Κατανεμημένα Συστήματα

Εξαμηνιαία Εργασία

Γεζεκελιάν Βικέν ΑΜ: 03116112
Σιφναίος Σάββας ΑΜ: 03116080
Μπακιρτζή Χριστίνα ΑΜ: 03116013
Ακ. έτος: 2020 – 2021

Σκοπός

Στα πλαίσια της συγκεκριμένης εργασίας επιχειρήσαμε να υλοποιήσουμε μια απλοποιημένη έκδοση του Chord, ενός πρωτοκόλλου δηλαδή λειτουργίας που αφορά Distributed Hash Tables.

Στο πρωτόκολλο αυτό, οι κόμβοι που συμμετέχουν στο DHT διατάσσονται με τέτοιο τρόπο ώστε να σχηματίζεται ένας εικονικός δακτύλιος του οποίου οι κόμβοι γνωρίζουν μόνο τον άμεσο πρόγονο και τον απόγονό τους. Επιπλέον, στην αρχιτεκτονική αυτή κάθε κόμβος του δακτυλίου διαθέτει ένα μοναδικό ID ,το οποίο είναι το αποτέλεσμα της εφαρμογής της hash function ***SHA 1*** στην IP του, και βάσει του ID αυτού είναι υπεύθυνος να αποθηκεύει ορισμένα key-value ζεύγη. Συγκεκριμένα, ένα ζεύγος key-value λαμβάνει και αυτό ID μέσω της εφαρμογής της ίδιας hash function στο key και ανατίθεται στον κόμβο εκείνο που έχει το αμέσως μεγαλύτερο ID.

Υλοποίηση

Επιλέξαμε να υλοποιήσουμε το σύστημα χρησιμοποιώντας Python και το microframework Flask, με αποτέλεσμα οι κόμβοι να επικοινωνούν μεταξύ τους με http requests. Κάθε κόμβος υλοποιεί τις λειτουργίες insert(key, value), query(key), delete(key), ενώ το σύστημα διαχειρίζεται επιτυχώς εισαγωγές νέων κόμβων και αποχωρήσεις κόμβων, έχοντας ορίσει από την αρχή έναν bootstrap κόμβο ο οποίος δέχεται όλα τα αιτήματα για join και ο ίδιος δεν αποχωρεί ποτέ. Όλες οι παραπάνω λειτουργίες μεταφράζονται σε http requests που οι κόμβοι διαχειρίζονται, χρησιμοποιώντας βοηθητικά endpoints και εσωτερικά threads.

Η υλοποίησή μας υποστηρίζει 2 είδη συνέπειας για τα αντίγραφα των δεδομένων που είναι αποθηκευμένα στο σύστημα, linearizability και eventual consistency. Αυτό ορίζεται κάθε φορά από μια μεταβλητή k, το replication factor, που ουσιαστικά εξασφαλίζει την αποθήκευση κάθε <key, value> ζεύγους, εκτός από τον κόμβο που είναι υπεύθυνο για το hash(key), και στους k-1 επόμενους κόμβους του λογικού δακτυλίου. Το replication λήφθηκε υπόψιν για κάθε μία από τις βασικές λειτουργίες του DHT που αναφέρθηκαν παραπάνω.

Πειράματα

Εισαγωγή

Για την εκτέλεση των πειραμάτων χρησιμοποιηθήκαν Virtual Machines (VMs) , οι πόροι των οποίων μας παραχωρήθηκαν μέσω της υπηρεσίας OKEANOS. Συγκεκριμένα, στην διάθεσή μας είχαμε 5 Virtual Machines των οποίων οι τοπικές διευθύνσεις IP ήταν οι 192.168.0.1/5. Επομένως, για την εκτέλεση των πειραμάτων, που αφορούν ένα DHT με 10 κόμβους, σε κάθε VM αναθέσαμε δύο κόμβους μεριμνώντας ώστε κάθε κόμβος να "ακούει" σε διαφορετικό port. Επιπλέον, αξίζει να σημειωθεί πως κατά την εκτέλεση των πειραμάτων, ο κώδικας που υλοποιεί το backend του bootstrap node ,που εισάγεται πρώτος στο DHT και μέσω του οποίου πραγματοποιείται η εισαγωγή και των υπόλοιπων κόμβων στον δακτύλιο, εκτελέστηκε στο VM με τοπική IP 192.168.0.1 και "άκουγε" στο port 5000.

