Ατομική Διπλωματική Εργασία

ΥΛΟΠΟΙΗΣΗ ΚΑΙ ΠΕΙΡΑΜΑΤΙΚΗ ΑΞΙΟΛΟΓΗΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ ΑΤΟΜΙΚΩΝ ΑΝΤΙΚΕΙΜΕΝΩΝ ΓΡΑΦΗΣ/ΑΝΑΓΝΩΣΗΣ SLIQ

Αντιγόνη Χατζηδημητρίου

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Ιούνιος 2009

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Υλοποίηση και Πειραματική Αξιολόγηση Αλγορίθμου Ατομικών Αντικειμένων Γραφής/Ανάγνωσης

Αντιγόνη Χατζηδημητρίου

Επιβλέπων Καθηγητής
Δρ. Χρύσης Γεωργίου

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

Ιούνιος 2009

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου Δρ. Χρύση Γεωργίου για την υποστήριξη και την καθοδήγηση που μου προσέφερε κατά την διάρκεια της εκπόνησης αυτής της εργασίας. Η άψογη συνεργασία που είχαμε συνέβαλε στην επιτυχή ολοκλήρωση της εργασίας αυτής.

Θα ήθελα, επίσης, να ευχαριστήσω τον Νικόλα Νικολάου για το πρόγραμμα που μου έδωσε. Το πρόγραμμα αυτό μου φάνηκε χρήσιμο τόσο για την καλύτερη κατανόηση κάποιων εννοιών που χρησιμοποιούνται σε αυτή την εργασία όσο και για τη χρήση των αποτελεσμάτων του σε συνδυασμό με το δικό μου κώδικα.

Περίληψη

Πολλές εφαρμογές στις μέρες μας υλοποιούνται σε κατανεμημένα συστήματα. Εφαρμογές όπως τις υπηρεσίες Διαδικτύου, peer-to-peer εφαρμογές και cloud computing χρησιμοποιούν την αφηρημένη έννοια των κατανεμημένων συστημάτων. Κύριος στόχος στον κατανεμημένο υπολογισμό είναι να μοιάζει σαν τον παραδοσιακό υπολογισμό σε μια μηχανή. Ένα από τα μείζων θέματα είναι η συνέπεια μνήμης μεταξύ των διεργασιών στο σύστημα. Η διατήρηση της ατομικότητας των δεδομένων είναι σημαντική για την αποτελεσματικότητα των εφαρμογών στο σύστημα. Ένας αλγόριθμος εγγραφής/ανάγνωσης ατομικών αντικειμένων έχει σχεδιαστεί και αποδειχθεί για την ορθότητα του στο [6].

Σε αυτή τη διπλωματική εργασία γίνεται η πειραματική αξιολόγηση του αλγορίθμου στο [6] σε ρεαλιστικό, ασύγχρονο δίκτυο και κάτω από την παρουσία σφαλμάτων. Συγκεκριμένα η απόδοση του αλγορίθμου αξιολογείται στο κατανεμημένο δίκτυο Planet Lab, που παρέχει τις συνθήκες που απαιτούνται για την αξιολόγηση του αλγόριθμου, την ασυγχρονία και την παρουσία σφαλμάτων δικτύου. Το Planet Lab είναι ένα ερευνητικό δίκτυο παγκοσμίου κλίμακας το οποίο υποστηρίζει την ανάπτυξη νέων δικτυακών εφαρμογών και υπηρεσιών. Αποτελείται από 1006 σταθμούς σε όλο τον κόσμο, από τους οποίους οι δύο ανήκουν στο πανεπιστήμιο μας.

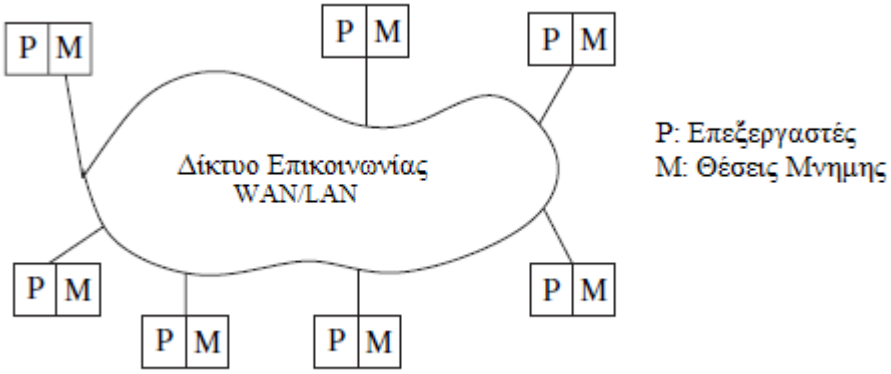
Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

1.1 Κατανεμημένα Συστήματα

Ένα κατανεμημένο σύστημα όπως περιγράφεται στο [9] είναι μια συλλογή από αυτόνομους επεξεργαστές οι οποίοι επικοινωνούν μέσω ενός δικτύου και χαρακτηρίζονται από τη γεωγραφική απόσταση, την αυτονομία, την ασυγχρονία και τη διαφορετική μνήμη. Ένα τυπικό κατανεμημένο σύστημα φαίνεται στο Σχήμα 1.1. Κάθε σταθμός έχει δικό του επεξεργαστή και μνήμη και όλοι οι σταθμοί είναι ενωμένοι μεταξύ τους μέσω ενός δικτύου. Το δίκτυο, εξυπηρετεί την ανταλλαγή πληροφοριών μεταξύ των επεξεργαστών. Επίσης, σε ένα τέτοιο δίκτυο πρέπει να έχουμε υπόψη ότι τα μηνύματα μπορεί να χαθούν, να φτάσουν στον προορισμό τους με διαφορετική σειρά από αυτήν που στάλθηκαν, να αλλοιωθεί το περιεχόμενο τους ή και να υπάρξουν διπλά πακέτα. Τέλος, οι σταθμοί και οι γραμμές επικοινωνίας μεταξύ των σταθμών μπορεί να καταρρεύσουν.



Σχήμα 1.1: Ένα κατανεμημένο Δίκτυο ενώνει τους επεξεργαστές μέσω ενός Δικτύου Επικοινωνίας [9].

Ένα κατανεμημένο πρόγραμμα είναι ένα σύνολο από N ασύγχρονες διεργασίες Π_1, Π_2, Π_3, . . . , Π_N που επικοινωνούν με ανταλλαγή μηνυμάτων μέσω ενός δικτύου. Έστω $K_{\chi\psi}$ το κανάλι από τη διεργασία Π_χ στη διεργασία Π_ψ και $M_{\chi\psi}$ το μήνυμα που στάλθηκε από την Πχ στη Π_ψ. Όπως αναφέραμε και προηγουμένως, η καθυστέρηση επικοινωνίας είναι περιορισμένη και μη προβλέψιμη. Επιπλέον, κάθε διεργασία είναι αυτόβουλη, δηλαδή αν στέλνει κάποιο μήνυμα δεν περιμένει ώσπου το μήνυμα να παραδοθεί.

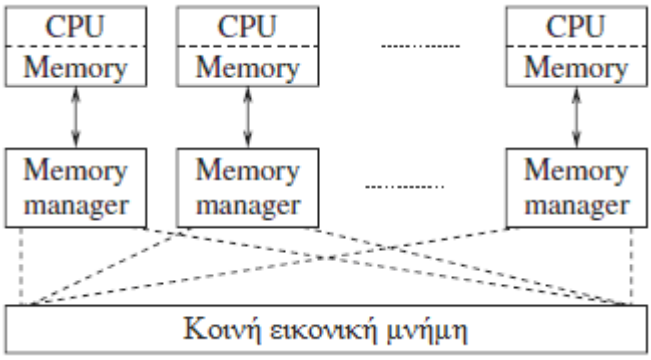
Ένας κατανεμημένος υπολογισμός αποτελείται από ένα σύνολο καθορισμένων διεργασιών, οι οποίες επικοινωνούν αποκλειστικά με ανταλλαγή μηνυμάτων. Όλα τα μηνύματα λαμβάνονται ορθά μετά από ένα τυχαίο πεπερασμένο χρονικό διάστημα. Η επικοινωνία όπως αναφέρθηκε και πιο πάνω είναι ασύγχρονη, δηλαδή μια διεργασία δεν περιμένει για τον παραλήπτη να είναι έτοιμος πριν στείλει ένα μήνυμα.

1.2 Κατανεμημένη Κοινή Μνήμη

Η κατανεμημένη κοινή μνήμη (Distributed Shared Memory) [9] είναι μια αφηρημένη έννοια στα κατανεμημένα συστήματα. Δίνει την αίσθηση μιας μόνο μνήμης με μόνο ένα χώρο διευθύνσεων. Οι προγραμματιστές έχουν πρόσβαση στα δεδομένα διαμέσου του δικτύου χρησιμοποιώντας τις αρχέγονες συναρτήσεις read και write, με παρόμοιο

τρόπο όπως σε συστήματα με ένα μόνο επεξεργαστή. Επιπρόσθετα, οι προγραμματιστές δεν είναι απαραίτητο να ασχοληθούν με τις συναρτήσεις send και receive και την πολυπλοκότητα του συγχρονισμού και της συνέπειας στο μοντέλο κατανεμημένης μνήμης με ανταλλαγή μηνυμάτων.

Ένα μέρος της μνήμης κάθε σταθμού παραμερίζεται κατά μέρος για την κοινή χρήση από το σύστημα, ενώ το υπόλοιπο αφήνεται για την προσωπική χρήση μόνο αυτού του υπολογιστή. Για καλύτερη κατανόηση η κατανεμημένη κοινή μνήμη παραλληλίζεται με μνήμη ενός χώρου διευθύνσεων, την κοινή εικονική μνήμη (Σχήμα 1.2).



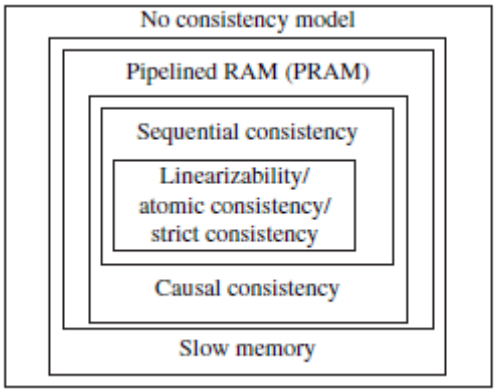
Σχήμα 1.2: Απεικόνιση της κοινής κατανεμημένης μνήμης [9].

Μολονότι πλασάρονται οικείες συναρτήσεις (read, write) για διασύνδεση, εξακολουθεί να υπάρχει η ανάγκη διαμοιρασμού των δεδομένων μέσω του δικτύου στο κατανεμημένο σύστημα. Συν τοις άλλοις, υπάρχει η πιθανότητα ταυτόχρονης πρόσβασης στα δεδομένα, για αυτό επιβάλλεται κάποιος μηχανισμός ταυτοχρονίας. Ειδικότερα, όταν περισσότεροι του ενός επεξεργαστές επιθυμούν πρόσβαση στο ίδιο αντικείμενο πρέπει να αποφασιστεί πως θα χειριστούν οι ταυτόχρονες προσβάσεις στη μνήμη. Θα αναφερθούν στη συνέχεια διάφορες υλοποιήσεις αλγορίθμων που σχεδιάστηκαν για αυτό το σκοπό και για να αντιμετωπίσουν προβλήματα σαν τα πιο πάνω. Επίσης, το θέμα αυτής της εργασίας αφορά ένα τέτοιο αλγόριθμο.

1.2.1 Μοντέλα συνοχής μνήμης

Συνοχή μνήμης είναι η ικανότητα του συστήματος να εκτελεί ορθά λειτουργίες μνήμης. Ας υποθέσουμε ότι έχουμε N διεργασίες που εκτελούν X λειτουργίες μνήμης σειριακά. Τότε, υπάρχει ένας μεγάλος αριθμός διαφορετικών συνδυασμών της εκτέλεσης αυτών των λειτουργιών. Για να εξασφαλιστεί η συνοχή μνήμης είναι αρκετό να βρεθούν ποιοι συνδυασμοί είναι ορθοί. Αυτό οδηγεί στην ανάγκη ορισμού της ορθότητας. Το μοντέλο συνοχής μνήμης ορίζει το σύνολο από τις επιτρεπτές διατάξεις των προσβάσεων στη μνήμη. Αν και ένας παραδοσιακός ορισμός της ορθότητας λέει ότι μια ορθή εκτέλεση στη μνήμη είναι αυτή στην οποία κάθε λειτουργία ανάγνωσης επιστρέφει την τιμή που αποθηκεύτηκε στην πιο πρόσφατη λειτουργία εγγραφής, η έννοια του πιο πρόσφατου καθίσταται διφορούμενη στην παρουσία ταυτόχρονων προσβάσεων στη μνήμη και πολλαπλών αντιγράφων του αντικειμένου.

