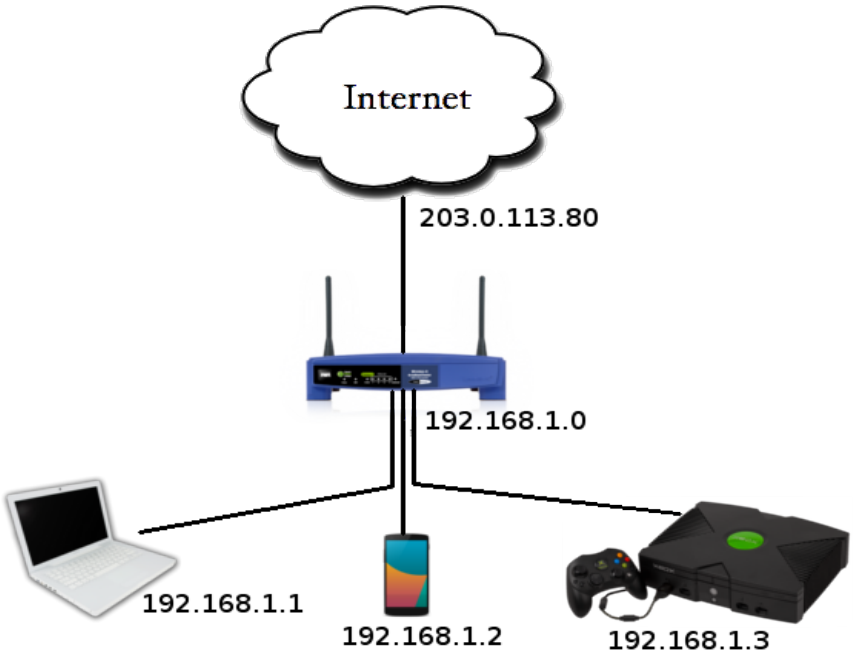
Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών

Ακαδημαϊκό Έτος: 2020-2021



Μάθημα: Κατανεμημένα Συστήματα

Εξάμηνο: 9ο

Εξαμηνιαία εργασία

Θέμα: Σχεδιασμός απλοποιημένης έκδοσης του Chord

Τζε Χριστίνα-Ουρανία | 03116079
Ψαρουδάκης Ανδρέας | 03116001

Εισαγωγή

Σκοπός της παρούσας εργαστηριακής άσκησης είναι η υλοποίηση του ToyChord, μιας απλοποιημένης έκδοσης του Chord [1]. Συγκεκριμένα, η εφαμοργή που αναπτύσουμε είναι ένα file sharing application με πολλαπλούς κατανεμημένους κόμβους DHT. Κάθε κόμβος διατηρεί δείκτες προς τον προηγούμενο και τον επόμενο του λογικού δακτυλίου, προωθώντας το οποιοδήποτε αίτημα στον επόμενο του, μέχρι να φτάσει στον σωστό κόμβο, ο οποίος το επεξεργάζεται και απαντάει απευθείας σε αυτόν που ξεκίνησε το αίτημα. Ένας node λαμβάνει μοναδικό id και υλοποιεί όλες τις λειτουργίες για την επικοινωνία με τους υπόλοιπους, ενώ μπορεί να εκτελεί inserts, queries και deletes δεδομένων. Το σύστημα είναι σε θέση να διαχειρίζεται τόσο εισαγωγές (joins) όσο και αποχωρήσεις (departs) νέων κόμβων ενώ υποστηρίζει και replication για τα δεδομένα. Υλοποιούμε δύο είδη συνέπειας για τα αντίγραφα: 1) linearizability με chain replication και 2) eventual consisteny. Αναπτύσουμε το ToyChord σε γλώσσα Python, πάνω σε υποδομή που μας παραχωρείται από τον ∼ okeanos και εκτελούμε τα πειράματα που ζητούνται στην εκφώνηση.

