Ατομική Διπλωματική Εργασία

ΥΛΟΠΟΙΗΣΗ ΚΑΙ ΠΕΙΡΑΜΑΤΙΚΗ ΑΞΙΟΛΟΓΗΣΗ ΑΛΓΟΡΙΘΜΩΝ ΚΑΤΑΝΕΜΗΜΕΝΩΝ ΑΝΤΙΚΕΙΜΕΝΩΝ ΚΑΤΑΣΤΙΧΟΥ ΜΕ ΤΟ ΕΡΓΑΛΕΙΟ DISTALGO

Λοΐζος Μαρνέρος

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Ιούνιος 2020

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Υλοποίηση και Πειραματική Αξιολόγηση Αλγορίθμων Κατανεμημένων Αντικειμένων Κατάστιχου με το εργαλείο DistAlgo

Λοΐζος Μαρνέρος

Επιβλέπων Καθηγητής
Χρύσης Γεωργίου

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

Ιούνιος 2020

Ευχαριστίες

Θα ήθελα να ευχαριστήσω θερμά τον Επιβλέπων καθηγητή μου, Χρύση Γεωργίου, ο οποίος επέβλεπε σε εβδομαδιαία βάση την δουλεία μου. Κάλυψε όλες τις απορίες που υπήρξαν, όχι μονο σε αλγοριθμικό επίπεδο, αλλά και σε προγραμματιστικό επίπεδο. Γενικότερα, με βοήθησε να καταλάβω τη σημαντικότητα της γνώσης Κατανεμημένων Αλγόριθμων και να διευρύνω τους ορίζοντες μου σε διαφορετικές τεχνολογίες.

Επιπρόσθετα η πτυχιακή αυτή μου πρόσφερε την ευκαιρία να βελτιωθώ και σε άλλους τομείς όπως στην έρευνα πληροφοριών που σχετίζονται με το θέμα μελέτης, στην πειραματική αξιολόγηση ενός προγράμματος και στη σημαντικότητα καταγραφής των ενεργειών και μεθοδολογίας υλοποίησης ενός έργου.

Περίληψη

Στη σύγχρονη επόχη υπάρχει αυξημένη ζήτηση σε αξιόπιστα και ασφαλή συστήματα αποθήκευσης δεδομένων. Μια νεα τεχνολογία που έχει γίνει πολύ δημοφιλής είναι η τεχνολογία των Αλυσίδων Κοινοποίησης (Blockchain), η οποία προέκυψε λόγω της ανάγκης ασφάλειας κρυπτονομισμάτων όπως είναι το Bitcoin και το Ethereum. Οι Αλυσίδες Κοινοποίησης προσφέρουν διαφάνεια και αξιοπιστία στα δεδομένα, έχοντας τα αποθηκευμένα σε διάφορες οντότητες (στην δική μας περίπτωση είναι οι εξυπηρετητές) που βρίσκονται σε διαφορετικές τοποθεσίες του διαδικτύου. Αυτό δημιουργεί την πρόκληση της διατήρησης της συνέπειας των δεδομένων, δηλαδή όλοι οι εξυπηρετητές πρέπει να έχουν τα ίδια δεδομένα.

Η παρούσα εργασία ασχολείται με τα συστήματα που αναφέραμε πιο πάνω. Αρχικά υλοποιήθηκε μια υπηρεσία Κατανεμημένων Αλυσίδων Κοινοποίησης με την χρήση Αντικείμενων Κατάστιχου (Ledger Objects), τα οποία θα ονομάσουμε Κατανεμημένα Αντικειμενα Κατάστιχου (ΚΑΚ). Τα ΚΑΚ προσφέρουν δύο διαφορετικές εγγυήσεις ως προς την συνέπεια των δεδομένων. Η μια προσφέρει μια πιο χαλαρή εκδοχή που ονομάζεται τελική συνέπεια (eventual consistency) και η άλλη ονομάζεται ατομική συνέπεια (atomic consistency). Ο πυρήνας της υλοποίησης βασίζεται σε ένα αλγόριθμο που ονομάζουμε Υπηρεσία Ατομικής Πολυεκπομπής (Atomic Broadcast Service) που ανέχεται σφάλματα κατάρρευσης.

Η υλοποίηση των αλγορίθμων έγιναν με τη χρήση του εργαλείου DistAlgo. Το DistAlgo είναι μια βιβλιοθήκη της γλώσσας Python, όπου προσφέρει προγραμματισμό διεργασιών σε πολύ ψηλό επίπεδο προγραμματισμού. Καθώς αυτή είναι μια νέα βιβλιοθήκη, η οποία συνεχώς εξελίσσεται και η οποία δεν έχει και πολλές υλοποιήσεις αλγορίθμων, ένας από τους σκοπούς της εργασίας ήταν να μελετήσουμε τις δυνατότητες και τις αδυναμίες της. Επιπρόσθετα, δημιουργήθηκε μια διαπροσωπεία για την υπηρεσία, ώστε να θέσουμε πιο εύκολη την χρήση της από τους χρήστες του συστήματος και να γίνουν πιο ευανάγνωστα τα αποτελέσματα. Τέλος, κάναμε μια πειραματική αξιολόγηση της υπηρεσίας, κυρίως για να δείξουμε την ορθή λειτουργία της (proof of concept).

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

1.1 Κίνητρο

Με την ανάπτυξη της τεχνολογίας προέρχεται και η ανάγκη συστημάτων που προσφέρουν ασφάλεια και αξιοπιστία στην αποθήκευση πληροφοριών. Οι αλυσίδες κοινοποίησης (blockchains) αποτελούν τέτοια συστήματα. Η κύρια ιδέα τέτοιων συστημάτων είναι ο διαμοιρασμός των αποθηκευμένων πληροφοριών σε διάφορους εξυπηρετητές από διάφορες περιοχές, ώστε να παρέχεται πλήρης διαφάνεια από τη μια, και αξιοπιστία των δεδομένων από την άλλη, δηλαδή τα δεδομένα δεν μπορούν να παραποιηθούν. Μέχρι στιγμής χρησιμοποιήθηκε κυρίως σε κρυπτονομίσματα και ένα από τα πιο γνωστά παραδείγματα είναι το Bitcoin [8].

Παρ'όλα αυτά γενικότερα δεν υπάρχουν αρκετές αλγοριθμικές λύσεις που να δίνουν αποδεδειγμένες εγγυήσεις για την αξιοπιστία και την συνέπεια των δεδομένων στις πλατφόρμες αυτές. Μια από τις λίγες εργασίες που βρέθηκαν είναι το άρθρο *Formalizing and Implementing Distributed Ledger Objects* [10], όπου αναλύονται με λεπτομέρεια οι αλγόριθμοι συνέπειας που υπάρχουν και αποδεικνύεται η ορθότητά τους. Συνεχίζοντας με την εργασία αυτή θα μελετήσουμε 2 αλγόριθμους συνέπειας και θα δημιουργήσουμε μια

διαπροσωπεία, φιλική προς τους χρήστες, για να χειριζόμαστε και να επιβλέπουμε τους αλγόριθμους.

1.2 Στόχος Διπλωματικής Εργασίας

Η παρούσα Διπλωματική Εργασία έχει δύο βασικούς στόχους. Ο πρώτος στόχος είναι η μελέτη και η υλοποίηση των αλγόριθμων της υπηρεσίας Ατομικής Πολυεκπομπής, των Κατανεμημένων Αντικειμένων Κατάστιχου (Distributed Ledger Objects) και των αλγόριθμών τελικής (eventual) και ατομικής (atomic) συνέπειας στο εργαλείο DistAlgo. Για τους αλγόριθμους αυτούς θα εφαρμόσουμε και μηχανισμό ανοχής σφαλμάτων μεταξύ των εξυπηρετητών, για τις περιπτώσεις κατάρρευσης. Επιπρόσθετα θέλουμε να δημιουργήσουμε μια διαπροσωπεία, η οποία θα βοηθά το χρήστη να ορίζει τα αιτήματα που θα στέλνονται στους εξυπηρετητές και στο τέλος να εμφανίζονται τα Αντικείμενα Κατάστιχου σε γραφική μορφή. Με αυτή την υλοποίηση θα κάνουμε και πειραματική αξιολόγηση του αλγόριθμου με ατομική συνέπεια, χρησιμοποιώντας διάφορα σενάρια, για να κατανοήσουμε καλύτερα τι επηρεάζει την ταχύτητα του χρόνου ανταπόκρισης και κατά πόσο οι αλγόριθμοι δουλεύουν σωστά.

Ο δεύτερος στόχος της εργασίας είναι να γίνει μελέτη των δυνατοτήτων αλλά και των αδυναμιών του εργαλείου DistAlgo. Το εργαλείο είναι καινούργιο οπότε δεν υπάρχουν αρκετές υλοποιήσεις με την χρήση του, οπότε θα δημιουργήσουμε την δική μας υλοποίηση και ταυτόχρονα να καταγράψουμε τις εμπειρίες μας.

1.3 Μεθοδολογία

Αρχικά μελετήσαμε Άρθρα σχετικά με τα Κατανεμημένα Αντικείμενα Κατάστιχου [10], την Υπηρεσία Ατομικής Πολυεκπομπής [1,2] και τους αλγόριθμους ατομικής και τελικής συνέπειας [2,3,7] ως βάση της υλοποίησης. Στη συνέχεια μελέτησα διάφορους αλγόριθμους αξιόπιστης Πολυεκπομπής [2], μελετώντας τις αδυναμίες τους και κατανοώντας πως ο αλγόριθμος ατομικής συνέπειας θα τα επιλύσει.

Αφότου κατανοήθηκαν οι αλγόριθμοι αξιόπιστης Πολυεκπομπής, μελετήθηκαν προσεκτικά τα Αντικείμενα Κατάστιχου, η ΥΑΠ (Υπηρεσία Ατομικής Πολυεκπομπής) και οι αλγόριθμοι συνέπειας. Γνωρίζοντας την συμπεριφορά κάθε αλγόριθμου συνέπειας, άρχισε η μελέτη της γλώσσας προγραμματισμού Python, στην οποία δεν είχα και τόσο μεγάλο υπόβαθρό, και της βιβλιοθήκης DistAlgo [11]. Ακόμη και με την ύπαρξη πολλών και

εύκολων μεθόδους εγκατάστασης της βιβλιοθήκης και με αναλυτική περιγραφή, υπήρξαν δυσκολίες στην υλοποίηση των βημάτων, λόγω της λάθος εγκατάστασης της γλώσσας Python στον υπολογιστή που χρησιμοποιήθηκε.

Μετά την εγκατάσταση έτρεξα κάποια από τα παραδείγματα που προσφέρονται, για να κατανοήσω τον τρόπο λειτουργίας του. Μαζί με αυτά μελέτησα τα άρθρα[4-7] που υπήρχαν στο tutorial της ιστοσελίδας τους [11], όπου επεξηγούν της λειτουργίες της κάθε εντολής της DistAlgo και τα δομικά στοιχεία ενός Κατανεμημένου Συστήματος. Στη συνέχεια μελέτησα την περιγραφή της γλώσσας επίσης από την ιστοσελίδα του DistAlgo, όπου δίνει λεπτομερή περιγραφή της γλώσσας και των εντολών της.

Έπειτα, προσπάθησα να δημιουργήσω τα δικά μου scripts στο DistAlgo για καλύτερη κατανόηση των λειτουργίων που προσφέρονται. Συνεχίζοντας άρχισε η υλοποίηση της Υπηρεσίας Ατομικής Πολυεκπομπής σε τοπικό επίπεδο. Σε αυτό το στάδιο, υπήρξε μεγάλη δυσκολία, λόγω της πολυπλοκότητας των αλγόριθμων Κατανεμημένων Συστημάτων, το οποίο κάθε process έτρεχε σε διαφορετική ταχύτητα από τα υπόλοιπα. Οπότε σε κάθε στάδιο των επεξεργαστών πρόβαλα τα δεδομένα που λαμβάναν και τα σύγκρινα με τα αναμενόμενα αποτελέσματα.

Μετά από την ΥΑΠ δημιουργήθηκε στην Python ο αλγόριθμος τελικής συνέπειας και στη συνέχεια της ατομικής συνέπειας, τα οποία χρησιμοποιούν το ΥΑΠ. Στη συνέχεια μετά από μελέτη για μηχανισμούς σφαλμάτων[2] ,προστέθηκε ο μηχανισμός στην διεργασία των εξυπηρετητών. Η χρήση του μηχανισμού σφαλμάτων είναι να ελέγχει για οποιοδήποτε αναπάντεχο τερματισμό ή αργοπορία ενός εξυπηρετητή ώστε να προσαρμόσει την ρουτίνα.

Ακολουθώντας μελετήθηκε η βιβλιοθήκη της Python PyQt5[10,11] ,η οποία είναι υπεύθυνη για την δημιουργία του γραφικού περιβάλλοντος για την εύκολη λειτουργία του προγράμματος. Αρχικά ο στόχος μας ήταν να δημιουργήσουμε ένα εργαλείο για τους πελάτες και ένα για τους εξυπηρετητές αλλά λόγω του περιορισμου της γλώσσας όσο αφορά την επικοινωνία εξ’αποστάσεως αλλάξαμε σε μια προσομοίωση μεταξύ πελατών και εξυπηρετητών.

Τέλος κάναμε τις απαραίτητες αλλαγές για την πειραματική αξιολόγηση με βάση τον αριθμό των πελατών και τον αριθμό των εξυπηρετητών για την ταχύτητα του συστήματος.

1.4 Δομή Εργασίας

Στο Κεφάλαιο 2 παρουσιάζονται οι απαραίτητες γνώσεις που χρειάζονται για την κατανόηση της μελέτης, καθώς και την περιγραφή των εργαλείων που θα χρησιμοποιηθούν για την υλοποίηση των αλγορίθμων και του γραφικού περιβάλλοντος.

Συνεχίζοντας, στο Κεφάλαιο 3 γίνεται λεπτομερής περιγραφή των δύο αλγόριθμων συνεπειας των δεδομένων για Κατανεμημένα Αντικείμενα Κατάστιχου, δηλαδή με τελική και ατομική συνέπεια και του πυρήνα των αλγορίθμων που είναι η Υπηρεσία Ατομικής Πολυεκπομπής.

Ακολούθως στο Κεφάλαιο 4 περιγράφονται οι λειτουργίες της διαπροσωπείας και οι περιορισμοί που υπήρξαν κατά την δημιουργία της.

Επιπρόσθετά, στο Κεφάλαιο 5 περιγράφεται η διαδικασία της πειραματικής αξιολόγησης, αναφέροντας τους παράγοντες που μελετούνται και τις κύριες παραμέτρους, παρουσιάζοντας σε μορφή γραφικής παράστασης τα αποτελέσματα, ακολουθόμενη από ανάλυσή τους.

Τέλος, στο Κεφάλαιο 6 θέτουμε τα συμπεράσματα που αφορούν τα εργαλεία που χρησιμοποιήσαμε, τα μοντέλα και τους αλγόριθμους συνέπειας, καθώς επίσης γίνεται αναφορά σε μελλοντική εργασία που μπορεί να γίνει με βάση την παρούσα εργασία.

1.5 Γλωσσάριο

Εδώ αναφέρονται οι έννοιες που θα χρησιμοποιούμε καθ’ολη τη διάρκεια του κειμένου με τις αντίστοιχες αγγλικες ορολογίες τους

- Αλυσίδα Κοινοποίησης – Blockchain
- Αντικείμενο Κατάστοιχου – Ledger Object
- Κατανεμημένα Συστήματα – Distributed Systems
- Υπηρεσία Ατομικής Πολυεκπομπής – Atomic Broadcast Service
- Τελική Συνέπεια – Eventual Consistency
- Ατομική Συνέπεια – Atomic Consistency
- Μηχανισμός Ανοχής Σφαλμάτων – Fault Tolerance Module
- Διαπροσωπία – Interface

- Διεργασία – Process
- Πελάτης – Client
- Εξυπηρετητής – Server
- Ετικέτα – Label

Κεφάλαιο 2

Υπόβαθρο

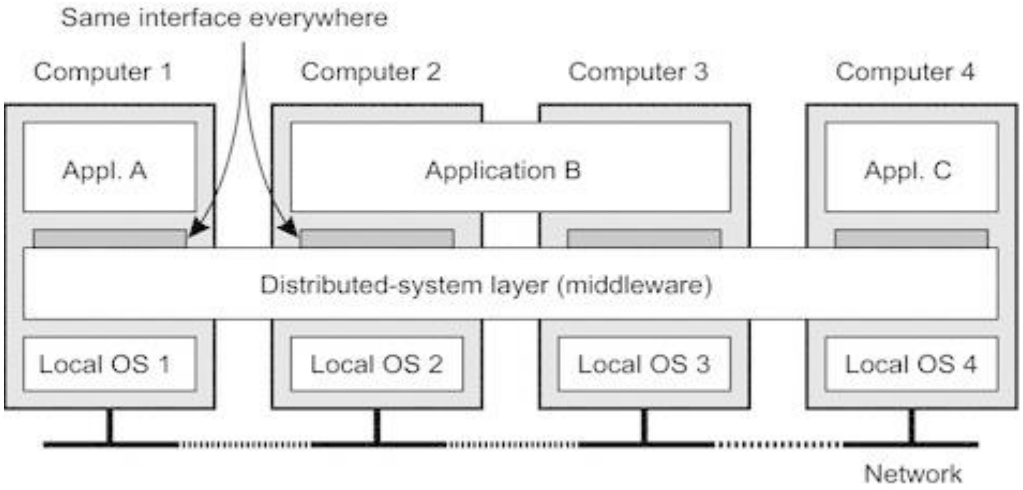
2.1 Κατανεμημένα Συστήματα

Θεωρούμε ενα Κατανεμημένο Σύστημα ένα σύνολο από αυτόνομες (μεταξύ τους) μηχανές, οι οποίες λειτουργούν και δίνουν την αίσθηση στους χρήστες ότι υπάρχουν ως ένα ενιαίο σύστημα. Μεταξύ τους δεν υπάρχει κοινή μνήμη, υπάρχει μόνο τοπική μνήμη, και ο μόνος τρόπος επικοινωνίας και ανταλλαγής δεδομένων είναι με ανταλλαγή μηνυμάτων είτε μέσω του ίδιου δικτύου, είτε μέσω μιας ενδιάμεσης συσκευής γνωστή ως middleware (Σχήμα 2.1).

Στη δική μας περίπτωση θα χρησιμοποιήσουμε ενα τοπικό δίκτυο από αυτόνομες διεργασίες, οι οποίες θα ανταλλάσουν μηνύματα ασυχρόνιστα, χωρίς να χρειάζεται να περιμένουν το μήνυμα να παραδοθεί στις υπόλοιπες διεργασίες, αφού αναμένονται να υπάρχουν καθυστερήσεις στην ανταλλαγή μηνυμάτων.

2.2 Αντικείμενα Κατάστιχου

Ένα Αντικείμενο Κατάστιχου (ΑΚ) [10] καθορίζεται ως μια ακολουθία από



Σχήμα 2.1: Κατανεμημένο σύστημα με ενδιάμεση συσκευή για επικοινωνία [3]

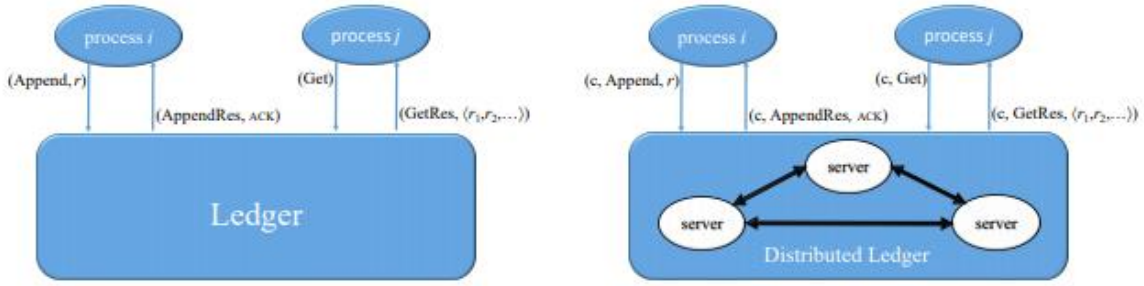
καταγραφές (από κείμενο σε αρχείο κλπ..), στις οποίες μπορούμε να εφαρμόσουμε δύο είδη βασικών λειτουργιών, το get και το append. Η λειτουργία του get είναι να επιστρέφει την ακολουθία των αποθηκευμένων καταγραφών S και του append είναι να προσθέτει στο τέλος της ακολουθίας καινούργια εγγραφή.

Οι εγγραφές έχουν την μορφή (c,r,p) όπου c είναι ένας μοναδικός συντελεστής που αναθέτει στην εγγραφή μια διεργασία, το r είναι οι πληροφορίες που μεταφέρονται για προσθήκη στο Αντικείμενο Κατάστιχου και το p είναι η ταυτότητα του αποστολέα. Αυτά τα στοιχεία θα χρησιμοποιηθούν για να ξεχωρίσουμε το κάθε μήνυμα, ώστε οι επεξεργαστές να μπορούν να γράψουν τα ίδια δεδομένα στο Αντικείμενο.

Με τα Αντικείμενα Κατάστιχου προσθέτουμε μια νέα ορολογία που τα χρησιμοποιεί ως βάση της λειτουργίας τους και ονομάζονται Κατανεμημένα Αντικείμενα Κατάστιχου.

