

Ατομική Διπλωματική Εργασία

**ΥΛΟΠΟΙΗΣΗ ΚΑΙ ΠΕΙΡΑΜΑΤΙΚΗ ΑΞΙΟΛΟΓΗΣΗ
ΑΛΓΟΡΙΘΜΩΝ ΑΛΥΣΙΔΑΣ ΚΟΙΝΟΠΟΙΗΣΕΩΝ ΜΕ ΤΗ ΧΡΗΣΗ
ΤΟΥ ZEROMQ**

Γεώργιος Παναγιώτου

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ



ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάιος 2019

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Υλοποίηση και Πειραματική Αξιολόγηση Αλγορίθμων Αλυσίδας
Κοινοποιήσεων με τη χρήση του ZeroMQ

Γεώργιος Παναγιώτου

Επιβλέπων Καθηγητής
Δρ. Χρύσης Γεωργίου

Η Ατομική Διπλωματική Εργασία υποβλήθηκε προς μερική εκπλήρωση των
απαιτήσεων απόκτησης του πτυχίου Πληροφορικής του Τμήματος Πληροφορικής του
Πανεπιστημίου Κύπρου

Μάιος 2019

Ευχαριστίες

Θα ήθελα να εκφράσω τις πολύ θερμές μου ευχαριστίες στον επιβλέποντα καθηγητή μου, Δρ. Χρύση Γεωργίου, για την τιμή πρώτιστα που μου έκανε, αναλαμβάνοντας την καθοδήγησή μου, δίνοντας μου παράλληλα την ευκαιρία να εκπονήσω τη διπλωματική εργασία μου με θέμα την Υλοποίηση και Πειραματική Αξιολόγηση Αλγορίθμων Αλυσίδας Κοινοποιήσεων, με τη χρήση του ZeroMQ.

Η εντριβή μου με το συγκεκριμένο θέμα, αδιαμφισβήτητα με βοήθησε να διευρύνω τους ορίζοντες μου ανακαλύπτοντας κλάδους της Πληροφορικής με τους οποίους μέχρι τώρα δεν είχα ασχοληθεί ιδιαίτερα, όπως τα κατανεμημένα συστήματα, ο προγραμματισμός συστημάτων και τα blockchains, καθώς και να αποκτήσω τη σημαντική για μένα εμπειρία της ενασχόλησης με ερευνητικό έργο, κάτι με το οποίο θα ήθελα να ασχοληθώ στη μετέπειτα πορεία μου.

Η πάντοτε χρήσιμη καθοδήγηση του συνέβαλε τα μέγιστα στην ολοκλήρωση της εργασίας αυτής, και οδήγησε σε μια ευχάριστη συνεργασία κατά τη διάρκεια των δύο προηγούμενων εξαμήνων.

Περίληψη

Ένα από τα πλέον “καυτά” θέματα στο χώρο της τεχνολογίας είναι οι αλυσίδες κοινοποιήσεων (blockchains), καθώς έχουν αποκτήσει φήμη πρόσφατα, λόγω και της ραγδαίας ανάπτυξης τεχνολογιών που τα χρησιμοποιούν, με πιο γνωστά τα κρυπτονομίσματα όπως το Bitcoin και Ethereum. Τέτοια συστήματα λειτουργούν σε κατακεντρωμένα συστήματα, με συναλλαγές ομοτίμων (peer-to-peer) χωρίς κάποιο μεσάζοντα, και αποθηκεύουν τις συναλλαγές τους σε πολλά συστήματα ταυτόχρονα, δημιουργώντας έτσι μια ψηφιακή αλυσίδα κοινοποιήσεων.

Στη διπλωματική εργασία αυτή, υλοποιούνται τρεις αλγόριθμοι αλυσίδας κοινοποιήσεων, με τη χρήση της βιβλιοθήκης ZeroMQ για προγραμματισμό υποδοχών, και αξιολογείται πειραματικά η απόδοσή τους στο ερευνητικό δίκτυο PlanetLab Europe, χρησιμοποιώντας μηχανές σε όλη την Ευρώπη, ώστε να δημιουργηθεί ένα ασύγχρονο, πραγματικό κατακεντρωμένο δίκτυο, με την πιθανότητα σφαλμάτων. Οι αλγόριθμοι αυτοί διαφέρουν στο βαθμό της συνέπειας που μας προσφέρουν, και η πειραματική αξιολόγηση μας δείχνει τις διαφορές που παρουσιάζουν στο χρόνο που χρειάζονται να εξυπηρετήσουν αιτήματα και να προσθέσουν νέες εγγραφές.

Περιεχόμενα

Κεφάλαιο 1	Εισαγωγή.....	1
	1.1 Κίνητρο	1
	1.2 Σκοπός Διπλωματικής Εργασίας	2
	1.3 Μεθοδολογία	2
	1.4 Δομή Εργασίας	4
	1.5 Γλωσσάρι	6
Κεφάλαιο 2	Προαπαιτούμενες Γνώσεις	7
	2.1 Κατανεμημένα Συστήματα	7
	2.2 Αλυσίδες Κοινοποιήσεων	8
	2.3 Αντικείμενα Κατάστιχου	9
	2.4 ZeroMQ	11
	2.5 PlanetLab	12
Κεφάλαιο 3	Μοντέλο και Περιγραφή Αλγορίθμου.....	13
	3.1 Μοντέλο Αλγορίθμων για ΚΑΚ	13
	3.1.1 Συνέπεια ΚΑΚ	14
	3.2 Περιγραφή Αλγορίθμων	15
	3.2.1 Υπηρεσία Ατομικής Πολυεκπομπής	15
	3.2.2 Περιγραφή	17
	3.2.2.1 ΚΑΚ με Τελική Συνέπεια	17
	3.2.2.2 ΚΑΚ με Ατομική Συνέπεια	19
	3.2.2.3 ΚΑΚ με Ακολουθιακή Συνέπεια	20
	3.2.2.4 Επικυρωμένο Κατάστιχο	21
Κεφάλαιο 4	Υλοποίηση Αλγορίθμων	23
	4.1 Μορφή Μηνυμάτων	23

4.2 Λειτουργίες Συστήματος	25
4.3 Προγραμματισμός Υποδοχών με ZeroMQ	27
4.3.1 Μοντέλο Πελάτη - Εξυπηρετητή	28
4.3.2 Μοντέλο Εκδότη - Συνδρομητή	29
4.3.3 Αποστολή μηνυμάτων μεταξύ υποδοχών	31
4.4 Μηχανισμός Ανοχής Σφαλμάτων	32
4.5 Δομές Δεδομένων	34
4.6 Προγραμματισμός Διεργασιών AK	36
4.7 Προγραμματισμός Διεργασιών Πελάτη	40
4.8 Παραδοχές για Υλοποίηση	42
 Κεφάλαιο 5 Πειραματική Αξιολόγηση Αλγορίθμων	44
5.1 Προεργασία και Μεθοδολογία	44
5.2 Προβλήματα και Παρατηρήσεις για το δίκτυο	45
5.3 Σενάρια Αξιολόγησης	47
5.4 Αποτελέσματα	54
5.5 Παρατηρήσεις και Συμπεράσματα	64
 Κεφάλαιο 6 Συμπεράσματα	65
6.1 Γενικότερα Συμπεράσματα	65
6.2 Μελλοντική Εργασία	66
 Βιβλιογραφία	67

Κεφάλαιο 1

Εισαγωγή

1.1 Κίνητρο

1.2 Σκοπός Διπλωματικής Εργασίας

1.3 Μεθοδολογία

1.4 Δομή Εργασίας

1.5 Γλωσσάρι

1.1 Κίνητρο

Οι αλυσίδες κοινοποιήσεων είναι ένα θέμα το οποίο προσφάτως έχει αποκτήσει πολύ μεγάλες διαστάσεις, λόγω της ευρείας εξάπλωσης τεχνολογιών που τις υλοποιούν, με πιο γνωστό παράδειγμα τα κρυπτονομίσματα (cryptocurrencies) όπως το Bitcoin [11], που ήταν και η πρώτη εφαρμογή της τεχνολογίας αυτής.

Παρ' όλη τη δημοσιότητα, όμως, που έχουν αποκτήσει οι τεχνολογίες αυτές, και την εξάπλωση των κρυπτονομισμάτων, υπάρχουν λίγοι ορισμοί τους και ακόμα λιγότερες επιστημονικές μελέτες τους σε θεωρητικό επίπεδο.

Με την εργασία αυτή, επιδιώκεται να υλοποιηθούν αλγόριθμοι για αλυσίδες κοινοποιήσεων, και να αξιολογηθούν πειραματικά σε πραγματικό κατανεμημένο δίκτυο.

1.2 Σκοπός Διπλωματικής Εργασίας

Σκοπός της εργασίας αυτής είναι η υλοποίηση αλγορίθμων για αλυσίδες κοινοποιήσεων (blockchains), οι οποίοι είναι βασισμένοι πάνω σε αλγόριθμους για Κατανεμημένα Αντικείμενα Κατάστιχου (Distributed Ledger Objects) [4] που είναι και η βάση τέτοιων συστημάτων, όπως και η πειραματική αξιολόγηση των αλγορίθμων αυτών σε πραγματικό κατανεμημένο δίκτυο. Για προγραμματισμό υποδοχών χρησιμοποιείται η βιβλιοθήκη ZeroMQ [7].

Στόχος της υλοποίησής μας είναι να μελετηθεί η συμπεριφορά των αλγορίθμων αυτών μέσα από πειραματικά σενάρια. Αυτό γίνεται, ούτως ώστε να αποδειχθεί πως οι αλγόριθμοι αυτοί μπορούν να τρέξουν σε πραγματικό κατανεμημένο δίκτυο, όπως είναι το PlanetLab [8] και άρα δύνανται να χρησιμοποιηθούν ως μια γενικότερη βάση για μελλοντικές υλοποιήσεις τέτοιων τεχνολογιών, τόσο σε θεωρητικό όσο και σε πρακτικό επίπεδο. Επίσης, μέσα από τα πειράματα θέλουμε να γίνει μια σύγκριση των τριών αυτών αλγορίθμων που υλοποιούμε και να διαφανεί κατά πόσο η απόδοσή τους επηρεάζεται από τον βαθμό της συνέπειάς τους.

1.3 Μεθοδολογία

Αρχικά, διαβάστηκαν άρθρα και βιβλία για κατανεμημένα συστήματα [1, 10] και για αλυσίδες κοινοποιήσεων [11], για την απόκτηση γνωστικής βάσης επί του αντικειμένου, και τη βαθύτερη κατανόηση των βασικών εννοιών των συστημάτων αυτών.

Έπειτα, μελετήθηκε εκτενώς το άρθρο που περιγράφει τους αλγόριθμους που υλοποιήθηκαν [4], ούτως ώστε να κατανοηθεί ο ψευδοκώδικας τους, οι διαφορές μεταξύ τους και η ορθότητά τους. Οι διαφορές που έχουν οι αλγόριθμοι αυτοί σχετίζονται με το βαθμό της συνέπειάς τους, ξεκινώντας από την τελική συνέπεια, η οποία είναι και η πιο απλή σε υλοποίηση, και υπόσχεται πως σε κάποιο χρονικό σημείο

όλα τα αντικείμενα κατάστιχου στο δίκτυο θα προσθέσουν μια νέα εγγραφή. Στη συνέχεια, έχουμε τον αλγόριθμο για διαδοχική συνέπεια, που δίνει έμφαση στη σειρά με την οποία θα προστεθούν οι εγγραφές στα αντικείμενα κατάστιχου, ο οποίος και είναι αρκετά πιο περίπλοκος στην υλοποίηση. Τέλος, υπάρχει και ο αλγόριθμος με ατομική συνέπεια, που είναι και ο πιο ψηλός βαθμός συνέπειας που έχουμε. Στον αλγόριθμο αυτό, όλα τα αντικείμενα κατάστιχου στο σύστημα συμπεριφέρονται σαν να είναι ένα μοναδικό αντικείμενο που σέβεται τη χρονική σειρά με την οποία φτάνουν τα αιτήματα, ακόμα και αν δουλεύουν όλα μαζί ταυτόχρονα σε ένα ασύγχρονο σύστημα. Για την ολοκλήρωση του συστήματος, υπάρχουν και αλγόριθμοι για εξωτερικές διεργασίες πελάτη, οι οποίες ουσιαστικά στέλνουν αιτήματα στο δίκτυο για να χρησιμοποιήσουν τα αντικείμενα κατάστιχου.

Ακολούθως, μελετήθηκαν οι βασικές αρχές του προγραμματισμού υποδοχών με τη βοήθεια διαφανειών του μαθήματος ΕΠΛ601 του Πανεπιστημίου Κύπρου, και διαβάστηκαν βιβλία για εισαγωγή στις λειτουργίες της βιβλιοθήκης ZeroMQ [12, 13]. Το βήμα αυτό ακολούθησε εξοικείωση με τη βιβλιοθήκη, με τη μορφή υλοποίησης παραδειγμάτων από τα βιβλία, και ελέγχου τους σε τοπικό δίκτυο με μικρά μηνύματα.

Ύστερα, διαβάστηκε το άρθρο για την Υπηρεσία Ατομικής Πολυεκπομπής (Atomic Broadcast Service) [6], την οποία χρησιμοποιούν και οι τρεις αλγόριθμοι που υλοποιήθηκαν, για την αποστολή μηνυμάτων μεταξύ τους. Ακολούθησε υλοποίηση της υπηρεσίας αυτής, και έλεγχος για να διαπιστωθεί η ορθή λειτουργία της στη συστάδα υπολογιστών του επιβλέποντα καθηγητή, στο τμήμα Πληροφορικής.

Στο σημείο αυτό, βασισμένοι πάνω σε αλγόριθμους που μελετήθηκαν από το [10] υλοποιήθηκε ο Μηχανισμός Ανοχής Σφαλμάτων, ο οποίος και στη συνέχεια ελέγχθηκε για την ορθότητά του μαζί με την ΥΑΠ στη συστάδα υπολογιστών που προαναφέραμε.

Έπειτα, υλοποιήθηκαν οι τρεις αλγόριθμοι που περιγράφονται στο [4] και ελέγχθηκαν για ορθότητα και πάλι στη συστάδα που αναφέρεται πιο πάνω.

Στη συνέχεια, δημιουργήθηκε μέρος στο ερευνητικό δίκτυο PlanetLab Europe, και έγινε προσθήκη μηχανών σε αυτό, αρχικοποίησή τους και εγκατάσταση μεταφραστών για προγραμματισμό, όπως και της ZeroMQ. Επίσης, δημιουργήθηκαν αρχεία εντολών (scripts) για συγχρονισμό των αρχείων κώδικα στις μηχανές του μερίσματος, και αναζητήθηκαν τρόποι για την ταυτόχρονη σύνδεση σε όλες τις μηχανές, ώστε να τρέχουν παράλληλα οι αλγόριθμοι που υλοποιήσαμε. Μόλις το δίκτυο ετοιμάστηκε, έγιναν και έλεγχοι ορθότητας των αλγορίθμων και αφαιρέθηκαν οι μη λειτουργικές μηχανές.

Τέλος, έγιναν οι απαραίτητες αλλαγές στον κώδικα για τους σκοπούς της πειραματικής αξιολόγησης, και ακολούθως εκτελέστηκε πληθώρα πειραμάτων για διαφορετικά σενάρια, έγινε η καταγραφή της απόδοσης των αλγορίθμων, καθώς επίσης και σχολιασμός των αποτελεσμάτων και μια ευρύτερη αξιολόγησή τους.

1.4 Δομή Εργασίας

Στο Κεφάλαιο 1 της εργασίας περιγράφονται τα κίνητρα και ο σκοπός της εργασίας, η μεθοδολογία που ακολουθήθηκε για την ολοκλήρωσή της και η δομή αυτής, ενώ περιλαμβάνεται και ένα γλωσσάρι με έννοιες που παρουσιάζονται συχνά στην εργασία, μαζί με τις αγγλικές ορολογίες τους.

Στο Κεφάλαιο 2 δίνεται σύντομη περιγραφή των προαπαιτούμενων γνώσεων για την κατανόηση της εργασίας, καθώς και των υπηρεσιών που χρησιμοποιήθηκαν για την υλοποίησή της.

Έπειτα, στο Κεφάλαιο 3, γίνεται μια περιγραφή του μοντέλου των αλγορίθμων και της συνέπειάς τους, καθώς επίσης και περιγραφή των ιδίων των αλγορίθμων.

Στο Κεφάλαιο 4 περιγράφεται λεπτομερώς η υλοποίηση των αλγορίθμων, συμπεριλαμβανομένης της μορφής που έχουν τα δεδομένα και τα αντικείμενα, καθώς και οι διάφορες παραδοχές που έγιναν για τους σκοπούς της υλοποίησης.

Στη συνέχεια, στο Κεφάλαιο 5 αναφέρεται η προεργασία που έλαβε χώρα για την πειραματική αξιολόγηση των αλγορίθμων, δίνεται μια περιγραφή των σεναρίων για τα οποία γίνονται οι μετρήσεις, και αναλύονται τα αποτελέσματα της αξιολόγησης.

Τέλος, στο Κεφάλαιο 6 παρουσιάζονται τα συμπεράσματα που προέκυψαν από την ολοκλήρωση της εργασίας αυτής, ενώ επίσης γίνεται αναφορά σε μελλοντική εργασία που μπορεί να γίνει με βάση την παρούσα εργασία, και την επέκταση και βελτίωση κάποιων παραδοχών.

1.5 Γλωσσάρι

Πιο κάτω αναφέρονται έννοιες που εμφανίζονται συχνά στο κείμενο, και οι αντίστοιχες αγγλικές ορολογίες τους.

- Κατανεμημένα συστήματα - Distributed Systems
- Αλυσίδα Κοινοποιήσεων - Blockchain
- Αντικείμενο Κατάστιχου - Ledger Object
- Κατανεμημένο Αντικείμενο Κατάστιχου - Distributed Ledger Object
- Τελική Συνέπεια - Eventual Consistency
- Ατομική Συνέπεια - Atomic Consistency
- Ακολουθιακή Συνέπεια - Sequential Consistency
- Υπηρεσία Ατομικής Πολυεκπομπής - Atomic Broadcast Service
- Μηχανισμός Ανοχής Σφαλμάτων - Fault Tolerance Module
- Κόμβος - Node
- Υποδοχή - Socket
- Διεργασία - Process
- Πελάτης - Client
- Εξυπηρετητής - Server
- Αρχεία εντολών - Scripts

Κεφάλαιο 2

Προαπαιτούμενες γνώσεις

2.1 Κατανεμημένα Συστήματα

2.2 Αλυσίδες Κονοποιήσεων

2.3 Αντικείμενα Κατάστιχου

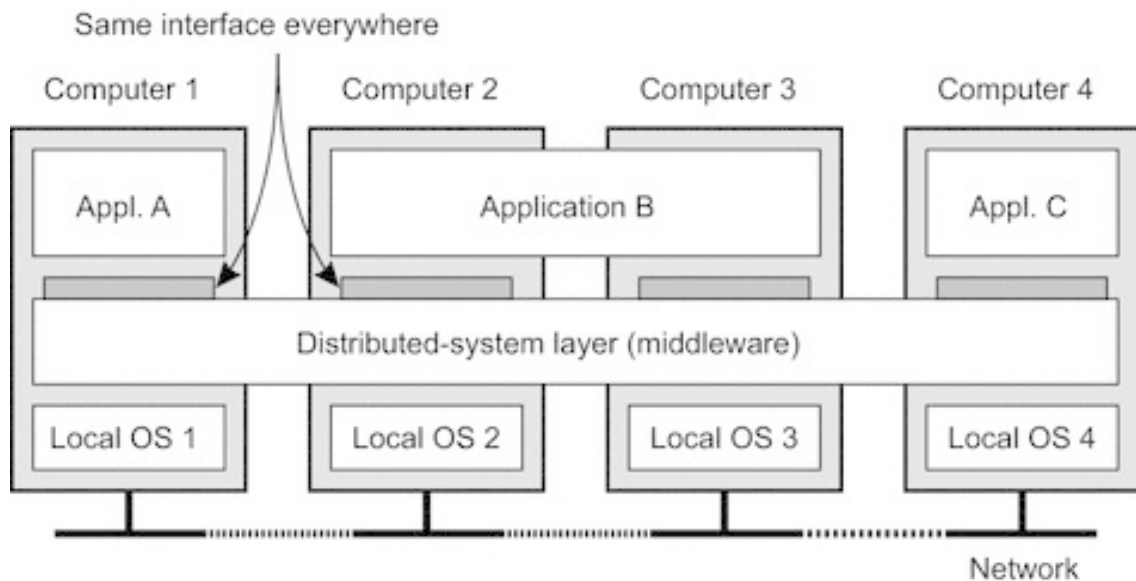
2.4 ZeroMQ

2.5 PlanetLab

2.1 Κατανεμημένα Συστήματα

Τα Κατανεμημένα Συστήματα χαρακτηρίζονται ως ένα σύνολο από υπολογιστές αυτόνομους μεταξύ τους, το οποίο στους χρήστες του δίνει την αίσθηση πως είναι ένα συνεχές σύστημα [1]. Οι υπολογιστές αυτοί, ή κόμβοι, έχουν το δικό τους επεξεργαστή και μνήμη, και μπορούν να εκτελούν παντός είδους διεργασίες. Οι κόμβοι επικοινωνούν ανταλλάσσοντας πληροφορίες μεταξύ τους μέσω ενός δικτύου, ενώ πολλές φορές έχουν ένα “επίπεδο” εφαρμογών, γνωστό και ως ενδιάμεσο λογισμικό (middleware) [2], το οποίο είναι υπεύθυνο, μεταξύ άλλων, για την κατανομή των πόρων του συστήματος και την επικοινωνία των κόμβων μεταξύ τους. Ένα τέτοιο κατανεμημένο σύστημα με ενδιάμεσο λογισμικό φαίνεται και στο Σχήμα 2.1.

Καθώς τέτοια συστήματα βασίζονται πάνω στην επικοινωνία μέσω του δικτύου, περιμένουμε πως σε κάποιες περιπτώσεις τα δεδομένα που στέλνουν οι κόμβοι πιθανώς να φτάσουν με διαφορετική σειρά, να αλλοιωθούν ή να χαθούν, ενώ πάντα υπάρχει και η πιθανότητα κατάρρευσης των καναλιών επικοινωνίας μεταξύ των κόμβων, του δικτύου, ή ακόμα και των ιδίων των κόμβων.



Σχήμα 2.1: Ένα κατακευμαμένο σύστημα με ενδιαμέσο λογισμικό [2]

Ένα κατακευμαμένο πρόγραμμα είναι ένα σύνολο από διεργασίες, που επικοινωνούν μέσω ενός δικτύου, ανταλλάσσοντας μηνύματα μεταξύ τους. Η κάθε διεργασία λειτουργεί αυτόβουλα και ασύγχρονα, και δεν περιμένει μέχρι να παραδοθούν τα μηνύματα στις υπόλοιπες διεργασίες του συστήματος. Αυτό γίνεται γιατί σε τέτοια προγράμματα αναμένεται να υπάρχουν καθυστερήσεις στην επικοινωνία. Επειδή, όπως αναφέραμε και πιο πάνω, είναι πιθανό να καταρρεύσουν κάποιοι κόμβοι του συστήματος, αρκετά κατακευμαμένα προγράμματα υλοποιούν ένα μηχανισμό ανίχνευσης κόμβων οι οποίοι έχουν αποσυνδεθεί από το σύστημα, έτσι ώστε να αποφευχθεί τυχόν κατάρρευση του προγράμματος ως σύνολο.