Υπάρχουν έξι διαφορετικά μοντέλα συνοχής μνήμης τα οποία φαίνονται στο Σχήμα 1.3. Για τους σκοπούς αυτής της διπλωματικής εργασίας μας ενδιαφέρει μόνο αυτό της ατομικής μνήμης ή αλλιώς της γραμμικής μνήμης (linearizable) [13] το οποίο ορίζεται σε επόμενο κεφάλαιο.



Σχήμα 1.3: Ιεραρχία των μοντέλων συνοχής μνήμης [9].

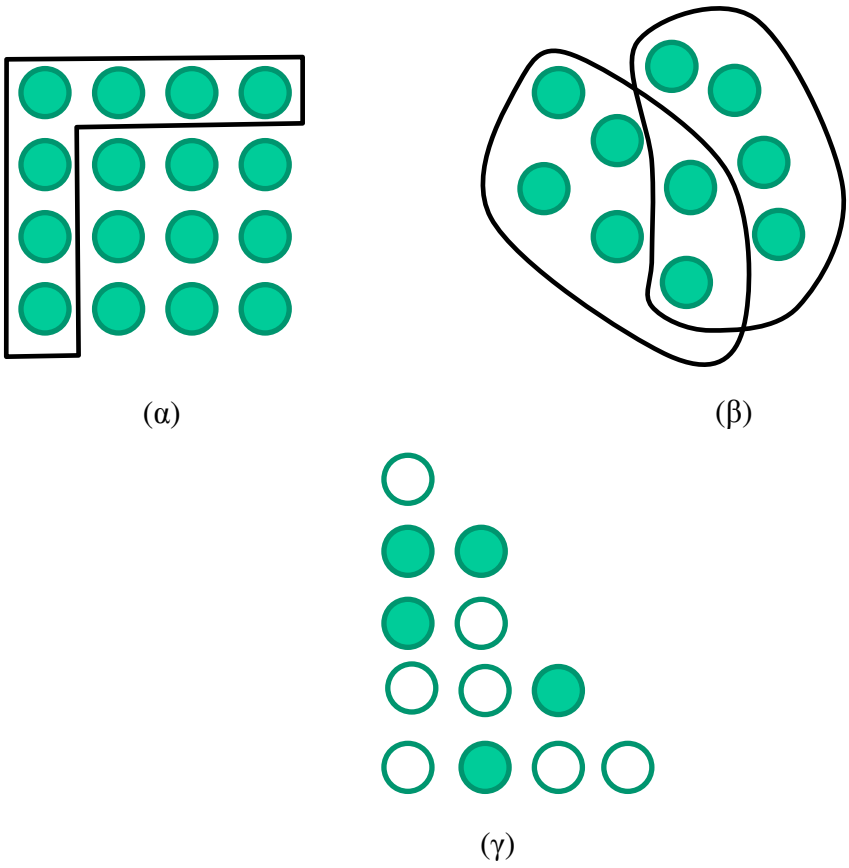
1.3 Συστήματα Απαρτίας (Quorum Systems)

Τα συστήματα απαρτίας [13] είναι μια θεμελιώδης αφηρημένη έννοια η οποία χρησιμοποιείται για τον αξιόπιστο συντονισμό των επεξεργαστών σε ένα κατανεμημένο σύστημα. Ορίζονται ως ακολούθως. Έστω Π = {Π1, . . . , ΠΝ} ένα σύνολο από εξυπηρετητές. Το σύστημα απαρτίας, $Q \subset 2^{P,}$ είναι ένα υποσύνολο του Π τέτοιο ώστε κάθε δύο ζεύγη υποσυνόλων να τέμνονται μεταξύ τους.

Ας μελετήσουμε ένα απλό αντιπροσωπευτικό παράδειγμα, τον αλγόριθμο του αμοιβαίου αποκλεισμού, χρήσης των συστημάτων απαρτίας σε κατανεμημένο σύστημα για να αντιληφθούμε τη συνεισφορά και τον τρόπο χρήσης των συστημάτων αυτών στον κατανεμημένο υπολογισμό. Υποθέτουμε ότι οι εξυπηρετητές στο Π μοιράζονται μεταξύ τους κάποιο κοινό πόρο, ο οποίος απαιτεί αμοιβαίο αποκλεισμό για την πρόσβαση του. Όταν ένας εξυπηρετητής επιθυμεί να έχει πρόσβαση στο κρίσιμο τμήμα, πρέπει πρώτα να πάρει άδεια από όλους τους εξυπηρετητές ενός quorum. Αυτό γίνεται διαλέγοντας ο εξυπηρετητής ένα τυχαίο quorum και ζητώντας από όλους τους εξυπηρετητές σε αυτό το quorum άδεια για το κρίσιμο τμήμα. Όταν εγκαταλείπει το κρίσιμο τμήμα ένας εξυπηρετητής, με τον ίδιο τρόπο ενημερώνει τους εξυπηρετητές του quorum. Το κάθε quorum, θυμάται αν έχει δώσει το δικαίωμα για πρόσβαση στο κρίσιμο τμήμα και δίνει άδεια μόνο αν δεν έχει δώσει. Όταν ενημερωθεί ότι ένας εξυπηρετητής έχει αποχωρήσει από το κρίσιμο τμήμα τότε μπορεί να ξαναδώσει άδεια. Η ιδιότητα της τομής του συστήματος απαρτίας διαβεβαιώνει την ακεραιότητα του κρίσιμου τμήματος.

Υπάρχουν ποικίλα παραδείγματα συστημάτων απαρτίας. Αυτά όμως που μας ενδιαφέρουν για τους σκοπούς της συγκεκριμένης εργασίας (αργότερα στα πειράματα), είναι το πλέγμα (matrix) [13], το σύστημα απαρτίας της πλειονότητας (majority) [13] και το crumbling walls [13]. Στα matrix συστήματα απαρτίας, οι εξυπηρετητές σχηματίζουν νοερά ένα ορθογώνιο πλέγμα και το κάθε quorum αποτελείται από μια γραμμή και μια στήλη του πλέγματος όπως φαίνεται στο Σχήμα 1.4 (α). Όλα τα quorums έχουν το ίδιο μέγεθος και το οποίο είναι ίσο με $2\sqrt{n}$-1. Με παρόμοιο τρόπο είναι και τα crumbling walls, με τη διαφορά ότι οι γραμμές μπορεί να είναι διαφορετικού μεγέθους και το κάθε quorum σχηματίζεται από μια ολόκληρη γραμμή

και ένα στοιχείο από τις υπόλοιπες γραμμές κάτω από αυτήν. Ένα παράδειγμα συστήματος απαρτίας crumbling walls απεικονίζεται στο Σχήμα 1.4 (γ). Οι πράσινοι κύκλοι είναι αυτοί που ανήκουν στο quorum. Το quorum όπως φαίνεται στο Σχήμα 1.4 (γ) αποτελείται από τη δεύτερη γραμμή και ένα στοιχείο από τις πιο κάτω γραμμές. Το καλύτερο crumbling walls σύστημα απαρτίας είναι το CWlog το οποίο έχει μικρά quorums του με μέγεθος O(lg n). Τέλος, στα συστήματα απαρτίας της πλειονότητας κάθε quorum πρέπει να αποτελείται από τουλάχιστον $\lceil (n+1) / 2 \rceil$ εξυπηρετητές Σχήμα 1.5 (β).



Σχήμα 1.4: (α) Σύστημα απαρτίας Matrix (β) Σύστημα απαρτίας Majority (γ) Σύστημα Απαρτίας Crumbling Walls

1.4 Κατανεμημένο Δίκτυο Planet Lab

Το Planet Lab [15] είναι ένα κατανεμημένο ερευνητικό υπερεπίπεδο δίκτυο το οποίο υποστηρίζει την ανάπτυξη νέων δικτυακών εφαρμογών και υπηρεσιών παγκοσμίου κλίμακας. Αποτελείται από 1006 σταθμούς, σε 484 διαφορετικές τοποθεσίες παγκοσμίως, σε ακαδημαϊκά και βιομηχανικά ιδρύματα. Οι πόροι του μοιράζονται σε μερίσματα (slices) τα οποία παρουσιάζονται ως ένα δίκτυο από εικονικές μηχανές. Κάθε μέρισμα μπορεί να έχει δικαίωμα χρήσης το πολύ μέχρι 32 σταθμούς, όπου ο καθένας μόνο ένα μέρος των πόρων (εύρος ζώνης δικτύου, μνήμη, χώρος δίσκου) κάθε σταθμού είναι δυνατόν να έχει στη διάθεση του. Από τη δημιουργία ενός μερίσματος υπάρχει κάποιο χρονικό περιθώριο χρήσης του και όταν ξεπεραστεί αυτό, λήγει αποδεσμεύοντας του πόρους του. Το χρονικό διάστημα πριν από τη λήξη ενός μερίσματος είναι συνήθως ένας μήνας, εν τούτοις μπορεί να ανανεωθεί απεριόριστες φορές. Ακόμη, κακόβουλες ή λογισμικά που βρίθουν από λάθη μπορούν να επηρεάσουν την επικοινωνία και την επίδοση άλλων μερισμάτων, γι αυτό επιβάλλονται αυστηροί όροι και προϋποθέσεις για την παροχή ασφάλειας στο Planet Lab. Η πρόσβαση στους σταθμούς μπορεί να γίνει μέσω SSH για ασφαλή, κρυπτογραφημένη επικοινωνία. Επιπλέον, οι σταθμοί μπορεί να επανεκκινήσουν ανά πάσα στιγμή, χωρίς να χαθούν οποιαδήποτε δεδομένα στο δίσκο, αλλά να είναι μη αξιόπιστοι. Ολοκληρώνοντας, να αναφέρουμε ότι το πανεπιστήμιο μας, το Πανεπιστήμιο Κύπρου, συμμετέχει στο Planet Lab με δύο σταθμούς.

1.5 Προηγούμενες Εργασίες

Ειπώθηκε ότι η ατομική μνήμη είναι μια από τις πιο σημαντικές αφηρημένες έννοιες στον κατανεμημένο προγραμματισμό. Υλοποιήσεις ατομικών αντικειμένων σε συστήματα ανταλλαγής μηνυμάτων με ανοχή σφαλμάτων, επιτρέπουν στις διεργασίες να μοιράζονται πληροφορίες με ακρίβεια και συνέπεια παρά την παρουσία των σφαλμάτων και της ασυγχρονίας. Μια τέτοια υλοποίηση της ατομικής μνήμης στο [1], δίνει μια λύση με ένα εγγραφέα και πολλούς αναγνώστες (SWMR), στην οποία κάθε αντικείμενο αντιγράφεται σε n κόμβους με ανταλλαγή μηνυμάτων. Σε αυτή τη λύση, λειτουργίες με πρόσβαση στη μνήμη εγγυούνται ότι θα τερματίσουν έως ότου ο

αριθμός των κόμβων που κατέρρευσαν είναι μικρότερος από n/2. Για την εκτέλεση μιας εγγραφής, ο εγγραφέας αυξάνει την τοπική του χρονοσφραγίδα κατά ένα. Ακολούθως στέλνει μήνυμα με τη χρονοσφραγίδα και την αντίστοιχη τιμή του ατομικού αντικειμένου σε όλες τις διεργασίες. Όταν πάρει απάντηση από την πλειοψηφία των διεργασιών τότε η εγγραφή ολοκληρώνεται. Για την εκτέλεση μιας λειτουργίας ανάγνωσης, η διεργασία του αναγνώστη στέλνει μηνύματα σε όλες τις διεργασίες για να μάθει την τιμή του ατομικού αντικειμένου. Όταν πάρει απάντηση από την πλειοψηφία των διεργασιών, βρίσκει τη μέγιστη χρονοσφραγίδα και την αντίστοιχη τιμή του αντικειμένου και τα στέλνει σε όλες τις διεργασίες όπως τον εγγραφέα. Με τον ίδιο τρόπο όταν πάρει απάντηση από την πλειοψηφία των διεργασιών η λειτουργία ανάγνωσης ολοκληρώνεται. Άρα, η γραφή των δεδομένων ολοκληρώνεται σε ένα γύρο επικοινωνίας (RTT) ενώ το διάβασμα χρειάζεται δύο γύρους επικοινωνίας, στους οποίους ο δεύτερος γύρος επικοινωνίας γράφει την τιμή που πήρε από το πρώτο. Ακολουθώντας αυτή την ιδέα επικράτησε η άποψη ότι το διάβασμα πρέπει να γράφει.

