Κατανεμημένα Συστήματα

Εξαμηνιαία Εργασία

ToyChord

Θωμάς Δούκας	Φοίβος-Ευστράτιος Καλεμκερής	Ιωάννης Ψαρράς
03116081	03116010	03116033

1 Εισαγωγή

Στην εργασία αυτή κληθήκαμε να σχεδιάσουμε μια απλοποιημένη εκδοχή του πρωτοκόλλου Chord για την υλοποίηση μιας file sharing εφαρμογής με πολλαπλούς κατανεμημένους κόμβους DHT. Ειδικότερα, χρειάστηκε να διαχειριστούμε τις εισαγωγές, αναζητήσεις, αλλά και διαγραφές εγγραφών, υλοποιώντας ταυτόχρονα δύο είδη συνέπειας, eventual consistency και linearizability.

2 Τεχνικά Χαρακτηριστικά

Για την υλοποίηση χρησιμοποποιήθηκαν οι ακόλουθες τεχνολογίες:

1. **Python3**
2. **HTTP requests** για την υλοποίηση της επικοινωνίας μεταξύ των κόμβων. Ειδικότερα:
   - **requests:** python package για την εκτέλεση HTTP requests
   - **Flask:** framework για τη διαχείριση του routing
3. **PyInquirer:** python package για την υλοποίηση διαδραστικού CLI

3 Υλοποίηση ToyChord

Κάθε κόμβος του συστήματος που κληθήκαμε να σχεδιάσουμε υλοποιεί όλες τις λειτουργίες του DHT. Πιο συγκεκριμένα, λειτουργώντας ταυτόχρονα ως server και client, υλοποιεί το πρωτόκολλο δρομολόγησης του Chord, λαμβάνοντας μηνύματα τόσο από τον client, όσο και από τους υπόλοιπους κόμβους του συστήματος και μεριμνώντας για τη διαχείριση ή και δρομολόγησή τους. Όπως αναφέραμε ήδη, η επικοινωνία μεταξύ των κόμβων πραγματοποιείται με HTTP requests. Για την υλοποίηση έχουμε θεωρήσει ότι οι κόμβοι δεν αποτυγχάνουν, αλλά αποχωρούν gracefully από το σύστημα. Στη συνέχεια θα περιγράψουμε αναλυτικά τους αλγορίθμους που ακολουθεί η υλοποίησή μας τόσο για τις βασικές λειτουργίες του Chord (join/departure κόμβων, κατανομή κλειδιών, replication), όσο και για εκείνες που παρέχονται στον client (search, insert, delete, network overlay). Αξίζει να σχολιάσουμε στο σημείο αυτό ότι, για τη σχεδίαση όλων των αλγορίθμων, βασικό μας μέλημα υπήρξε η κατά το δυνατόν μη απόκλιση από τη "νοοτροπία" του Chord:

- Ο κάθε κόμβος γνωρίζει μόνο τις απολύτως απαραίτητες πληροφορίες για το σύστημα:
  - Τη διεύθυνση (IP), τη θύρα (port) και το αναγνωριστικό (ID) του
  - Τη διεύθυνση (IP) και τη θύρα (port) του bootstrap κόμβου
  - Τη διεύθυνση (IP), τη θύρα (port) και το αναγνωριστικό (ID) του προηγούμενου (predecessor) και του επόμενου (successor) κόμβο
  - Το σύνολο των key-value pairs για τα οποία είναι υπεύθυνος
  - Εφαρμοζόμενη μέθοδος συνέπειας και replication factor
- Ο κάθε κόμβος δρομολογεί μηνύματα αυστηρά και μόνο στους άμεσους γείτονές του.

3.1 Λειτουργίες Chord

Θα αναλύσουμε πρώτα το σύνολο των λειτουργιών που αφορούν τη διαχείριση του δακτυλίου των κόμβων σύμφωνα με το πρωτόκολλο του Chord. Θεωρούμε δεδομένη την ύπαρξη ενός bootstrap κόμβου, ο οποίος εισέρχεται υποχρεωτικά πρώτος στο σύστημα, ενώ η διεύθυνση και η θύρα του είναι γνωστές σε όλους. Κατά την αρχικοποίησή του είναι απαραίτητο να δοθούν ως είσοδος το replication factor και το είδος συνέπειας (eventual/linearizability).

3.1.1 Join

Η πρώτη λειτουργία που επιτελείται κατά την εκτέλεση του server σε κάποιο μηχάνημα είναι η εισαγωγή αυτού ως κόμβο στο σύστημα (δακτύλιος κόμβων). Για να γίνει αυτό απαιτείται να δοθεί ως είσοδος η θύρα στην οποία θα ακούει στο εξής ο κόμβος. Από το hash του συνδυασμού διεύθυνσης και θύρας (hash(ip:port)) προκύπτει το αναγνωριστικό (ID) του κόμβου. Για την εισαγωγή στο σύστημα, ο νέος κόμβος αποστέλλει μήνυμα στον bootstrap, επισυνάπτοντας τα στοιχεία του (**action: JOIN**). Ο δεύτερος, αφού λάβει το μήνυμα, προσθέτει σ' αυτό το replication factor και το είδος συνέπειας και εκκινεί τη διαδικασία εύρεσης του successor του νέου κόμβου. Η διαδικασία αυτή είναι κομβικής σημασίας για το Chord, αφού σ' αυτή βασίζεται το σύνολο των λειτουργιών του. Στο εξής θα αναφέρεται ως **find_successor** (από το όνομα της συνάρτησης στην υλοποίησή μας), ενώ η αρχές λειτουργίας της (γνωστές από το πρωτόκολλο Chord) περιγράφονται συνοπτικά στη συνέχεια.

Σύμφωνα με την τεκμηρίωση του Chord, δεδομένου του ότι δε χρησιμοποιούνται finger tables, το σύνολο των λειτουργιών του DHT επιτελούνται με προώθηση μηνυμάτων στον αμέσως επόμενο κόμβο του δακτυλίου, στον οποίο για ευκολία θα αναφερόμαστε απλά ως successor. Πολύ συχνά, κατά την εφαρμογή του πρωτοκόλλου χρειάζεται να βρούμε τον κόμβο που είναι υπεύθυνος για ένα κλειδί, τον successor, δηλαδή, του κλειδιού. Η διαδικασία αυτή που ονομάσαμε **find_successor** βασίζεται στη σύγκριση κάθε φορά του αναζητούμενου κλειδιού με το ID του κόμβου που την εκτελεί και του προκατόχου του- στο εξής predecessor. Συγκεκριμένα, αν το κλειδί βρίσκεται μεταξύ των δύο ID (pred.ID $<$ key $\leq$ self.ID), σημαίνει ότι ο ίδιος ο κόμβος είναι υπεύθυνος για το κλειδί (είναι ο successor του), με αποτέλεσμα να πυροδοτούνται οι απαραίτητες ενέργειες (ποικίλλουν ανάλογα με το action που επιτελείται, όπως θα δούμε στη συνέχεια). Στην περίπτωση που η συνθήκη δεν ισχύει, ο κόμβος θα πρέπει να προωθήσει το μήνυμα στον επόμενό του, προκειμένου να συνεχιστεί η διαδικασία εύρεσης του successor node.

Όπως αναφέραμε ήδη, ο bootstrap εκκινεί μια διαδικασία **find_successor** με κλειδί το ID του νέου κόμβου. Μόλις το μήνυμα- προωθούμενο κυκλικά στο δακτύλιο- φτάσει στον successor node, εκείνος προωθεί το μήνυμα απευθείας στον νεοεισαχθέντα επισυνάπτοντας τα στοιχεία του ιδίου, αλλά και του προκατόχου του. Από το μήνυμα αυτό ο νέος κόμβος πληροφορείται για τη διεύθυνση και τη θύρα του successor του, τη διεύθυνση και τη θύρα του predecessor του, το replication factor και το είδος συνέπειας, πληροφορίες που αποθηκεύονται τοπικά. Ακολούθως, οφείλει να πληροφορήσει, αρχικά, τον predecessor για την είσοδό του στο σύστημα (**notify_predecessor**) προκειμένου κι εκείνος να ενημερώσει την πληροφορία για τον successor του. Ύστερα, οφείλει να επικοινωνήσει με τον successor, ζητώντας τα δεδομένα (key-value pairs) που του αντιστοιχούν (**request_items**). Υπενθυμίζουμε ότι κάθε κόμβος είναι υπεύθυνος για τα κλειδιά με τιμή μεγαλύτερη από το ID του προηγούμενου και μικρότερη ή ίση με το δικό του. Ο successor με τη σειρά του, λαμβάνοντας το μήνυμα, ενημερώνει την τοπική πληροφορία σχετικά με την ταυτότητα του predecessor του και έπειτα εντοπίζει τα δεδομένα που πρέπει να αποδοθούν σε εκείνον, τα οποία και διαγράφει από το τοπικό storage και αποστέλλει ως απάντηση στον νεοεισαχθέντα (**send_items**). Με τη λήψη τους από τον τελευταίο ολοκληρώνεται η διαδικασία του join και το σύστημα βρίσκεται σε συνεπή κατάσταση.

