Σχολη Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών

ΕΞΑΜΗΝΙΑΙΑ ΕΡΓΑΣΙΑ ΣΤΑ ΚΑΤΑΝΕΜΗΜΕΝΑ ΣΥΣΤΗΜΑΤΑ

Ακ. έτος 2024-2025, 9ο Εξάμηνο

Γεώργιος Γεωργακόπουλος, 03120827
Εμμανουήλ Νεοφώτιστος, 03119128

Πίνακας Περιεχομένων

1. Εισαγωγή

Στην παρούσα εργασία αναπτύξαμε το Chordify, μια P2P εφαρμογή ανταλλαγής τραγουδιών βασισμένη στο πρωτόκολλο Chord DHT. Ο στόχος ήταν η υλοποίηση ενός κατανεμημένου συστήματος file sharing, όπου κάθε κόμβος του δικτύου συνεργάζεται για την αποθήκευση και εύρεση των τραγουδιών με βάση τον τίτλο τους.

Για την ανάπτυξη του συστήματος χρησιμοποιήθηκε αρχικά το εργαλείο Docker, τρέχοντας τοπικά τους κόμβους μας, ενώ στο τέλος αξιοποιήσαμε τους πόρους του AWS που μας δόθηκαν, δημιουργώντας σε κάθε VM, containers για τους κόμβους. Η υλοποίηση του backend πραγματοποιήθηκε με το Flask, το οποίο διευκόλυνε την υλοποίηση των απαραίτητων λειτουργιών και την επικοινωνία μεταξύ των κόμβων.

Η εργασία περιλάμβανε την ανάπτυξη βασικών λειτουργιών όπως η διαίρεση του χώρου των IDs, η δρομολόγηση στον δακτύλιο, η εισαγωγή και αναχώρηση κόμβων καθώς και ο μηχανισμός αναπαραγωγής (replication) των δεδομένων.

Ο κώδικας της εφαρμογής βρίσκεται στο ακόλουθο repository:

https://github.com/ntua-el20827/NTUA_Distributed_Systems

2. Σχεδιασμός και Υλοποίηση

2.1. Αρχιτεκτονική του Συστήματος

Κόμβοι DHT: Κάθε κόμβος λαμβάνει μοναδικό ID μέσω της εφαρμογής της συνάρτησης κατακερματισμού (SHA1) στη διεύθυνση IP και στο port επικοινωνίας.

Επικοινωνία μέσω HTTP (Flask): Χρησιμοποιήσαμε το Flask, το οποίο δημιουργεί αυτόματα τις αναγκαίες διεργασίες/νήματα και τα TCP sockets. Η επικοινωνία μεταξύ των κόμβων πραγματοποιείται, επομένως, με HTTP requests (GET/POST), διευκολύνοντας τον χειρισμό των αιτημάτων και τις αποκρίσεις τους. Ταυτόχρονα, σε πολλά σημεία δημιουργούμε χειροκίνητα νεα threads, ώστε να εκτελέσουμε παράλληλα κάποιες διαδικασίες.

Λογική "Fire, Forget and Callback": Για την αποστολή αιτημάτων από έναν κόμβο προς έναν άλλο, ακολουθήθηκε ασύγχρονη λογική με callbacks. Συγκεκριμένα, ο κόμβος-αποστολέας "πυροδοτεί" (fire) το αίτημα, χωρίς να μπλοκάρει (forget), ενώ ο τελικός (υπεύθυνος) κόμβος, όταν ολοκληρώσει την εισαγωγή, καλεί το endpoint στον αρχικό κόμβο (callback) για να τον ειδοποιήσει με τα τελικά αποτελέσματα. Στον κώδικα, αυτό υλοποιείται με χρήση pending requests, όπου κάθε αίτημα συνδέεται με ένα μοναδικό request_id. Όταν ληφθεί το callback, το αντίστοιχο νήμα "ξεκλειδώνει" και καταχωρίζει το αποτέλεσμα.

Διαχείριση Δακτυλίου (Chord): Εφαρμόστηκε ένας λογικός δακτύλιος όπου κάθε κόμβος διατηρεί δείκτες στον προηγούμενο και επόμενο κόμβο, προωθώντας τα αιτήματα έως τον υπεύθυνο κόμβο για το αντίστοιχο κλειδί. Η συγκεκριμένη υλοποίηση είναι απλοποιημένη, καθώς δεν περιλαμβάνει finger tables.

2.1.1. Στρατηγική Fire, Forget and Callback

Η επιλογή αυτής της στρατηγικής σε ένα κατανεμημένο περιβάλλον προσφέρει βελτιωμένη απόδοση, αυξημένη ευελιξία και πιο ανεκτική συμπεριφορά σε σφάλματα ή καθυστερήσεις, καθιστώντας την ιδιαίτερα δημοφιλή σε πραγματικά P2P συστήματα.