Πειράματα

- Εισάγουμε σε ένα DHT με 10 κόμβους όλα τα κλειδιά που βρίσκονται στο αρχείο **insert.txt** με k=1 (χωρίς replication), k=3 και k=5 τόσο με linearizability όσο και με eventual consistency (δλδ 6 πειράματα) και καταγράφουμε το **write throughput** του συστήματος (πόσο χρόνο πήρε η εισαγωγή των κλειδιών προς τον αριθμό τους). Σημειώνεται πως τα inserts ξεκινούν κάθε φορά από τυχαίο κόμβο του συστήματος. Τα αποτελέσματα φαίνονται στον ακόλουθο πίνακα:

Write throughput		
Replication factor (k)	**Linearizability**	**Eventual Consistency**
1	0.00677	0.00651
3	0.00931	0.00598
5	0.01591	0.00722

  Για την περίπτωση του **linearizability consistency** με **chain replication**, παρατηρούμε πως όσο **αυξάνει το replication factor (k)** τόσο **αυξάνεται το write throughput**. Το γεγονός αυτό είναι αναμενόμενο καθώς ένα write ξεκινά πάντα από τον πρωτεύοντα κόμβο που είναι υπεύθυνος για ένα κλειδί και προχωρά με τη σειρά στους k-1 υπόλοιπους που έχουν αντίγραφα. Ο τελευταίος κόμβος στη σειρά επιστρέφει το αποτέλεσμα του write. Έτσι, καθώς αυξάνει το k, το μέγεθος της αλυσίδας των αντιγράφων μεγαλώνει, με αποτέλεσμα να εμφανίζεται στο δίκτυο μεγαλύτερο overhead μηνυμάτων για την ενημέρωση κάθε RM, μέχρι και τον τελευταίο.

Αντίστοιχα, για την περίπτωση του **eventual consistency**, οι αλλαγές διαδίδονται lazily στα αντίγραφα. Αυτό σημαίνει ότι ένα write πηγαίνει στον πρωτεύοντα κόμβο που είναι υπεύθυνος για το συγκεκριμένο κλειδί και ο κόμβος αυτός επιστρέφει το αποτέλεσμα του write. Στη συνέχεια, φροντίζει να στείλει τη νέα τιμή στους k-1 επόμενους κόμβους, ενώ παράλληλα είναι και διαθέσιμος να επεξεργαστεί νέα αιτήματα. Είναι λοιπόν εμφανές πως **η μεταβολή του k δεν επηρεάζει το write throughput** καθώς σε κάθε περίπτωση η απάντηση λαμβάνεται από τον primary κόμβο. Επομένως, ο αριθμός των μηνυμάτων που αποστέλλονται στο δίκτυο μέχρι το insert request να φτάσει σε αυτόν, εξαρτάται αποκλειστικά και μόνο από την τυχαία επιλογή του κόμβου, στον οποίο δρομολογείται αρχικά το αίτημα. Συγκεκριμένα, αν ο τυχαίος αυτός κόμβος που επιλέγεται βρίσκεται αρκετά μακριά από τον υπεύθυνο για το συγκεκριμένο κλειδί, τότε αναμένουμε μεγάλο overhead και άρα μεγάλο write throughput. Σε αντίθετη περίπτωση, ο primary κόμβος επιστρέφει πιο γρήγορα το αποτέλεσμα του write και το throughput είναι χαμηλό.

Αν τώρα θεωρήσουμε **μια συγκεκριμένη τιμή του replication factor (k)** και για τα δύο είδη συνέπειας, παρατηρούμε πως για το **linearizability λαμβάνουμε μεγαλύτερη τιμή write throughput**. Η διαφορά τους έγκειται στο ποιος κόμβος είναι εκείνος που επιστρέφει την τιμή του write, όπως εξηγήθηκε παραπάνω.

- Για τα 6 διαφορετικά setups του προηγούμενου ερωτήματος, διαβάζουμε όλα τα keys που βρίσκονται στο αρχείο **query.txt** και καταγράφουμε το **read throughput**. Σημειώνεται ότι τα queries ξεκινούν κάθε φορά από τυχαίο κόμβο. Οι τιμές αναγράφονται στον ακόλουθο πίνακα:

Read throughput		
Replication factor (k)	**Linearizability**	**Eventual Consistency**
1	0.00648	0.00648
3	0.01534	0.00606
5	0.01946	0.00280

Για την περίπτωση του **linearizability consistency** με **chain replication**, παρατηρούμε πως όσο **αυξάνει το replication factor (k)** τόσο **αυξάνεται το read throughput**. Το γεγονός αυτό είναι αναμενόμενο καθώς ένα read πρέπει να διαβάζει την τιμή από τον τελευταίο κόμβο στη σειρά, δηλαδή από τον τελευταίο Replica Manager. Έτσι, καθώς αυξάνει το k, το μέγεθος της αλυσίδας των αντιγράφων μεγαλώνει, με αποτέλεσμα να εμφανίζεται στο δίκτυο μεγαλύτερο overhead μηνυμάτων για το διάβασμα της τιμής από τον τελευταίο κόμβο που κατέχει replica.

