Distributed Systems
ToyChord

1. Εισαγωγή

Αυτή η εργασία είναι μια απλή υλοποίηση του ToyChord [1] σε Python, με χρήση του Flask Rest API, με δύο τρόπους συνέπειας, chain replication [2] και eventual consistency, όπως αναφέρονται στην εκφώνηση του project. Τα finger tables δεν υλοποιήθηκαν. Η εφαρμογή αποτελείται από 4 αρχεία, το dht_server.py που είναι το κύριο πρόγραμμα που γίνονται τα περισσότερα services των κόμβων όπως η επικοινωνία τους, ο διαμοιρασμός αρχείων και replicas κ.τ.λ., το dht_node.py που είναι το module όπου αποθηκεύονται τα αρχεία και οι πληροφορίες του κάθε κόμβου, το cli_receiver.py που είναι το backend του command line interface, δηλαδή εκεί έχουμε τις functions που καλούνται από το cli για την κάθε λειτουργία που απαιτείται, και το cli.py που είναι ουσιαστικά το frontend.

2. Πειράματα

Τα πειράματα έγιναν στο okeanos, όπου είχαμε έναν master κόμβο ο οποίος έχει σύνδεση στο διαδίκτυο, και πάνω σε αυτόν στήσαμε ένα local δίκτυο με άλλους 4 workers. Συνδεόμαστε στον master κόμβο με απομακρυσμένη σύνδεση ssh, και μέσω του master κόμβου μέσω του τοπικού δικτύου μπορούμε και κάνουμε ssh στους workers. Έτσι, με 5 διαφορετικά μηχανήματα, σηκώνουμε 5 κόμβους σε ένα port και άλλους 5 σε ένα διαφορετικό port ώστε να έχουμε συνολικά 10 κόμβους. Έπειτα, αφού τρέχει το dht_server.py σε 10 κόμβους, σηκώνουμε το cli_receiver.py καθώς και το cli.py για να έχουμε το command line interface.

Σε όλες τις περιπτώσεις, τρέξαμε τα πειράματα μερικές φορές (3-4) και πήραμε τον μέσο όρο τους. Η χρονομέτρηση δεν γίνεται με μεγάλη ακρίβεια, απλώς τυπώνουμε την ώρα κάθε φορά και κάνουμε την αφαίρεση.
1. 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;

2. 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;

Παρακάτω φαίνονται οι δύο πίνακες με τους χρόνους εκτέλεσης σε δευτερόλεπτα καθώς και το throughtput που είναι seconds ανά key. Και στο insert.txt και στο query.txt περιέχονται 500 κλειδιά.

1. Όσον αφορά το linearizability, υλοποιήθηκε μόνο το **chain replication**, παρατηρούμε ότι ο χρόνος εισαγωγής των 500 κλειδιών παραμένει σταθερός ανεξάρτητα από το replication factor. Αυτό εξηγείται από το γεγονός ότι το σύστημά μας είναι σχετικά γρήγορο επειδή τα inserts γίνονται ασύγχρονα στους κόμβους, και ότι 500 κλειδιά είναι μικρός αριθμός για να παρατηρήσουμε διαφορές. Επίσης, ο δακτύλιος του DHT έχει μόνο 10 κόμβους.
   Όσον αφορά το eventual consistency συμβαίνει το ίδιο, με ελάχιστη αύξηση η οποία δεν είναι σημαντική και πιθανόν να ευθύνεται σε ανακρίβεια κατά την χρονομέτρηση. Και σε αυτή την περίπτωση είναι σταθεροί οι χρόνοι ανεξάρτητα από τον replication factor.
2. Για το linearizability, το read throughput αναμένουμε να αυξάνεται καθώς αυξάνει το replication factor καθώς λόγω του chain replication, αναμένουμε την απάντηση του τελευταίου κόμβου. Αυτό παρατηρείται στο σύστημά μας, όπου υπάρχει μια αύξηση 2 δευτερόλεπτων για κάθε αύξηση του replication factor που ζητήθηκε.
   Στο eventual consistency αναμένουμε το αντίθετο, καθώς σε αυτή την περίπτωση replication απαντάει ο πρώτος κόμβος και στην συνέχεια το προωθεί στους υπόλοιπους, όπως και παρατηρούμε μια μείωση στο χρόνο απάντησης του query.

k	1	3	5	Consistency Type
write throughput	$4sec$	$4sec$	$4sec$	Linearizability
read throughput	$16sec$	$18sec$	$20sec$	Linearizability
write throughput	$4sec$	$5sec$	$5sec$	Eventual Consistency
read throughput	$17sec$	$16sec$	$15sec$	Eventual Consistency

k	1	3	5	Consistency Type
write throughput (sec/key)	0.008	0.008	0.008	Linearizability
read throughput (sec/key)	0.032	0.036	0.040	Linearizability
write throughput (sec/key)	0.008	0.010	0.010	Eventual Consistency
read throughput (sec/key)	0.034	0.032	0.030	Eventual Consistency

3. 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;

Στην περίπτωση αυτού του πειράματος, κάθε request περιμένει απάντηση και μετά γίνεται το επόμενο request, ώστε αυτά να εκτελούνται σειριακά. Αναμένουμε το linearizability να μας δίνει πάντα τις πιο fresh τιμές, χωρίς να υπάρχει περίπτωση να πάρουμε stale αντίγραφο, καθώς γίνεται propagate κάθε

query μέχρι να απαντήσει ο τελευταίος κόμβος. Στο eventual consistency, αντιθέτως, υπάρχει περίπτωση να πάρουμε κάποιο stale αντίγραφο και είναι το κόστος που πρέπει να πληρώσουμε ώστε το σύστημά μας να είναι ταχύτερο. Αυτό συμβαίνει διότι κάθε query παίρνει απάντηση από τον πρώτο κόμβο και στην συνέχεια προωθείται στους υπόλοιπους.
Στην περίπτωσή μας, δεν βρήκαμε κάποιο stale αντίγραφο και τα δύο αρχεία είναι consistent μεταξύ τους και αυτό οφείλεται στο γεγονός ότι έχουμε μικρό αριθμό κόμβων και μικρό αριθμό tests. Για να το ελέγξουμε διεξοδικότερα αυτό θα μπορούσαμε να έχουμε λόγου χάρη 50 κόμβους, με έναν πολύ μεγαλύτερο αριθμό requests και queries ώστε να υπάρχει μεγαλύτερη πιθανότητα ένα query να επιστρέψει stale αντίγραφο και να παρατηρήσουμε το ελάττωμα αυτό του eventual consistency που όμως το κάνει ταχύτερο από ότι η ισχυρή συνέπεια του Linearizability.

References

---

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 ↩
2. Van Renesse, Robbert & Schneider, Fred. (2004). Chain Replication for Supporting High Throughput and Availability... 91-104. ↩