Η λογική της ασύγχρονης επικοινωνίας και callbacks είναι διαδεδομένη σε κατανεμημένα συστήματα και ειδικά σε εφαρμογές P2P file sharing. Για παράδειγμα, πρωτόκολλα όπως το **BitTorrent** υλοποιούν ασύγχρονες ανταλλαγές μηνυμάτων μεταξύ peers (π.χ. αιτήματα λήψης τμημάτων ενός αρχείου και αποστολή "piece messages"). Αν και δεν χρησιμοποιούν ακριβώς HTTP endpoints, βασίζονται σε παρόμοιες αρχές ασύγχρονης επικοινωνίας: ένας κόμβος ζητά (fire) συγκεκριμένα τμήματα και, μόλις τα λάβει, ενημερώνεται (callback) για την επιτυχή μεταφορά.

2.2. Βασικές Λειτουργίες

- **Εισαγωγή Δεδομένων (insert)** : Η λειτουργία εισάγει ή ενημερώνει ένα <key, value> ζεύγος. Το key (ο τίτλος του τραγουδιού) μετατρέπεται σε hash και το value αντιπροσωπεύει τον κόμβο, όπου στα πλαίσια της εργασίας χρησιμοποιήθηκαν τυχαία string.
- **Διαγραφή (delete)** : Διαγραφή του συγκεκριμένου ζεύγους βάσει του κλειδιού.

- **Αναζήτηση (query)** : Εύρεση του κόμβου που είναι υπεύθυνος για το hash του key ή, σε περίπτωση ειδικού χαρακτήρα "*", επιστροφή όλων των <key, value> ζευγών από ολόκληρο το DHT.
- **Εισαγωγή/Αναχώρηση Κόμβων** : Διαχείριση των εισερχόμενων και αποχωρούμενων κόμβων, με ενημέρωση των δεικτών του δακτυλίου και ανακατανομή των κλειδιών.
- **Replication των Δεδομένων** : Υλοποιήθηκε ο μηχανισμός replication, με εισαγωγή των <key, value> δεδομένων που είναι αποθηκευμένα στο σύστημα. Το replication factor k υποδηλώνει ότι κάθε <key,value> ζεύγος αποθηκεύεται εκτός από τον κόμβο που είναι υπεύθυνος και στους k-1 επόμενους κόμβους στον λογικό δακτύλιο (γίνονται μετρήσεις για διαφορετικές τιμές του k). Το replication λαμβάνεται υπόψη σε όλες τις βασικές λειτουργίες του DHT (insert, delete, query, join, depart).

2.3. Consistency - Είδη Συνέπειας

Linearizability:
Το μοντέλο linearizability διασφαλίζει ότι όλα τα αντίγραφα του κάθε κλειδιού είναι ταυτόχρονα ενημερωμένα και κάθε ανάγνωση επιστρέφει την πιο πρόσφατη τιμή. Στην υλοποίησή μας, αυτό επιτυγχάνεται μέσω chain replication.

- **Insert:** Κατά την εκτέλεση ενός insert, το αίτημα διαδίδεται διαδοχικά μέσω της αλυσίδας των κόμβων. Ο τελικός (tail) κόμβος επιβεβαιώνει την επιτυχή ενημέρωση όλων των αντίγραφων και στη συνέχεια στέλνει το callback στον αρχικό κόμβο, ενημερώνοντάς τον ότι η εισαγωγή ολοκληρώθηκε πλήρως.
- **Delete:** Αντίστοιχα, το delete ακολουθεί την ίδια διαδικασία: το αίτημα διαδίδεται στο chain, και μόνο όταν διαγραφούν όλα τα αντίγραφα αποστέλλεται το callback, επιβεβαιώνοντας την επιτυχή διαγραφή.
- **Query:** Για την ανάγνωση, το σύστημα κατευθύνει το query προς τον tail node, ο οποίος, κατέχοντας το πιο ενημερωμένο replica, εγγυάται ότι επιστρέφεται πάντα η πιο πρόσφατη τιμή.

Eventual Consistency:
Στο μοντέλο του eventual consistency, η αναπαραγωγή των δεδομένων πραγματοποιείται με καθυστέρηση (lazy replication), δημιουργώντας έτσι πιθανότητα ανάγνωσης παλαιότερων τιμών.

- **Insert:** Κατά την εκτέλεση ενός insert, ο αρχικός κόμβος στέλνει άμεσα την απάντηση στον χρήστη χωρίς να περιμένει την ολοκληρωμένη αναπαραγωγή των δεδομένων. Η διαδικασία του replication γίνεται ασύγχρονα, σε ξεχωριστό thread, ώστε να μην επηρεάζει τον χρόνο απόκρισης.
- **Delete:** Για το delete, ισχύει το ίδιο: η διαγραφή επιβεβαιώνεται στον αρχικό κόμβο αμέσως, ενώ η ενημέρωση στα αντίγραφα πραγματοποιείται σε ξεχωριστή διαδικασία.
- **Query:** Όταν εκτελείται ένα query, το σύστημα αναζητά έναν κόμβο που διαθέτει το ζητούμενο entry, είτε αυτό είναι το αρχικό είτε κάποιο αντίγραφο του. Ως αποτέλεσμα, υπάρχει η πιθανότητα να επιστραφεί μια τιμή που δεν είναι πλέον η πιο ενημερωμένη (stale), αν η ασύγχρονη αναπαραγωγή δεν έχει ολοκληρωθεί πλήρως.