2.2 Αλυσίδες Κοινοποιήσεων

Οι Αλυσίδες Κοινοποιήσεων περιγράφονται ως αμετάβλητα αντικείμενα κατάστιχων, τα οποία επιτρέπουν συναλλαγές μεταξύ διεργασιών, χωρίς την ανάγκη για κάποιο μεσάζοντα [3]. Η τεχνολογία αυτή έχει αποκτήσει φήμη λόγω του ότι είναι και η βασική τεχνολογία πίσω από κρυπτονομίσματα όπως το Bitcoin [11], τα οποία είδαν μια ραγδαία ανάπτυξη τα τελευταία χρόνια. Έχουν, όμως, χρησιμοποιηθεί για πολλές

περισσότερες εφαρμογές, όπως το Διαδίκτυο των Πραγμάτων (Internet of Things), αλλά και συστήματα αξιολόγησης (reputation systems) [14].

Η βασική αρχή πίσω από την τεχνολογία αυτή είναι πως όλες οι μηχανές που συμμετέχουν στο δίκτυο, δημιουργούν και μοιράζονται ένα αρχείο από κοινού, με κάποιους συγκεκριμένους κανόνες που ονομάζονται πρωτόκολλο συναίνεσης. Οι συναλλαγές γίνονται με την προσθήκη νέων κοινοποιήσεων, που έχουν τη μορφή τμημάτων (blocks), πάνω στο αρχείο αυτό, δημιουργώντας έτσι μια σειρά από κοινοποιήσεις συνδεδεμένες μεταξύ τους, θυμίζοντας έτσι αλυσίδα.

Όπως είπαμε και πιο πριν, οι αλυσίδες κοινοποιήσεων είναι αμετάβλητα αντικείμενα. Οι κοινοποιήσεις που προσκολλούνται στην αλυσίδα είναι αδύνατο να αφαιρεθούν, λόγω και της στενής σύνδεσης που έχουν τα τμήματα της αλυσίδας, καθώς η κάθε μία αποθηκεύει την αριθμητική τιμή κατακερματισμού (hash value) της προηγούμενης κοινοποίησης, ενώ για να αλλάξει ή να διαγραφεί μια κοινοποίηση πρέπει ουσιαστικά να διορθωθεί η τιμή κατακερματισμού σε όλες οι προηγούμενες κοινοποιήσεις, όπως επίσης και οι τιμές στο μέρος της αλυσίδας που αρχίζει από την κοινοποίηση που αφαιρείται. Αυτό καθιστά την αλλαγή ή διαγραφή κοινοποιήσεων από την αλυσίδα κάτι το υπολογιστικά πολύ δύσκολο, άρα πρακτικά αδύνατο.

2.3 Αντικείμενα Κατάστιχου

Αντικείμενο κατάστιχου (AK) ορίζεται ένα αντικείμενο που αποθηκεύει μια απολύτως διατεταγμένη ακολουθία από εγγραφές S , και υποστηρίζει τις λειτουργίες *get* και *append* [4]. Για ένα αντικείμενο κατάστιχου L , η λειτουργία $L.get()$ επιστρέφει την ακολουθία $L.S$ με τις συναλλαγές που έχουν εγγραφεί στο L , ενώ η $L.append(r)$ έχει ως αποτέλεσμα την προσθήκη της εγγραφής r στο τέλος της ακολουθίας $L.S$. Τα αντικείμενα αυτά είναι κοινόχρηστα, και καθώς σε αυτά μπορούν να έχουν πρόσβαση ταυτόχρονα πολλές διεργασίες, θεωρούνται και ταυτόχρονα αντικείμενα (concurrent objects), σύμφωνα με τον ορισμό που δίνεται στο [5].

Η ολοκλήρωση μιας λειτουργίας στα ΑΚ αποτελείται από δύο γεγονότα, την κλήση της, και την απάντηση του κατάστιχου. Η κλήση της λειτουργίας πάντοτε πρέπει να προηγείται της αντίστοιχης απάντησης, και μια λειτουργία Π θεωρείται ολοκληρωμένη μόνο αν η κλήση της Π έχει απαντηθεί από το ΑΚ. Μια ακολουθία από κλήσεις λειτουργιών και απαντήσεις σε χρονική σειρά λέγεται ιστορικό, και μια λειτουργία Π_1 προηγείται μιας άλλης λειτουργίας Π_2 σε ένα ιστορικό (συμβολίζεται ως $\Pi_1 \rightarrow \Pi_2$) αν η απάντηση του ΑΚ για τη λειτουργία Π_1 προηγείται της κλήσης της Π_2 στο ιστορικό αυτό.

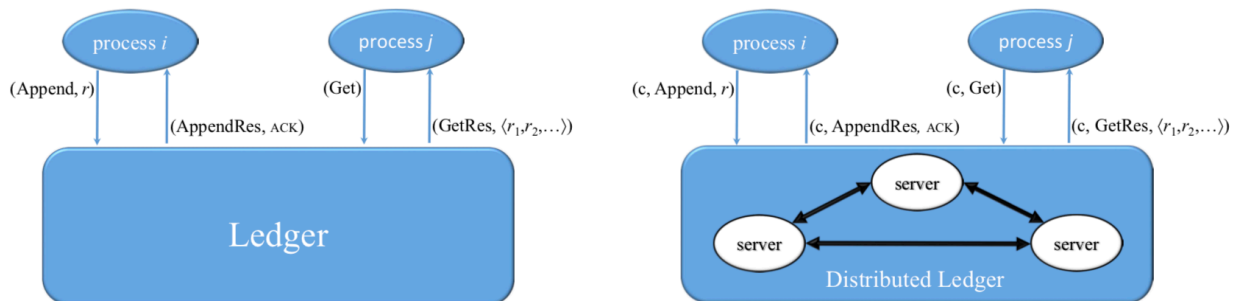
Η κάθε εγγραφή r που αποθηκεύεται στο αντικείμενο κατάστιχου έχει τη μορφή $r = \langle t, p, v \rangle$, όπου t είναι ένας μοναδικός αριθμός που αντιπροσωπεύει την εγγραφή r , p είναι ο αριθμός που αντιπροσωπεύει τη διεργασία που δημιούργησε την εγγραφή, και v είναι τα δεδομένα της εγγραφής αυτής.

Ένα Κατανεμημένο Αντικείμενο Κατάστιχου (ΚΑΚ), όπως ορίζεται και πάλι στο [4], είναι ένα Αντικείμενο Κατάστιχου το οποίο είναι υλοποιημένο σε ένα σύνολο από n διεργασίες τύπου εξυπηρετητή (server), δημιουργώντας έτσι ένα κατανεμημένο σύστημα, με σκοπό να εξυπηρετηθούν πολλές διεργασίες τύπου πελάτη (client) ανά τον κόσμο, οι οποίες καλούν λειτουργίες του κατάστιχου ταυτόχρονα.

Όπως βλέπουμε και δεξιά στο Σχήμα 2.2, οι πελάτες επικοινωνούν με το ΚΑΚ στέλλοντας αίτημα για να καλέσουν τις λειτουργίες του, *get* και *append*, και παίρνουν μια απάντηση στο αίτημα τους από το σύστημα, ενώ οι εξυπηρετητές που υλοποιούν το ΚΑΚ επικοινωνούν μεταξύ τους με εσωτερικά μηνύματα.

Καθώς τα ΚΑΚ είναι αντικείμενα σε κατανεμημένο σύστημα, σχεδιάζονται με γνώμονα το ότι κάποιοι από τους κόμβους πιθανώς να σταματήσουν να λειτουργούν όπως αναμένεται, έτσι το ΑΚ αντιγράφεται σε πολλούς εξυπηρετητές για να συνεχίσει να είναι διαθέσιμο το σύστημα. Για να λειτουργεί ορθά το σύστημα όμως, υποθέτουμε πως μπορεί να καταρρεύσει μόνο ένας αριθμός $f < n$ από εξυπηρετητές, ενώ μια διεργασία πελάτη επικοινωνεί με ένα σύνολο τουλάχιστον $f+1$ εξυπηρετητών, έτσι ώστε να είμαστε σίγουροι πως τουλάχιστον ένας από τους εξυπηρετητές λειτουργεί ορθά και θα απαντήσει στο αίτημα. Για να διασφαλίσουμε την ορθότητα της παράδοσης των

εσωτερικών μηνυμάτων που στέλνονται μεταξύ των εξυπηρετητών που υλοποιούν το ΚΑΚ, χρησιμοποιείται μια υπηρεσία Ατομικής Πολυεκπομπής (Atomic Broadcast) [6], την οποία θα αναλύσουμε πιο ενδελεχώς αργότερα.



Σχήμα 2.2: Η αλληλεπίδραση διεργασιών πελάτη και ΑΚ [4]

2.4 ZeroMQ

Η ZeroMQ (γνωστή και ως ZMQ, 0MQ, ØMQ) [7] είναι μια προγραμματιστική βιβλιοθήκη για υποδοχές (sockets), που έχει δημιουργηθεί με σκοπό να διευκολύνει τη μετάδοση μηνυμάτων μέσω δικτύων και προσφέρει επιπλέον λειτουργίες για τις υποδοχές, από εκείνες που προσφέρουν τα λειτουργικά συστήματα.

Οι πιο σημαντικές για την υλοποίηση της εργασίας επιλογές που προσφέρει η ZeroMQ, είναι το ασύγχρονο μοντέλο επικοινωνίας που υλοποιεί και η ταυτόχρονη μετάδοση μηνυμάτων σε ένα αριθμό συσκευών με βάση διάφορα μοντέλα υποδοχών. Για την εργασία αυτή συγκεκριμένα, χρησιμοποιείται το μοντέλο Εκδότη - Συνδρομητή για να επικοινωνούν μεταξύ τους οι εξυπηρετητές, και το κλασικό μοντέλο Πελάτη - Εξυπηρετητή για αιτήματα από τους πελάτες προς το ΚΑΚ. Τα μοντέλα αυτά, και το πώς δουλεύουν στην υλοποίηση των αντικειμένων θα αναλυθούν με περισσότερη λεπτομέρεια στην ενότητα 5.3.

2.5 PlanetLab

Το PlanetLab [8] είναι ένα κατανεμημένο ερευνητικό δίκτυο, που ιδρύθηκε το 2003 με σκοπό την ανάπτυξη νέων δικτυακών εφαρμογών και υπηρεσιών σε παγκόσμια κλίμακα. Το δίκτυο αυτή τη στιγμή απαριθμεί πάνω από 1000 κόμβους, εκ των οποίων περίπου 50 είναι ενεργοί στο τμήμα της Ευρώπης, PlanetLab Europe [9], ενώ υπολογίζεται πως ανά πάσα στιγμή περισσότερα από 600 έργα αναπτύσσονται στο δίκτυο αυτό.

Οι χρήστες του δικτύου αυτού δημιουργούν μερίσματα (slices) προσθαφαιρώντας μηχανές - κόμβους σε αυτά, δεσμεύοντας έτσι πόρους για το έργο τους. Σε όλες τις μηχανές που έχουν προστεθεί στο μέρισμα δημιουργείται μια “καθαρή” εγκατάσταση λειτουργικού συστήματος τύπου Linux και ένας χρήστης στην εγκατάσταση αυτή με δικαιώματα διαχειριστή, έτσι ώστε να μπορέσει να διαμορφώσει την εγκατάσταση με βάση τις ανάγκες του έργου. Οι χρήστες έχουν πρόσβαση στους κόμβους του μερίσματος με το όνομα χρήστη τους, το οποίο είναι το ίδιο με το όνομα του μερίσματός τους και σε ένα ιδιωτικό κλειδί RSA για πιστοποίηση της ταυτότητας, ενώ η σύνδεση γίνεται μέσω του πρωτοκόλλου SSH για να επιτευχθεί κρυπτογραφημένη και ασφαλής επικοινωνία.

Για την πειραματική αξιολόγηση των αλγορίθμων της εργασίας αυτής έχει δημιουργηθεί ένα μέρισμα (slice) στο PlanetLab Europe.

Κεφάλαιο 3

Μοντέλο και Περιγραφή Αλγορίθμου

3.1 Μοντέλο Αλγορίθμων για Κατανεμημένα Αντικείμενα Κατάστιχου

3.2 Περιγραφή Αλγορίθμων

3.1 Μοντέλο Αλγορίθμων για Κατανεμημένα Αντικείμενα Κατάστιχου

Οι αλγόριθμοι που υλοποιούνται σε αυτή την εργασία, βασίζονται σε ένα ασύγχρονο μοντέλο ανταλλαγής μηνυμάτων, το οποίο απαρτίζουν οι εξυπηρετητές του συστήματος που δημιουργούν ένα κατανεμημένο σύστημα μεταξύ τους και οι πελάτες του συστήματος, οι οποίοι ενώνονται με το δίκτυο και καλούν με αιτήματα τις λειτουργίες που προσφέρει το αντικείμενο κατάστιχου. Το όλο σύστημα αποτελείται από διεργασίες δύο τύπων. Συγκεκριμένα, υπάρχουν n πανομοιότυπες διεργασίες αντικειμένων κατάστιχου L , οι οποίες είναι υλοποιημένες πάνω σε n εξυπηρετητές και παίρνουν ένα μοναδικό ακέραιο αριθμό ταυτότητας $i \in [0, n)$ (για το κατάστιχο με αριθμό i γράφουμε L_i), και m διεργασίες πελάτη C , οι οποίες έχουν και εκείνες με τη σειρά τους τη δική τους μοναδική ταυτότητα $j \in [0, m)$ (για τον πελάτη με ταυτότητα j γράφουμε C_j).

Οι διεργασίες C μπορούν να επικοινωνήσουν μόνο με διεργασίες L με τη μορφή αιτήματος χρήσης του ΑΚ, ενώ οι διεργασίες L επικοινωνούν τόσο με άλλες διεργασίες L για να ανταλλάσσουν εσωτερικά μηνύματα για το δίκτυο και για συναλλαγές που έχουν γίνει με πελάτες, όσο και με διεργασίες C με τη μορφή απάντησης σε αιτήματα που το δίκτυο έχει δεχτεί.

3.1.1 Συνέπεια ΚΑΚ

Καθώς το σύνολο των διεργασιών L υλοποιεί σε ένα σύστημα το αντικείμενο κατάστιχου όπως έχει οριστεί στην Ενότητα 2.3, περιμένουμε από το σύστημα αυτό να είναι συνεπές, δηλαδή η ακολουθία εγγραφών $L.S$ να είναι η ίδια σε όλες τις διεργασίες. Για αυτό τον σκοπό, ορίζονται τρεις διαφορετικοί βαθμοί συνέπειας των αλγορίθμων για το ΚΑΚ, την τελική (eventual), τη διαδοχική (sequential) και την ατομική (atomic) [4].

Για να θεωρείται ένα κατανεμημένο αντικείμενο κατάστιχου, L , διαδοχικά συνεπές, πρέπει να υπάρχει μια μετάθεση σ ενός οποιουδήποτε ιστορικού H_L τέτοια ώστε $\forall \Pi_1, \Pi_2$ που έχουν κληθεί από μια διεργασία p και $\Pi_1 \rightarrow \Pi_2$ στο H_L , τότε η Π_1 εμφανίζεται πριν το Π_2 στο σ . Άρα, με διαδοχική συνέπεια στο L είμαστε σίγουροι πως ανά πάσα στιγμή θα εμφανίζει όλες τις εγγραφές στο $L.S$ με την ίδια ακριβώς σειρά. Προφανώς, με τη διαδοχική συνέπεια δίνεται περισσότερη σημασία στη σειρά με την οποία θα αποθηκευτούν τα μηνύματα στα αντικείμενα, παρά στην πραγματική χρονική σειρά με την οποία φτάνουν τα αιτήματα στο ΚΑΚ.

Ένα κατανεμημένο αντικείμενο κατάστιχου L είναι τελικά συνεπές, εάν υπάρχει ένα πλήρες ιστορικό H'_L που επεκτείνει το H_L (ένα ιστορικό X επεκτείνει ένα ιστορικό Y , αν το Y είναι πρόθεμα του X), τέτοιο ώστε για όλα τα πλήρη ιστορικά H''_L που επεκτείνουν το H'_L , οι ολοκληρωμένες λειτουργίες $get()$ στο ιστορικό που προκύπτει από την αφαίρεση του H'_L από το H''_L να περιέχουν όλες ανεξαιρέτως τις εγγραφές r που προέκυψαν από όλες τις λειτουργίες $L.append(r) \in H_L$. Αυτό για το L πρακτικά σημαίνει πως αν μια εγγραφή r αποθηκευτεί επιτυχημένα, τότε σε κάποιο πεπερασμένο χρονικό σημείο, κλήσεις $L.get()$ από οποιαδήποτε από τις διεργασίες που υλοποιούν το L , θα περιέχουν την r στην ίδια θέση.

Τέλος, αν ένα κατανεμημένο αντικείμενο κατάστιχου L έχει ατομική συνέπεια, τότε πρέπει να υπάρχει μια μετάθεση σ ενός οποιουδήποτε ιστορικού H_L τέτοια ώστε $\forall \Pi_1, \Pi_2$ που έχουν κληθεί από μια διεργασία p , και $\Pi_1 \rightarrow \Pi_2$ στο H_L , τότε και στο σ η Π_1

εμφανίζεται πριν την Π_2 . Με την ατομική συνέπεια, για να απαντηθεί ένα μήνυμα, πρέπει πρώτα να διασφαλίσουμε πως έχουν όλες οι μηχανές του συστήματος που υλοποιούν το L , τη γνώση πως το μήνυμα αυτό θα απαντηθεί, με τη βοήθεια και της υπηρεσίας Ατομικής Πολυεκπομπής. Έτσι, δίνεται η αίσθηση στους χρήστες πως το ΚΑΚ συμπεριφέρεται σαν να είναι ένα και μοναδικό αντικείμενο κατάστιχου, που πραγματοποιεί τις λειτουργίες του με χρονική σειρά, παρόλο που οι κλήσεις των λειτουργιών μπορούν να γίνονται ταυτόχρονα σε όλα τα αντικείμενα που υλοποιούν το L .

3.2 Περιγραφή Αλγορίθμων

Οι αλγόριθμοι που περιγράφονται είναι τρεις στο σύνολο, ένας για κάθε βαθμό συνέπειας, όπως αυτοί έχουν οριστεί στην προηγούμενη ενότητα. Όλοι οι αλγόριθμοι περιλαμβάνουν ένα κομμάτι κώδικα για τους εξυπηρετητές που θα τρέχουν το ΚΑΚ, και κώδικα για διεργασία πελάτη.

3.2.1 Υπηρεσία Ατομικής Πολυεκπομπής

Προτού προχωρήσουμε με την περιγραφή του κάθε αλγόριθμου ξεχωριστά, θα περιγράψουμε την Υπηρεσία Ατομικής Πολυεκπομπής (ΥΑΠ) [6], η οποία χρησιμοποιείται σε όλους τους αλγόριθμους, έτσι ώστε να σταλούν και να παραδοθούν ορθά τα μηνύματα που ανταλλάσσονται μεταξύ των διεργασιών. Η υπηρεσία προσφέρει δύο βασικές λειτουργίες: την $ABroadcast(m)$, που μεταδίδει ένα μήνυμα m σε όλους τους εξυπηρετητές που ανήκουν σε ένα σύνολο S , και την $ADeliver(m)$, που παραδίδει ένα μήνυμα m στον εξυπηρετητή που καλεί τη διαδικασία. Γίνεται χρήση της υπηρεσίας αυτής, καθώς πληροί τις ιδιότητες που χρειαζόμαστε, για να είμαστε βέβαιοι πως τα μηνύματα θα παραδοθούν σίγουρα και με εντελώς σωστή σειρά, αποφεύγοντας διπλά μηνύματα [4, 6].

Βασικό συστατικό της λειτουργίας της Ατομικής Πολυεκπομπής είναι μια μορφή μετρητή, αρχικά μηδενικού, που η κάθε διεργασία διατηρεί για όλους τους εξυπηρετητές, ο οποίος βρίσκεται στο σύνολο S που ανταλλάζει μηνύματα, ονόματι λογικό ρολόι (γράφουμε $LC_p[q]$ για το λογικό ρολόι της διεργασίας q , όπως το βλέπει η διεργασία p). Επίσης, για την κάθε διεργασία p υπάρχουν δύο “αποθήκες” μηνυμάτων, το *received* όπου αποθηκεύονται τα μηνύματα που έχουν φτάσει στην p αλλά δεν έχουν τύχει ακόμα επεξεργασίας, και το *delivered*, όπου προσθέτονται τα μηνύματα που έχουν παραδοθεί στην p με τη διαδικασία $ADeliver()$.

Η διαδικασία $ABroadcast(m)$ (Σχήμα 3.1) είναι φαινομενικά απλή. Για κάθε μήνυμα m που θα σταλεί, γίνεται αύξηση του $LC_p[p]$ κατά ένα, και η νέα τιμή του ονομάζεται χρονική σήμανση του μηνύματος m (γράφουμε $ts(m)$). Στη συνέχεια, γίνεται ενθυλάκωση του m με το αντίστοιχο $ts(m)$ του, και το τελικό προϊόν στέλλεται σε όλους τους εξυπηρετητές στο σύνολο S . Υποθέτουμε πως έχουμε κανάλια επικοινωνίας που επικοινωνούν με τη μέθοδο πρώτης εισαγωγής - πρώτης εξαγωγής (First In First Out) με τους υπόλοιπους εξυπηρετητές.

Οι υπόλοιποι εξυπηρετητές με τη σειρά τους, μόλις λάβουν το ενθυλακωμένο μήνυμα m , “κουρδίζουν” τα λογικά ρολόγια τους με το $ts(m)$ που μόλις πήραν από τη διεργασία που έστειλε το μήνυμα, και προσθέτουν το μήνυμα m στο *received*. Έπειτα, κάθε μήνυμα m' στο *received* που δεν έχει τύχει ακόμα επεξεργασίας και το $ts(m')$ είναι μικρότερο από το μικρότερη τιμή που υπάρχει στα LC της διεργασίας, προστίθεται σε μια ενδιάμεση “αποθήκη” μηνυμάτων, το *deliverable*. Τέλος, παραδίδεται ($ADeliver(x)$) κάθε μήνυμα x που βρίσκεται στο *deliverable*, σε αύξουσα σειρά $ts(x)$ και σε περίπτωση ισότητας των χρονικών σημάνσεων, σε αύξουσα σειρά ως προς την ταυτότητας αποστολέα του x .

Τέλος, πρέπει να σημειωθεί πως για να δουλέψει σωστά η υπηρεσία Ατομικής Πολυεκπομπής σε ένα δίκτυο με n εξυπηρετητές, απαιτείται η πλειοψηφία των εξυπηρετητών να εργάζεται όπως αναμένεται. Ο μέγιστος αριθμός διεργασιών που μπορούν να καταρρεύσουν είναι το πολύ $f < n/2$. [10]

Senders and destinations (code of process p ; assumes FIFO channels):

```

Initialization:
  receivedp := ∅                                { Messages received by process p }
  deliveredp := ∅                                { Messages delivered by process p }
  LCp[p1 ... pn] := {0, ..., 0}    { LCp[q]: logical clock of process q, as seen by p }
procedure TO-multicast( $m$ )                { To TO-multicast a message  $m$  }
  LCp[p] := LCp[p] + 1
  ts( $m$ ) := LCp[p]
  send FIFO ( $m$ , ts( $m$ )) to all
when receive ( $m$ , ts( $m$ ))
  LCp[p] := max(ts( $m$ ), LCp[p]) + 1
  LCp[sender( $m$ )] := ts( $m$ )
  receivedp := receivedp ∪ { $m$ }
  deliverable := ∅
  for each message  $m'$  in receivedp \ deliveredp do
    if ts( $m'$ ) ≤ minq∈Π LCp[q] then
      deliverable := deliverable ∪ { $m'$ }
  deliver all messages in deliverable, in increasing order of (ts( $m$ ), sender( $m$ ))
  deliveredp := deliveredp ∪ deliverable

```

Σχήμα 3.1: Ψευδοκώδικας για την υπηρεσία Ατομικής Πολυεκπομής [10]

3.2.2 Περιγραφή

Στη συνέχεια θα περιγράψουμε τους αλγόριθμους για κάθε βαθμό συνέπειας, όπως αυτοί δίνονται στο [4]. Σημειώνεται πως, σε κάθε περίπτωση, κατά την αρχικοποίηση του δικτύου, η κάθε διεργασία i που υλοποιεί το ΑΚ, έχει κενή την ακολουθία $L_i.S$ με τις αποθηκευμένες εγγραφές της.

