Κατανεμημένα Συστήματα

Εξαμηνιαία εργασία

Πενταράκης Μανώλης 03111048

Χατζηκυριάκος Γιώργος 03111164

Εισαγωγή

Σκοπός αυτής της άσκησης είναι η δημιουργία ενός DHT και συγκεκριμένα μιας απλοποιημένης έκδοσης του Chord. Οι λειτουργίες που επιτελεί το DHT είναι join και depart κόμβων και κάθε κόμβος προσφέρει υπηρεσίες insert, delete και query για τα αρχεία που έχει αποθηκευμένα. Επίσης υπάρχει και replication των αρχείων που διατηρεί κάθε κόμβος στους K επόμενούς του, ακολουθώντας διαφορετικές στρατηγικές όσον αφορά τη συνέπεια των replicas.

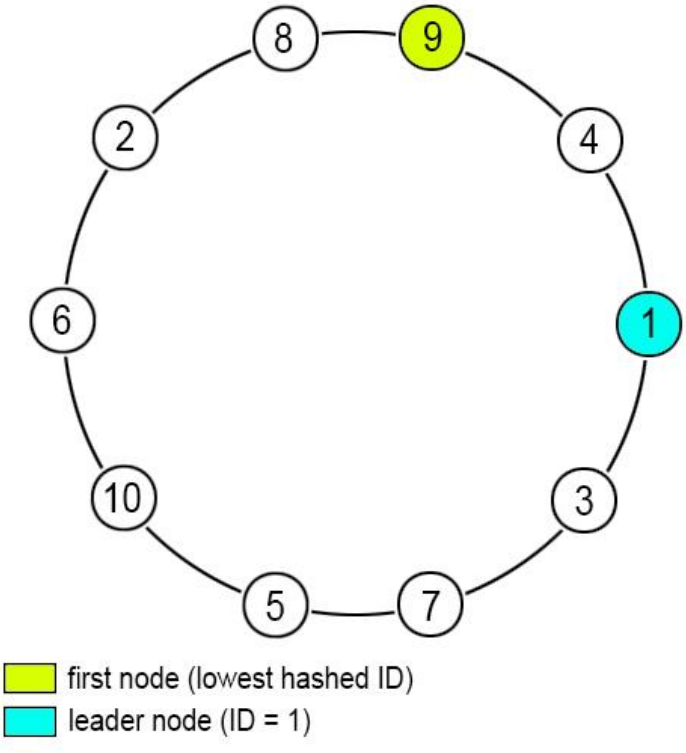
Στην υλοποίησή μας η επικοινωνία μεταξύ των κόμβων γίνεται με sockets. Ο κάθε κόμβος αποτελεί ένα διαφορετικό process που ακούει σε ένα συγκεκριμένο socket για αιτήματα. Όταν παραλάβει ένα αίτημα δημιουργείται ένα client thread για την επεξεργασία του, ενώ ο κόμβος εξακολουθεί να ακούει το socket του για επόμενα αιτήματα (server thread).

Οι κόμβοι που συμμετέχουν στο δίκτυο του Chord είναι οργανωμένοι σε ένα κύκλο και ταξινομούνται σε αυτόν με βάση ένα ID. Κάθε κόμβος γνωρίζει τον επόμενο και τον προηγούμενό του (δηλαδή το socket τους έτσι ώστε να μπορέσει να επικοινωνήσει με αυτούς όταν είναι αναγκαίο).

Δημιουργία κόμβων - δικτύου

Κατά τη δημιουργία ενός νέου process-κόμβου δίνεται σε αυτό ένα socket number και ένα ID που είναι μοναδικά για κάθε κόμβο. Το ID κάθε κόμβου περνάει από μια hash function και στη συνέχεια τοποθετείται ο κόμβος στο δίκτυο. Η hash function που χρησιμοποιήθηκε είναι η SHA1.

Για 10 κόμβους το δίκτυο Chord φαίνεται στο παρακάτω σχήμα. Για κάθε κόμβο βλέπουμε το ID του πριν την εφαρμογή της hash function. Για παράδειγμα ο κόμβος με ID = 9 τοποθετείται πρώτος, αφού μετά την εφαρμογή της SHA1 έχει τη μικρότερη τιμή.

