03117408– ΡΟΥΣΣΗΣ ΔΗΜΗΤΡΙΟΣ

03116051 – ΣΟΥΛΙΩΤΗΣ ΠΑΝΑΓΙΩΤΗΣ

Αναφορά Εξαμηνιαίας Εργασίας στα Κατανεμημένα Συστήματα

Εισαγωγή

Σκοπός της παρούσης εργασίας είναι η δημιουργία μιας απλοποιημένης μορφής ενός Chord DHT. Το DHT επιτελεί λειτουργίες join και depart κόμβων και ο κάθε κόμβος παρέχει υπηρεσίες insert, query, delete για αντίστοιχα προσθήκη, εύρεση και διαγραφή αρχείου από το κατανεμημένο σύστημα αποθήκευσης αρχείων. Παρέχεται επίσης η δυνατότητα για replication των αρχείων που διατηρεί ένας κόμβος στους K-1 επόμενους του. Υπάρχουν τέλος 2 στρατηγικές για την διαχείριση των replicas : eventual και linear.

Περιγραφή υλοποίησης συστήματος

Η υλοποίηση έγινε σε γλώσσα Python και οι κόμβοι επικοινωνούν μεταξύ τους μέσω sockets. Ο κάθε κόμβος είναι ένα διαφορετικό (server.py) process που ακούει σε συγκεκριμένο socket τα διάφορα αιτήματα. Όταν σηκώνεται ο κάθε κόμβος δημιουργεί ένα client thread που περιμένει να του έρθουν αιτήματα τα οποία στην συνέχεια θα επεξεργαστεί και θα απαντήσει ή θα προωθήσει.

Οι κόμβοι είναι οργανωμένοι σε λογικό δακτύλιο, όπου ο καθένας μαθαίνει τον επόμενο και τον προηγούμενό του κατά την είσοδο του στο σύστημα και λαμβάνει ένα μοναδικό id.

Η υποδομή που χρησιμοποιήσαμε για το project προσφέρεται από την υπηρεσία ~okeanos-knossos. Έχουμε δημιουργήσει 5 VMs (Ubuntu Server LTS), καθένα από τα οποία έχει 2 cores, 2GB μνήμης και 30GB δίσκου. Έχουμε δημιουργήσει επίσης ένα private network στο οποίο συνδέονται όλα τα VMs και επικοινωνούν μεταξύ τους χρησιμοποιώντας private διευθύνσεις. Η σύνδεση μας με τον master (node0) γίνεται με ssh, καθώς τον έχουμε συνδέσει με μία public IP (83.212.72.100).

Δημιουργία κόμβων

Κατά την δημιουργία ενός κόμβου και σύνδεση του στο DHT (αποστολή αντίστοιχου αιτήματος στον master node του συστήματος) λαμβάνει ένα μοναδικό ID και δίνει στον master τα στοιχεία του (IP και Port που ακούει). Το ID αυτό χρησιμοποιείται μόνο στο depart του κόμβου ώστε ο master να τον αφαιρέσει από την λίστα των συνδεδεμένων κόμβων . Το id του κόμβου με το οποίο γίνονται οι υπόλοιπες λειτουργίες είναι το sha1("self.host:self.port") όπου self.host είναι η IP του κόμβου και self.port είναι η θύρα που ακούει.

Ο πρώτος κόμβος που δημιουργεί το DHT θεωρούμε ότι είναι ο master, ο οποίος δεν αποχωρεί ποτέ εκτός αν γίνει destroy ολόκληρο το DHT, μέσω κατάλληλης εντολής. Ο master είναι και αυτός που διαχειρίζεται αιτήματα join και depart.

Αιτήματα insert, delete, query στέλνονται σε τυχαίο κόμβο του δικτύου και ο κόμβος που απαντά διαφέρει ανάλογα με το είδος του consistency.