Στην συνέχεια, αξίζει να αναφερθεί πως κατά την δημιουργία ενός νέου κόμβου στέλνεται αυτόματα και ένα αίτημα εισαγωγής στον δακτύλιο από τον νεο-δημιουργηθέντα κόμβο προς τον bootstrap node ώστε να μην απαιτείται κάποια περαιτέρω ενέργεια από πλευρές του χρήστη για την συμμετοχή στο DHT. Για την υλοποίηση της λειτουργικότητας αυτής, χρησιμοποιήθηκε η μέθοδος fork της βιβλιοθήκης os της python, η οποία δημιουργεί ένα νέο process. Συγκεκριμένα, η γονική διεργασία είναι υπεύθυνη για την εκτέλεση του backend κώδικα του εκάστοτε κόμβου ενώ η διεργασία απόγονος είναι αυτή που στέλνει το αίτημα εισαγωγής στον δακτύλιο και η οποία "πεθαίνει" μόλις λάβει την σχετική απάντηση από τον bootstrap node.

Εκτέλεση πειραμάτων

1. Αρχικά δοκιμάσαμε να εισάγουμε σε ένα DHT με 10 κόμβους όλα τα κλειδιά που βρίσκονται στο αρχείο insert.txt, ξεκινώντας τα inserts κάθε φορά από τυχαίο κόμβο του συστήματος, χρησιμοποιώντας το αρχείο *test_insert.py*. Διεξήγαμε 6 πειράματα – για replication factor k =1, k=3 και k=5 με linearizability και eventual consistency σε κάθε περίπτωση.

Ως προς την συνέπεια, για την περίπτωση του linearizability, υπάρχουν ισχυρές εγγυήσεις ότι όλα τα αντίγραφα έχουν πάντα την ίδια τιμή. Υλοποιήσαμε linearizability με chain replication. Ένα write ξεκινάει κάθε φορά από τον πρωτεύοντα κόμβο που είναι υπεύθυνος για ένα κλειδί και προχωρά με τη σειρά στους k-1 υπόλοιπους κόμβους που έχουν αντίγραφα. Ο τελευταίος κόμβος στη σειρά επιστρέφει το αποτέλεσμα του write .Αντίστοιχα στην περίπτωση του read η τιμή που αντιστοιχεί σε κάποιο συγκεκριμένο key επιστρέφεται πάντα από τον τελευταίο κόμβο στην σειρά που διαθέτει αντίγραφο (k-οστό αντίγραφο).

Στην περίπτωση που χρησιμοποιείται eventual consistency, ένα write πηγαίνει στον πρωτεύοντα κόμβο που είναι υπεύθυνος για το συγκεκριμένο κλειδί και ο κόμβος αυτός επιστρέφει το αποτέλεσμα του write προτού ολοκληρωθεί η εγγραφή των υπόλοιπων k-1 αντιγράφων του κλειδιού. Αφού επιστρέψει, φροντίζει να στείλει τη νέα τιμή στους επόμενους k-1 επόμενους κόμβους και να ενημερώσει τα αντίγραφα. Ένα αίτημα αναζήτησης κάποιου key, λαμβάνεται από κάποιο κόμβο του dht και προωθείται στον επόμενό του μέχρι ότου να βρεθεί κάποιο από τα k αντίγραφα. Σε αυτή την περίπτωση δεν αναζητούμε απαραίτητα το τελευταίο αντίγραφο αποσκοπώντας στην επιτάχυνση της διαδικασίας αναζήτησης, εισάγοντας παράλληλα , όμως, τον κίνδυνο να διαβάσουμε κάποια stale τιμή .