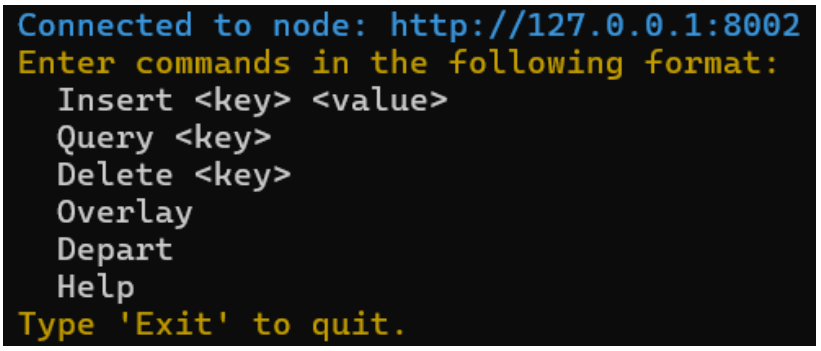
3. Διεπαφή Χρήστη - Client και Frontend

3.1. CLI για Κάθε Κόμβο

Υλοποιήθηκε ένας απλός CLI (Command Line Interface) που αφορά κάθε φορά έναν μόνο κόμβο, όπως συμβαίνει και σε ένα πραγματικό κατανεμημένο σύστημα. Ο χρήστης, μέσω του CLI, έχει τη δυνατότητα να εκτελεί τις εξής εντολές:

- Insert <key> <value> - Εισαγωγή δεδομένων.
- Delete <key> - Διαγραφή δεδομένων.
- Query <key> - Αναζήτηση δεδομένων.
- Depart - Ομαλή αποχώρηση του κόμβου.
- Overlay - Εκτύπωση της τοπολογίας του δικτύου.
- Exit - Έξοδος από το cli.
- Help - Παροχή οδηγιών χρήσης των παραπάνω εντολών.

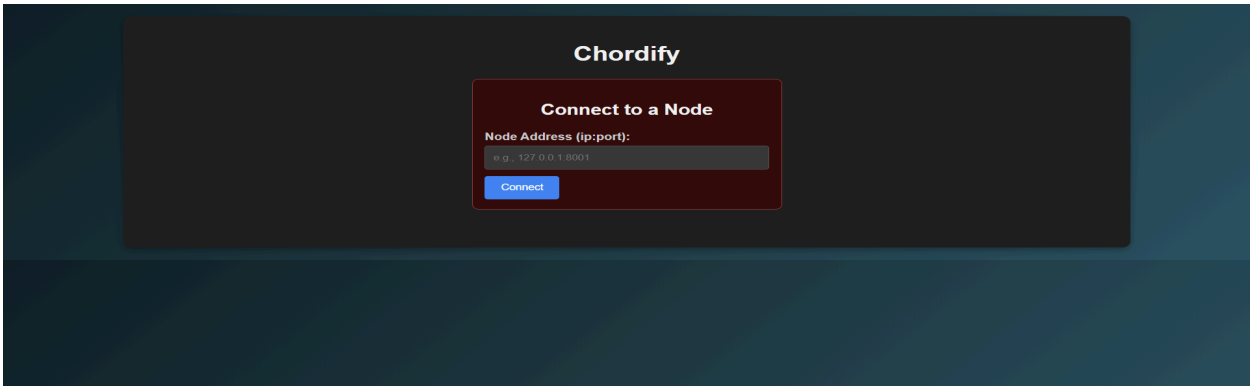
Παραθέτουμε το αρχικό interface από το CLI και ένα παράδειγμα λειτουργίας του μέσα από την εντολή Overlay :



