

Πείραμα 2 - Read Throughput

Replication Factor	Linerizability	Eventual Consistency
1	16.59 sec	16.59 sec
3	16.19 sec	11.31 sec
5	16.86 sec	7.70 sec