Οι εισαγωγές-αποχωρήσεις κόμβων έχουν σχεδιαστεί ώστε μετά την ολοκλήρωση της λειτουργίας αυτής να διατηρείται το σωστό replication για όλες τις εγγραφές. Είσοδοι και έξοδοι κόμβων επιτρέπονται ένας μετά τον άλλον, ενώ θεωρούμε πως οι κόμβοι εισέρχονται και εξέρχονται οικειοθελώς.

CLI

Το cli δημιουργείται στον master και δίνει στον χρήστη τις δυνατότητες που περιγράφονται στην εκφώνηση και φαίνονται στον παρακάτω πίνακα.

```
depart,ID                       : Node with ID departs from the DHT
insert,key,value                : Insert key,value pair in DHT
delete,key                      : Delete key from DHT
query,key                       : Query key,value pair in DHT
insert_from,Node_ID,key,value : Insert key,value in DHT starting from nodeID
delete_from,Node_ID,key         : Delete key from DHT starting from nodeID
query_from ,Node_ID,key         : Query key,value pair in DHT starting
                                  from nodeID
insertfile,insert.txt           : Inserts file insert.txt in DHT
queryfile,query.txt             : Queries file query.txt in DHT
parsefile,requests.txt          : Parses file requests.txt in DHT
help                            : Prints commands of DHT cli
overlay                         : Prints the DHTs overlay
exit                            : Destroys DHT
```

Consistency

Στην συνέχεια θα παρουσιάσουμε τις 2 στρατηγικές που υλοποιήσαμε:

Για **linear consistency** χρησιμοποιήσαμε το chain replication. Πιο συγκεκριμένα τα insert ξεκινάνε από τον κόμβο που ανήκουν , ως προς το hashed key τους. Η εισαγωγή-ενημέρωση μιας εγγραφής τερματίζει με την εγγραφή του τελευταίου κόμβου. Αντίθετα για query, επιστρέφει το value ο κόμβος που έχει το τελευταίο replica. Έτσι, εξασφαλίζεται ότι όλα τα replicas έχουν πάντα την ίδια τιμή.

Για **eventual consistency** θεωρούμε ότι το DHT κάνει τα insert,delete lazily δηλαδή όταν ένας κόμβος λάβει ένα τέτοιο μήνυμα απαντάει σε αυτόν που του έστειλε το αίτημα και μετά το προωθεί. Σχετικά με το query, μπορεί οποιοσδήποτε κόμβος που έχει τα δεδομένα ή replicas αυτών να απαντήσει, με πιθανότητα τα δεδομένα να είναι stale, σε περίπτωση που δεν έχει προλάβει το σύστημα μετά από insert να ενημερώσει όλα τα replicas με την νέα τιμή.

Πειράματα

Δημιουργήσαμε ένα δίκτυο με 10 κόμβους σε σταθερή IP και port ώστε να διατηρείται η τοπολογία του δικτύου. Εκτελέσαμε τα πειράματα για replication size K = 1 (χωρίς replication) , K = 3 και K = 5 και με τις 2 στρατηγικές για consistency. Τα αποτελέσματα παρουσιάζονται στον παρακάτω πίνακα:

Throughput (msec/key)	insert.txt		query.txt	
	Eventual	Linear	Eventual	Linear
K = 1	6.88	6.7	6.94	6.98
K = 3	9.9	9.95	4.6	9.5
K = 5	8.96	11.66	3.6	11.17

Παρατηρούμε ότι με την αύξηση του K έχουμε αύξηση του χρόνου που απαιτείται για τα inserts για linearizability ενώ για eventual φαίνεται να συγκλίνει προς μία τιμή, μιας και η εισαγωγή των replicas διαδίδεται lazily στο σύστημα.

