ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ & ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Εξαμηνιαία Εργασία στα Κατανεμημένα Συστήματα

9ο Εξάμηνο, 2024-2025

Chordify - μια P2P εφαρμογή ανταλλαγής τραγουδιών βασισμένη στο Chord DHT

Υλοποίηση από τους φοιτητές:

Λάζου Μαρία-Αργυρώ (el20129) Κατσικόπουλος Κωστής (el20103)

Ομάδα: team_17

Περιγραφή

Σκοπός της εργασίας είναι η σχεδίαση ενός κατανεμημένου συστήματος key-value store βασισμένο στο μοντέλο Chord DHT. Όλες οι μέθοδοι και τα μηνύματα που στέλνει ο client είτε ανταλλάζουν οι κόμβοι μεταξύ τους είναι πλήρως ασύγχρονα και τρέχουν safely μέσα σε multithreaded servers (nodes). Τα δεδομένα κάθε κόμβου αποθηκεύονται σε HashMaps και η προσπέλαση τους γίνεται με την απόκτηση reader-writer locks.

Οι υποστηριζόμενες λειτουργίες ανά κόμβο είναι οι ακόλουθες:
- Join
- Depart
- Insert
- Delete
- Query

Τα ήδη συνέπειας που εξετάστηκαν είναι **Chain Replication** και **Eventual Consistency** , τα οποία διαφοροποιούν τις υλοποίησεις των μεθόδων (Insert, Query, Delete) όπως περιγράφονται παρακάτω.

Join
Για τον προσδιορισμό του ID κάθε νέου κόμβου χρησιμοποείται η συνάρτηση κατακερματισμού SHA-1 πάνω στον συνδυασμό του IP address και του TCP port που ακούει ο κόμβος. Το πρώτο μήνυμα **Join** λαμβάνεται από τον Bootsrap και προωθείται διαδοχικά σε μηνύματα τύπου **FwJoin** μέχρι να βρεθεί ο successor του. Εκείνος με την σειρά του στέλνει ένα μήνυμα **Ack** με τις σωστές πληροφορίες των γειτόνων (previous, successor), τα διαστήματα κλειδιών που διατηρεί αντίγραφα (replica ranges), όλα τα νεά αντίγραφα που θα αναλάβει και τα χαρακτηριστικά του δικτύου (replication factor, consistency mode).

Depart
Για την αποχώρηση κόμβου χρησιμοποιείται ένα μήνυμα **Quit**. Ύστερα, στέλνονται 2 μηνύματα **Update** στους γείτονές του για να ανανεώσουν τα πληροφορίες των previous, successor αντίστοιχα και τα διαστήματα αντιγράφων που είναι υπεύθυνοι.

Replicas Transfer
Κατά τις λειτουργίες Join / Depart, απαιτείται μια αναδίαταξη των αντιγράφων στους k διαδοχικούς κόμβους και η αλλαγή των replica ranges του κάθε κόμβου, τα οποία ρυθμίζονται με μηνύματα τύπου **Relocate**. Για μείωση του overhead, σε κάθε μήνυμα Relocate, κάθε κόμβος μεταφέρει μόνο τα «τελευταία» αντίγραφά του (replica_idx == k) και ανανεώνει τοπικά τα indices όλων των προηγούμενων που δεν είναι γνήσια (0 < replica_idx < k). Τα μηνύματα αυτά, προωθοόυνται αναδρομικά μέχρι να φτάσουν στον τελευταίο replica manager.

Chain Replication

Πρόκειται για σχήμα που εξασφαλίζει strong consistency με linearizability. Οι εγγραφές / διαγραφές πραγματοποιούνται μόνο από τον primary node της αλυσίδας - **head** και οι αναγνώσεις από τον τελευταίο replica manager - **tail**. Για να διαβάζουμε πάντα fresh τιμές απαιτούνται έξτρα μηχανισμοί συγχρωισμού στο head που είναι κρίσιμης σημασίας.
Έστω το εξής προβληματικό σενάριο: ένας πελάτης στέλνει Insert Request σε κάποιον ενδιάμεσο κόμβο i με (0 <= i < k) και αμέσως μετά ένας άλλος στέλνει Query Requst στο tail, πρωτού προλάβει να διαδοθεί η ανανέωση του Insert που προηγήθηκε. Εαν το Query «προχωρήσει» ο πελάτης θα διαβάσει stale data.