Όπως γνωρίζουμε και από το CAP Theorem, είναι αδύνατον για μια κατανεμημένη αποθήκη δεδομένων να παρέχει ταυτόχρονα περισσότερες από δύο εγγυήσεις εκ των τριών: Consistency, Availability, Partition Tolerance.
Ουσιαστικά, χρησιμοποιώντας linear consistency επιλέγουμε συνέπεια και αντοχή στην κλιμάκωση "θυσιάζοντας" λίγη από την άμεση διαθεσιμότητα του συστήματος, ενώ με eventual consistency παρέχουμε στο χρήστη συνεχώς διαθέσιμα δεδομένα και αντοχή στην κλιμάκωση, με κίνδυνο όμως

αυτά να μην είναι ενημερωμένα.

Παρακάτω παρουσιάζουμε τα αποτελέσματα από την εκτέλεση των έξι προαναφερθέντων πειραμάτων.

- k=1, linear consistency:

```
Write Throughput is:  0.12523482990264892

Total time required for executing all the requests:  62.617507219314575 sec
```

- k=1, eventual consistency:

```
Write Throughput is:  0.12490203285217286

Total time required for executing all the requests:  62.45109796524048 sec
```

- k=3, linear consistency:

```
Write Throughput is:  0.14731177282333374

Total time required for executing all the requests:  73.65599226951599 sec
```

- k=3, eventual consistency:

```
Write Throughput is:  0.12553965616226195

Total time required for executing all the requests:  62.770025968551636 sec
```

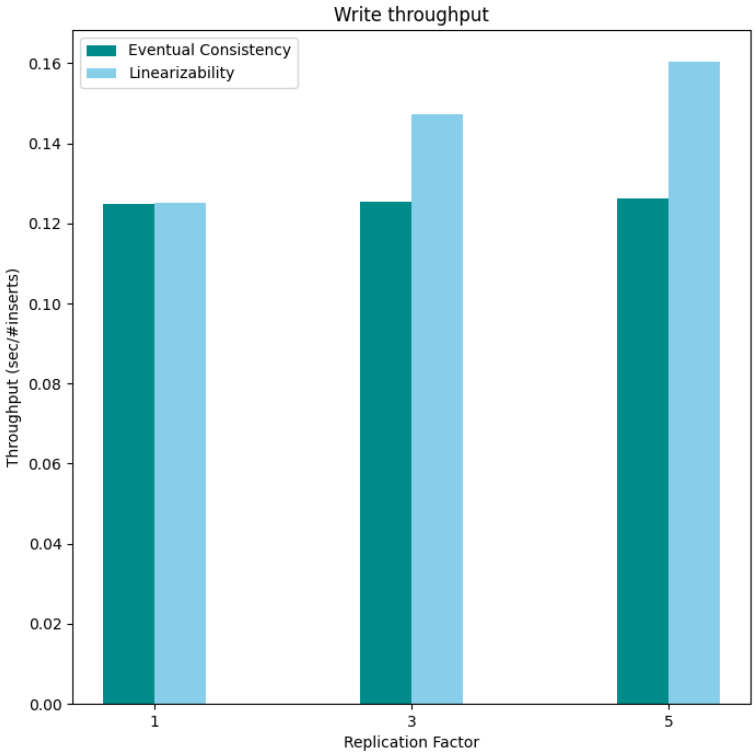
- k=5, linear consistency:

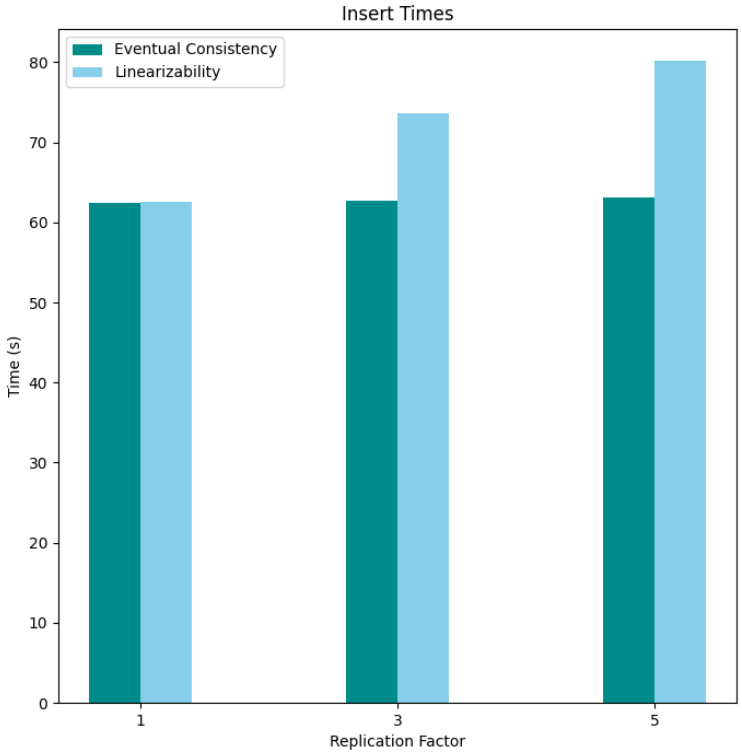
```
Write Throughput is:  0.16031630659103394

Total time required for executing all the requests:  80.15839719772339 sec
```

- k=5, eventual consistency:

```
Write Throughput is:  0.12613303232192993

Total time required for executing all the requests:  63.06677770614624 sec
```

Τα αποτελέσματά μας συνοψίζονται στα παρακάτω διαγράμματα:


Write throughput
Eventual Consistency
Linearizability
Throughput (sec/#inserts)
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.14
0.16
1
3
5
Replication Factor


Insert Times
Eventual Consistency
Linearizability
Time (s)
0
10
20
30
40
50
60
70
80
1
3
5
Replication Factor

Παρατηρούμε ότι με linear consistency, καθώς αυξάνεται ο αριθμός των αντιγράφων k, αυξάνεται το throughput καθώς και ο χρόνος που απαιτείται για να ολοκληρωθούν τα αιτήματα insert. Αυτό ήταν αναμενόμενο, αφού όπως περιγράφηκε και παραπάνω, για να ολοκληρωθεί κάποιο αίτημα write με linearizability, χρειάζεται πρώτα να ολοκληρωθεί η ενημέρωση όλων των αντιγράφων του αντίστοιχου κλειδιού. Επομένως όσο μεγαλύτερος είναι ο αριθμός των αντιγράφων, τόσο μεγαλύτερος θα είναι και ο απαιτούμενος χρόνος για την ολοκλήρωση των εγγραφών.

Ως προς την eventual consistency, όπως προαναφέρθηκε, ένα write ολοκληρώνεται τη στιγμή που θα ενημερωθεί η τιμή του πρωτεύοντος αντίγραφου του κλειδιού, ενώ τα υπόλοιπα αντίγραφα ενημερώνονται lazily,ακόμα και αφού επιστραφεί το αποτέλεσμα του write. Αυτό επιβεβαιώνεται από το γεγονός ότι, παρά την αύξηση των αντιγράφων, ο χρόνος και το throughput για k=1, k=3 και k=5, παρουσιάζει ελάχιστες διακυμάνσεις.

Συγκρίνοντας τα δύο είδη συνέπειας, το throughput φαίνεται να είναι ίδιο για εκτελέσεις χωρίς αντίγραφα (k=1), αφού πρόκειται για ακριβώς τις ίδιες λειτουργίες, όμως το linearizability απαιτεί σαφέστατα περισσότερο χρόνο για την ολοκλήρωση των αιτημάτων inserts καθώς αυξάνεται το πλήθος των αντιγράφων ανά κλειδί, σε αντίθεση με το eventual consistency που τόσο το throughput όσο και ο χρόνος εισαγωγής των κλειδιών παραμένει πρακτικά σταθερός.