3.2.2.1 ΚΑΚ με Τελική Συνέπεια

Ο αλγόριθμος ΚΑΚ που ικανοποιεί τελική συνέπεια είναι απλός. Ο εξυπηρετητής που υλοποιεί το αντικείμενο περιμένει επ' άπειρον μήνυμα από διεργασία πελάτη, ή παράδοση με την ΥΑΠ ενός από τα μηνύματα, όπως βλέπουμε και στον ψευδοκώδικα στο Σχήμα 3.2.

Εάν φτάσει στον εξυπηρετητή L_i μήνυμα από πελάτη p που περιέχει αίτηση GET, τότε επιστρέφεται στον p η λίστα $L_i.S$, ενώ αν πρόκειται για αίτηση APPEND για εγγραφή r ,

τότε η r μεταδίδεται στο δίκτυο με τη χρήση της $ABroadcast(r)$ της ΥΑΠ, και έπειτα επιστρέφεται στον πελάτη p αναγνώριση (ACK). Όταν η ΥΑΠ παραδώσει στον εξυπηρετητή μήνυμα που αναφέρεται σε εγγραφή r , τότε εάν η r δεν περιέχεται ήδη στο $L_i.S$, την αποθηκεύουμε στο κατάστιχο.

Ο αλγόριθμος για τη διεργασία του πελάτη είναι εξίσου απλός. Αρχικά, ο πελάτης αρχικοποιεί ένα μοναδικό κωδικό c για τα αιτήματά του, ο οποίος αυξάνεται κάθε φορά που στέλλει κάποιο αίτημα στο δίκτυο. Ο κάθε πελάτης έχει ορισμένο ένα σύνολο από εξυπηρετητές L , $L \subseteq S$, μεγέθους μεγαλύτερου ή ίσου του $f+1$, όπου f είναι ο μικρότερος αριθμός εξυπηρετητών στο δίκτυο για να λειτουργεί ορθά το σύστημα, στους οποίους και μπορεί να στείλει τα αιτήματά του. Αυτό γίνεται ώστε να είμαστε σίγουροι πως τουλάχιστον ένας από τους εξυπηρετητές στο L θα λειτουργεί κανονικά και συνεπώς θα απαντήσει στο αίτημα.

Για να στείλει ένας πελάτης κάποιο αίτημα, όπως βλέπουμε και στο Σχήμα 3.3, πρέπει να στείλει ένα μήνυμα στο σύνολο L , το οποίο θα περιέχει τον κωδικό αιτήματος c και το είδος αιτήματος, μαζί με την εγγραφή r που θέλει να στείλει, αν πρόκειται για αίτημα APPEND. Έπειτα, περιμένει μέχρι να λάβει απάντηση από ένα εξυπηρετητή i , $i \in L$, η οποία θα περιέχει την απάντηση στο αίτημα του, δηλαδή τη λίστα με εγγραφές σε περίπτωση GET, ή επιβεβαίωση πως η εγγραφή έχει προστεθεί στο ΚΑΚ σε περίπτωση APPEND.

Code 6 Eventually Consistent Distributed Ledger $\mathcal{L}$;
Code for Server $i \in \mathcal{S}$

```

1: Init:  $S_i \leftarrow \emptyset$ 
2: receive ( $c$ , GET) from process  $p$ 
3:   send response ( $c$ , GETRES,  $S_i$ ) to  $p$ 
4: receive ( $c$ , APPEND,  $r$ ) from process  $p$ 
5:    $ABroadcast(r)$ 
6:   send response ( $c$ , APPENDRES, ACK) to  $p$ 
7: upon (ADeliver( $r$ )) do
8:   if  $r \notin S_i$  then  $S_i \leftarrow S_i || r$ 

```

Σχήμα 3.2: Ψευδοκώδικας για εξυπηρετητή i που υλοποιεί ΚΑΚ με Τελική Συνέπεια [4]

Code 5 External Interface of a Distributed Ledger Object $\mathcal{L}$ Executed by a Process p

1: $c \leftarrow 0$	8: function $\mathcal{L}.\text{append}(r)$
2: Let $L \subseteq \mathcal{S} : L \geq f + 1$	9: $c \leftarrow c + 1$
3: function $\mathcal{L}.\text{get}()$	10: send request (c, APPEND, r) to the servers in
4: $c \leftarrow c + 1$	L
5: send request (c, GET) to the servers in L	11: wait response $(c, \text{APPENDRES}, res)$ from
6: wait response (c, GETRES, V) from some	some $i \in L$
$i \in L$	12: return res
7: return V	

Σχήμα 3.3: Ψευδοκώδικας για διεργασία πελάτη που χρησιμοποιεί το KAK [4]

3.2.2.2 KAK με Ατομική Συνέπεια

Ο αλγόριθμος για Ατομική Συνέπεια υλοποιεί δύο επιπλέον δομές για αποθήκευση μηνυμάτων, τις *get_pending* και *pending*, οι οποίες κρατούν τα μηνύματα των πελατών μέχρι την παράδοση τους, ανάλογα με το είδος του αιτήματος.

Ο κώδικας για το KAK μοιάζει πολύ με τον πιο πάνω αλγόριθμο, όμως έχουμε περισσότερα μηνύματα να στέλλονται και να παραδίδονται, καθώς πλέον εκπέμπονται με την ΥΑΠ και αιτήματα τύπου GET για να εξυπηρετηθούν αργότερα.

Όπως φαίνεται και στο Σχήμα 3.4, ένας εξυπηρετητής L_i που υλοποιεί AK με Ατομική Συνέπεια, περιμένει μέχρι να λάβει κάποιο μήνυμα από πελάτη, ή να παραδοθεί μήνυμα από την ΥΑΠ. Αν παραλάβει μήνυμα από πελάτη p , το μήνυμα μεταδίδεται με την ΥΑΠ, και εάν πρόκειται για αίτημα GET, τότε προστίθεται στη δομή *get_pending*, ενώ αν είναι αίτημα APPEND με εγγραφή r , το μήνυμα προστίθεται στη δομή *pending*.

Όταν η ΥΑΠ παραδώσει μήνυμα που αντιστοιχεί σε αίτημα GET, ελέγχεται η δομή *get_pending* για να διαπιστωθεί αν υπάρχει το αντίστοιχο μήνυμα. Αν ναι, τότε η λίστα $L_i.S$ αποστέλλεται στον πελάτη p , και το μήνυμα αφαιρείται από την *get_pending*.

Αν η ΥΑΠ παραδώσει μήνυμα που αντιστοιχεί σε APPEND για εγγραφή r , τότε αν η r δεν υπάρχει στην $L_i.S$, την προσθέτουμε σε αυτήν. Επιπλέον, εάν το αντίστοιχο αίτημα υπάρχει στη δομή *pending*, στέλλεται πίσω στον πελάτη επιβεβαίωση για το αίτημα και ακολούθως αφαιρείται από τη δομή *pending*.

Έτσι, βεβαιώνουμε πως η διεργασία που έλαβε αρχικά το αίτημα του πελάτη θα είναι και αυτή που τελικά θα απαντήσει, παρ' όλο που όλες οι διεργασίες έχουν γνώση του ότι υπάρχει το αίτημα αυτό.

Η διεργασία του αντίστοιχου πελάτη για το ΚΑΚ με Ατομική Συνέπεια είναι ακριβώς η ίδια με αυτή που περιγράφεται στην Ενότητα 3.2.2.1 και φαίνεται στο Σχήμα 3.3, και δε χρειάζεται κάποια αλλαγή για να λειτουργήσει το σύστημα όπως αναμένεται.

Code 8 Atomic Distributed Ledger; Code for Server i

1: Init: $S_i \leftarrow \emptyset$; $get_pending_i \leftarrow \emptyset$; $pending_i \leftarrow \emptyset$ 2: receive (c , GET) from process p 3: ABroadcast(get , p , c) 4: add (p , c) to $get_pending_i$ 5: upon (ADeliver(get , p , c)) do 6: if (p , c) $\in$ $get_pending_i$ then 7: send response (c , GETRES, S_i) to p 8: remove (p , c) from $get_pending_i$	9: receive (c , APPEND, r) from process p 10: ABroadcast($append$, r) 11: add (c , r) to $pending_i$ 12: upon (ADeliver($append$, r)) do 13: if $r \notin S_i$ then 14: $S_i \leftarrow S_i \parallel r$ 15: if $\exists (c, r) \in pending_i$ then 16: send response (c , APPENDRES, ACK) to $r.p$ 17: remove (c , r) from $pending_i$
---	---

Σχήμα 3.4: Ψευδοκώδικας για εξυπηρετητή i που υλοποιεί ΚΑΚ με Ατομική Συνέπεια [4]

3.2.2.3 ΚΑΚ με Ακολουθιακή Συνέπεια

Ο αλγόριθμος για ΚΑΚ με Ακολουθιακή Συνέπεια ίσως να είναι και ο πιο πολύπλοκος από τους τρεις που περιγράφονται, καθώς και εδώ υπάρχουν οι ιδέες των δομών $get_pending$ και $pending$, μαζί με επιπλέον παραμέτρους στα μηνύματα των πελατών.

Πρώτα, να αναφέρουμε πως ο κώδικας του πελάτη είναι σχεδόν ο ίδιος με τους προηγούμενους δύο αλγόριθμους, με μια επιπλέον προσθήκη, όπως φαίνεται και δεξιά στο Σχήμα 3.5: υπάρχει ένας μετρητής l_{last} που αποθηκεύει τον αριθμό της τελευταίας εγγραφής που υπάρχει στο ΑΚ. Οι λειτουργίες του πελάτη παραμένουν οι ίδιες, με κάποιες μικρές αλλαγές. Συγκεκριμένα, όταν στέλλονται αιτήματα GET, εκτός από τον κωδικό αιτήματος c περνά σαν παράμετρος στο μήνυμα και το l_{last} , το οποίο ενημερώνεται με το μέγεθος της λίστας που παίρνει σαν απάντηση στο αίτημα.

Επίσης, όταν ο πελάτης λάβει απάντηση σε αίτημα APPEND, υπάρχει μια επιπλέον παράμετρος στην απάντηση, η *pos*, που αντιστοιχεί στη θέση της σειράς S του ΑΚ που έχει προστεθεί η εγγραφή r . Έτσι, μόλις φτάσει η απάντηση, ο μετρητής l_{last} ενημερώνεται με τη θέση *pos*.

Όσο αφορά στο ΑΚ, ο αλγόριθμος για Ακολουθιακή Συνέπεια, τον ψευδοκώδικα του οποίου βλέπουμε αριστερά στο Σχήμα 3.5, απαιτεί ο εξυπηρετητής L_i να περιμένει και πάλι, μέχρι είτε να παραλάβει μήνυμα από πελάτη p , ή να παραδοθεί μήνυμα από την ΥΑΠ. Εάν το μήνυμα που φτάνει είναι αίτημα GET, τότε ελέγχεται το $p.l_{last}$, και αν αυτό είναι μικρότερο ή ίσο με το μέγεθος της $L_i.S$, τότε απαντά στον p με την $L_i.S$. Σε αντίθετη περίπτωση, το μήνυμα αυτό αποθηκεύεται στην *get_pending*.

Αν, πάλι, το μήνυμα που φτάσει αντιστοιχεί σε αίτημα APPEND για εγγραφή r , τότε θα γίνει η μετάδοση της r με την ΥΑΠ και το μήνυμα θα αποθηκευτεί στη δομή *pending*.

Μόλις η ΥΑΠ παραδώσει μήνυμα που αντιστοιχεί σε αίτημα APPEND για εγγραφή r , τότε όπως και στις προηγούμενες περιπτώσεις, αν η r δεν υπάρχει στην $L_i.S$, θα πρέπει να την προσθέσουμε. Έπειτα, αν το αντίστοιχο μήνυμα υπάρχει στη δομή *pending*, απαντά με επιβεβαίωση στον πελάτη p και το μήνυμα αφαιρείται από την *pending*. Τέλος, αν υπάρχουν μηνύματα για αιτήματα GET στη δομή *get_pending*, για τα οποία πλέον το μέγεθος της $L_i.S$ είναι μεγαλύτερο από το l_{last} του αιτήματος, απαντούμε στο αίτημα αυτό με τη λίστα $L_i.S$ και αφαιρούμε το αντίστοιχο μήνυμα από την *get_pending*.

Και πάλι, με τη διαδικασία αυτή είμαστε σίγουροι πως ο εξυπηρετητής που έλαβε αρχικά το αίτημα θα απαντήσει σε αυτό, ενώ είμαστε σίγουροι και για τα αιτήματα GET πως θα είναι ενήμερα με την τελευταία εγγραφή που προστέθηκε στο σύστημα.

3.2.2.4 Επικυρωμένο Κατάστιχο

Στο [4], εκτός από τους αλγόριθμους που περιγράψαμε πιο πάνω, αναφέρεται και αλγόριθμος για αντικείμενο επικυρωμένου κατάστιχου. Ο αλγόριθμος αυτός

Code 9 Sequentially Consistent Distributed Ledger;
Code for Server i

```
1: Init:  $S_i \leftarrow \emptyset$ ;  $pending_i \leftarrow \emptyset$ ;  $get\_pending_i \leftarrow \emptyset$ 
2: receive ( $c$ , GET,  $\ell$ ) from process  $p$ 
3:   if  $|S_i| \geq \ell$  then
4:     send response ( $c$ , GETRES,  $S_i$ ) to  $p$ 
5:   else
6:     add ( $c, p, \ell$ ) to  $get\_pending_i$ 
7: receive ( $c$ , APPEND,  $r$ ) from process  $p$ 
8:   ABroadcast( $c, r$ )
9:   add ( $c, r$ ) to  $pending_i$ 
10: upon (ADeliver( $c, r$ )) do
11:   if  $r \notin S_i$  then  $S_i \leftarrow S_i \parallel r$ 
12:   if ( $c, r$ )  $\in pending_i$  then
13:     send resp. ( $c$ , APPENDRES, ACK,  $|S_i|$ ) to
       $r.p$ 
14:     remove ( $c, r$ ) from  $pending_i$ 
15:   if  $\exists (c', p, \ell) \in get\_pending_i : |S_i| \geq \ell$  then
16:     send response ( $c'$ , GETRES,  $S_i$ ) to  $p$ 
17:     remove ( $c', p, \ell$ ) from  $get\_pending_i$ 
```

Code 10 External Interface for Sequential Consistency Executed by a Process p

```
1:  $c \leftarrow 0$ ;  $\ell_{last} \leftarrow 0$ 
2: Let  $L \subseteq \mathcal{S} : |L| \geq f + 1$ 
3: function  $\mathcal{L}.get()$ 
4:    $c \leftarrow c + 1$ 
5:   send request ( $c$ , GET,  $\ell_{last}$ ) to the servers in
      $L$ 
6:   wait response ( $c$ , GETRES,  $V$ ) from some
      $i \in L$ 
7:    $\ell_{last} \leftarrow |V|$ 
8:   return  $V$ 
9: function  $\mathcal{L}.append(r)$ 
10:   $c \leftarrow c + 1$ 
11:  send request ( $c$ , APPEND,  $r$ ) to the servers in
      $L$ 
12:  wait response ( $c$ , APPENDRES,  $res, pos$ ) from
     some  $i \in L$ 
13:   $\ell_{last} \leftarrow pos$ 
14:  return  $res$ 
```

Σχήμα 3.5: Αριστερά, ψευδοκώδικας εξυπηρετητή που υλοποιεί ΚΑΚ με Ακολουθιακή Συνέπεια. Δεξιά, ο αντίστοιχος ψευδοκώδικας για τη διεργασία πελάτη [4]

υλοποιείται με τη χρήση ενός από τους προαναφερθέντες αλγόριθμους για ΑΚ, και η βασική του προσθήκη είναι μία συνάρτηση $Valid()$, η οποία ελέγχει αν τα δεδομένα των εγγραφών στο ΑΚ είναι συνεπή μετά την προσθήκη της r σε αυτό. Αν ναι, συνεχίζει κανονικά τη διαδικασία απαντώντας με επιβεβαίωση στον πελάτη. Στην αντίθετη περίπτωση, απαντά με άρνηση επιβεβαίωσης (NACK) στον πελάτη και η εγγραφή δεν προστίθεται στην ακολουθία S .

Στις υλοποιήσεις που πραγματοποιήθηκαν για την εργασία αυτή, υπάρχει η έννοια της συνάρτησης $Valid()$ στους αλγόριθμους με τη μορφή ελέγχου για διπλή εγγραφή στο ΑΚ. Αν αναγνωριστεί πως δύο συνεχόμενα αιτήματα από τον ίδιο πελάτη προσπαθούν να προσθέσουν την ίδια ακριβώς εγγραφή, επιστρέφεται άρνηση επιβεβαίωσης στον πελάτη και η εγγραφή δε θα προστεθεί στο ΑΚ.

Κεφάλαιο 4

Υλοποίηση Αλγορίθμων

- 4.1 Μορφή Μηνυμάτων
 - 4.2 Λειτουργίες Συστήματος
 - 4.3 Προγραμματισμός Υποδοχών με ZeroMQ
 - 4.4 Μηχανισμός Ανοχής Σφαλμάτων
 - 4.5 Δομές Δεδομένων
 - 4.6 Προγραμματισμός Διεργασιών Αντικειμένων Κατάστιχου
 - 4.7 Προγραμματισμός Διεργασιών Πελάτη
 - 4.8 Παραδοχές για Υλοποίηση
-

Στο κεφάλαιο αυτό αναλύεται η υλοποίηση των αλγορίθμων για Αλυσίδες Κοινοποιήσεων, που έγινε με τη χρήση των αλγορίθμων στο [4] όπως τους έχουμε περιγράψει στο προηγούμενο κεφάλαιο, και η πειραματική αξιολόγησή τους σε πραγματικό καταναμημένο δίκτυο, όπως το PlanetLab. Οι αλγόριθμοι έχουν υλοποιηθεί εξ' ολοκλήρου στη γλώσσα C, χρησιμοποιώντας τη χαμηλού επιπέδου διεπαφή (API) της βιβλιοθήκης ZeroMQ (αν και υπάρχουν ψηλού επιπέδου συνδέσεις της ZeroMQ για πολλές γλώσσες, δεν έχει χρησιμοποιηθεί κάποια) για τον προγραμματισμό των υποδοχών.

4.1 Μορφή Μηνυμάτων

Όλα τα μηνύματα που ανταλλάσσονται μεταξύ των διεργασιών είναι οργανωμένα σε τμήματα (blocks) μεγέθους 256 bytes, για να προσομοιώνεται η λειτουργία αλυσίδας

κοινοποιήσεων με σταθερό μήκος τμήματος. Όσα μηνύματα χρειάζονται μικρότερο χώρο στη μνήμη από ένα τμήμα, γεμίζουν τον υπόλοιπο χώρο με χαρακτήρες συμπλήρωσης (padding). Για μηνύματα μεγαλύτερα των 256 bytes, κάτι που παρατηρούμε πολλές φορές σε απαντήσεις για αιτήματα GET, θα δημιουργηθούν περισσότερα από ένα τμήματα. Μεταξύ των διαφόρων πεδίων που έχουμε στα μηνύματα, υπάρχει ένας χαρακτήρας '/', που λειτουργεί σαν διαχωριστικό (delimiter).

Τα μηνύματα m που ανταλλάσσονται μεταξύ των εξυπηρετητών έχουν την εξής μορφή:

m (size=256): $\langle header || server \#, timestamp, operation type, data, padding \rangle$

Header για τα μηνύματα των εξυπηρετητών είναι πάντα ο χαρακτήρας 'S'. Έτσι αναγνωρίζουμε πως πρόκειται για εσωτερικό μήνυμα που έχει στείλει ένας από τους εξυπηρετητές του συστήματος. *Server #* είναι ο μοναδικός αριθμός του εξυπηρετητή αποστολέα και *timestamp* είναι ο αριθμός που είχε το λογικό ρολόι του εκείνη τη στιγμή, όπως το έχουμε περιγράψει στην ενότητα 3.2.1, για τους σκοπούς της ΥΑΠ [10]. Το *operation type* πρόκειται για έναν ακέραιο μεταξύ 0 και 5, ο οποίος αντιστοιχεί σε μια λειτουργία από τις ακόλουθες: NO_OP, GET_OP, GETK_OP, APP_OP, SYNC_OP, TERM_OP. Τέλος, στο κομμάτι *data* αποθηκεύονται τα απαραίτητα δεδομένα για κάθε λειτουργία. Οι λειτουργίες του συστήματος θα αναλυθούν με περισσότερη λεπτομέρεια σε επόμενη ενότητα.

Τα μηνύματα *req* που μπορούν να στείλουν οι πελάτες προς τους εξυπηρετητές του συστήματος έχουν την ακόλουθη μορφή:

req (size=256): $\langle request \#, operation type, data, padding \rangle$

Το *request #* πρόκειται για τον μοναδικό κωδικό αιτήματος που δημιουργούν οι πελάτες για τα αιτήματα που δημιουργούν και έχει τη μορφή $\langle c_c.i \rangle$ για πελάτη c , όπου i είναι ο αριθμός αιτήματος που στέλλει ο πελάτης, όπως περιγράφεται στους αλγορίθμους του προηγούμενου κεφαλαίου. *Operation type* είναι ο τύπος λειτουργίας που αιτείται ο πελάτης από το ΑΚ, ενώ *data* είναι, και πάλι, τα δεδομένα που χρειάζονται για τη

διεκπεραίωση της λειτουργίας από τον εξυπηρετητή. Οι τύποι λειτουργιών που ορίζονται για διεργασίες πελάτη είναι οι `GET_OP`, `GETK_OP`, και `APP_OP`, που αντιστοιχούν στους ακέραιους αριθμούς 1 μέχρι 3.

Τέλος, τα μηνύματα *resp* που μπορούν οι εξυπηρετητές να απαντήσουν στους πελάτες έχουν ως εξής:

resp (size=256): *<client #, request #, data, (pos), padding>*

Client # είναι ο μοναδικός αριθμός για τη διεργασία πελάτη που είχε στείλει το αρχικό αίτημα, και *request #* είναι ο κωδικός για το αίτημα, όπως έχουμε σημειώσει και πιο πάνω. *Data* είναι, προφανώς, τα δεδομένα που επιστρέφει η λειτουργία που αιτήθηκε, ενώ για τον πελάτη που ικανοποιεί Ακολουθιακή Συνέπεια προσθέτουμε ένα ακόμα πεδίο, το *pos*, όπως ζητείται από την περιγραφή του αλγορίθμου. Να σημειωθεί εδώ πως δε βλέπουμε κάπου τον κωδικό του εξυπηρετητή (ίσως εκτός από αναφορά στο κομμάτι *data* αναλόγως της λειτουργίας), καθώς δε χρειάζεται στον πελάτη η πληροφορία πως του έχει απαντήσει συγκεκριμένος εξυπηρετητής.

4.2 Λειτουργίες συστήματος

Εδώ θα αναλύσουμε όλες τις λειτουργίες που λαμβάνουν χώρα στο δίκτυο. Οι λειτουργίες αυτές, όπως αναφέραμε και πριν, είναι έξι: `NO_OP`, `GET_OP`, `GETK_OP`, `APP_OP`, `SYNC_OP` και `TERM_OP`, και η κάθε μία από αυτές στα μηνύματα που στέλλονται αντιστοιχεί σε ένα αριθμό από 0 μέχρι και 5, με αυτή τη σειρά. Οι λειτουργίες αυτές ορίζονται στον κώδικα σαν νέος τύπος `operation`, ο οποίος ουσιαστικά είναι μια απαρίθμηση (enumeration) των λειτουργιών.

```
typedef enum {NO_OP, GET_OP, GETK_OP, APP_OP, SYNC_OP, TERM_OP}
operation;
```

Ο NO_OP είναι κωδικός για κενή λειτουργία. Ο κωδικός της χρησιμοποιείται αποκλειστικά για την ενδοεπικοινωνία των εξυπηρετητών, αν και στον αλγόριθμο ΚΑΚ με Τελική Συνέπεια, όλα τα μηνύματα που έρχονται από διαδικασίες πελατών μετατρέπουν τον κωδικό λειτουργίας τους σε NO_OP για σκοπούς απλοποίησης του κώδικα. Τα δεδομένα που παίρνει η λειτουργία μπορεί να είναι οτιδήποτε.

