ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ & ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ
www.cslab.ece.ntua.gr

ΕΞΑΜΗΝΙΑΙΑ ΕΡΓΑΣΙΑ ΣΤΑ ΚΑΤΑΝΕΜΗΜΕΝΑ ΣΥΣΤΗΜΑΤΑ
Ακ. έτος 2020-2021, 9ο Εξάμηνο, Σχολή ΗΜ&ΜΥ

Ορέστης Μπάτσης - Α.Μ.: 03116647

Χρήστος Παπακωστόπουλος - Α.Μ.: 03116662

Ζωή Κόρδα - Α.Μ.: 03116046

ΕΙΣΑΓΩΓΗ

Στην παρούσα εργασία σχεδιάστηκε σε γλώσσα python το «ToyChord», μια file sharing απλοποιημένη εκδοχή του Chord με πολλαπλούς κατανεμημένους κόμβους DHT στους οποίους αποθηκεύονται με συστηματικό τρόπο τραγούδια σύμφωνα με το τίτλο τους. Για την σωστή κατανομή των εν λόγω τραγουδιών έγινε χρήση κατάλληλης hash function, όπως υποδεικνύεται στην εκφώνηση.

Αφού υλοποιήθηκαν οι βασικές λειτουργίες εισαγωγής, διαγραφής, αναζήτησης τραγουδιών και προσθήκης, αποχώρησης κόμβων, εισήχθη και η έννοια του replication. Σύμφωνα με αυτήν, κάθε ζεύγος <key,value> που αντιπροσωπεύει ένα τραγούδι, αποθηκεύεται εκτός από τον κόμβο που είναι υπεύθυνος για αυτό, και στους επόμενους k-1 κόμβους του λογικού δακτυλίου, όπου k: replication factor. Υπό το παραπάνω πλαίσιο, χρειάστηκε να εξασφαλιστεί η συνέπεια των επιμέρους αντιγράφων είτε κατά την μεταβολή της δομής του chord (join/departure), είτε κατά την ανανέωση των στοιχείων του δακτυλίου (insert/delete). Ειδικότερα, υλοποιήθηκαν δύο είδη συνέπειας για τα replicas:

I. *Linearizability (chain replication):*
   Στην περίπτωση αυτή όλα τα replicas έχουν πάντα την ίδια τιμή για κάθε κλειδί, ενώ κάθε query επιστρέφει την πιο πρόσφατη τιμή που έχει γραφτεί. Ειδικότερα, μιας και επιλέχθηκε το chain replication, κατά την εγγραφή μιας νέας τιμής, το write ξεκινά από τον πρωτεύοντα κόμβο και αφού δημιουργηθούν όλα τα replicas στους επόμενους κόμβους, επιστρέφεται το αποτέλεσμα από τον τελευταίο στη σειρά. Ενώ, κατά τη λειτουργεία read, η ανάγνωση της τιμής γίνεται από τον κόμβο που περιέχει το τελευταίο(πιο fresh) replica.

II. *Eventual consistency:*
   Κατά την υλοποίηση αυτού του είδους συνέπειας εξασφαλίστηκε η lazy μετάδοση των αλλαγών στα εκάστοτε αντίγραφα. Πιο συγκεκριμένα, ένα write μεταβαίνει στον κύριο υπεύθυνο κόμβο και επιστρέφει το αποτέλεσμα την εγγραφής. Στη συνέχεια, ο κόμβος αυτός είναι υπεύθυνος να μεταδώσει τη νέα τιμή στους επόμενους k-1 κόμβους. Ενώ, ένα

read διαβάζει από οποιοδήποτε κόμβο βρει τη ζητούμενη τιμή, είτε αυτός είναι ο πρωτεύοντας, είτε περιέχει κάποιο από τα replicas.

ΠΕΙΡΑΜΑΤΑ

1. Σε τοπολογία DHT 10 κόμβων εισήχθησαν τα τραγούδια του αρχείου insert.txt με τυχαίο τρόπο, χρησιμοποιώντας τη συνάρτηση random(). Για κάθε τιμή του replication factor και για κάθε είδος consistency, εκτελέστηκε το ίδιο πείραμα και καταγράφηκε το εκάστοτε write throughput του συστήματος, όπως παρατίθεται παρακάτω.