2. Στη συνέχεια για κάθε μία από τις 6 διαφορετικές περιπτώσεις του προηγούμενου ερωτήματος επιχειρήσαμε να αναζητήσουμε στο DHT όλα τα keys που συμπεριλαμβάνονται στο αρχείο query.txt, με τη βοήθεια του αρχείου *test_query.py*, ξεκινώντας τα queries κάθε φορά από έναν τυχαίο κόμβο του συστήματος. Καταγράψαμε τους χρόνους εκτέλεσης και το read throughput για κάθε setup:

- k=1, linear consistency:

```
Read Throughput is:  0.030045846462249757

Total time required for executing all the requests:  15.022923231124878  sec
```

- k=1, eventual consistency:

```
Read Throughput is:  0.0313429741859436

Total time required for executing all the requests:  15.671487092971802  sec
```

- k=3, linear consistency:

```
Read Throughput is:  0.03214139127731323

Total time required for executing all the requests:  16.070695638656616  sec
```

- k=3, eventual consistency:

```
Read Throughput is:  0.020973093032836915

Total time required for executing all the requests:  10.486546516418457  sec
```

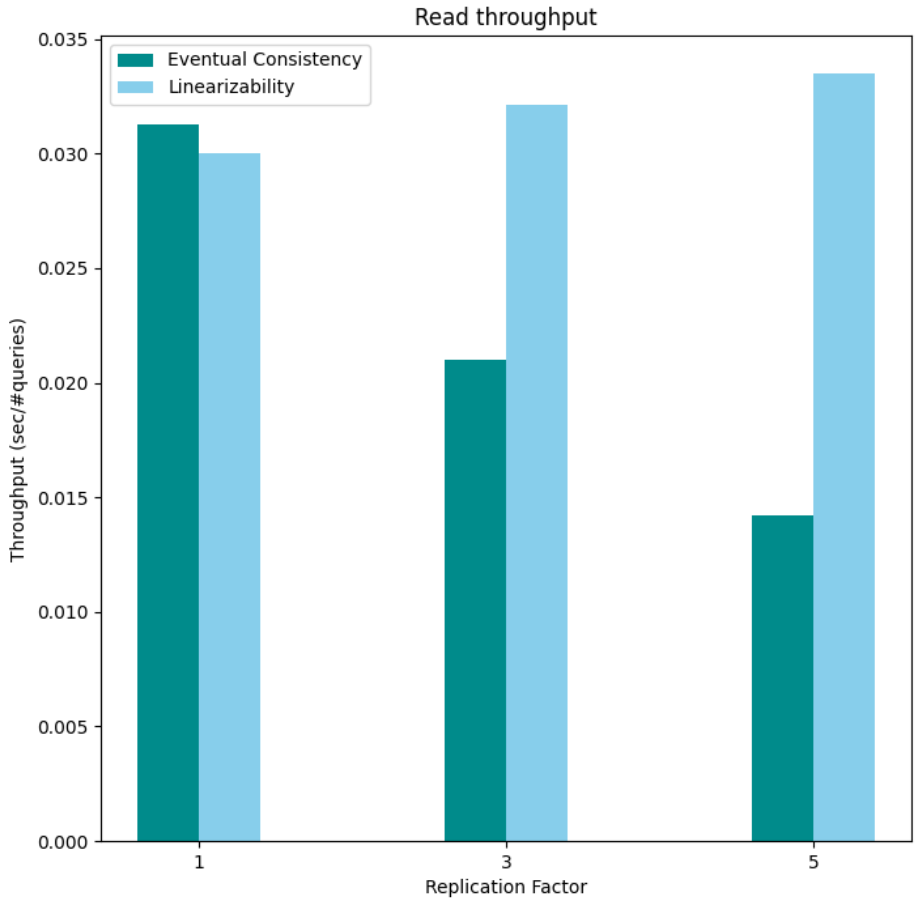
- k=5, linear consistency:

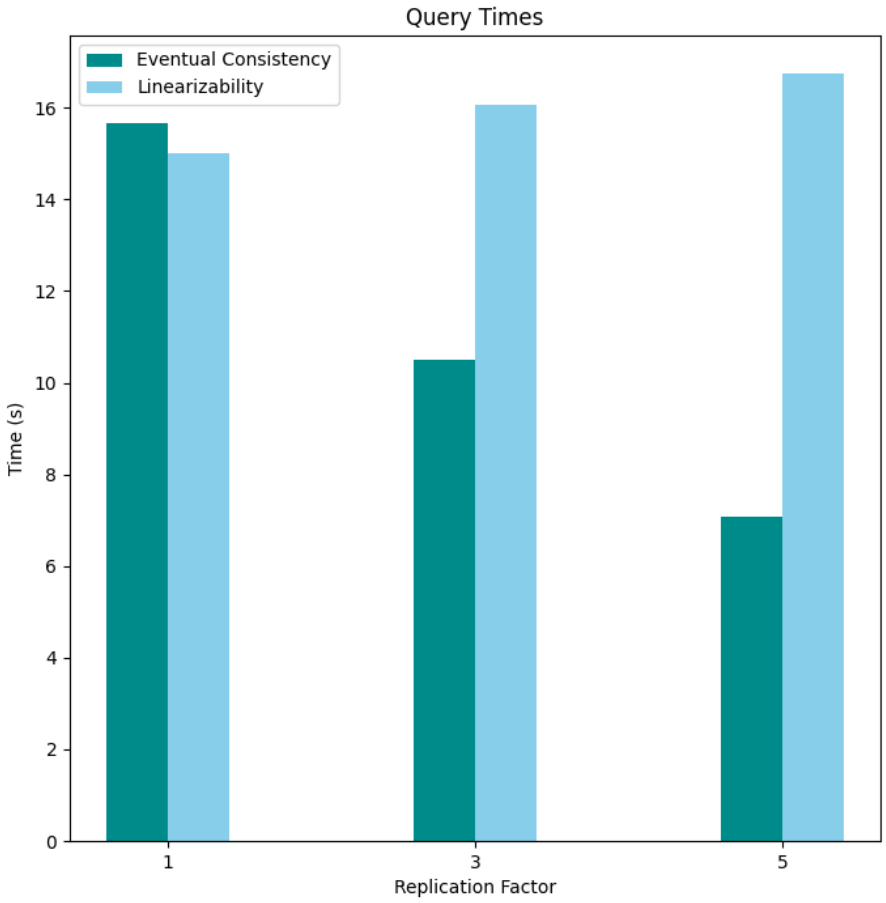
```
Read Throughput is:  0.03347954893112182

Total time required for executing all the requests:  16.739774465560913  sec
```

- k=5, eventual consistency:

```
Read Throughput is:  0.014177134990692139

Total time required for executing all the requests:  7.088567495346069  sec
```

Τα παραπάνω αποτελέσματα συνοψίζονται και στα παρακάτω διαγράμματα:





Καθώς το k αυξάνεται, παρατηρούμε ότι για την περίπτωση του linearizability το read throughput παρουσιάζει μια μικρή αύξηση, καθώς παρότι πλέον περισσότεροι του ενός κόμβοι περιλαμβάνουν κάποιο αντίγραφο του κλειδιού, αναζητείται κάθε φορά το τελευταίο από τα αντίγραφα αυτά για να επιστραφεί. Επομένως ακόμα κι αν ο κόμβος που ερωτήθηκε έχει κάποιο αντίγραφο του κλειδιού που ψάχνει δεν θα το επιστρέψει, αλλά θα αναζητήσει το τελευταίο αντίγραφο που περιέχει το σύστημα.

Αντίθετα, όσον αφορά το eventual consistency, το αίτημα αναζήτησης κάποιου κλειδιού προωθείται στους γείτονες κάθε κόμβου μέχρι να ανιχνευθεί κάποιο από τα k αντίγραφα χωρίς κατ' ανάγκη αυτό να είναι το τελευταίο, με κίνδυνο βέβαια να επιστραφεί stale τιμή. Αυτό επιβεβαιώνεται από τα πειράματά μας, αφού το read throughput μειώνεται όσο αυξάνουμε των αριθμό των replicas. Συγκεκριμένα, όσο περισσότερα είναι τα αντίγραφα, τόσο περισσότερες είναι και οι πιθανότητες ο κόμβος από τον οποίο ξεκίνησε ένα query ερώτημα να περιέχει ένα από τα αντίγραφα του κλειδιού αυτού, και εφόσον δεν μας ενδιαφέρει ο αριθμός του αντίγραφου, να επιστρέψει αμέσως με την τιμή του.