Η GET_OP αντιστοιχεί σε αίτηση για τη λειτουργία *get()* στο ΑΚ και μπορεί να χρησιμοποιηθεί τόσο από πελάτες, όσο και εξυπηρετητές. Τα δεδομένα που παίρνει η GET_OP μπορεί να είναι οτιδήποτε, αλλά στο σενάριο με Τελική Συνέπεια χρησιμοποιούνται για το πεδίο *pos* που απαντά ένας εξυπηρετητής σε αιτήματα GET.

Η GETK_OP είναι παρόμοια με την GET_OP, αλλά αντιστοιχεί στη λειτουργία *get(k)* που θα ορίσουμε για την υλοποίηση: η *L.get(k)* επιστρέφει στον πελάτη τις *k* τελευταίες εγγραφές που βρίσκονται αποθηκευμένες στο *L.S*, δεδομένου ότι $|L.S| \geq k$ και $k \geq 0$. Τα δεδομένα που χρειαζόμαστε για τη λειτουργία είναι ο αριθμός *k*, ο οποίος και θα πρέπει να είναι αποθηκευμένος στο *data*.

Η APP_OP αντιστοιχεί σε αίτηση για λειτουργία *append(r)* στο ΑΚ και ο κωδικός της μπορεί να χρησιμοποιηθεί και από διεργασίες πελάτη. Σαν είσοδο για τη διαδικασία θα χρειαστούμε την εγγραφή *r* που θα προστεθεί. Η είσοδος θα έχει την ακόλουθη μορφή: $\langle request\#, rec_data \rangle$ (ο διαχωριστικός χαρακτήρας για την εγγραφή θα είναι ο χαρακτήρας '|'), όπου *request #* ο κωδικός αιτήματος που δημιούργησε την εγγραφή της *r*, και *rec_data* τα επιπλέον δεδομένα της εγγραφής (τα οποία στην υλοποίησή μας είναι απλές συμβολοσειρές). Τα δεδομένα αυτά εννοείται πως πρέπει να βρίσκονται στο πεδίο *data* στα μηνύματα που ανταλλάσσονται μεταξύ των διεργασιών.

Ο κωδικός SYNC_OP υποδηλώνει κενό μήνυμα για σκοπούς συγχρονισμού του δικτύου, το οποίο χρειάζεται τόσο για τη λειτουργία *ADeliver()* της ΥΑΠ αλλά και για τον αλγόριθμο που υλοποιήσαμε για το Μηχανισμό Ανοχής Σφαλμάτων στο σύστημα, τον οποίο θα περιγράψουμε σε μεταγενέστερο σημείο. Πρόκειται πράγματι για

λειτουργία που συγχρονίζει το σύστημα, καθώς η ΥΑΠ απαιτεί να σταλούν επιπλέον μηνύματα από όλους τους εξυπηρετητές για να παραδοθεί επιτυχώς ένα μήνυμα [6]. Τα δεδομένα που χρειάζονται σαν είσοδος μπορεί να είναι οτιδήποτε, αλλά στην υλοποίηση μας τα θέτουμε πάντα σαν τη συμβολοσειρά “empty0”.

Τέλος, ο `TERM_OP` είναι κωδικός λειτουργίας για τη λειτουργία *Terminate(s)* του Μηχανισμού Ανοχής Σφαλμάτων, την οποία επίσης θα περιγράψουμε αργότερα. Σαν δεδομένα η λειτουργία πρέπει να αποθηκεύσει στο πεδίο *data* τον εξυπηρετητή *s*, που είναι υποψήφιος για τερματισμό λειτουργίας.

4.3 Προγραμματισμός Υποδοχών με το ZeroMQ

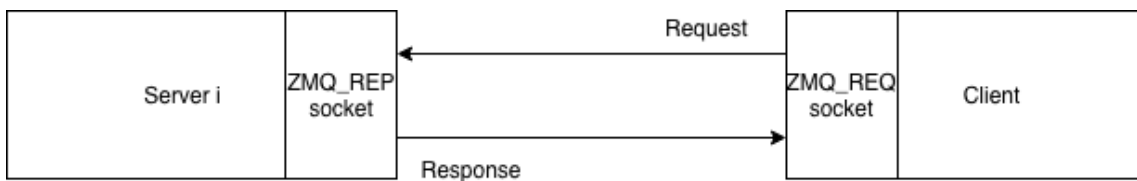
Οι διεργασίες στο δίκτυο χρειάζεται να επικοινωνούν με μηνύματα διαρκώς, δημιουργώντας έτσι και το πρόβλημα του ότι έπρεπε να προγραμματιστούν οι υποδοχές κατάλληλα, μια διαδικασία καθόλου εύκολη. Για τον λόγο αυτό, η υλοποίηση έγινε με τη βοήθεια της βιβλιοθήκης ZeroMQ [7], η οποία προσφέρει λειτουργίες όπως τη δημιουργία έτοιμων υποδοχών, και αλγορίθμους για μετάδοση μηνυμάτων σε πολλούς εξυπηρετητές ταυτόχρονα, κάτι εξαιρετικά χρήσιμο για την υλοποίηση της ΥΑΠ. Ως πρωτόκολλο επικοινωνίας χρησιμοποιείται το TCP, καθώς θέλουμε να είμαστε σίγουροι πως όλα τα μηνύματα που ανταλλάσσουμε στο δίκτυο μας θα φτάσουν στον προορισμό τους.

Η χρήση της βιβλιοθήκης στον κώδικά μας γίνεται με την εντολή `#include <zmq.h>`, δεδομένου του ότι έχει προηγηθεί η επιτυχής εγκατάστασή της.

Για τη δημιουργία υποδοχών πρέπει πρώτα να δημιουργηθεί ένα πλαίσιο (context) στο ZeroMQ, καλώντας την εντολή `void *zmq_new_ctx()`, η οποία αναμένουμε να μας επιστρέψει ένα δείκτη τύπου `void` με τη διεύθυνση του πλαισίου στη μνήμη. Μέσα στο πλαίσιο αυτό δημιουργούμε τις υποδοχές, όπως θα δούμε στις επόμενες υποενότητες.

4.3.1 Μοντέλο Πελάτη - Εξυπηρετητή

Το μοντέλο αυτό είναι το κλασικό μοντέλο επικοινωνίας δύο διεργασιών, το οποίο συναντούμε πιο συχνά στο διαδίκτυο. Απαρτίζεται από ένα πελάτη, ο οποίος στέλλει μία αίτηση (request) στον εξυπηρετητή, και ο εξυπηρετητής με τη σειρά του απαντά με τις πληροφορίες που ζητά η αίτηση (response). Ένα παράδειγμα του πιο πάνω μοντέλου αποτελεί και ο τρόπος επικοινωνίας μεταξύ των πελατών του συστήματος και των διεργασιών που υλοποιούν το ΚΑΚ, όπως βλέπουμε και στο Σχήμα 4.1. Στην υλοποίησή μας, η κάθε διεργασία πελάτη έχει μια υποδοχή για να επικοινωνεί με τους εξυπηρετητή, ενώ η κάθε διεργασία εξυπηρετητή έχει μια υποδοχή που ακούει στη θύρα 32072.



Σχήμα 4.1: Το μοντέλο πελάτη-εξυπηρετητή στο σύστημα

Για να δημιουργήσουμε υποδοχές με το ZeroMQ, πρέπει να καλέσουμε την εντολή `void *zmq_socket(void* context, int type);` η οποία θα μας επιστρέψει ένα δείκτη υποδοχής ZMQ, ή NULL σε περίπτωση σφάλματος. Η παράμετρος `context` είναι το πλαίσιο το οποίο έχουμε δημιουργήσει για τη διεργασία μας, ενώ η παράμετρος `type` είναι το είδος της υποδοχής που δημιουργούμε. Για την περίπτωση μας, ο πελάτης πρέπει να χρησιμοποιήσει την τιμή `ZMQ_REQ`, ενώ ο εξυπηρετητής την τιμή `ZMQ_REP`.

Στη συνέχεια, για να γίνει σύνδεση, θα πρέπει ο εξυπηρετητής να καλέσει τη συνάρτηση

```
int zmq_bind(void* socket, char* endpoint);
```

που θα ενώσει την υποδοχή του εξυπηρετητή με μια διεύθυνση IP. Εδώ ο εξυπηρετητής θα πρέπει να δώσει ως είσοδο τη διεύθυνση tcp://*:32072, ώστε να μπορούν να εξυπηρετηθούν όλες οι πιθανές εισερχόμενες συνδέσεις στη θύρα (port) αυτή.

Από την πλευρά του, ο πελάτης πρέπει να καλέσει τη συνάρτηση

```
int zmq_connect(void* socket, char* endpoint);
```

δίνοντας σαν παραμέτρους τον δείκτη στην υποδοχή του και τη διεύθυνση IP του εξυπηρετητή με τον οποίο θέλει να ενωθεί, μαζί με τον αριθμό θύρας στον οποίο ακούει ο εξυπηρετητής, παραδείγματος χάριν <tcp://1.2.3.4:32072>.

Σημειώνεται για το μοντέλο αυτό πως οι δύο διεργασίες που επικοινωνούν πρέπει να στείλουν μηνύματα με συγκεκριμένη σειρά: πρώτα αιτείται ο πελάτης, έπειτα απαντά ο εξυπηρετητής. Οποιαδήποτε αλλαγή στη σειρά (για παράδειγμα, αν ο πελάτης δοκιμάσει να στείλει δύο αιτήματα πριν πάρει απάντηση από εξυπηρετητή) δε γίνεται αποδεκτή από το ZeroMQ.

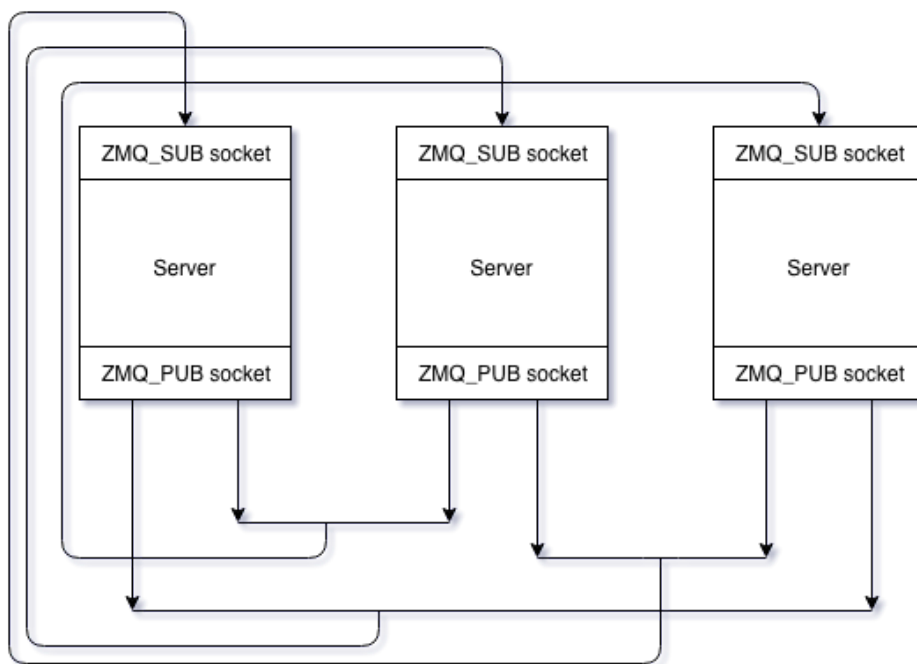
4.3.2 Μοντέλο Εκδότη - Συνδρομητή

Το μοντέλο αυτό είναι λίγο πιο πολύπλοκο, καθώς επιτρέπει σε ένα κόμβο να στείλει ταυτόχρονα σε πολλούς άλλους κόμβους στο δίκτυο ένα μήνυμα. Αυτό γίνεται με sockets του τύπου ZMQ_PUB και ZMQ_SUB, που ενώνονται με τις συναρτήσεις που περιγράφονται πιο πάνω. Ο εκδότης (publisher), αφού δημιουργήσει την υποδοχή του με την επιλογή ZMQ_PUB σαν type, καλεί την `zmq_bind(socket, "tcp://\*:32073")`, ενώ όλοι οι συνδρομητές (subscribers) για να έχουν τη δυνατότητα να λαμβάνουν μηνύματα στην υποδοχή τους, πρέπει να καλέσουν την `zmq_connect(socket, "tcp://1.2.3.4:32073")`, όπου 1.2.3.4 η διεύθυνση IP του εκδότη.

Για να μπορούν να λάβουν, όμως, μηνύματα κανονικά οι συνδρομητές, πρέπει πρώτα να καλέσουν τη συνάρτηση `int zmq_setsockopt(void* socket, int option_name, const void *option_value, size_t option_len)` η οποία προσθέτει ιδιότητες στην υποδοχή που δίνεται σαν παράμετρος, με το `option name` να είναι `ZMQ_SUBSCRIBE`, έτσι ώστε να λαμβάνει μηνύματα από εκδότες. Αν χρειαστεί, στην παράμετρο `option_value` μπορεί να προστεθεί ένα φίλτρο για τα μηνύματα που θα φτάνουν στην υποδοχή αυτή. Με παράδειγμα την υλοποίηση μας, αν το φίλτρο είναι “S” (και συνεπώς, το `option_len` γίνει 1), τότε ο συνδρομητής θα παίρνει μόνο τα μηνύματα που αρχίζουν με την κεφαλίδα “S”.

Το μοντέλο αυτό χρησιμοποιείται στην υλοποίηση μας για την ενδοεπικοινωνία των εξυπηρετητών που υλοποιούν το ΚΑΚ. Ο κάθε εξυπηρετητής L_i έχει δύο υποδοχές για επικοινωνία με τους υπόλοιπους: μια υποδοχή εκδότη, η οποία μεταδίδει μηνύματα στη θύρα 32073, και μια υποδοχή συνδρομητή, που λαμβάνει μηνύματα ενωμένη σαν συνδρομητής σε όλες τις υποδοχές εκδοτών των εξυπηρετητών $L_j \in L, i \neq j$. Για να γίνει αυτό φυσικά, πρέπει να γνωρίζουμε από πριν όλες τις διευθύνσεις των υπόλοιπων εξυπηρετητών. Η επικοινωνία μεταξύ των εξυπηρετητών φαίνεται σχηματικά στο Σχήμα 4.2.

Για το μοντέλο αυτό πρέπει να προσθέσουμε το ότι οι υποδοχές εκδότη μπορούν μόνο να στείλουν μηνύματα και όχι να λάβουν, ενώ οι υποδοχές συνδρομητή μπορούν μόνο να λάβουν μηνύματα, ενώ δε μπορούν να στείλουν.



Σχήμα 4.2: Το μοντέλο εκδότη - συνδρομητή στο σύστημα

4.3.3 Αποστολή μηνυμάτων μεταξύ υποδοχών

Τα μηνύματα που ανταλλάσσονται με τη ZeroMQ αποθηκεύονται σε δομές τύπου `zmq_msg_t`, στις οποίες μπορούμε να αποκτήσουμε πρόσβαση με διάφορες συναρτήσεις που ορίζει η βιβλιοθήκη. Μία από αυτές είναι η `size_t zmq_msg_size(zmq_msg_t* m)`, που επιστρέφει το μέγεθος ενός μηνύματος `m`, του οποίου ο δείκτης δίνεται ως παράμετρος. Επίσης, σημαντική συνάρτηση είναι η `void* zmq_msg_data(zmq_msg_t* m)`, η οποία επιστρέφει ένα δείκτη στην τοποθεσία των δεδομένων του μηνύματος `m` που θα δοθεί σαν παράμετρος με τη μορφή δείκτη. Χρησιμοποιούμε την προηγούμενη συνάρτηση και για να αντιγράψουμε τα δεδομένα που θέλουμε να στείλουμε σε αυτό το μήνυμα, με τη βοήθεια και της συνάρτησης `memset`.

Για να στείλουμε μηνύματα στην υποδοχή, χρησιμοποιούμε τη συνάρτηση `int zmq_msg_send(zmq_msg_t m, void* socket, int flag)`; η οποία θα επιστρέψει το μέγεθος του μηνύματος που έστειλε, ή -1 αν αποτύχει. Για μηνύματα που

θέλουμε να σταλούν σε πολλαπλά μέρη, όπως μηνύματα πολλαπλών τμημάτων στην υλοποίησή μας, πρέπει να δηλωθεί σαν τιμή για το `flag` η `ZMQ_SNDMORE` για όλα τα τμήματα, εκτός από το τελευταίο.

Για να λάβουμε ένα μήνυμα στην υποδοχή, καλούμε την `int zmq_msg_recv(zmq_msg_t* m, void* socket, int flag);` που επίσης επιστρέφει το μέγεθος του μηνύματος που έλαβε, ή -1 αν αποτύχει. Αν ξέρουμε πως μπορεί να λάβουμε πολλά τμήματα, τότε πρέπει να χρησιμοποιηθεί η συνάρτηση `zmq_setsockopt()` με `option=ZMQ_RCVMORE`, για να διαπιστωθεί κατά πόσο ακολουθούν περαιτέρω μηνύματα.

4.4 Μηχανισμός Ανοχής Σφαλμάτων

Προτού περιγράψουμε το πώς έχουν υλοποιηθεί οι αλγόριθμοι, ας θυμηθούμε πως στο δίκτυο που έχουμε, υπάρχει πάντα η πιθανότητα να καταρρεύσει κάποιος από τους εξυπηρετητές. Κάτι τέτοιο θα ήταν καταστροφικό για το σύστημα μας, επειδή η ΥΑΠ αδυνατεί να παραδώσει ένα μήνυμα, αν ο εξυπηρετητής στον οποίον έχει φτάσει το συγκεκριμένο μήνυμα `m` δεν έχει λάβει τουλάχιστον ένα μήνυμα από όλους τους υπόλοιπους εξυπηρετητές στο σύστημα, μετά από τη λήψη του `m` [6]. Έτσι, προκύπτει το εξής ερώτημα: πώς αναγνωρίζουμε το ότι κάποιος από τους υπόλοιπους εξυπηρετητές στο δίκτυο έχει αποσυνδεθεί;

Για το πρόβλημα αυτό υλοποιήθηκε ο Μηχανισμός Ανοχής Σφαλμάτων (Fault Tolerance Module) του συστήματος. Βασιζόμενοι πάνω στο σύστημα με τα λογικά ρολόγια της ΥΑΠ, ορίσαμε ένα δεύτερο μετρητή $FTLC_i[j]$, που αντιπροσωπεύει το “ρολόι” για ανοχή σφαλμάτων που κρατά ο εξυπηρετητής i για άλλον εξυπηρετητή j στο δίκτυο. Κάθε φορά που μια διεργασία εξυπηρετητή i λάβει οποιοδήποτε μήνυμα από τον εξυπηρετητή j , το $FTLC_i[j]$ μηδενίζεται, ενώ για τις υπόλοιπες διεργασίες k , $k \neq j$ το $FTLC_i[k]$ αυξάνεται κατά ένα. Μόλις παρέλθει κάποιο χρονικό σημείο (στην

υλοποίηση επέλεξα να περάσει απλά αρκετός χρόνος ώστε να φτάσει ένας αριθμός μηνύματων στον εξυπηρετητή ίσος με τον αριθμό των εξυπηρετητών που βρίσκονται στο δίκτυο), υπολογίζεται η διαφορά μεταξύ του μεγαλύτερου και μικρότερου *FTLC*. Αν η διαφορά αυτή ξεπερνά ένα σταθερό αριθμό *FT_THRESHOLD* (στην υλοποίηση επέλεξα μετά από πειράματα να ισούται με $N*5+1$, όπου N ο αριθμός των εξυπηρετητών στο δίκτυο), τότε ο εξυπηρετητής j με το μικρότερο *FTLC* προστίθεται σε μια λίστα *exceptions*, στην οποία δεν θα επεξεργάζονται πλέον μηνύματα από τον εξυπηρετητή εκείνο.

Σημειώνεται εδώ, πως ο αλγόριθμος αυτός δεν θα είναι αρκετά αποτελεσματικός στο να διώξει εντελώς από το σύστημα ένα εξυπηρετητή που είναι ακόμα ζωντανός αλλά έχει τύχει να καθυστερήσει να φτάσει το μήνυμά του. Για αυτό το σκοπό, έχω υλοποιήσει παίρνοντας ιδέες από τον αλγόριθμο Terminating Reliable Broadcast [10], τη λειτουργία *Terminate(j)*, η οποία έχει σκοπό να τερματίσει τη διεργασία j που καθυστερεί το σύστημα. Αυτό γίνεται και για να αποφευχθούν λάθη συγχρονισμού με τους πελάτες, οι οποίοι πιθανώς να ζητήσουν αιτήματα από τη διεργασία j , επειδή δε γνωρίζουν πως πλέον δε θεωρείται μέλος του δικτύου με τους ζωντανούς εξυπηρετητές.

Για να γίνει αυτή η διαδικασία, πρέπει να σταλεί από κάθε εξυπηρετητή που αναγνωρίζει την εσφαλμένη διεργασία j μήνυμα, χρησιμοποιώντας την ΥΑΠ. Το μήνυμα αυτό, tm , θα έχει σαν κωδικό λειτουργίας το *TERM_OP* που αναφέραμε νωρίτερα, και σαν δεδομένα τη διεργασία j που είναι υποψήφια για τερματισμό. Λόγω του ότι χρησιμοποιείται η ΥΑΠ, ξέρουμε από τις ιδιότητες της πως αν ο εξυπηρετητής είναι πράγματι ζωντανός και άρα μπορεί ακόμα να δεχτεί και να απαντήσει μηνύματα, τότε το μήνυμα tm πράγματι θα παραληφθεί κανονικά. Μόλις παραληφθεί το μήνυμα tm από τη διεργασία j , ελέγχεται αν ο αποστολέας του μηνύματος βρίσκεται στη λίστα του j με τα *exceptions*. Αν όχι, η j συνεχίζει κανονικά τη λειτουργία της, καθώς θεωρούμε πως μπορεί να πρόκειται για εσφαλμένο μήνυμα. Αν ναι, καλείται διαδικασία για να κλείσουν όλες οι ανοικτές υποδοχές, και ακολούθως η διεργασία j τερματίζεται.

4.5 Δομές Δεδομένων

Για να οργανωθεί καλύτερα ο κώδικας, έχουν δημιουργηθεί δομές για σημαντικά αντικείμενα του συστήματος, οι οποίες και θα αναλυθούν πιο κάτω.

Οι εγγραφές του κατάστιχου είναι αναμφίβολα το πιο σημαντικό αντικείμενο στο ΑΚ, καθώς είναι και ο λόγος ύπαρξης του. Για τις εγγραφές ορίζεται η εξής δομή:

```
typedef struct _trans_struct{  
    int index;  
    int client_id;  
    int req_code;  
    char client_req[10];  
    char data[100];  
}transaction;
```

Το πεδίο `index` αντιστοιχεί στον μοναδικό αριθμό που θα πάρει στο ιστορικό του ΑΚ η εγγραφή αυτή. Τα `client_id` και `req_code` μαζί συνθέτουν τον κωδικό αιτήματος `client_req` που έχει δημιουργήσει την εγγραφή. Στον πίνακα `data` αποθηκεύονται τα δεδομένα, τα οποία περιλαμβάνει η εγγραφή.