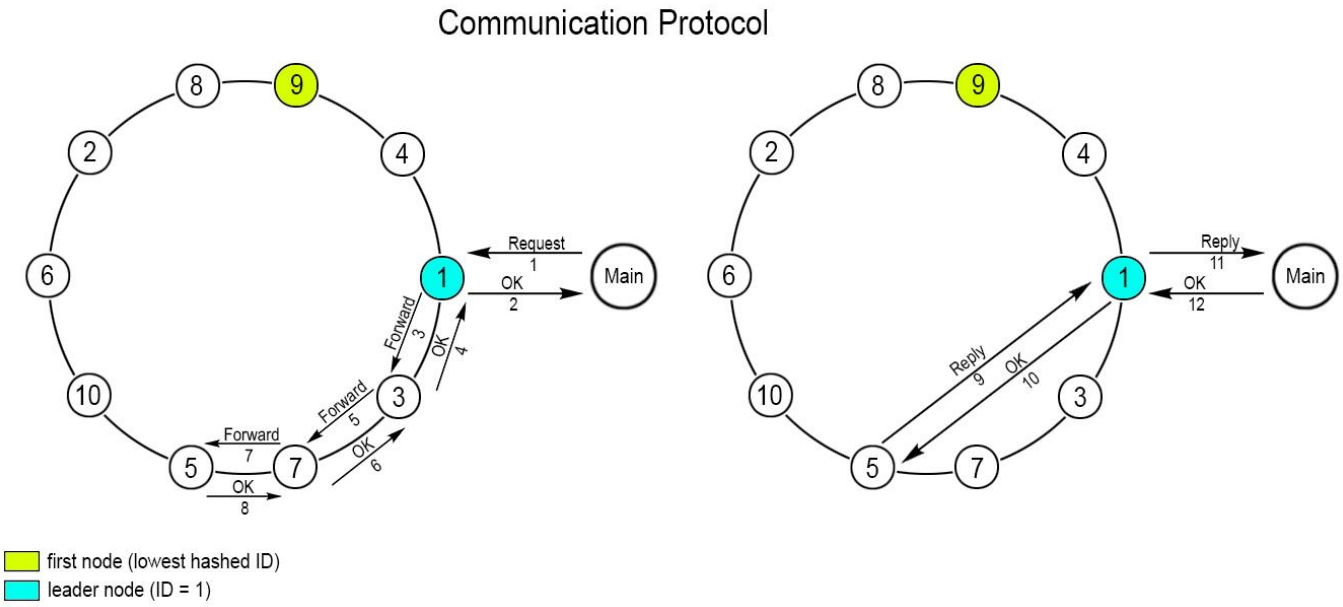


Θέσαμε επίσης ένα κόμβο ως leader για να δέχεται τα αιτήματα join και depart. Leader εκλέξαμε τυχαία τον πρώτο κόμβο που δημιουργήθηκε και μπήκε στο DHT.

Η main διεργασία λειτουργεί σαν πελάτης και από αυτή μπορούμε να στείλουμε αιτήματα στους κόμβους του δικτύου. Αιτήματα τύπου insert,delete,query στέλνονται σε τυχαίο κόμβο του δικτύου, ενώ αιτήματα join,depart στέλνονται στον leader.

Πρωτόκολλο επικοινωνίας μεταξύ κόμβων

Για κάθε αίτημα που στέλνεται, ο αποστολέας περιμένει ένα ack τύπου 'OK' από τον παραλήπτη. Το αίτημα προωθείται από κόμβο σε κόμβο στο δίκτυο μέχρι βρεθεί ο κόμβος που μπορεί να το εξυπηρετήσει. Μετά την επεξεργασία του αιτήματος στέλνεται απάντηση απευθείας στον κόμβο που ξεκίνησε το αίτημα (δηλαδή στον κόμβο με τον οποίο επικοινώνησε αρχικά η main).