Replication
Στην περίπτωση που έχουμε replication factor μεγαλύτερο του 1, κατά τη διαδικασία του join πυροδοτούνται ορισμένες επιπρόσθετες ενέργειες. Πιο συγκεκριμένα, όπως προαναφέραμε, αφού βρεθεί η θέση του νεοεισαχθέντα κόμβου στον δακτύλιο, πρώτα ενημερώνεται ο predecessor του. Αυτός θα πρέπει να διατρέξει ολόκληρο το τοπικό storage του, προκειμένου να εντοπίσει εγγραφές για τις οποίες πρέπει να παραχθούν replicas, δηλαδή για όσα key-value pairs δεν είναι ο ίδιος το replication tail (δηλαδή, τοπικές εγγραφές με αριθμό replica $<$ replication factor). Για κάθε μία απ' αυτές θα πρέπει να αυξήσει τον αριθμό replica κατά 1 και να την επισυνάψει σε ένα μήνυμα προς τον νέο κόμβο. Εκείνος, αφού λάβει το συγκεκριμένο μήνυμα, επικοινωνεί με τον successor του, προκειμένου να λάβει όσες εγγραφές του αντιστοιχούν βάσει του ID του. Στη συνέχεια, εκκινεί μια διαδικασία replication συγκεντρώνοντας όλες τις εγγραφές με αριθμό replica $<$ replication factor, αυξάνοντας τον αριθμό κατά 1 και αποστέλοντάς τες στον επόμενο.

3.1.2 Departure

Εξίσου σημαντική με τη λειτουργία της εισόδου ενός νέου κόμβου στο σύστημα είναι εκείνη της αποχώρησης. Είναι κρίσιμο να τονίσουμε ότι θεωρούμε πως οι κόμβοι αποχωρούν πάντα gracefully από τον δακτύλιο, δεν αποτυγχάνουν, δηλαδή, και δεν αποσυνδέονται απροειδοποίητα. Αυτό σημαίνει ότι δίνεται η δυνατότητα λήψης όλων των αναγκαίων μέτρων προκειμένου το σύστημα να βρίσκεται σε συνεπή κατάσταση ακόμα και μετά την αποχώρηση.
Σύμφωνα με το πρωτόκολλο του Chord, κάθε κόμβος προτού αποχωρήσει οφείλει να ενημερώσει τον successor του, γνωστοποιώντας του τον νέο predecessor (δηλ. τον predecessor του κόμβου που αποχωρεί), αλλά και αποστέλλοντάς του το σύνολο των key-value pairs που διαθέτει. Ο successor, οφείλει να ενημερώσει τον νέο predecessor του (**notify_predecessor**) και να προσθέσει τις εγγραφές που έλαβε στο storage του.

Replication
Στην περίπτωση που έχουμε replication factor μεγαλύτερο του 1 και αφού ολοκληρωθεί η παραπάνω διαδικασία, ο successor οφείλει να ξεκινήσει διαδικασία replication για τις εγγραφές που έλαβε με αριθμό replica $<$ k. Αυτό, όπως και στην περίπτωση του join, επιτυγχάνεται προαυξάνοντας τον συγκεκριμένο αριθμό στις κατάλληλες εγγραφές και να τις προωθήσει στον επόμενο.

3.1.3 Replication

Στις προηγούμενες ενότητες είδαμε ότι το replication αποτελεί μια από τις βασικές λειτουργίες του συστήματος με αρκετά λεπτά σημεία, τα οποία θεωρούμε σκόπιμο να ξεκαθαρίσουμε. Ειδικότερα, η διαδικασία που περιγράψαμε προηγουμένως αντιστοιχεί σε μια ειδική περίπτωση replication, αυτή που εκτελεί το σύστημα από μόνο του προκειμένου να βρεθεί σε ισορροπία. Η συγκεκριμένη περίπτωση εμφανίζεται στην υλοποίηση με τη σημαία **action: REPL**. Αξίζει να σημειώσουμε ότι ακολουθούμε μια στρατηγική active replication, όπου κάθε κόμβος αποφασίζει ποια από τα key-value pairs που διαθέτει χρειάζεται να αντιγραφούν και εκκινεί τη διαδικασία replication αυξάνοντας τον αριθμό replica και στέλνοντας την εγγραφή στον επόμενο με σκοπό, αν αυτή πληροί ορισμένες προϋποθέσεις, να αποθηκευτεί απευθείας στο storage εκείνου και- αν απαιτείται- να συνεχιστεί με τον ίδιο τρόπο η διαδικασία.
Ενδιαφέρον παρουσιάζουν οι προαναφερθείσες προϋποθέσεις, αφού αυτές διασφαλίζουν τη συνέπεια του συστήματος. Πιο συγκεκριμένα, ένας κόμβος προσθέτει ένα key-value pair στο storage του μόνο αν:

- το αντίστοιχο key δεν υπήρχε προηγουμένως στο storage
- το αντίστοιχο key υπήρχε στο storage με αριθμό replica:
  - **ίσο** με αυτόν που λαμβάνει *(συχνή περίπτωση σε updates)*
  - **μικρότερο κατά 1** από αυτόν που λαμβάνει *(departure από προηγούμενη θέση στο δακτύλιο)*
  - **μεγαλύτερο κατά 1** από αυτόν που λαμβάνει *(join σε προηγούμενη θέση στο δακτύλιο)*

Όλες οι υπόλοιπες περιπτώσεις αντιμετωπίζονται ως overlap, αφού αποκλείεται σε κανονικές συνθήκες λειτουργίας (δεν υπάρχουν ταυτόχρονες αποχωρήσεις, όλες οι αποχωρήσεις είναι graceful) ένας κόμβος να λάβει replica με αριθμό που απέχει κατά 2 από το τοπικό αντίγραφο. Αν συμβεί κάτι τέτοιο, αντιλαμβανόμαστε ότι οι κόμβοι του συστήματος είναι λιγότεροι από το replication factor και, συνεπώς, το replication πρέπει να σταματήσει, αφού έχει γίνει ήδη κύκλος. Από την άλλη, αν κατά τη διαδικασία του replication κάποιος κόμβος συνειδητοποιήσει ότι έλαβε το τελευταίο αντίγραφο (είναι ο ίδιος το replication tail), οφείλει να μεριμνήσει προκειμένου να διαγραφούν επόμενα αντίγραφα που έχουν τυχόν ξεμείνει στο δακτύλιο, προωθώντας κατάλληλο μήνυμα διαγραφής στον successor του.
Εκτός, βέβαια, της περίπτωσης replication συστήματος, υπάρχουν 2 ακόμα περιπτώσεις που οφείλονται σε ενέργειες του client, συγκεκριμένα η αντιγραφή που ενεργοποιείται λόγω εισαγωγής νέου key-value pair ή διαγραφής ενός υπάρχοντος. Η πρώτη περίπτωση, η οποία σημειώνεται στην υλοποίηση με τη σημαία **action: INS_REPL** απαιτεί την εισαγωγή του απαιτούμενου αριθμού αντιγράφων στο σύστημα, ανάλογα με τον αριθμό κόμβων και το replication factor. Η δεύτερη, σημειώνεται με τη σημαία **action: DEL_REPL** και συνίσταται στη διαγραφή όλων των αντιγράφων του ζητούμενου κλειδιού. Οι δύο τελευταίες περιπτώσεις ακολουθούν παρόμοια λογική με το replication συστήματος και θα αναλυθούν περισσότερο στις αντίστοιχες ενότητες των λειτουργιών client.

Επισημαίνουμε, καταληκτικά, ότι το σύνολο των ελέγχων και υποπεριπτώσεων που αναφέραμε, εξετάζονται στη συνάρτηση **query** της υλοποίησης εφόσον k > 1, η οποία αποτελεί ουσιαστικά το "σημείο εισόδου" για τη **find_successor**, όπως αναλύθηκε προηγουμένως.