Ο τρόπος επικοινωνίας μεταξύ όλων των διεργασιών στο δίκτυο είναι τα μηνύματα που ανταλλάσσονται. Για σκοπούς ευκολότερης επεξεργασίας των μηνυμάτων, υλοποιείται η πιο κάτω δομή:

```
typedef struct _data_struct{  
    char buf[200];  
    int sender;  
    int ts;  
    operation op;  
    transaction* t;  
}data_s;
```

Σύμφωνα με τη μορφή των μηνυμάτων που δηλώσαμε σε πιο πάνω ενότητα, το πεδίο `sender` αντιστοιχεί στον αποστολέα του μηνύματος, το `ts` πρόκειται για τη χρονική σήμανση του μηνύματος, το πεδίο `op` είναι ένας από τους κωδικούς λειτουργίας που αναλύσαμε πιο πριν, ενώ στον πίνακα `buf` αποθηκεύονται τα δεδομένα του μηνύματος για την κάθε λειτουργία. Αν τα δεδομένα εκείνα αναφέρονται σε μια εγγραφή του κατάστιχου, τότε το `t` είναι ένας δείκτης σε δομή `transaction`, που δείχνει προς την εγγραφή εκείνη.

Για υλοποίηση των “αποθηκών” μηνυμάτων `deliverable` και `delivered` που έχουμε στην ΥΑΠ, έχουν δημιουργηθεί δομές κόμβου (`node`) και ουράς (`queue`) για να έχουμε μια διατεταγμένη ακολουθία της σειράς παράδοσης των μηνυμάτων και συναρτήσεις για δημιουργία νέων κόμβων και προσθαφαίρεση κόμβων στην ουρά. Οι δομές έχουν ως εξής:

```
typedef struct _node_s{          typedef struct
    struct _node_s* next;        _queue_struct{
    int index;                    node* head;
    data_s* data;                int size;
}node;                           }queue;
```

Τέλος, δε θα μπορούσε να λείπει και αντικείμενο για το ίδιο το ΑΚ. Τα πεδία του είναι:

```
typedef struct _ledger_struct{
    transaction** seq_history; //HL
    int size;
    int t_str_size;
}ledger;
```

Το πεδίο `seq_history` είναι η ακολουθία S , και έχει τη μορφή πίνακα δεικτών σε αντικείμενα `transaction`, που αντιπροσωπεύουν τις εγγραφές που είναι αποθηκευμένες στο ΑΚ. `Size` είναι το πραγματικό μήκος της ακολουθίας S , και το `t_str_size` είναι ένα επιπλέον πεδίο για σκοπούς διαχείρισης μνήμης, και αντιπροσωπεύει το πόσο μεγάλος είναι αυτή τη στιγμή ο πίνακας `seq_history`.

Για το ΑΚ ορίζονται, επίσης, συναρτήσεις με τις οποίες μπορούμε να έχουμε πρόσβαση στο αντικείμενο. Πρόκειται για τις εξής συναρτήσεις:

```
ledger* ledger_init(); που αρχικοποιεί και επιστρέφει ένα αντικείμενο
κατάστιχου,
transaction** ledger_get(ledger* l); η οποία επιστρέφει την
ακολουθία LS σε μορφή πίνακα δεικτών σε αντικείμενα εγγραφών,
transaction** ledger_getk(ledger* l,int k); που επιστρέφει ένα
μέρος της ακολουθίας LS όπως περιγράψαμε πιο πάνω, σαν πίνακα από δείκτες σε
αντικείμενα εγγραφών,
transaction* ledger_getop(ledger* l,int o); που επιστρέφει σαν
δείκτη σε αντικείμενο την εγγραφή που βρίσκεται στη θέση o της ακολουθίας LS,
int ledger_append(ledger* l,transaction* t); η οποία λειτουργεί
σαν τη Valid() που ορίσαμε για τα Επικυρωμένα Κατάστιχα, ελέγχοντας κατά πόσο η
εγγραφή t που δίνεται στις παραμέτρους υπάρχει ήδη στο ιστορικό και τέλος,
int ledger_addentry(ledger* l,transaction* t,int index); που
προσθέτει μια νέα εγγραφή t στη θέση index της ακολουθίας LS του ΑΚ.
```

4.6 Προγραμματισμός Διεργασιών Αντικειμένου Κατάστιχου

Οι αλγόριθμοι που υλοποιούμε είναι παρόμοιοι μεταξύ τους, έτσι για απλοποίηση της περιγραφής της διαδικασίας που ακολουθείται θα αναλύσουμε την κάθε ενέργεια ξεχωριστά, δίνοντας τις ομοιότητες και τις διαφορές που έχουν οι αλγόριθμοι μεταξύ τους, για να ικανοποιήσουν τις ιδιότητες συνέπειάς τους.

Πριν γίνει οποιαδήποτε άλλη ενέργεια, γίνεται η αρχικοποίηση του συστήματος. Η πρώτη ενέργεια που γίνεται, είναι να πάρει η κάθε διεργασία τον δικό της μοναδικό κωδικό *MY_ID*. Για τον σκοπό αυτό, υπάρχει ένα αρχείο *server_index* στον κάθε εξυπηρετητή που περιέχει τον κωδικό αυτό, ο οποίος δημιουργείται στατικά,

χρησιμοποιώντας ένα αρχείο εντολών, το οποίο και διαβάζεται. Επίσης, διαβάζεται το αρχείο `config_srv` που και πάλι υπάρχει στον εξυπηρετητή από πριν, το οποίο περιέχει όλες τις διευθύνσεις IP των N εξυπηρετητών που συμμετέχουν στο δίκτυο, και οι διευθύνσεις αποθηκεύονται στον πίνακα `dest_ip`. Έπειτα, δημιουργούνται και αρχικοποιούνται οι ουρές `deliverable`, `delivered`, και όπου ορίζονται, οι πίνακες `get_pending`, `pending` και `received`, για να αποθηκεύονται τα μηνύματα που φτάνουν στους εξυπηρετητές, αλλά και οι πίνακες για τους μετρητές LC και $FTLC$, οι οποίοι και αρχικοποιούνται σε 0. Για να ολοκληρωθεί η διαδικασία, δημιουργούνται με τις συναρτήσεις που περιγράψαμε πιο πάνω, πλαίσια και υποδοχές για τον κάθε εξυπηρετητή στο σύστημα. Πρώτα ανοίγονται υποδοχές τύπου Συνδρομητή, οι οποίοι ενώνονται με τις διευθύνσεις IP για όλους τους υπόλοιπους εξυπηρετητές, μετά ανοίγει η υποδοχή Εκδότη, που με τη σειρά της θα δεθεί με τη θύρα 32073 για να στέλλει μηνύματα στους εξυπηρετητές. Τέλος, ανοίγει και δένεται στη θύρα 32072 η υποδοχή για τα αιτήματα των πελατών. Δημιουργείται, επίσης, και ένας πίνακας με αντικείμενα δημοσκόπησης για την υποδοχή (socket poll items) με τις υποδοχές συνδρομητή και πελάτη, για να αναγνωρίζεται πότε κάποια από τις δύο υποδοχές έχει λάβει μήνυμα, χωρίς να μπλοκάρει το σύστημα. Αν απλά αναμέναμε να φτάσει μήνυμα στις υποδοχές διαδοχικά, χρησιμοποιώντας τη συνάρτηση `zmq_msg_recv()`, δε θα μπορούσαμε να επεξεργαστούμε μηνύματα που φτάνουν στη δεύτερη υποδοχή πριν να φτάσει τουλάχιστον ένα μήνυμα στην υποδοχή που περιμένουμε.

Αφού τελειώσει η αρχικοποίηση, δηλώνονται οι μεταβλητές που θα χρησιμοποιηθούν κατά τη διάρκεια της εκτέλεσης, συμπεριλαμβανομένων των μετρητών μηνυμάτων, και μεταβλητών που μετρούν τον χρόνο εκτέλεσης για να προγραμματίζονται αποστολές μηνυμάτων συγχρονισμού. Επίσης, στέλλεται το πρώτο μήνυμα `SYNC_OP` προς όλες τις διεργασίες εξυπηρετητή, ώστε να αρχίσει η διαδικασία του συγχρονισμού.

Έπειτα, η διεργασία μπαίνει σε ατέρμονα βρόγχο, περιμένοντας μηνύματα να φτάσουν. Σε αυτή τη διαδικασία γίνεται έλεγχος αν υπάρχουν μηνύματα στις υποδοχές (poll) και

ενημερώνονται τα ρολόγια `e_timeout` και `timeout` με τον χρόνο που χρειάστηκε η εκτέλεση των εντολών. Αν το `e_timeout` έχει ξεπεράσει ένα αριθμό `EMP_TMO` που ορίζεται σαν σταθερά, τότε γίνεται μετάδοση ενός μηνύματος συγχρονισμού `SYNC_OP`. Πριν επιστρέψουμε στην αρχή των εντολών του βρόγχου, καλείται η διαδικασία `to_deliver()`, την οποία και θα αναλύσουμε μετά, καθώς διαφέρει για τον κάθε αλγόριθμο.

Στην περίπτωση που υπάρχει σε κάποια από τις δύο υποδοχές μήνυμα, η διαδικασία ακολουθείται όπως έχουμε πει σε προηγούμενες ενότητες. Πρώτη ελέγχεται η υποδοχή που λαμβάνει μηνύματα από τους πελάτες, καθώς αναμένουμε πως τα μηνύματα αυτά θα είναι πολύ λιγότερα από τα εσωτερικά μηνύματα του δικτύου και έτσι θα ήταν καλύτερο να τύχουν γρηγορότερης επεξεργασίας. Μόλις επεξεργαστεί τυχόν μήνυμα στην υποδοχή πελάτη, ελέγχεται και επεξεργάζεται και το πρώτο μήνυμα που μπορεί να υπάρχει στην υποδοχή συνδρομητή για τους εξυπηρετητές.

Εάν υπάρχει μήνυμα από πελάτη, τότε αυτό κατακερματίζεται σύμφωνα με την αναμενόμενη μορφή του και δημιουργούνται αντικείμενα `data_s` και, αναλόγως του τύπου του αιτήματος, `transaction`, όπου και θα αποθηκευτούν τα δεδομένα του. Έπειτα, ανάλογα με τη λειτουργία που ζητάται στο αίτημα, γίνονται οι απαραίτητες ενέργειες ανάλογα με το ποιον αλγόριθμο υλοποιούμε. Συγκεκριμένα, για αιτήματα `GET` και `GETK` απαντούμε αμέσως στον πελάτη, καλώντας τη συνάρτηση `send_oplist()` που θα στείλει τη λίστα με εγγραφές που ζητάται στα σενάρια που ορίζεται άμεση απάντηση, ή μεταδίδουμε το μήνυμα αυτό στους υπόλοιπους εξυπηρετητές, χρησιμοποιώντας τη συνάρτηση `to_multicast()`, και έπειτα προσθέτουμε τη δομή που δημιουργήσαμε στους πίνακες `received` και `get_pending`. Όσον αφορά αιτήματα τύπου `APPEND`, το μήνυμα μεταδίδεται αμέσως στους υπόλοιπους εξυπηρετητές μέσω της υποδοχής εκδότη και έπειτα, για το σενάριο με Τελική Συνέπεια, απαντούμε αμέσως στον πελάτη χρησιμοποιώντας τη συνάρτηση `send_single()`, ή για τα άλλα δύο σενάρια, προσθέτουμε το αντικείμενο `data_s` στον πίνακα `received` και την ουρά `pending`.

Αφού επεξεργαστούμε το μήνυμα του πελάτη, ελέγχουμε και την υποδοχή των εξυπηρετητών. Αν υπάρχει μήνυμα σε αυτήν, τότε και πάλι κατακερματίζουμε το μήνυμα, όπως περιγράφεται στην Ενότητα 4.1, αφού όμως προηγηθεί ένας έλεγχος για τη μορφή του μηνύματος, το οποίο και απορρίπτεται από τον εξυπηρετητή αν δεν πληροί τις προδιαγραφές που ορίσαμε. Παρομοίως με τη διαδικασία για την υποδοχή πελάτη, δημιουργούνται αντικείμενα `data_s` και `transaction` για το μήνυμα και γίνονται ενέργειες ανάλογα με τη λειτουργία που ζητείται στο μήνυμα. Πρώτα, ελέγχεται αν πρόκειται για μήνυμα `SYNC_OP`, που θα συγχρονίσει τους μετρητές *LC* και *FTLC* όπως περιγράψαμε στις ανάλογες υποενότητες, ή μήνυμα `TERM_OP` το οποίο θα τερματίσει τη διαδικασία. Αν δεν είναι τίποτα από τα δύο και αν το μήνυμα δεν έχει φτάσει από έναν εκ των εξυπηρετητών που ανήκουν στα *exceptions*, τότε το μήνυμα προστίθεται στον πίνακα `received` για να τύχει περαιτέρω επεξεργασίας. Τέλος, ελέγχονται οι μετρητές *FTLC* για τον Μηχανισμό Ανοχής Σφαλμάτων, ακριβώς όπως περιγράφεται στην Ενότητα 4.4, και αν χρειαστεί, προστίθενται οι εσφαλμένοι εξυπηρετητές στον πίνακα `exceptions`, ο οποίος είναι ουσιαστικά ένας πίνακας με *N* θέσεις, ο οποίος παίρνει την τιμή 1 στη θέση *i* αν ο εξυπηρετητής *i* αναγνωριστεί ως εσφαλμένος, ή την τιμή 0 στην αντίθετη περίπτωση.

Στο τέλος του βρόγχου, όπως αναφέραμε και πριν, λαμβάνει χώρα η διαδικασία `to_deliver()`, που είναι υπεύθυνη για να παραδώσει τα μηνύματα που έχουν φτάσει στον εξυπηρετητή, χρησιμοποιώντας την ΥΑΠ. Αρχικά, η διαδικασία αυτή θα βρει τα μηνύματα *m* από τη `received` για τα οποία ισχύει πως $ts(m) < \min(LC_i[j])$, $j \in S$ και θα τα προσθέσει στην ουρά `deliverable` σε αύξουσα σειρά $ts(m)$, όπως περιγράφεται στην ΥΑΠ. Έπειτα, από την ουρά εκείνη, αφαιρούμε με τη σειρά, μέχρι να αδειάσει, τους κόμβους τους οποίους ακολούθως θα προσθέσουμε στην `delivered`, αφού όμως κάνουμε τις ενέργειες που περιγράφονται στον κάθε αλγόριθμο για τη στιγμή που γίνεται η παράδοση των μηνυμάτων με την ΥΑΠ.

Συγκεκριμένα, στον αλγόριθμο με Τελική Συνέπεια, αν ο κόμβος που αφαιρείται αντιστοιχεί σε μήνυμα APPEND, προσθέτουμε την εγγραφή του μηνύματος στο AK, χρησιμοποιώντας τη συνάρτηση `ledger_addentry()`.

Για τον αλγόριθμο με Ακολουθιακή Συνέπεια, εάν ο κόμβος αντιστοιχεί σε μήνυμα APPEND, προσθέτουμε την εγγραφή στο AK με την `ledger_addentry()`. Επίσης, ελέγχουμε αν υπάρχει στον πίνακα `pending` το αντίστοιχο αίτημα. Αν ναι, απαντούμε στον πελάτη με επιβεβαίωση ή όχι ανάλογα, και αφαιρούμε το αίτημα από την `pending`. Τέλος, ελέγχουμε αν υπάρχουν στην `get_pending` αιτήματα με μικρότερο l_{last} από το μέγεθος της ακολουθίας $L.S$ και αν ναι, στέλλουμε τη νέα ακολουθία στους πελάτες που είχαν αρχικά στείλει το αίτημα και αφαιρούμε το αίτημα.

Για AK με Ατομική Συνέπεια, αν ο αφαιρεθέντας κόμβος αντιστοιχεί σε μήνυμα GET και υπάρχει αίτημα που εκκρεμεί στην `get_pending` για το μήνυμα αυτό, τότε απαντούμε στον πελάτη με την ακολουθία $L.S$ και αφαιρούμε το αίτημα. Αν ο κόμβος πάλι αντιστοιχεί σε αίτημα APPEND, τότε ελέγχουμε για διπλή εγγραφή και ανάλογα προσθέτουμε την εγγραφή στο AK, ενώ αν το αίτημα αυτό εκκρεμεί στην `pending`, απαντάται με επιβεβαίωση προσθήκης της εγγραφής ή όχι ανάλογα, και διαγράφεται από αυτήν.

Τέλος, για σκοπούς παρουσίασης και ελέγχου της διαδικασίας, εάν ο μετρητής `timeout` ξεπεράσει τα 10", τυπώνεται η ακολουθία $L.S$ του AK και ο μετρητής μηδενίζεται.

4.7 Προγραμματισμός Διεργασιών Πελάτη

Αφού είδαμε το πώς προγραμματίζονται οι διεργασίες για τους εξυπηρετητές, θα δούμε και το τι γίνεται στις διεργασίες του πελάτη. Για αυτές έχουμε δύο μόνο σενάρια με μικρές διαφορές, οπότε η διαδικασία που ακολουθείται είναι η ίδια.

Παρόμοια με τους εξυπηρετητές, οι πελάτες κατά την εκκίνησή τους πρέπει να διαβάσουν από το αρχείο `client_index` τον μοναδικό κωδικό τους στο δίκτυο και να αποθηκεύσουν όλες τις διευθύνσεις IP των εξυπηρετητών που βρίσκονται στο δίκτυο. Για σκοπούς απλοποίησης της διαδικασίας, το σύνολο L που ορίζεται σε κάθε πελάτη ισούται με το σύνολο S όλων των εξυπηρετητών στο δίκτυο, έτσι δε χρειάζεται να γίνει οποιαδήποτε νέα επιλογή. Έπειτα, δημιουργείται το πλαίσιο στο ZeroMQ και αρχικοποιείται ο πίνακας `srv_exceptions`, ο οποίος και πάλι θα παίρνει την τιμή 1 στη θέση i αν ο εξυπηρετητής i έχει αναγνωριστεί ως εσφαλμένος και 0 αν δεν ισχύει κάτι τέτοιο. Για να ολοκληρωθεί η αρχικοποίηση, δημιουργούνται οι μεταβλητές που θα χρειαστούν για την εκτέλεση, συμπεριλαμβανομένου του l_{last} και του μετρητή με τα μηνύματα που έχουν σταλεί μέχρι τώρα, και μετρητές για επανασύνδεση, αλλά και για να μετρηθούν οι χρόνοι που χρειάζεται το κάθε αίτημα να απαντηθεί.

Υστερα, η διεργασία πελάτη μπαίνει και αυτή σε βρόγχο, ο οποίος για σκοπούς πειραματικής αξιολόγησης επαναλαμβάνεται `MSG_REP` φορές, όπου `MSG_REP` μια σταθερά που ορίζεται στην αρχή, η οποία ισούται με το πόσα αιτήματα θα σταλούν συνολικά από τον πελάτη αυτό προς το δίκτυο. Με τη βοήθεια της συνάρτησης `rand()`, επιλέγεται τυχαία ένας εξυπηρετητής i από τους εξυπηρετητές στο S που δεν ανήκει στο σύνολο `exceptions`, και γίνεται προσπάθεια σύνδεσης με αυτόν με τη `zmq_connect()`. Αν η σύνδεση αυτή αποτύχει `RECONN_ATT` φορές, ή παρέλθει χρονικό διάστημα ίσο με `TIMEOUT_LIM`, όπου `RECONN_ATT` και `TIMEOUT_LIM` σταθερές που ισούνται με τον μέγιστο αριθμό προσπαθειών επανασύνδεσης και τον μέγιστο χρόνο που μπορεί να περάσει στην προσπάθεια σύνδεσης με έναν εξυπηρετητή αντίστοιχα, τότε ο εξυπηρετητής i προστίθεται στη λίστα με τα `exceptions` και η διαδικασία επαναλαμβάνεται.

Έπειτα, αποφασίζεται τυχαία μια λειτουργία εκ των `GET_OP`, `GETK_OP` και `APPEND_OP` που θα σταλεί σε εκείνο τον εξυπηρετητή και δημιουργείται ένα μήνυμα με τη μορφή που αναφέραμε στην αρχή του κεφαλαίου για την ανάλογη λειτουργία. Ο κωδικός αιτήματος είναι πάντοτε “<c#>_<s#>”, ενώ για τα μηνύματα `APPEND`, η

εγγραφή που δημιουργείται είναι “msg<s#>”, όπου <s#> είναι ο αύξων αριθμός του αιτήματος που στέλλει ο πελάτης και <c#> ο μοναδικός αριθμός ταυτότητας του πελάτη. Το μήνυμα αυτό στέλλεται από την υποδοχή και αυξάνεται ο μετρητής `sent_ctr`, για τον αριθμό μηνυμάτων που έχουν σταλεί. Ακολούθως, ο πελάτης μπλοκάρει με τη `zmq_msg_recv()`, αναμένοντας απάντηση από εξυπηρετητή στο αίτημα του. Μόλις πάρει απάντηση, τότε υπολογίζεται η ώρα που χρειάστηκε το μήνυμα να απαντηθεί από τον εξυπηρετητή στον μετρητή `trans_time` και η απάντηση που έλαβε κατακερματίζεται (στην περίπτωση του αλγόριθμου για Ακολουθιακή Συνέπεια μόνο, ώστε να συγχρονιστεί η μεταβλητή `llast`) και τυπώνεται στην οθόνη, μαζί με τον μετρητή `trans_time`, ο οποίος και μηδενίζεται.

Τέλος, όταν ο μετρητής `sent_ctr` φτάσει τον αριθμό μηνυμάτων `MSG_REP` που ορίσαμε, ο βρόγχος τερματίζει, και υπολογίζεται και τυπώνεται ο μέσος όρος των χρόνων που χρειάστηκαν τα αιτήματα να απαντηθούν από τους εξυπηρετητές, ανάλογα και με τη λειτουργία τους.

Σημειώνεται ότι για παραλλαγές που έγιναν για την πειραματική αξιολόγηση, τα αιτήματα στέλλονται με συγκεκριμένη σειρά και όχι τυχαία, για να μπορούν να συγκριθούν οι χρόνοι των αλγορίθμων.

4.8 Παραδοχές για Υλοποίηση

Κάπου εδώ πρέπει να σημειώσουμε και κάποιες παραδοχές που γίνονται από τα προγράμματα που υλοποιήθηκαν, για λόγους πολυπλοκότητας. Αρχικά, θεωρούμε ως δεδομένο το ότι όλες οι μηχανές που βρίσκονται στο δίκτυο έχουν από πριν τον δικό τους αριθμό, τόσο σαν πελάτες όσο και σαν εξυπηρετητές. Εξίσου βασικό για τη λειτουργία του συστήματος είναι το να έχουν όλες οι μηχανές ένα αρχείο που περιέχει όλες τις διευθύνσεις IP των μηχανών που απαρτίζουν το ΚΑΚ.

Όσο αφορά στην αλγοριθμική σκοπιά, θεωρούμε πως όλες οι μηχανές που συνδέονται στο σύστημα στέλλουν ορθά δεδομένα στο δίκτυο, ενώ επίσης θεωρούμε πως μια διεργασία AK δε θα αποσυνδεθεί ενώ έχει παραλάβει μήνυμα από πελάτη, πριν να απαντήσει στο αίτημα αυτό.

Τέλος, έχουμε σαν δεδομένο πως αν μια διεργασία αποσυνδεθεί από το δίκτυο δε μπορεί να επιστρέψει πίσω στο σύστημα, παρόλο που κάτι τέτοιο είναι δυνατό να γίνει με τη βοήθεια του ZeroMQ και πως όταν μια τέτοια διεργασία λάβει το μήνυμα TERM_OP θα τερματίσει πράγματι τη λειτουργία της.