Παρόλα αυτά, μια πιο πρόσφατη εργασία [2], δημοσίευσε ότι αν ο αριθμός των αναγνωστών (readers) R περιοριστεί κατάλληλα ως προς τον αριθμό των αντιγράφων του αντικειμένου S και το μέγιστο αριθμό καταρρεύσεων t στο σύστημα ($R < S/t -2$), τότε διάβασμα με ένα RTT είναι εφικτό. Τέτοιες υλοποιήσεις όπως αυτή που δίνεται στο [2] αποκαλούνται γρήγορες (fast). Κατόπιν, στο [5] χαλάρωσαν τον περιορισμό του [2], και πρότειναν ημιγρήγορες (semifast) υλοποιήσεις με απεριόριστο αριθμό readers. Με άλλα λόγια πρότειναν υλοποιήσεις στις οποίες οι εγγραφές στο ατομικό αντικείμενο είναι πάντα γρήγορες ενώ οι αναγνώσεις του αντικειμένου μπορεί να χρειαστούν είτε ένα είτε δύο γύρους επικοινωνίας. Η δοκιμή του αλγορίθμου κάτω από ρεαλιστικές συνθήκες έδειξε ότι οι πλείστες αναγνώσεις του αντικειμένου χρειάζονται μόνο ένα RTT για να ολοκληρωθούν. Επιπρόσθετα οι αναγνώστες (readers) ομαδοποιούνται σε νοητούς κόμβους (virtual nodes). Τέλος, ο αλγόριθμος διατηρεί αυτή την ημιγρήγορη συμπεριφορά ωσότου ο αριθμός των νοητών κόμβων περιορίζεται κάτω του $S/t -2$.

Τα συστήματα απαρτίας όπως αναφέρθηκε είναι γνωστά μαθηματικά εργαλεία τα οποία προσφέρουν τα μέσα για το συντονισμό των επεξεργαστών στα κατανεμημένα συστήματα. Υλοποιήσεις λειτουργιών ανάγνωσης/εγγραφής, βασισμένες σε συστήματα

απαρτίας έχουν τη δυνατότητα να χρειάζονται πολύ λιγότερους κόμβους από ότι σε προηγούμενες γρήγορες [2] και ημιγρήγορες υλοποιήσεις [5].

Στο [6] δείχθηκε ότι η ιδιότητα της τομής μεταξύ όλων των συνόλων quorum, είναι αναγκαία και απαραίτητη συνθήκη για γρήγορες υλοποιήσεις βασισμένες σε συστήματα απαρτίας. Να σημειώσουμε ότι αυτή η ιδιότητα επιτρέπει γρήγορες υλοποιήσεις με συστήματα απαρτίας χωρίς περιορισμό στον αριθμό των αναγνωστών, σε αντίθεση με το [2]. Πάρα ταύτα, ικανοποιώντας αυτήν την ιδιότητα υποβαθμίζεται η ανοχή σφαλμάτων του συστήματος απαρτίας, εφόσον υπάρχει ένα μόνο σημείο σφάλματος (single point of failure). Ακόμη δείχθηκε ότι σε ημιγρήγορες υλοποιήσεις με συστήματα απαρτίας, μια από τις ιδιότητες των ημιγρήγορων υλοποιήσεων παραβιάζεται εάν δεν υπάρχει κοινή τομή. Συνεπώς, γρήγορες και ημιγρήγορες υλοποιήσεις βασισμένες σε συστήματα απαρτίας πάσχουν από ευρωστία.

Επίσης έγιναν υλοποιήσεις δυναμικής ατομικής μνήμης, όπως στο [12], με χρήση ανασχηματιζόμενων (reconfigurable) quorums, στις οποίες το σύνολο των αντιγράφων του αντικειμένου αλλάζει αυθαίρετα καθώς διεργασίες μπαίνουν και βγαίνουν από το σύστημα. Στα [3,4,7] δίνονται τελειοποιήσεις αυτής της υλοποίησης της δυναμικής μνήμης, οι οποίες βελτιώνουν την απόδοση του αλγορίθμου σε ρεαλιστικό περιβάλλον.

1.6 Κίνητρα

Στο [6] εισάγεται η έννοια του ασθενές-ημιγρήγορου (weak-semifast). Βασικά επιτυγχάνεται η ευρωστία, ανοχή στα σφάλματα, με κόστος ένα μέρος της ταχύτητας της υλοποίησης. Πιο συγκεκριμένα, είναι δυνατές περισσότερες από μια αργή λειτουργία ανάγνωσης (2 RTT) για κάθε λειτουργία γραφής. Για αυτή την περίπτωση δίνεται ένας αλγόριθμος [6], ο SLIQ, ο οποίος βασίζεται σε συστήματα απαρτίας και υλοποιείται με ασθενές-ημιγρήγορους ατομικούς καταχωρητές. Ο αλγόριθμος αυτός χρησιμοποιεί την έννοια των Quorum Views για την απεικόνιση όλων των πιθανών κατανεμημένων χρονοσφραγίδων του αντικειμένου που μπορεί συναντήσει μια λειτουργία ανάγνωσης. Αξίζει να σημειωθεί ότι αυτός ο αλγόριθμος επιτρέπει γρήγορες λειτουργίες διαβάσματος έστω και αν αυτές είναι ταυτόχρονες με άλλες λειτουργίες,

είτε ανάγνωσης είτε εγγραφής. Τελειώνοντας, να πούμε ότι ο αλγόριθμος αυτός προσομοιώθηκε με το εργαλείο NS-2 και τα αποτελέσματα του έδειξαν ότι υπό κάποιες συνθήκες, ένα ποσοστό μικρότερο του 13% των λειτουργιών ανάγνωσης είναι αργές. Άρα οι πλείστες λειτουργίες χρειάζονται ένα μόνο RTT.

1.7 Σκοπός Διπλωματικής Εργασίας

Σκοπός της παρούσας διπλωματικής εργασίας είναι η υλοποίηση και η πειραματική αξιολόγηση αλγόριθμου ατομικών αντικειμένων γραφής/ανάγνωσης και πιο συγκεκριμένα του αλγόριθμου ασθενές-ημιγρήγορου SLIQ, όπως περιγράφεται στο [6]. Ο βασικότερος στόχος είναι να εκτιμηθεί η συμπεριφορά του αλγόριθμου σε ρεαλιστικό ασύγχρονο δίκτυο και κάτω από την παρουσία σφαλμάτων. Εφόσον το πανεπιστήμιο μας έχει πρόσβαση στο Planet Lab, η πειραματική αξιολόγηση του αλγόριθμου πραγματοποιήθηκε στο κατανεμημένο αυτό δίκτυο που παρέχει τις συνθήκες που απαιτούνται για την αξιολόγηση του αλγόριθμου SLIQ, την ασυγχρονία και την παρουσία σφαλμάτων δικτύου. Στη συνέχεια σχολιάζεται και συγκρίνεται η απόδοση του αλγορίθμου σε πραγματικό δίκτυο, με την απόδοση του στις προσομοιώσεις στο [6].

1.8 Μεθοδολογία

Η μεθοδολογία που ακολουθήθηκε για να ολοκληρωθεί αυτή η εργασία είναι η εξής:

1. Μελέτη άρθρων και εργασιών, κυρίως για τον αλγόριθμο SLIQ [6] και προηγούμενων αλγόριθμων για την γραφή και ανάγνωση ατομικών αντικειμένων, έτσι ώστε να κατανοηθεί πλήρες το θέμα. Για τον ίδιο λόγο ήταν αναγκαία η μελέτη του υπόβαθρου, για εισαγωγή στον κατανεμημένο υπολογισμό και τα θέματα που προκύπτουν σχετικά με την κατανεμημένη μνήμη.
2. Εκμάθηση προγραμματισμού με υποδοχές ροής που εξυπηρετεί στην επικοινωνία διεργασιών σε ένα κατανεμημένο σύστημα, όπως αυτό που προϋποθέτει ο SLIQ. Έγινε ακόμη η υλοποίηση μικρών προγραμματιστικών

παραδειγμάτων για την καλύτερη κατανόηση προγραμματισμού με υποδοχές. Αυτά τα παραδείγματα υπήρξαν η βάση αργότερα για την υλοποίηση της επικοινωνίας μεταξύ των διεργασιών του αλγορίθμου.

3. Υλοποίηση του αλγόριθμου όπως περιγράφεται στο [6] σε γλώσσα C κάνοντας χρήση προγραμματισμού υποδοχών ροής. Η υλοποίηση και η αποσφαλμάτωση έγινε πρώτα στο τοπικό δίκτυο του Πανεπιστημίου Κύπρου και αργότερα στο κατανεμημένο δίκτυο Planet Lab.
4. Διερεύνηση του κατανεμημένου συστήματος Planet Lab και εκμάθηση του τρόπου λειτουργίας του.
5. Προτού αρχίσει η διαδικασία για την πειραματική αξιολόγηση του αλγορίθμου στο Planet Lab, υλοποιήθηκαν κάποια αρχεία εντολών (scripts) για την εξυπηρέτηση αυτού του σκοπού. Στη συνέχεια, δοκιμάστηκε ο αλγόριθμος κάτω από διαφορετικά σενάρια και καταγράφηκε η απόδοση του. Μετά από την αποκόμιση των αποτελεσμάτων έγινε η σύγκριση με τα αποτελέσματα της προσομοίωσης του αλγόριθμου και εξάχθηκαν τα γενικά συμπεράσματα.

Κεφάλαιο 2

Μοντέλο και Περιγραφή Αλγορίθμου

2.1 Μοντέλο και Ατομικότητα στον Αλγόριθμο SLIQ

Ο αλγόριθμος SLIQ [6] ανήκει στο ασύγχρονο μοντέλο ανταλλαγής μηνυμάτων με ένα γραφέα και πολλαπλούς αναγνώστες (SWMR). Το μοντέλο συνέπειας μνήμης που επιβάλει είναι η ατομική μνήμη. Το σύστημα αποτελείται από τέσσερα διαφορετικά σύνολα διεργασιών: μια διεργασία w που είναι ο εγγραφέας, ένα σύνολο από R αναγνώστες με μοναδικές ταυτότητες από το σύνολο $R=\{r_1, \ldots, r_R\}$ και ένα σύνολο από S εξυπηρετητές με μοναδικές ταυτότητες από το σύνολο $S=\{s_1, \ldots, s_S\}$. Οποιοσδήποτε συνδυασμός των διεργασιών είναι δυνατόν να καταρρεύσει ανά πάσα στιγμή της εκτέλεσης του αλγόριθμου, εκτός από ενός quorum. Όλες οι διεργασίες στο R και w έχουν επικοινωνία μέσω ανταλλαγής μηνυμάτων μόνο με το σύνολο S και αντίστροφα, οι διεργασίες στο S έχουν επικοινωνία μόνο με την w και R. Δεν υπάρχει οποιαδήποτε επικοινωνία μεταξύ των διεργασιών σε ένα σύνολο (R ή S) ούτε μεταξύ της w και οποιασδήποτε διεργασίας $r_i \in R$.

Αυτό που επιδιώκει ο αλγόριθμος SLIQ είναι να υλοποιήσει ένα αλγόριθμο για γραφή/ανάγνωση ατομικού αντικειμένου σε ένα σύστημα ανταλλαγής μηνυμάτων, αντιγράφοντας την τιμή του αντικειμένου σε όλους τους εξυπηρετητές. Κάθε αντίγραφο αποτελείται από την τιμή v και μια αντίστοιχη χρονοσφραγίδα (timestamp) Ts.

Η εφαρμογή πελάτης (client) στη διεργασία p μπορεί να αιτήσει μια λειτουργία ανάγνωσης ρ του ατομικού αντικειμένου x αν το p $\in$ R. Ομοίως, ο πελάτης ζητά μια λειτουργία εγγραφής γ αν η διεργασία p είναι ο εγγραφέας w. Το βήμα με τη λειτουργία ανάγνωσης ή εγγραφής, η αίτηση του πελάτη, ονομάζεται βήμα επίκλησης (invocation step) και το βήμα με την απάντηση readack ή writeack, βήμα απόκρισης(response step). Αν για μια λειτουργία (ρ ή γ) συντελέστηκε η αίτηση, το βήμα επίκλησης, αλλά όχι το βήμα απόκρισης τότε η λειτουργία είναι ανολοκλήρωτη. Σε αντίθετη περίπτωση, που συντελείται και το βήμα επίλυσης μετά το βήμα απόκρισης της λειτουργίας, η λειτουργία θεωρείται ολοκληρωμένη. Οι αιτήσεις του πελάτη είναι δομημένες με τέτοιο τρόπο ώστε να μην αιτήσει μια ενέργεια εγγραφής ή ανάγνωσης στο αντικείμενο x αν δεν έχει πάρει ήδη απάντηση από την τελευταία ενέργεια που αίτησε στο x. Ο αλγόριθμος υποθέτει ένα μόνο ατομικό καταχωρητή αντικειμένου. Για την απόκτηση ολόκληρης της ατομικής κοινής μνήμης, πρέπει να γίνει η σύνθεση πολλών υλοποιήσεων ατομικών καταχωρητών.