Τα ΚΑΚ είναι Αντικείμενα Κατάστιχου τα οποία είναι αποθηκευμένα σε διάφορες μηχανές σε απόλυτη σειρά, είτε στην ίδια περιοχή, είτε διασκορπισμένες σε διάφορες περιοχές. Με αυτά τα αντικείμενα μπορούμε να δημιουργήσουμε ένα ασφαλές και αξιόπιστο δίκτυο μηχανών, στο οποίο πελάτες μπορούν να αποθηκεύουν πληροφορίες ή να παίρνουν πληροφορίες, δίνοντας την ψευδαίσθηση ότι υπάρχει μόνο μια μηχανή που διαχειρίζεται τα δεδομένα (Σχήμα 2.2)

Οι αλγόριθμοι που θα μελετήσουμε έχουν ως στόχο την διατήρηση τέτοιων αντικειμένων μέσω διεργασιών που θα ονομάσουμε εξυπηρετητές και δημιουργούνται για να δέχονται τις εντολές get και append στο σύνολο S.



Σχήμα 2.2: Στα αριστερά έχουμε ένα σύστημα που χρησιμοποιεί ένα AK και στα δεξιά έχουμε ένα σύστημα που χρησιμοποιεί ένα KAK.

2.3 Προηγούμενη Εργασία

Αυτή η εργασία βασίζεται κυρίως στο άρθρο *Formalizing and Implementing Distributed Ledger* [10], όπου παρουσιάζονται και επεξηγούνται οι αλγόριθμοι που αναφέρθηκαν πιο πάνω. Επίσης απέδειξαν την ορθότητα των αλγορίθμων οπότε η υλοποίηση τους στην γλώσσα DistAlgo θα είναι εφικτή.

Συνεχίζοντας η εργασία μας έχει ως βάση πτυχιακή με τίτλο *Υλοποίηση και πειραματική αξιολόγηση Αλγόριθμων Αλυσίδας Κοινοποιήσεων με τη χρήση ZeroMQ* [9] και ο στόχος της ήταν η υλοποίηση και η αξιολόγηση των αλγόριθμων τελικής, ατομικής και ακολουθιακής συνέπειας για Αλυσίδες Κοινοποιήσεων σε ένα Κατανεμημένο Σύστημα. Για την υλοποίηση των αλγορίθμων χρησιμοποιήθηκε η βιβλιοθήκη ZeroMQ της γλώσσας C η οποία διευκολύνει τον προγραμματισμό υποδοχών (sockets) για επικοινωνία μεταξύ δύο ή περισσότερων συσκευών. Επιπρόσθετα έγινε πειραματική αξιολόγηση στο PlanetLab, το οποίο είναι ένα κατανεμημενο ερευνητικό δίκτυο με σκόπο την ανάπτυξη νέων δικτυακών εφαρμογών και υπηρεσιών σε παγκόσμια κλίμακα, για αξιολόγηση των αλγορίθμων.

Η εργασία μου θα κάνει χρήση των αλγόριθμων που διατυπώθηκαν στην προηγούμενη εργασία. Οι αλγόριθμοι αυτοί υλοποιούνται στην εργασία μου με την χρήση του εργαλείου DistAlgo της Python. Ακομή με την χρήση της PyQt5 δημιουργούμε μια διαπροσωπεία για την εύκολη χρήση και παρουσίαση των αλγόριθμων, όπου διαπροσωπεία δεν υπάρχει στην ΑΔΕ που μελετήθηκε. Η PyQt5 είναι επίσης μια βιβλιοθήκη της γλώσσας Python και βοηθά στην δημιουργία διαδραστικών διαπροσωπειών.

2.4 DistAlgo

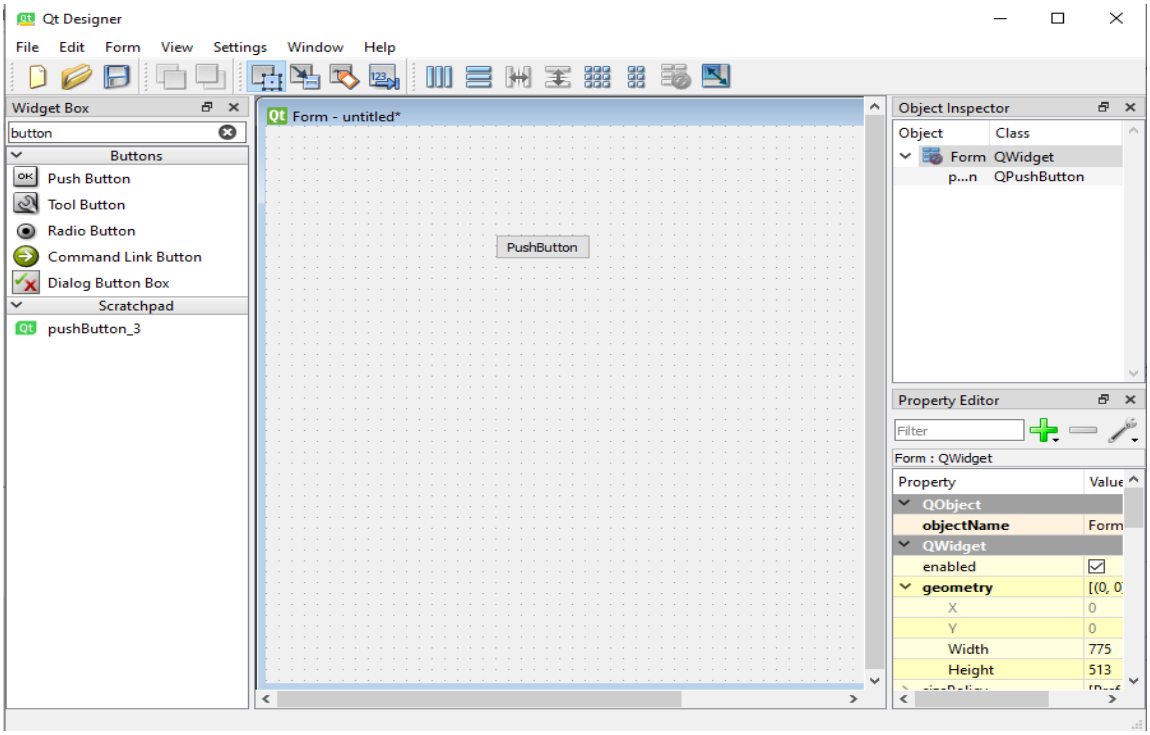
Η DistAlgo [11], όπως αναφέρθηκε πιο πάνω, είναι μια βιβλιοθήκη της γλώσσας Python η οποία λειτουργεί σε πολύ ψηλό επίπεδο προγραμματισμού αλγόριθμων κατανεμημένων συστημάτων. Στόχος της γλώσσας είναι η διευκόλυνση του προγραμματισμού αλγόριθμων και η δημιουργία ενός καθαρού και ευανάγνωστου κώδικα. Ο τρόπος με τον οποίο το καταφέρνει αυτό είναι με την αυτοματοποίηση της δημιουργίας των υποδοχών (sockets) που χρειάζονται δύο διεργασίες για να ανταλλάξουν μηνύματα και με συναρτήσεις που μπορούν να χρησιμοποιηθούν για βασικές λειτουργίας μιας διεργασίας. Ο μόνος περιορισμός στην επικοινωνία δύο διεργασιών είναι πως ο αποστολέας πρέπει να πάρει σαν παράμετρο τη διεργασία του αποδέκτη και αυτός είναι και ο λόγος ο οποίος δεν κατάφερα να δημιουργήσω το μοντέλο client-server σε περισσότερο από μια συσκευές.

Τα αρχεία της παίρνουν την επέκταση .da και για να τρέξει ένα πρόγραμμα της DistAlgo πρέπει να καλέσουμε το module του μεταγλωττιστή da, που προσφέρεται με την βιβλιοθήκη. Για παράδειγμα, για να τρέξω ένα πρόγραμμα με όνομα pingpong.da χρειάζεται να τρέξω την εντολή python -m da pingpong.da όπου -m για να διαλέξω το module da.

2.5 PyQt5

Η PyQt5 [12] είναι μια βιβλιοθήκη της Python που βοηθά στη δημιουργία διαδραστικής διαπροσωπεία μέσω κώδικα. Τα αντικείμενα που προσθέτουμε στο παράθυρο ονομάζονται Widgets και υπάρχουν πληθώρες επιλογές που μπορούμε να διαλέξουμε. Υπάρχει η επιλογή στοίχισης των αντικειμένων με τη χρήση layouts, το οποίο είναι σύστημα παρόμοιο με τα grid της bootstrap στο web design και χρήση κώδικα παρόμοιου με css για το styling των αντικειμένων. Ακόμη μπορούμε να δημιουργήσουμε κινούμενα σχέδια, τα οποία θα χρησιμοποιήσουμε και στο γραφικό μας περιβάλλον για να δείξουμε την πορεία των εγγραφών μέχρι το Αντικείμενο Κατάστιχου.

Μαζί με την PyQt5 υπάρχει και η επιλογή του προγράμματος Qt Designer [14] που βοηθά τον προγραμματιστή να δημιουργήσει μια διαπροσωπεία με χρήση γραφικού περιβάλλοντος αντί για κώδικα, το οποίο μπορεί στη συνέχεια να μετατρέψει σε κώδικα με το executable pyuic5.exe. Βλέποντας και το Σχήμα 2.5 αριστερά βρίσκονται όλα τα αντικείμενα που μπορούμε να προσθέσουμε στο παράθυρο μας με ένα απλό Drag and Drop. Το παράθυρο στην μέση είναι το παράθυρο που υλοποιούμε και του οποίου μπορούμε να δούμε τις ιδιότητες στο δεξιό παράθυρο και να κάνουμε διάφορες αλλαγές.



Σχήμα 2.5: Qt Designer API με ένα παράθυρο και ένα κουμπί

Κεφάλαιο 3

Αλγόριθμοι προς Υλοποίηση

3.1 Εισαγωγή

Σε αυτό το κεφάλαιο θα παρουσιαστούν οι αλγόριθμοι που θα υλοποιηθούν στην γλώσσα DistAlgo. Αυτοί οι αλγόριθμοι, όπως αναφέρθηκαν και σε προηγούμενες ενότητες, είναι οι αλγόριθμοι Κατανεμημένων Αντικειμένων Κατάστιχου με τελική και ατομική συνέπεια και ο πυρήνας των αλγόριθμών, η Υπηρεσία Ατομικής Πολυεκπομπής.

3.2 Υπηρεσία Ατομικής Πολυεκπομπής

Η Υπηρεσία Ατομική Πολυεκπομπής (Atomic Broadcast service), επίσης γνωστή και ως Υπηρεσία Ολικής Διατεταγμένης Πολυεκπομπής, είναι η υπηρεσία που φροντίζει τη σωστή μετάδοση των μηνυμάτων μεταξύ των εξυπηρετητών στο σύνολο S. Μπορεί να εκτελέσει δύο ειδών ενέργειες:

- **ABroadcast(m):** για να στείλει αντίγραφα του μηνύματος m σε όλους τους υπόλοιπους εξυπηρετητές στο σύνολο S

- **ADeliver(m):** για να στείλει ένα μήνυμα m σε ένα εξυπηρετητή στο σύνολο S

Η υπηρεσία αυτή έχει ως στόχο να διατηρήσει τις τέσσερις παρακάτω ιδιότητες:

- Εγκυρότητα (Validity): Αν ένας σωστός εξυπηρετητής στείλει ένα μήνυμα σε ένα άλλο εξυπηρετητή, τότε το μήνυμα θα φτάσει στον προορισμό του σε κάποια στιγμή στο μέλλον.
- Ομοιόμορφη συμφωνία (Uniform Agreement): Εάν ένας εξυπηρετητής στείλει ένα μήνυμα, τότε όλοι οι σωστοί εξυπηρετητές θα το παραλάβουν σε κάποια στιγμή.
- Ομοιόμορφη ακεραιότητα (Uniform Integrity): Ένα μήνυμα μπορεί να παραδοθεί από κάθε εξυπηρετητή το πολύ μια φορά και αυτό ισχύει μόνο εάν υπήρξε εξυπηρετητής που το μετάδωσε σε όλους τους υπόλοιπους προηγουμένως, δηλαδή εκτέλεσε την ενέργεια ABroadcast.
- Ομοιόμορφη απόλυτη διάταξη (Uniform Total Order): η σειρά στα οποία τα μηνύματα παραδίδονται πρέπει να είναι η ίδια μεταξύ όλων των εξυπηρετητών στο σύνολο S. Για παράδειγμα αν ένας server παραδώσει μήνυμα m πριν να παραδώσει ένα μήνυμα m', τότε κάθε server πρέπει να τα παραδώσει με αυτή την σειρά.

Ο αλγόριθμος που θα χρησιμοποιήσουμε για την Υπηρεσία Ατομικής Πολυεκπομπής ονομάζεται αιτιώδης ιστορικό (causal history, Σχήμα 3.2.1). Η ιδέα αυτού του αλγόριθμου είναι να βοηθάει τις διεργασίες τις οποίες καθυστερούν να παραλάβουν το μήνυμα να προλάβουν τις διεργασίες που παίρνουν τα μηνύματα πιο γρήγορα. Υποθέτουμε πως η αποστολή μηνυμάτων είναι FIFO (First In First Out) για μηνύματα που στέλνονται από μια διεργασία. Για παράδειγμα, αν μια διεργασία παραλάβει τα μηνύματα m1 και m2 με την αντίστοιχη σειρά, τότε θα αποστείλουν τα μήνυμα στις υπόλοιπες διεργασίες με αυτή την σειρά. Δηλαδή δεν θα υπάρχει περίπτωση η οποία θα ισχύει ότι $m2 \rightarrow m1$.

Ο causal history αλγόριθμος λειτουργεί με το σύστημα λογικού ρολογιού που ονομάζεται Lamport clock. Το ρολόι Lamport αυξάνεται κατά 1 πριν από κάθε πράξη που εκτελεί, ενθυλακώνει με αυτό τα μηνύματα που θα στείλει σε άλλες διεργασίες και κατά την

```
Senders and destinations (code of process p; assumes FIFO channels):
    Initialization:
        received_p := ∅                                   { Messages received by process p }
        delivered_p := ∅                                  { Messages delivered by process p }
        LC_p[p_1 ... p_n] := {0, ..., 0}    { LC_p[q]: logical clock of process q, as seen by p }
    procedure TO-multicast(m)                             { To TO-multicast a message m }
        LC_p[p] := LC_p[p] + 1
        ts(m) := LC_p[p]
        send FIFO (m, ts(m)) to all
    when receive (m, ts(m))
        LC_p[p] := max(ts(m), LC_p[p]) + 1
        LC_p[sender(m)] := ts(m)
        received_p := received_p ∪ {m}
        deliverable := ∅
        for each message m' in received_p \ delivered_p do
            if ts(m') ≤ min_{q∈Π} LC_p[q] then
                deliverable := deliverable ∪ {m'}
        deliver all messages in deliverable, in increasing order of (ts(m), sender(m))
        delivered_p := delivered_p ∪ deliverable
```

Σχήμα 3.2.1: Απλός αλγόριθμος επικοινωνίας με ιστορικό (causal history) [1]

παραλαβή μηνύματος αλλάζει ανάλογα με το μέγιστη τιμή του ρολογιού μεταξύ του αποστολέα και της ίδιας της διεργασίας.

Ο αλγόριθμος αρχίζει δημιουργώντας τις λίστες received, delivered και LC. Η λίστα received κρατάει όλα τα μηνύματα τα οποία παραλαμβάνει η διεργασία, η delivered περιέχει τα μηνύματα τα οποία αποστέλνονται στη διεργασία που δημιούργησε το μήνυμα. Επιπρόσθετα η λίστα LC σε μια διεργασία p περιέχει τα ρολόγια Lamport για κάθε διεργασία, όπως είναι γνωστά στην p. Για κάθε διεργασία υπάρχουν δύο διαφορετικές ενέργειες που μπορεί να εκτελέσουν.

Η πρώτη ενέργεια είναι να στείλει σε όλες τις διεργασίες ένα μήνυμα m, όμως πριν την αποστολή του μηνύματος, το ρολόι της διεργασίας θα αυξηθεί κατά 1 και μετά θα ενθυλακώσει το ρολόι στο μήνυμα με τη μορφή (m, ts(m)) όπου το ts(m) το timestamp του μηνύματος. Η δεύτερη ενέργεια που μπορεί να πράξει μια διεργασία είναι να περιμένει να παραλάβει ένα ενθυλακωμένο μήνυμα (m, ts(m)) και να αποφασίσει ποια αποθηκευμένα μηνύματα θα παραδοθούν. Με την παραλαβή του ενθυλακωμένου μηνύματος από μια διεργασία p, το ρολόι της διεργασίας παίρνει τιμή ίση με το μεγαλύτερο αριθμό ανάμεσα στο ρολόι της και του timestamp του μηνύματος και προσθέτει ένα. Επιπρόσθετα αλλάζει το ρολόι του αποστολέα του ενθυλακωμένου μηνύματος με το timestamp που παρέλαβε. Επειδή υποθέτουμε πως η επικοινωνία γίνεται με FIFO, υποθέτουμε επίσης ότι με κάθε μήνυμα από μια συγκεκριμένη διεργασία το timestamp θα είναι μεγαλύτερο.

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

```
1: c ← 0
2: Let L ⊆ S : |L| ≥ f + 1
3: function L.get( )
4:     c ← c + 1
5:     send request (c, GET) to the servers in L
6:     wait response (c, GETRES, V) from some
   i ∈ L
7:     return V
8: function L.append(r)
9:     c ← c + 1
10:    send request (c, APPEND, r) to the servers in
   L
11:    wait response (c, APPENDRES, res) from
   some i ∈ L
12:    return res
```

Σχήμα 3.2.2: Αλγόριθμος για διεργασία τύπου πελάτη που χρησιμοποιεί το ΚΑΚ [10]

Είναι σημαντικό να προσθέσουμε πως με τον υφιστάμενο αλγόριθμο, υπάρχει ένα πρόβλημα στο θέμα ενημέρωσης του ρολογιού κάποιας άλλης διεργασίας. Για να παραλάβει μια διεργασία s την επικράτουσα τιμή του ρολογιού μιας διεργασίας p, χρειάζεται ο p να εκπέμψει σε όλες τις διεργάσιες ένα ενθυλακωμένο μήνυμα. Άρα συμπεραίνουμε πως αν τύχει μια διεργασία να μην αποστείλει μήνυμα, τότε η τιμή του ρολογιού που γνωρίζουν οι υπόλοιπες διεργασίες θα παραμείνει 0.

Ο τρόπος αντιμετώπισης αυτού του προβλήματος είναι με κάθε παραλαβή ενθυλακωμένου μηνύματος, η διεργασία μετά από την αλλαγή του δικού της ρολογιού, θα αποστείλει ένα άδειο μήνυμα που περιέχει την τιμή του ρολογιού LC της διεργασίας σε όλες τις υπόλοιπες διεργασίες. Με αυτή τη μέθοδο εξασφαλίζουμε πως το κάθε process θα παραλαμβάνει τακτικά τις αναβαθμισμένες τιμές ρολογιού όλων των διεργασιών.

3.3 Διεργασία Πελάτης

Σε ένα σύστημα πελάτες – εξυπηρετητές, οι πελάτες έχουν την δυνατότητα να διεξάγουν δύο ενέργειες, append και get (Σχήμα 3.2.2). Το append για ένα μήνυμα m σημαίνει πως αποστέλνεται το m στους εξυπηρετητές για να προστεθούν στο τέλος του ΚΑΚ. Η ενέργεια get είναι η αίτηση για να παραλάβει ο πελάτης τις πληροφορίες που είναι αποθηκευμένες στο ΚΑΚ.

Οι διεργασίες πελάτων έχουν δύο δεδομένα: (α) το λογικό ρολόι c και (β) το σύνολο L το οποίο είναι υποσύνολο του συνόλου S όλων των εξυπηρετητών. Ο ρόλος του ρολογιού είναι για να γίνεται κάθε μήνυμα μοναδικό, ώστε να μπορούν οι επεξεργαστές να καθορίσουν την σειρά εισαγωγής τους στο ΚΑΚ. Το υποσύνολο L είναι το σύνολο των επεξεργαστών στους οποίους ο πελάτης θα στέλνει τις αιτήσεις του. Ο λόγος που είναι αναγκαίο να

μειώσουμε τον αριθμό των επεξεργαστών που αποστέλνονται οι αιτήσεις είναι για να μειωθεί η κυκλοφορία του δικτύου, αλλά ταυτόχρονα υποθέτουμε πως ο αριθμός των ενεργών εξυπηρετητών είναι ίση ή μεγαλύτερη την τιμή f. Σε περίπτωση που δεν ισχύει, τότε υπάρχει πιθανότητα κάποιος πελάτης να μην μπορεί να αποστείλει το αίτημα του.

Η ενέργεια append και η ενέργεια get έχουν την ίδια δομή. Πρώτα το λογικό ρολόι c αυξάνει την τιμή του κατά 1, μετά στέλνει το αίτημα σε όλους τους επεξεργαστές στο υποσύνολο L και περιμένουν απάντηση για να την επιστρέψουν. Η απάντηση του εξυπηρετητή για το αίτημα append είναι 'ACK', όπου επιβεβαιώνει την προσθήκη του αντικειμένου στο KAK και η απάντηση του get είναι το σύνολο των αποθηκευμένων εγγραφών.

3.4 Διεργασία Εξυπηρετητή

Η διεργασία του Εξυπηρετητή είναι υπεύθυνη να παραλαμβάνει τις αιτήσεις GET και APPEND των πελατών. Με την παραλαβή της αίτησης GET από τον εξυπηρετητή, αναλαμβάνει να επιστρέψει στον πελάτη αποστολέα τον Ledger που ονομάζουμε σύνολο εγγραφών S. Με την παραλαβή της αίτησης APPEND για ένα μήνυμα m, ο εξυπηρετητής προσθέτει το m στο σύνολο S και επιστρέφει στον πελάτη το μήνυμα 'ACK' που δηλώνει πως το μήνυμα παραλήφθηκε και προστέθηκε στον Ledger.