Κεφάλαιο 5

Πειραματική αξιολόγηση αλγορίθμων

5.1 Προεργασία και Μεθοδολογία

5.2 Προβλήματα και Παρατηρήσεις για το δίκτυο

5.3 Σενάρια Αξιολόγησης

5.4 Αποτελέσματα

5.5 Παρατηρήσεις και Συμπεράσματα

5.1 Προεργασία και Μεθοδολογία

Οι τρεις αλγόριθμοι που υλοποιήθηκαν δοκιμάστηκαν αρχικά σε μια συστάδα οκτώ υπολογιστών του τμήματος Πληροφορικής του Πανεπιστημίου Κύπρου, ώστε να ελεγχθεί η ορθότητά τους, με τους υπολογιστές να παίρνουν τον ρόλο του εξυπηρετητή, και ένα αρχείο εντολών που υλοποίησα και εκτελούσα από τον προσωπικό μου υπολογιστή να έχει τον ρόλο του πελάτη. Αφού πήραμε θετικές ενδείξεις από τους ελέγχους, έγιναν κατάλληλες προσαρμογές στον κώδικα των αλγορίθμων, έτσι ώστε να μπορεί να τρέξει στο κατανεμημένο δίκτυο του PlanetLab Europe (στο εξής PLE), ώστε να πάρουμε πραγματικές καθυστερήσεις, καθώς το περιβάλλον δεν είναι ελεγχόμενο.

Πρώτα όμως, έπρεπε να γίνει η κατάλληλη προεργασία για να προετοιμαστεί το δίκτυο. Δημιουργήθηκε σετ κλειδιών RSA με το οποίο θα αποκτούσα πρόσβαση στις μηχανές του δικτύου. Μόλις αποκτήθηκε πρόσβαση στο δίκτυο, σε όλες τις μηχανές εγκαταστάθηκαν μεταγλωττιστές για τη γλώσσα C, η βιβλιοθήκη ZeroMQ και τα απαραίτητα αρχεία για χρήση της βιβλιοθήκης σε προγράμματα.

Για τον ίδιο σκοπό, έπρεπε να δημιουργηθούν νέα αρχεία εντολών, καθώς πλέον είχαμε να χειριστούμε ένα αρκετά μεγαλύτερο αριθμό εξυπηρετητών, που έκανε τη διαδικασία εκκίνησης των διεργασιών, αλλά και της αντιγραφής και μεταγλώττισης των αρχείων κώδικα αρκετά κοπιώδη αν γινόταν δια χειρός. Έτσι, δημιουργήθηκαν νέα αρχεία εντολών για τη bash για να γράφονται εύκολα τα αρχεία που περιέχουν τους μοναδικούς κωδικούς για τις διεργασίες, ένα πρόγραμμα σε Python για να μπορούν να μεταφέρονται αρχεία σε πολλούς υπολογιστές ταυτόχρονα με τη βοήθεια της βιβλιοθήκης parallel-ssh [12], ενώ χρησιμοποιήθηκε και το πρόγραμμα csshx [15] για παράλληλη σύνδεση σε πολλούς υπολογιστές με τη χρήση του εργαλείου ssh.

Επίσης, όπως αναφέραμε και πριν, έγιναν τροποποιήσεις στον κώδικα των διεργασιών για τα σενάρια τα οποία θα περιγράψουμε σε λίγο.

Στον Πίνακα 5.1 βρίσκονται οι διευθύνσεις IP των μηχανών στο PLE που χρησιμοποιήθηκαν για τους σκοπούς της πειραματικής αξιολόγησης της εργασίας.

5.2 Προβλήματα και Παρατηρήσεις για το Δίκτυο

Ενόσω εκτελούνταν τα πειράματα, παρατηρήθηκε πως, τόσο οι χρόνοι απόκρισης των κόμβων στο δίκτυο, όσο και οι χρόνοι καθυστερήσεων που παίρναμε, συνήθως διέφεραν ανάλογα με την ώρα που εκτελούνταν, με αποτέλεσμα να έχουμε λίγο διαφορετικά αποτελέσματα ανάλογα με τη συμφόρηση του δικτύου.

Ένα από τα προβλήματα που έπρεπε να αντιμετωπιστούν, εκτός από τις δυσκολίες στην προσαρμογή του κώδικα για τις μηχανές στο PLE, είναι η αδυναμία κάποιων από τους κόμβους του συστήματος να εγκαταστήσουν τα ζητούμενα προγράμματα, αφού παρουσιάστηκε σφάλμα αδυναμίας ενημέρωσης από τα ψηφιακά αποθετήρια (repositories). Επιπλέον, μόλις εγκαταστήθηκαν τα προγράμματα, υπήρχε διαφορά των εκδόσεων των βιβλιοθηκών της ZMQ και της ZMQ για προγραμματισμό, κάτι που προκάλεσε αδυναμία να τρέξει ο κώδικας χωρίς προβλήματα. Η λύση στο προηγούμενο

Όνομα διακομιστή	Διεύθυνση IPv4	Χώρα
planetlab-1.ida.liu.se	192.36.94.2	Σουηδία
cse-yellow.cse.chalmers.se	129.16.20.70	Σουηδία
mars.planetlab.haw-hamburg.de	141.22.213.35	Γερμανία
pl1.uni-rostock.de	139.30.241.191	Γερμανία
ple4.planet-lab.eu	132.227.123.14	Γαλλία
planetlab1.cs.uit.no	129.242.19.196	Νορβηγία
pl2.prakinf.tu-ilmenau.de	141.24.249.131	Γερμανία
planetlab3.di.unito.it	130.192.157.138	Ιταλία
icnalplabs1.epfl.ch	192.33.193.16	Ελβετία
icnalplabs2.epfl.ch	192.33.193.18	Ελβετία
merkur.planetlab.haw-hamburg.de	141.22.213.34	Γερμανία
planetlab-1.ing.unimo.it	155.185.54.249	Ιταλία
planetlab1.informatik.uni-kl.de	131.246.19.201	Γερμανία
planetlab1.cs.aueb.gr	195.251.248.180	Ελλάδα
aladdin.planetlab.extranet.uni-passau.de	195.37.16.121	Γερμανία
planetlab1.xeno.cl.cam.ac.uk	128.232.103.201	Μεγάλη Βρετανία
planetlab-2.cs.ucy.ac.cy	194.42.17.164	Κύπρος
planetlab1.upm.ro	193.226.19.30	Ρουμανία
planetlabeu-2.tssg.org	193.1.201.27	Ιρλανδία
cse-white.cse.chalmers.se	129.16.20.71	Σουηδία
planet4.cs.huji.ac.il	132.65.240.103	Ισραήλ
ple1.cesnet.cz	195.113.161.13	Τσεχία
planetlabeu-1.tssg.org	193.1.201.26	Ιρλανδία

Πίνακας 5.1: Μηχανές PLE που χρησιμοποιήθηκαν για την πειραματική αξιολόγηση

δόθηκε με τη μεταγλώττιση και εγκατάσταση της τελευταίας έκδοσης της ZMQ από τον κώδικα της. Επίσης, κάποιες από τις μηχανές αρνούνται να ανοίξουν θύρες (έγιναν δοκιμές με πάρα πολλούς αριθμούς θυρών, οπότεν θεωρώ πως το πρόβλημα δεν οφείλεται στους αριθμούς τους οποίους επέλεξα, ενώ ταυτόχρονα δεν υπάρχει κάπου κάποιο επίσημο εγχειρίδιο ή οδηγία που να δηλώνει ποιες θύρες είναι πάντα διαθέσιμες στις μηχανές του PLE), κάτι που καθιστά το πρόγραμμα αδύνατο να εκτελεστεί στις μηχανές αυτές. Τέλος, κάποιες από τις μηχανές που αρχικά ήταν διαθέσιμες σταμάτησαν να λειτουργούν, σύμφωνα και με την ιστοσελίδα του PLE. Αυτά τα προβλήματα έκαναν τον αριθμό των διαθέσιμων μηχανών που υπάρχουν πλέον στο μέρισμα να μειωθεί από 40, που ήταν αρχικά, σε 23.

Τέλος, ένα ακόμα πρόβλημα που παρατηρήθηκε ήταν με τη χρήση της ZeroMQ, η οποία κάποιες φορές, αν το πρόγραμμα σταματούσε να λειτουργεί λόγω σφάλματος ή αποσύνδεσης, άφηνε ανοικτούς τους περιγραφείς αρχείων (file descriptors) του λειτουργικού συστήματος για τις υποδοχές, με αποτέλεσμα να μη μπορούσαν να χρησιμοποιηθούν σε μεταγενέστερες εκτελέσεις. Το πρόβλημα λύθηκε με το να διαγραφούν χειροκίνητα τα αρχεία από το κάθε σύστημα.

5.3 Σενάρια Αξιολόγησης

Στις επόμενες υποενότητες θα περιγραφούν διάφορα σενάρια για τα οποία ελέγχεται η απόδοση των τριών αλγορίθμων που υλοποιήθηκαν. Για να μπορεί να γίνει σύγκριση μεταξύ των χρόνων που παρατηρούνται για τους αλγόριθμους, γίνεται τροποποίηση στη διεργασία πελάτη, η οποία πλέον δε στέλλει τυχαία δημιουργημένα μηνύματα στους εξυπηρετητές. Αντ' αυτού, θα αποστέλλονται αιτήματα με συγκεκριμένη σειρά: ο κάθε πελάτης θα στέλλει τέσσερα συνεχόμενα, διαφορετικά μεταξύ τους, αιτήματα APPEND για προσθήκη εγγραφής, και έπειτα ένα αίτημα GET για ανάκτηση των εγγραφών. Η διαδικασία αυτή επαναλαμβάνεται μέχρι να σταλούν συνολικά 50 μηνύματα. Ο πελάτης, μόλις λάβει μια απάντηση στο αίτημά του, περιμένει 1 δευτερόλεπτο μέχρι να στείλει το επόμενό του αίτημα, έτσι ώστε να υπάρχει αρκετός χρόνος από την πλευρά

των εξυπηρετητών να παραδώσουν ένα αριθμό μηνυμάτων και να προσθέσουν νέες εγγραφές, πριν οι πελάτες “πλημμυρίσουν” τους εξυπηρετητές με αιτήματα.

Η σειρά με την οποία εκτελούνται οι λειτουργίες για τα πειράματα μπορεί να ξενίζει, καθώς στις αλυσίδες κοινοποιήσεων αναμένεται πως θα καλείται περισσότερο η λειτουργία GET για να συγχρονίζεται το αντικείμενο κατάστιχου μεταξύ των υπολογιστών. Αυτό, ωστόσο, γίνεται κυρίως με σκοπό να παρατηρηθούν οι διαφορές στο χρόνο απόκρισης σε αιτήματα GET, ανάλογα με το πόσες εγγραφές έχουν προστεθεί κάθε φορά στο σύστημα. Γι’ αυτό, έχουμε πρώτα την προσθήκη πολλαπλών εγγραφών με τη λειτουργία APPEND, και μετά την κλήση της GET με τις νέες εγγραφές.

Ο χρόνος υπολογίζεται με τις συναρτήσεις `gettimeofday()` και `timersub()` της βιβλιοθήκης `sys/time.h`. Για τις διεργασίες πελατών, ο χρόνος για κάθε αίτημα αρχίζει να μετρά μόλις πριν σταλεί το μήνυμα αιτήματος από τον πελάτη και σταματά μόλις έχουν φτάσει όλα τα μέρη του μηνύματος που αναμένεται, ενώ υπάρχουν ξεχωριστοί μετρητές για τα αιτήματα GET και APPEND. Για τις διεργασίες εξυπηρετητών, ο χρόνος προσθήκης στο ΑΚ αρχίζει να μετρά μόλις η διεργασία λάβει το μήνυμα από τον πελάτη και τελειώνει μόλις εκτελεστεί η συνάρτηση `ledger_addentry()`. Σε κάθε περίπτωση, μόλις τελειώσει η εκτέλεση του πειράματος που αναμένεται, υπολογίζεται ένας μέσος όρος των χρόνων που μετρήθηκαν στην κάθε μηχανή και τυπώνεται στην οθόνη. Για κάθε εκτέλεση, ο χρόνος που υπολογίζεται είναι ένας μέσος όρος των χρόνων που έδωσε η κάθε μηχανή.

Τα αποτελέσματα κάθε εκτέλεσης καταγράφονται σε αρχείο καταγραφής (log file) στην κάθε μηχανή, για σκοπούς ελέγχου. Εάν υπάρχουν, σε συγκεκριμένη μηχανή, χρόνοι που διαφέρουν κατά πολύ από τους μέσους όρους που υπολογίσαμε, τότε ελέγχονται τα αρχεία καταγραφής για να διαπιστωθεί εάν πρόκειται για γενικότερη καθυστέρηση της μηχανής στο δίκτυο, ή αν μια συγκεκριμένη λειτουργία καθυστέρησε να εκτελεστεί. Αν ισχύει η πρώτη περίπτωση, κρατούμε τα αποτελέσματα ως έχουν, καθώς τέτοιες

καθυστερήσεις στο δίκτυο είναι αναπόσπαστο μέρος του συστήματος και πρέπει να είναι μέρος της πειραματικής αξιολόγησής μας. Στη δεύτερη περίπτωση, όμως, ο μέσος χρόνος που υπολογίζεται αμβλύνεται με την αφαίρεση της συγκεκριμένης λειτουργίας.

Για να υπολογιστούν οι τελικοί χρόνοι στα πειράματα, γίνονται δύο εκτελέσεις για κάθε σενάριο και υπολογίζεται ένας μέσος όρος τους, ο οποίος και δίνεται μαζί με το αντίστοιχο γράφημα για το πείραμα.

Τέλος, να αναφέρουμε πως ο χρόνος που περιμένει ένας εξυπηρετητής σε αδράνεια μέχρι να στείλει ένα μήνυμα συγχρονισμού, είναι σε όλα τα πειράματα ίσος με 2 δευτερόλεπτα. Ο χρόνος αναμονής σε αδράνεια έχει προϋπολογιστεί με σκοπό να γίνεται γρήγορα ο συγχρονισμός του συστήματος όταν οι εξυπηρετητές δεν έχουν νέα αιτήματα από πελάτες. Από την άλλη, ο χρόνος αναμονής αυτός είναι αρκετός έτσι ώστε εξυπηρετητές που έχουν λάβει πολλά αιτήματα από πελάτες να μπορούν να έχουν χρόνο να παραδώσουν τα μηνύματά τους και να προσθέσουν νέες εγγραφές, χωρίς να θεωρηθούν ως ανενεργοί από το Μηχανισμό Ανοχής Σφαλμάτων του συστήματος.

5.3.1 Χρόνος εξυπηρέτησης αιτημάτων / αριθμός πελατών

Το πρώτο σενάριο που θα μελετήσουμε είναι το πώς αλλάζει ο χρόνος εξυπηρέτησης των αιτημάτων στον κάθε αλγόριθμο, σε σχέση με τον αριθμό των πελατών. Για τον σκοπό αυτό, κρατούμε τον αριθμό των εξυπηρετητών σταθερό και ίσο με $L=5$, και θεωρούμε πως κανένας τους δεν θα καταρρεύσει, ενώ σε κάθε εκτέλεση αυξάνουμε τον αριθμό των πελατών C , $C \in [5, 10, 15]$. Ο κάθε πελάτης στέλλει 50 μηνύματα στο σύνολο των εξυπηρετητών, με τον τρόπο που περιγράφεται στην εισαγωγή της ενότητας.

Αναμένουμε πως ο χρόνος εξυπηρέτησης των αιτημάτων στο σενάριο αυτό θα αυξάνεται όσο αυξάνεται ο αριθμός των πελατών, καθώς θα υπάρξουν περισσότερα μηνύματα, που συνεπάγεται συμφόρηση στο δίκτυο μας. Συγκεκριμένα, περιμένουμε

να δούμε μεγάλη αύξηση ιδίως στον χρόνο ανταπόκρισης για τα αιτήματα GET, καθώς με την αύξηση των πελατών και συνεπώς, των μηνυμάτων, γίνεται μεγαλύτερο και το μέγεθος της απάντησης GET που θα παίρνει ο πελάτης.

Όσο αφορά τη χρονική διαφορά μεταξύ των αλγορίθμων, αναμένουμε πως θα υπάρχει μεγάλη διαφορά μεταξύ του αλγορίθμου για Τελική Συνέπεια και των άλλων δύο, καθώς ο πρώτος απαντά αμέσως σε μηνύματα πελατών, ενώ οι υπόλοιποι ακολουθούν μια διαδικασία μέχρι να απαντήσουν. Περιμένουμε, επίσης, πως ο αλγόριθμος για Ατομική Συνέπεια θα κάνει και τον περισσότερο χρόνο να απαντήσει στα αιτήματα, και ειδικότερα στα GET, καθώς η διαδικασία που ακολουθείται περιλαμβάνει την παράδοση όλων των αιτημάτων σε όλους τους εξυπηρετητές, πριν σταλεί απάντηση στον πελάτη.

5.3.2 Χρόνος εξυπηρέτησης αιτημάτων / αριθμός εξυπηρετητών

Στο σενάριο αυτό εξετάζουμε κατά πόσο επηρεάζεται ο χρόνος εξυπηρέτησης αιτημάτων στον κάθε αλγόριθμο, όταν αλλάζει ο αριθμός των εξυπηρετητών στο δίκτυο. Για το πείραμα αυτό, δημιουργούμε 5 διεργασίες πελάτη και εκτελούμε το πείραμα για αριθμό εξυπηρετητών L , $L \in [5, 10, 15]$. Όπως και στο προηγούμενο σενάριο, ο πελάτης στέλνει 50 μηνύματα στους εξυπηρετητές με τη σειρά που περιγράφεται πιο πάνω.

Τα αναμενόμενα αποτελέσματα είναι να μειώνεται ο χρόνος εξυπηρέτησης των αιτημάτων, καθώς αυξάνεται ο αριθμός εξυπηρετητών που μπορούν να εξυπηρετήσουν ένα αίτημα και μπορεί να γίνει καλύτερη κατανομή των αποτελεσμάτων. Για τα αιτήματα GET όμως, περιμένουμε να μην υπάρχει μεγάλη αλλαγή στο χρόνο εξυπηρέτησης κάθε αλγορίθμου, καθώς ο αριθμός των αιτημάτων APPEND από τους πελάτες, άρα και ο αριθμός εγγραφών που θα βρίσκεται στο σύστημα, παραμένει ο ίδιος σε κάθε περίπτωση.

Από την αλγοριθμική σκοπιά, αναμένουμε και πάλι ο αλγόριθμος με Τελική Συνέπεια να παρουσιάσει μικρότερο χρόνο ανταπόκρισης σε αιτήματα από τους υπόλοιπους, και ο αλγόριθμος με Ατομική Συνέπεια να είναι ο πιο αργός στα αιτήματα GET.

5.3.3 Χρόνος προσθήκης εγγραφής / αριθμός πελατών

Εδώ θα αναλύσουμε πώς αλλάζει ο χρόνος προσθήκης μιας νέας εγγραφής στο κατάστιχο, άρα και παράδοσης ενός μηνύματος σε διεργασία AK, σε σχέση με τον αριθμό πελατών που στέλνουν αιτήματα. Το πείραμα είναι πανομοιότυπο με το Σενάριο 5.3.1, που εξετάζει τον χρόνο απόκρισης. Ο αριθμός των πελατών C αυξάνεται κατά 5, $C \in [5, 10, 15]$, ενώ ο αριθμός των διεργασιών AK θα είναι 5 σε κάθε εκτέλεση.

Από τις μετρήσεις που θα πάρουμε αναμένουμε ο χρόνος να κυμαίνεται στα ίδια επίπεδα, καθώς η παράδοση του κάθε μηνύματος στο AK, με βάση την παράδοση μηνυμάτων με την ΥΑΠ, γίνεται μόλις πάρει ο κάθε εξυπηρετητής τουλάχιστον ένα μήνυμα από όλους τους άλλους, άρα έχει περισσότερη σχέση με τον αριθμό των εξυπηρετητών στο σύστημα. Περιμένουμε, όμως, να επηρεάσει ελάχιστα ο μεγαλύτερος αριθμός μηνυμάτων που θα βρίσκεται στο σύστημα, καθώς για τη συνάρτηση Valid(), όπου γίνεται έλεγχος για διπλά μηνύματα στο AK, θα είναι μεγαλύτερος ο χρόνος αναζήτησης.

Οι αλγόριθμοι μεταξύ τους δεν περιμένουμε να έχουν ιδιαίτερη διαφορά στο χρόνο προσθήκης των εγγραφών, καθώς, όπως αναφέραμε και πιο πριν, δεν υπάρχει διαφορά στον τρόπο παράδοσης και προσθήκης των μηνυμάτων, ενώ η μοναδική διαφορά παρουσιάζεται μετά την προσθήκη της νέας εγγραφής.

5.3.4 Χρόνος προσθήκης εγγραφών / αριθμός εξυπηρετητών

Στο σενάριο αυτό που εξετάζουμε, βλέπουμε πόσο αλλάζει ο χρόνος προσθήκης μιας εγγραφής στο κατάστιχο, ανάλογα με τον αριθμό των διεργασιών εξυπηρετητών που απαρτίζουν το καταναμημένο μας δίκτυο. Το πείραμα είναι πανομοιότυπο με το Σενάριο

5.3.2, καθώς ο αριθμός εξυπηρετητών σε κάθε εκτέλεση είναι $L \in [5, 10, 15]$, ενώ υπάρχουν πάντα 5 πελάτες.

Εδώ αναμένουμε ο χρόνος να μειώνεται με την προσθήκη νέων εξυπηρετητών, καθώς περισσότεροι εξυπηρετητές θα έχουν αιτήματα APPEND από πελάτες να εξυπηρετήσουν, συνεπώς θα γίνονται πιο γρήγορα εκπομπές για νέες εγγραφές στο σύστημα και θα γίνεται πιο γρήγορα η παράδοση των εγγραφών και η προσθήκη τους στο ΑΚ. Αυτό, καθώς όπως περιγράψαμε για την υλοποίηση του πελάτη, για το κάθε αίτημα ο πελάτης συνδέεται τυχαία με ένα διαφορετικό εξυπηρετητή, άρα με την αύξηση των διεργασιών εξυπηρετητών θα υπάρχει και μεγαλύτερος αριθμός με διεργασιών που δεν θα εξυπηρετούν αιτήματα πελατών τη στιγμή που θα φτάσει ένα νέο αίτημα λειτουργίας. Ο αριθμός των πελατών παραμένει ο ίδιος, οπότε δεν περιμένουμε να επηρεάσει κάποιος άλλος παράγοντας τον χρόνο της προσθήκης νέων εγγραφών.

Σε σχέση με τους αλγόριθμους, δεν περιμένουμε κάποια ιδιαίτερη διαφορά, επειδή, όπως εξηγήσαμε και πριν, οι διαφορές τους φαίνονται μετά την προσθήκη της νέας εγγραφής.

5.3.5 Διαφορά χρόνου εξυπηρέτησης αιτημάτων APPEND / αριθμός εξυπηρετητών που καταρρέουν

Στο σενάριο που εξετάζεται εδώ, θα δούμε πόσο επηρεάζεται ο χρόνος εξυπηρέτησης των αιτημάτων APPEND σε σχέση με τον αριθμό εξυπηρετητών που καταρρέουν στο σύστημα. Για το πείραμα αυτό, δημιουργούμε ένα δίκτυο 10 εξυπηρετητών και 10 πελατών, οι οποίοι στέλνουν μηνύματα όπως περιγράφεται στην εισαγωγή της ενότητας. Στο σημείο όπου ο κάθε πελάτης έχει στείλει τα μισά μηνύματα (δηλαδή στο μήνυμα με κωδικό αιτήματος #249), προγραμματίζεται να καταρρεύσει ένας αριθμός εξυπηρετητών $F \in [0, 1, 2, 3, 4]$. Το F δε μπορεί να είναι μεγαλύτερο από $N/2$, όπου N ο αριθμός εξυπηρετητών, ώστε να μπορεί το σύστημα να λειτουργεί ορθά. Για τα πειράματα αυτά, υπολογίζεται ο μέσος όρος απόκρισης στα αιτήματα APPEND πριν να

σταλούν 25 μηνύματα, και μετά την αποστολή τους. Από το δεύτερο μέσο όρο αφαιρείται ο πρώτος, και υπολογίζεται η διαφορά στον χρόνο απόκρισης στα αιτήματα για τον πελάτη εκείνο.