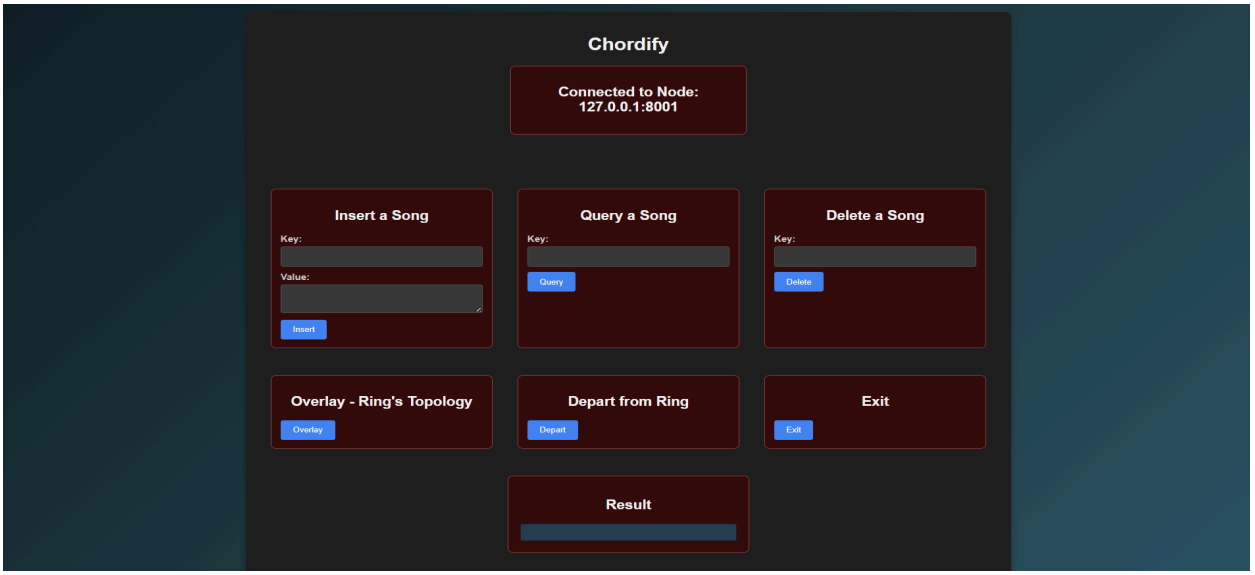
```
[Overlay info]
Node 1:
  ID: 0
  IP: bootstrap
  Port: 8000
  Predecessor: node3:8003
  Successor: node1:8001
----------------------------------------
Node 2:
  ID: 171865491289448819003469989719053766790160449571
  IP: node1
  Port: 8001
  Predecessor: bootstrap:8000
  Successor: node4:8004
----------------------------------------
Node 3:
  ID: 300958178807431293100459636149632972812029243419
  IP: node4
  Port: 8004
  Predecessor: node1:8001
  Successor: node2:8002
----------------------------------------
Node 4:
  ID: 109459246833468211599340679915701582479111365915 5
  IP: node2
  Port: 8002
  Predecessor: node4:8004
  Successor: node3:8003
----------------------------------------
Node 5:
  ID: 115341232355360843340900351654437544568925521453 9
  IP: node3
  Port: 8003
  Predecessor: node2:8002
  Successor: bootstrap:8000
----------------------------------------
```

3.2. Frontend - Web Interface

Πέραν του CLI, δημιουργήσαμε επίσης ένα απλό website (frontend) που παρέχει τις ίδιες λειτουργίες με το CLI. Το frontend υλοποιήθηκε με HTML, CSS και JavaScript. Ο χρήστης αρχικά συνδέεται στο σύστημα μέσα από έναν συγκεκριμένο κόμβο **(πρώτη εικόνα)**, και στη συνέχεια έχει την δυνατότητα να εκτελεί εντολές εισαγωγής, διαγραφής και αναζήτησης τραγουδιών, overlay για την τοπολογία του δικτύου, depart για την αποχώρηση ενός κόμβου από τον δακτύλιο και exit για επιστροφή στην αρχική σελίδα **(2η εικόνα)**. Εκτελώντας οποιαδήποτε λειτουργία λαμβάνουμε και το αντίστοιχο αποτέλεσμα - result **(3η εικόνα)**.



***1η εικόνα** : Είσοδος στο Chordify μέσω συγκεκριμένου Node Address (ip:port)*



***2η εικόνα** : UI Chordify, με όλες τις απαραίτητες λειτουργίες του συστήματος*

3η εικόνα *: Παράδειγμα λειτουργίας Overlay και αντίστοιχο αποτέλεσμα με την τοπολογία του δικτύου*

4. Πειράματα

Για την υλοποίηση των πειραμάτων αξιοποιήσαμε τους πόρους του AWS. Συγκεκριμένα, με τη βοήθεια scripts που υλοποιήσαμε και χρησιμοποιώντας ένα **DockerFile** αρχείο, σε κάθε VM δημιουργήσαμε 2 containers (ένα για κάθε κόμβο), με αποτέλεσμα να έχουμε συνολικά 10 κόμβους στο δίκτυο. Αυτή η προσέγγιση εξασφαλίζει ένα πλήρως κατανεμημένο περιβάλλον, στο οποίο τα αιτήματα (είτε για εισαγωγή είτε για αναζήτηση) εκτελούνται ταυτόχρονα από όλους τους κόμβους. Σε αυτό το περιβάλλον, εκτελέστηκαν τα ακόλουθα πειράματα:

Πείραμα 1: Μέτρηση Write Throughput

Στόχος του πρώτου πειράματος είναι η εισαγωγή όλων των κλειδιών που βρίσκονται στα txt αρχεία που δόθηκαν απο τους διδάσκοντες, ταυτόχρονα από τους 10 κόμβους του συστήματος. Σε αυτό το πείραμα εξετάζονται οι εξής παράμετροι:

- **Replication Factor (k):** Δοκιμάζουμε k=1 (χωρίς replication), k=3 και k=5.
- **Μοντέλα Συνέπειας:** Υλοποιούμε τόσο το μοντέλο **linearizability** όσο και το **eventual consistency**, οδηγώντας σε 6 διαφορετικά setups.