3.2 Λειτουργίες Client

Ακολούθως, θα μελετήσουμε το σύνολο των λειτουργιών που εκκινούνται από την διεπαφή του client με σκοπό την τροποποίηση και ανάκτηση των διαθέσιμων πληροφοριών του συστήματος. Είναι ιδιαίτερα σημαντικό να αναφέρουμε πως η υλοποίηση των λειτουργιών που ακολουθούν χαρακτηρίζεται από μια κοινή βασική στρατηγική. Συγκεκριμένα, κάθε μια από τις συναρτήσεις αξιοποιεί τις διαδικασίες **query** και **find_successor** οι οποίες συγκεντρώνουν το σύνολο των απαιτούμενων ελέγχων και μετασχηματισμών για καλύτερη εποπτεία της ροής εκτέλεσης. Ταυτόχρονα, είναι απαραίτητο να διασφαλίσουμε την ορθή επικοινωνία ανάμεσα στους κόμβους που αλληλεπιδρούν για την επιτυχή ολοκλήρωση μιας διαδικασίας. Προς το σκοπό αυτό επιλέξαμε να χρησιμοποιήσουμε διαφορετικά νήματα για την έναρξη δρομολόγησης της εκάστοτε λειτουργίας και την λήψη επιβεβαίωσης ολοκλήρωσης του αιτήματος. Για την αποφυγή της εμπλοκής τρίτων μηνυμάτων στη διαδικασία, λόγω της ασύγχρονης φύσης της εφαρμογής, κάθε λειτουργία χαρακτηρίζεται από ένα μοναδικό timestamp. Το συγκεκριμένο αξιοποιείται από την διαδικασία **eureka** του client η οποία καλείται από τον κόμβο του συστήματος που επιβάλει ο τύπος της συνέπειας. Εξασφαλίζεται, επομένως, πως ο χρήστης θα ενημερωθεί σωστά για την έκβαση της λειτουργίας. Η πρακτική αυτή χρησιμοποιείται αυτούσια στο σύνολο των λειτουργιών client που υλοποιήθηκαν.

3.2.1 Insert

Με την επιλογή της λειτουργίας εισαγωγής δεδομένων (*Insert a Song*) από το CLI ο χρήστης καλείται να προσδιορίσει το ζεύγος <key, value>. Ο client κόμβος του συστήματος θα εκκινήσει την διαδικασία **insert** κατά την οποία θα επιχειρήσει να αποθηκεύσει την συγκεκριμένη πληροφορία στους κατάλληλους κόμβους του συστήματος. Αξιοποιώντας πληροφορίες όπως το hash του κλειδιού εισαγωγής, την τιμή value, το μοναδικό timestamp, αλλά και τη σημαία **action: INSERT**, εκτελείται η συνάρτηση **query** του κόμβου εκκίνησης. Αυτή, με τη σειρά της θα εκτελέσει τη μέθοδο **find_successor** η οποία, μέσω προώθησης μηνυμάτων αν απαιτείται, συμβάλει στον προσδιορισμό του κόμβου εκείνου που είναι υπεύθυνος για την αποθήκευση του κλειδιού εισαγωγής. Έπειτα από την εύρεση του primary replica manager η ροή εκτέλεσης διασπάται σε δύο διαφορετικές, αναλόγως του τύπου συνέπειας που ορίζεται από τον χρήστη. Βασικό άξονα και στις 2 ροές αποτελεί η διαδικασία δημιουργίας αντιγράφων και η ενημέρωση των αντίστοιχων replica managers.

Eventual Consistency
Στην περίπτωση όπου το replication factor k είναι μεγαλύτερο της μονάδας και ακολουθείται eventual consistency, ο primary replica manager οφείλει να ενημερώσει τον client για την επιτυχή ολοκλήρωση της διαδικασίας, απαντώντας κατευθείαν σε αυτόν (dest_IP, dest_port). Επομένως, η ακολουθία προωθήσεων (**action: INSERT**) τερματίζεται από τον successor του κλειδιού εισαγωγής, ο οποίος ενημερώνει το τοπικό του storage προσθέτοντας την κατάλληλη εγγραφή. Έτσι, εφόσον ικανοποιείται η συνθήκη k>1 στη συνάρτηση **find_successor**, ο υπεύθυνος κόμβος θα δημιουργήσει νέο thread, αρμόδιο για την ενημέρωση των διάδοχων κόμβων και στην συνέχεια θα προβεί σε ενημέρωση του client. Κατά την επικοινωνία με τον client κόμβο θα ενεργοποιηθεί η λειτουργία **eureka** του δεύτερου, η οποία είναι υπεύθυνη για την αποδέσμευση του νήματος αναμονής επιβεβαίωσης και για την εμφάνιση του κατάλληλου μηνύματος στο χρήστη. Ο συγκεκριμένος τρόπος διαχείρισης διασφαλίζει πως οι αλλαγές διαδίδονται lazily στα αντίγραφα και ο πρωτεύων κόμβος επιστρέφει το αποτέλεσμα του write.
Προκειμένου να σταλεί η νέα τιμή εγγραφής στους k-1 επόμενους κόμβους, ο primary replica manager συντάσσει νέο μήνυμα που προωθείται στον επόμενο κόμβο. Για την νέα ακολουθία προωθήσεων απαιτείται αύξηση κατά 1 του αριθμού replica και χρήση της σημαίας **action: INS_REPL**, η οποία εκκινεί τη διαδικασία του replication για την νέα εγγραφή στο σύστημα. Κάθε ένας από τους κόμβους που λαμβάνει το παραπάνω μήνυμα ακολουθεί διαδικασία όμοια με αυτή που περιγράψαμε για το replication συστήματος, προσθέτοντας την εγγραφή στο τοπικό storage ή ενημερώνοντάς τη εάν υπάρχει ήδη και πληρούνται οι προϋποθέσεις που περιγράψαμε. Κάθε κόμβος ελέγχει αν το αντίγραφο που λαμβάνει έχει αριθμό replica ίσο με το replication factor ή αριθμό που απέχει περισσότερο από 1 από εκείνον του τοπικού αντιγράφου.

Αν ισχύει κάτι από τα παραπάνω ο κόμβος αντιλαμβάνεται ότι αποτελεί το replication tail και σταματά την προώθηση. Διαφορετικά, αυξάνει τον αριθμό replica κατά 1 και συνεχίζει τη διαδικασία **find_successor**.

Linearizability
Παρόμοια διαδικασία, αλλά με μερικές ουσιαστικές διαφορές ακολουθείται και στην περίπτωση του linearizability, για την υλοποίηση του οποίου εφαρμόσαμε **chain replication**. Σε αντίθεση με το eventual consistency ο τελευταίος κόμβος στη σειρά (replication tail) είναι εκείνος που πρέπει να ενημερώσει των χρήστη για την έκβαση του αιτήματος. Για το σκοπό αυτό, αφού προσδιοριστεί ο primary replica manager, ο ίδιος θα ενημερώσει κατάλληλα το τοπικό storage, προετοιμάζοντας στη συνέχεια κατάλληλο μήνυμα προς προώθηση στον επόμενο replica manager, χωρίς όμως να απαντήσει στον client. Απεναντίας, το μήνυμα **action: INS_REPL** θα διαδοθεί στο δακτύλιο, μέχρι την ανίχνευση του replication tail κόμβου με τον τρόπο που περιγράψαμε στην περίπτωση του eventual consictency, ο οποίος επιφορτίζεται με την ευθύνη διακοπής της διαδικασίας replication (αλλαγή σημαίας σε **STOP_INS**). Ταυτόχρονα, ο ίδιος έχει την ευθύνη της απάντησης στον client με το αντίγραφο με τον μεγαλύτερο αριθμό replica. Για την περίπτωση του linearizability που το replication διακοπεί πρόωρα λόγω ανίχνευσης overlap, έχουμε μεριμνήσει, ώστε στο μήνυμα **action: INS_REPL** να επισυνάπτεται κάθε φορά το τελευταίο αντίγραφο που προστέθηκε, το οποίο και επιστρέφεται τελικά στον client μόλις βρεθεί το tail (ως συνήθως μέσω της **eureka**).
Συνοψίζοντας, καταληκτικά, αυτή είναι και η θεμελιώδης διαφορά κατά την εγγραφή μεταξύ του eventual consistency και του chain replication: Ενώ στο πρώτο επιστρέφουμε στον client αμέσως μετά την εισαγωγή του πρώτου αντιγράφου στον primary replica manager, στη δεύτερη περίπτωση επιστρέφουμε μόνο αφού ολοκληρωθεί η διαδικασία replication.
Στο σημείο αυτό, αναφέρουμε πως η συγκεκριμένη λειτουργία καλύπτει τόσο την περίπτωση της εισαγωγής νέου κλειδιού με προσθήκη νέας εγγραφής στο τοπικό storage, όσο και την ενημέρωση της τιμής value ενός ήδη αποθηκευμένου στο σύστημα κλειδιού. Τονίζουμε επίσης, πως λαμβάνεται ειδική μέριμνα για την διαγραφή επιπλέον αντιγράφων που έχουν λανθασμένα παραμείνει αποθηκευμένα στο δακτύλιο.