   - *k = 1 (no replication) :*

     Για k=1 τα αποτελέσματα του chain και eventual consistency θεωρητικά πρέπει είναι ίδια, καθώς δεν υπάρχουν replicas για καθένα από τα τραγούδια. Παρόλα αυτά, παρατηρούνται μικρές διαφοροποιήσεις στα δύο throughputs, οι οποίες προφανώς οφείλονται στην τυχαιότητα με την οποία κάθε φορά πραγματοποιούνται οι εισαγωγές των τραγουδιών στους επιμέρους κόμβους.

     - **linearizability :**

       ```
       ? Test:  1
        Running automated test: insert ...
        0.059702227115631105
        Done!
       ```

       Write throughput = 0.059702227115631105 (sec/key)

     - **eventual consistency :**

       ```
       Select which test you wish to run (1 = insert, 2 = query, 3 = requests)
       ? Test:  1
        Running automated test: insert ...
        0.06282506799697876
        Done!
       ```

       Write throughput = 0.06282506799697876 (sec/key)

   - *k = 3:*

     - **linearizability :**

       ```
       Select which test you wish to run (1 = insert, 2 = query, 3 = requests)
       ? Test:  1
        Running automated test: insert ...
        0.06023351764678955
        Done!
       ```

       Write throughput = 0.06023351764678955 (sec/key)

- **eventual consistency** :

```
Select which test you wish to run (1 = insert, 2 = query, 3 = requests)
? Test:  1
 Running automated test: insert ...
 0.06029791736602783
 Done!
```

Write throughput = 0.06029791736602783 (sec/key)

➢ *k = 5:*

- **linearizability** :

```
Select which test you wish to run (1 = insert, 2 = query, 3 = requests)
? Test:  1
 Running automated test: insert ...
 0.061616645812988828
 Done!
```

Write throughput = 0.061616645812988828 (sec/key)

- **eventual consistency** *:*

```
Select which test you wish to run (1 = insert, 2 = query, 3 = requests)
? Test:  1
 Running automated test: insert ...
 0.058903039455413815
 Done!
```

Write throughput = 0.058903039455413815 (sec/key)

Παρατηρείται ότι στις δύο περιπτώσεις συνέπειας η αύξηση του replication factor(k) επιδρά διαφορετικά στο throughput του συστήματος, δηλαδή στον χρόνο εισαγωγής των ζευγών <key,value> . Πιο αναλυτικά, στην περίπτωση του linearization, εφόσον η τιμή επιστρέφει αφού έχει γραφεί και το τελευταίο replica, η αύξηση του k ,δηλαδή των αντιγράφων, απαιτεί περισσότερο χρόνο για την εισαγωγή κάθε τραγουδιού. Αντίθετα, κατά την εφαρμογή eventual consistency, η τιμή επιστρέφει μετά την πρώτη εγγραφή και επομένως αν και πειραματικά φαίνεται πως η σχέση του replication factor και του throughput, είναι αντιστρόφως ανάλογη, θεωρητικά η τιμή του k δεν θα έπρεπε να επηρεάζει τον χρόνο εγγραφής, μιας και οι αλλαγές στα replicas διαδίδονται lazily.

2. Σε τοπολογία DHT 10 κόμβων αναζητήθηκαν τα τραγούδια του αρχείου query.txt με τυχαίο τρόπο, χρησιμοποιώντας τη συνάρτηση random(). Για κάθε τιμή του replication factor και για κάθε είδος consistency, εκτελέστηκε το ίδιο πείραμα και καταγράφηκε το εκάστοτε read throughput του συστήματος, όπως παρατίθεται παρακάτω.

- *k = 1 (no replication) :*

  Για k=1 τα αποτελέσματα του chain και eventual consistency θεωρητικά πρέπει είναι ίδια, καθώς δεν υπάρχουν replicas για καθένα από τα τραγούδια. Παρόλα αυτά, παρατηρούνται μικρές διαφοροποιήσεις στα δύο throughputs, οι οποίες προφανώς οφείλονται στην τυχαιότητα με την οποία κάθε φορά ζητούνται τα εκάστοτε keys από τους επιμέρους κόμβους.