Η υλοποίηση βασίζεται σε ένα κεντρικό script που εκκινεί τα πειράματα. Το script, δημιουργεί ξεχωριστά νήματα (threads) για κάθε κόμβο, όπου σε κάθε νήμα, στελνεται ένα HTTP POST αίτημα στο endpoint που ξεκινά το πειραμα στον κόμβο, δείχνοντας του ποιο txt αρχείο να χρησιμοποιήσει. Ετσι οι εισαγωγές ξεκινούν ταυτόγχρονα σε κάθε κόμβο

Καταγράφεται ο χρόνος απόκρισης κάθε κόμβου και υπολογίζεται το write throughput ως ο λόγος των εισαγόμενων στοιχείων προς τη διάρκεια του αιτήματος.

Πείραμα 2: Μέτρηση Read Throughput

Στο δεύτερο πείραμα μελετάμε το read throughput για τα 6 διαφορετικά setups που δημιουργήθηκαν στο προηγούμενο πείραμα.
Η υλοποίηση είναι παρόμοια με αυτή του προηγούμενου πειράματα, εξασφαλίζοντας οτι κάθε κόμβος θα ξεκινήσει ταυτόγχρονα τις αναζητήσεις (query), διαβάζοντας το σωστό .txt αρχείο.

Καταγράφεται ο χρόνος ολοκλήρωσης όλων των αιτημάτων ώστε να υπολογιστεί το read throughput.

Πείραμα 3: Fresh or Stale Reads

Το τρίτο πείραμα αφορά την εκτέλεση ενός σεναρίου που περιέχει εναλλάξ αιτήματα εισαγωγής και αναζήτησης, όπως ορίζεται στα αντίστοιχα αρχεία .txt. Μας αποσχολει περισσότερο η διαφορά μεταξύ των 2 μοντέλων συνέπειας, οπότε τρέξαμε 2 διαφορετικά setups,χρησιμοποιώντας σε κάθε ένα Replication Factor k=3.
Σε αυτό το πείραμα, κάθε κόμβος διαβάζει το αρχείο του και στέλνει τα αιτήματα στο δίκτυο, καταγράφοντας για κάθε εντολή πληροφορίες όπως ο χρόνος έναρξης και λήξης, ο τύπος της ενέργειας και τα αποτελέσματα που επιστρέφονται. Οπως και στα προηγούμενα πειράματα, τα αιτήματα που διαβάζονται στελνονται στο δίκτυο ταυτόχρονα από όλους τους κόμβους.

Μετά την ολοκλήρωση των αιτημάτων, τα καταγεγραμμένα logs συλλέγονται και ταξινομούνται χρονικά, ώστε να υπολογιστούν οι αναμενόμενες τιμές για κάθε κλειδί και να συγκριθούν με τα αποτελέσματα που επέστρεψαν τα queries. Η σύγκριση αυτή επιτρέπει να προσδιοριστεί αν τα δεδομένα που διαβάστηκαν είναι "ενημερωμένα"(fresh) ή "παρωχημένα" (stale).

5. Αποτελέσματα Πειραμάτων και Συμπεράσματα

Αποτελέσματα (1ο και 2ο Πείραμα)

Στον παρακάτω πίνακα παρουσιάζονται οι μέσες τιμές που μετρήθηκαν για τους 10 κόμβους του συστήματος, τόσο για την περίπτωση **eventual consistency** όσο και για την περίπτωση **linearizability**, με διαφορετικο replication factor. Οι μετρήσεις αφορούν το 1ο και το 2ο πείραμα και συγκεκριμένα τις μετρικές:

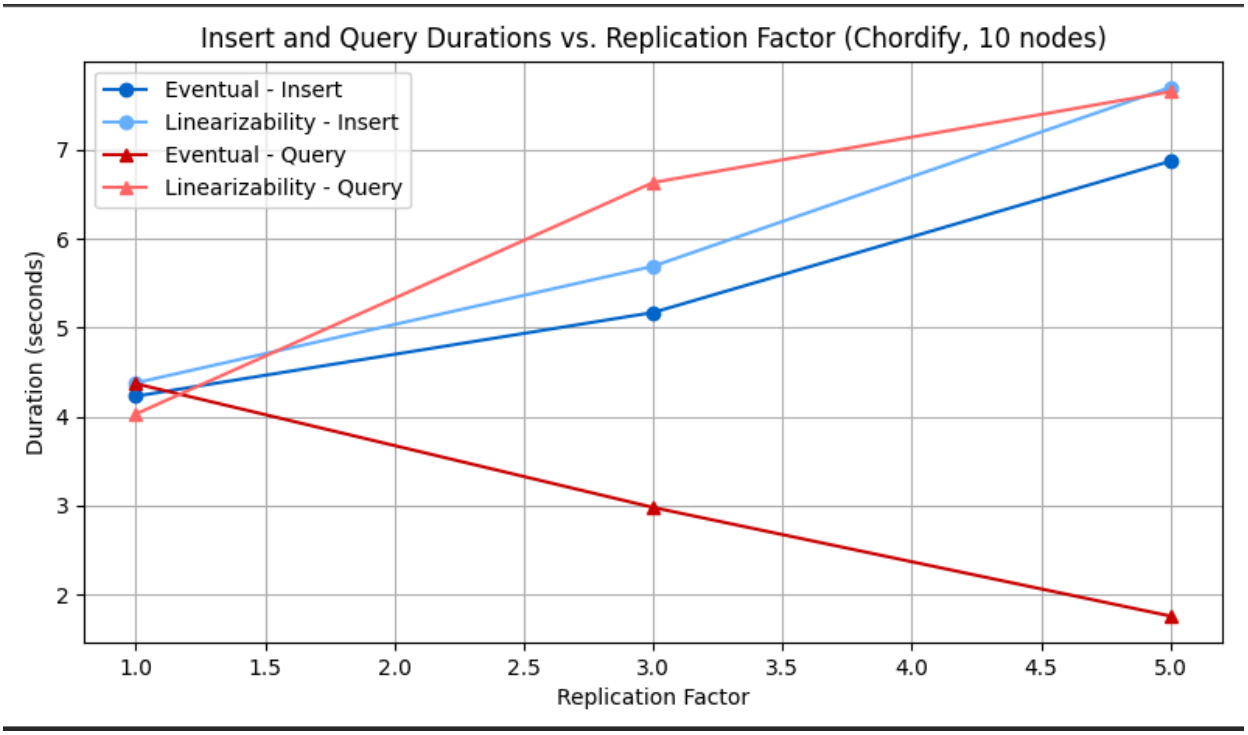
- **Write Throughput (ops/sec):** Ρυθμός εισαγωγής δεδομένων.
- **Insert Duration (sec):** Μέσος χρόνος διεκπεραίωσης των inserts.
- **Read Throughput (ops/sec):** Ρυθμός αναζήτησης δεδομένων.
- **Query Duration (sec):** Μέσος χρόνος διεκπεραίωσης των queries.

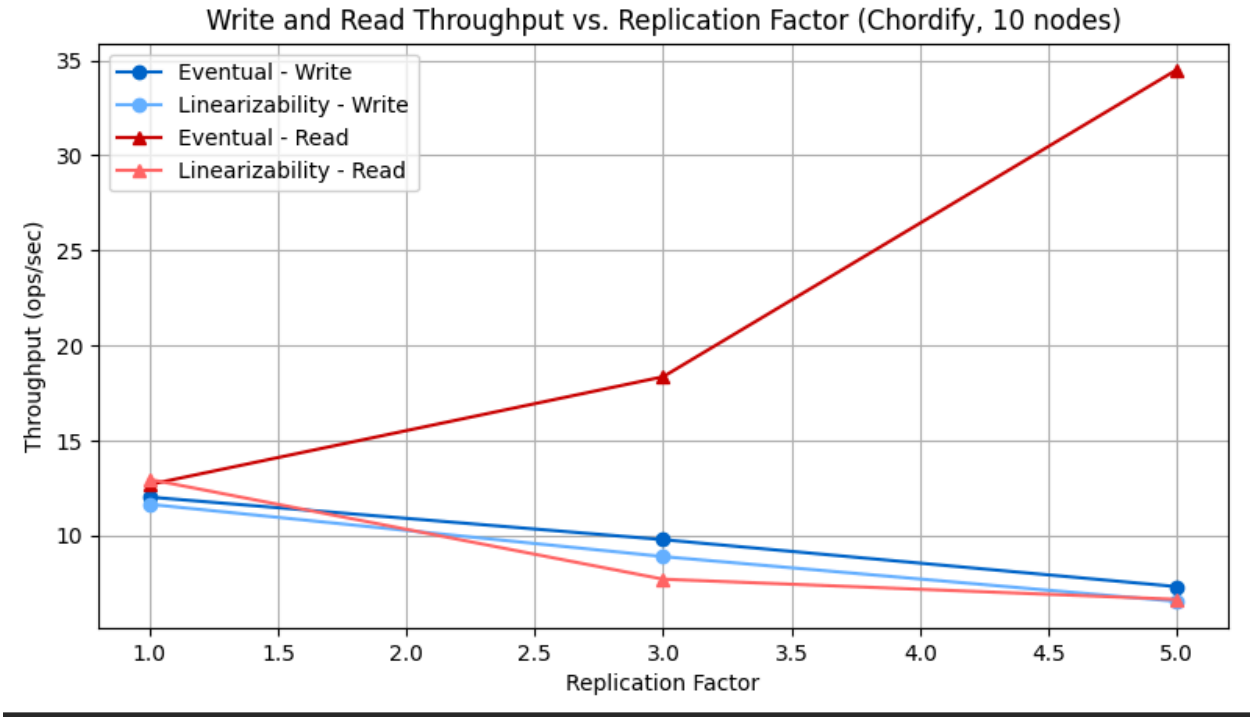
Σημείωση: Τα αναλυτικά αποτελέσματα για κάθε κόμβο για κάθε set up, βρίσκονται σε .csv αρχεία, στο repository της εφαρμογής, στον φάκελο chordify/expirements/results