Σε αυτή την εργασία θα μελετήσουμε 2 αλγόριθμους συνέπειας, οι οποίοι έχουν διαφορετικό στόχο μεταξύ τους αλλά παρόμοια δομή, τον αλγόριθμο της τελικής συνέπειας και τον αλγόριθμο της ατομικής συνέπειας.

3.4.1 Τελική Συνέπεια

Ο αλγόριθμος Τελικής Συνέπειας λειτουργεί με τον εξής τρόπο. Για κάθε αίτημα append που παραλαμβάνει ένας εξυπηρετητής, εκπέμπει το μήνυμα για προσθήκη στους υπόλοιπους εξυπηρετητές και αμέσως μετά στέλνει μήνυμα acknowledgement στον αποστολέα. Όταν το μήνυμα παραδοθεί, τότε το μήνυμα μπαίνει στο σύνολο S. Για κάθε αίτημα get που παραλαμβάνει ο εξυπηρετητής, στέλνει στον αποστολέα το σύνολο S.

Η ιδέα του αλγόριθμου τελικής συνέπειας είναι να παραδίδουν την απάντηση τους στους αποστολείς, αμέσως μετά από την πολυεκπομπη του μηνύματος. Αυτό έχει ως

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

```
1: Init: S_i ← ∅
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 ∉ S_i then S_i ← S_i‖r
```

Σχήμα 3.4.1: Αλγόριθμος Τελικής Συνέπειας για εξυπηρετητές [10]

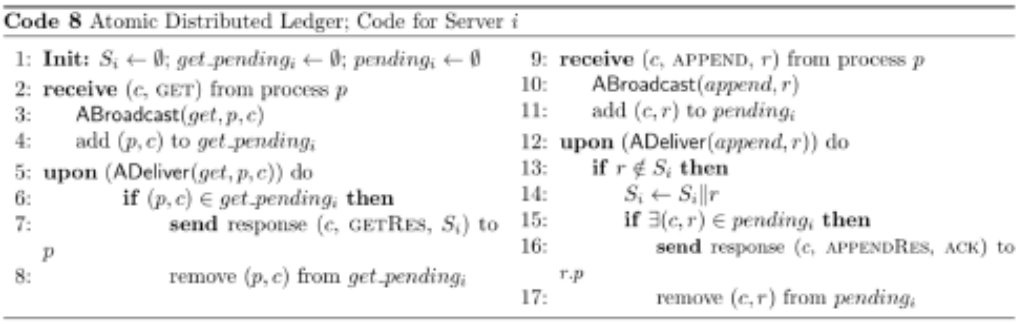
αποτέλεσμα η απάντηση του εξυπηρετητή να μην ακολουθεί τον πραγματικό χρόνο, αλλά να εξαρτάται από την ταχύτητα που ο εξυπηρετητής τελειώσει την ενέργειά τους.

Για παράδειγμα έστω ένα πελάτης c στέλνει ένα αίτημα append για ένα μήνυμα m και στη συνέχεια στείλει ένα αίτημα get στο σύνολο εξυπηρετητών L. Αφού γνωρίζουμε πως το η απάντηση του εξυπηρετητή στέλνεται αμέσως μετά από το εκπομπή του λαμβάνοντως μηνύματος στους υπόλοιπους εξυπηρετητές, τότε μπορούμε να συμπεράνουμε πως υπάρχει περίπτωση το αίτημα get να εξυπηρετηθεί πριν το μήνυμα m να προστεθεί στο σύνολο S. Οπότε παρ'όλο που το append υπήρξε το πρώτο αίτημα του c, το get ολοκληρώθηκε πρώτο.

3.4.2 Ατομική Συνέπεια

Ο αλγόριθμος Ατομικής Συνέπειας έχει την ίδια δομή με αυτό της Τελικής Συνέπειας. Η μόνη τους διαφορά είναι στο πότε στέλνουν απάντηση στον αποστολέα του αιτήματος. Στην Ατομική Συνέπεια η ανταπόκριση αποστέλνεται μέτα από την παράδοση, οπότε εξουδετερώνεται η πιθανότητα ένα get αίτημα από ένα πελάτη c που στάλθηκε μετά από ένα αίτημα append για μήνυμα r να μην αποτελείται από το μήνυμα r.

Στον αλγόριθμο Ατομικής Συνέπειας εισάγουμε 2 νέες λίστες, τη pending και τη get_pending. Η pending κρατάει όλες τις αιτήσεις append των πελατών και η get_pending κρατάει όλες τις αιτήσεις get που αναμένουν ανταπόκριση. Όταν ένας επεξεργαστής ο οποίος τρέχει τον αλγόριθμο ατομικής συνέπειας παραλάβει αίτημα τύπου get, τότε όπως και στο append εκπέμπει το μήνυμα σε όλους τους ενεργούς εξυπηρετητές. Όμως για αυτό τον τύπο συνέπειας, οι εξυπηρετητές αντί να ανταποκρίνονται κατευθείαν μετά την πολυεκπομπή, αποθηκεύουν το αίτημα στην λίστα pending και get_pending αντίστοιχα.



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: $S_i \leftarrow S_i \| r$
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 3.4.2: Αλγόριθμος εξυπηρετητή με Ατομική Συνέπεια σε ΚΑΚ [10]

Κατά την παράδοση get αίτηματος, γίνεται έλεγχος αν το αίτημα υπάρχει στην λίστα get_pending. Αν υπάρχει τότε γίνεται ανταπόκριση στον αποστολέα. Παρόμοια κατά την παράδοση append αιτήματος, γίνεται έλεγχος αν υπάρχει το αντικείμενο στο σύνολο S που αποθηκεύονται οι εγγραφές των πελατών. Αν υπάρχει τότε γίνεται έλεγχος αν υπάρχει το αίτημα στη λίστα pending. Αν υπάρχει τότε στέλνεται ανταπόκριση στον αποστολέα με το ACK.

Η υλοποίηση αυτή σε συνδυασμό με το ΥΑΠ έχει τις εξής ιδιότητες:

- Αν δύο διεργασίες p1 και p2 αποστείλουν αίτημα append για r1 και r2 αντίστοιχα ώστε το append (r1) να φτάσει πιο γρήγορα από το append (r2) τότε το r1 εμφανίζεται πριν το r2 σε κάθε ακολουθία που επιστρέφεται από το Ledger.
- Αν υπάρχει append αίτημα από διεργασία p1 για αντικείμενο r1 και ακολουθήσει αίτημα get από διεργασία p2, τότε το r1 θα εμφανιστεί στην ακολουθία που θα επιστραφεί για το αίτημα get.
- Αν π1 και π2 δύο ενέργειες get από διεργασίες p1 και p2 αντίστοιχα ώστε το π2 ακολουθεί το π1 και τα οποία get επιστρέφουν τις ακολουθίες S1 και S2 αντίστοιχα, τότε η ακολουθία S1 πρέπει να είναι πρόθεμα της S2.
- Αν υπάρχει αίτημα get από διεργασία p1 το οποίο ακολουθείται από αίτημα append από διεργασία p2 για αντικείμενο r2, τότε το r2 δεν θα εμφανίζεται στην ακολουθεία που θα επιστραφεί με το αίτημα get της p1.

3.5 Μηχανισμός Ανοχής Σφαλμάτων

Ο Μηχανισμός Ανοχής Σφαλμάτων που θα χρησιμοποιήσουμε είναι ο εξής. Όπως προαναφέρθηκε στην υποενότητα 3.2. οι πελάτες έχουν την μεταβλητή L, η οποία ορίζεται ως ένα υποσύνολο του συνόλου S των εξυπηρετητών με μέγεθος f + 1. Η πελάτες για να στείλουν ένα αίτημα προς το Αντικείμενο Κατάστιχου, στέλνουν αντίγραφα του αιτήματος

σε f + 1 εξυπηρετητές, όπου f είναι μια προκαθορισμένη σταθερά για τους λανθασμένους εξυπηρετητές. Αυτό σημαίνει πως υποθέτουμε πως δεν θα υπάρξει μεγαλύτερος αριθμός εσφαλμένων εξυπηρετητών από το f, άρα οι πελάτες θα μπορέσουν να περάσουν το αίτημα τους για το Αντικείμενο Κατάστιχου σε τουλάχιστον ένα εξυπηρετητή.

Οι εξυπηρετητές με την σειρά τους χρειάζεται να βρίσκουν τον εσφαλμένο εξυπηρετητή, ώστε να τον αφαιρούν από τους ενεργούς. Οπότε θα χρησιμοποιηθούν ιδέες από τον Τερματικο Αξιόπιστο αλγόριθμο Πολυεκπομπής [2] (Terminating Reliable Broadcast). Η ιδέα του Τερματικού αλγόριθμου είναι για κάθε εξυπηρετητή να βρίσκει ποιος εξυπηρετητής προκαλεί καθυστέρηση στο σύστημα και να ενημερώνει τους υπόλοιπους εξυπηρετητές. Μετά την ενημέρωση, κάθε εξυπηρετητής θα αφαιρέσει τον εξυπηρετητή από το ενεργό σύνολο που φυλάει στην προσωπική του μνήμη. Το ερώτημα που δημιουργείται είναι πως ένας επεξεργαστής βρίσκει τον επεξεργαστή αυτό.

Για να βρεθεί ένας εσφαλμένος εξυπηρετητής δηλώνουμε έναν μετρητή FTLC για καθε εξυπηρετητή παρόμοιος με τον LC, δηλαδή για κάθε εξυπηρετητή. Κάθε μήνυμα που παραλαμβάνεται από έναν εξυπηρετητή προσθέτει ένα σε όλους τους εξυπηρετητές στην FTLC, ενώ για τον αποστολέα εξυπηρετητή μηδενίζεται η τιμή. Αν υπάρχει FTLC τιμή που ξεπερνά ένα συγκεκριμένο THRESHOLD, τότε ο εξυπηρετητής που το ανακάλυψε στέλνει σε όλους να τον αφαιρέσουν από το σύνολο των ενεργών. Έτσι σιγουρεύεται η ομαλή λειτουργία του αλγόριθμου χωρίς καθυστερήσεις.

3.6 Προκλήσεις

Μια από τις μεγαλύτερες προκλήσεις στον προγραμματισμό είναι το debugging για λάθη στον αλγόριθμο του προγράμματος. Το λάθος μπορεί να βρίσκεται σε ένα ψηφίο ή σε μια μεταβλητή ή στην λογική του προγράμματος. Η ασύγχρονη φύση της εργασίας μεγενθύνει το πρόβλημα διότι δεν υπάρχει σταθερή συμπεριφορά στο πρόγραμμα, όποτε υπάρχει δυσκολία στην εύρεση του προβλήματος. Για παράδειγμα έστω υπάρχουν τρεις διεργασίες c1,c2,c3 που στέλνουν ταυτόχρονα τον αριθμό 1,2,3 αντίστοιχα στην διεργασία s και η διεργασία s τα βάζει σε μια λίστα r. Αφού έχουμε ασύγχρονο περιβάλλον, τότε η τελική λίστα της διεργασίας s μπορεί να είναι ένα από τα ακόλουθα: (1 2 3), (1 3 2), (2 3 1), (2 1 3), (3 2 1), (3 1 2).

Μια άλλη εξίσου σημαντική πρόκληση είναι η εύρεση της σταθεράς THRESHOLD για την εύρεση των εσφαλμένων εξυπηρετητών, δηλαδή ποια θα πρέπει να είναι η διαφορά

του μεγαλύτερου και του μικρότερου μετρητή FTLC για να διαγραφθεί ένας εξυπηρετητής από τους ενεργούς εξυπηρετητές. Θα χρειαστούν συνεχείς προσπάθειες για να βρεθεί ένα σημείο αυτό.

Κεφάλαιο 4

Υλοποίηση Αλγορίθμων

4.1 Εισαγωγή

Στο κεφάλαιο αυτό θα επεξηγήσουμε την υλοποίηση που δημιουργήσαμε για τους αλγόριθμους που αναλύθηκαν στο προηγούμενο κεφάλαιο και για την μορφή των μηνυμάτων. Όπως προαναφέρθηκε χρησιμοποιούμε την γλώσσα DistAlgo, οπότε χρειάζεται να επεξηγήσουμε τις συναρτήσεις που χρησιμοποιήθηκαν.

4.2 Λειτουργίες Συστήματος DistAlgo

Εδώ θα αναλύσουμε την κάθε λειτουργία που προσφέρει η DistAlgo και θα χρησιμοποιήσουμε στο κώδικά μας. Περισσότερες πληροφορίες υπάρχουν στη βιβλιογραφία [4].

Αρχικα για να ορίσουμε μια διεργασία πρέπει να ορίσουμε την κλάση που θα τρέχει η διεργασία. Για την δημιουργία της κλάσης πρέπει να υιοθετήσει από την in-build κλάση process όπως φαίνεται πιο κάτω:

def class_name(process):

process_body

Στη συνέχεια όπως είναι αναμενόμενο από μια πολύ ψηλού επιπέδου γλώσσας προγραμματισμού, υπάρχει μια συνάρτηση η οποία δημιουργεί τις διεργασίες και ονομάζεται new. Η μορφή που θα χρησιμοποιήσουμε στον κώδικά μας είναι η εξής: *new(p,num,at)* όπου p είναι κλάση που υιοθετά από την κλάση process, το *num* είναι ο αριθμός των διεργασιών που θα δημιουργηθούν και το *at* είναι η διεύθυνση της μηχανής που θέλουμε να τρέξει η διεργασία. Για την προετοιμασία των διεργασιών υπάρχει επίσης και η συνάρτηση *setup (pexp,(args))* όπου pexp είναι η διεργασία για setup και οι args είναι οι παραμέτροι που θα μεταφερθούν στην διεργασία. Στην κλάση ορίζουμε την συνάρτηση setup όπου αντικαθιστά την setup της κλάσης process και ορίζουμε τα δεδομένα της κλάσης πριν να αρχίσει η διεργασία να τρέχει. Είναι σημαντικό να προστεθεί πως για παραμέτρους θα παίρνει ότι προσθέσουμε στο κάλεσμα της συνάρτησης στην main του αρχείου .da. Επίσης για να ορίσουμε τι θα τρέχει η διεργασία μετά το setup πρέπει να ορίσουμε στην κλάση τη συνάρτηση run όπου αντικαθιστά την συνάρτηση τις process, η οποία δεν παίρνει παραμέτρους. Για να τρέξει η διεργασία που ορίσαμε, αρκεί να τρέξουμε στην main() του αρχείου .da την εντολή start(pexp) όπου pexp η διεργασία.

Αναφέραμε πιο πάνω πως το ΥΑΠ για να λειτουργεί, χρειάζεται να λειτουργεί με κάτι που ονομάζεται Lamport clock και η επικοινωνία μεταξύ των διεργασιών να είναι με σειρά FIFO. Για να επιτευχθεί αυτό στην γλώσσα DistAlgo αρκεί να γράψουμε την ακόλουθη εντολή στην main του αρχείου .da πριν να τρέξουμε την εντολή start() :

config (channel=’fifo’, clock=’Lamport’)

Η αποστολή μηνυμάτων μεταξύ δύο διεργασιών υπό κανονικές συνθήκες χρειάζεται την διαδικασία δημιουργίας υποδοχών, ωστόσο στο DistAlgo αρκεί να τρέξει η εντολή *send (mexp, to = pexp)*, όπου *mexp* το μήνυμα που θα σταλεί και το *to* ο προορισμός του μηνύματος (διεργασία). Για να ορίσουμε ποια μηνύματα θα δέχεται μια διεργασία ορίζουμε συνάρτηση με τον εξής τρόπο:

def receive (msg=mexp, from_= pexp):

handler_body

Το msg είναι το μήνυμα που περιμένει η διεργασία και το from είναι από ποιο process.

Επιπρόσθετα η DistAlgo δίνει την δυνατότητα μιας διεργασίας να περιμένει μέχρι να πληρούν κάποιες απαιτήσεις. Η εντολη που το καθιστά δυνατό είναι η εξής:

if await (bexp1): *stmt1*
elif timeout(t): stmt

όπου bexp1 μια συνθήκη όπου πρέπει να γίνει Αληθές για να εκτελέσει το stmt1. Αν περάσει ο χρόνος t που καθορίζεται στο timeout, πριν το bexp1 να γίνει Αληθές τότε τρέχει το stmt.
Οι επόμενες εντολές μπορούν να χρησιμοποιηθούν μέσα στο await ως συνθήκες:

- received (mexp, from_=pexp) διαφέρει από την συνάρτηση receive στο γεγονός ότι απλά ελέγχει αν πάρει η διεργασία το μήνυμα mexp. Η τιμή του είναι False μέχρι να παραλάβει το μήνυμα
- some (p in pexp, has=received (mexp, from_=p)) όπου ελέγχει αν παραλάβει η διεργασία μήνυμα mexp από τουλάχιστον μια διεργασία p στο σύνολο pexp. Όπως και το πρώτο, η τιμή του είναι False μέχρι να παραλάβει το μήνυμα από τουλάχιστον ένα από τα pexp.
- each (p in pexp, has=received (mexp, from_=p)) όπου ισχύει το ίδιο με το some απλά πρέπει να παραλάβει το μήνυμα mexp από όλες τις διεργασίες στο σύνολο pexp για να γίνει Αληθές.

Τέλος για εκτύπωση μπορούμε να χρησιμοποιήσουμε κανονικά την print ή αν θέλουμε να εμφανίζεται η διεργασία που επιστρέφει το μήνυμα στην γραμμή εντολής, τρέχουμε την εντολη output(exp).

4.3 Μορφή Μηνυμάτων

Όπως προαναφέραμε, ένα Κατανεμημένο Σύστημα είναι ένα σύστημα με διάφορες μηχανες είτε τοπικές, είτε κατανεμημένες σε διαφορετικές περιοχές, οπότε στις πλήστες περιπτώσεις δεν υπάρχει κοινή μνήμη. Ο μοναδικός τρόπος να ανταλλάξουν δεδομένα δύο μηχανές στο ίδιο Κατανεμημένο Σύστημα είναι με ανταλλαγή μηνυμάτων μεταξυ τους. Οπότε χρειάζεται να θέσουμε δομή για κάθε είδους μηνύματος ώστε η κάθε μηχανή να μπορεί να αποφασίσει την ορθή ενέργεια.

Αρχικά έχουμε το αίτημα GET από τον client προς τον server το οποίο έχει την μορφή (c,’GET’) όπου c είναι ο λογικός χρόνος που έχει κατεγραμμένος η διεργασία πελάτη. Με παρόμοιο τρόπο έχουμε και το αίτημα APPEND από τον client προς τον server το οποίο έχει την μορφή (c,’APPEND’,r) όπου r το μήνυμα που στέλνει ο ο client για προσθήκη στο σύνολο S.

Ακολούθως να αναφέρουμε τα μηνύματα που στέλνουν οι servers. Το πρώτο μήνυμα που ανταλλάσουν οι εξυπηρετητές μεταξύ τους είναι το μήνυμα της μεθόδου ABroadcast() με την μορφή (cserver,c,action,r,p) όπου το cserver είναι το ρολόι του server, και γνωστό ως timestamp του μηνύματος, το c είναι ο χρόνος του μηνύματος από τον client, το action μπορεί να είναι ‘GET’ ή ‘APPEND’, το r είναι το μήνυμα για εγγραφή και το p είναι ο client αποστολέας. Σε περίπτωση που το action είναι ‘GET’ τότε το r είναι πάντοτε άδειο. Παράδειγμα ενός μηνύματος APPEND που έτυχε πολυεκπομπής είναι (10,1,’APPEND’,’test1’,Client1) όπου 10 είναι ο χρόνος του εξυπηρετητή που έκανε την πολυεκπομπή, 1, ο χρόνος του πελάτη Client1 που έστειλε το αίτημα APPEND για το αντικείμενο test1. Αντίστοιχα για GET αίτημα η μορφή είναι η εξής (15,2,’GET’,’’,Client1) όπου 15 ο χρόνος πολυεκπομπής του μηνύματος, 2 ο χρόνος του πελάτη Client1 για αίτημα GET. Άλλο μήνυμα που ανταλλάσουν οι servers μεταξύ τους είναι το άδειο μήνυμα για συγχρονισμό των ρολογιών τους που έχει την μορφή (‘’,’’,’EMPTY’,’’,’’).

Τελευταίο τύπος μηνυμάτων είναι τα μηνύματα που στέλνουν οι εξυπηρετητές στους client ως ανταπόκριση στο αίτημα τους. Για ανταπόκριση σε αίτημα ‘APPEND’ έχουμε το μήνυμα (c,'APPENDRES', 'ACK') όπου c ο χρόνος του μηνύματος του αποστολέα πελάτη, ενώ για ανταπόκριση σε αίτημα ‘GET’ παίρνει την μορφή (c,'GETRES', S) όπου S είναι το σύνολο των εγγραφών που προσθέσαν οι πελάτες. Αν για παράδειγμα έτυχε ένα αίτημα APPEND για αντικείμενο ‘500’ στο σύνολο S και στη συνέχεια σταλεί ανταπόκριση για αίτημα GET, τότε το μήνυμα θα έχει την εξής μορφή (2,’GETRES’,{500}) όπου 2 ο χρόνος που στάληκε το αίτημα από τον πελάτη και {500} το σύνολο S.