  - **linearizability :**

    ```
    Select which test you wish to run (1 = insert, 2 = query, 3 = requests)
    ? Test:  2
     Running automated test: query ...
     0.058681703090667725
     Done!
    ```

    Write throughput = 0.058681703090667725 (sec/key)

  - **eventual consistency :**

    ```
    Select which test you wish to run (1 = insert, 2 = query, 3 = requests)
    ? Test:  2
     Running automated test: query ...
     0.06126284646987915
     Done!
    ```

    Write throughput = 0.06126284646987915 (sec/key)

- *k = 3:*

  - **linearizability :**

    ```
    Select which test you wish to run (1 = insert, 2 = query, 3 = requests)
    ? Test:  2
     Running automated test: query ...
     0.06160194778442383
     Done!
    ```

    Write throughput = 0.06160194778442383 (sec/key)

  - **eventual consistency :**

    ```
    Select which test you wish to run (1 = insert, 2 = query, 3 = requests)
    ? Test:  2
     Running automated test: query ...
     0.05760585021972656
     Done!
    ```

    Write throughput = 0.05760585021972656 (sec/key)

- *k = 5:*

  - **linearizability :**

```
Select which test you wish to run (1 = insert, 2 = query, 3 = requests)
? Test:  2
 Running automated test: query ...
 0.06227189445495605
 Done!
```

Write throughput = 0.06227189445495605 (sec/key)

  - **eventual consistency** *:*

```
Select which test you wish to run (1 = insert, 2 = query, 3 = requests)
? Test:  2
 Running automated test: query ...
 0.057075050354003905
 Done!
```

Write throughput = 0.057075050354003905 (sec/key)

Παρατηρείται ότι στις δύο περιπτώσεις συνέπειας η αύξηση του replication factor(k) επιδρά διαφορετικά στο throughput του συστήματος, δηλαδή στον χρόνο αναζήτησης του <key> . Πιο αναλυτικά, στην περίπτωση του linearization, εφόσον η τιμή επιστρέφει αφού έχει βρεθεί και το τελευταίο replica, η αύξηση του k ,δηλαδή των αντιγράφων, απαιτεί περισσότερο χρόνο για την αναζήτηση κάθε τραγουδιού. Αντίθετα, κατά την εφαρμογή eventual consistency, η τιμή επιστρέφει εφόσον βρεθεί το τραγούδι από οποιονδήποτε κόμβο το περιέχει, είτε είναι ο πρωτεύων είτε περιέχει αντίγραφό του. Αυτό συμβαίνει γιατί όσο αυξάνεται το k, δηλαδή όσοι περισσότεροι κόμβοι περιέχουν ένα τραγούδι, τόσο πιο πιθανό να βρεθεί το key σε κάποιον από αυτούς, γεγονός που αποδεικνύεται και πειραματικά. Διαπιστώνεται λοιπόν πως η σχέση του replication factor και του throughput, είναι αντιστρόφως ανάλογη στην περίπτωση του eventual consistency.

3. Εκτελώντας σε τοπολογία 10 κόμβων για k=3, τα requests του αρχείου requests.txt, καταγράφηκαν για τα δύο είδη συνέπειας που έχουν υλοποιηθεί τα αποτελέσματα των queries, τα οποία περιέχονται στα αρχεία 3c_requests.txt και 3l_requests.txt. Μελετώντας τα παραπάνω, παρατηρείται όπως είναι και αναμενόμενο πως εφαρμόζοντας linearizability λαμβάνονται οι πιο πρόσφατα εγγεγραμμένες τιμές και αυτό διότι τόσο η εγγραφή όσο και η ανάγνωση επιστρέφουν από το τελευταίο αντίγραφο. Με άλλα λόγια, δεδομένης της ύπαρξης μιας καθολικής διάταξης, μια λειτουργία read βλέπει τη τελευταία λειτουργία εγγραφής που έχει πραγματοποιηθεί για τη ζητούμενη τιμή. Αντίθετα, το eventual consistency επιτρέπει stale reads, εφόσον οι αναγνώσεις αυτές ανακλούν τιμές που έχουν γραφεί στο παρελθόν, καθώς παρέχει μεγαλύτερη δυνατότητα ταυτοχρονισμού από ό,τι το chain replication, θυσιάζοντας ωστόσο την εγγύηση των πιο fresh τιμών, ενώ σε περίπτωση ταυτόχρονων αιτημάτων απαιτείται η διαχείριση των conflicts που πιθανόν να προκύψουν.