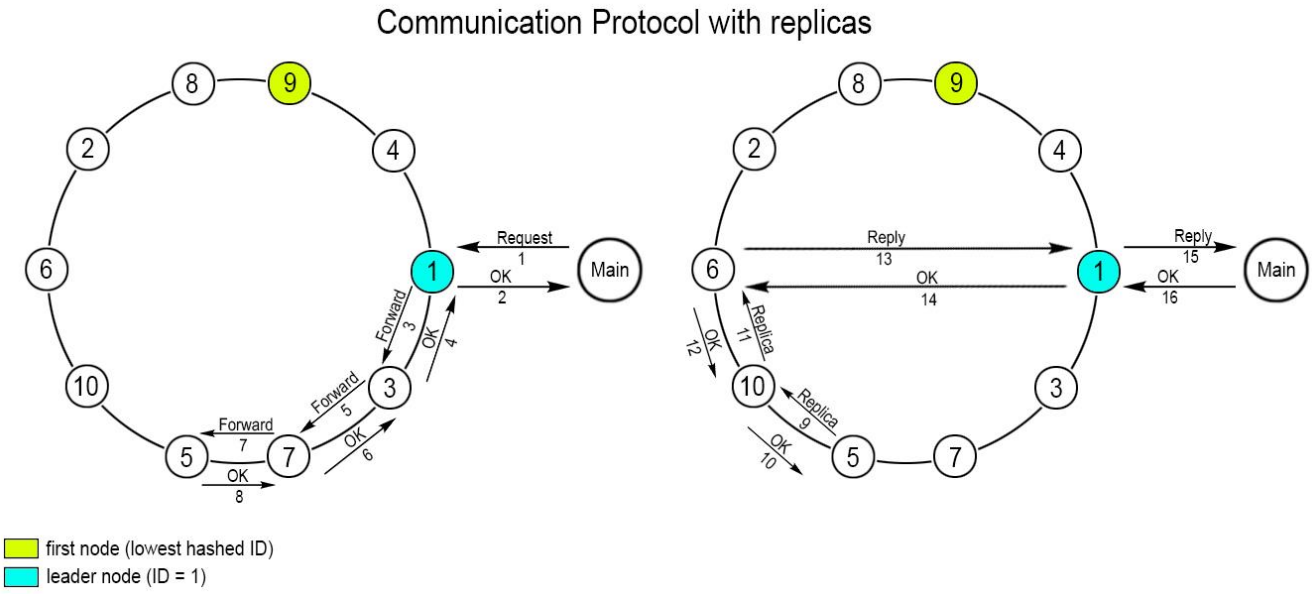
Διαδικασία επικοινωνίας μεταξύ κόμβων(K=1). Η main στέλνει αίτημα στον '1' με τελικό παραλήπτη τον '5'

Το παραπάνω σχήμα αφορά όλα τα αιτήματα για K=1 (χωρίς replication) ή τα αιτήματα join,depart για οποιοδήποτε K.

Για K>1 , δηλαδή για ύπαρξη replicas, το πρωτόκολλο επικοινωνίας αλλάζει ανάλογα με το είδος συνέπειας. Παρακάτω παρουσιάζονται τα δύο είδη συνέπειας που υλοποιήσαμε.

Για την υλοποίηση του *linear consistency* χρησιμοποιήσαμε *chain replication*. Κάθε write (insert ή delete) ξεκινά από τον κόμβο που είναι υπεύθυνος για ένα κλειδί και στη συνέχεια προχωράει στους K-1 επόμενους κόμβους για να κρατήσουν αντίγραφα. Απάντηση στο αίτημα δίνει ο τελευταίος κόμβος της αλυσίδας.

Αντίστοιχα για read αιτήματα (queries) την ζητούμενη τιμή πρέπει να την επιστρέφει ο τελευταίος κόμβος από την αλυσίδα (κάθε κόμβος γνωρίζει για ποια αλυσίδα είναι τελευταίος). Έτσι εξασφαλίζεται ότι τα όλα replicas έχουν πάντα την ίδια τιμή.



Διαδικασία επικοινωνίας μεταξύ κόμβων (K=3 , linear). Η main στέλνει ένα write αίτημα στον '1' με τελικό παραλήπτη τον '5'.Ο '5' στη συνέχεια στέλνει το αρχείο στους 2 επόμενους του και ο τελευταίος απαντά.

Για την υλοποίηση του *eventual consistency* οι αλλαγές διαδίδονται lazily στα αντίγραφα και άρα ακολουθείται το πρωτόκολλο του πρώτου σχήματος. Μετά την απάντηση στον αρχικό παραλήπτη ενός write αιτήματος, ο κόμβος φροντίζει να στείλει τα αντίγραφα στους επόμενους κόμβους. Ένα read διαβάζει από οποιονδήποτε κόμβο έχει το αρχείο (είτε κανονικά είτε replica) με κίνδυνο να επιστραφεί stale τιμή αν δεν έχουν διαδοθεί οι αλλαγές από ένα write στα replicas.

Πειράματα και αποτελέσματα

Στο DHT που περιγράψαμε προηγουμένως, δηλαδή με 10 κόμβους, εκτελέσαμε κάποια πειράματα για να ελέγξουμε την απόδοση περαίωσης των αιτημάτων καθώς και τη συνέπεια στα αποτελέσματα που μας έδιναν.

Γι'αυτό το σκοπό εκτελέστηκαν πειράματα με K=1 (χωρίς replication), K=3 και K=5 και με linearizability και eventual consistency για εισαγωγή αρχείων (insert.txt) και ερωτήματα για αρχεία που υπάρχουν στο DHT (query.txt).