3.2.2 Delete

Η επόμενη λειτουργία που θα μελετήσουμε είναι η διαγραφή μιας εγγραφής από το σύστημα. Η επιλογή *Delete a song* του CLI ζητά από τον χρήστη να συμπληρώσει τον τίτλο του τραγουδιού προς διαγραφή και ενεργοποιεί την διαδικασία **delete**. Η τελευταία, χρησιμοποιώντας ως κλειδί το hash του δοθέντα τίτλου, πραγματοποιεί αναζήτηση του υπεύθυνου για αυτό κόμβου (successor) μέσω της διαδικασίας **find_successor** χρησιμοποιώντας τη σημαία **action: DELETE**. Και πάλι, μετά την εύρεση του υπεύθυνου, η αντιμετώπιση διαφέρει, ανάλογα με τον τύπο συνέπειας, αλλά και την τιμή του replication factor k.

Eventual Consistency
Στην περίπτωση του eventual consistency και για replication factor k>1 ο υπεύθυνος κόμβος για το προς διαγραφή κλειδί θα επιδιώξει να διαγράψει την εγγραφή από το τοπικό του storage. Όπως και στην περίπτωση του eventual insert, θα χρησιμοποιηθεί ξεχωριστό thread για την προώθηση του μηνύματος deletion στους επόμενους κόμβους, ενώ το κύριο νήμα εκτέλεσης θα επιστρέψει την απάντηση στον χρήστη (**eureka**). Εάν επιδιώκεται διαγραφή στοιχείου που δεν υπάρχει αποθηκευμένο στο σύστημα, ο χρήστης ενημερώνεται σχετικά με κατάλληλο μήνυμα και η διαδικασία προώθησης μηνυμάτων διακόπτεται.

Ακολουθώντας την ροή ενημέρωσης των replica managers, τα μηνύματα προωθούνται με ενεργοποιημένη τη σημαία **action: DEL_REPL** και με κενή την τιμή value για το κλειδί διαγραφής. Η επιλογή αυτή αποσκοπεί στην τροποποίηση της λειτουργίας **query** ώστε κάθε κόμβος που λαμβάνει το αντίστοιχο μήνυμα να αφαιρεί από το storage την αντίστοιχη εγγραφή. Κάθε κόμβος ελέγχει τον τοπικό αριθμό replica και μόνο σε περίπτωση που αυτός είναι μικρότερος του replication factor k, προχωρά σε προώθηση του μηνύματος διαγραφής στον επόμενο κόμβο. Σε περίπτωση που ένας κόμβος εντοπίσει replica με αριθμό ίσο του k, τότε αναγνωρίζει ότι ο ίδιος αποτελεί το replication tail και εκτελεί την μέθοδο **find_successor** η οποία θα τερματίσει τη διαδικασία προώθησης.

Linearizability
Σε πλήρη αντιστοιχία με το linearizablility insert, η ενέργεια της διαγραφής οφείλει να ενημερώσει τον client μετά την ολοκλήρωση της αφαίρεσης όλων των αντιγράφων από το σύστημα. Επομένως, ο πρωτεύων κόμβος θα εκκινήσει την διαδικασία διαγράφοντας το στοιχείο από την τοπική μνήμη του και προωθώντας κατάλληλο μήνυμα στον successor του, χωρίς όμως να απαντήσει στον χρήστη. Το μήνυμα που προωθείται περιλαμβάνει και πάλι τη σημαία **action: DEL_REPL** με κάθε κόμβο να διαγράφει την τοπική εγγραφή και να επαναπροωθεί το μήνυμα μέχρι την εύρεση του replication tail. Μόλις διαγραφεί και το στοιχείο με αριθμό replica ίσο με το replication factor ή ανιχνευθεί overlap η προώθηση διακόπτεται (**action: STOP_DEL**) και ο κόμβος (replication tail) οφείλει να επιστρέψει στον client με την τελευταία εγγραφή που διαγράφηκε απ' το σύστημα. Για την επίτευξη αυτού, και πάλι κάθε κόμβος επισυνάπτει στο προς προώθηση μήνυμα την εγγραφή που διέγραψε από το τοπικό storage του, ώστε ακόμα και αν η διαδικασία διακοπεί πρόωρα (overlap) το tail να γνωρίζει την τελευταία τιμή που βρέθηκε.

Καταλήγοντας, η βασική διαφορά ανάμεσα στο eventual consistency και το linearizability είναι ίδια με εκείνη που παρατηρήσαμε για το insert. Στην πρώτη περίπτωση απαιτείται η επιστροφή αποτελέσματος στον χρήστη μετά την διαγραφή του στοιχείου από τον κύριο replica manager, ενώ στη δεύτερη οφείλουμε να εξασφαλίσουμε την διαγραφή του συνόλου των αντιγράφων από το σύστημα προτού επιστρέψουμε στον client.

3.2.3 Query

Μια από τις πιο σημαντικές λειτουργίες που συντελεί καθοριστικά στην επιλογή του Chord για την κατανεμημένη αποθήκευση εγγραφών είναι η αναζήτηση στο σύστημα. Αυτή ενεργοποιείται με την επιλογή *Search for a Song* από το CLI, για την οποία ο χρήστης θα κληθεί να δώσει τον τίτλο του προς αναζήτηση τραγουδιού, ο οποίος έχει χρησιμοποιηθεί για την παραγωγή του κλειδιού. Δεδομένου του ότι δεν χρησιμοποιούνται finger tables, η αναζήτηση θα πραγματοποιηθεί βάσει του hash του τίτλου με μια διαδικασία **find_successor**, όπως και στις προηγούμενες περιπτώσεις, με την προώθηση του μηνύματος (**action: SEARCH**) μέχρι την εύρεση του κόμβου στόχου. Ο τελευταίος διαφέρει ανάλογα με το εφαρμοζόμενο είδος συνέπειας.

Eventual Consistency
Η περίπτωση του eventual consistency για την αναζήτηση εγγραφών είναι, ομολογουμένως, πολύ απλή. Ειδικότερα, μας ενδιαφέρει να επιστρέψουμε το πρώτο αντίγραφο που θα βρεθεί για το ζητούμενο key. Για την επίτευξη αυτού, ελέγχουμε σε κάθε κόμβο από τον οποίο περνά το μήνυμα αναζήτησης αν το κλειδί υπάρχει στο τοπικό storage, ανεξάρτητα απ' το εάν ο κόμβος είναι ο primary replica manager. Στην περίπτωση που βρεθεί κάποιο αντίγραφο, ο κόμβος που το κατέχει καλείται να διακόψει την αναζήτηση και να απαντήσει στο χρήστη. Όπως καθίσταται αντιληπτό, η παραπάνω τακτική δε μας εγγυάται με κανέναν τρόπο ότι θα λάβουμε την πιο πρόσφατη έκδοση του αντιγράφου.
Φυσικά αυτό αποτελεί μια σχεδιαστική επιλογή, αφού με το eventual consistency θυσιάζουμε τη συνέπεια, αποσκοπώντας στη μεγαλύτερη δυνατή ταχύτητα απόκρισης, γεγονός που επιτυγχάνεται όπως θα φανεί και ύστερα στις μετρήσεις. Η προαναφερθείσα επιλογή μπορεί να είναι βιώσιμη μόνο σε εφαρμογές όπου η ανάγνωση πάντα της πιο πρόσφατης τιμής δεν είναι ζωτικής σημασίας (όπως για παράδειγμα σε μια εφαρμογή αναζήτησης τραγουδιών). Σε περίπτωση που η ζητούμενη τιμή δεν υπάρχει στο σύστημα, η αναζήτηση θα ολοκληρωθεί μόλις το προωθούμενο μήνυμα φτάσει στον κόμβο που θα ήταν θεωρητικά υπεύθυνος για το συγκεκριμένο κλειδί βάσει του ID του (primary replica manager).