Σε μια εκτέλεση λέμε ότι μια λειτουργία (εγγραφής ή ανάγνωσης) λ_1 προηγείται μιας δεύτερης λ_2 ή η λ_2 προηγείται της λ_1, αν το βήμα απόκρισης της λ_1 προηγείται του βήματος επίκλησης της λ_2(συμβολίζεται το $\lambda_1 \rightarrow \lambda_2$. Δύο λειτουργίες είναι ταυτόχρονες αν καμιά από τις δύο δεν προηγείται της άλλης.

Η ορθότητα της υλοποίησης ενός ατομικού αντικειμένου προσδιορίζεται υπό τους όρους του τερματισμού και της ατομικότητας. Στην ιδιότητα του τερματισμού οποιαδήποτε λειτουργία εμπλεκόμενη από συνεπής διεργασία στο τέλος αποπερατώνεται. Η ατομικότητα ορίζεται [11] όπως παρακάτω:
Δεδομένου ενός συνόλου Λ από τις ολοκληρωμένες λειτουργίες σε μια εκτέλεση, υπάρχει μια μερική διάταξη $<$ των λειτουργιών του Λ ικανοποιώντας τα πιο κάτω:

1) Για οποιαδήποτε λειτουργία $\lambda \in \Lambda$, υπάρχουν πολλές πεπερασμένες λειτουργίες λ' τέτοιες ώστε $\lambda' < \lambda$.
2) Αν η λειτουργία λ_1 προηγείται της λ_2 στο Λ τότε δεν υπάρχει περίπτωση του $\lambda_2 < \lambda_1$.
3) Αν η λ είναι μια λειτουργία εγγραφής και λ' οποιαδήποτε άλλη λειτουργία στο Λ, τότε ισχύει είτε $\lambda < \lambda'$ είτε $\lambda' < \lambda$.

4) Η τιμή που επιστρέφεται από τη λειτουργία ανάγνωσης είναι η τιμή που έγγραψε η τελευταία λειτουργία εγγραφής σύμφωνα με $<$ (ή $\perp$ αν δεν υπάρχει προηγούμενη λειτουργία εγγραφής).

Μια ασθενές-ημιγρήγορη υλοποίηση ορίζεται ως η υλοποίηση στην οποία η μερική διάταξη των λειτουργιών της ισχύουν τα τέσσερα πιο πάνω. Επιπλέον σε μια ασθενές-ημιγρηγορη υλοποίηση όλες οι εγγραφές στο ατομικό αντικείμενο είναι γρήγορες, δηλαδή ολοκληρώνονται σε ένα γύρο επικοινωνίας. Τέλος οι αναγνώσεις του αντικειμένου μπορεί να είναι αργές, δηλαδή να ολοκληρωθούν σε δύο γύρους επικοινωνίας.

2.2 Συστήματα απαρτίας

Ο αλγόριθμος που περιγράφεται στη συνέχεια του κεφαλαίου είναι βασισμένος σε quorum συστήματα. Το σύστημα απαρτίας είναι τα αντίγραφα του ατομικού αντικειμένου, με άλλα λόγια τις διεργασίες των εξυπηρετητών εφόσον κρατούν τα αντίγραφα των αντικειμένων στην τοπική τους μνήμη. Οι υπόλοιπες διεργασίες του συστήματος, ο εγγραφέας και οι αναγνώστες, είναι ενημερωμένες για τη δομή του συστήματος απαρτίας.

Μια διεργασία είναι επιρρεπής σε σφάλματα (faulty) αν καταρρεύσει σε οποιαδήποτε φάση εκτέλεσης του αλγόριθμου. Ένα quorum καλείται επιρρεπές σε σφάλματα αν μια από τις διεργασίες του, εξυπηρετητές, Π είναι επιρρεπής σε σφάλματα. Ο αλγόριθμος SLIQ προϋποθέτει ότι ένα τουλάχιστον quorum δεν είναι επιρρεπές σε σφάλματα. Συνεπώς, εξασφαλίζεται η ομαλή λειτουργία του αλγόριθμου. Λόγω της ιδιότητας της τομής που προαναφέρθηκε στα συστήματα απαρτίας, είναι αρκετό οι διεργασίες του εγγραφέα και του αναγνώστη να πάρουν απάντηση μόνο από ένα quorum για να ολοκληρώσουν την εκτέλεση τους. Να προσθέσουμε ότι, αν οι διεργασίες περίμεναν απάντηση από περισσότερα του ενός quorums, τότε θα ήταν πιθανό η λειτουργία να μην τερματίσει εφόσον μόνο ένα quorum εγγυάται ότι βρίσκεται σε εκτέλεση. Ολοκληρώνοντας, να πούμε ότι το συστήματα απαρτίας δεν είναι δυναμικό, αλλά παραμένει το ίδιο από την αρχή μέχρι το τέλος του αλγόριθμου.

2.3 Αλγόριθμος SLIQ

Πριν περιγράψουμε τον αλγόριθμο SLIQ θα εισάγουμε την έννοια των Quorum Views [6] τα οποία περιγράφουν τις καταστάσεις με τις οποίες μπορεί να βρεθεί αντιμέτωπος ένας αναγνώστης κατά τον πρώτο κύκλο επικοινωνίας. Ακολούθως, θα δούμε πως χρησιμοποιείται αυτή η έννοια στην ασθενές-ημιγρήγορη υλοποίηση ατομικών καταχωρητών σχετικά με το πότε θα εκτελέσει μια λειτουργία ανάγνωσης δεύτερο κύκλο επικοινωνίας. Να θυμίσουμε μια ασθενές-ημιγρήγορη υλοποίηση επιτρέπει περισσότερες της μιας αργής ανάγνωσης για κάθε εγγραφή, ενώ οι εγγραφές είναι πάντα γρήγορες.

2.3.1 Quorum Views

Το Quorum View αναφέρεται στην κατανομή της μέγιστης χρονοσφραγίδας μέσα στο σύστημα, από την οπτική εικόνα του αναγνώστη στον πρώτο γύρο επικοινωνίας. Ας θεωρήσουμε ότι μια λειτουργία ανάγνωσης ρ_ι έρχεται σε επικοινωνία μόνο με το quorum Q_i κατά τον πρώτο γύρω επικοινωνίας της και έστω maxTs η μέγιστη χρονοσφραγίδα που συναντά η ρ_ι κατά αυτόν το γύρο. Κάθε μέλος $s \in Q_i$ απαντά με χρονοσφραγίδα s.ts στην ρ_ι. Τα quorum Views ορίζονται με της τρεις ακόλουθες περιπτώσεις για την ρ_ι:

1. [qView(1)] $\forall s \in Q_i$: s.ts = maxTs
2. [qView(2)] $\forall Q_j \in \mathbb{Q}$, $i \neq j$, $\exists A \subseteq Q_i \cap Q_j$, τέτοιο ώστε $A \neq \varnothing$ και $\forall s \in A$: s.ts < maxTs
3. [qView(3)] $\exists Q_j \in \mathbb{Q}$, $i \neq j$ και $\forall s \in Q_i \cap Q_j$: s.ts = maxTs

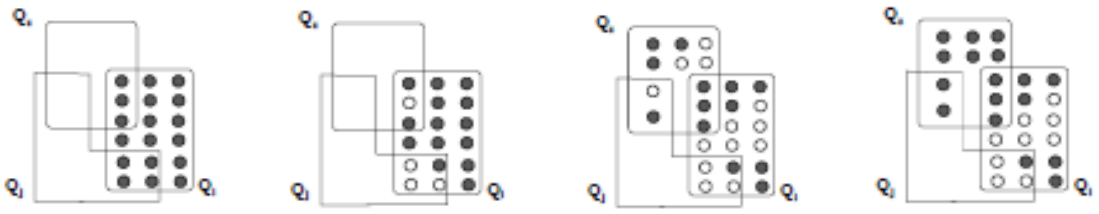
Από τα πιο πάνω Quorum Views μπορούμε να συμπεράνουμε την κατάσταση της λειτουργίας εγγραφής (ολοκληρωμένης ή μη) χρησιμοποιώντας τη μέγιστη χρονοσφραγίδα maxTs. Στο Σχήμα 2.1 απεικονίζονται οι τρεις περιπτώσεις Quorum Views. Οι σκούροι κόμβοι αντιπροσωπεύουν τη μέγιστη χρονοσφραγίδα του συστήματος και οι άδειοι παλαιότερες χρονικές σφραγίδες. Να θυμίσουμε ότι οι διεργασίες δεν περιμένουν αποκρίσεις από περισσότερα του ενός quorums. Έτσι κατά

αυτόν τον τρόπο όταν ένα quorum δίνει το ίδιο maxTs, Σχήμα 2.1(α), υπονοεί την πιθανή ολοκλήρωση της λειτουργίας εγγραφής.

Να επισημάνουμε ότι αν ένα quorum περιέχει το maxTs τότε και οι κόμβοι οποιασδήποτε τομής του θα έχουν το maxTs. Ως εκ τούτου, παρατηρώντας ένα μέρος των κόμβων κάθε τομής του Q_i (Σχήμα 2.1 (β)) να έχει παλαιότερη χρονοσφραγίδα συμπεραίνουμε ότι η λειτουργία εγγραφής που διαδίδει το MaxTs δεν έχει ολοκληρωθεί ακόμη.

Το Quorum View(3) δίνει ανεπαρκείς πληροφορίες σχετικά με την κατάσταση της λειτουργίας εγγραφής. Το Σχήμα 2.1 (γ) και το Σχήμα 2.1 (δ) δείχνουν αύτη την περίπτωση. Στην πρώτη μια ανολοκλήρωτη λειτουργία εγγραφής διαδίδει το MaxTs στην τομή (σκούροι κόμβοι) ενώ στη δεύτερη ολοκληρώνεται η λειτουργία εγγραφής παίρνοντας απαντήσεις από το Q_z. Να σημειώσουμε ότι αν μια λειτουργία ανάγνωσης ρ επικοινωνεί μόνο με το Q_i δε θα είναι σε θέση να διαχωρίσει τις δύο περιπτώσεις. Γι' αυτό αν μια λειτουργία βρει σε μια τομή $Q_i \cap Q_j$ όλοι οι κόμβοι να έχουν το MaxTs, τότε η λειτουργία εγγραφής:

1. έχει ολοκληρωθεί επικοινωνώντας με το Q_z
2. δεν έχει ολοκληρωθεί και έχει επικοινωνήσει με ένα σύνολο εξυπηρετητών B τέτοιων ώστε $Q_i \cap Q_j \subseteq B$ και $\forall Q_j \in \mathbb{Q}$, $Q_j \not\subseteq B$.



Σχήμα 2.1: Πιθανές κατανομές της μέγιστης χρονοσφραγίδας[6].

2.3.2 Περιγραφή Αλγορίθμου

Πριν περιγράψουμε πιο αναλυτικά τον αλγόριθμο κάποια γενικά στοιχεία που ισχύουν σε όλες τις διεργασίες. Κάθε μήνυμα που στέλνεται στο δίκτυο είναι της μορφής:

<*Τύπος μηνύματος*, *χρονοσφραγίδα*, *τιμή αντικειμένου*, *προηγούμενη τιμή αντικειμένου*, *αριθμός αίτησης*>, όπου ο τύπος μπορεί να είναι μόνο ένα από τα 6: *WRITE*, *READ*, *INFORM*, *WRITEACK*, *READACK*, *INFORMACK*.

Ο εγγραφέας και ένας αναγνώστης πριν στείλουν μια αίτηση (*READ* ή *WRITE*) στους εξυπηρετητές, αυξάνουν το μετρητή αιτήσεων κατά ένα. Όταν πάρουν μια απάντηση από έναν εξυπηρετητή ελέγχουν καταρχάς αν ο μετρητής στο μήνυμα είναι ο ίδιος με τον τρέχων μετρητή. Με άλλα λόγια ελέγχει αν η απάντηση που έλαβε ήταν για την τελευταία αίτηση που έστειλε και όχι για κάποια παλαιότερη. Αν ναι, επεξεργάζεται περαιτέρω το μήνυμα αλλιώς το αγνοεί.

Ας δούμε τώρα ξεχωριστά και με λεπτομέρεια τη λειτουργία της κάθε μιας από τις τρεις διεργασίες.