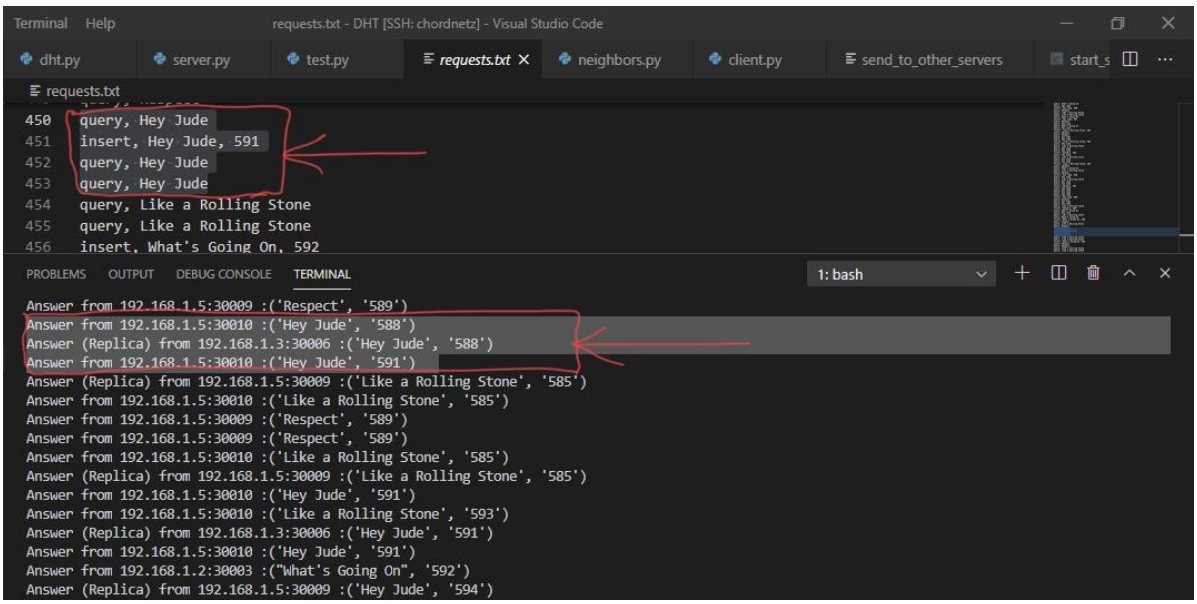
Συγκρίνοντας τους χρόνους των inserts μεταξύ eventual και linear παρατηρούμε ότι σε όλες τις περιπτώσεις (εκτός του K=1 που δεν έχουμε replicas) γίνονται γρηγορότερα για eventual σε σχέση με linear αφού επιστρέφονται τα αποτελέσματα του write απευθείας ενώ στο linear πρέπει να εγγραφούν πρώτα όλα τα replicas. Για K = 1, δηλαδή χωρίς replicas θα περιμέναμε να έχουν ίδιο χρόνο. Εντούτοις, η διαφορά είναι πολύ μικρή.

Όσον αφορά τα queries, για eventual consistency γίνονται όλο και γρηγορότερα με την αύξηση του αριθμού των replicas μιας και οποιοσδήποτε από τους κόμβους που έχουν τα δεδομένα ή replicas αυτών μπορούν να απαντήσουν. Αντίθετα στη περίπτωση του linearizability, ο χρόνος ανά κλειδί αυξάνει αφού πρέπει να απαντήσει ο τελευταίος κόμβος που έχει τα replicas. Όπως ήταν αναμενόμενο, μεταξύ linear και eventual απαντάει πιο γρήγορα στα queries η στρατηγική με eventual.