Query

Προς αποφυγήν περιπτώσεων όπως η παραπάνω, εισάγουμε το πεδίο pending σε κάθε αντικέιμενο της δομής και προωθούμε όλα τα μηνύματα **Query** στο head. Εαν το pending είναι true, το thread που διαχιρίζεται το εν λόγω Query γίνεται **schedule out** και περιμένει κατάλληλο **signal** από το λειτουργικό για να συνεχίσει την δρομολόγηση προς το tail.

Insert

Σε κάθε μήνυμα τύπου **Insert**, το pending τίθεται σε true και η διάδοση του αντιγράφου προωθείται με **FwInsert** αναδρομικά μέχρι με το tail. Τότε ενημερώνεται ο client με μήνυμα **Reply** και ξεκινούν να στέλνονται προς τα πίσω πάλι αναδρομικά μηνύματα τύπου **AckInsert** με τα οποία κάθε κόμβος ενημερώνει τον πρηγούμενό του για την επιτυχή εισαγωγή. Κάθε κόμβος που λαμβάνει AckInsert μαζί με το αντίστοιχο κλειδί, αλλάζει το pendig σε false και ειδοποιεί όσους readers έχουν μπλοκαριστεί από αυτό.

Delete

Η λογική για την διάδοση ενός **Delete** και την ανανεώση του pending ακολουθεί αυτήν του Insert. Τα μηνύματα που χρησιμποιούνται εδώ είναι **FwDelete, AckDelete** αντιστοίχως. Η μόνη διαφορά έγκειται στο γεγονός ότι κατά το πρώτο Delete το αντικείμενο **διαγράφεται λογικά** (pendig = true) και η **φυσική διαγραφή** έπεται όταν λάβει το ack.

Eventual Consistency

Ο επίσημος ορισμός του Eventual Consistency επιτρέπει την εισαγωγή κλειδιών σε οποιδήποτε replica manager και όχι μόνο στον primary node όπως αναφέρει η εκφώνηση, γεγονός που καθιστά το σύστημα περισσότερο αποδοτικό και scalable. Για τον λόγο αυτό, δίνεται ως επιπλεόν και αυτή η υλοποίηση παρόλο που εισάγει πολύ πιο σύνθετους μηχανισμούς στον κώδικά μας. Το write throughput που πετυχαίνει είναι σαφώς μεγαλύτερο.

Insert / Delete

Τα μηνύματα **Insert**, **Delete**, διαχειρίζονται ισότιμα από τον πρώτο κόμβο που λαμβάνει το request. Εαν αυτός είναι δυνητικά replica manager (ελέγχει εαν το ζητούμενο βρίσκεται σε κάποιο εύρος κλειδιών από όσα του αναλογούν) πραγματοποιεί την ζητούμενη εισαγωγή (ή διαγραφή). Σε αυτήν την περίπτωση, ο χρήστης ενημερώνεται αμέσως για την αλλαγή με μήνυμα τύπου **Reply** και έπειτα διαδίδεται στους υπόλοιπους replica managers με μήνυμα τύπου **FwInsert (FwDelete)** με lazy τρόπο. Διαφορετικά,προωθεί το ίδιο μήνυμα προς την κατεύθυνση του primary node.

Σημείωση 1: Εαν ο κόμβος είναι ενδιάμεσος *(0 < replica_idx < k)* χρειάζεται προώθηση και προς τις 2 κατευθύνσεις οπότε στα μηνύματα τύπου Fw χρησιμοποιούμε το πεδίο forward_back που δηλώνει την κατεύθυνση που πρέπει να προωθηθεί το επόμενο μήνυμα ώστε να αποφύγουμε τον καταιγισμό μηνυμάτων.