Κάποια επιπρόσθετα μηνύματα που στέλνονται για λόγους συγχρονισμού simulation:

- ('stop',self) από servers σε clients για να σταματήσουν οι clients
- (‘done’,self) από servers και clients προς την διεργασία Main (περισσότερα για αυτή πιο κάτω) για να διακοπεί το πρόγραμμα και από την Main στον γονέα, ο οποίος είναι η main του αρχείου

4.4 Μηχανισμός Ανοχής Σφαλμάτων

Σε αυτό το σημείο ο Μηχανισμός Ανοχής Σφαλμάτων που υλοποιήθηκε δεν συνάδει τελείως με τον αλγόριθμο που αναφέραμε πιο πάνω στο υποκεφάλαιο 3.5. Αλλά η κύρια ιδέα του αλγόριθμου υπάρχει στην υλοποίηση των διεργασιών , δηλαδή οι πελάτες στέλνουν σε f + 1 εξυπηρετητές το αίτημα τους για εξυπηρέτηση για να μπορεί να υποστηρίξει το σύστημα καταρρεύσης των server.

Αρχίζοντας με τους πελάτες, οι πελάτες αντί να στέλνουν σε όλους του εξυπηρετητές, στέλνουν σε ένα από τους f + 1 εξυπηρετητές. Αν το αίτημα τους δεν εξυπηρετηθεί σε ένα χρονικό διάστημα που έθεσα τότε, ο πελάτης αλλάζει σε διαφορετικό server. Μέτα από πειράματα θέσαμε τον χρόνο ίσο με 0,5*L, όπου L είναι ο αριθμός των server που ο πελάτης θα επικοινωνεί.

Για τους servers έπρεπε να υλοποιηθούν 2 λειτουργίες. Η πρώτη λειτουργία είναι ο μηχανισμός για τερματισμό του server. Αποφασίσαμε πως ο καλύτερος τρόπος να το υλοποιήσουμε είναι με το να δίνουμε στην διεργασία των εξυπηρετητών μας μια παράμετρο crash, η οποία είναι ένας αριθμός τύπου float με πεδίο ορισμού [0,1] και να δηλώσουμε μια private μεταβλητή shutdown με αρχική τιμή το False. Για κάθε μήνυμα APPEND που παραλαμβάνει ένας επεξεργαστή δημιουργείται ένας ψευδοτυχαίος αριθμός με πεδίο ορισμού [0,1] με τη χρήση της συνάρτησης random.random() και το βάζω στην μεταβλητή rand. Μετά γίνεται έλεγχος if (crash>rand and faulty_servers<len (allservers)- f) με faulty_servers ο αριθμός των εσφαλμένων εξυπηρετητών, το len (allservers) ο αριθμός όλων των εξυπηρετητών και το f η προκαθορισμένη τιμή μέγιστων εσφαλμένων εξυπηρετητών. Αν ο έλεγχος είναι αληθής, τότε η μεταβλητή shutdown του εξυπηρετητή γίνεται True και σταματά να δέχεται μηνύματα ή να πράττει οτιδήποτε. Η άλλη εναλλακτική ήταν να κλείσει εντελώς η διεργασία αλλά καθυστερούσε πολύ τον τερματισμό του προγράμματος λόγω της λειτουργίας send, όπου δεν υπήρχει τρόπο να αλλάξουμε τον προκαθορισμένο χρόνο αναμονής της αποστολής του αιτήματος.

Η δεύτερη λειτουργία των επεξεργαστών είναι η εύρεση των τερματισμένων εξυπηρετητών. Στο μοντέλο που υλοποιήσαμε , αν ένας επεξεργαστής απενεργοποιηθεί τότε δεν θα στέλνει μηνύματα στους υπόλοιπους, άρα το ρολόι του δεν θα αλλάζει στους υπόλοιπους εξυπηρετητές. Αυτό έχει ως αποτέλεσμα οι εξυπηρετητές να σταματήσουν την παράδοση μηνυμάτων και το σύστημα να φτάσει σε αδιέξοδο. Ο μηχανισμός που διάλεξα να χρησιμοποιήσω εμπνεύστηκε από τον αλγόριθμο Τερματική Αξιόπιστη Πολυεκπομπή

(Terminating Reliable Broadcast). Έθεσα μια λίστα FLTC με μια θέση για κάθε εξυπηρετητή που υπάρχει, και όπου αυξάνεται κατά ένα με κάθε Broadcast που παραλαμβάνει. Αν το Broadcast είναι από το process p, τότε όλοι οι εξυπηρετητές μηδενίζουν τον μετρητή τους για τον συγκεκριμένο. Μετά από έναν αριθμό μηνυμάτων, βρίσκουμε την μεγαλύτερη και την μικρότερη τιμή του μετρητή και βρίσκουμε την διαφορά τους. Αν η διαφορά είναι μεγαλύτερη από μια προκαθορισμένη τιμή THRESHOLD, τότε ο εξυπηρετητής με την μεγαλύτερη τιμή διαγράφεται από την λίστα. Το THRESHOLD μετά από πειράματα αποφασίστηκε να πάρει την τιμή 5*N, όπου N ο αριθμός των ορθών εξυπηρετητών.

4.5 Προγραμματισμός Διεργασιών Εξυπηρετητή

Η διεργασία εξυπηρετητής χωρίζεται σε δύο κλάσεις: (α) την *AtomicBroadcastService* και (β) την *Atomic_Server*. Ο λόγος που χωρίστηκε ο εξυπηρετητής σε δύο κλάσεις είναι για να είναι εύκολη η δοκιμή του κάθε κομματιού του εξυπηρετητή, να γίνει ο κώδικας πιο ευανάγνωστος και να ξεχωρίσουν οι διαφορετικοί αλγόριθμοι που αναλύθηκαν στο κεφάλαιο 3. Η κλάση α είναι η υλοποίηση της ΥΑΠ και κρατάει τις λειτουργίες των συναρτήσεων *ABroadcast* και *ABroadcastGET*, όπου κάνουν πολυεκπομπή μηνύματος στους υπόλοιπους εξυπηρετητές, και της συνάρτησης που αναλαμβάνει τα εισερχόμενα μηνύματα broadcast. Η κλάση β είναι ότι αναφέραμε για Ατομική Συνέπεια μέχρι στιγμής.

4.5.1 Υπηρεσίας Ατομικής Προεκμπομπής

Η Υπηρεσία Ατομικής Πολυεκπομπής είναι μέρος της υλοποίησης της διεργασίας επεξεργαστή με Ατομική Συνέπεια. Ο ρόλος του είναι να παραδίδει τα μηνύματα που παραλαμβάνει στην ίδια σειρα (total order) με τους υπόλοιπους εξυπηρετητές. Εμπεριέχονται 4 λειτουργίες στην κλάση: (α) αναμετάδοση APPEND μηνύματος, (β) αναμετάδοση GET μηνύματος (γ) παραλαβή ενθυλακωμένου μηνύματος APPEND από άλλο εξυπηρετητή και (δ) παραλαβή ενθυλακωμένου μηνύματος GET από άλλο εξυπηρετητή.

Αρχίζουμε με την συνάρτηση αναμετάδοσης μηνύματος APPEND. Παίρνει ως παραμέτρους το c, που είναι ο λογικός χρόνος του αποστολέα του παραλαμβάνοντος αιτήματος, το r, το οποίο είναι το κυρίως μήνυμα και το p, το οποίο είναι η ταυτότητα της διεργασίας του αποστολέα. Στην συνάρτηση αυξάνεται η τιμή του λογικού ρολογιού LC του εξυπηρετητή p κατά 1. Στην συνέχεια ελέγχει αν υπάρχει το αίτημα αυτό στα εισερχόμενα

Σχήμα 4.5.1: *Ένας πελάτης στέλνει αίτημα σε όλους τους servers και οι servers ανταλλάσουν μηνύματα μεταξύ τους [1]*

μηνύματα *received* με τη χρήση μιας προσαρμοσμένης συνάρτησης has*SubTuple(subTuple,tuples,start),* όπου ελέγχει αν υπάρχει η υποπλειάδα δεδομένων subTuple που δίνεται μέσα σε ένα array από πλειάδες tuples αρχίζοντας τον έλεγχο σε κάθε πλειάδα από το start. Επιστρέφει True αν ισχύει, αλλιώς επιστρέφει False. Η υλοποίηση της *hasSubTuple* βρίσκεται στο Σχήμα 4.5.2. Αν η hasSubTuple επιστρέψει True δηλαδή υπάρχει στην *received*, τότε δεν θα την ξαναπροσθέσει. Τέλος ο εξυπηρετητής εκπέμπει το μήνυμα ενθυλακωμένο με τον δικό του λογικό χρόνο LC. Παρόμοια λειτουργεί και η αναμετάδοση του αιτήματος GET.

Συνεχίζουμε με την συναρτήση παραλαβής μηνυμάτων. Δηλώνεται με το όνομα def receive (msg=(cserver, c,action,r,p),from_=s) και η κάθε παράμετρος έχει τον εξή ρόλο:

- cserver: λογικός χρόνος του εξυπηρετητή που έπραξε την αναμετάδοση. Χρησιμοποιείται για την ενημέρωση του λεξιλογίου LC στην θέση c
- c: λογικός χρόνος του αποσταλμένου μηνύματος. Δίνει μοναδικότητα στο αίτημα ενός πελάτη
- action: Καθορίζει την ενέργεια του αιτήματος. Μπορεί να είναι ‘APPEND’, ‘GET’ και ‘EMPTY’ (άδειο), όπου το άδειο είναι για ενημέρωση των εξυπηρετητών για τους χρόνους των υπολοίπων.
- r: κυρίως μήνυμα του αποστολέα πελάτη
- p: διεργασία αποστολέας. Ακόμη ένα στοιχείο που καθορίζει την μοναδικότητα του αιτήμα

Στη συνέχεια βλέπουμε πως λειτουργεί η συνάρτηση που αναλύσαμε στο Κεφάλαιο 3 στην Python. Πρώτα απ’όλα ελέγχουμε αν ο επεξεργαστής είναι απενεργοποιημένος (self.shutdown==True). Αν δεν είναι Αληθής, συνεχίζει η συνάρτηση.

```
def hasSubTuple(subTuple,tuples,start):
    try:
        j=0
        for tupl in tuples:
            correct=True
            i=start
            for s in subTuple:
                if (str(s)!=str(tupl[i])):
                    correct=False
                    break
                i+=1
            if (correct):
                return j
            j+=1
        return False
    except Exception as e:
        output('Error')
```

Σχήμα 4.5.2: Συνάρτηση, η οποία ελέγχει αν μια υποπλειάδα ανήκει σε ένα πίνακα από πλειάδες.

Μετά από τον έλεγχο ενημερώνουμε το λογικό ρολόι LC του επεξεργαστή που παρέλαβε την εκπομπή του αιτήματος (self.LC[self], όπου self ο ίδιος ο επεξεργαστής που τρέχει την συνάρτηση) με βάση την μέγιστη τιμή μεταξύ του χρόνου του επεξεργαστή και του μηνύματος και στο αποτέλεσμα προσθέτει 1. Βρίσκουμε τον μέγιστο μεταξύ των 2 αριθμών με την in-build συνάρτηση max και έχει την μορφή *self.LC[self]=max(cserver,self.LC[self])+1*. Στη συνέχεια όπως και στην συνάρτηση αναμετάδοσης, γίνεται έλεγχος αν υπάρχει η πλειάδα (c,action,r,p) σε κάποιο από τις πλειάδες της self.received με τη χρήση της συνάρτησης hasSubTuple().

Ενδιάμεσα γίνονται οι ελέγχοι που αναλύθηκαν στην υποενότητα 4.4 για Μηχανισμό Ανοχής Σφαλμάτων. Στη συνέχεια δημιουργούμε μια άδεια λίστα deliverable και τη λίστα temp όπου σχηματίζεται από τα μηνύματα που υπάρχουν στην λίστα self.received και όχι στη λίστα self.delivered. Αυτή η λίστα θα χρησιμοποιηθεί στο επόμενο βήμα για επιλογή των μηνυμάτων για παράδοση. Για την επιλογή τρέχει ένα for loop ανάμεσα σε όλα τα μηνύματα στη λίστα temp. Για κάθε μήνυμα αν ο χρόνος *cserver* του επεξεργαστή είναι μικρότερος από την ελάχιστη τιμή λογικού χρόνου στο λεξιλόγιο LC, τότε προστίθεται στην λίστα deliverable για να προστεθεί στην λίστα *self.delivered.*

Μετά τον βρόγχο, τα στοιχεία ταξινομούνται με βάση τον χρόνο cserver του server και σε περίπτωση ισοδυναμίας της τιμής, με το όνομα του αποστολέα. Αυτό γίνεται εφικτό με μια προσαρμοσμένη συνάρτηση sort(list1:list,key1:int,key2:int,reverse:bool) όπου list1 η λίστα προς ταξινόμηση , το key1 είναι ο δείκτης στην πλειάδα που θα ελέγχουμε σαν πρώτο

παράγοντα ταξινόμησης, το key2 είναι ο δεύτερος δείκτης στην πλειάδα που θα ελέγχουμε και το reverse για καθορισμό αύξουσας ή φθίνουσας (True= φθίνουσα, False=αύξουσα). Η ταξινομημένη λίστα γίνεται iterate για να προστεθούν στην λίστα self.delivered. Για αίτημα 'APPEND' τρέχει η συνάρτηση uponDeliverAPPEND(c,r,p,s) και για αίτημα 'GET' τρέχει η συνάρτηση uponDeliverGET(c,p). Υπάρχει ο πλήρης κώδικας της ΥΑΠ στο Παράρτημα Α

4.5.2 Προγραμματισμός Εξυπηρετητή με Ατομική Συνέπεια

Για τον εξυπηρετητή δημιουργήθηκε μια κλάση Atomic_Server που υιοθετεί την κλάση process και την κλάση AtomicBroadcastService η οποία είναι το ΥΑΠ. Για το setup της κλάσης ορίσαμε τις εξής μεταβλητές:

- Κενή λίστα self.received όπου θα προστείθονται τα αιτήματα που γίνονται Broadcast.
- Κενή λίστα S για τα αντικείμενα.
- Κενές λίστες pending και get_pending για τα αιτήματα που περιμένουν ανταπόκριση.
- Μεταβλητή self.shutdown=False που καθορίζει αν ο εξυπηρετητής είναι ενεργός.
- Μεταβλητή self.faulty_servers=0 που μετρά τους εσφαλμένους εξυπηρετητές.

Οι παραμέτροι που δέχεται η συνάρτηση setup είναι το sid, το οποίο είναι η ταυτότητα του εξυπηρετητή, clients και servers τα οποία είναι τα σύνολα των διεργασιών για τους πελάτες και τους εξυπηρετητές αντίστοιχα, η προκαθορισμένη τιμή f για τον μέγιστο αριθμό εσφαλμένων εξυπηρετητών, η διεργασία main στην οποία θα σταλούν από τους εξυπηρετητές το σύνολο S και τον συνολικό αριθμό αναμενόμενων αντικειμένων που θα αποθηκευτούν στο σύνολο S.

Εκτός από την συνάρτηση setup δημιουργήθηκε και η συνάρτηση run, την οποία αντικαθιστά την in-build συνάρτηση run της κλάσης process και είναι οι πράξεις που θα εκτελεστούν μετά από την συνάρτηση setup. Στην συνάρτηση αυτή προστέθηκαν όλες οι αναγκαίες ενέργειες για να μπορέσει το πρόγραμμα να τερματίσει όλες τις διεργασίες με την λήξη όλων των παραλαμβάνοντων μηνυμάτων από τις διεργασίες πελάτες. Επίσης για περίπτωσεις που το πρόγραμμα καθυστερήσει λόγω αναπάντεχων γεγονότων δίνεται χρόνος ίσος με C*s όπου C και s ο αριθμός των client και server αντίστοιχα. Στο τέλος της συνάρτησης ο κάθε εξυπηρετητής στέλνει το σύνολο self.S στην διεργασία Main.

```
def receive(msg=(c,'GET'),from_=p):
    if (self.shutdown):
        output('Ignoring get request')
    else:
        rand=random.random()
        if (self.chance>rand and self.faulty_servers<len(self.allservers)-self.f):
            send("faulty server",to=self.servers)
            self.faulty_servers+=1
            #output('Unexpected shut down')
            self.shutdown=True
        else:
            self.get_pending.append((c,p))
            ABroadcastGET(c,p)
def receive(msg=(c,'APPEND',r),from_=p):
    if (self.shutdown):
        output('Ignoring append request')
    else:
        rand=random.random()
        if (self.chance>rand and self.faulty_servers<len(self.allservers)-self.f):
            send("faulty server",to=self.servers)
            self.faulty_servers+=1
            #output('Unexpected shut down!')
            self.shutdown=True
        else:
            self.pending.append((c,r,p))
            ABroadcast(c,r,p)
```

Σχήμα 4.5.3: Οι συναρτήσεις παραλαβής αιτημάτων από τους πελάτες. Αρχίζει από έλεγχο τερματισμού, ακολουθώντας από πιθανότητα κατάρρευσης και τέλος η ενέργεια Broadcast.

Για την ακρόαση μηνυμάτων του process έχουμε τις ακόλουθες συναρτήσεις :

1. def receive(msg=(c,'GET'), from_=p):
2. def receive(msg=(c, 'APPEND', r), from_=p):

οι οποίες είναι συναρτήσεις που περιμένουν για μηνύματα τύπου GET και APPEND στην μορφή που αναγράφεται στην παράμετρο msg (Σχήμα 4.5.3). Η μεταβλητή p που βρίσκετε στην παράμετρο from_ παίρνει την τιμή της διεργασίας αποστολέα, και αυτό δίνει την δυνατότητα να στείλει κάποιος πίσω μήνυμα σε αυτόν. Για να μπορεί ένας εξυπηρετητής να επεξεργαστεί τα μηνύματα πρέπει η self.shutdown να ισούται με False, αλλιώς αγνοεί το μήνυμα. Οι ενέργειες που ακολουθεί το πρόγραμμα είναι οι ίδιες με αυτές που αναλύσαμε στην υποενότητα 3.3.2 μαζί με μια επιπλέον λειτουργία που μπορεί να τερματίσει τον εξυπηρετητή. Πριν από κάθε αναμετάδοση του παραλαμβάνοντος μηνύματος με την βοήθεια της συνάρτησης random της βιβλιοθήκης random, παράγεται ένας δεκαδικός αριθμός rand από το 0 μέχρι το 1 συμπεριλαμβανομένων τα άκρα. Αν ο αριθμός αυτός είναι πιο μικρός από την τιμή της μεταβλητής crash, τότε η διεργασία μπαίνει σε ανενεργή κατάσταση, δίνοντας της self.shutdown την τιμή True, αλλίως συνεχίζει κανονικά τις ενέργειές της. Οι υπόλοιπες ενέργειές της είναι να προσθέσει τη παράμετρο msg και την from_ στην μεταβλητή self.get_pending για αναμονή ανταπόκρισης και να αναμεταδώσει τις παραμέτρους με την βοήθεια της συνάρτησης ABroadcast/ABroadcastGET.

```
def uponDeliverGET(c,p):
    if (not self.shutdown):
        if ((c,p) in self.get_pending):
            send((c,'GETRES',self.S),to=p)
            self.get_pending.remove((c,p))

def uponDeliverAPPEND(c,r,p,s):
    if (not self.shutdown):
        if (not((c,r,str(p),str(s)) in self.S)):
            self.S.append((c,r,str(p),str(s)))
            #output('len(self.S):',len(self.S))
            for m in self.pending:
                if ((m[0],r,m[2]) in self.pending):
                    send((m[0],'APPENDRES','ACK'),to=m[2])
                    self.pending.remove((m[0],r,m[2]))
                    break;
```

Σχήμα 4.5.4: Οι ενέργειες μετά την παράδοση. Έλεγχος για ανταπόκριση του πελάτη.

Υπάρχουν ακόμη δύο συναρτήσεις στην διεργασία εξυπηρετητής:

1. def uponDeliverGET (c, p)
2. def uponDeliverAPPEND (c, r, p, s):

Οι συναρτήσεις αυτές είναι οι ενέργειες που γίνονται μετά την παράδοση του μηνύματος (Σχήμα 4.5.4).

Η συνάρτηση 1 τρέχει μετά την παραλαβή μηνύματος τύπου GET και εφόσον δεν τερματίστηκε η διεργασία (self.shutdown=False). Είναι υπεύθυνη να ανταποκριθεί στο αίτημα του αποστολέα, δηλαδή να του στείλει μήνυμα (c,'GETRES',self.S) και να αφαιρέσει το μήνυμα από την self.get_pending εφόσον υπάρχει στην λιστα αναμονής.

Η συνάρτηση 2 τρέχει μετά την παραλαβή μηνύματος τύπου APPEND και εφόσον δεν τερματίστηκε η διεργασία. Αρχικά ελέγχει εάν το μήνυμα υπάρχει στο σύνολο αντικειμένων self.S. Αν δεν υπάρχει, τότε το προσθέτει και μετά ελέγχει αν υπάρχει πλειάδα (c,r,π) στην λίστα self.pending. Τέλος αν υπάρχει το αφαιρεί από την λίστα και στέλνει ανταπόκριση στον αποστολέα πελάτη, δηλαδή του στέλνει μήνυμα (c,'APPENDRES','ACK') και αφαιρεί το αίτημα από την λίστα self.pending εφόσον υπάρχει στη λίστα. Ο πλήρης κώδικας βρίσκεται στο Παράρτημα Β.