Η διάρκεια επεξεργασίας των insert.txt και query.txt φαίνεται στον παρακάτω πίνακα:

Time (sec)	**insert.txt**		**query.txt**	
	Eventual	Linear	Eventual	Linear
K=1	9,06	9,01	5,81	6,33
K=3	11,42	12,29	4,29	5,37
K=5	13,47	14,85	2,95	4,77

Παρατηρούμε ότι όσο αυξάνεται το K αυξάνεται και ο χρόνος για να γίνουν όλα τα inserts. Αυτό το φαινόμενο παρατηρείται πιο πολύ για Linear, ενώ για eventual θα περιμέναμε να είναι περίπου ίδιος ο χρόνος για οποιοδήποτε K, που όμως δεν συμβαίνει γιατί στο δίκτυο κυκλοφορούν παράλληλα πολλά αιτήματα για insert των replicas και έτσι οι κόμβοι αργούν να απαντήσουν στα insert ερωτήματα της main (δηλαδή υπάρχει πολύ κίνηση στο δίκτυο).

Συγκριτικά τα inserts για eventual consistency είναι πιο γρήγορα από τα αντίστοιχα για linear, αφού κάθε κόμβος επιστρέφει το αποτέλεσμα του write αμέσως, ενώ για linear πρώτα τα γράφει στα replicas και στη συνέχεια επιστρέφει το αποτέλεσμα του write. Για K=1 δεν υπάρχουν διαφορές για eventual ή linear consistency, καθώς δεν υπάρχουν replicas.

Τα queries για eventual consistency γίνονται γρηγορότερα για μεγαλύτερα K, αφού βρίσκονται σε περισσότερους κόμβους που μπορούν να δώσουν απάντηση.

Αντίθετα για linear consistency απάντηση πρέπει να δώσει πάντα ένας συγκεκριμένος κόμβος, οπότε δεν παρατηρούμε μεγάλες αλλαγές για διαφορετικά K.

Για το αρχείο requests.txt που περιέχει αιτήματα insert και query καταγράψαμε τις απαντήσεις των κόμβων του δικτύου και για τα δύο είδη συνέπειας (K=3).

Παρακάτω φαίνεται ένα μέρος των απαντήσεων για τα ίδια αιτήματα με eventual και linear consistency:

<table>
<tr><th>Request</th><th colspan="2">Answer</th></tr>
<tr><td rowspan="2">insert, Hey Jude, 501
query, Hey Jude
query, Hey Jude
query, Hey Jude
query, What's Going On</td><td>Eventual</td><td>INSERT Answer from: 2
Hey Jude 500 |QUERY Answer from: 9
Hey Jude 501 |QUERY Answer from: 9
Hey Jude 501 |QUERY Answer from: 2
What's Going On 4 |QUERY Answer from: 5</td></tr>
<tr><td>Linear</td><td>INSERTREPLICA Answer from: 9
Hey Jude 501 |QUERY Answer from: 9
Hey Jude 501 |QUERY Answer from: 9
Hey Jude 501 |QUERY Answer from: 9
What's Going On 4 |QUERY Answer from: 6</td></tr>
</table>

Παρατηρούμε ότι για eventual consistency το query μετά το insert έτυχε στον κόμβο 9 που είχε το αρχείο με stale τιμή, αφού δεν είχαν προλάβει να διαδοθούν οι αλλαγές στα replicas. Το δεύτερο query που έγινε πάλι στον 9 βλέπουμε ότι γύρισε την καινούργια τιμή και άρα είχαν προλάβει να διαδοθούν οι αλλαγές.

Απάντηση στα ερωτήματα μας δίνει όποιος κόμβος έχει το κλειδί ή αντίγραφό του (στο συγκεκριμένο παράδειγμα μπορούσαν να δώσουν απάντηση ο 2,8 ή 9).

Για linear consistency παρατηρούμε ότι η απάντηση στα query ερωτήματα είναι η ίδια και πάντα η νέα τιμή. Αυτό συμβαίνει καθώς απάντηση μας δίνει πάντα ο τελευταίος κόμβος της αλυσίδας (στο παράδειγμα αυτό ο 9) και έτσι οι αλλαγές στα αρχεία είναι εγγυημένο ότι θα έχουν διαδοθεί και όλοι οι κόμβοι έχουν την ίδια τιμή για το αρχείο.

Με βάση όλα τα παραπάνω καταλήγουμε στο συμπέρασμα ότι χρησιμοποιώντας eventual consistency πετυχαίνουμε μεγαλύτερο write και read throughput με κίνδυνο όμως να διαβάζουμε κάποιες φορές stale τιμές για τα δεδομένα που έχουμε.

Με linear consistency τα αιτήματα αργούν παραπάνω να διεκπεραιωθούν, όμως είμαστε σίγουροι ότι το δίκτυο έχει συνεπή κατάσταση και όλοι οι κόμβοι θα έχουν τις ίδιες τιμές για τα δεδομένα.