Αντίστοιχα, στην περίπτωση του **eventual consistency**, ένα read διαβάζει από οποιονδήποτε κόμβο έχει αντίγραφο του κλειδιού που ζητά, με κίνδυνο να διαβάσει stale τιμή. Όσο μεγαλύτερη είναι η τιμή του k τόσο περισσότεροι κόμβοι του δικτύου κατέχουν ένα αντίγραφο ενός συγκεκριμένου κλειδιού. Συνεπώς, για **μεγάλο replication factor** είναι περισσότερο πιθανό ένα αίτημα query να δρομολογηθεί γρηγορότερα σε κάποιον από αυτούς τους RMs και έτσι το **read throughput μειώνεται**.

Αν τώρα θεωρήσουμε **μια συγκεκριμένη τιμή του replication factor (k)** και για τα δύο είδη συνέπειας, παρατηρούμε πως για το **linearizability λαμβάνουμε μεγαλύτερη τιμή read throughput** όταν υπάρχει replication (k > 1). Η απόκλιση αυτή στις τιμές οφείλεται στην επιλογή του κόμβου από τον οποίο γίνεται ένα read, όπως εξηγήθηκε παραπάνω. Όταν δεν υπάρχουν αντίγραφα των δεδομένων (k=1), δεν υπάρχει κάποια διαφορά μεταξύ των δύο ειδών συνέπειας ως προς το overhead μηνυμάτων στο δίκτυο (και άρα και ως προς το read throughput), καθώς όλα τα αιτήματα read εξυπηρετούνται από τον primary κόμβο του αντίστοιχου κλειδιού.

- Για DHT με 10 κόμβους και k=3, εκτελούμε τα requests του αρχείου **requests.txt**. Στο αρχείο αυτό η πρώτη τιμή κάθε γραμμής δείχνει αν πρόκειται για insert ή query και οι επόμενες τα ορίσματά τους.

  1. Για την περίπτωση του **linearizability consistency** με **chain replication** διαπιστώνουμε πως **κάθε αίτημα query επιστρέφει την πιο πρόσφατη τιμή του write**. Το γεγονός αυτό είναι αναμενόμενο, καθώς πρόκειται για την πιο αυστηρή μορφή συνέπειας, η οποία μας εξασφαλίζει την ίδια τιμή σε όλα τα αντίγραφα ενός συγκεκριμένου κλειδιού (**single copy semantics**). Βέβαια, η αυστηρή αυτή εγγύηση έχει μεγαλύτερο κόστος, όπως διαπιστώσαμε και στα προηγούμενα δύο πειράματα.
  2. Αντίθετα, στην περίπτωση του **eventual consistency** διακρίνουμε ορισμένα αιτήματα query τα οποία επιστρέφουν παλιά (stale) τιμή για ένα συγκεκριμένο κλειδί. Στις ακόλουθες εικόνες παραθέτουμε δύο τέτοια παραδείγματα.

```
Key-value pair (Hey Jude,573) was inserted successfully
Node with id 192.168.0.1:5003 has key Hey Jude with value 572
Node with id 192.168.0.2:5006 has key Hey Jude with value 573
Node with id 192.168.0.5:5010 has key Hey Jude with value 573
Node with id 192.168.0.5:5010 has key Hey Jude with value 573
```

Πράγματι, μετά την εισαγωγή του τραγουδιού Hey Jude με value 573, το πρώτο αίτημα query που πραγματοποιείται επιστρέφει stale τιμή (572 αντί για 573). Τα επόμενα τρία reads διαβάζουν την πιο πρόσφατη τιμή (573).

```
Key-value pair (Like a Rolling Stone,545) was inserted successfully
Node with id 192.168.0.1:5003 has key Like a Rolling Stone with value 536
```

Αντίστοιχα, μετά το insert του key-value pair (Like a Rolling Stone , 545) το πρώτο query request επιστρέφει παλιά τιμή (536 αντί για 545).

Αναφορές

[1] Stoica, Ion, et al. "Chord: A scalable peer-to-peer lookup service for internet applications." ACM SIGCOMM Computer Communication Review 31.4 (2001): 149-160.