4.6 Προγραμματισμός Διεργασιών Πελάτη

Οι διεργασίες πελάτη χρειάζονται για να στέλνουν αιτήματα στις διεργασίες εξυπηρετητή, ώστε να προσθέτονται στο Αντικείμενο Κατάστιχου (στην περίπτωσή μας στη

λίστα πλειάδων self.S). Το όνομα που δόθηκε στην κλάση είναι Sender και η υλοποίησή της ήταν ιδιαίτερα απλή.

Αρχικά η κλάση setup, παίρνει 4 arguments: (α) sid για αρίθμηση της διεργασίας, (β) allservers είναι όλοι οι διεργασίες εξυπηρετητές, (γ) servers είναι οι εξυπηρετητές που θα στέλνει ο πελάτης τα αιτήματά του και (δ) commands είναι τα αιτήματα που θα στείλει ο πελάτης. Τα commands είναι ειδικά δομημένα, ώστε να μπορεί η διαπροσωπεία να τα στείλει ως arguments στο πρόγραμμα στην γραμμή εντολών. Έστω οι εντολές του πρώτου πελατη είναι μια αίτηση APPEND ενός αντικειμένου r και μια αίτηση GET. Τότε στις παραμέτρους της γραμμής εντολών, η εντολή θα έχει την εξής δομή: *APPEND::r,GET::none.*

Αν η εντολή APPEND προορίζεται για τον πελάτη 1 και η εντολή GET προορίζεται για τον πελάτη 2, τότε θα έχει την εξής δομή: APPEND::r GET:none. Ο διαχωρισμός γίνεται στην συνάρτηση main του αρχείου της προσομοίωσης.

Στην συνάρτηση setup ορίζονται οι εξής private μεταβλητές:

- self.LC: το λογικό ρολόι του πελάτη. Με αυτό μπορούν να θέτουν τα μηνύματα να είναι μοναδικά μεταξύ τους, ανεξαρτήτως περιεχομένου.
- self.servers: Το σύνολο των εξυπηρετητών που θα απευθύνεται ο πελάτης.
- self.commands: Τα αιτήματα που θα στείλει των εξυπηρετητών στο σύνολο self.servers.
- self.wait: Λίστα αναμονής ανταπόκρισης από εξυπηρετητές. Σε συνδυασμό με συνάρτηση παραλαβής ανταπόκρισης, ο ρόλος της μεταβλητής είναι για να αναμένει μέχρι ανταπόκρισης από τον εξυπηρετητή για να συνεχίσει με τα αιτήματα (False για να περιμένει, True για να προχωρήσει).

Αφού εκτελεστεί η συνάρτηση setup ακολουθεί η συνάρτηση run, όπου θα εκτελέσουν τα αιτήματα. Αρχικά περιμένουν όλους τους εξυπηρετητές να στείλουν το μήνυμα (‘online’,self) για να δείξουν ότι είναι έτοιμοι να παραλάβουν μηνύματα. Εφόσον παραλάβουν τα μηνύματα, εκτελείται ένα for-loop ανάμεσα στα commands. Αν η εντολή είναι APPEND::r τότε εκτελείται η συνάρτηση appendItem(r), ενώ αν είναι GET::none εκτελείται η συνάρτηση get(). Μόλις τελειώσουν τα αιτήματα περιμένουν μήνυμα από όλους τους εξυπηρετητές για να τερματιστούν.

Η appendItem(r), όπως υποδηλώνει και το όνομά της, είναι η συνάρτηση όπου αποστέλνει ο πελάτης το αίτημα append στους εξυπηρετητές του συνόλου servers. Η

```
counter=0
while(counter<3):
    for i in range(0,length):
        start_time=time.time()
        try:
            send((c,'GET'),to=self.servers[i])
            #output('Sending ','(',(c,'GET'),') to server ',self.servers[i])
            if (await(self.wait[c])):
                exec_time=time.time()-start_time
                writeToExcel(exec_time)
                counter=3
                break
            elif timeout(1*len(allservers)):
                output(str(self.servers[i])+' is not responding.Switching to another server')
        except Exception as e:
            output("Couldn't send to "+str(self.servers[i]))
    counter+=1
if (self.wait[c]!=True):

    #output("Couldn't send the message to servers")
    send(("-1",self),to=main)
```

Σχήμα 4.6.1: Απόσπασμα της συνάρτησης αποστολής αιτήματος GET. Δοκιμάζει να στείλει στους εξυπηρετητές αίτημα και να παραλάβει ανταπόκριση μέχρι counter=3.

υλοποίηση έγινε με βάση τον αλγόριθμο που εξηγήσαμε πιο πάνω. Αρχικά ανεβάζει το λογικό ρολόι c του κατά 1, η self.wait[c] γίνεται False και αρχικοποιούμε την μεταβλητή counter=0. Στη συνέχεια αρχίζει ένα for-loop φωλιασμένο μέσα σε ένα while loop (Σχήμα 4.6.1). Στο for-loop γίνονται iterate οι εξυπηρετητές στο σύνολο self.servers. Για κάθε server, ο πελάτης του στέλνει το αίτημα append με την εντολή *send((c,'GET'),to=self.servers[i]),* όπου το *(c,'APPEND',r)* το μήνυμα και το *self.servers[i]* ο εξυπηρετητής που θα παραλάβει το μήνυμα.

Μετά ο πελάτης περιμένει το self.wait[c] να γίνει True, δηλαδή περιμένει για ανταπόκριση από τον εξυπηρετητή. Σε αυτή τη φάση μετά από πειράματα προστέθηκε ένα timeout σε δευτερόλεπτα ίσο με 0.5*L, όπου L ο αριθμός των εξυπηρετητών που θα απευθύνεται ο πελάτης. Αν παραλάβει ανταπόκριση πριν τον καθορισμένο χρόνο, τότε οι βρόγχοι σταματούν, αλλιώς στέλνει στον επόμενο εξυπηρετητή. Αν δεν παραλάβει ανταπόκριση από κανέναν εξυπηρετητή, τότε το counter αυξάνεται κατά 1 και επαναλαμβάνεται το for loop. Αν το counter γίνει 3 τότε σταματά το βρόγχος και δεν αποστέλνεται άλλο αίτημα. Παρόμοια δουλεύει και η συνάρτηση get(), όμως ο πελάτης στέλνει (c,'GET') και για ανταπόκριση παίρνει το μήνυμα (c,'GETRES',V) όπου το V το σύνολο των εγγραφών.

Προηγουμένος αναφέρθηκε πως ο πελάτης περιμένει το self.wait[c] να γίνει True.

```
def receive(msg=(c,'GETRES',V),from_=m):
    if (self.wait[c]==False):
        output('I got '+str(len(V))+' items from ',str(m),'get: ',V)
        self.wait[c]=True
def receive(msg=(c,'APPENDRES',r),from_=m):
    if (self.wait[c]==False):
        #output(r)
        self.wait[c]=True
```

Σχήμα 4.6.2: Συναρτήσεις παραλαβής ανταπόκρισης από εξυπηρετητή για διεργασία πελάτη. Έλεγχος αν πήρε ήδη ανταπόκριση για να παρουσιάσει τα αποτελέσματα.

Αυτό γίνεται με την βοήθεια των δύο ακόλουθων συναρτήσεων παραλαβής μηνυμάτων (Σχήμα 4.6.2):

1. def receive(msg=(c,'GETRES',V), from_=s).
2. def receive(msg=(c,'APPENDRES','ACK'), from_=s).

Η πρώτη συνάρτηση είναι για παραλαβή ανταπόκρισης από έναν εξυπηρετητή για το αιτήμα 'GET' και η δεύτερη για παραλαβή ανταπόκρισης από έναν εξυπηρετητή για το αιτήμα 'APPEND'. Με την παραλαβή αυτών των μηνυμάτων ελέγχεται αν το self.wait[c] ισούται με True, όπου c ο χρόνος του αιτήματος. Αν ισχύει, τότε εμφανίζεται η ανταπόκριση του εξυπηρετητή στην γραμμή εντολών και το self.wait[c] γίνεται True. Η συνάρτηση δομήθηκε με τέτοιο τρόπο, ώστε να αγνοεί τις υπόλοιπες ανταποκρίσεις. Αν το self.wait[c] είναι ήδη αληθές, σημαίνει πως έλαβε ανταπόκριση, οπότε η διεργασία δεν θα συνεχίσει με την παρουσίαση των αποτελεσμάτων της ανταπόκρισης του εξυπηρετητή.

Κεφάλαιο 5

Διαπροσωπεία

5.1 Εισαγωγή

Σε αυτό το κεφάλαιο θα παρουσιάσουμε τις λειτουργίες που προστέθηκαν στην διαπροσωπεία και θα προβάλουμε τις εξόδους και τις εισόδους της. Στο τέλος θα αναφέρουμε τους περιορισμούς που εμφανίστηκαν κατά την υλοποίησή της.

5.2 Περιγραφή

Ο στόχος της διαπροσωπείας είναι η εύκολη χρήση της προσομοίωσης της Ατομικής Συνέπειας και της Τελικής Συνέπειας και η εμφάνιση των περιεχομένων του Αντικείμενου Κατάστιχου. Δίνεται αρχικά η επιλογή του αριθμού των πελατών και των εξυπηρετητών. Πατώντας την επιλογή Run ανοίγει το Παράθυρο Διαμόρφωσης, όπου καθορίζονται οι ενέργειες των πελατών και την πιθανότητα να απενεργοποιηθεί ένας εξυπηρετητής με κάθε πολυεκπομπή ενός μηνύματος. Τρέχοντας το πρόγραμμα τρέχει η υλοποίηση που αναλύθηκε στο Κεφάλαιο 4 και μετά εμφανίζει το Παράθυρο Αποτελεσμάτων, όπου εμφανίζονται τα αποθηκευμένα μηνύματα των επεξεργαστών σε γραφική μορφή.

5.3 Παράθυρα προς Υλοποίηση

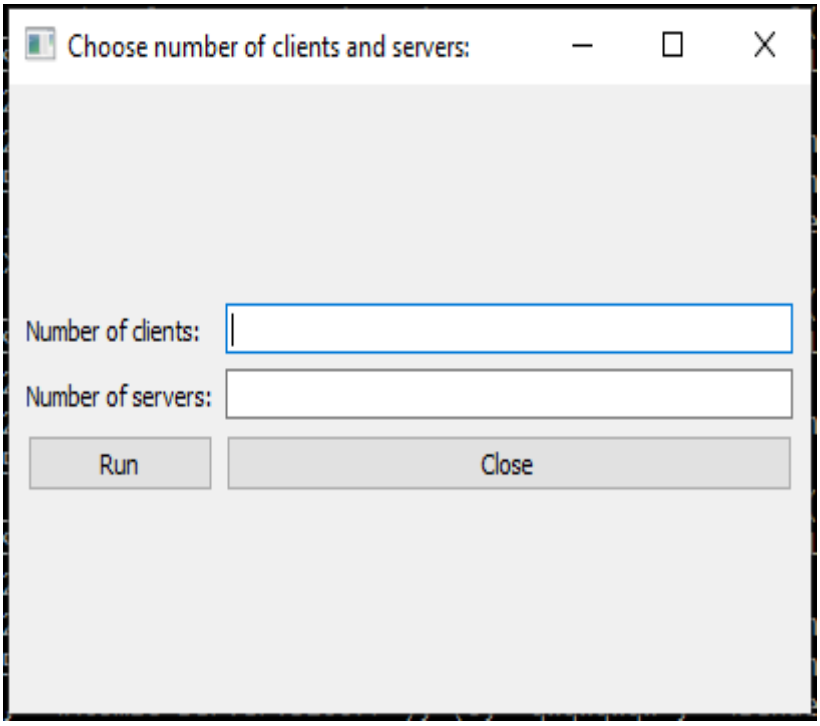
Η διαπροσωπεία που υλοποιήθηκε με την βιβλιοθήκη της Python PyQt5 αποτελείται από τρία βασικά παράθυρα: (α) το Αρχικό Παράθυρο (First Window), (β) το παράθυρο διαμόρφωσης (Configuration Window) και (γ) το παράθυρο αποτελεσμάτων (Result Window). Το Αρχικό Παράθυρο είναι το πρώτο παράθυρο που εμφανίζεται και μπορεί να τρέξει αυτόνομα από την γραμμή εντολής, όμως το Παράθυρο Διαμόρφωσης και το Παράθυρο Αποτελεσμάτων χρειάζονται τα κατάλληλα δεδομένα ως παραμέτρους για να λειτουργήσουν ορθά. Στα επόμενα υποκεφάλαια αναλύονται λεπτομερώς τα παράθυρα που αναφέρθηκαν.

Για να δημιουργηθεί η διαπροσωπεία πρέπει πρώτα να δημιουργηθεί ένα QApplication() στην συνάρτηση main() του αρχείου για να εμφανίσει την διαπροσωπεία. Το QApplication δέχεται παραμέτρους από την γραμμή εντολών και τους προσθέτει στην κλάση της διαπροσωπείας, όπου θα οριστούν τα αντικείμενα με τις λειτουργίες τους. Στη συνέχεια στην main δημιουργείται ένα αντικείμενο της κλάσης της διαπροσωπείας, το οποίο για να μείνει ανοικτό πρέπει να ακολουθήσει η εντολή *sys.exit(app.exec_()),* όπου το sys.exit() τερματίζει το πρόγραμμα και το app.exec() ενεργοποιεί την εφαρμογή app. Όταν κλείσει η εφαρμογή, τότε η app.exec() επιστρέφει False στο sys.exit() και το πρόγραμμα τερματίζεται.

Για την δημιουργία της κλάσης διαπροσωπείας, χρησιμοποιήθηκε ως ένα βαθμό το πρόγραμμα QtDesigner που αναφέραμε στο κεφάλαιο 2, δηλαδή προσθέσαμε τα αντικείμενα (Widgets) στο παράθυρο με Drag and Drop. Για την δημιουργία των ενεργειών των κουμπιών και την αλληλοσύνδεση των παραθύρων, ορίστηκαν οι απαραίτητες συναρτήσεις μέσα στην κλάση, όπου δικτυώνονται με την συνάρτηση .connect(). Τα αντικείμενα δηλώνονται στην κλάση __init__(self), η οποία τρέχει πρώτη στην κλάση. Τέλος, όταν όλα τα αντικείμενα δηλωθούν στην κλάση, τρέχει την εντολή self.show() όπου εμφανίζει την διαπροσωπεία.

5.3.1 Αρχικό Παράθυρο

Η υλοποίησή του Αρχικό Παράθυρο είναι σχετικά απλή (Σχήμα 5.3.1).. Η διαπροσωπεία αποτελείται από 2 ζευγάρια καρτέλας QLabel και κουτιού εισαγωγής κειμένου QLineEdit στοιχισμένα το ένα κάτω από το άλλο για εισδοχή του αριθμού clients και servers, και 2 κουμπιά Run και Quit. Το κουμπί Run τρέχει το Παράθυρο Διαμόρφωσης με βάση τον

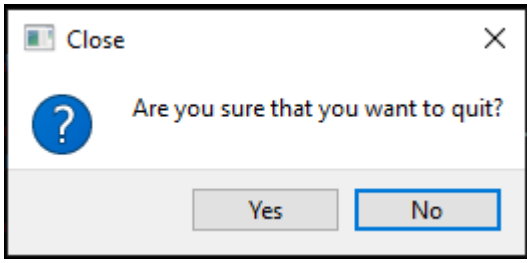


Σχήμα 5.3.1: Το Αρχικό Παράθυρο του προγράμματος. Ορίζεται ο αριθμός των πελατών και των εξυπηρετητών.

αριθμό των Clients και των Servers που εισαχθηκαν στα QLineEdit και το κουμπί Quit κλείνει το παράθυρο, αφου βγαλει πρώτα ένα Παράθυρο Διαλόγου που ρωτά αν όντως θέλει ο χρήστης να κλείσει το πρόγραμμα (Σχήμα 5.3.2).

Για την δημιουργία του παράθυρου ορίστηκε το πλάτος του στα 400 pixels και το ύψος του στα 250. Στη συνέχεια ορίστηκε ένα QGridLayout() με όνομα gridLayoutOutside όπου δίνει την δυνατότητα στοίχισης των Widgets σε οριζόντιο και σε κάθετο επίπεδο. Μέσα στο QGridLayout() δημιουργήθηκε ακόμη ένα QGridLayout() με όνομα gridLayout, όπου θα περιέχει τα αντικείμενα που δηλώθηκαν πιο πάνω. Ο λόγος που υπάρχουν 2 φωλιασμένα GridLayout είναι για να γίνει η στοίχιση μεταξύ του gridLayoutOutside και του παραθύρου, αλλά ταυτόχρονα να γίνει και η σωστή στοίχιση των αντικειμένων μέσα στο εσωτερικό QGridLayout.

Για την εμφάνιση του Παραθύρου Dialog πριν από τον τερματισμό μιας εφαρμογής ορίστηκε η συνάρτηση closeEvent(self,event) η οποία είναι συνάρτηση που είναι σχεδιασμένη να τρέχει πριν τον τερματισμό. Σε αυτή την συνάρτηση ελέγχεται αν προσπαθεί ο χρήστης να κλείσει την εφαρμογή ή να ανοίξει το Παράθυρο Διαμόρφωσης. Αν ο χρήστης προσπαθεί να τερματίσει την διαπροσωπεία με το κουμπί εξόδου, τότε δημιουργείται ένας διάλογος με την ερώτηση “Are you sure do you want to quit?” (Σχήμα 5.3.2).



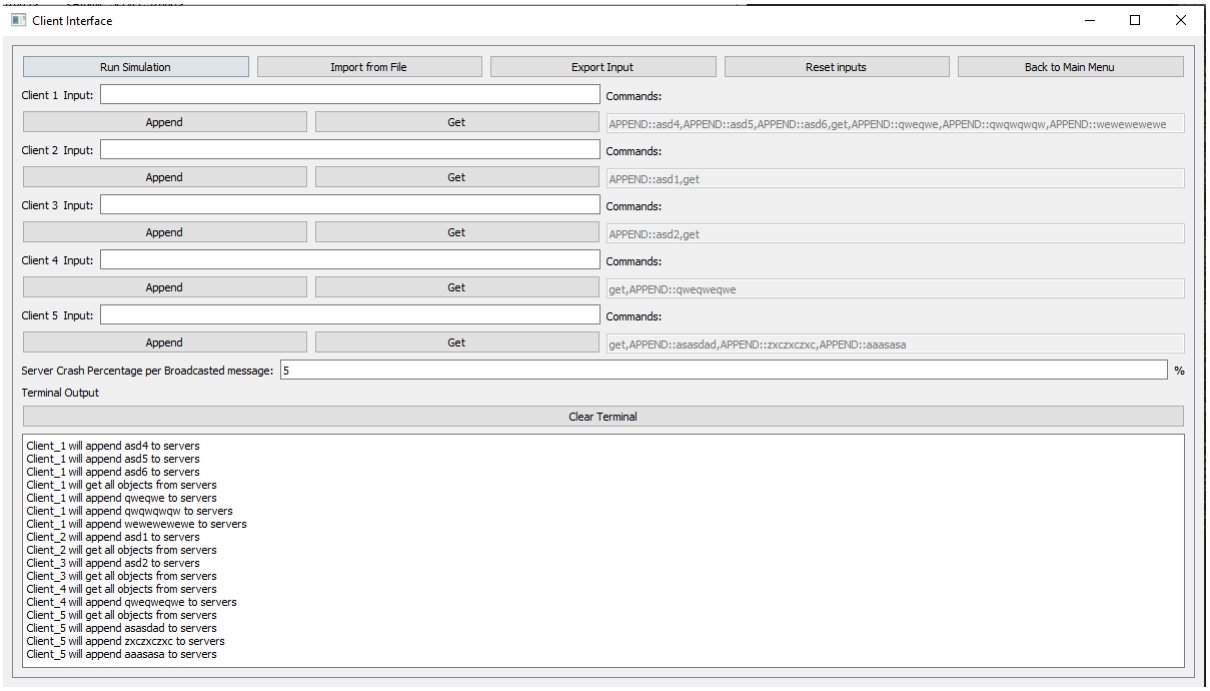
Σχήμα 5.3.2: Παράθυρο επιβεβαίωσης τερματισμού εφαρμογής.

Αν ο χρήστης επιλέξει «Yes», τότε ο τερματισμός ολοκληρώνεται με την εντολή event.accept(), αλλιώς η εντολή αγνοείται με την εντολή event.ignore(). Αν ο χρήστης προσπαθήσει να ανοίξει το Παράθυρο Διαμόρφωσης με το κουμπί Run τότε η συνάρτηση closeEvent μένει αδρανής

5.3.2 Παράθυρο Διαμόρφωσης

Το Παράθυρο Διαμόρφωσης είναι το κύριως παράθυρο της διαπροσωπείας και είναι υπεύθυνο να ορίσει τις παραμέτρους που χρειάζονται για να λειτουργήσει η προσομοίωση. Αποτελείται από 5 κύρια κουμπιά QButton: (α) Run Simulation, (β) Import from file (β) Export Input (δ) Reset Input (ε) Back to Main Menu. Στη συνέχεια για κάθε πελάτη εμφανίζεται χώρος που αποτελείται από ένα QLineEdit για τα μηνύματα που θα προστέθονται στο Αντικείμενο Κατάστιχου, 2 κουμπιά που σημάνουν τις ενέργειες APPEND και GET, μια καρτέλα QLabel που γράφει Commands και ένα QLineEdit το οποίο είναι μόνο για διάβασμα και δείχνει τις εντολές του πελάτη. Ακολουθεί μια γραμμή με ένα QLabel που γράφει “Server Crash Percentage per Broadcasted message” και ένα πεδίο QLineEdit. Τέλος υπάρχει ένα QTextBox το οποίο χρησιμοποιείται ως αντικατάστατο της γραμμής εντολής και γράφει ολες τις ενέργειες που πάρθηκαν στο παράθυρο διαμόρφωσης, μαζί με ένα κουμπί “Clear Terminal” για να αδειάζει το Terminal (Σχήμα 5.3.3).