Linearizability
Από την άλλη, στην περίπτωση του linearizability η αναζήτηση αποτελεί μια ελαφρώς πιο σύνθετη διαδικασία. Όπως ήδη αναφέραμε, υλοποιούμε chain replication, σύμφωνα με το οποίο, οι αναγνώσεις γίνονται μόνο από το replication tail. Ταυτόχρονα, θεωρούμε ότι ένα write έχει ολοκληρωθεί μόνο αφού η τιμή του γραφτεί σε όλους τους replica managers. Βάσει αυτών, αντιλαμβανόμαστε ότι αρκεί να εντοπίσουμε την τιμή του ζητούμενου κλειδιού που βρίσκεται αποθηκευμένη στο replication tail. Στη γενική περίπτωση η αναζήτηση του τελευταίου replica manager διακρίνεται σε δύο φάσεις:

1. Εύρεση του primary replica manager
2. Εύρεση του tail

Η υλοποίηση της πρώτης φάσης δε διαφέρει σε τίποτα από τη συνηθισμένη διαδικασία **find_successor** που ακολουθήθηκε και κατά τα insert/delete για την εύρεση του primary replica manager, αυτή τη φορά με σημαία **action: SEARCH** και, επιπρόσθετα, **consistency: LINEARIZABILITY**. Φυσικά, αν η αναζήτηση ξεκινήσει από κάποιον κόμβο που διαθέτει τοπικά αντίγραφο για το συγκεκριμένο κλειδί, η εύρεση του primary replica manager θεωρείται περιττή. Συνεπώς στην περίπτωση αυτή περνάμε απευθείας στη δεύτερη φάση της αναζήτησης με σκοπό τον εντοπισμό του tail, με σημαίες **action: SEARCH** και **consistency: LINEARIZABILITY_PHASE_2**. Αντίστοιχα με τις περιπτώσεις insert/delete για linearizability, αρκεί να βρούμε τον κόμβο με το μεγαλύτερο αριθμό αντιγράφου. Και πάλι, χρειάζεται να προωθούμε το τελευταίο αντίγραφο που εντοπίζουμε, ώστε ακόμα και αν υπάρξει overlap, ο κόμβος που θα το ανιχνεύσει να μπορεί να επιστρέψει στον client. Αν από την άλλη, το κλειδί δε βρεθεί στον primary, η αναζήτηση διακόπτεται και επιστρέφουμε αμέσως στο χρήστη με κατάλληλο μήνυμα, χωρίς να μπούμε καν στη δεύτερη φάση.

Τέλος, υπάρχει μια ειδική περίπτωση της λειτουργίας αναζήτησης που ενεργοποιείται όταν ως τίτλος τραγουδιού δοθεί το σύμβολο '*'. Τότε, πραγματοποιείται μια διαδικασία παρόμοια με αυτή που θα περιγράψουμε παρακάτω στην ενότητα Overlay, όπου ο κάθε κόμβος επισυνάπτει εκτός από το συνδυασμό διεύθυνση:θύρα και το σύνολο των εγγραφών που διαθέτει αποθηκευμένες τοπικά. Ως έξοδος της συγκεκριμένης λειτουργίας λαμβάνεται μια λίστα με όλους τους κόμβους του δακτυλίου, για καθέναν από τους οποίους παρατίθεται μια λίστα των αποθηκευμένων εγγραφών.

3.2.4 Overlay

Η τελευταία από τις λειτουργίες που παρέχονται στον client είναι αυτή της εκτύπωσης της τοπολογίας του δακτυλίου Chord (Network Overlay) από την οπτική του κόμβου που εκκινεί τη διαδικασία. Πρόκειται για την πιο απλή λειτουργία, αφού είναι ανεξάρτητη του είδους consistency και του replication factor, μιας και δεν ασχολείται καθόλου με τις εγγραφές κάθε κόμβου. Η υλοποίησή της συνίσταται απλώς σε μια διαδικασία **find_successor** με κλειδί το αναγνωριστικό (ID) του predecessor του κόμβου που την εκκινεί, προκειμένου να διασφαλίσουμε ότι το μήνυμα του overlay θα προωθηθεί σε ολόκληρο τον δακτύλιο. Κάθε κόμβος που λαμβάνει το εν λόγω μήνυμα με σημαία **action: OVERLAY**, επισυνάπτει σ' αυτό τη διεύθυνση και τη θύρα του ως συμβολοσειρά 'ip:port'. Ο τελευταίος κόμβος του δακτυλίου, αφού προσθέσει τα δικά του στοιχεία, προωθεί το μήνυμα στον αρχικό, ο οποίος είναι υπεύθυνος για την εκτύπωση των 'ip:port' συμβολοσειρών χωριζόμενων με '->', ώστε να φαίνεται η διασύνδεσή τους.

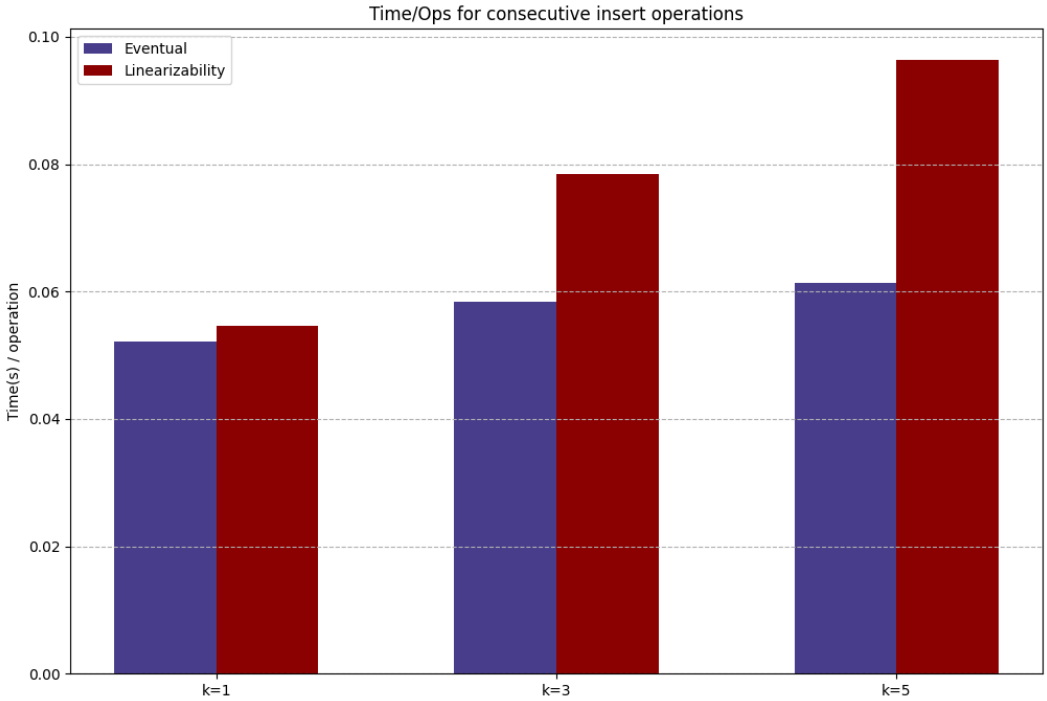
4 Πειράματα

Σε αυτό το σημείο, κληθήκαμε να εκτελέσουμε 3 πειράματα για να εξακριβώσουμε την απόδοση του συστήματός μας. Τα ζητούμενα πειράματα εξετάζουν, κατά σειρά, το write throughput, το read throughput και, τέλος, τη συμπεριφορά του συστήματος για συνδυασμό read/write requests.

Ξεκινάμε την εκτέλεση των πειραμάτων σε αρχικά άδειο DHT με 10 κόμβους. Για τη λήψη των μετρήσεων διαβάζουμε τα αντίστοιχα requests από τα αρχεία της εκφώνησης, δρομολογώντας καθένα εξ αυτών σε έναν τυχαίο κόμβο του συστήματος (χρησιμοποιούμε γεννήτρια τυχαίων αριθμών [1, 10] με αντιστοίχιση σε γνωστές διευθύνσεις-θύρες) και υπολογίζοντας το συνολικό χρόνο από τη διαφορά μεταξύ της στιγμής έναρξης και ολοκλήρωσης των αιτημάτων. Σε όλες τις περιπτώσεις θα χρησιμοποιήσουμε ως βασική μετρική επίδοσης τον χρόνο ανά request (Time(sec)/Ops).