Για το πείραμα αυτό αναμένουμε πως ο χρόνος εξυπηρέτησης αιτημάτων APPEND δε θα παρουσιάσει ιδιαίτερη αλλαγή, καθώς το σύστημα ανέχεται καταρρεύσεις των εξυπηρετητών, οπότεν θα έχουμε μια μικρή καθυστέρηση στην αρχή, αλλά μόλις αναγνωριστεί η κατάρρευση των εξυπηρετητών θα συνεχίσει η λειτουργία όπως προβλέπεται. Έτσι, αναμένουμε τον μέσο όρο να μην έχει μεγάλη απόκλιση.

5.3.6 Διαφορά χρόνου προσθήκης εγγραφών / αριθμός εξυπηρετητών που καταρρέουν

Στο τελευταίο σενάριο που εξετάζουμε, θα δούμε αν επηρεάζεται ο χρόνος προσθήκης νέων εγγραφών στο ΚΑΚ σε σχέση με τον αριθμό των εξυπηρετητών του συστήματος που καταρρέουν. Ακριβώς όπως στο Σενάριο 5.3.5, έχουμε ένα δίκτυο 10 εξυπηρετητών, εκ των οποίων οι $F \in [0, 1, 2, 3, 4]$ θα καταρρεύσουν, και 10 πελατών οι οποίοι στέλλουν μηνύματα με τον τρόπο που περιγράφεται στην εισαγωγή της ενότητας. Μόλις έχουν παραδοθεί 25 μηνύματα σε όλους τους εξυπηρετητές, F από τους εξυπηρετητές έχουν προγραμματιστεί να καταρρεύσουν. Για αυτά τα πειράματα, υπολογίζουμε τον μέσο όρο του χρόνου που χρειαζόταν η προσθήκη μιας νέας εγγραφής στο ΑΚ πριν και μετά την κατάρρευση. Ο δεύτερος μέσος χρόνος αφαιρείται από τον πρώτο και υπολογίζεται η διαφορά στον χρόνο προσθήκης εγγραφών στο ΑΚ.

Παρομοίως με τα πειράματα στο Σενάριο 5.3.5, αναμένεται πως δε θα έχουμε σημαντική αλλαγή στους χρόνους προσθήκης εγγραφών, καθώς θα υπάρξει μεν μια μικρή καθυστέρηση την ώρα της κατάρρευσης, η οποία όμως θα αναγνωριστεί και το σύστημα θα συνεχίσει να λειτουργεί όπως πριν. Ίσως σε κάποιες περιπτώσεις να παρατηρηθεί μικρότερος χρόνος μέχρι την παράδοση μηνυμάτων μετά την αναγνώριση της κατάρρευσης, καθώς λόγω του μικρότερου αριθμού εξυπηρετητών θα χρειάζεται λιγότερος χρόνος μέχρι να κληθεί ξανά η διαδικασία για παράδοση μηνυμάτων.

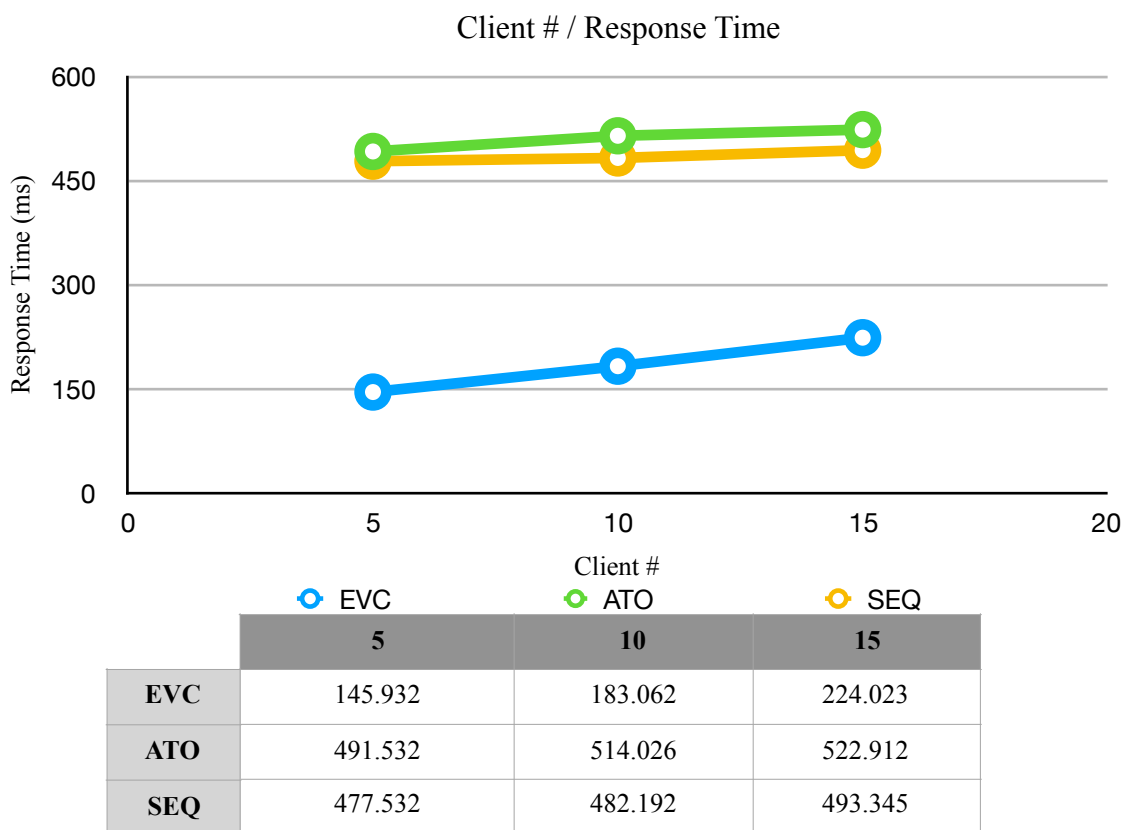
5.4 Αποτελέσματα

Πιο κάτω παρουσιάζονται οι μετρήσεις που παίρνουμε από τα προαναφερθέντα πειράματα. Ο χρόνος που υπολογίζεται σε κάθε περίπτωση είναι ένας μέσος όρος από τους χρόνους που παρουσίασε η κάθε διεργασία για κάθε εκτέλεση. Για σκοπούς συντομογραφίας, αναφερόμαστε στους αλγόριθμους ως εξής: EVC είναι ο αλγόριθμος για Τελική Συνέπεια, και στα γραφήματα παρουσιάζεται με μπλε χρώμα, ΑΤΟ είναι ο αλγόριθμος που ικανοποιεί Ατομική Συνέπεια και θα τον βλέπουμε με πράσινο, ενώ SEQ είναι ο αλγόριθμος για Ακολουθιακή Συνέπεια, που στα γραφήματα είναι με κίτρινο χρώμα. Όλοι οι χρόνοι των μετρήσεων που δίνονται είναι σε χιλιοστά.

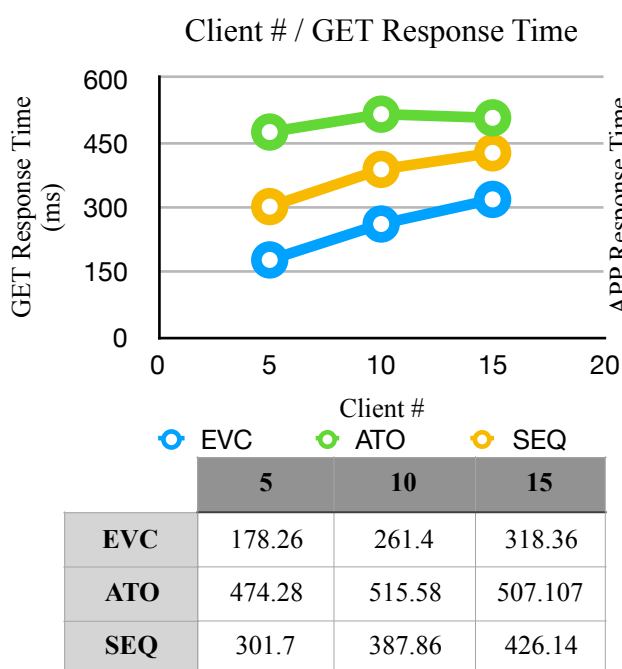
5.4.1 Χρόνος εξυπηρέτησης αιτημάτων / αριθμός πελατών

Τα αποτελέσματα που παίρνουμε από το πρώτο πείραμα είναι τα αναμενόμενα. Στο Γράφημα 5.1 παρατηρούμε πράγματι μια μικρή αύξηση στον χρόνο ανταπόκρισης σε αιτήματα με την αύξηση των πελατών. Υπάρχει, επίσης, μεγάλη διαφορά μεταξύ του αλγόριθμου για Τελική Συνέπεια και των άλλων δύο. Αυτό δικαιολογείται λόγω του ότι ο πρώτος απαντά στον πελάτη σε αιτήματα APPEND πριν παραδώσει τα μηνύματα, ενώ οι υπόλοιποι περιμένουν την παράδοση του μηνύματος πριν να απαντήσουν. Η διαφορά αυτή φαίνεται καθαρά και στο Γράφημα 5.3, όπου παρουσιάζονται οι χρόνοι ανταπόκρισης σε αιτήματα APPEND: παρατηρούμε πως οι αλγόριθμοι για Ατομική και Ακολουθιακή Συνέπεια απαντούν σε παρόμοιους χρόνους, ενώ ο αλγόριθμος για Τελική Συνέπεια έχει αρκετά μικρότερο χρόνο απόκρισης.

Κάτι αξιοσημείωτο είναι ο χρόνος που χρειάζονται για να απαντήσουν αιτήματα GET, που βλέπουμε στο Γράφημα 5.2. Πράγματι ο αλγόριθμος για Ατομική Συνέπεια είναι αρχικά ο πιο αργός, αλλά όσο μεγαλώνει ο αριθμός των πελατών και, συνεπώς, ο αριθμός των εγγραφών, οι χρόνοι των άλλων δύο αλγορίθμων φαίνεται να τείνουν προς το χρόνο του, καθώς η αύξηση των χρόνων τους γίνεται με αρκετά μεγαλύτερο ρυθμό.



Γράφημα 5.1: Χρόνος ανταπόκρισης εξυπηρετητών ως προς τον αριθμό πελατών



Γράφημα 5.2: Χρόνος ανταπόκρισης σε αιτήματα GET ως προς τον αριθμό πελατών

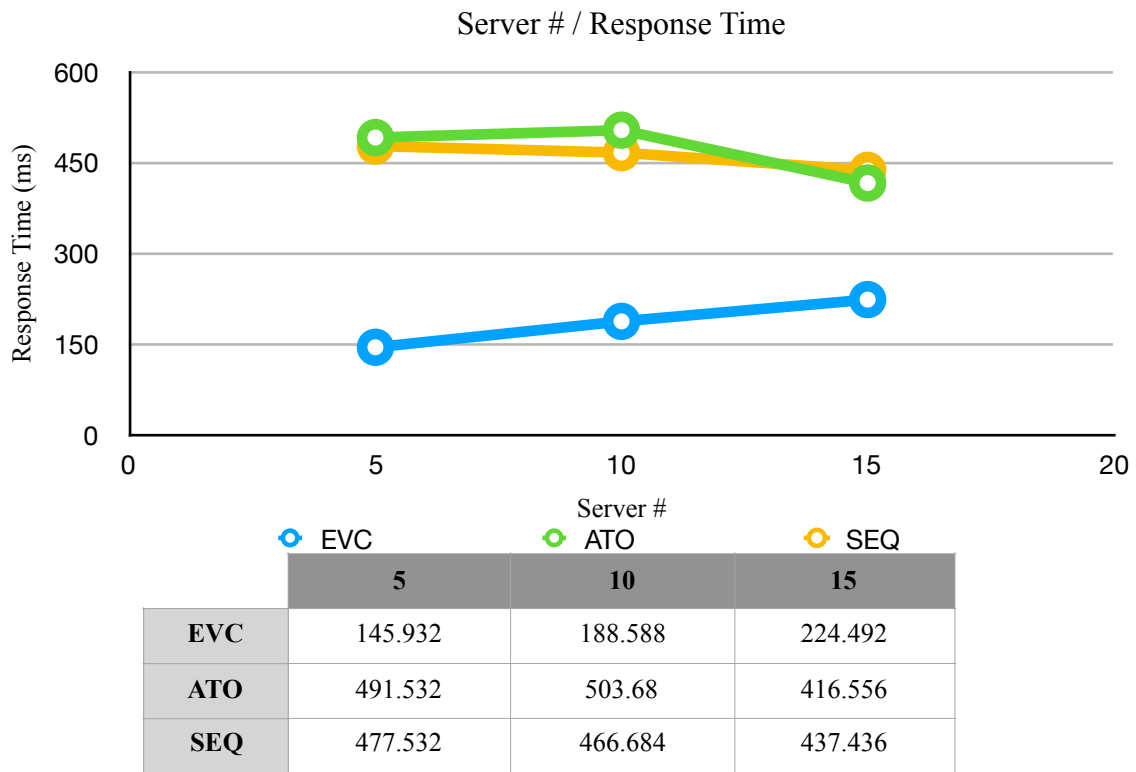


Γράφημα 5.3: Χρόνος ανταπόκρισης σε αιτήματα APPEND ως προς τον αριθμό πελατών

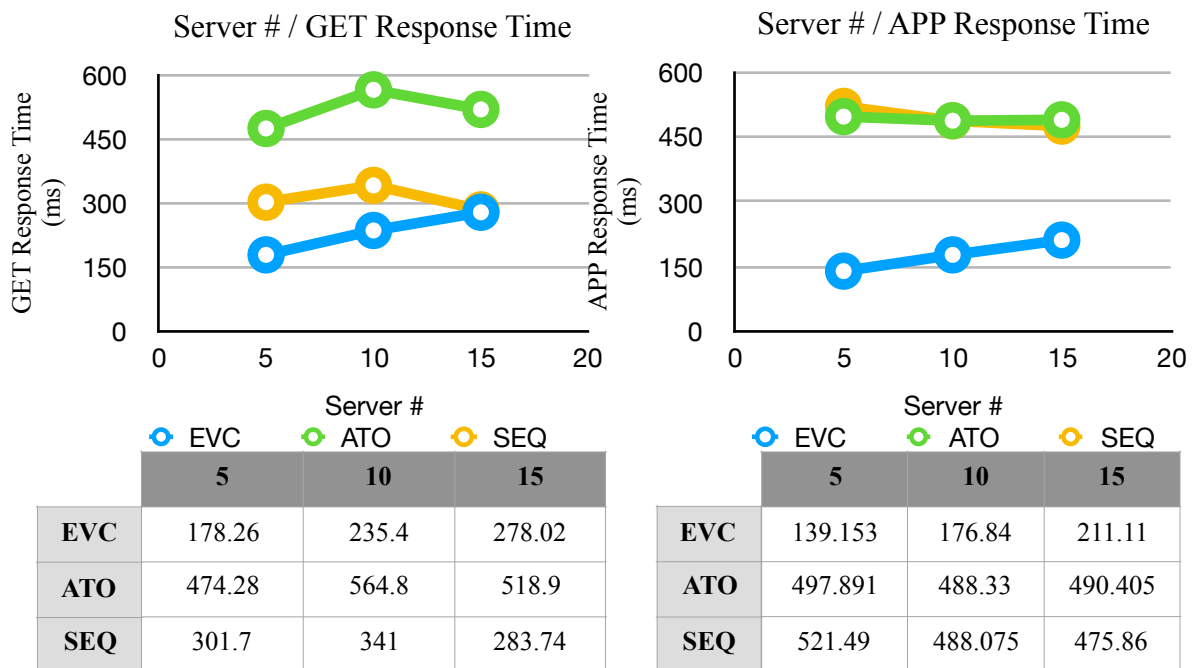
5.4.2 Χρόνος εξυπηρέτησης αιτημάτων / αριθμός εξυπηρετητών

Από τις μετρήσεις εδώ, παίρνουμε αρκετά αποτελέσματα που δε συμφωνούν με τις εκτιμήσεις μας. Αν και παρατηρούμε στο Γράφημα 5.4 πως ο χρόνος για τον αλγόριθμο με Ακολουθιακή Συνέπεια μειώνεται, υπάρχει μία μικρή αύξηση στο χρόνο που χρειάζεται το ΑΚ με Ατομική Συνέπεια για να απαντήσει αιτήματα με 10 εξυπηρετητές πριν ακολουθήσει μεγάλη μείωση, ενώ παραδόξως, ο χρόνος που χρειάζεται ο αλγόριθμος για Τελική Συνέπεια, αυξάνεται όσο αυξάνεται και ο αριθμός των εξυπηρετητών. Συνδυάζοντας και το Γράφημα 5.5, βλέπουμε πως ο χρόνος που χρειάστηκε για την εξυπηρέτηση αιτημάτων GET ήταν πράγματι μεγαλύτερος για τον αλγόριθμο με Ατομική Συνέπεια. Από τα αρχεία καταγραφής του πειράματος, παρατηρούμε πως ήταν πράγματι θέμα τυχαίων καθυστερήσεων στο PlanetLab, καθώς τα αιτήματα GET για τρεις από τους εξυπηρετητές στο δίκτυο είχαν πολύ μεγαλύτερο χρόνο από τα υπόλοιπα και στις δύο εκτελέσεις. Ενδεικτικά, παραθέτουμε στο Σχήμα 5.1 δύο μικρά κομμάτια από ένα αρχείο καταγραφής, όπου βλέπουμε πως ο χρόνος εκτέλεσης για τα δύο τελευταία αιτήματα GET είναι κατά πολύ μεγαλύτερος από τον αναμενόμενο, και προκαλεί την αύξηση αυτή στο μέσο όρο.

Βλέπουμε όμως, επίσης, τόσο στο Γράφημα 5.5 όσο και στο Γράφημα 5.6, πως ο χρόνος που χρειάζεται ο αλγόριθμος με Τελική Συνέπεια αυξάνεται με την αύξηση των διεργασιών εξυπηρετητή για όλα τα είδη αιτημάτων. Αυτό θεωρούμε πως είναι αποτέλεσμα του ότι φτάνουν πολλά περισσότερα μηνύματα από εξυπηρετητές και άρα καλείται πιο συχνά η πλέον χρονοβόρος διαδικασία του συστήματος, αυτή της παράδοσης μηνυμάτων στην ΥΑΠ, η οποία δυσκολεύει την επεξεργασία των μηνυμάτων που φτάνουν στην υποδοχή πελάτη.



Γράφημα 5.4: Χρόνος ανταπόκρισης ως προς τον αριθμό εξυπηρετητών στο δίκτυο



Γράφημα 5.5: Χρόνος ανταπόκρισης σε αιτήματα GET ως προς τον αριθμό εξυπηρετητών

Γράφημα 5.6: Χρόνος ανταπόκρισης σε αιτήματα APPEND ως προς τον αριθμό εξυπηρετητών

```

[...]  

update created==0_44/1/156  

sending 0_44/1/156 to server (1): tcp://141.22.213.35:32072  

Llast found: 175  

Replied to transaction in 806.000 ms  

[...]  

update created==0_49/1/175  

sending 0_49/1/175 to server (3): tcp://132.227.123.14:32072  

Llast found: 190  

Replied to transaction in 741.000 ms  

[...]
```

Σχήμα 5.1: Μέρος του αρχείου καταγραφής του C_0 από την δεύτερη εκτέλεση του Σεναρίου 5.3.2, για Ατομική Συνέπεια και $L=10$

5.4.3 Χρόνος προσθήκης εγγραφής / αριθμός πελατών

Τα αποτελέσματα που έχουμε από το πείραμα αυτό, τα οποία και φαίνονται στο Γράφημα 5.7, μας επιβεβαιώνουν τις αρχικές μας εκτιμήσεις του ότι ο χρόνος προσθήκης εγγραφής δε θα επηρεάζεται κατά πολύ από τον αριθμό των πελατών. Πράγματι, ο χρόνος προσθήκης μιας νέας εγγραφής κυμαίνεται περίπου στα ίδια επίπεδα σε όλους τους αλγορίθμους, ακόμα και με τη μείωση των πελατών.

Όμως, παρατηρούμε μια πολύ μικρή αύξηση στον χρόνο που χρειάζονται να προσθέσουν μια νέα εγγραφή οι αλγόριθμοι στην εκτέλεση με 15 πελάτες, το οποίο οφείλεται στο μεγαλύτερο αριθμό εγγραφών που πρέπει να προστεθούν λόγω των πελατών, καθώς γίνεται ο έλεγχος για διπλές εγγραφές.

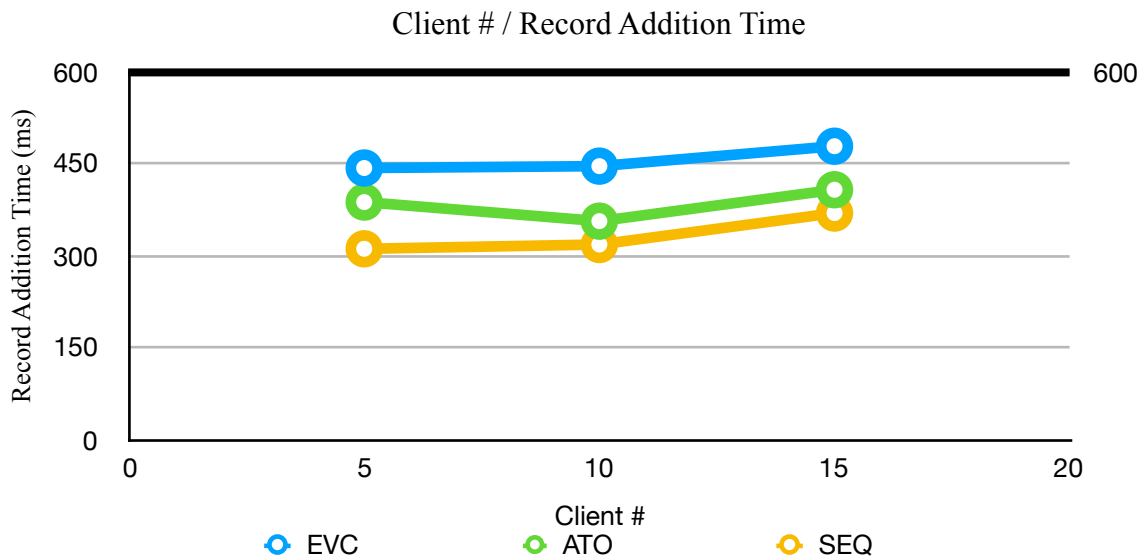
5.4.4 Χρόνος προσθήκης εγγραφής / αριθμός εξυπηρετητών

Για το σενάριο αυτό, βλέπουμε στο Γράφημα 5.8 όπως περιμέναμε, ότι ο χρόνος που χρειάζονται για να προσθέσουν νέα εγγραφή στο ΑΚ οι διεργασίες μειώνεται, όσο αυξάνεται ο αριθμός των εξυπηρετητών, για τους αλγορίθμους με Τελική και Ατομική Συνέπεια. Μάλιστα, στην τελευταία εκτέλεση για $L=15$, βλέπουμε ο χρόνος των τριών αλγορίθμων να τείνει προς μια τιμή κοντά στα 345ms.

