ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Κατανεμημένα Συστήματα

Χειμερινό Εξάμηνο 2016-2017

Εξαμηνιαία Εργασία

Ημερομηνία: 17/3/2017

Καλογερόπουλος Δημήτριος (03111179)
Κοτομάτας Φοίβος (03111137)

Περιγραφή της υλοποίησης του DHT

Στόχος του συγκεκριμένου project ήταν ο σχεδιασμός και η υλοποίηση μιας προσομοίωσης της λειτουργίας του DHT (Distributed Hash Table) τύπου δακτυλίου (Chord). Η συγκεκριμένη τοπολογία αποτελείται από κόμβους-servers κάθε ένας από τους οποίους γνωρίζει την ύπαρξη μόνο του επόμενου και του προηγούμενου κόμβου από αυτόν στο δακτύλιο. Ακόμα κάθε κόμβος διατηρεί ένα μόνο τμήμα της συνολικής πληροφορίας που περιέχεται στο δακτύλιο. Έτσι, όταν ένας client συνδέεται με οποιονδήποτε από αυτούς του κόμβους για να κάνει ένα αίτημα, ο κόμβος λαμβάνει το αίτημα και φροντίζει να το διαδώσει σειριακά μέσα στο δακτύλιο μέχρι να φτάσει στον κόμβο υπεύθυνο για το συγκεκριμένο κλειδί του αιτήματος, στη συνέχεια το αίτημα εξυπηρετείται και ο client λαμβάνει την απάντηση από τον κόμβο στον οποίο είχε κάνει την αρχική ερώτηση.

Για την υλοποίηση του παραπάνω δικτύου χρησιμοποιήσαμε τη γλώσσα Java. Ακολουθήθηκαν οι εξής σχεδιαστικές επιλογές:

1) Κάθε κόμβος υλοποιείται από ένα thread το οποίο διατηρεί ένα hashtable (key, value) στο οποίο αποθηκεύεται ένα τμήμα της πληροφορίας του δακτυλίου. Ως hash function για το κλειδί χρησιμοποιήθηκε ο αλγόριθμος SHA1. Η συγκεκριμένη κλάση (Node.java) προβάλλει προς τα έξω διάφορες μεθόδους για την τροποποίηση του hash table.
2) Η επικοινωνία του συστήματός μας βασίζεται στη χρήση sockets μέσω τον οποίων διαδίδονται μηνύματα (Message.java) τα οποία περιέχουν τα δεδομένα του αιτήματος καθώς και πληροφορίες σχετικά με τον αποστολέα, τον παραλήπτη και άλλα τεχνικά στοιχεία. Συγκεκριμένα, κάθε κόμβος ακούει σε ένα συγκεκριμένο socket (Listener.java) και για κάθε αίτημα που φτάνει δημιουργείται ένας νέος εξυπηρετητής (handler.java), ο οποίος ανάλογα με τον τύπο του μηνύματος και τις παραμέτρους του αποφασίζει αν θα το προωθήσει ή αν θα το επεξεργαστεί κάνοντας χρήση του API που παρέχει ο κόμβος.
3) Ακόμα κάθε client υλοποιείται ως ένα ακόμα thread το οποίο συνδέεται με έναν τυχαίο κόμβο και θέτει το αίτημά του. Στη συνέχεια, αναμένει την απάντηση και μόλις τη λάβει την τυπώνει και τερματίζει.

Το σύστημά μας υποστηρίζει και τήρηση αντιγράφων των δεδομένων. Συνεπώς για το χειρισμό τους έχουν υλοποιηθεί δύο είδη consistency. Το πιο χαλαρό είδος eventual consistency επιτρέπει στα writes (insert, delete) να διαδίδονται με το δικό τους ρυθμό στο σύστημα αφού ο client έχει πάρει την απάντησή του. Επίσης, τα reads (query) εξυπηρετούνται από οποιονδήποτε κόμβο έχει το συγκεκριμένο δεδομένο.
Από την άλλη έχουμε και linear consistency (η οποία υλοποιείται με chain replication) κατά την οποία τα write πρώτα διαδίδονται στο δακτύλιο και μόνο στο τέλος ενημερώνεται ο client για την επιτυχία ή όχι του αιτήματός τους. Τα reads εξυπηρετούται από τον τελευταίο στη σειρά ο

οποίος διαθέτει το δεδομένο, έτσι ώστε να διασφαλίζεται ότι πράγματι οι αλλαγές έχουν διαδοθεί πλήρως και δεν υπάρχει πουθενά στο δακτύλιο παλιό αντίγραφο. Για αυτό το είδος consistency έχουμε προσθέσει στον κώδικά μας locks (συγκεκριμένα Reentrant Locks) έτσι ώστε να αποτρέπουμε την ταυτόχρονη προσπέλαση των hashTables από διάφορους handlers και να σειριοποιούμε τα αιτήματα.