4.1 Write Throughput

Για την διερεύνηση του write throughput, κληθήκαμε να ελέγξουμε την συμπεριφορά του συστήματός μας απέναντι σε διαδοχικά inserts, όπως δίνονται στο αρχείο insert.txt, για τους 6 διαφορετικούς συνδυασμούς replication factor/consistency type που υποδεικνύονται από την εκφώνηση. Σε μια πρώτη προσπάθεια ερμηνείας των αποτελεσμάτων, παράγουμε το παρακάτω διάγραμμα, το οποίο μας οδηγεί σε κάποια αρχικά συμπεράσματα:



Σχήμα 1: Αποτελέσματα πειράματος Write Throughput

Αρχικά, παρατηρούμε ότι στην περίπτωση του linearizability, αύξηση του k συνεπάγεται απότομη αύξηση του απαιτούμενου χρόνου ανά insertion. Αυτό συμβαίνει διότι σύμφωνα με το chain replication που έχει υλοποιηθεί, ένα write ολοκληρώνεται μόνο αφότου πραγματοποιηθεί εγγραφή και στους k replica managers, με τον τελευταίο να είναι υπεύθυνος για την επιστροφή του αποτελέσματος του write στον client. Απεναντίας, στην περίπτωση του eventual consistency το write ολοκληρώνεται αμέσως μετά την εγγραφή στον primary replica manager, ο οποίος απαντά σχετικά στον client, με τη διάδοση των αντιγράφων να γίνεται lazily στη συνέχεια. Βάσει των παραπάνω, τα αποτελέσματα που λάβαμε είναι απολύτως αναμενόμενα. Ειδικότερα, για k=1 ο χρόνος για linearizability και eventual consistency είναι περίπου ο ίδιος (θεωρητικά ακριβώς ίδιος), αφού και στις δύο περιπτώσεις αναζητείται ο υπεύθυνος κόμβος για το προς εισαγωγή κλειδί και το write ολοκληρώνεται με την εγγραφή σ' αυτόν (δεν υπάρχουν αντίγραφα), με αποτέλεσμα ο ίδιος να απαντά στον client σχετικά. Όσο αυξάνεται, ωστόσο, το replication factor, η συμπεριφορά του συστήματος για linearizability αποκλίνει, αφού στην περίπτωση αυτή απαιτείται, μετά την εισαγωγή του κλειδιού στον πρωτεύοντα κόμβο, η διάδοση και αποθήκευση των αντιγράφων στους επόμενους k-1 replica managers, με τον τελευταίο εξ αυτών (replication tail) να είναι υπεύθυνος για την απάντηση στο χρήστη (ολοκλήρωση write).

Ενώ για την περίπτωση του linearizability τα περιαματικά αποτελέσματα επαληθεύονται πλήρως θεωρητικά, για το eventual consistency παρατηρούμε μια χαρακτηριστική απόκλιση. Πιο συγκεκριμένα, δεδομένου του ότι, ανεξαρτήτως του replication factor, τα write επιστρέφουν (ολοκληρώνονται) μετά την εγγραφή στον primary replica manager, θα περιμέναμε ο χρόνος/operation να παραμένει σταθερός και στα 3 πειράματα. Αντιθέτως, αυτός εμφανίζει μια πολύ μικρή, αλλά σταθερή αύξηση, όσο το replication factor μεγαλώνει. Αυτή αποδίδεται στο γεγονός ότι σε ένα πραγματικό σύστημα, η επεξεργασία των εισερχόμενων request ακολουθεί, αφενός, FIFO διάταξη και, αφετέρου, απαιτεί κάποιον μη μηδενικό χρόνο προκειμένου να ολοκληρωθεί. Συνεπώς, παρόλο που το write επιστρέφει αμέσως μετά την εγγραφή στον υπεύθυνο κόμβο, η lazy διάδοση των αντιγράφων απασχολεί τους επόμενους κόμβους, με αποτέλεσμα όταν έρθει το επόμενο write να τους βρει στιγμιαία κατειλημμένους, με κάποια πιθανότητα που αυξάνεται:

- όσο αυξάνεται το replication factor
- όσο μειώνεται ο αριθμός κόμβων στο σύστημα

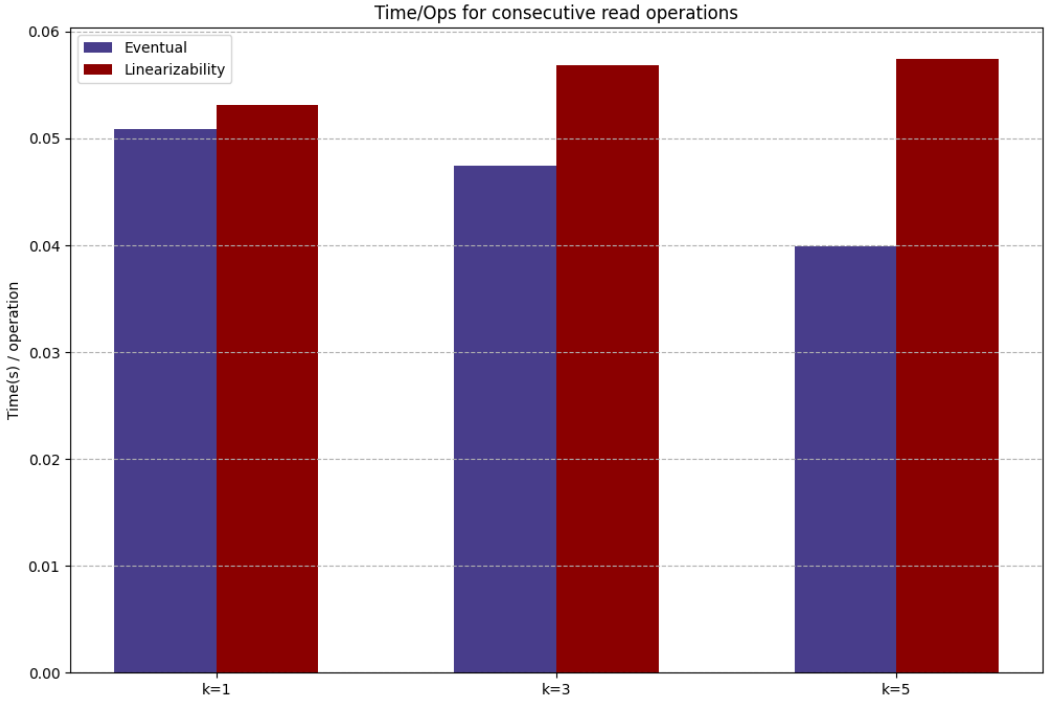
Για επαλήθευση των παραπάνω παρατηρήσεων, παραθέτουμε τις ακριβείς μετρήσεις για τους χρόνους εκτέλεσης των παραπάνω 6 περιπτώσεων:

Consistency	Replication Factor	Time(sec)/WriteOps
Eventual	1	0.0522086386680603
Linearizability	1	0.05468967294692993
Eventual	3	0.058438993453979494
Linearizability	3	0.07842908382415771
Eventual	5	0.06140097570419312
Linearizability	5	0.09642767238616944

Πίνακας 1: Consecutive Writes

4.2 Read Throughput

Παρόμοια με την περίπτωση του write throughput, για τον έλεγχο του read throughput δοκιμάσαμε την ανταπόκριση των 6 περιπτώσεων replication factor / consistency, σε μια σειρά διαδοχικών queries, όπως αυτά βρίσκονται στο αρχείο query.txt. Για καλύτερη οπτικοποίηση των αποτελεσμάτων, παραθέτουμε ξανά το σχετικό ραβδόγραμμα, το οποίο και θα σχολιάσουμε, όπως προηγουμένως:



Σχήμα 2: Αποτελέσματα πειράματος Read Throughput

Στην περίπτωση των queries, όπως αναδεικνύεται και από το διάγραμμα, οι δύο διαφορετικοί τύποι συνέπειας εμφανίζουν αντιδιαμετρικά αντίθετη συμπεριφορά. Από τη μία, στην περίπτωση του eventual consistency, είναι εμφανές πως ο χρόνος που απαιτείται για να ολοκληρωθεί ένα query, μειώνεται για μεγαλύτερες τιμές του k, δηλαδή για μεγαλύτερο αριθμό replicas. Το αποτέλεσμα αυτό είναι, βεβαίως, αναμενόμενο, καθώς μεγαλύτερος αριθμός replicas, εξασφαλίζει μεγαλύτερη πιθανότητα να βρεθεί αντίγραφο της ζητούμενης εγγραφής σε περιοχή κοντινή του κόμβου από τον οποίο ξεκινάει το request (ή στον κόμβο τον ίδιο), ενώ στη χειρότερη περίπτωση, κατά την οποία η ζητούμενη εγγραφή δεν υπάρχει στο σύστημα, αυτό θα διαπιστωθεί μόλις το query φτάσει στο υπεύθυνο βάσει ID κόμβο (primary).
Από την άλλη, στο linearizability, αύξηση του replication factor συνεπάγεται αύξηση του απαιτούμενου χρόνου για την ολοκλήρωση του query. Στην περίπτωση αυτή, λόγω του chain replication, η ανάγνωση για κάθε κλειδί μπορεί να γίνει μόνο από έναν συγκεκριμένο κόμβο, το replication tail της ζητούμενης εγγραφής. Σύμφωνα με την παρατήρηση αυτή, ο χρόνος αναζήτησης στο δακτύλιο εξαρτάται άμεσα από τον αριθμό κόμβων N του συστήματος, αφού στη χειρότερη περίπτωση ένα query θα χρειαστεί να προωθηθεί σε ολόκληρη την αλυσίδα μέχρι να φτάσει στον υπεύθυνο για την απάντηση κόμβο (O(N)). Στη δική μας περίπτωση, ωστόσο, τα πειράματα πραγματοποιούνται για σταθερό αριθμό κόμβων (10).