Σημείωση 2: Στο απλοποιημένο Eventual που ζητείται απλά προωθείται το μήνυμα **Insert** (ή **Delete**) μέχρι να βρεθεί ο primary node.

Query

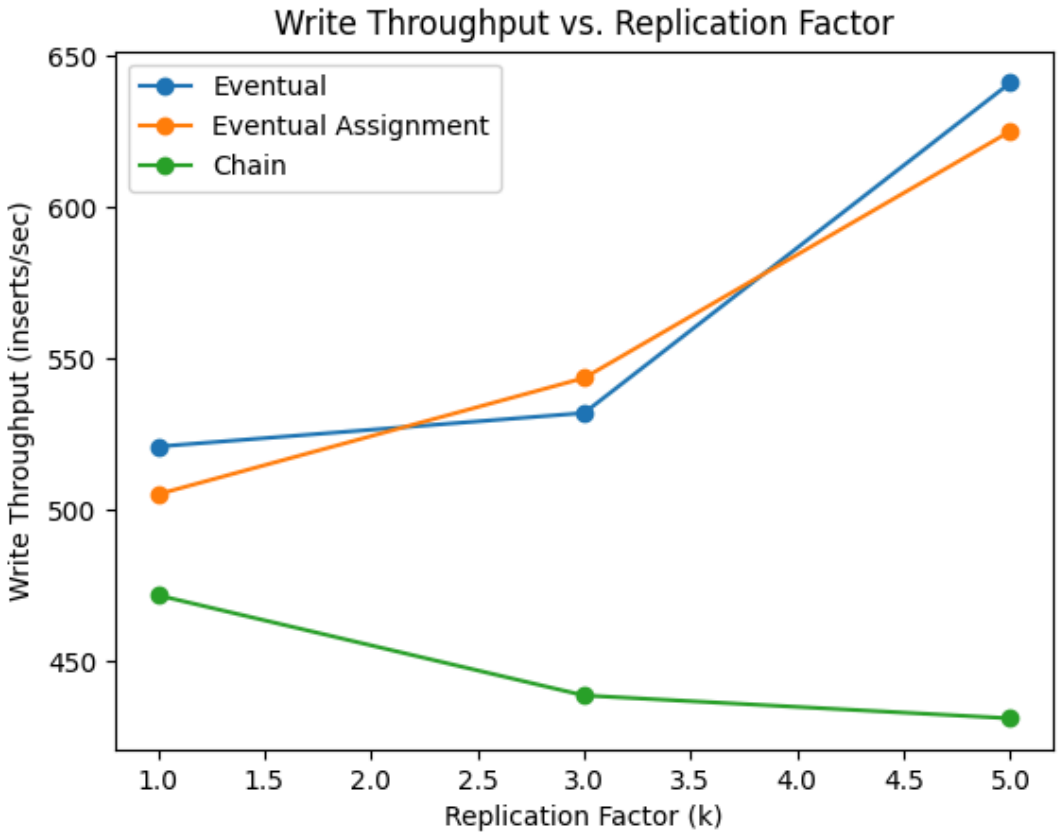
Κάθε κόμβος που λαμβάνει μήνυμα **Query** ελέγχει αν βρίσκεται σε κάποιο εύρος κλειδιών από όσα του αναλογούν και είτε απαντάει με **Reply** είτε προωθεί το Query στην κατεύθυνση του primary node.

Πειράματα

Για την διεξαγωγή των τριών πειραμάτων που ζητούνται, προστέθηκαν στο cli τα options -f στο insert και query καθώς και μία νέα εντολή requests.