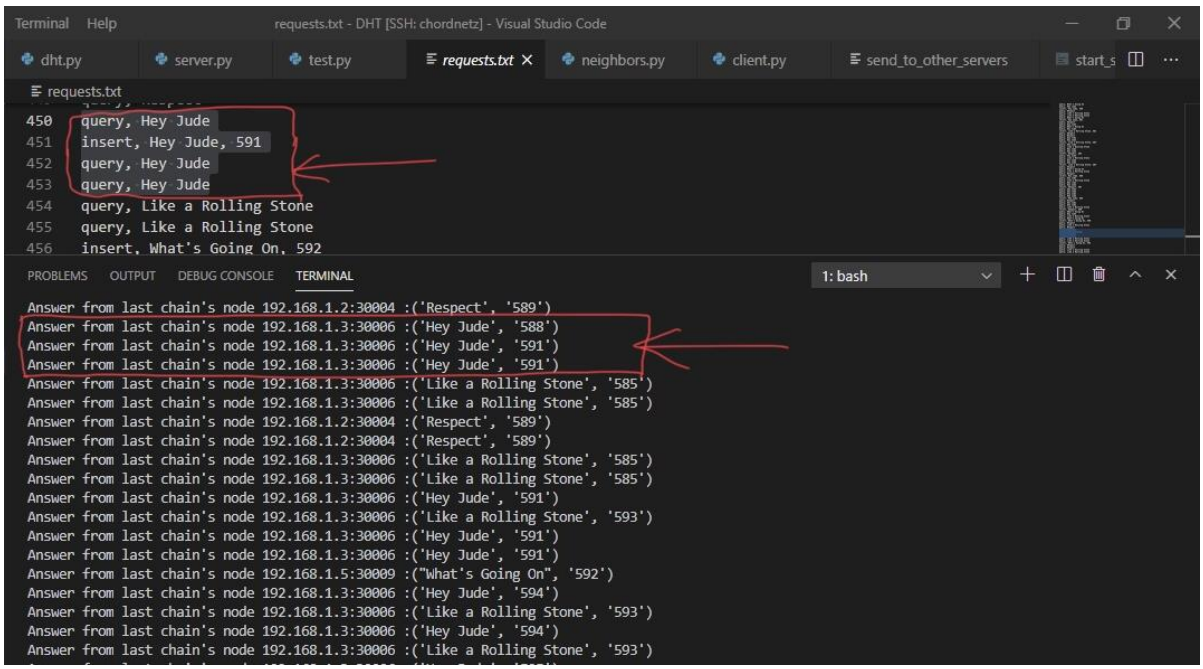
Τέλος όσον αφορά το αρχείο requests.txt που περιέχει insert και query αιτήματα καταγράψαμε τις απαντήσεις των κόμβων στα αρχεία “requests_eventual_3.txt” και “requests_linear_3.txt”, που βρίσκονται μέσα στο tarball που έχει κατατεθεί.

Παρατηρούμε ότι πιο fresh τιμές μας δίνει πάντα το linearization.

Όπως βλέπουμε στα αποτελέσματα του eventual consistency, το query ‘Hey Jude’ εκτελείται 2 φορές μετά από το insert (νέα τιμή 591). Την πρώτη φορά τυχαίνει στον κόμβο 192.168.1.3:30006 που είχε το αρχείο με stale τιμή (588), αφού δεν είχαν προλάβει να διαδοδούν οι αλλαγές των replicas. Τη δεύτερη φορά τυχαίνει στον κόμβο που ανήκουν τα data και συνεπώς απάντησε την πιο πρόσφατη τιμή.



Αντίθετα στο linear consistency, απαντάει πάντα ο τελευταίος κόμβος που έχει τα replicas , όπως ορίζεται στο chain replication και έχει πάντα την πιο πρόσφατη (fresh) τιμή.



Συμπεράσματα

Συνεπώς με βάση τα πειράματα , καταλήγουμε στο συμπέρασμα ότι χρησιμοποιώντας το eventual consistency πετυχαίνουμε μικρότερο write και read throughput με κίνδυνο όμως να διαβαστούν stale τιμές. Ωστόσο σε περιπτώσεις κατανεμημένης εφαρμογής στο Internet , και σύμφωνα με το θεώρημα CAP , το eventual consistency είναι ο μόνος τρόπος για να αποφύγουμε πιθανό partitioning στο κατανεμημένο δίκτυο. Αντίθετα με linear consistency, τα αιτήματα αργούν παραπάνω να γίνουν αλλά είμαστε πάντα σίγουροι ότι το δίκτυο βρίσκεται σε συνεπή κατάσταση και όλοι οι κόμβοι θα έχουν τις πιο φρέσκες τιμές.