Συγκρίνοντας τα δύο είδη συνέπειας, εύκολα μπορούμε να παρατηρήσουμε πως το eventual consistency εξασφαλίζει με σχετικά σημαντική διαφορά ταχύτερης εκτελέσεις των αιτημάτων query και insert. Επομένως, αν χρησιμοποιούσαμε ως μετρική αξιολόγησης του DHT την ποσότητα throughput τότε σίγουρα το βέλτιστο είδος consistency θα ήταν το eventual. Ωστόσο, η χρήση αυτού του είδους συνέπειας ενέχει τον κίνδυνο εντοπισμού μίας outdated τιμής στο DHT με αποτέλεσμα να εισάγεται ένα trade-off μεταξύ ταχύτητας και εγγύησης ότι η επιστρεφόμενη τιμή είναι και η πιο πρόσφατη. Συνεπώς, η επιλογή ενός καθολικά βέλτιστου είδους συνέπειας ,

μεταξύ των δύο εξεταζομένων, δεν είναι εφικτή και προφανώς εξαρτάται άμεσα από τις απαιτήσεις που προκύπτουν από την εφαρμογή στην οποία θα χρησιμοποιηθεί.

3. Τέλος, για ένα DHT με 10 κόμβους και k=3 αντίγραφα, εκτελέσαμε τα requests του αρχείου requests.txt, μέσω του script *test_requests.py* και με τα δύο είδη συνέπειας. Καταγράψαμε τις απαντήσεις των queries που περιέχει το αρχείο σε σχέση με τα αιτήματα insert και ελέγξαμε το freshness των τιμών που παίρνουμε:

-Linearizability:

```
The number of not so fresh returned from queries are: 0
Write Throughput is:  0.26742145538330075
Read Throughput is  0.06685536384582519

Total time required for executing all the requests:  26.742145538330078 sec
```

-Eventual Consistency:

```
The number of not so fresh returned from queries are: 1
Write Throughput is:  0.19443076610565185
Read Throughput is  0.04860769152641296

Total time required for executing all the requests:  19.443076610565186 sec
```

Όπως είχαμε παρατηρήσει και παραπάνω, με eventual consistency έχουμε σαφώς μικρότερο read και write throughput σε σχέση με αυτό που παρέχεται μέσω linear consistency και τα requests ολοκληρώνονται αρκετά πιο γρήγορα στην περίπτωση του eventual. Ωστόσο, με linear consistency μπορούμε να εγγυηθούμε την συνέπεια των αποτελεσμάτων, όπως φαίνεται και από το γεγονός ότι η εκδοχή αυτή δίνει μόνο fresh τιμές και δεν υπάρχει κάποιο query το οποίο να επιστρέφει αποτέλεσμα που δεν έχει ενημερωθεί. Αντίθετα, η eventual εκδοχή, παρά την υψηλότερη διαθεσιμότητα, δεν μπορεί να εγγυηθεί στο απόλυτο τη συνέπεια των αποτελεσμάτων, καθώς βλέπουμε ότι υπάρχει μια απάντηση query αιτήματος που επιστράφηκε ενώ δεν ήταν τόσο fresh – δεν είχε προλάβει να ενημερωθεί το αντίγραφο που επιστράφηκε. Αυτό συμβαίνει διότι στη συγκεκριμένη περίπτωση αποφασίζουμε να επικεντρωθούμε στην γρήγορη απόκριση και όχι στη συνέπεια. Μάλιστα αν είχαμε ακόμα περισσότερα αντίγραφα ανά κλειδί ή λιγότερους κόμβους στο σύστημα, οι ασυνέπειες στις τιμές που επιστρέφονται σε query ερωτήματα θα ήταν ενδεχομένως ακόμα περισσότερες, καθώς θα χρειαζόταν περισσότερος χρόνος για να ενημερωθούν όλα τα αντίγραφα και επιπλέον θα αυξάνονταν οι πιθανότητες ο κόμβος από τον οποίο ξεκίνησε ένα query για κάποιο κλειδί να περιέχει κάποιο αντίγραφο του κλειδιού αυτού και να το επιστρέψει κατευθείαν, ασχέτως με το αν έχει προλάβει να ενημερώσει την τιμή του.