Εγγραφέας(Writer)

Ο εγγραφέας είναι αυτός που εκτελεί κάθε λειτουργία εγγραφής. Όταν εκτελείται μια λειτουργία εγγραφής, ο εγγραφέας στέλνει σε όλους του εξυπηρετητές μήνυμα της μορφής που αναφέρθηκε. Ο εγγραφέας στέλνει μηνύματα μόνο του τύπου *WRITE*, δηλώνοντας την εγγραφή της νέας τιμής του αντικειμένου στα αντίγραφα του αντικειμένου που βρίσκονται στους εξυπηρετητές. Μόλις πάρει απάντηση από ένα ολόκληρο quorum η εγγραφή ολοκληρώνεται. Τα μηνύματα που είναι δυνατό να λάβει είναι μόνο του τύπου *WRITEACK*, εφόσον στέλνει μόνο *WRITE* αιτήσεις. Εδώ να σημειώσουμε ότι ο εγγραφέας ολοκληρώνεται σε ένα γύρο επικοινωνίας.

Πριν σταλεί η αίτηση για εγγραφή στους εξυπηρετητές αυξάνεται η χρονοσφραγίδα κατά ένα. Η χρονοσφραγίδα αναγνωρίζει μοναδικά κάποια εγγραφή του αντικειμένου αφού μόνο ένας εγγραφέας υπάρχει στο σύστημα.

Αναγνώστης (Reader)

Κατά παρόμοιο τρόπο με τη λειτουργία της εγγραφής και μια λειτουργία ανάγνωσης στέλνει μήνυμα σε όλους τους εξυπηρετητές. Ένας αναγνώστης στέλνει πρώτα μηνύματα του τύπου *READ*, για να μάθει την τιμή του αντικειμένου, και λαμβάνει αντίστοιχα μηνύματα *READACK*. Μόλις λάβει απάντηση από ένα ολόκληρο quorum εξετάζει την κατανομή του MaxTs μέσα στο σύστημα για να βρει τον τύπο του quorum View. Αν βρει Quorum View(1) ή Quorum View(2) τότε ο αναγνώστης τερματίζει στον πρώτο κύκλο επικοινωνίας και επιστρέφει το MaxTs και MaxTs – 1 αντίστοιχα. Αν είναι Quorum View(3) τότε συνεχίζει και με δεύτερο γύρο επικοινωνίας, στον οποίο μεταδίδει τη μέγιστη χρονοσφραγίδα και την αντίστοιχη τιμή του αντικειμένου στο σύστημα όπως το γραφέα. Σε αυτή την περίπτωση τα μηνύματα που στέλνει είναι του τύπου *INFORM* και οι αντίστοιχες απαντήσεις που λαμβάνει από τους εξυπηρετητές, του τύπου *INFORMACK* . Όταν πάρει απάντηση από ένα ολόκληρο quorum τότε τερματίζει και επιστρέφει το MsxTs. Μια λειτουργία ανάγνωσης σε αντίθεση με τη λειτουργία εγγραφής μπορεί να ολοκληρωθεί είτε σε ένα γύρο επικοινωνίας είτε σε δύο.

Εξυπηρετητής (Server)

Τελειώνοντας να πούμε ότι οι εξυπηρετητές έχουν ένα παθητικό ρόλο. Παίρνουν μηνύματα, ανάλογα με τα περιεχόμενα του μηνύματος ενημερώνουν τα προσωπικά τους αντίγραφα και απαντούν στα μηνύματα.

Τα μηνύματα που λαμβάνουν ανήκουν σε ένα από τους τύπους *WRITE*, *READ*, *INFORM* και τα μηνύματα που στέλνουν σε ένα από τους τύπους *WRITEACK*, *READACK*,*INFORACK* αντίστοιχα. Για κάθε διεργασία κρατούν τον αριθμό της τελευταίας αίτησης που έλαβαν από αυτή. Έτσι με το που φθάνει ένα νέο μήνυμα ελέγχουν αν ο αριθμός είναι μεταγενέστερος από αυτόν της τελευταίας αίτησης. Αν ναι τότε προχωρεί σε περαιτέρω επεξεργασία του μηνύματος, αλλιώς αγνοεί το μήνυμα και δεν στέλνει απάντηση. Έτσι δε στέλνονται μηνύματα στο δίκτυο αχρείαστα αποφεύγοντας όσο το δυνατό γίνεται τη συμφόρηση του δικτύου.

Κεφάλαιο 3

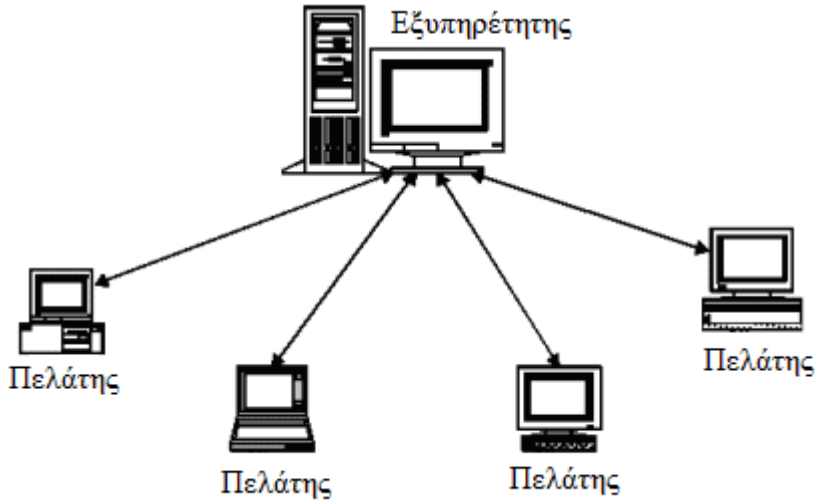
Υλοποίηση Αλγορίθμου

Όπως ανάφερα σε προηγούμενη ενότητα, σκοπός της εργασίας είναι η υλοποίηση του αλγορίθμου SLIQ και ακολούθως η αξιολόγηση του σε ένα ρεαλιστικό δίκτυο, το Planet Lab. Για την υλοποίηση του συστήματος αυτού ακολουθήθηκε το μοντέλο πελάτη-εξυπηρετητή (client-server) με προγραμματισμό υποδοχών στη γλώσσα C (C socket programming). Προτού όμως περιγράψω πιο λεπτομερώς την υλοποίηση του αλγορίθμου, πρέπει πρώτα να προσδιορίσω και να εξηγήσω τα μέσα που χρησιμοποιήθηκαν για την ανάπτυξη του. Στην συνέχεια παρουσιάζονται τα μοντέλα και οι βιβλιοθήκες που χρησιμοποιήθηκαν για την επιτυχή δημιουργία του συστήματος.

3.1 Μοντέλο Πελάτη Εξυπηρετητή

Το μοντέλο πελάτη εξυπηρετητή απευθύνεται στη δικτύωση υπολογιστών διαχωρίζοντας δύο συστήματα, αυτό του πελάτη και αυτό του εξυπηρετητή, το καθένα σχεδιασμένο για συγκεκριμένο σκοπό. Μοντέλα πελάτη εξυπηρετητή έχουν ευρεία χρήση στο Διαδίκτυο αλλά και σε τοπικά δίκτυα (LAN). Μολονότι λογισμικά πάνω στον ίδιο υπολογιστή μπορούν να εφαρμόσουν την ιδέα του πελάτη εξυπηρετητή

παρουσιάζει περισσότερο ενδιαφέρον όταν εφαρμόζεται σε κάποιο δίκτυο. Σε ένα δίκτυο το μοντέλο αυτό είναι πρόσφορο για την διασύνδεση προγραμμάτων που είναι κατανεμημένα σε διαφορετικές τοποθεσίες. Ένα παράδειγμα πελάτη-εξυπηρετητή είναι ο φυλλομετρητής Διαδικτύου (Web Browser) και ο διακομιστής HTTP (Web Server).



Σχήμα 3.1: Μοντέλο πελάτη εξυπηρτητή

Η βασική ιδέα αυτού του μοντέλου είναι ότι ο πελάτης εγκαθιδρύει τη σύνδεση με τον εξυπηρετητή και στη συνέχεια κάνει αιτήσεις με βάση της υπηρεσίας που υποστηρίζει. Από την άλλη πλευρά, ο εξυπηρετητής αποδέχεται αιτήσεις από τους πελάτες και αφού τις επεξεργαστεί απαντά στους πελάτες επιστρέφοντας τους τις αιτούμενες πληροφορίες. Το Σχήμα 3.1 απεικονίζει την επικοινωνία σε ένα μοντέλο πελάτη εξυπηρετητή.

Οι εξυπηρετητές κατηγοριοποιούνται σε δύο κατηγορίες [17], τους σειριακούς (iterative) και τους ταυτόχρονους(concurrent). Ένας εξυπηρετητής στην πρώτη κατηγορία ακολουθεί τα παρακάτω βήματα:

1. Περιμένει να φτάσει κάποια αίτηση από πελάτη.
2. Επεξεργάζεται την αίτηση του πελάτη.

3. Απαντά στην αίτηση.
4. Συνεχίζει από το βήμα 1.

Αφετέρου ο ταυτόχρονος εξυπηρετητής εκτελεί τα πιο κάτω βήματα:

1. Περιμένει να καταφθάσει κάποια αίτηση.
2. Αναθέτει αυτήν την αίτηση σε ένα άλλο εξυπηρετητή για να τη χειριστεί. Ο άλλος εξυπηρετητής μπορεί να είναι μια νέα διεργασία στο σύστημα ή κάποιο νήμα(thread), ανάλογα με την υλοποίηση της εφαρμογής. Από αυτό το σημείο και έπειτα ο νέος εξυπηρετητής χειρίζεται τη σύνδεση με τον πελάτη μέχρις ότου να κλείσει και τότε τερματίζει.
3. Συνεχίζει από το βήμα 1.

Το πλεονέκτημα στο τελευταίο είναι ότι ο εξυπηρετητής αναπαράγει άλλους εξυπηρετητές και αναθέτει σε αυτούς το χειρισμό των αιτήσεων από τους πελάτες. Κατά συνέπεια πολλαπλοί πελάτες εξυπηρετούνται ταυτόχρονα και πιο γρήγορα. Κατά γενικό κανόνα οι TCP εξυπηρετητές είναι ταυτόχρονοι ενώ οι UDP επαναληπτικοί.

3.2 Πρωτόκολλο Ρύθμισης Μεταβιβάσεων

Εφόσον έχουμε δει τον τρόπο λειτουργίας του μοντέλου πελάτη-εξυπηρετητή ας δούμε συνοπτικά τι είναι το πρωτόκολλο ρύθμισης μεταβιβάσεων [16]. είναι το αξιόπιστο, συνδεσιστρεφές (connection-oriented) πρωτόκολλο μεταφοράς του επιπέδου μεταφοράς του Διαδικτύου.

Καλείται συνδεσιστρεφές, επειδή πριν να μπορέσει να αρχίσει να στέλνει δεδομένα μια διεργασία εφαρμογής σε μια άλλη, οι δύο διεργασίες πρέπει να κάνουν «χειραψία» μεταξύ τους – δηλαδή πρέπει να στείλουν κάποια αρχικά τμήματα η μια στην άλλη για να καθορίσουν τις παραμέτρους της επικείμενης μεταφοράς δεδομένων.

Συνάμα μια «σύνδεση» TCP παρέχει αμφίδρομη μεταφορά δεδομένων. Αν υπάρχει μια σύνδεση TCP μεταξύ ανάμεσα στη διεργασία Α σε έναν υπολογιστή υπηρεσίας και στη διεργασία Β σε έναν άλλο υπολογιστή υπηρεσίας, τότε δεδομένα εφαρμογής μπορούν να ρέουν από την Α στην Β ταυτόχρονα με δεδομένα εφαρμογής που ρέουν από την Β στην Α. Μια σύνδεση TCP, είναι επίσης πάντα από σημείο προς σημείο, ανάμεσα σε ένα μόνο αποστολέα και ένα μόνο δέκτη.

Από τη σκοπιά της εφαρμογής η σύνδεση TCP είναι μια απευθείας εικονική σωλήνωση ανάμεσα στην υποδοχή ροής του πελάτη και την υποδοχή ροής του εξυπηρετητή. Η διεργασία πελάτη μπορεί να στείλει αυθαίρετα bytes μέσα από αυτήν την υποδοχή ροής και το TCP εγγυάται ότι η διεργασία εξυπηρετητή θα δεχτεί κάθε byte με τη σειρά την οποία στέλνεται. Παρόμοια και η διεργασία εξυπηρετητή, όπως μπορεί να δεχτεί δεδομένα έτσι μπορεί να στείλει δεδομένα μέσω της θύρας υποδοχής στη διεργασία πελάτης.