Πειράματα

- Εισάγετε 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 και με linearizability και eventual consistency (δλδ 6 πειράματα) και καταγράψτε το write throughput του συστήματος (Πόσο χρόνο πήρε η εισαγωγή των κλειδιών προς τον αριθμό τους). Τα inserts θα ξεκινούν κάθε φορά από τυχαίο κόμβο του συστήματος. Τι συμβαίνει με το throughput όταν αυξάνεται το k στις δύο περιπτώσεις consistency; Γιατί;

Όπως περιγράφηκε και παραπάνω η χρήση linearizability απαιτεί χρήση locks. Συνεπώς εισάγεται μεγάλη καθυστέρηση και μειώνεται και η δυνατότητα παραλληλοποίησης των αιτημάτων. Όπως φαίνεται και από τους παρακάτω χρόνους, και όπως είναι και αναμενόμενο, όσο αυξάνεται το k, ταυτόχρονα αυξάνεται και το πλήθος των κόμβων οι οποίοι πρέπει να κλειδώσουν για να ικανοποιηθεί έστω και ένα αίτημα. Συνεπώς, στην περίπτωση του linearizability το throughput μειώνεται αισθητά όσο αυξάνεται το k.
Αντίθετα, με χρήση eventual consistency δεν έχουμε επιβάλλει κανέναν περιορισμό στα writes που γίνονται στο σύστημά μας. Συνεπώς, μόλις ολοκληρωθεί στον υπεύθυνο κόμβο το αίτημα, απαντάμε στον client και τα replications διαδίδονται αργά. Παρατηρούμε λοιπόν ότι το throughput δεν επηρεάζεται ιδιαιτέρως. Συγκεκριμένα, η πτώση που παρατηρείται οφείλεται κυρίως στην κίνηση που εισάγεται στο δίκτυο αφού αυξάνεται κατά πολύ ο αριθμός των μηνυμάτων που στέλνονται μεταξύ των nodes.

Consistency / k	**1**	**3**	**5**
linearizability	3572	4442	5208
eventual	3155	4089	3999

Write time(ms)

Consistency / k	1	3	5
linearizability	139.98	112.56	96.01
eventual	158.48	123.30	125.03

Write Throughput(inserts/sec)

- Για τα 6 διαφορετικά setups του προηγούμενου ερωτήματος, διαβάστε όλα τα keys που βρίσκονται στο αρχείο query.txt και καταγράψτε το read throughput. Τα queries ξεκινούν κάθε φορά από τυχαίο κόμβο. Ο κόμβος ελέγχει αν έχει το κλειδί ή αντίγραφό του, αλλιώς προωθεί το query στον επόμενο. Τι γίνεται όσο το k αυξάνεται; Γιατί;

Η δική μας υλοποίηση του linearizability έχει σχεδιαστεί έτσι ώστε ένα query να προωθείται πάντα στον responsible node για το συγκεκριμένο εύρος key και στη συνέχεια από εκεί να προωθείται στον τελευταίο κόμβο που διαθέτει replication και αυτός να απαντάει (όπως ζητείται από την εκφώνηση). Συνεπώς, το σύστημά μας έχει μεγάλο overhead μέχρι το αίτημα να φτάσει στον υπεύθυνο κόμβο. Ίσως να μπορούσαμε να σχεδιάσουμε ένα καλύτερο σύστημα το οποίο θα κρατούσε και version στα replication hashtables έτσι ώστε να ξέρει αν ένας κόμβος μπορεί να απαντήσει στον client ή όχι, αλλά το θεωρήσαμε υπερβολική πληροφορία. Παρόλα αυτά, το read στο linearizability είναι σίγουρα πιο αργό από το eventual διότι πρέπει να διασφαλίσουμε ότι επιστρέφεται η τιμή του τελευταίου που διαθέτει αντίγραφο.Συνεπώς για την περίπτωση του linearizability το read throughput όπως και προηγουμένως το write throughput μειώνεται όσο αυξάνεται το k εξαιτίας των προϋποθέσεων που χρειάζονται για να ικανοποιηθεί το αίτημα. Από την άλλη στο σύστημα με το eventual consistency υπάρχει μεγάλη βελτίωση του read throughput όσο αυξάνεται το k το οποίο είναι αναμενόμενο αφού για κάθε query που φτάνει σε έναν κόμβο αυτός πάντα ελέγχει και τα replications που διαθέτει πριν προωθήσει το μήνυμα. Άρα όσο μεγαλύτερο το k τόσο μεγαλύτερη η πιθανότητα να το βρει.

Consistency / k	1	3	5
linearizability	2009	2514	2918
eventual	2133	1800	1305

Read Time(ms)

Consistency / k	1	3	5
linearizability	248.88	198.89	171.35
eventual	234.41	277.78	383.14

Read Throughput(ms)

- Για DHT με 10 κόμβους και k=3, εκτελέστε 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. Καταγράψτε τις απαντήσεις των queries σε περίπτωση linearization και eventual consistency. Ποια εκδοχή μας δίνει πιο fresh τιμές;

Όπως έχει ήδη αναφερθεί το tradeoff μεταξύ linearizability και eventual consistency είναι μεταξύ του χρόνου απόκρισης του συστήματος και της δυνατότητας διάδοσης των αλλαγών σε αυτό. Συνεπώς, όταν έχουμε linearizability μας επιστρέφεται πάντα η πιο fresh τιμή στο σύστημα αλλά αυτή μπορεί να καθυστερήσει, ενώ όταν έχουμε eventual consistency έχουμε πολύ πιο γρήγορες απαντήσεις, αλλά μπορεί να μην επιστρέψει το query την πιο πρόσφατη τιμή που έχει γίνει write.

Τα αποτελέσματα από τις δύο διαφορετικές εκτελέσεις βρίσκονται στα αρχεία requests_eventual.txt και requests_linear.txt που περιέχονται στο zip της παρούσας αναφοράς.