Αρχίζοντας από τα 5 βασικά κουμπιά, το Run Simulation είναι υπεύθυνο να τρέχει την προσομοίωση του μοντέλου Πελάτη-Εξυπηρετητή με εξυπηρετητές που χρησιμοποιούν Ατομική Συνέπεια ή Τελική Συνέπεια. Λόγω περιορισμού που προαναφέρθηκε, χρησιμοποιήθηκε η βιβλιοθήκη subprocess για την χρήση του προγράμματος προσομοίωσης, όπου δίνει τη δυνατότητα να τρέξει ένα πρόγραμμα με την συνάρτηση Popen() σε υποδιεργασία της διαπροσωπείας. Για να τρέξει την προσομοίωση, χρειάζεται να δομηθούν τα arguments που πρέπει να περάστουν στην Popen(). Η δομή της εντολή στην Popen είναι η εξής:

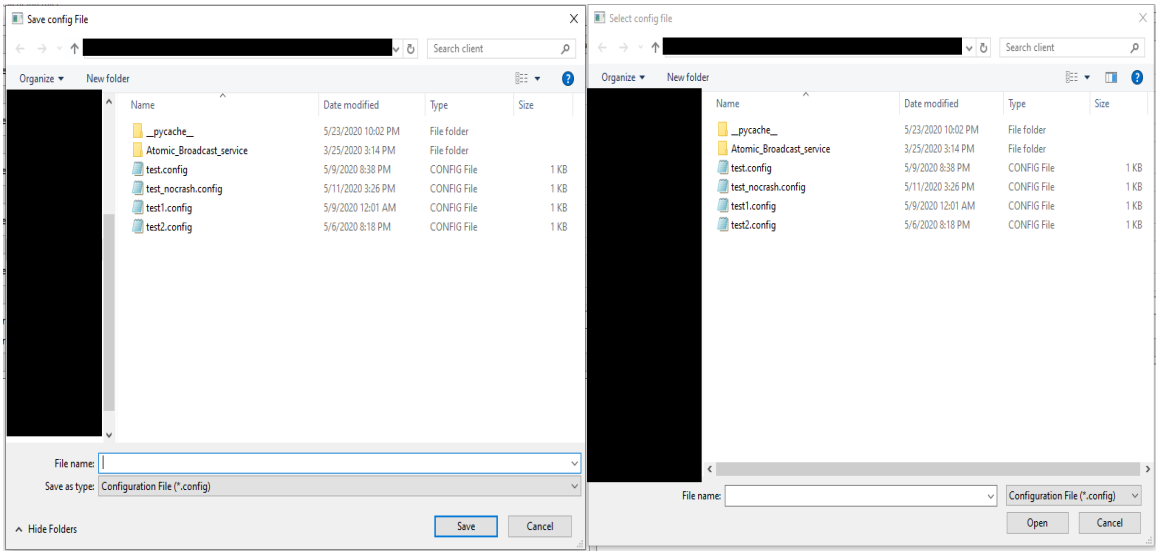


Σχήμα 5.3.3: Παράθυρο Διαμόρφωσης με 5 ορισμένους πελάτες.

python -m da atomic_consistency_crash_tolerant_UI.da [servers_num] [clients_num] [crash_chance] [commands]

Το servers_num και το clients_num αντιπροσωπεύει τον αριθμό των εξυπηρετητών και τον αριθμό των πελατών αντίστοιχα, το crash_chance την πιθανότητα απρόσμενου τερματισμού του εξυπηρετητή μετά από κάθε Broadcast αιτήματος από πελάτη και το commands τις εντολές των πελατών. Το commands δημιουργείται από μιά δυσδιάστατη λίστα με όνομα self.clientInput, όπου κρατεί όλα τα αιτήματα που ο κάθε πελάτης θα στείλει. Το κάθε αίτημα έχει την μορφή action::message, με το action να είναι APPEND ή GET και το message να είναι το μήνυμα που θα σταλεί με το αίτημα (στο GET το message είναι πάντα null). Για την δομή της εντολής, τα αιτήματα χωρίζονται με τον χαρακτήρα ‘,’ και κάθε πελάτης χωρίζεται με κενό,όπου στο αρχείο της προσομοίωσης χωρίζονται πίσω στην πρώτη τους μορφή.

Για την ευκολία επαναχρησιμοποίησης του προγράμματος δημιουργήθηκαν τα κουμπιά import file και export input, όπου φορτώνουν και αποθηκεύουν τις εντολές που ορίστηκαν στο πρόγραμμα αντίστοιχα. Το import file ανοίγει ένα παράθυρο για επιλογή του αρχείου, και δέχεται μόνο αρχεία τύπου .config και το export ανοίγει ένα παράθυρο για αποθήκευση των εντολών των πελατών που εμφανίζονται στην διαπροσωπεία (Σχήμα 5.3.4).



Σχήμα 5.3.4: Παράθυρο import .config και παράθυρο export .config.

Τα αρχεία .config έχουν την εξής δομή για 10 Clients:

```
crash::5
APPEND::1,APPEND::2,APPEND::3
APPEND::1,APPEND::2,APPEND::3
APPEND::1,APPEND::2,APPEND::3
APPEND::1,APPEND::2,APPEND::3
APPEND::1,APPEND::2,APPEND::3
null
null
null
null
null
```

Αρχίζει με δήλωση της πιθανότητας τερματισμού του επεξεργαστή για κάθε broadcast ενός αιτήματος πελάτη και συνεχίζει με τις εντολές των πελατών. Κάθε γραμμή αντιπροσωπεύει τα αιτήματα ενός πελάτη χωρισμένα με τον χαρακτήρα ‘,’. Όπως αναφέρθηκε και πιο πάνω, η δομή της κάθε εντολής είναι action::message. Επιπρόσθετα για τους πελάτες που δεν θα πράξουν τίποτα, απλά ορίζουμε την λέξη null. Είναι σημαντικό να υποθεί πως σε περίπτωση που οι γραμμές είναι πιο λίγες από τον αριθμό των πελατών, τότε οι υπόλοιποι λαμβάνουν την λέξη null. Επίσης στην περίπτωση που οι γραμμές είναι περισσότερες από τους δεδομένους πελάτες, τότε τα υπόλοιπα αγνοούνται. Ακόμη αν η πρώτη γραμμή crash παραληφθεί, τότε το πεδίο του θα πάρει την τιμή 0. Για όλες τις προαναφερόμενες περιπτώσεις εμφανίζονται μηνύματα ενημέρωσης στο terminal της διαπροσωπείας. Τέλος αν παραβιαστεί με οποιοδήποτε άλλο τρόπο η δομή του αρχείου τότε θα εμφανίσει μήνυμα στο terminal.

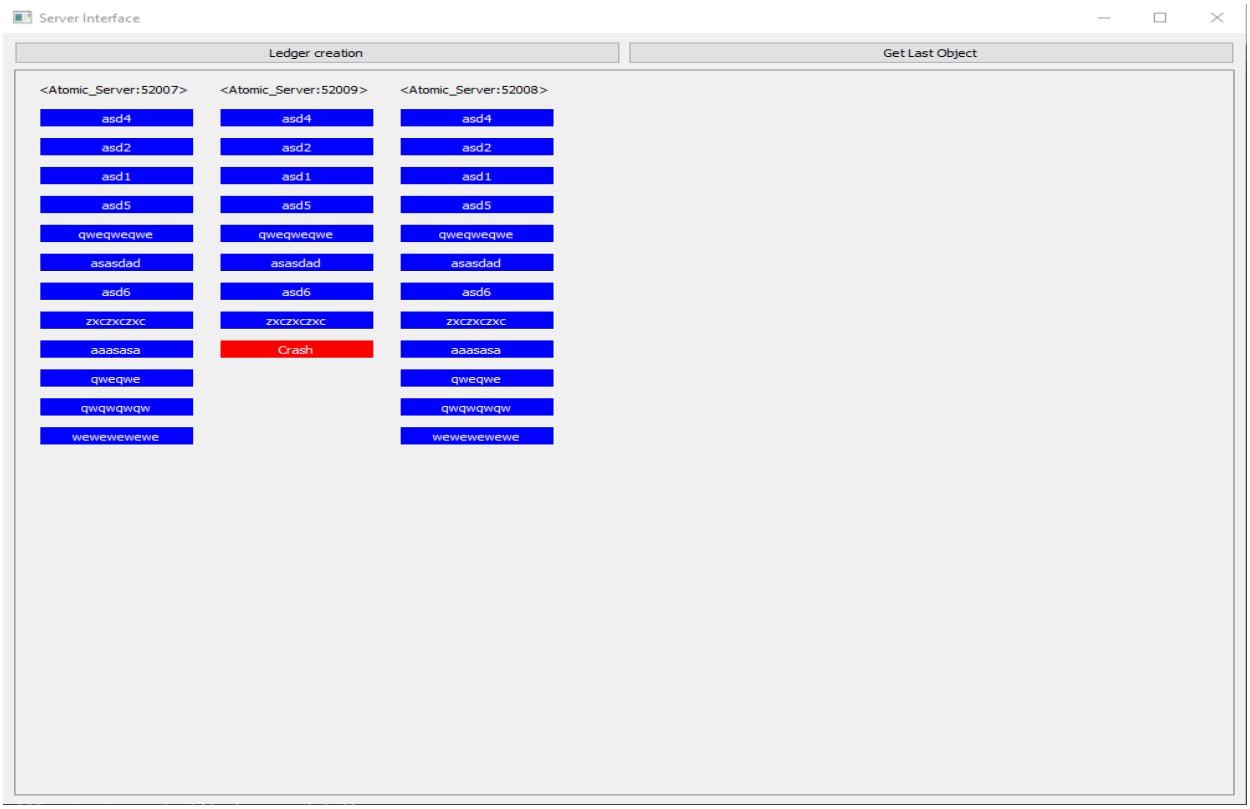
Τα τελευταία δύο βασικά κουμπιά Reset Input και Back to Main Menu κάνουν αυτό που γράφουν. Το Reset Input μηδενίζει το παράθυρο, διαγράφοντας όλες τις αλλαγές που έγιναν μετά που εμφανίστηκε (το παράθυρο) και το Back to Main Menu επιστρέφει τον χρήστη στο Αρχικό Παράθυρο.

Συνεχίζοντας θα αναλυθούν τα Widgets για κάθε ένα από τους πελάτες. Για κάθε πελάτη που δηλώθηκε στο Αρχικό Παράθυρο, δημιουργείται μια πλειάδα από Widgets που αναφερθήκαν πιο πάνω. Επειδή τα αντικείμενα δημιουργήθηκαν σε ένα QScrollArea και επειδή στο QScrollArea υπάρχει ένα gridLayout, τότε αν δηλωθεί μεγάλος αριθμός πελατών ώστε το παράθυρο να μην μπορεί να τα εμφανίσει στο παράθυρο, τότε εμφανίζεται ένα scroll bar. Για κάθε πελάτη υπάρχει 2 κουμπιά 'APPEND' και 'GET', τα οποία είναι οι επιλογές αιτημάτων που ο πελάτης έχει την δυνατότητα να στείλει. Το APPEND παίρνει το περιεχόμενο του QLineEdit που υπάρχει πιο πάνω, προσθέτει το αίτημα στην μορφή APPEND::message στην δυσδιάστατη λίστα self.clientInput και καθαρίζει το περιεχόμενο από το πεδίο. Αν ο χρήστης δεν καθορίσει input, τότε το APPEND αίτημα αγνοείται. Το κουμπί GET προσθέτει το αίτημα στην λίστα self.clientInput στην μορφή GET::null. Για κάθε αίτημα που ορίζεται σε ένα πελάτη, προστίθεται και στο πεδίο commands διαχωρισμένο με τον χαρακτήρα ',' (βλέπε Σχήμα 5.3.2.1).

Κάτω από τα πεδία των πελατών ορίζεται το πεδίο QLineEdit crash percentage. Το πεδίο παίρνει τιμές από 0 μέχρι 100, και η τιμή συμβολίζει την πιθανότητα ανά 100 για έναν εξυπηρετητή να τερματιστεί πριν την κάθε εκτέλεση της ABroadcast. Αν δωθεί τιμή έξω από τα όρια που αναφέρθηκαν, τότε εμφανίζεται στο terminal το μήνυμα «Crash percentage must be between 0 and 100 %». Τέλος είναι το κουμπί που αδειάζει το QTextBox terminal.

5.3.3 Παράθυρο Αποτελεσμάτων

Το Παράθυρο Αποτελεσμάτων εμφανίζεται μετά την ολοκλήρωση της προσομοίωσης Πελάτη-Εξυπηρετητή και σκοπός του είναι να εμφανίσει γραφικά την δημιουργία του Αντικείμενο Κατάστιχου σε κάθε εξυπηρετητή. Στην κορυφή του υπάρχουν δύο κουμπιά, για επιπρόσθετες λειτουργίες (α) Ledger creation και (β) Get Last Object. Κάτω από τα κουμπιά υπάρχει το ολοκληρωμένο σχέδιο του Αντικείμενου για κάθε επεξεργαστή (βλέπε Σχήμα 5.3.5). Για κάθε μήνυμα στο Αντικείμενο αντιπροσωπεύεται με ένα μπλε ορθογώνιο. Αν ένας εξυπηρετητής τερματιστεί πριν να γράψει όλα τα αντικείμενα στο σύνολο, τότε εμφανίζεται στο σημείο αυτό ένα κόκκινο ορθογώνιο με όνομα Crash.



Σχήμα 5.3.5: Παράθυρο Αποτελεσμάτων που εμφανίζει τα Αντικείμενα Κατάστιχου των εξυπηρετητών.

Το κουμπί Ledger creation δείχνει με την βοήθεια των αντικειμένων QPropertyAnimation και QParallelAnimationGroup την δημιουργία των Ledgers από την αρχή. Για να πετύχει η δημιουργία του animation αρχικά για κάθε αντικείμενο δημιουργείται ένα ορθογώνιο 0x0 διαστάσεις σε συντεταγμένες στο παράθυρο x,y που ορίζονται ανάλογα με τον αριθμό του πελάτη/εξυπηρετητή και τον αριθμό του αντικειμένου στην λίστα. Στη συνέχεια δημιουργούνται τα animations για το κάθε μήνυμα στο Αντικείμενο Κατάστιχου και προστίθονται σε μιά λίστα από animations. Επιπρόσθετα το κάθε animation τρέχει με την σειρά που ορίστηκε στην λίστα.

Η κίνηση των αντικειμένων είναι με την εξής σειρά. Αρχικά το κουτί μένει στάσιμο κάτω από την ετικέτα του αποστολέα πελάτη για 500 ms. Στη συνέχεια το κουτί μετακινείται κάτω από την ετικέτα του εξυπηρετητή που αποφάσισε ο πελάτης να αποστείλει το αίτημα και τέλος μοιράζεται στους υπόλοιπους εξυπηρετητές που παρέλαβαν το αντικείμενο. Για τα πρώτα δύο animations δημιουργούνται δύο αντικείμενα QPropertyAnimation, για το τρίτο animation πρώτα δημιουργείται ένα QParallelAnimationGroup και μετά δημιουργούνται QPropertyAnimation για κάθε αντικείμενο που μεταφέρεται από τον εξυπηρετητή. Υπάρχει ακόμη το κουμπί Get Last object όπου δημιουργεί ένα Dialog Window με το τελευταίο μήνυμα στο Αντικείμενο Κατάστιχου.

5.4 Περιορισμοί

Χρησιμοποιώντας το πρόγραμμα PyQt5 παρουσιάστηκαν κάποιοι περιορισμοί στην υλοποίηση της διαπροσωπείας.

- Δεν υπήρχε τρόπος να εμφανίσω τα αποτελέσματα παράλληλα με την προσομοίωση, γιατί δεν υπήρχε τρόπος να δημιουργήσω μια κλάση διεργασίας και γραφικού περιβάλλοντος ταυτόχρονα.
- Δεν υπήρχε τρόπος να τρέξει η προσομοίωση του προγράμματος Πελάτη-Εξυπηρετητή μέσω του προγράμματος της διαπροσωπείας ώστε να μην μπορεί να τρέξει ανεξαρτήτως. Ο λόγος ήταν επειδή η main του συγκεκριμένου αρχείου δεν μπορεί να καλεστεί από άλλο αρχείο, ούτε να τρέξει συνάρτηση που να τρέχει την main.

Κεφάλαιο 6

Πειραματική Αξιολόγηση

6.1 Εισαγωγή

Εδώ παρουσιάζονται οι αλλαγές που έγιναν για την πειραματική αξιολόγηση και τα σενάρια που χρησιμοποιήθηκαν για αξιολόγηση. Στη συνέχεια παρουσιάζονται τα αποτελέσματα σε μορφή γραφική παράστασης, αφού γίνουν προβλέψεις για τα αποτελέσματα και τα σχολιάζουμε.

6.2 Προεργασία και Μεθοδολογία

Για την Προετοιμασία της Πειραματικής Αξιολόγησης χρειάστηκαν να γίνουν κάποιες αλλαγές, ώστε να απομονωθούν οι κύριες λειτουργίες για να μπορεί να μετρηθεί ο χρόνος εκτέλεσης τους. Αρχικά διαγράφουμε όλες τις εντολές output, επειδή η εμφάνιση μηνυμάτων στην γραμμή εντολών καθυστερεί αρκετά το πρόγραμμα. Τα output που κρατήθηκαν είναι το μήνυμα για άγνοια του μηνύματος όταν παραλαμβάνει αίτημα ο εξυπηρετητής από έναν αποστολέα και το μήνυμα αδυναμίας αποστολής μηνύματος του αποστολέα σε εξυπηρετητή από try-catch. Επίσης αφαιρέθηκε η δημιουργία της διαπροσωπείας για τον λόγω ότι θα τρέξει αρκετές φορές το πρόγραμμα. Τέλος αφαιρέθηκε το timeout που υπήρχε στην συνάρτηση run της διεργασίας εξυπηρετητή, ώστε να μετρήσουμε τον πραγματικό χρόνο ολοκλήρωσης των ενεργειών του.

```
import time

start_time=time.time()
for i in range(0,1000000):
    k=0
exec_time=time.time()-start_time
print(exec_time)
```

Σχήμα 6.1.1: Διαδικασία καταμέτρηση χρόνου εκτέλεσης για τυχαίο πρόγραμμα βρόγχου

Για την μέτρηση του χρόνου χρησιμοποιείται η συνάρτηση time της in-build βιβλιοθήκη time. Για την χρήση της προσθέτουμε την γραμμή *import time* και καλούμε την συνάρτηση *time.time()*, ώστε να παραλάβουμε τον χρόνο που δείχνει το ρολόι της μηχανής σε δευτερόλεπτα. Για την παραλαβή του χρόνου εκτέλεσης ενός προγράμματος, αρχικά τρέχουμε την εντολή start_time=time.time() στο σημείο που θέλουμε να αρχίσει η καταμέτρηση, όπου start_time η μεταβλητή του αρχικού χρόνου που δηλώνουμε και στο σημείο τερματισμού της καταμέτρησης την γραμμή exec_time=time.time()-start_time, όπου exec_time η μεταβλητή για τον χρόνο εκτέλεσης (παράδειγμα στο Σχήμα 6.1.1).

Συνεχίζοντας με την μεθοδολογία καταμέτρησης χρόνου στα Σενάρια Αξιολόγησης, είναι σημαντικό να υποθεί πως σε κάθε σενάριο μια μεταβλητή θα αλλάζει τιμές, ενώ οι υπόλοιπες θα μένουν στατικές. Επίσης τα πειράματα θα διεξαχθούν σε προσωπικό υπολογιστή, με σχετικά παλιό επεξεργαστή, οπότε οι τιμές που επιλέγονται δεν θα είναι ψηλές για να μην υπάρχουν προβλήματα στην μεταφορά μηνυμάτων, όσο αφορά το μέγεθος, αλλά ταυτόχρονα δεν θα είναι και μικρές για να μπορούμε να παρατηρήσουμε τις διαφορές.. Επιπρόσθετα θα υπάρχουν άλλοι εξωτερικοί παράγοντες που επηρεάζουν τα αποτελέσματα, όπως άλλες διεργασίες στον υπολογιστή και σφάλματα του υλικού.

Η μεθοδολογία αρχίζει ως εξής: Αρχικά διαγράφονται όλα τα output που αναφέρθηκαν πιο πάνω. Στη συνέχεια προσθέτουμε το start_time στην γραμμή πριν από την αποστολή ενός αιτήματος ενός πελάτη και το exec_time μετά από την παραλαβή της ανταπόκρισης από εξυπηρετητή. Σε περίπτωση αδυναμίας ανταπόκρισης, δεν σημειώνεται καμία τιμή . Για κάθε χρόνο εκτέλεσης που καταγράφουμε, γράφεται σε ένα αρχείο times_[sid].csv (sid είανι η ταυτότητα της διεργασίας που ορίσαμε) που είναι μια απλή μορφή της excel που βοηθά στην καταγραφή απλού κειμένου. Για να γραφτεί κείμενο σε διαφορετικό κελί οριζόντια, προσθέτουμε τον χαρακτήρα ‘,’(κόμμα) και για να γράψουμε σε διαφορετική γραμμή γράφουμε τον χαρακτήρα αλλαγής γραμμής “\n”.