Πείραμα 1

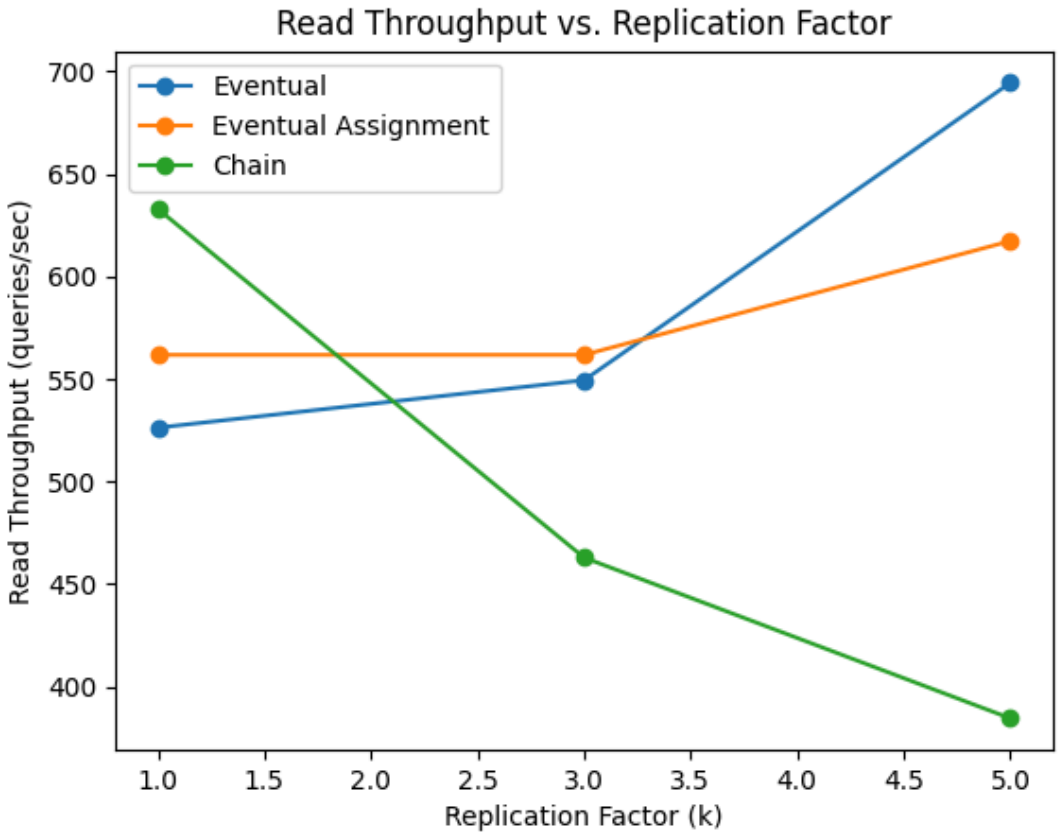
Στο πείραμα 1 ζητείται εκτέλεση 500 εισαγωγών στο σύστημα για παράγοντες replication 1, 3 και 5 και τα 2 είδη consistency που υλοποιούμε. Οι εισαγωγές αυτές έγιναν σειριακά για κάθε κόμβο και ξεκίνησαν παράλληλα από κάθε κόμβο. Μας ζητείται να δημιουργήσουμε το αντίστοιχο διάγραμμα του write throughput με replication factor, που παραθέτεται παρακάτω:



Παρατηρούμε ότι το write throughput για το Eventual Consistency αυξάνεται με την αύξηση του replication factor, γεγονός που είναι αναμενόμενο αφού μπορούν να γίνουν εγγραφές σε οποιονδήποτε replica manager και όχι μόνο στον primary node. Το ίδιο ισχύει και για την απολοποιημένη περίπτωση του Eventual, καθώς η εγγραφή ολοκληρώνεται με την επιτυχία της εγγραφής στον primary node. Αντίθετα για το Chain Replication, το write throughput μειώνεται με την αύξηση του replication factor, καθώς η εγγραφή ολοκληρώνεται με την επιτυχία της εγγραφής σε ολόκληρη την αλυσίδα, που είναι σαφώς πιο χρονοβόρο όσο αυξάνεται το μήκος της αλυσίδας. Προφανώς αυτός είναι και ο λόγος που το write throughput για το Chain Replication είναι πολύ χαμηλότερο από το Eventual Consistency για κάθε k.