Πίνακας1

Consistency Mode	Replication Factor	Write Throughput (ops/sec)	Insert Duration se(sec)	Read Throughput (ops/sec)	Query Duration (sec)
Eventual	1	12,03	4,23	12,69	4,37
Eventual	3	9,79	5,17	18,36	2,98
Eventual	5	7,32	6,87	34,51	1,76
Linearizability	1	11,65	4,38	12,96	4,03
Linearizability	3	8,9	5,69	7,71	6,63
Linearizability	5	6,54	7,7	6,65	7,65

Στις επόμενες εικόνες φαίνεται η γραφική αναπαράσταση αυτών των τιμών. Στην πρώτη γραφική απεικονίζεται ο χρόνος εκτέλεσης των inserts και των queries συναρτήσει του replication factor, ενώ στη δεύτερη απεικονίζεται το write και το read throughput για τα ίδια πειράματα.





<u>Παρατηρήσεις (1ο και 2ο Πείραμα)</u>

Χρόνος Διεκπεραίωσης (Insert & Query):

- Στο **insert**, το μοντέλο eventual consistency εμφανίζει σταθερά καλύτερους χρόνους σε σχέση με το linearizability. Η διαφορά μάλιστα μεγαλώνει όσο αυξάνεται το replication factor. Αυτή η παρατήρηση, συμφωνεί με την αναμενόμενη επιβάρυνση του strong consistency που παρεχει το linearazability όταν υπάρχουν περισσότερα αντίγραφα που

πρέπει να ενημερωθούν ή να συγχρονιστούν. Αντίθετα στο μοντέλο eventual consistency, η απάντηση του insert έρχεται κατευθείαν μετά την ενημέρωση του υπευθυνου κόμβου, ενώ η δημιουργία των αντιγράφων εκτελείται απο διαφορετικό thread που δημιουργείται για αυτόν τον σκοπό, σαν ανεξάρτητη διαδικασία.

- Στο **query**, το eventual consistency και πάλι υπερτερεί, καθώς επιτρέπει την ανάγνωση δεδομένων από οποιοδήποτε αντίγραφο (όχι αναγκαστικά απο το tail node). Ειδικά για μεγαλύτερα replication factors, η πρόσβαση σε οποιοδήποτε από τα διαθέσιμα αντίγραφα μειώνει παραπάνω τον χρόνο απόκρισης.
- Για το σύστημα χωρίς καθόλου replication (k=1), οι χρόνοι είναι σχεδόν ίδιοι και στα δύο μοντέλα, όπως αναμενόταν, εφόσον δεν υπάρχει επιπλέον επιβάρυνση από τον συντονισμό πολλαπλών αντιγράφων.

Throughput (Write & Read):

- Οι καμπύλες του write throughput και του read throughput ακολουθούν αντίστοιχα μοτίβα με τους χρόνους διεκπεραίωσης. Το μοντέλο με eventual consistency διατηρεί υψηλότερο throughput, ειδικά σε μεγαλύτερους replication factors, ενώ το linearizability εμφανίζει σημαντική μείωση, εξαιτίας των επιπλέον μηχανισμών συγχρονισμού που απαιτούνται
- Η απουσία replication (k=1) δεν προκαλεί ιδιαίτερες διαφορές μεταξύ των δύο μοντέλων, όμως μόλις εισάγεται επιπλέον replication (k=3 ή k=5), η απόδοση του linearizability πέφτει πιο απότομα συγκριτικά με το eventual consistency.

<u>Αποτελέσματα (3ο Πείραμα)</u>

Στο τρίτο πείραμα, κάθε κόμβος εκτελεί μια ακολουθία από αιτήματα insert και query που βρίσκονται στα αρχεία request.txt. Για κάθε query, καταγράφεται αν η τιμή που επέστρεψε είναι “φρέσκια” (fresh) ή “παρωχημένη” (stale). Στον παρακάτω πίνακα φαίνονται τα συνολικά αποτελέσματα για το σύστημα με **eventual consistency** και με **linearizability**

Πίνακας2

Consistency Mode	Stale Reads	Fresh Reads
Eventual	91	299
Linearizabilty	28	364

Σημείωση: Στο σύνολο των queries (400), υπάρχουν ορισμένα αιτήματα που δεν καταγράφονται ούτε ως fresh ούτε ως stale, διότι το αντίστοιχο τραγούδι δεν υπήρχε ακόμη στο σύστημα. Αυτά τα αιτήματα αποκλείστηκαν από τον παραπάνω πίνακα.