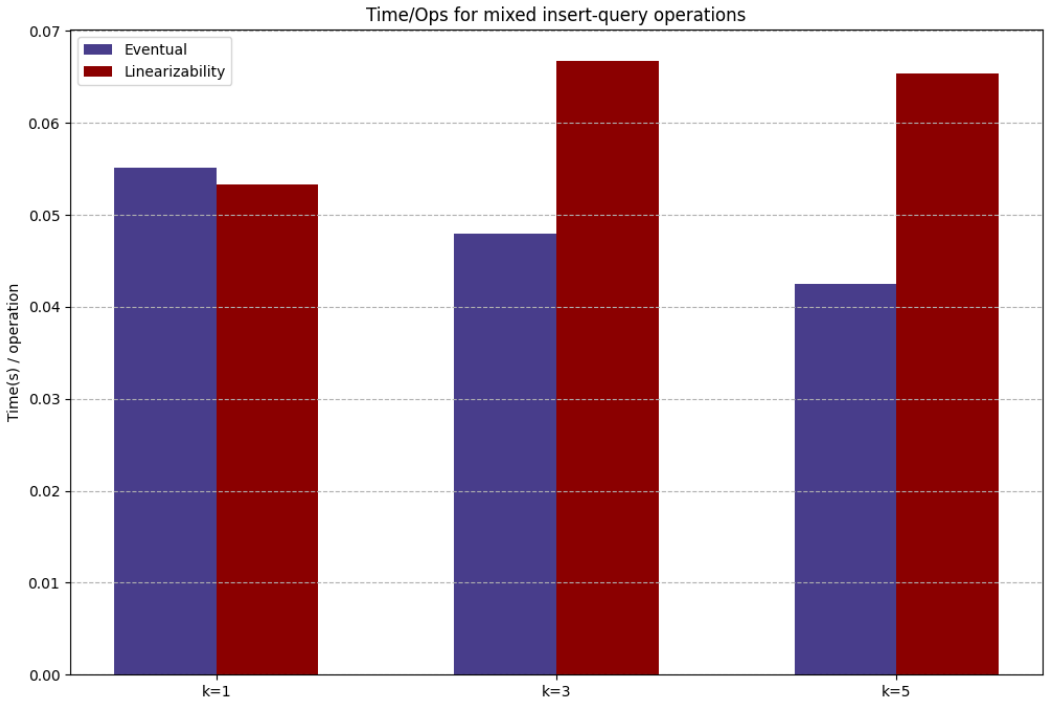
Χρειάζεται να σημειώσουμε, συνεπώς, ότι η παρατηρούμενη αύξηση του χρόνου για μεγαλύτερα k, δεν οφείλεται στην αύξηση του replication factor καθαυτή, αλλά στο αυξημένο διαχειριστικό κόστος που εκείνη συνεπάγεται, καθώς όπως περιγράψαμε σε προηγούμενη ενότητα, η αναζήτηση του replication tail αποτελεί μη τετριμμένη διαδικασία. Αυτή η συμπεριφορά των δύο τύπων συνέπειας, απεικονίζεται εξίσου ξεκάθαρα και στους ακριβείς μέσους χρόνους ανά query που λάβαμε από την εκτέλεση των 6 πειραμάτων:

Consistency	**Replication Factor**	**Time(sec)/ReadOps**
Eventual	1	0.05086453104019165
Linearizability	1	0.053095159530639646
Eventual	3	0.047408805847167966
Linearizability	3	0.0568283953666687
Eventual	5	0.03993151330947876
Linearizability	5	0.057477904319763184

Πίνακας 2: Consecutive Reads

4.3 Συνδυασμός Read-Write αιτημάτων

Στο τελευταίο στάδιο των πειραμάτων, κληθήκαμε να εξετάσουμε την περίπτωση όπου το σύστημά μας δέχεται διαδοχικά read και write requests, όπως δίνονται στο αρχείο requests.txt. Για λόγους συνέπειας με τα προηγούμενα πειράματα, θα ελέγξουμε αρχικά τη χρονική επίδοση του συστήματος, παραθέτοντας το σχετικό ραβδόγραμμα.



Σχήμα 3: Αποτελέσματα συνδυαστικού πειράματος

Όπως ήταν αναμενόμενο, η περίπτωση των ανάμεικτων read-write λειτουργιών παρουσιάζει μεγάλη ομοιότητα με εκείνη των reads που μελετήσαμε προηγουμένως. Για ακόμη μια φορά, η δυνατότητα, που παρέχεται από το eventual consistency, του να απαντάμε σε ένα query από τον κοντινότερο κόμβο στον οποίο βρίσκεται η ζητούμενη τιμή, αποδεικνύεται ευεργετική για τον μέσο χρόνο ολοκλήρωσης ενός query. Εξαιτίας αυτού, μειώνεται και ο συνολικός μέσος χρόνος ανά οποιαδήποτε εντολή, εφόσον, όπως είδαμε στην περίπτωση του write throughput, ο μέσος χρόνος ανά insert δεν αυξάνεται σημαντικά για μεγαλύτερες τιμές του k.

Από την άλλη, στην περίπτωση του linearizability παρατηρείται κι αυτή τη φορά αύξηση του απαιτούμενου χρόνου για k>1. Ωστόσο, το γεγονός ότι ο χρόνος για k=3 είναι μεγαλύτερος απ' ότι για k=5, επιβεβαιώνει την παρατήρησή μας πως κυρίαρχο ρόλο παίζει ο συνολικός αριθμός κόμβων στο δακτύλιο και όχι το replication factor. Παρακάτω φαίνονται και οι ακριβείς μέσοι χρόνοι ανά οποιαδήποτε εντολή, που επιτεύχθηκαν για τις 6 διαφορετικές περιπτώσεις που εξετάζουμε:

Consistency	**Replication Factor**	**Time(sec)/Ops**
Eventual	1	0.055170466899871824
Linearizability	1	0.053294196128845214
Eventual	3	0.04799947261810303
Linearizability	3	0.0667893385887146
Eventual	5	0.042521190643310544
Linearizability	5	0.06537123250961303

Πίνακας 3: Mixed Consecutive Read-Writes

Πέραν της διερεύνησης ως προς τον απαιτούμενο χρόνο ανά request, άξια μελέτης είναι και η ικανότητα του συστήματος να παρέχει, όσο το δυνατόν, πιο επικαιροποιημένα δεδομένα στον χρήστη. Θα πρέπει να εξετάσουμε, δηλαδή, κατά πόσο τα δεδομένα που επιστρέφει ένα query είναι πράγματι τα πιο πρόσφατα. Άλλωστε, το βασικό trade-off που αναδεικνύεται κατά την επιλογή του τύπου συνέπειας σε ένα σύστημα αφορά τις δύο αυτές μετρικές, το χρόνο απόκρισης και την ανάγκη να διαβάζουμε πάντα τα πιο πρόσφατα δεδομένα.
Από τη μία, επιλογή ενός χαλαρού μοντέλου συνέπειας- όπως το eventual consistency- σημαίνει ότι μπορούμε να έχουμε όσο το δυνατόν πιο γρήγορες εγγραφές, εκμεταλλευόμενοι, δε, κατά την ανάγνωση την ύπαρξη πολλαπλών αντιγράφων προκειμένου να ελαχιστοποιήσουμε το χρόνο αναζήτησης. Φυσικά, το τίμημα για την προσφερόμενη ταχύτητα είναι η πιθανότητα ανάγνωσης stale αντιγράφων. Το linearizability, από την άλλη, ως η πιο αυστηρή μορφή συνέπειας, στοχεύει στην εξάλειψη αυτής της πιθανότητας, θυσιάζοντας το χρόνο απόκρισης και κατά την ανάγνωση και κατά την εγγραφή. Τελικά, η επιλογή μεταξύ των δύο τύπων συνέπειας εξαρτάται από το είδος της εφαρμογής και την κρισιμότητα των δεδομένων, όπως έχουμε ήδη αναφέρει. Αυτό που μένει να διαπιστωθεί για το παρόν σύστημα είναι η "πιθανότητα" να επιστραφούν stale δεδομένα στην περίπτωση του eventual consistency. Για το σκοπό αυτό πραγματοποιήσαμε το ακόλουθο πείραμα.