Πριν την εκκίνηση των πειραμάτων χρειάζεται η πρόβλεψη των αποτελεσμάτων με την αυξομείωση των τιμών των μεταβλητών που επιλέξαμε. Μόλις τελειώσουν οι προβλέψεις δημιουργείται ένα csv αρχείο για κάθε σενάριο που εξετάζεται και κάθε csv αρχείο θα περιέχει 12 επιλεκτές της προσομοίωσης για κάθε διαφορετική τιμή της μεταβλητής (για μείωση των σφαλμάτων). Στη συνέχεια για κάθε 12 τιμές που δημιουργήθηκαν για μια συγκεκριμένη τιμή της μεταβλητής θα υπολογισθεί ο μέσος όρος των τιμών, που στη συνέχεια θα συμπεριληφθεί στην τελική γραφική παράσταση του σεναρίου. Τέλος για κάθε αποτέλεσμα που θα αντλήσουμε θα συγκριθεί με της αρχικές προβλέψεις για να βγουν τα συμπεράσματά.

6.3 Σενάρια Αξιολόγησης

Τα σενάρια αξιολόγησης επιλέχτηκαν γύρω από τις κύριες μεταβλητές του προγράμματος, ώστε να αξιολογήσουμε την επιρροή τους που έχουν στο πρόγραμμα. Επίσης όλα τα πειράματα θα γίνουν πάνω στην υλοποίηση της προσομοίωσης με τους εξυπηρετητές να έχουν το μοντέλο Ατομικής Συνέπειας. Όπως αναφέρθηκε στην πιο πάνω ενότητα, σε κάθε σενάριο θα αλλάζει μόνο μια μεταβλήτη, ενώ οι υπόλοιπες θα παραμείνουν οι ίδιες (όχι οι εξωτερικοί παράγοντες), για την απομόνωση της μεταβλήτης που εξετάζεται. Οι μεταβλητές που αποφασίστηκε να μελετήσουμε είναι ο αριθμός πελατών, ο αριθμός των εξυπηρετητών, η πιθανότητα τερματισμού των εξυπηρετητών και ο αριθμός των μηνυμάτων κάθε πελάτη. Επιπρόσθετα η τιμή της μεταβλητής f που ορίζει τον μέγιστο αριθμό εσφαλμένων εξυπηρετητών είναι N/2, όπου N ο αριθμός των εξυπηρετητών.

Για το κάθε σενάριο θα υποθέσουμε το πως θα επηρεαστούν τα αποτελέσματα με την αντίστοιχη αλλαγή και θα επεξηγήσουμε τον λόγο της υπόθεσης. Αρχικά η αύξηση των πελατών θα επιφέρει τετραγωνική αλλαγή (N^2) στον χρόνο εκτέλεσης, όπου N ο αριθμός των μηνυμάτων στην λίστα deliverable. Ο λόγος είναι επειδή αλλάζει ο αριθμός των μηνυμάτων N που επεξεργάζονται, άρα η αλλαγή αυτή υψώνεται στην δεύτερη δύναμη, λόγω της συνάρτησης sort που ταξινομεί την λίστα. Επίσης οι εξυπηρετητές έχουν μεγαλύτερο αριθμό αιτημάτων να ελέγξουν στα received για κάθε μήνυμα που παραλαμβάνουν, οπότε υπάρχει και γραμμική αύξηση. Για την περίπτωση που οι ενεργοί εξυπηρετητές αυξάνονται, ο χρόνος επίσης αυξάνεται τετραγωνικά αφού θα υπάρχει μεγαλύτερος αριθμός αποστολών σε κάθε Πολυεκπομπή και μεγαλύτερος αριθμός εξυπηρετητών που πιθανόν να καθυστερήσουν το σύστημα, αφού οι παραδόσεις εξαρτούνται από την πιο χαμηλή τιμή λογικού ρολογιου των

εξυπηρετητών. Επιπρόσθετα υπάρχει το σενάριο όπου αλλάζουμε τον αριθμό των μηνυμάτων.

Τέλος η καθυστέρηση του προγράμματος σε σχέση με την πιθανότητα τερματισμού του εξυπηρετητή πριν την Πολυεκπομπή ενός αιτήματος. Αυτό το σενάριο είναι πιο περίπλοκο για πειραματική αξιολόγηση. Για να βγουν τα απαραίτητα αποτελέσματα, χρειάζεται να τερματιστεί συγκεκριμένος αριθμός επεξεργαστών, ώστε να παρατηρήσουμε την διαφορά του χρόνου απάντησης του αιτήματος. Οπότε αντί για την πιθανότητα τερματισμού, θα χρησιμοποιήθει ο αριθμός των εσφαλμένων εξυπηρετητών ως μια πιο αξιόπιστη μεταβλητή. Η πρόβλεψη που γίνεται για αυτή την περίπτωση είναι η ίδια με αυτή του αριθμού των εξυπηρετητών, δηλαδή με την αύξηση των ενεργών εξυπηρετητών, αυξάνεται τετραγωνικά ο χρόνος ανταπόκρισης. Όμως σε αυτή την περίπτωση όσο αυξάνεται ο αριθμός των εσφαλμένων εξυπηρετητών, τόσο μειώνονται οι ενεργοί εξυπηρετητές. Άρα η πρόβλεψη είναι πως με την αύξηση των εσφαλμένων εξυπηρετητών, μειώνεται τετραγωνικά ο χρόνος επιστροφής μηνύματος του αιτήματος. Πάρακατω αριθμούνται τα σενάρια που θα εξεταστούν για κάθε μεταβλητή:

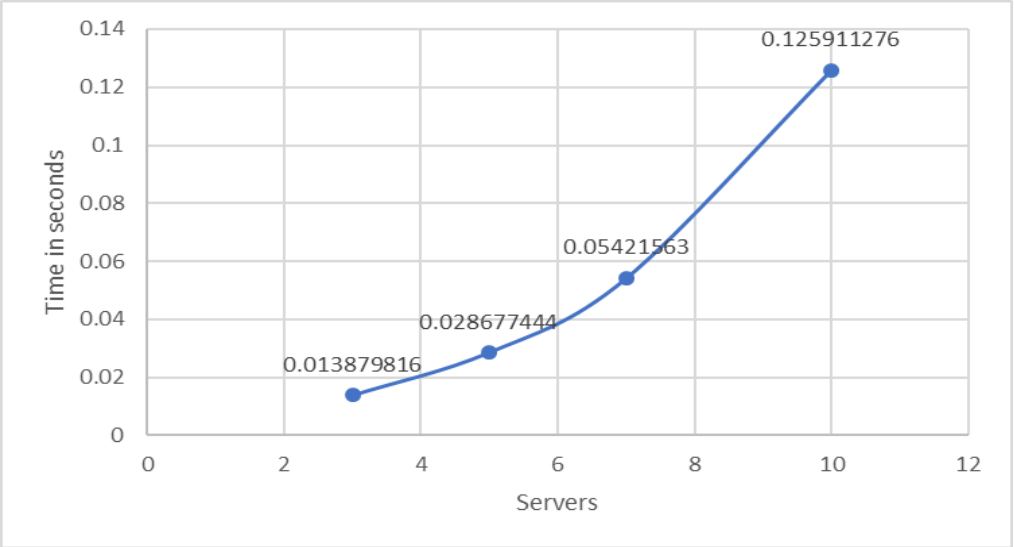
Συμβολισμός	**Όνομα**
s	Αριθμός εξυπηρετητών
cl	Αριθμός πελατών
cr	Αριθμός εσφαλμένων εξυπηρετητών
m	Αριθμός μηνυμάτων κάθε πελάτη

Μεταβλητή	**Σταθερά 1**	**Σταθερά 2**	**Σταθερά 3**
s=3,5,7,10	cl=10	cr=0	m=3
cl=5,7,10,20	s=5	cr=0	m=3
m=1,2,3,4	s=5	cl=10	cr=0
cr=0,1,2	s=5	cl=10	m=3

Όπως αναφέραμε και στο προηγούμενο κεφάλαιο, οι τιμές που επιλέχθηκαν δεν είναι μεγάλες, λόγω περιορισμού του υπολογιστή όπου θα επηρεάσει δραματικά την ταχύτητα, αλλά δεν πρέπει να είναι και μικρές οι τιμές για να μπορούμε να εξάγουμε ικανοποιητικά αποτελέσματα. Αφού το μέγιστο που μπορούμε να τρέξουμε για έναν ικανοποιητικό χρόνο είναι 10 εξυπηρετητές και 30 πελάτες, τότες επιλέγουμε μέγιστο 5 εξυπηρετητές και 20 πελάτες.

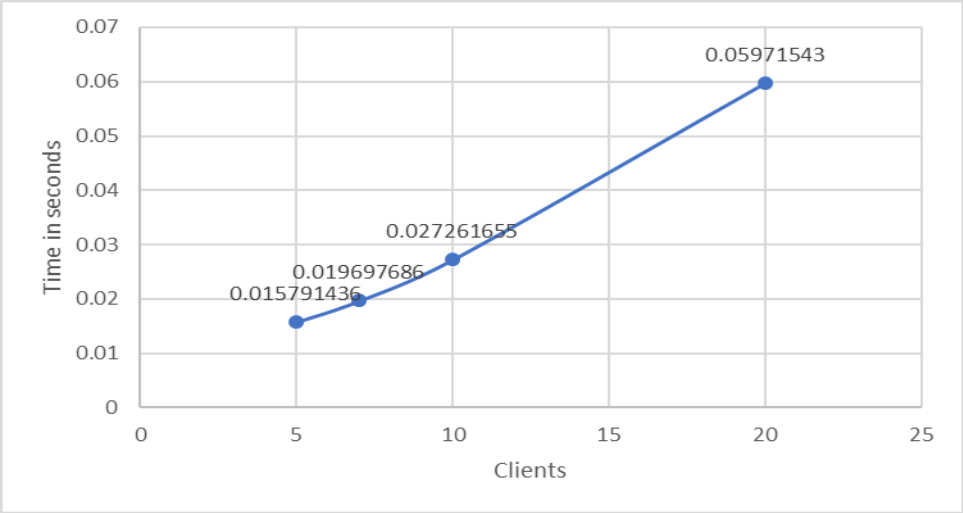
6.4 Αποτελέσματα

Όπως προαναφέρθηκε, για κάθε μέτρηση στην γραφική παράσταση πάρθηκε ο μέσος όρος του χρόνου 12 ανταποκρίσεων αιτημάτων του προγράμματος από τους εξυπηρετητές (χρήση των 10). Μετά από εκτελέσεις της προσομοίωσης βγήκαν οι ακόλουθες γραφικές για κάθε ένα από τα σενάρια που αναφέρθηκαν στην προηγούμενη υποενότητα. Αρχικά παρουσιάζουμε τα αποτελέσματα της καταμέτρησης χρόνου ανταπόκρισης με βάση τον αριθμό των εξυπηρετητών:



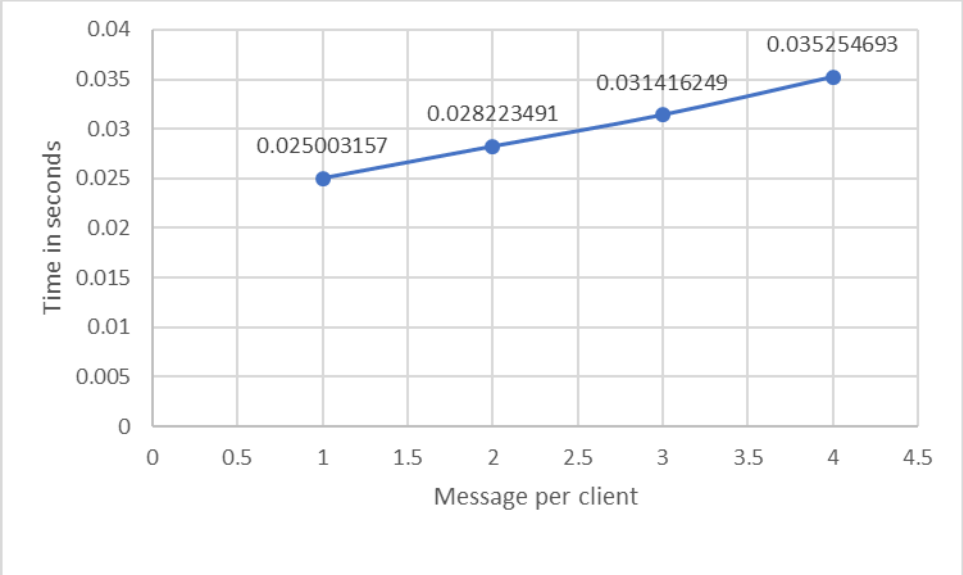
Σχήμα 6.4.1: Χρόνος ανταπόκρισης εξυπηρετητών με βάση τον αριθμό των εξυπηρετητών

Αρχικά δείχνει (Σχήμα 6.4.1) μια μικρή αλλαγή στην κλίση της γραμμής αλλα για 7 και 10 servers η αλλαγή κλίσης γίνεται μεγαλύτερη. Αυτό υποδηλώνει πως η γραφική είναι της μορφής x^2, δηλαδή για μεγαλύτερες τιμές, η κλίση θα είναι μεγαλύτερη , άρα η γραμμή θα ανέρχεται σε μεγαλύτερο βαθμό. Συνεχίζοντας παρουσιάζονται οι μέσοι όροι των μετρήσεων με βάση τον αριθμό των εξυπηρετητών:



Σχήμα 6.4.2: Χρόνος ανταπόκρισης εξυπηρετητών με βάση τον αριθμό των πελατών

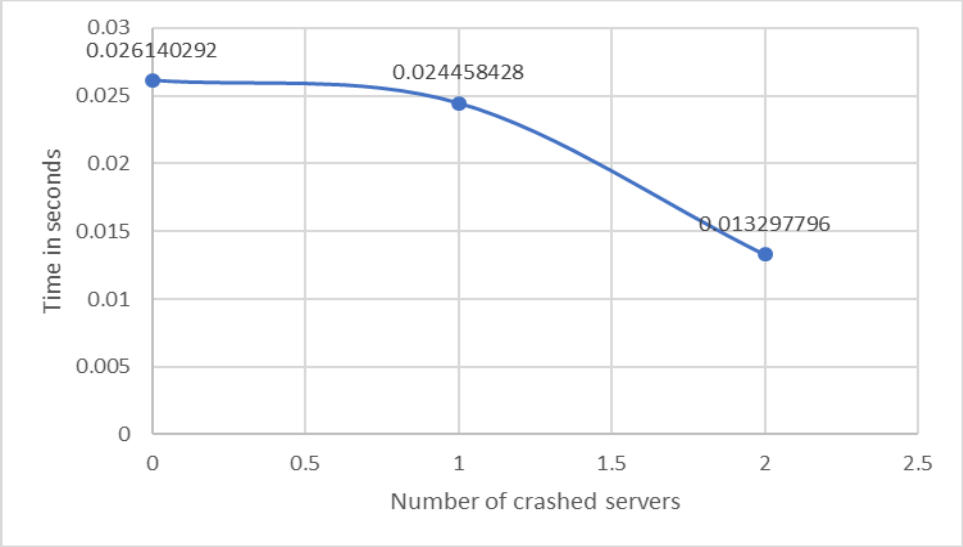
Όπως φαίνεται στην γραφική Σχήμα 6.4.2 αν και η κλίση αλλάζει λίγο για 5 μέχρι 7 πελάτες, στη συνέχεια μένει σταθερή, το οποίο μας δείχνει πως η αλλαγή του χρόνου εξυπηρέτησης των πελατων με βάση τον αριθμό των πελατών είναι γραμμική. Αυτό σημαίνει πως η πρόβλεψη που κάναμε για τον χρόνο ανταπόκρισης εξυπηρετητών δεν είναι αληθής, δηλαδή καταλήξαμε σε χρόνο τάξης N αντί N^2. Στη συνέχεια παρουσιάζονται οι μετρήσεις που έγιναν με μεταβλητή τον αριθμό των μηνυμάττων από τον πελάτη:



Σχήμα 6.4.3: Χρόνος ανταπόκρισης εξυπηρετητών με βάση τον αριθμό των αιτημάτων κάθε πελάτη.

Με βαση την γραφική (Σχήμα 6.4.3**)** φαίνεται πως η αύξηση του χρόνου ανταπόκρισης σε σχέση με τον αριθμό των αιτημάτων κάθε πελάτη είναι γραμμική, όμως η αύξηση δεν είναι μεγάλη. Άρα συμπεραίνουμε πως είναι λιγότερη η επιρροή του χρόνου ανταπόκρισης από

τον αριθμό μηνυμάτων που στέλνονται από τον πελάτη σε σχέση με τον αριθμό των πελατών. Τέλος παρουσιάζονται τα αποτελέσματα των μετρήσεων του χρόνου ανταπόκρισης του εξυπηρετητή με βαση τον αριθμό των εσφαλμένων εξυπηρετητών που εμφανίζονται:



Σχήμα 6.4.4: Χρόνος ανταπόκρισης εξυπηρετητών με βάση τον αριθμό εσφαλμένων εξυπηρετητών.

Όπως παρατηρείται στην γραφική Σχήμα 6.4.4 υπάρχει αλλαγή κλίσης που δηλώνει πως με την αύξηση των εσφαλμένων εξυπηρετητών, μειώνεται τετραγωνικά ο χρόνος ανταπόκρισης των εξυπηρετητών. Τα αποτελέσματά μας συνάδουν με τις προβλέψεις μας στην προηγούμενη υποενότητα.

6.5 Συμπεράσματα

Στην ανάλυση των σεναριών που συζητήθηκαν στις προηγούμενες υποενότητες, παρατηρείται σε γενικές γραμμές η συμπεριφορά του αλγόριθμου Ατομικής Συνέπειας όταν αλλάζουν κάποιες μεταβλητές. Επίσης αναλύεται η επιρροή που έχει η κάθε μεταβλητή αυξομειώνοντας τις τιμές τους. Οι μεταβλητές που χρησιμοποιήθηκαν για αυτά τα πειράματα είναι ο αριθμός των μη εσφαλμένων εξυπηρετητών, ο αριθμός των πελατών, ο αριθμός των μηνυμάτων για κάθε πελάτη και ο αριθμός των εσφαλμένων εξυπηρετητών (εξυπηρετητές που τερματίστικαν πρόωρα).

Αρχικά ο αριθμός των ορθών εξυπηρετητών αυξάνει τετραγωνικά τον χρόνο ανταπόκρισης των εξυπηρετητών, όπως ακριβώς προβλέπτηκε. Ο λόγος είναι επειδή με κάθε εξυπηρετητή που προστίθεται στο σύνολο των ορθών εξυπηρετητών, πληθύνεται ο αριθμός

των μηνυμάτων που χρειάζεται να στείλει ένας εξυπηρετητής κατά την Πολυεκπομπή (Broadcast) ενός αιτήματος και τα μήνυματα για τον συγχρονισμό των εξυπηρετητών. Επιπρόσθετα σε αυτή την καθυστέρηση προσθέτουμε και τον χρόνο που χρειάζεται ένας εξυπηρετητής για να προσθέσει το μήνυμα στο σύνολο των αιτημάτων, διότι χρειάζεται ο ενθυλακωμένος χρόνος του μηνυματος να είναι πιο χαμηλός από το ελάχιστο γνωστό λογικό ρολόι του κάθε εξυπηρετητή (LC). Ο αριθμός των εσφαλμένων εξυπηρετητών ωστόσο δίνει αντίθετο αποτέλεσμα, διότι η αύξηση τους σημαίνει η τετραγωνική μείωση των ορθών εξυπηρετητών.

Η επόμενη μεταβλητή που εξετάστηκε είναι ο αριθμός των πελατών, όπου κάθε πελάτης στέλνει από 3 μηνύματα APPEND. Η πρόβλεψη που έγινε στην υποενότητα 6.3 ήταν ότι με την αύξηση των πελατών, αυξάνεται τετραγωνικά ο χρόνος ανταπόκρισης των εξυπηρετητών, όμως στα πειράματα μπορούμε να δούμε πως η αύξηση ήταν γραμμική. Ο πιθανός λόγος για την γραμμική αύξηση είναι πως η αύξηση των πελατών προσθέτει στον αριθμό των αντικειμένων στο σύνολο S, γεγονός όπου αυξάνει τετραγωνικά τον χρόνο της ταξινόμησης της λίστας deliverable, όμως δεν υπολογίστηκε πως η deliverable χωρίζεται σε κομμάτια του συνολικού αριθμού μηνυμάτων που παραλαμβάνονται, οπότε πολυπλοκότητα του πέφτει δραμματικά. Οι υπόλοιπες λειτουργίες φέρουν γραμμική αύξηση επειδή αυξάνονται και οι λίστες receive και deliver, άρα αυξάνονται και οι ελέγχοι που γίνονται.

Τέλος είναι ο αριθμός των μηνυμάτων που στέλνονται μεταξύ των πελατών, όπου προβλέψαμε λάθος πως δεν επηρεάζει αρκετά τον χρόνο ανταπόκρισης, ενώ επηρεάζει γραμμικά το χρόνο. Ο λόγος είναι πως παρ'όλο που αλλάζει ο αριθμός των συνολικών αντικειμένων στο S, όπως αναλύσαμε και για το προηγούμενο σενάριο, η πολυπλοκότητα χρόνου πέφτει δραματικά, οπότε γίνεται αμελητέα. Ωστόσο επηρεάζει τον αριθμό των αιτημάτων στην λίστα received και delivered, γι'αυτό και υπάρχει αυτή η μικρή γραμμική αλλαγή στο χρόνο.