<u>Παρατηρήσεις (3ο Πείραμα)</u>

Παρατηρούμε ότι στην περίπτωση του eventual consistency καταγράφεται σημαντικά μεγαλύτερος αριθμός από stale reads, κάτι που επιβεβαιώνει τη θεωρητική προσέγγιση: επειδή οι ενημερώσεις διαδίδονται “lazy” στα υπόλοιπα αντίγραφα, είναι πιο πιθανό ένα query να λάβει μια πιο παλιά έκδοση των δεδομένων. Ωστόσο, όπως είδαμε και από τα προηγούμενα πειράματα, το eventual consistency ολοκληρώνει τα αιτήματα σε μικρότερο χρόνο συνολικά, ενώ το linearizability, παρότι απαιτεί περισσότερο χρόνο, εμφανίζει λιγότερες περιπτώσεις παρωχημένων τιμών.

Ερμηνεία και Επαλήθευση Θεωρίας:

Τα αποτελέσματα των τριών πειραμάτων, επιβεβαιώνουν τις θεωρητικές προσδοκίες:

Σχετικά με το Eventual Consistency, παρατηρούμε:

- Καλύτερους χρόνους (insert & query)**:** Δεν απαιτείται άμεσος συγχρονισμός μεταξύ όλων των αντιγράφων.
- Περισσότερα stale reads**:** Εξαιτίας του lazy propagation, αυξάνεται η πιθανότητα να διαβαστούν πιο παλιά δεδομένα. Όσο περισσότερες ενημερώσεις εκκρεμούν ή όσο μεγαλύτερος ο αριθμός των αντιγράφων, τόσο αυξάνεται αυτή η πιθανότητα.

Ενώ σχετικά με το Linearizability, παρατηρούμε:

- **Μεγαλύτερη καθυστέρηση (insert & query):** Για κάθε ενημέρωση, ο συντονιστής πρέπει να συγχρονίσει όλα τα αντίγραφα ή να επιβεβαιώσει ότι η εγγραφή ολοκληρώθηκε στο “tail” (στην περίπτωση chain replication), αυξάνοντας σημαντικά τον χρόνο απόκρισης.
- **Λιγότερα stale reads:** Χάρη στις ισχυρές εγγυήσεις συνέπειας, οι τιμές που επιστρέφονται είναι σχεδόν πάντα οι πιο πρόσφατες. Αυτό σημαίνει υψηλότερη ακρίβεια των δεδομένων, αλλά σε βάρος της απόδοσης (throughput) και του χρόνου διεκπεραίωσης.

6. Συμπεράσματα

Τα παραπάνω αποτελέσματα επαληθεύουν ότι το **eventual consistency** ενδείκνυται για συστήματα που στοχεύουν στην υψηλή διαθεσιμότητα και στο scalability (π.χ. spotify), ενώ το **linearizability** επιλέγεται σε περιπτώσεις όπου η ορθότητα των πιο πρόσφατων δεδομένων είναι κρίσιμη (π.χ. τραπεζική εφαρμογή).

Ετσι, η επιλογή μοντέλου συνέπειας εξαρτάται πάντα από τις ανάγκες της εφαρμογής. Εάν μια εφαρμογή απαιτεί τα πιο φρέσκα δεδομένα σε κάθε ανάγνωση (π.χ. τραπεζικές συναλλαγές, κρίσιμα ιατρικά δεδομένα), τότε το linearizability είναι σχεδόν μονόδρομος, παρά το επιπλέον κόστος σε χρόνο και πόρους. Αντιθέτως, εάν η εφαρμογή μπορεί να ανεχθεί περιστασιακά παρωχημένα δεδομένα, αλλά δίνει προτεραιότητα στην ταχύτητα και την ευελιξία (π.χ. κοινωνικά δίκτυα, P2P file sharing, υπηρεσίες streaming), τότε το eventual consistency προσφέρει μεγαλύτερο throughput και καλύτερη απόκριση.

Στην περίπτωσή μας, όπου οι χρήστες μοιράζονται τραγούδια και δεν είναι απαραίτητο να έχουν άμεσα “φρέσκα” δεδομένα (π.χ. την πιο πρόσφατη λίστα peers), το **eventual consistency** είναι η πιο συμφέρουσα επιλογή. Συνδυάζει ταχεία απόκριση και χαμηλή επιβάρυνση στους κόμβους, χωρίς να διακυβεύει την τελική ορθότητα των δεδομένων.