Αρχικά, καταγράψαμε την έξοδο του συστήματος για τη "σειριακή" εκτέλεση του πειράματος των requests, της εκτέλεσης, δηλαδή, όπου όλα τα αιτήματα ξεκινούν από τον ίδιο κόμβο. Ύστερα, καταγράψαμε τις αντίστοιχες εξόδους για την περίπτωση του eventual consistency και του linearizability, χρησιμοποιώντας την υποδομή των προηγούμενων πειραμάτων για δρομολόγηση κάθε request σε τυχαίο κόμβο. Και για τις τρεις περιπτώσεις ξεκινήσαμε με κενό DHT.

Για τη σύγκριση των εξόδων υλοποιήσαμε το script **check_freshness.py**. Αυτό διαβάζει τα τρία αρχεία και, για τις γραμμές που δεν αντιστοιχούν σε insert, συγκρίνει τις εξόδους των eventual και linearizability με εκείνη της σειριακής εκτέλεσης. Αν το value είναι ίδιο ή η απάντηση είναι not found και στα δύο, προχωράμε στην επόμενη γραμμή. Διαφορετικά, αυξάνουμε έναν μετρητή λαθών για το αρχείο στο οποίο παρατηρήθηκε απόκλιση από τη σειριακή εκτέλεση. Η διαδικασία πραγματοποιείται για k=3 και k=5. Επισημαίνουμε ότι κατά τη σύγκριση μας απασχολεί μόνο το value της εγγραφής και όχι ο κόμβος που απαντά ή ο αριθμός replica που είναι αναμενόμενο να διαφέρουν. Στην επόμενη σελίδα παραθέτουμε την έξοδο του script.

```
We have compared the results to the serial execution
(all requests starting from a single node):
-----------------------------------------------------
Replication Factor k: 3
Eventual Consistency mistakes: 0
Linearizability mistakes: 0
-------------------------------------
Replication Factor k: 5
Eventual Consistency mistakes: 1
Linearizability mistakes: 0
-------------------------------------
```

Τα αποτελέσματα της παραπάνω εξόδου είναι ιδιαίτερα ενδιαφέροντα. Ειδικότερα, παρατηρούμε ότι για k=3 και οι δύο εκτελέσεις επιστρέφουν τις ίδιες τιμές με τη σειριακή, ενώ για k=5 εμφανίζεται μόλις 1 λάθος στην περίπτωση του eventual consistency. Φυσικά αυτό απέχει πολύ από τη θεωρία, σύμφωνα με την οποία θα περιμέναμε πολύ μεγαλύτερη απόκλιση για το eventual, λόγω του lazy replication. Αναδεικνύεται, ωστόσο, μια πραγματικότητα: στη γενική περίπτωση, όπου τα requests ξεκινούν από τυχαίους κόμβους, είναι μάλλον ασυνήθιστο να διαβάσουμε stale τιμές, αφού οι αλλαγές προλαβαίνουν να διαδοθούν. Για να διαπιστώσουμε την υπεροχή του linearizability θα χρειαζόταν ένα προσεκτικά σχεδιασμένο test case με διαδοχικές εγγραφές και αναγνώσεις στην ίδια περιοχή του δακτυλίου από συγκεκριμένους- όχι τυχαίους- κόμβους.
Με τα δεδομένα που διαθέτουμε, ωστόσο, είναι ξεκάθαρο ότι για τη συγκεκριμένη εφαρμογή το eventual consistency υπερτερεί, παρουσιάζοντας πολύ μικρότερο χρόνο απόκρισης και πολύ ικανοποιητική ακρίβεια στις απαντήσεις του. Εξάλλου, ακόμα και αν τα παρατηρούμενα λάθη ήταν περισσότερα, η φύση της εφαρμογής είναι τέτοια, ώστε να μη θεωρούνται κρίσιμα για τη λειτουργία της.

5 Φάκελος Project

Στον φάκελο του project, εκτός από την αναφορά, περιλαμβάνεται ο κώδικας της εφαρμογής, αλλά και script για τη λήψη και οπτικοποίηση των μετρήσεων. Ειδικότερα, εντός του φακέλου dingane βρίσκονται:

1. **server.py**: εκκινεί έναν κόμβο
2. **Node.py**: περιλαμβάνει την κλάση του κόμβου του δακτυλίου, καθώς και τις σχετικές μεθόδους για την εξυπηρέτηση λειτουργιών συστήματος
3. **endpoints/chord.py**: περιλαμβάνει τα endpoints για τις λειτουργίες συστήματος (επικοινωνία κόμβων)
4. **endpoints/client.py**: περιλαμβάνει τα endpoints για τις λειτουργίες που παρέχονται στον client
5. **cli.py**: CLI εργαλείο
6. **tests**/: ο φάκελος των μετρήσεων, περιλαμβάνει:
   (α′) **insert.py, query.py, req.py**: η υλοποίηση των πειραμάτων
   (β′) **check_freshness**: script για σύγκριση αποτελεσμάτων των τύπων συνέπειας
   (γ′) **plot.py**: script για τη δημιουργία των barplots.
   (δ′) **plots**/: οι γραφικές μετρήσεις των πειραμάτων

5.1 Εκτέλεση

Για την εκτέλεση της εφαρμογής απαιτείται να έχει πραγματοποιηθεί εγκατάσταση του συνόλου των εξαρτήσεων από pip packages, η οποία μπορεί να γίνει με την εκτέλεση του script **dependencies.sh**.

5.1.1 Εκκίνηση Server

Για την εκκίνηση της εφαρμογής είναι απαραίτητη η εισαγωγή του bootstrap κόμβου στο σύστημα. Αυτός θεωρούμε ότι ακούει σε διεύθυνση γνωστή σε όλους (192.168.1.1:5000). Αρκεί να τρέξουμε στο αντίστοιχο μηχάνημα την εντολή:

```
python3 server.py -bs -p 5000 -k factor -c linearizability
```

Οι σημαίες -bs και -p 5000 είναι αναγκαίες προκειμένου να δηλώσουμε ότι πρόκειται για τον bootstrap κόμβο που ακούει στη θύρα 5000. Οι δύο επόμενες σημαίες δεν είναι υποχρεωτικές. Ειδικότερα, αν παραληφθεί η σημαία k, θα χρησιμοποιηθεί η default τιμή για το replication factor, δηλαδή 1, ενώ αν παραληφθεί η σημαία c, θα εφαρμοστεί το default είδος consistency, το eventual (στην πραγματικότητα για να επιλεγεί το linearizability αρκεί να δοθεί οποιαδήποτε τιμή στο c πχ linearizability, "l", 42, Indlovu, κλπ).

Αφού έχουμε εκκινήσει τον boostrap κόμβο, μπορούμε να προσθέσουμε άλλους κόμβους στο σύστημα, αυτή τη φορά προσδιορίζοντας μόνο έναν έγκυρο αριθμό θύρας (port).

```
python3 server.py -p port
```

Ο bootstrap θα μεριμνήσει, ώστε ο νέος κόμβος να πληροφορηθεί για όσες ιδιότητες του συστήματος χρειάζεται.

5.1.2 Εκκίνηση CLI

Αφού έχουμε ξεκινήσει κάποιος κόμβους (τουλάχιστον τον boostrap), μπορούμε να ξεκινήσουμε το CLI για κάποιον από τους ενεργούς παρέχοντας τη θύρα στην οποία αυτός ακούει, προκειμένου να πραγματοποιήσουμε τις παρεχόμενες στον client λειτουργίες.

```
python3 cli.py -p port
```

Με τη βοήθεια του PyInquirer έχουμε υλοποιήσει ένα διαδραστικό CLI εργαλείο, στο οποίο ο χρήστης μπορεί να πλοηγηθεί χρησιμοποιώντας τα arrow keys, ενώ για την εκτέλεση των requests καθοδηγείται από την εφαρμογή. Παρέχονται οι εξής λειτουργίες, όπως ζητούνται στην εκφώνηση (σε παρένθεση το όνομα όπως εμφανίζεται στην εφαρμογή):

- Overlay (Network Overlay)
- Query (Search for a Song)
- Insert (Insert a Song)
- Delete (Delete a Song)
- Depart (Depart)
- Help (Help)
- Exit

Με την επιλογή του exit ο χρήστης βγαίνει από το CLI, χωρίς ο αντίστοιχος κόμβος να κάνει depart.