Πείραμα 2

Στο πείραμα 2 ζητείται εκτέλεση 500 ερωτημάτων στο σύστημα για παράγοντες replication 1, 3 και 5 και τα 2 είδη consistency που υλοποιούμε. Τα ερωτήματα αυτά, όμοια με το πείραμα 1, έγιναν σειριακά για κάθε κόμβο και ξεκίνησαν παράλληλα από κάθε κόμβο. Μας ζητείται να δημιουργήσουμε το αντίστοιχο διάγραμμα του read throughput με replication factor, που παραθέτεται παρακάτω:



Παρατηρούμε ότι το read throughput για το Eventual Consistency αυξάνεται με την αύξηση του replication factor, γεγονός που είναι αναμενόμενο αφού μπορούν να διαβαστούν από οποιονδήποτε replica manager και όχι μόνο από τον primary node(και στις 2 υλοποίησεις). Αντίθετα για το Chain Replication, το read throughput μειώνεται με την αύξηση του replication factor, καθώς τα queries(όπως και τα inserts) για ένα συγκεκριμένο key, αναμένουν την ολοκλήρωση προηγούμενων inserts σε όλη την σχετική αλυσίδα. Αξιοσημείωτο είναι ότι για k=1 το Chain Replication πετυχαίνει το καλύτερο read throughput, γεγονός που μπορεί να οφείλεται σε ευνοϊκότερη σειρά εκτέλεσης των ερωτημάτων σε συνδυασμό με την απουσία overhead που επιφέρει η ανάγνωση σε όλη την αλυσίδα(0 replicas για k=1).

Πείραμα 3

Στο πείραμα 3 ζητείται εκτέλεση 500 ανάμεικτων αιτημάτων(queries και inserts) για replication factor 3 και τα 2 είδη consistency με σκοπό να παρατηρήσουμε τι συμβαίνει σχετικά με το freshness των δεδομένων.

Eventual Consistency

Παρατηρώντας μερικά από τα αρχεία εξόδου που προκύπτουν από την εκτέλεση των requests, ελέγχουμε για ενδεχόμενες ασυνέπειες. Παρατηρούμε ότι πάρα πολύ συχνά τα αποτελέσματα των queries δεν είναι συνεπή με τα αποτελέσματα των inserts σε ίδιους κόμβους. Συγκεκριμένα είναι πολλές οι περιπτώσεις στις οποίες στον κόμβο με διεύθυνση 10.0.24.219:8002 έχει γίνει Insert μία τιμή για συγκεκριμένο κλειδί και στην συνέχεια το Query που γίνεται για το ίδιο κλειδί επιστρέφει ότι το κλειδί δεν υπάρχει(stale data), πχ:

```
(🔑 Hey Jude : 🔒523) at 🕰 2025-03-17 16:37:00.630927350 UTC successfully!
Error: 🔑Hey Jude doesn't exist
```

Αυτό συμβαίνει γιατί το Query δεν γίνεται στο primary node, αλλά στον τελευταίο replica manager της συγκεκριμένης τιμής, όπου τα δεδομένα είναι πιο παλιά και στην συγκεκριμένη περίπτωση ανύπαρκτα.

Chain Replication

Για το Chain Replication δεν παρατηρούμε κάποια ασυνέπεια που να σπάει το linearizability, κάθε δεδομένο που διαβάζεται έχει την πιο φρέσκια τιμή που έχει εισαχθεί στο σύστημα, δηλαδή έχει timestamp μεγαλύτερο ή ίσο από εκείνο του τελευταίου write και query στο ίδιο κλειδί. Ισοδύναμα να έχει τιμή που περιέχει τις τιμές που είχαν εισαχθεί στον συγκεκριμένο κόμβο για το κλειδί αυτό πρίν αυτό το query καθώς και υπερσύνολα των τιμών που έχουν διαβαστεί στο παρελθόν.

ΥΓ: Ο πηγαίος κώδικας και όλα τα αρχεία με τα logs βρίσκονται στο zip της υποβολής καθώς και στο ακόλουθο Git Repo:
https://github.com/mlazoy/Chord-DHT.git