Συμπερασματικά, η υλοποίηση που δημιουργήθηκε σε αυτή την εργασία δεν είναι η πιο βέλτιστη που θα μπορούσε να γίνει, ωστόσο ακολουθεί πιστά τον αλγόριθμο που αναφέρθηκε στο Κεφάλαιο 3. Παρ'όλα αυτά καταφέραμε να βγάλουμε τα απαραίτητα συμπεράσματα για την επιρροή κάθε μεταβλητής προς τον χρόνο ανταπόκρισης. Ο αριθμός των ορθών εξυπηρετητών επηρεάζει σε μεγάλο βαθμό αρνητικά τον χρόνο ανταπόκρισης του συστήματος, ενώ ο αριθμός των πελατών και των μηνυμάτων επηρρεάζει αρνητικά σε γραμμικό επίπεδο λόγω της αύξησης των ολικών αιτημάτων που παραλαμβάνονται από κάθε επεξεργαστή.

Κεφάλαιο 7

Συμπεράσματα

7.1 Περίληψη

Εν συντομία, στην παρούσα διπλωματική εργασία μελετήθηκε και υλοποιήθηκε υπηρεσία Κατανεμημένων Αντικειμένων Κατάστιχου, που βασίζονται σε Αντικείμενα Κατάστιχου και προσφέρουν δύο εγγυήσεις ως προς την συνέπεια (τελική και ατομική). Επίσης έχουν ως πυρήνα την Υπηρεσία Ατομικής Πολυεκπομπής με ανοχή σφαλμάτων που βοηθάει στην ανακατανομή των παραλαμβάνοντος εγγραφών, ώστε να παραδοθούν με την ίδια σειρά. Επιπρόσθετα δημιουργήθηκε γραφικό περιβάλλον που καθιστά πιο εύκολη την χρήση του προσομοιωτή για έναν κοινό χρήστη, αποφεύγοντας την χρήση της γραμμής εντολών. Τέλος έγινε πειραματική αξιολόγηση για να δούμε πως επηρεάζεται ο χρόνος εκτέλεσης του προγράμματος με κάθε βασική μεταβλητή ενός τέτοιου συστήματος.

Η εργασία αυτή έδωσε την δυνατότητα να παρατηρήσουμε την χρηστικότητα και την σημαντικότητα ενός Κατανεμημένου συστήματος και να αποφασίσουμε πως όντως τέτοιοι αλγόριθμοι είναι υλοποιήσιμοι σε ένα περιβάλλον ασύγχρονο και με σφάλματα. Επίσης βλέπουμε από τους χρόνους εκτέλεσης πως ακόμη και με μια υλοποίηση που δεν θεωρείται βέλτιστη, καταφέραμε να βγάλουμε ορθολογιστικά αποτελέσματα. Όμως ταυτόχρονα αναλύοντας τις προκλήσεις που υπήρξαν και με τους αλγόριθμους, αλλά και με την

υλοποίηση με το DistAlgo μπορούμε να συμπεράνουμε πως χρειάζεται ακόμη πολλή μελέτη για να υπάρξει τέτοιο σύστημα.

7.2 Δυσκολίες και Προκλήσεις

Καθ'όλη την διάρκεια της διεξαγωγής της εργασίας, όπως αναφέρθηκαν και σε προηγούμενα κεφάλαια, υπήρξαν αρκέτες δυσκολίες στην κατανόηση των αλγόριθμων, στην υλοποίηση τους και στην υλοποίηση της διαπροσωπείας. Ένας από αυτές υπήρξε η κατανόηση των λειτουργιών της DistAlgo. Αν και υπάρχει εύκολο και αρκετά κατανοητό εγχειρίδιο [4] για την χρήση των εντολών, παρ'όλα αυτά υπήρχαν κάποια μικρά κενά τα οποία χρειάστηκε καιρός για τον εντοπισμό τους. Το πιο σημαντικό ήταν πως μια διεργασία έπρεπε να γνωρίζει την άλλη διεργασία για να σταλεί ένα μήνυμα. Το δύσκολο κομμάτι του εντοπισμού υπήρξε το μήνυμα που επιβεβαίωνε την επιτυχή σύνδεση μεταξύ των διεργασιών το οποίο έδινε την ψευδαίσθηση πως μπορούσε να πετύχει η ανταλλαγή μηνυμάτων χωρίς να οριστεί η κλάση στο ίδιο αρχείο .da. Επίσης, κατά την επικοινωνία με τους προγραμματιστές ανέφεραν πως υπήρξε δυνατότητα επικοινωνίας μεταξύ δύο προγραμμάτων DistAlgo σε διαφορετικές μηχανές, ωστόσο δεν φαινόταν εφικτό στην πράξη. Στο τέλος η επικοινωνία με τους προγραμματιστές δεν επέφερε θετική κατάληξη στο θέμα αυτό, και για αυτό η πειραματική αξιολόγηση έγινε σε μια μηχανή.

Στην υλοποίηση η πρόκληση που είχαμε είναι η εύρεση πηγών που να συμπεριλαμβάνει πλήρη επεξήγηση του αλγόριθμου Υπηρεσίας Ατομικής Πολυεκπομπής. Για να γίνει αυτό χρειάστηκε η δοκιμή των αλγόριθμων που βρέθηκαν και η μελέτη των Reliable Broadcast αλγόριθμων[2] για καλύτερη κατανόηση. Η μελέτη του βιβλίου επίσης βοήθησε στην υλοποίηση του fail-stop κομματιού του Τερματικού αλγόριθμου (**Terminating Broadcast**). Επιπρόσθετα υπήρξε αδυναμία στην υλοποίηση του ΥΑΠ, ο οποίος να δέχεται την ταυτόχρονη αποστολή ενός αιτήματος σε πολλούς εξυπηρετητές από έναν πελάτη, Ο τρόπος που έγιναν οι δοκιμές του προγράμματος ήταν με κάθε διεργασία να μοιράζει από 6 μηνύματα σε όλους τους υπόλοιπους μέχρι όλοι οι εξυπηρετητές να παραδόσουν τις εγγραφές με την ίδια σειρά. Ενδιάμεσα τυπώνονταν και οι ενέργειες που λαμβάνονταν από κάθε εξυπηρετητή.

Στη διαπροσωπεία η μεγαλύτερη δυσκολία που υπήρξε ήταν η υλοποίηση της κίνησης των αντικειμένων του Αντικειμένου Κατάστιχου. Υπήρξαν πληθώρα προβλήματα στην δημιουργία της κίνησης,άλλα η μεγαλύτερη δυσκολία ήταν πως οι κινήσεις έτρεχαν παράλληλα και δεν περίμεναν τον τερματισμό της προηγούμενης. Ο τρόπος αντιμετώπισης

αυτού του προβλήματος ήταν να τα προσθέσω όλα σε μια λίστα από animations και μετά το ένα να τρέχει το άλλο μέχρι να εξαντληθούν, αλλά και αυτό είχε τον βαθμό δυσκολίας.

7.3 Μελλοντική Εργασία

Όπως συζητήθηκε πιο πάνω υπήρξαν αρκετά προβλήματα, κάποια εκ των οποίων δεν γνωρίζουμε αν είναι υλοποιήσιμα. Ωστόσο παρακάτω θα αναφέρονται κάποιες από τις πιθανές επεκτάσεις που μπορεί να αποκτήσει αυτή η εργασία. Αρχικά οι πελάτες να μπορούν να στείλουν ταυτόχρονα σε αρκετούς εξυπηρετητές αίτημα και να υπάρχει ολική διάταξη στα δεδομένα του Αντικείμενου Κατάστιχου. Αυτή η επέκταση θα μας βοηθήσει να υλοποιήσουμε και τον μηχανισμό Ανοχής Βυζαντινών Σφαλμάτων. Η ιδέα αυτού του μηχανισμού είναι να βεβαιώσει τον πελάτη πως παραλαμβάνει την σωστή απόκριση από τον εξυπηρετητή, δηλαδή να αποφύγει οποιαδήποτε παραποίηση της ανταπόκρισης. Το πως πετυγχαίνεται αυτό είναι να οριστούν 2f+1 εξυπηρετητές, που θα στείλουν ανταπόκριση στον εξυπηρετητή και να δεχτούν την απάντηση που έστειλαν οι περισσότερες διεργασίες. Επίσης μπορούμε να μελετήσουμε την δυνατότητα περισσότερων αιτημάτων, όπως για παράδειγμα η παραλαβή του τελευταίου μηνύματος στο σύνολο του Αντικείμενου Κατάστιχου. Ακόμη μπορούμε να δώσουμε την δυνατότητα ενός εξυπηρετητή να τερματίζεται σε οποιαδήποτε στιγμή της λειτουργίας του, αντί μόνο σε πολυεκπομπή αιτήματος, ώστε να προσομοιώσουμε καλύτερα το πραγματικό περιβάλλον.

Για την διαπροσωπεία υπάρχουν κάποιες ιδέες που μπορούν να υλοποιηθούν, οι οποίες την καθυστούν πιο εύχρηστη και πιο χρήσιμη. Πρώτο και πιο κύριο είναι η λεπτομερής κινήση των μηνυμάτων μεταξύ πελατών και εξυπηρετητών στο παράθυρο Αποτελεσμάτων. Αυτή την στιγμή δεν υπάρχει μεγάλη λεπτομέρεια, μόνο δείχνουμε τις κινήσης που προβλέπονται ότι έγιναν, με βάση τα δεδομένα που αποθηκεύτηκαν. Θα ήταν πιο χρήσιμο να δείχνει με κάθε κίνηση ακριβως πως έφτασε σε αυτό το σημείο, χωρίς προβλέψεις και με περισσότερα στοιχεία να εμφανίζονται, όπως είναι οι λίστες self.delivered και self.received. Επιπρόσθετα θα ήταν πιο χρήσιμο να δηλώναμε στο Παράθυρο Διαμόρφωσης την επιλογή να τερματιστεί πρόωρα ένας συγκεκριμένος αριθμός εξυπηρετητών, ώστε να υπάρχει μεγαλύτερος έλεγχος των αποτελεσμάτων.

Τέλος για την πειραματική αξιολόγηση θα μπορούσαμε να ελέγχθει με παραπάνω λεπτομέρια το πως επηρεάζεται ο χρόνος ανταπόκρισης των επεξεργαστών σε σχέση με τους παράγοντες που αναλύθηκαν στο Κεφάλαιο 6.

Βιβλιογραφία

[1] Xavier Defago, Andre´ Schiper and Peter Urban, *Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey*

[2] Rachid Guerraoui and Luis Rodrigues, *Introduction to Reliable Distributed Programming*, Springer-Verlag Berlin Heidelberg, 2006

[3] Y. Annie Liu and Scott Stoller, From Classical to Blockchain Consensus: What are the Exact Alogrithms?, 2019

Slides:https://drive.google.com/file/d/1uzxBVK2AQjnnx1E1fo5X4wZtSdXfNYKT/view

[4] Yanhong A. Liu, Bo Lin, and Scott Stoller DistAlgo Language Description 2017 Available: https://www3.cs.stonybrook.edu/~liu/distalgo/language.pdf [Accessed 10-February- 2020]

[5] Yanhong A., Liu Scott D. Stoller, Bo Lin and Michael Gorbovitski, "*From Clarity to Efficiency for Distributed Algorithms*, Computer Science Department, State University of New York at Stony Brook, Stony Brook, NY 11794, USA

[6] Yanhong A. Liu, Scott D. Stoller, and Bo Lin, *High-Level Executable Specifications of Distributed Algorithms*, Computer Science Department, State University of New York at Stony Brook

[7] Yanhong A. Liu, Scott D. Stoller, and Bo Lin, *High-Level Executable Specifications of Distributed Algorithms*.

Slides: https://drive.google.com/file/d/0B0MWH8ngLAIFbmU4SE5TbGRYRk0/view

[8] S. Nakamoto, Bitcoin: A Peer-to-Peer Electronic Cash System, 2008. [Online].

Available: https://bitcoin.org/bitcoin.pdf [Accessed: 16- May- 2019].

[9] George Panagiotou, *Υλοποίηση και πειραματική αξιολόγηση Αλγόριθμων Αλυσίδας Κοινοποιήσεων με τη χρήση ZeroMQ*, Πανεπιστήμιο Κύπρου, 2019

[10] Antonio Fernández Anta, Chryssis Georgiou, Kishori Konwar, and Nicolas Nicolaou, Formalizing and Implementing Distributed Ledger Objects, ACM SIGACT News, Volume 49, Issue 2, pp. 58-76, June 2018.

[11] DistAlgo, διαθέσιμο στο Github
Available: https://github.com/DistAlgo/distalgo

[12] PyQt5
Available: https://pypi.org/project/PyQt5/

[13] PyQt5 5.14.2.1 documentation
Available: https://doc.qt.io/qtforpython/

[14] Qt Designer
Available: https://build-system.fman.io/qt-designer-download

ΠΑΡΑΡΤΗΜΑ Α:

```
class AtomicBroadcastService(process):
    def setup(sid,clients,servers):
        self.received=[]
        self.delivered=[]
        self.allservers=servers
        self.servers=servers-{self}
        self.broadcast=0
        self.current_servers=[i for i in self.servers]
        self.FLTC={}
        for p in allservers:
            self.FLTC[p]=0
        self.shutdown=False
        self.LC={}
        self.counter=0
        for p in allservers:
            self.LC[p]=0
        #output('setup completed')
    def ABroadcastGET(c,p):
        self.LC[self]+=1
        if (not (hasSubTuple((c,'GET','',p),self.received,1))):
            self.received.append((self.LC[self],c,'GET','',p,str(self)))
        send((self.LC[self],c,'GET',None,p),to=self.current_servers)
    def ABroadcast(c,r,p):
        self.LC[self]+=1
        if (not (hasSubTuple((c,'APPEND',r,p),self.received,1))):
            self.received.append((self.LC[self],c,'APPEND',r,p,str(self)))
        send((self.LC[self],c,'APPEND',r,p),to=self.current_servers)
    def uponDeliverGET(c,p):
        i=0
    def deliver(m):
        if (not self.shutdown):
            self.delivered.append(m)
            uponDeliverAPPEND(m[1],m[3],m[4],m[5])
    def deliverGET(m):
        if (not self.shutdown):
            self.delivered.append(m)
            uponDeliverGET(m[1],m[4])
    def uponDeliverAPPEND(c,r,p,s):
        i=0
```

```
def receive(msg=('Exclude',server),from_=s):
    if (server in self.current_servers):
        #output("self.current_servers:",self.current_servers)
        self.current_servers.remove(server)
        self.LC.pop(server)
        output("remove done!")
        output('Exclude ',server, ' from the list of servers')
    if (server==self):
        self.shutdown=True
        output('shutdown due to delay')
    self.faulty_servers+=1

def receive(msg=(cserver,c,action,r,p),from_=s):
    if (not self.shutdown):
        self.counter+=1
        self.LC[self]=max(cserver,self.LC[self])+1
        for i in self.current_servers:
            self.FLTC[i]+=1
        if (action!='EMPTY'):
            if (not (hasSubTuple((c,action,r,p),self.received,1))):
                self.received.append((cserver,c,action,r,p,str(s)))
            send((cserver,'','EMPTY','',''),to=self.current_servers)
        #output('Change server timer')
        if s in self.LC:
            self.FLTC[s]=0
            if (self.LC[s]<cserver):
                self.LC[s]= cserver
        #output('Check servers clock')
        if (len(self.current_servers)!=0):
            if (self.counter%len(self.current_servers)/2==0):
                for i in self.current_servers:
                    if (self.FLTC[i]>5*len(self.current_servers)):
                        if (i==self):
                            #output('I delay so i close myself')
                            self.shutdown=True
                        else:
                            popped=self.LC.pop(i)
                            if (i in self.current_servers):
                                self.current_servers.remove(i)
                            #output('Exclude ',str(i), ' from the list of servers')
                        send(('Exclude',i),to=self.current_servers)

            #output('servers clock Checked ')

        deliverable=[]
        temp=[i for i in self.received if i not in self.delivered]
        #temp2=[i for i in self.current_servers]
        for m in temp:
            if (m[0]<=min(self.LC.values())):
                deliverable.append(m)
        deliverable=sort(deliverable,0,5,True)
        if (not self.shutdown):
            for m in deliverable:
                if(m[2]=='GET'):
                    deliverGET(m)
                elif(m[2]=='APPEND'):
                    deliver(m)
```

ΠΑΡΑΡΤΗΜΑ Β:

```
class Atomic_Server(process,AtomicBroadcastService):
    def setup(sid,clients,servers,f,main,chance,tic):
        self.S=[]
        self.f=f
        self.pending=[]
        self.get_pending=[]
        self.allservers=servers
        self.servers=servers-{self}
        self.current_servers=[i for i in self.servers]
        self.shutdown=False
        self.chance=chance
        self.faulty_servers=0
        self.main=main
        #output('setup complete')
    ######################################clients to servers functions#########################################
    def receive(msg=(c,'GET'),from_=p):
        if (self.shutdown):
            output('Ignoring get request')
        else:
            rand=random.random()

            if (self.chance>rand and self.faulty_servers<len(self.allservers)-self.f):
                send("faulty server",to=self.servers)
                self.faulty_servers+=1
                output('Unexpected shut down')
                self.shutdown=True
            else:
                self.get_pending.append((c,p))
                ABroadcastGET(c,p)
    def receive(msg=(c,'APPEND',r),from_=p):
        if (self.shutdown):
            output('Ignoring append request')
        else:
            rand=random.random()
            if (self.chance>rand and self.faulty_servers<len(self.allservers)-self.f):
                #output(rand,'<',self.chance)
                send("faulty server",to=self.servers)
                self.faulty_servers+=1
                output('Unexpected shut down')
                self.shutdown=True
            else:
                self.pending.append((c,r,p))
                ABroadcast(c,r,p)
```

```
######################################server to clients functions#########################################
def uponDeliverGET(c,p):
    if (not self.shutdown):
        if ((c,p) in self.get_pending):
            send((c,'GETRES',self.S),to=p)
            self.get_pending.remove((c,p))

def uponDeliverAPPEND(c,r,p,s):
    if (not self.shutdown):
        if (not((c,r,str(p),str(s)) in self.S)):
            self.S.append((c,r,str(p),str(s)))
            for m in self.pending:
                if ((m[0],r,m[2]) in self.pending):
                    send((m[0],'APPENDRES','ACK'),to=m[2])
                    self.pending.remove((m[0],r,m[2]))
                    break;

def run():
    send(('online',self),to=self.clients)
    #output("I'm online")
    if (await(len(self.S)==tic or self.shutdown)):
        send(('done',self),to=self.servers)
        await(each(c in self.servers ,has=received(('done',c))))
        send(('stop',self),to=self.clients)
        await(each(c in self.clients ,has=received(('stop',c))))
        send(('done',self.S),to=self.main)
    elif(timeout(tic)):
        output('timeout')
        send(('done',self),to=self.servers)
        await(each(c in self.servers ,has=received(('done',c))))
        send(('stop',self),to=self.clients)
        await(each(c in self.clients ,has=received(('stop',c))))
        send(('done',self.S),to=self.main)

    output('shut down server')
```

ΠΑΡΑΡΤΗΜΑ Γ:

```
class Sender(process):
    def setup(sid,allservers,servers,commands,main):
        self.LC=0
        self.servers=servers
        self.commands=commands
        self.wait=[]
        self.wait.append(True)
        output('setup complete')
    ###################################client functions#####################################
    def receive(msg=(c,'GETRES',V),from_=m):
        if (self.wait[c]==False):
            output('I got '+str(len(V))+' items from ',str(m),'get: ',V)
            self.wait[c]=True
    def get():
        self.LC+=1
        output('sending get request at time ' + str(self.LC))
        c=self.LC
        output('Waiting for reply for get')
        self.wait.append(False)
        length=len(self.servers)
        counter=0
        while(counter<3):
            for i in range(0,length):
                try:
                    send((c,'GET'),to=self.servers[i])
                    output('Sending ','(',(c,'GET'),') to server ',self.servers[i])
                    if (await(self.wait[c])):
                        counter=3
                        break
                    elif timeout(0.5*len(allservers)):
                        output(str(self.servers[i])+' is not responding.Switching to another server')
                except Exception as e:
                    output("Couldn't send to "+str(self.servers[i]))
            counter+=1
        if (self.wait[c]!=True):
            output("Couldn't send the message to servers")
            send(("-1",self),to=main)
    def receive(msg=(c,'APPENDRES',r),from_=m):
        if (self.wait[c]==False):
            output(r)
            self.wait[c]=True
```

```
    def appendItem(r):
        self.LC+=1
        c=self.LC
        self.wait.append(False)
        length=len(self.servers)
        counter=0
        while(counter<3):

            for i in range(0,length):
                try:
                    send((c,'APPEND',r),to=self.servers[i])
                    output('Sending ','(',(c,'APPEND',r),') to server ',self.servers[i])
                    if (await(self.wait[c])):
                        counter=3
                        break
                    elif timeout(0.5*len(allservers)):
                        output(str(self.servers[i])+' is not responding.Switching to another server')
                except Exception as e:
                    output("Couldn't send to "+str(self.servers[i]))
            counter+=1
        if (self.wait[c]!=True):
            output("Couldn't send the message to servers")
            send(("-1",self),to=main)

    def run():
        await(each(s in servers, has=received(('online',s))))
        for c in self.commands:
            if (c=='null'):
                break;
            temp=c.split("::")
            action=temp[0]
            if (action=='APPEND'):
                message=temp[1]
                appendItem(message)
            elif(action=='GET'):
                get()
            else:
                output('Invalid command "'+str(action))
        await(each(s in self.allservers, has=received(('stop',s))))
        send(('stop',self),to=self.allservers)
        output('shut down client')
```