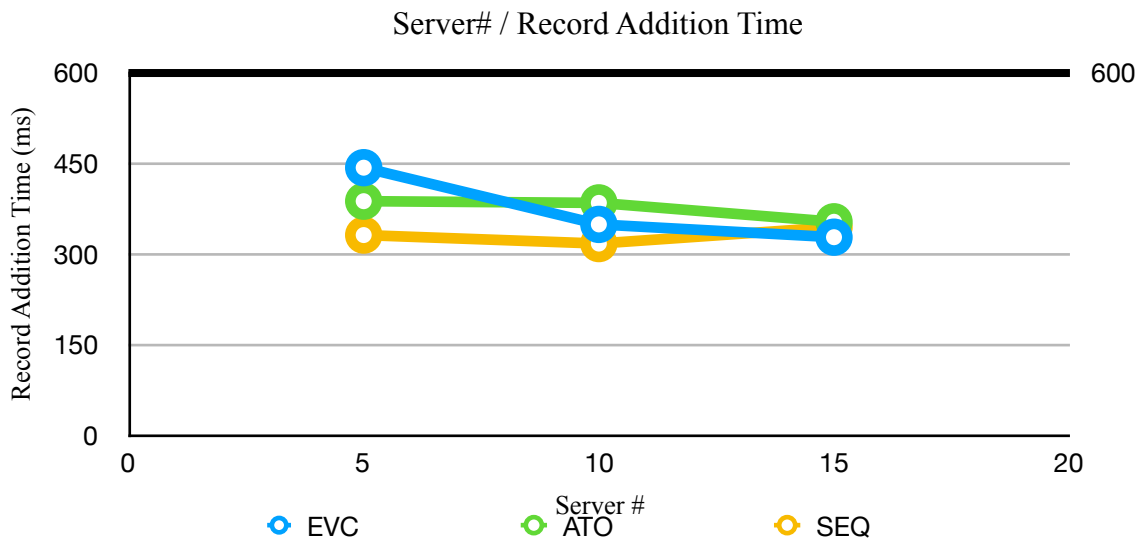
Είναι όμως παράξενο πως ο αλγόριθμος για Ακολουθιακή Συνέπεια παρουσιάζει μια μικρή αύξηση για $L=15$. Αυτό, όπως παρατηρήσαμε στα αρχεία καταγραφής για τις εκτελέσεις, μάλλον οφείλεται σε καθυστερήσεις στο PlanetLab, επειδή όλοι οι εξυπηρετητές έχουν σχετικά μεγαλύτερους χρόνους προσθήκης εγγραφών. Ενδεικτικά, και πάλι, δίνουμε ένα κομμάτι από τα αρχεία καταγραφής στο Σχήμα 5.2, όπου φαίνεται η διαφορά στο χρόνο προσθήκης νέων εγγραφών από τους μέσους όρους που πήραμε από τις εκτελέσεις για $L=5$, $L=10$.

```
[...]  
ADDING ENTRY TO LEDGER #190: 4/1996/3/3_42|msg42  
Added in 430.000 ms  
ADDING ENTRY TO LEDGER #191: 8/2035/3/3_43|msg43  
Added in 593.000 ms  
ADDING ENTRY TO LEDGER #192: 9/2043/3/2_45|msg45  
Added in 766.000 ms  
ADDING ENTRY TO LEDGER #193: 4/2074/3/2_46|msg46  
Added in 423.000 ms  
ADDING ENTRY TO LEDGER #194: 9/2097/3/3_45|msg45  
Added in 473.000 ms  
ADDING ENTRY TO LEDGER #195: 0/2108/3/2_47|msg47  
Added in 490.000 ms  
ADDING ENTRY TO LEDGER #196: 4/2131/3/3_46|msg46  
Added in 350.000 ms  
ADDING ENTRY TO LEDGER #197: 6/2144/3/2_48|msg48  
Added in 510.000 ms  
ADDING ENTRY TO LEDGER #198: 0/2169/3/3_47|msg47  
Added in 545.000 ms  
ADDING ENTRY TO LEDGER #199: 6/2199/3/3_48|msg48  
Added in 504.000 ms  
[...]
```

Σχήμα 5.2: Μέρος του αρχείου καταγραφής του L_5 από την πρώτη εκτέλεση του Σεναρίου 5.3.4, για Ατομική Συνέπεια και $L=10$



Γράφημα 5.7: Χρόνος προσθήκης εγγραφής στο ΑΚ ανάλογα με τον αριθμό πελατών



Γράφημα 5.8: Χρόνος προσθήκης εγγραφής στο ΑΚ ανάλογα με τον αριθμό εξυπηρετητών

5.4.5 Διαφορά χρόνου εξυπηρέτησης αιτημάτων APPEND / αριθμός εξυπηρετητών που καταρρέουν

Οι μετρήσεις που παίρνουμε για το σενάριο αυτό συμφωνούν με τις αρχικές προβλέψεις μας, καθώς οι διαφορές στον χρόνο εξυπηρέτησης των αιτημάτων APPEND είναι ελάχιστες σε όλα τα σενάρια. Στο Γράφημα 5.9 βλέπουμε τη διαφορά σε όλους τους αλγορίθμους να κυμαίνεται μεταξύ των 50ms, κάτι που οφείλεται τόσο στο αρχικό χρονικό περιθώριο που προβλέψαμε πως θα χρειαστεί για να επαναφέρει την ορθή λειτουργία του το σύστημα, όσο και στο ασύγχρονο δίκτυο του PlanetLab.

Οι καθυστερήσεις αυτές, μάλιστα, επιβεβαιώνονται και από τα αρχεία καταγραφής για τις εκτελέσεις. Στο σημείο που ο πελάτης έχει στείλει τα μισά μηνύματα, παρατηρείται να αυξάνεται κατά πολύ για τα αμέσως επόμενα αιτήματα ο χρόνος ανταπόκρισης του εξυπηρετητή και μετά να επανέρχεται κοντά στον μέσο όρο. Στο Σχήμα 5.3 παραθέτουμε ενδεικτικά ένα κομμάτι από τα αρχεία καταγραφής, όπου φαίνεται η αύξηση του χρόνου εξυπηρέτησης αιτημάτων αμέσως μόλις σταλούν τα μισά μηνύματα, και μετά να επανέρχονται κοντά στο μέσο όρο.

5.4.6 Διαφορά χρόνου προσθήκης νέας εγγραφής / αριθμός εξυπηρετητών που καταρρέουν

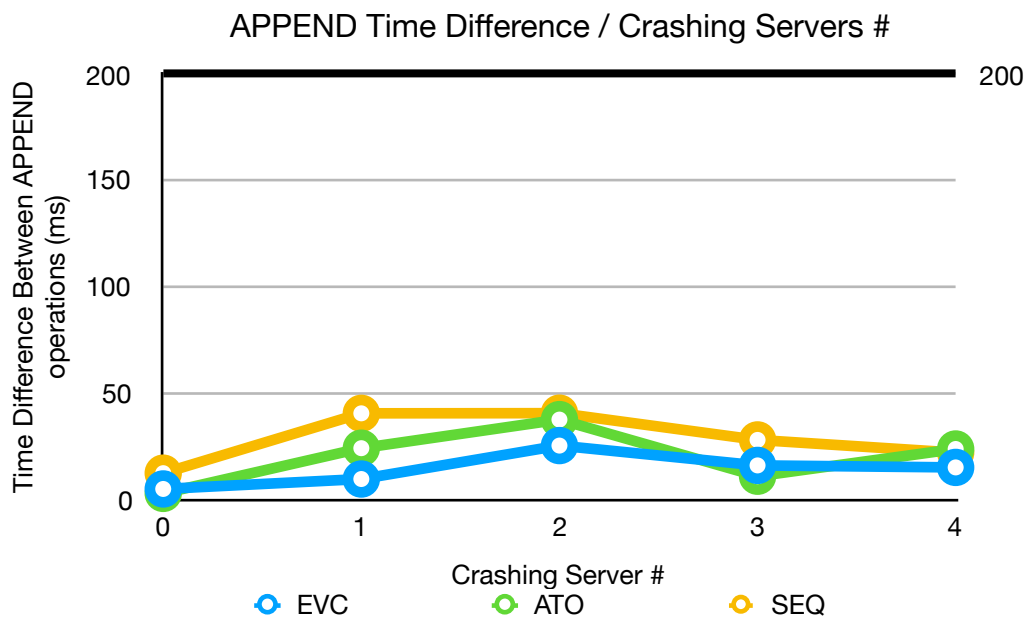
Τα αποτελέσματα που παίρνουμε για το τελευταίο αυτό πείραμα είναι σε μερική συμφωνία με τα αναμενόμενα αποτελέσματα που είχαμε. Συγκεκριμένα, όπως φαίνεται και στο Γράφημα 5.10, όπου παρουσιάζεται η καθυστέρηση στο χρόνο προσθήκης νέας εγγραφής πριν την κατάρρευση σε σχέση με το χρόνο που χρειάζεται μετά την κατάρρευση, παρατηρούνται μειώσεις στη διαφορά χρόνου προσθήκης νέας εγγραφής, ανάλογα με τον αριθμό των εξυπηρετητών που καταρρέει κάθε φορά μέχρι και για $L=10$. Για $L=15$, παρατηρείται μια μικρή αύξηση που οφείλεται στον περισσότερο χρόνο που χρειάζεται ο Μηχανισμός Ανοχής Σφαλμάτων να αναγνωρίσει τους επιπλέον

εξυπηρετητές που έχουν καταρρεύσει. Ακόμα και έτσι, όμως, ο χρόνος που χρειάζονται οι εξυπηρετητές μετά την κατάρρευση εξακολουθεί να είναι μικρότερος από το χρόνο που χρειάζονταν χωρίς εξυπηρετητές να καταρρέουν.

Από το πείραμα αυτό βλέπουμε πως ο μέσος χρόνος που χρειάζεται για να προστεθούν νέες εγγραφές μετά την κατάρρευση είναι λιγότερος από το χρόνο που χρειάζεται με όλους τους εξυπηρετητές στο σύστημα, κάτι που μας επιβεβαιώνουν και τα αρχεία καταγραφής. Ο χρόνος που περνά μέχρι την κλήση της διαδικασίας για παράδοση μηνυμάτων γίνεται κατά πολύ μικρότερος στο δεύτερο μισό της εκτέλεσης, ακόμα και στην περίπτωση που δεν έχουν καταρρεύσει εξυπηρετητές, κάνοντας έτσι την προσθήκη εγγραφών πιο γρήγορη. Έτσι, μπορούμε να συμπεράνουμε πως με την πάροδο του χρόνου και την αποστολή μηνυμάτων στο σύστημα, οι αλγόριθμοι μπορούν να προσθέσουν εγγραφές πιο γρήγορα. Παρατηρούμε, τέλος, πως η καμπύλη για το πόσο πιο γρήγορα προσθέτονται οι νέες εγγραφές και για τους τρεις αλγόριθμους ακολουθεί μια παρόμοια τροχιά, όσο αυξάνεται ο αριθμός των εξυπηρετητών που καταρρέουν.

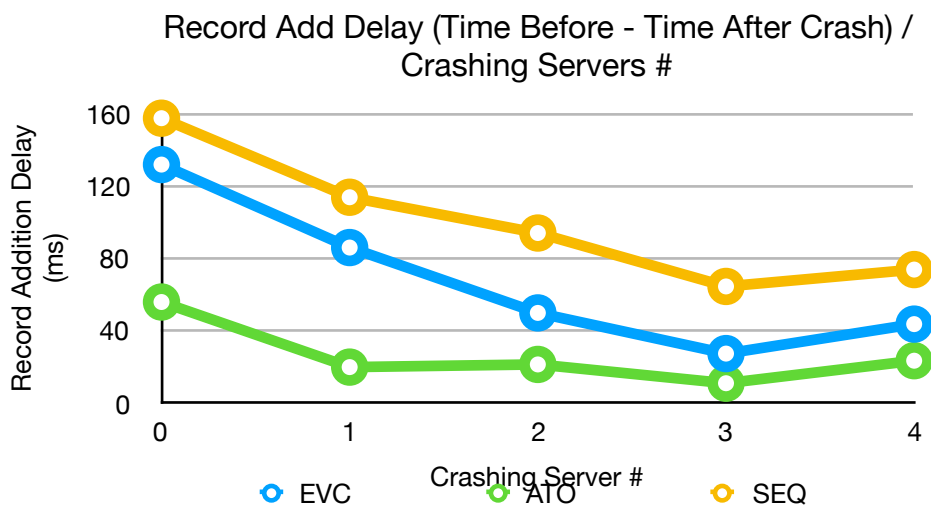
```
[...]
update created==2_23/3/msg23
sending 2_23/3/msg23 to server (8): tcp://192.33.193.18:32072
Llast found: 207
Replied to transaction in 169.000 ms
update created==2_24/1/207
sending 2_24/1/207 to server (6): tcp://130.192.157.138:32072
Replied to transaction in 235.000 ms
update created==2_25/3/msg25
sending 2_25/3/msg25 to server (7): tcp://141.22.213.34:32072
Llast found: 265
Replied to transaction in 530.000 ms
update created==2_26/3/msg26
sending 2_26/3/msg26 to server (1): tcp://141.22.213.35:32072
Llast found: 265
Replied to transaction in 621.000 ms
update created==2_27/3/msg27
sending 2_27/3/msg27 to server (3): tcp://132.227.123.14:32072
Llast found: 265
Replied to transaction in 811.000 ms
update created==2_28/3/msg28
sending 2_28/3/msg28 to server (6): tcp://130.192.157.138:32072
Llast found: 265
Replied to transaction in 170.000 ms
[...]
```

Σχήμα 5.3: Κομμάτι από το αρχείο καταγραφής του C_2 από την πρώτη εκτέλεση του Σεναρίου 5.3.5, για Ατομική Συνέπεια και $F=2$



	0	1	2	3	4
EVC	4.815	9.492	25.238	15.903	14.96
ATO	2.722	24.012	37.477	10.81	23.635
SEQ	12.25	40.43	40.503	27.91	21.93

Γράφημα 5.9: Διαφορά στο χρόνο εξυπηρέτησης αιτημάτων APPEND ως προς τον αριθμό εξυπηρετητών που καταρρέει



	0	1	2	3	4
EVC	131.696	85.923	49.87	27.431	43.69
ATO	55.88	19.975	21.45	11.084	23.504
SEQ	157.283	113.675	93.936	64.473	73.777

Γράφημα 5.10: Καθυστέρηση προσθήκης νέας εγγραφής πριν την κατάρρευση ως προς τον αριθμό εξυπηρετητών που καταρρέει

5.5 Παρατηρήσεις και Συμπεράσματα

Στην ανάλυση των πιο πάνω σεναρίων παρατηρήσαμε, σε γενικές γραμμές, τις συμπεριφορές που περιμέναμε από τους αλγορίθμους, και επιβεβαιώσαμε πως οι χρόνοι των τριών αλγορίθμων διαφέρουν ανάλογα και με το τι βαθμός συνέπειας επιδιώκεται. Ο πιο γρήγορος αλγόριθμος σε ανταπόκριση είναι αυτός με Τελική Συνέπεια, ο οποίος βασίζεται πάνω στο ότι θα παραδοθούν τα μηνύματα σε μεταγενέστερη φάση, ενώ οι υπόλοιποι περιμένουν να παραδοθούν πρώτα τα αιτήματα σε όλους τους εξυπηρετητές και μετά να στείλουν οποιαδήποτε απάντηση. Την αντίθετη σειρά παρατηρούμε στην πλευρά των εξυπηρετητών, όπου αυτός με Τελική Συνέπεια είναι ο πιο αργός στο να παραδώσει μηνύματα και άρα να προσθέσει εγγραφές, ενώ πιο γρήγορος είναι αυτός με Ακολουθιακή Συνέπεια, με πολύ λίγη όμως διαφορά.

Επίσης, παρατηρούμε για τον αλγόριθμο με Ατομική Συνέπεια, πως οι χρόνοι που χρειάζεται για να προσθέσει εγγραφές, άρα και να παραδώσει μηνύματα, συμβαδίζουν γενικότερα με το χρόνο απάντησής του σε αιτήματα. Αυτό επιβεβαιώνει και τον ισχυρισμό για το ότι δίνει την αίσθηση πως λειτουργεί σαν ένα μόνο αντικείμενο, ακόμα και αν οι κλήσεις στις λειτουργίες του γίνονται ταυτόχρονα.

Τέλος, στα πειράματα αυτά μπορούμε να δούμε και στοιχεία για τα ασύγχρονα, κατανεμημένα δίκτυα, καθώς σε πολλές περιπτώσεις παρουσιάζονται απροσδόκητες καθυστερήσεις στην αποστολή διαφόρων μηνυμάτων, οι οποίες οφείλονται στη συμφόρηση του δικτύου στο PlanetLab, αλλά και στις ίδιες τις μηχανές που απαρτίζουν το δίκτυο. Γενικότερα, όμως, οι χρόνοι που πήραμε είναι ικανοποιητικοί για τους αλγορίθμους, και με πιο αποδοτικές υλοποιήσεις θα μπορούσαν να βελτιωθούν ακόμα περισσότερο.

Κεφάλαιο 6

Συμπεράσματα

6.1 Γενικότερα Συμπεράσματα

6.2 Μελλοντική Εργασία

6.1 Γενικότερα Συμπεράσματα

Στην παρούσα διπλωματική εργασία υλοποιήθηκαν αλγόριθμοι για Αλυσίδες Κοινοποιήσεων, βασισμένοι σε Κατανεμημένα Αντικείμενα Κατάστιχου και αξιολογήθηκε η απόδοσή τους για διάφορους παράγοντες που μπορεί να επηρεάσουν, όπως ο αριθμός των πελατών και εξυπηρετητών που συμμετέχουν στο δίκτυο και ο αριθμός των εξυπηρετητών που μπορεί να καταρρεύσουν.

Από την εργασία αυτή μπορούμε να παρατηρήσουμε το πόσο χρήσιμη σε πραγματικές εφαρμογές τέτοιων αντικειμένων θα ήταν μια συνάρτηση όπως η GETK που ορίσαμε. Επίσης, χρήσιμο θα ήταν τέτοια αντικείμενα να συνδυαστούν με αποδοτικές μεθόδους για αναζήτηση εγγραφών, ή και μικρότερα μηνύματα για τα πεδία που ορίζουμε.

Σε κάθε περίπτωση, όμως, μπορεί ξεκάθαρα να διαφανεί ότι τέτοιοι αλγόριθμοι είναι πράγματι υλοποιήσιμοι, ενώ οι χρόνοι που παίρνουμε σαν αποτελέσματα, ακόμα και σε αυτή την κατά πολύ κατώτερη του βέλτιστου υλοποίησης, επιτρέπουν, κατά την άποψή μου, την υιοθεσία του ορισμού αυτού των ΑΚ που περιγράψαμε, ως πρότυπο για το νσχεδιασμό παρόμοιων υπηρεσιών.

6.2 Μελλοντική Εργασία

Η υλοποίηση των αλγορίθμων αυτών πράγματι δεν ήταν η βέλτιστη δυνατή και υπάρχουν πολλοί τομείς στους οποίους μπορεί να βελτιωθεί και να επεκταθεί. Πιο συγκεκριμένα, η υλοποίηση έχει γίνει με σκοπό να χρησιμοποιηθεί από μικρό αριθμό διεργασιών και για μικρό αριθμό μηνυμάτων, καθώς δε γίνεται η καλύτερη δυνατή διαχείριση της μνήμης. Έχουν επίσης γίνει αρκετές παραδοχές για απλοποίηση σχετικά με το ποιες διεργασίες είναι γνωστές στο σύστημα και για τη γνώση τους από πριν για τον μοναδικό κωδικό τους, καθώς και το ότι η κάθε διεργασία πελάτη μπορεί να συνδεθεί με έναν εξυπηρετητή κάθε φορά. Τέλος, υποθέτουμε σε όλα τα σενάρια πως μια διεργασία εξυπηρετητή δε θα σταματήσει να λειτουργεί ενόσω εξυπηρετεί έναν πελάτη. Για τους πιο πάνω λόγους, θα μπορούσε να επεκταθεί η υλοποίηση με συστήματα επανασύνδεσης διεργασιών στο δίκτυο, καθώς και σύνδεσης νέων εξυπηρετητών.

Επίσης, η εργασία αυτή μπορεί να επεκταθεί προσθέτοντας κρυπτογραφικές λειτουργίες στο σύστημα, ώστε να προσωμοιωθεί καλύτερα μια πραγματική αλυσίδα κοινοποιήσεων, αλλά και μακροπρόθεσμα να ενωθούν περισσότερες αλυσίδες και να προστεθεί περισσότερη ασφάλεια στις επικοινωνίες, ώστε να εξελιχθεί σε μια πλήρη υπηρεσία.

Τέλος, θα μπορούσε να γίνει και αρκετή δουλειά στο κομμάτι της πειραματικής αξιολόγησης, καθώς τα πειράματα που έχουμε κάνει περιλαμβάνουν ένα σχετικά μικρό αριθμό από πελάτες και εξυπηρετητές. Επίσης, υπάρχουν άλλες παράμετροι για τις οποίες δεν έχει γίνει ακόμα έλεγχος, όπως για παράδειγμα το πώς θα επηρέαζαν η χρήση μιας λειτουργίας `get(k)`, η αλλαγή του μεγέθους τμήματος στην αλυσίδα κοινοποιήσεων, καθώς και η προσθήκη, αφαίρεση και αναζήτηση εγγραφών με πιο αποδοτικές σε πολυπλοκότητα δομές δεδομένων και αλγόριθμους, και η αξιολόγηση των σεναρίων με άλλες ακολουθίες λειτουργιών, όπως με περισσότερα αιτήματα GET όπως σε πραγματικές αλυσίδες κοινοποιήσεων, ή με τυχαία αιτήματα.

Βιβλιογραφία

- [1] M. van Steen and A. Tanenbaum, "A brief introduction to distributed systems", *Computing*, vol. 98, no. 10, pp. 967-1009, 2016. Available: 10.1007/s00607-016-0508-7.
- [2] J. Gray and A. Reuter, *Transaction processing*. San Francisco, Calif.: Kaufmann, 2011.
- [3] Z. Zheng, S. Xie, H. Dai, X. Chen and H. Wang, "An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends," *2017 IEEE International Congress on Big Data (BigData Congress)*, Honolulu, HI, 2017, pp. 557-564.
- [4] A. Anta, K. Konwar, C. Georgiou and N. Nicolaou, "Formalizing and Implementing Distributed Ledger Objects", *ACM SIGACT News*, vol. 49, no. 2, pp. 58-76, 2018. Available: 10.1145/3232679.3232691.
- [5] M. Raynal, *Concurrent programming*. Berlin: Springer, 2013.
- [6] X. Défago, A. Schiper and P. Urbán, "Total order broadcast and multicast algorithms", *ACM Computing Surveys*, vol. 36, no. 4, pp. 372-421, 2004. Available: 10.1145/1041680.1041682.
- [7] "Distributed Messaging - zeromq", *Zeromq.org*, 2019. [Online]. Available: <http://zeromq.org/>. [Accessed: 01- May- 2019].
- [8] "About PlanetLab", *PlanetLab*, 2019. [Online]. Available: <https://www.planet-lab.org/>. [Accessed: 04- May- 2019].

- [9] "PlanetLab Europe: Monitor", *PlanetLab Europe*, 2019. [Online]. Available: <http://stats.planet-lab.eu/?page=1>. [Accessed: 16- May- 2019].
- [10] R. Guerraoui and L. Rodrigues, Introduction to reliable distributed programming. Berlin: Springer, 2011.
- [11] S. Nakamoto, Bitcoin: A Peer-to-Peer Electronic Cash System, 2008. [Online]. Available: <https://bitcoin.org/bitcoin.pdf> [Accessed: 16- May- 2019].
- [12] F. Akgul, *ZeroMQ*. Birmingham, UK: Packt Publishing, 2013.
- [13] P. Hintjens, *ZeroMQ*. Sebastopol, CA: O'Reilly Media, 2013.
- [14] K. Verbert, M. Sharples and T. Klobučar, *Adaptive and adaptable learning*.
- [15] Πρόγραμμα csshx, διαθέσιμο στο *GitHub* [Online]. Available: <https://github.com/brockgr/csshx>. [Accessed: 20- May- 2019].