3.3 Προγραμματισμός με Υποδοχές Ροής

Σε αυτό το υποκεφάλαιο εξετάζουμε τους μηχανισμούς για την επικοινωνία διεργασιών που τρέχουν σε διαφορετικούς υπολογιστές δια μέσω δικτύου. Δεδομένου ότι για την αξιολόγηση του SLIQ επιβάλλεται οι διεργασίες να είναι κατανεμημένες γεωγραφικά και να επικοινωνούν μέσω δικτύου έγινε χρήση τέτοιων μηχανισμών και συγκεκριμένα προγραμματισμού με υποδοχές ροής. Οι συναρτήσεις που αναφέρονται αφορούν τη γλώσσα C που χρησιμοποιείται ακολούθως στην υλοποίηση.

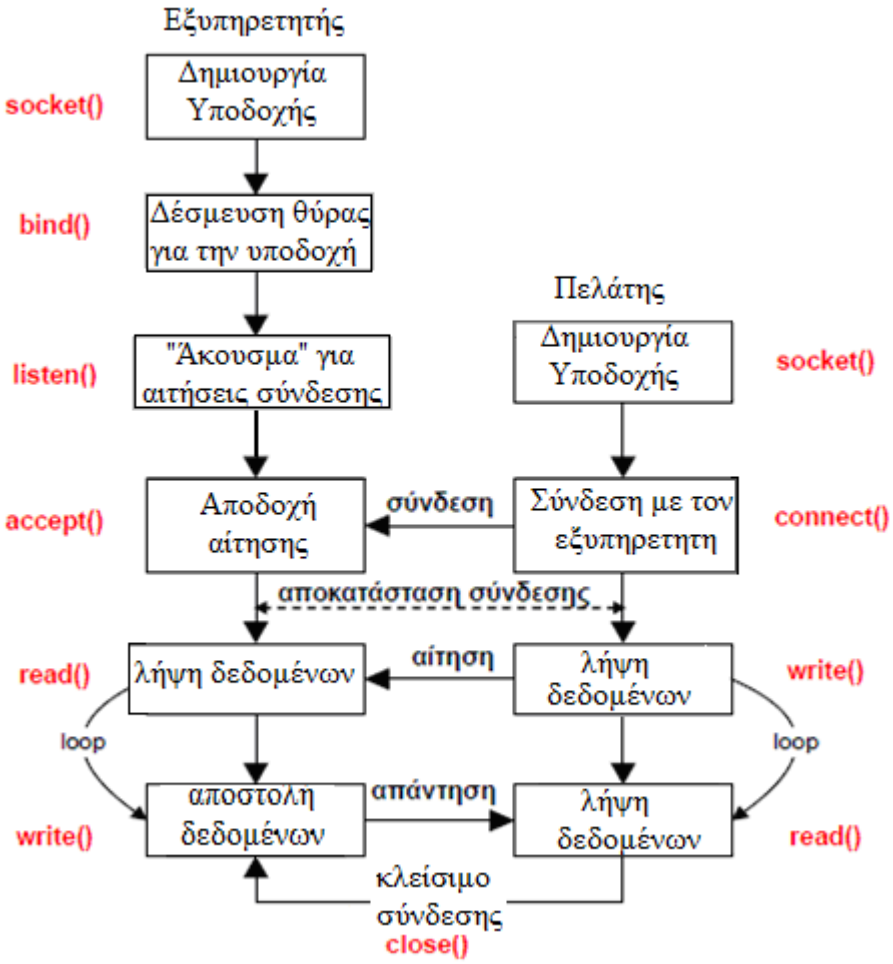
Να ανακαλέσουμε ότι πελάτης είναι η εφαρμογή η οποία ξεκινά τη σύνδεση και ο εξυπηρετητής αυτός που την αποδέχεται. Για να μπορέσει ο εξυπηρετητής να αντιδράσει στην αρχική επαφή του πελάτη, ο εξυπηρετητής πρέπει να είναι έτοιμος. Αυτό υπονοεί δύο πράγματα. Πρώτο, το πρόγραμμα εξυπηρετητή δε μπορεί να είναι σε νάρκη. Πρέπει να εκτελείται σαν μια διεργασία, πριν προσπαθήσει ο πελάτης να

εκκινήσει την επαφή. Δεύτερο το πρόγραμμα εξυπηρετητή πρέπει να έχει κάποιο είδος πόρτας (δηλαδή υποδοχή), που υποδέχεται κάποια αρχική επαφή από ένα πελάτη.

Με εκτελούμενη τη διεργασία του εξυπηρετητή, η διεργασία πελάτη μπορεί να εκκινήσει μια σύνδεση TCP προς τον εξυπηρετητή. Αυτό γίνεται στο πρόγραμμα πελάτη δημιουργώντας μια υποδοχή ροής (stream socket). Όταν ο πελάτης δημιουργήσει την υποδοχή ροής του, καθορίζει την διεύθυνση της διεργασίας εξυπηρετητή, δηλαδή την IP διεύθυνση του εξυπηρετητή και τον αριθμό θύρας της διεργασίας. Με τη δημιουργία της υποδοχής ροής, το TCP στον πελάτη εκκινεί μια τριμερή χειραψία και καθορίζει μια σύνδεση TCP με τον εξυπηρετητή. Η τριμερής χειραψία είναι εντελώς διάφανη στα προγράμματα πελάτη και εξυπηρετητή.

Κατά τη διάρκεια της τριμερούς χειραψίας, η διεργασία πελάτη «κτυπά» την πόρτα της διεργασίας εξυπηρετητή. Όταν ο εξυπηρετητής «ακούσει» το κτύπημα δημιουργεί μια νέα πόρτα (δηλαδή υποδοχή ροής), που αφοσιώνεται στο συγκεκριμένο πελάτη. Στην υλοποίηση η πόρτα είναι ένας περιγραφέας αρχείου (file descriptor) στην υποδοχή ροής, πάνω στη οποία ο εξυπηρετητής ακούει με τη συνάρτηση listen() για νέες συνδέσεις. Όταν μια νέα σύνδεση καταφθάνει, με τη συνάρτηση accept() δημιουργεί μια νέα πόρτα για τον πελάτη. Στο τέλος της χειραψίας υπάρχει μια TCP σύνδεση ανάμεσα στην υποδοχή ροής του πελάτη και την νέα υποδοχή ροής του εξυπηρετητή και οι δύο διεργασίες είναι έτοιμες για ανταλλαγή δεδομένων.

Η διαδικασία για την εγκαθίδρυση επικοινωνίας και την ανταλλαγή μηνυμάτων στο μοντέλο πελάτη-εξυπηρετητή απεικονίζεται στο Σχήμα 3.2 , συνοδευόμενη σε κάθε βήμα την αρμοστή συνάρτηση. Στη συνέχεια θα δούμε τεχνικές λεπτομέρειες όσον αφορά τον προγραμματισμό με υποδοχές και τις συναρτήσεις για κάθε βήμα.



Σχήμα 3.2: Βήματα για τη σύνδεση και επικοινωνία μεταξύ πελάτη και εξυπηρετητή.

3.3.1 Είδη Υποδοχών

Τα πιο συνηθισμένα είδη υποδοχών είναι οι υποδοχές ροής (stream sockets) και οι τηλεγραφικές υποδοχές (datagram sockets). Τα τελευταία καλούνται και connectionless sockets. Καλούνται έτσι επειδή δεν εγκαθιδρύουν την επικοινωνία με τον παραλήπτη πριν αρχίσουν να στέλνουν. Ακόμη τα πακέτα που στέλνονται μπορεί να φτάσουν αλλά πάλι και να μη φτάσουν. Αντίθετα οι υποδοχές ροής εγγυούνται αξιόπιστη διπλής ροής επικοινωνία. Με τη σειρά που θα στείλεις τα bytes μέσω της υποδοχής, με τη ίδια σειρά θα φτάσουν στον προορισμό τους και χωρίς λάθη. Για να το πετύχουν αυτό χρησιμοποιούν το πρωτόκολλο ρύθμισης μεταβιβάσεων που αναφέρθηκε. Είναι αυτά

που χρησιμοποιούνται και στην υλοποίηση του αλγορίθμου. Διότι ο αλγόριθμος μας απαιτεί αξιόπιστη μεταφορά των δεδομένων χωρίς χάσιμο πακέτων και σφάλματα.

3.3.2 Δημιουργία Υποδοχής

Οι εφαρμογές χρησιμοποιούν περιγραφείς υποδοχών για να έχουν πρόσβαση στις υποδοχές, όπως με ανάλογο τρόπο οι περιγραφείς αρχείων χρησιμοποιούνται για πρόσβαση αρχείων. Οι περιγραφείς υποδοχών υλοποιούνται ως περιγραφείς αρχείων στο UNIX σύστημα, γι αυτό διάφορες συναρτήσεις που σχετίζονται με περιγραφείς αρχείων, π.χ `read()`, `write()` δουλεύουν με ένα περιγραφέα υποδοχής. Για τη δημιουργία της υποδοχής καλείται η παρακάτω συνάρτηση:

```
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

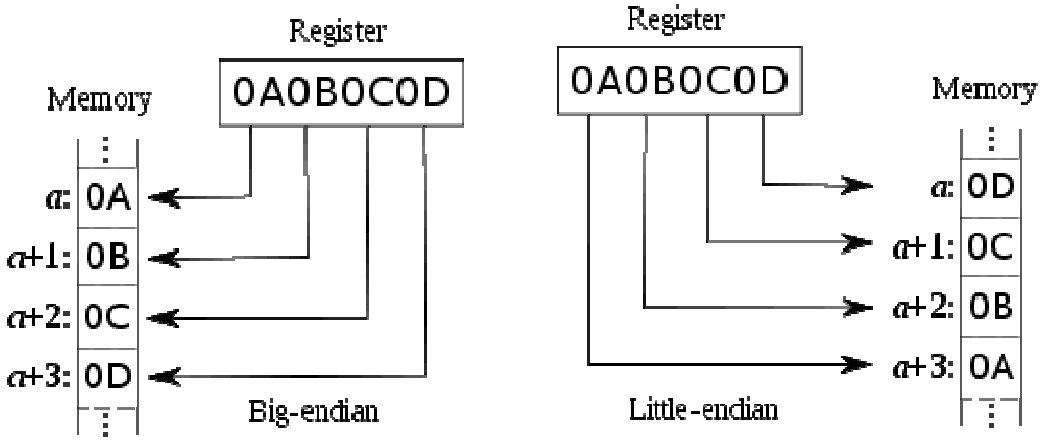
η οποία επιστρέφει το περιγραφέα του αρχείου ή -1 σε περίπτωση λάθους. Η παράμετρος `domain` προσδιορίζει τη φύση της επικοινωνίας. Στην περίπτωση μας θα είναι `AF_INET`, που προσδιορίζει το IPv4 internet domain. Η παράμετρος `type` περιγράφει τον τύπο της υποδοχής (`SOCK_STREAM`, `SOCK_DGRAM`). Η τελευταία παράμετρος, `protocol`, έχει συνήθως την τιμή 0, δηλώνοντας έτσι να χρησιμοποιηθεί το εξ' ορισμού πρωτόκολλο για τη συγκεκριμένη οικογένεια πεδίου υποδοχών και τύπο υποδοχών. Το προκαθορισμένο πρωτόκολλο για μια υποδοχή `SOCK_STREAM` και `AF_INET` πεδίο είναι το TCP.

3.3.3 Αναπαράσταση Διευθύνσεων

Πριν προχωρήσουμε σε βασικές λειτουργίες με υποδοχές, να δούμε πρώτα πως αναγνωρίζεται η διεργασία που επιθυμείται να εγκαθιδρυθεί επικοινωνία. Η διεύθυνση δικτύου της μηχανής βοηθά στην αναγνώριση του υπολογιστή και η θύρα (port) στην αναγνώριση συγκεκριμένης διεργασίας στον υπολογιστή.

Όταν δύο διεργασίες στον ίδιο υπολογιστή επικοινωνούν μεταξύ τους, δεν τίθεται θέμα ανησυχίας για τη σειρά των bytes. Η σειρά των bytes είναι χαρακτηριστικό της αρχιτεκτονικής του συστήματος που επιβάλλει πως θα φυλάγονται τα bytes στην μνήμη. Διαφορετικές αρχιτεκτονικές υπολογιστών αποθηκεύουν αριθμούς διαφορετικά. Το Σχήμα 3.3 δείχνει τους πιθανούς τρόπους.

Οι Little Endian αρχιτεκτονικές (π.χ.Intel, FreeBSD, Linux) αποθηκεύουν το Least Significant Byte (LSB) πρώτα, δηλαδή στη χαμηλότερη byte διεύθυνση. Από την άλλη οι Big Endian αρχιτεκτονικές (π.χ.Sun, SPARC, Solaris) αποθηκεύουν το Most Significant Byte (MSB) στη χαμηλότερη byte διεύθυνση. Η TCP/IP σουίτα πρωτοκόλλων χρησιμοποιεί τη big-endian byte ακολουθία.



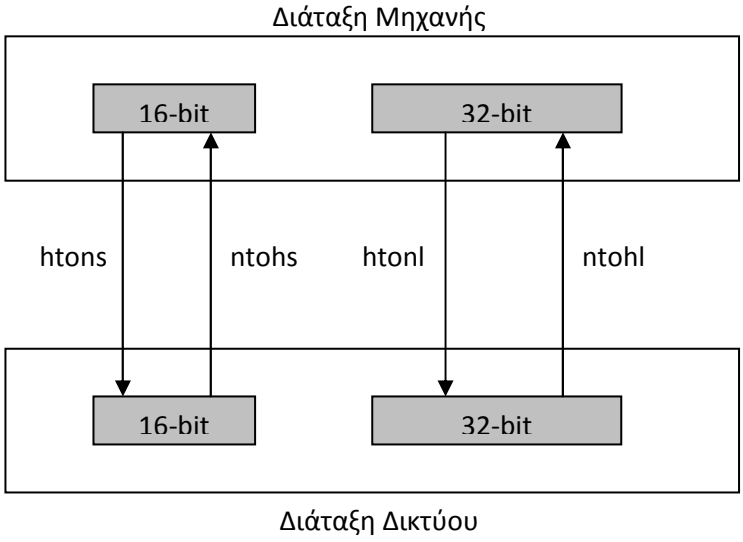
Σχήμα 3.3: Big-endian και Little-endian byte διάταξη

Οι IP διευθύνσεις και αριθμοί θυρών παρουσιάζονται σε bytes ακολουθία με διάταξη «δικτύου». Επομένως, οι εφαρμογές χρειάζεται να μεταφράσουν τις διευθύνσεις μεταξύ της ακολουθίας bytes σε διάταξη «μηχανής» και της ακολουθίας bytes σε διάταξη «δικτύου», οτιδήποτε έχει σχέση με multi-byte data (int, float/double, data structures). Αυτό σημαίνει ότι little-endian μηχανές χρειάζεται να μετατρέψουν τις IP διευθύνσεις και αριθμούς θυρών σε big-endian μορφή (διάταξη «δικτύου»), έτσι ώστε να γίνει με επιτυχία η ζητούμενη επικοινωνία. Τέσσερις συναρτήσεις προσφέρονται για αυτή τη μετατροπή:

```
#include <arpa/inet.h>

uint32_t htonl(uint32_t hostint32);
uint16_t htons(uint16_t hostint16);
uint32_t ntohl(uint32_t netint32);
uint16_t ntohs(uint16_t netint16);
```

Οι δύο πρώτες επιστρέφουν 32-bit (long) αριθμό και 16-bit (short) αριθμό σε διάταξη «δικτύου», ενώ οι δύο τελευταίες επιστρέφουν 32-bit (long) αριθμό ή 16-bit (short) σε διάταξη «μηχανής» (Σχήμα 3.4).



Σχήμα 3.4: Συναρτήσεις για μετατροπή από διάταξη μηχανής σε διάταξη δικτύου και αντίστροφα

3.3.4 Δομές Διευθύνσεων

Μια δομή διεύθυνσης προσδιορίζει μια υποδοχή σε μια συγκεκριμένη οικογένεια πεδίων υποδοχών. Η TCP/IP σουίτα απαιτεί μια IP διεύθυνση και ένα αριθμό θύρας για κάθε διεύθυνση υποδοχής. Η μορφή της διεύθυνσης είναι συγκεκριμένη για κάθε πεδίο (π.χ. πεδίο Internet IPv4, πεδίο Internet IPv6, UNIX πεδίο). Για να μπορούν διαφορετικές μορφές διευθύνσεων να περνούν σε συναρτήσεις υποδοχών, οι μορφές

αυτές μορφοποιούνται (casting) σε μια γενική δομή διεύθυνσης υποδοχής, της βιβλιοθήκη <netinet/in.h>:

```
struct sockaddr
{
  sa_family_t  sa_family;     /* address family */
  char         sa_data[14];   /* variable-length address */
};
```

Η δομή Internet διεύθυνσης για IPv4 πεδίο υποδοχών είναι:

```
struct in_addr
{
  in_addr_t       s_addr;       /* IPv4 address */
};

struct sockaddr_in {
  sa_family_t    sin_family;   /* address family */
  in_port_t      sin_port;     /* port number */
  struct in_addr sin_addr;     /* IPv4 address */
  unsigned char   sin_zero[8]; /* filler */
};
```

3.3.5 Μετατροπή Δυαδικών Διευθύνσεων από Διάταξη Δικτύου σε Συμβολοσειρά και Αντίστροφα

Κάποιες φορές είναι απαραίτητη η εκτύπωση μιας διεύθυνσης που να είναι κατανοητή από τον άνθρωπο. Οι δύο πιο κάτω συναρτήσεις μετατρέπουν δυαδικές διευθύνσεις από διάταξη δικτύου σε συμβολοσειρά και αντίστροφα.

```
#include <arpa/inet.h>

const char *inet_ntop(int domain, const void *restrict addr, char
*restrict str, socklen_t size);

int inet_pton(int domain, const char *restrict str, void *restrict
addr);
```

η πρώτη επιστρέφει δείκτη σε συμβολοσειρά ή NULL σε περίπτωση λάθους. Η δεύτερη επιστρέφει -1 σε περίπτωση αποτυχίας αλλιώς 1.

3.3.6 Δέσμευση Θύρας για την Υποδοχή

Στην εφαρμογή του πελάτη, για τη συσχέτιση της υποδοχής με κάποια διεύθυνση (IP διεύθυνση και αριθμό θύρας) αφήνουμε συνήθως το σύστημα να διαλέξει μια διαθέσιμη διεύθυνση. Για τον εξυπηρετητή θα καθορίσουμε εμείς μια γνωστή διεύθυνση στην οποία θα καταφθάνουν οι αιτήσεις από τους πελάτες. Η συνάρτηση bind() χρησιμοποιείται για τη συσχέτιση μιας διεύθυνσης με την υποδοχή.

```
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr *addr, socklen_t len);
```

επιστρέφει -1 σε περίπτωση λάθους αλλιώς 0.

Το `sockfd` είναι ο περιγραφέας του αρχείου που επιστράφηκε από την `socket`(). Το `addr` είναι δείκτης σε δομή `struct sockadd` που περιέχει της πληροφορίες για τη διεύθυνση και το len είναι το μέγεθος του `addr` (`sizeof`(`struct sockaddr`)).

Μπορούμε να αυτοματοποιήσουμε τη διαδικασία να πάρουμε την IP διεύθυνση και αριθμό θύρας για να συσχετιστούν με συγκεκριμένη υποδοχή. Αν θέσουμε το `addr.sin_port = 0` τότε η συνάρτηση θα επιλέξει μια ελεύθερη θύρα και θα τη δεσμεύσει με την υποδοχή. Αν το `addr.sin_addr.s_addr` έχει την τιμή `INADDR_ANY` τότε αυτόματα βάζει την IP διεύθυνση της μηχανής. Τέλος να σημειώσουμε ότι σε κάποιες περιπτώσεις (ένωση με `connect`()) δεν είναι υποχρεωτική η κλήση της `bind()`.

3.3.7 Σύνδεση με Εξυπηρετητή

Σε μια συνδεσιστρεφές υπηρεσία (`SOCK_STREAM`), πριν αρχίσει η ανταλλαγή των δεδομένων, πρέπει να εγκαθιδρυθεί μια σύνδεση μεταξύ της υποδοχής ροής του πελάτη και του εξυπηρετητή. Χρησιμοποιείται η συνάρτηση `connect`() για τη δημιουργία της σύνδεσης:

```
#include <sys/socket.h>
  int connect(int sockfd, const struct sockaddr *addr, socklen_t len);
```

, όπου σε περίπτωση λάθους επιστρέφει -1, αλλιώς 0. Το `sockfd` είναι ο περιγραφέας στην υποδοχή του πελάτη. Αν ο περιγραφέας δεν είναι συνδεδεμένος με κάποια διεύθυνση η συνάρτηση αυτή θα δεσμεύσει μια προκαθορισμένη διεύθυνση με τον περιγραφέα αυτό. Το `addr` είναι δείκτης στην δομή `sockaddr` που περιγράφει την διεύθυνση υποδοχής του εξυπηρετητή και είναι μεγέθους `len`.

Να μην ξεχνούμε ότι για να πετύχει η σύνδεση ο εξυπηρετητής πρέπει να είναι σε λειτουργία και να δέχεται συνδέσεις στη διεύθυνση που προσπαθούμε να ενωθούμε.

3.3.8 «Άκουσμα» αιτήσεων σύνδεσης

Ο εξυπηρετητής ανακοινώνει ότι είναι πρόθυμος να δεχτεί αιτήσεις σύνδεσης, στον περιγραφέα υποδοχής sockfd, από πελάτες με καλώντας την πιο κάτω συνάρτηση:

```
#include <sys/socket.h>

int listen(int sockfd, int backlog);
```

, η οποία επιστρέφει -1 σε περίπτωση λάθους, αλλιώς 0.

Η παράμετρος backlog δηλώνει το μέγιστο αριθμό αιτήσεων που μπορούν να περιμένουν στην ουρά. Η μεγαλύτερη τιμή που μπορεί να πάρει είναι *SOMAXCONN* (<*sys/socket.h*>). Αν η ουρά γεμίσει, οι νέες αιτήσεις που θα καταφθάνουν θα απορρίπτονται μέχρις ότου να υπάρξει χώρος πάλι στην ουρά.

3.3.9 Αποδοχή Αίτησης Σύνδεσης

Αφού δηλώσει ο εξυπηρετητής ότι είναι πρόθυμος να δεχτεί αιτήσεις, το επόμενο βήμα είναι να αποδεχτεί μια εισερχόμενη αίτηση σύνδεσης που έχει υποβληθεί στην υποδοχή sockfd που «ακούει» με τη συνάρτηση:

```
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *restrict addr, socklen_t
*restrict len);
```

, όπου επιστρέφει περιγραφέα υποδοχής/αρχείου σε επιτυχία, αλλιώς -1.

Η δομή addr μετά το κάλεσμα της συνάρτησης έχει πληροφορίες για τη διεύθυνση του πελάτη που συνδέθηκε και το μέγεθος της επιστρέφεται στο `len`. Ο περιγραφέας αρχείου που επιστρέφεται είναι ο περιγραφέας υποδοχής που είναι ενωμένος με τον πελάτη που αίτησε τη σύνδεση με την συνάρτηση `connect`(). Έχει τον ίδιο τύπο υποδοχής και οικογένεια διεύθυνσης με τον περιγραφέα που «ακούει» ο εξυπηρετητής.

3.3.10 Ανταλλαγή Δεδομένων

Τώρα που περιγράψαμε τα βήματα σχετικά με το πως εγκαθιδρύεται η σύνδεση επικοινωνίας μεταξύ ενός πελάτη και ενός εξυπηρετητή, ήρθε η στιγμή να δούμε πως γίνεται η επικοινωνία μεταξύ δύο διεργασιών. Για αυτό το σκοπό μπορούν να χρησιμοποιηθούν οι συναρτήσεις `read()` και `write()`, εφόσον η υποδοχή αντιπροσωπεύεται από ένα περιγραφέα αρχείου. Επιπλέον όμως υπάρχουν και οι συναρτήσεις `send()` και `recv()`, οι οποίες προσφέρουν επιπλέον δυνατότητες για τον περιγραφέα υποδοχής, όπως για παράδειγμα την αποστολή επειγόντων δεδομένων

Για την αποστολή δεδομένων μέσω του υποδοχέα `sockfd` η συνάρτηση `send()` ορίζεται ως εξής:

```
#include <sys/socket.h>

  ssize_t send(int sockfd, const void *buf, size_t nbytes, int flags);
```

, όπου επιστρέφει τον αριθμό των bytes που έστειλε σε επιτυχία, αλλιώς -1.

Στο `buf` βρίσκονται τα δεδομένα που θα αποσταλούν. Συγκεκριμένα θα σταλούν τα `nbytes` δεδομένα από το `buf`. Ορίζονται τέσσερα βασικά flags που μπορεί α πάρει αυτή η παράμετρος (`MSG_DONTROUTE`, `MSG_DONTWAIT`, `MSG_EOR`, `MSG_OOB`), αλλά στην περίπτωση μας είναι πάντα 0.

Παρόμοια η συνάρτηση για παραλαβή των δεδομένων από την υποδοχή `sockfd` ορίζεται ως:

```
#include <sys/socket.h>
```

```
ssize_t recv(int sockfd, void *buf, size_t nbytes, int flags);
```

, όπου επιστρέφει τον αριθμό των bytes που διαβάστηκαν, 0 αν δεν υπάρχουν δεδομένα και η διεργασία στο άλλο άκρο της σύνδεσης έχει καλέσει τη συνάρτηση `shutdown()` ή -1 σε περίπτωση λάθους. Όπως και στη `send()` και στην `recv()` υπάρχουν κάποιες πιθανές τιμές για αυτήν την παράμετρο flags (`MSG_OOB`, `MSG_PEEK`, `MSG_TRUNC`, `MSG_WAITALL`).

3.3.11 Ταυτόχρονος Χειρισμός Υποδοχών

Η συνάρτηση select() προσφέρει το χειρισμό πολλών υποδοχών ταυτόχρονα. Βρίσκει ποιες υποδοχές είναι έτοιμες για διάβασμα δεδομένων, για γράψιμο δεδομένων και ποιες έχουν εγείρει εξαίρεση (exception). Ορίζεται ως:

```
#include <sys/select.h>

int select(int maxfdp1, fd_set *restrict readfds, fd_set *restrict
writefds, fd_set *restrict exceptfds, struct timeval *restrict tvptr);
```

, η οποία επιστρέφει είτε τον αριθμό των περιγραφέων που είναι έτοιμοι είτε 0 σε περίπτωση λήξης του χρονικού ορίου (timeout) είτε -1 σε περίπτωση λάθους. Η συνάρτηση χειρίζεται σύνολα περιγραφέων. Στο `readfds`, βάζουμε τους περιγραφείς που θέλουμε να ελέγξουμε αν είναι έτοιμοι για διάβασμα. Παρόμοια και στα `writefds`, `exceptfds` βάζουμε τους περιγραφείς που θέλουμε να ελέγξουμε για γράψιμο ή εξαιρέσεις αντίστοιχα. Με το που θα επιστρέψει η συνάρτηση θα έχει στα σύνολα αυτά μόνο τους περιγραφείς που είναι έτοιμοι. Οποιοδήποτε σύνολο δε μας ενδιαφέρει το θέτουμε με `NULL`. Η πρώτη παράμετρος έχει πάντοτε το μεγαλύτερο από τους περιγραφείς συν ένα. Η τελευταία παράμετρος ορίζει το χρονικό όριο λήξης. Αν η χρονική περίοδος λήξης έχει φτάσει και η συνάρτηση δεν έχει βρει κάποιους «έτοιμους» περιγραφείς θα επιστρέψει για να συνεχίσει η εκτέλεση του προγράμματος. Το `srtuct timeval` έχει τα ακόλουθα πεδία:

```
struct timeval

{

      int tv_sec; // seconds
```

```
    int tv_usec; // microseconds
};
```

Αν ορίσουμε 0 στις τιμές και των δευτερολέπτων και των μικροδευτερολέπτων, δηλαδή να θέσουμε με 0 το χρονικό όριο λήξης, τότε η συνάρτηση θα τερματίζει αμέσως με τους έτοιμους περιγραφείς. Αντίθετα, αν θέσουμε αυτό το πεδίο με `NULL` τότε η συνάρτηση θα επιστρέψει μόνο όταν κάποιος περιγραφέας είναι έτοιμος, αλλιώς θα περιμένει για πάντα.

Τελειώνοντας να αναφέρουμε κάποιες μακροεντολές (macros) που εξυπηρετούν στο χειρισμό των συνόλων `fd_set`:

- `FD_ZERO(fd_set *set)`
  Σβήνει όλους τους περιγραφείς από το σύνολο `set`.

- `FD_SET(int fd, fd_set *set)`
  Προσθέτει τον περιγραφέα fd στο σύνολο `set`.

- `FD_CLR(int fd, fd_set *set)`
  Αφαιρεί τον περιγραφέα `fd` από το σύνολο `set`.

- `FD_ISSET(int fd, fd_set *set)`
  Ελέγχει εάν ο `fd` βρίσκεται στο σύνολο `set`.

3.3.12 Άλλες συναρτήσεις για Χειρισμό Υποδοχών

Κάποιες επιπλέον χρήσιμες συναρτήσεις, που γίνετε χρήση τους στην υλοποίηση του αλγορίθμου που περιγράφεται στη συνέχεια του κεφαλαίου είναι οι παρακάτω:

```
#include <netdb.h>
struct hostent *gethostbyname(char *name);
```

, η οποία επιστρέφει ένα δείκτη σε δομή `struct hostent` σε περίπτωση επιτυχίας ή ένα `NULL` δείκτη σε περίπτωση λάθους. Παίρνει ως παράμετρο το όνομα της μηχανής και επιστρέφει ένα δείκτη σε δομή `struct hostent`, όπου στο πεδίο `h_addr` της δομής φυλάγεται η δυαδική 32-bit διεύθυνση και στο πεδίο `h_length` το μέγεθός της.

Η παρακάτω συνάρτηση που είναι παρόμοια με την προηγούμενη ορίζεται ως:

```
#include <netdb.h>

struct hostent *gethostbyaddr(char *addr, int len, int type);
```

, η οποία επιστρέφει ένα δείκτη σε δομή `struct hostent` σε περίπτωση επιτυχίας ή ένα `NULL` δείκτη σε περίπτωση λάθους. Επιστρέφει ένα δείκτη σε δομή `struct hostent` για έναν υπολογιστή, που δεδομένης της διεύθυνσης του `addr`, του μεγέθους της `len` και του είδους της `type` (`AF_INET`), στο πεδίο `h_name` της επιστρεφόμενης δομής τοποθετείται το όνομα του υπολογιστή.

Η δομή `hostent (<netdb.h>)` έχει τα παρακάτω πεδία:

```
struct hostent {
     char   *h_name;       /* name of host */
     char  **h_aliases;    /* pointer to alternate host name array */
     int     h_addrtype;   /* address type */
     int     h_length;     /* length in bytes of address */
     char  **h_addr_list;  /* pointer to array of network addresses */
     .
     .
     .
 };
```

Τέλος όπως όλοι οι περιγραφείς έτσι και οι περιγραφείς υποδοχών μπορούν να κλείσουν με την κλήση της συνάρτησης `close()`. Επιπλέον υπάρχει η συνάρτηση `shutdown()` που επιτρέπει να απενεργοποιήσουμε την υποδοχή για συγκεκριμένες περιπτώσεις, όπως για γράψιμο ή για διάβασμα ή και για τα δυο. Ορίζεται ως:

```
#include <sys/socket.h>

int shutdown (int sockfd, int how);
```

και επιστρέφει -1 σε περίπτωση λάθους, αλλιώς 0. η παράμετρος `how` μπορεί να είναι:

- `SHUT_RD`: απενεργοποιείται το διάβασμα από την υποδοχή.
- `SHUT_WR`: δεν είναι εφικτή η αποστολή δεδομένων από τον περιγραφέα.
- `SHUT_RDWR`: δεν είναι δυνατή ούτε η αποστολή ούτε η λήψη δεδομένων από την υποδοχή.

3.4 Αρχεία Εντολών

Κατόπιν της ολοκλήρωσης της υλοποίησης του αλγορίθμου έγινε η αξιολόγηση του με τη διεξαγωγή πειραμάτων στο κατανεμημένο δίκτυο Planet Lab. Για την επιτυχή εξέταση της απόδοσης του αλγορίθμου κάθε πείραμα εξετάζει τον αλγόριθμο κάτω από διαφορετικά σενάρια μεταβάλλοντας κάποιες παραμέτρους. Κάποιες από αυτές έχουν σχέση με το χρόνο και είναι επιθυμητό όλες οι διεργασίες να ξεκινούν ταυτόχρονα. Για να πετύχω όσο δυνατό πιο ταυτόχρονη έναρξη των διεργασιών έγραψα κάποια αρχεία εντολών(scripts). Εκτός αυτού, λόγω του μεγάλου αριθμού διεργασιών σε κάποια πειράματα (μέχρι 80 αναγνώστες, 49 εξυπηρετητές και ένας εγγραφέας) είναι χρονοβόρο να ξεκινήσει η λειτουργία όλων αυτών των διεργασιών χειροκίνητα.

Συνολικά έγραψα δύο αρχεία εντολών που εξυπηρετούν την πιο πάνω ανάγκη. Το πρώτο περιέχει εντολές των βιβλιοθηκών του κελύφους bash (Bourn Again Shell) και το δεύτερο χρησιμοποιεί το εργαλείο expect [10], μια επέκταση της γλώσσας δέσμης ενεργειών (scripting language) TCL.

Το εργαλείο expect χρησιμοποιείται κυρίως για την αυτοματοποίηση εφαρμογών όπως telnet, ftp, passwd, fsck, rlogin, tip, ssh και πολλές άλλες. Στην περίπτωση μας χρησιμοποιείται για την αυτοματοποίηση της εφαρμογής ssh. Χρησιμοποιεί Unix ψευδοκελύφη για την αυτοματοποίηση αυθαίρετων εφαρμογών που έχουν πρόσβαση μέσω κελύφους.

Η expect χρησιμοποιεί κυρίως τέσσερις εντολές:

- `spawn`: χρησιμοποιείται για να αρχίσει κάποιο πρόγραμμα.
- `expect`: χρησιμοποιείται για να περιμένει κάποια συγκεκριμένη συμβολοσειρά. Εξ' ορισμού περιμένει μόνο 10 δευτερόλεπτα και μετά εκτελεί την ενέργεια. Αυτό μπορεί να οδηγήσει σε εσφαλμένη συμπεριφορά ειδικά σε χαμηλής ταχύτητας δίκτυα ή δίκτυα με συμφόρηση που και στις δύο περιπτώσεις κάποιο

πακέτο μπορεί να αργήσει να φτάσει στον προορισμό του. Έτσι μπορούμε να ορίσουμε στη μεταβλητή timeout για πόσο χρονικό διάστημα να περιμένει τη συμβολοσειρά. Αν η τιμή που θα θέσουμε είναι -1 τότε θα περιμένει για πάντα. Εδώ να προσθέσουμε ότι η expect υποστηρίζει κανονικές εκφράσεις.

- `send`: για να στείλει κάποια εντολή
- `interact`: για να δώσει τον έλεγχο στο χρήστη.

Όπως είπαμε το εργαλείο αυτό είναι προέκταση της TCL, γι αυτό οι πιο πάνω εντολές μπορούν να χρησιμοποιηθούν σε συνδυασμό με εντολές TCL.

3.5 Περιγραφή Κύριων Συστατικών του Προγράμματος

Ήρθε η στιγμή να μελετήσουμε εξονυχιστικά την υλοποίηση του αλγορίθμου για τον οποίο έχει ήδη γίνει λόγος. Ο κώδικας της υλοποίησης του αλγορίθμου βρίσκεται στο Παράρτημα Α. Η υλοποίηση έχει γίνει σε γλώσσα προγραμματισμού C στο μοντέλο πελάτη-εξυπηρετητή (πιο συγκεκριμένα του ταυτόχρονου) και με χρήση προγραμματισμού υποδοχών. Για να γίνουμε πιο επακριβής, δύο διαφορετικές εφαρμογές, ο αναγνώστης και ο εγγραφέας, παίζουν το ρόλο του πελάτη. Ο εξυπηρετητής είναι αυτός που κρατά την τιμή του ατομικού αντικειμένου και έχει επικοινωνία με τις δύο αυτές εφαρμογές. Να θυμίσουμε ότι ο εγγραφέας και το σύνολο των αναγνωστών επικοινωνούν μόνο με τους εξυπηρετητές και δεν ανταλλάζουν μηνύματα μεταξύ τους. Παρόμοια, το σύνολο των εξυπηρετητών επικοινωνούν μόνο με τον εγγραφέα και το σύνολο των αναγνωστών, χωρίς την μεταξύ τους ανταλλαγή μηνυμάτων. Οι εξυπηρετητές ανήκουν σε ένα σύστημα απαρτίας του οποίου δεν έχουν γνώση οι ίδιοι. Οι εφαρμογές εγγραφέα και αναγνώστη ενημερώνονται για το σύστημα απαρτίας μόνο με την εκκίνηση τους, χωρίς αυτό να προκαλεί κανένα πρόβλημα στη συνέχεια διότι το σύστημα απαρτίας παραμένει σταθερό και δεν αλλάζει δυναμικά. Τέλος τα είδη των μηνυμάτων που ανταλλάζονται στο σύστημα είναι της μορφής:

```
<Τύπος μηνύματος, Χρονοσφραγίδα, Τιμή Αντικειμένου, Προηγούμενη Τιμή Αντικειμένου, Ταυτότητα Διεργασίας, Μετρητής Αίτησης Διεργασίας#>
```

Ο τύπος του μηνύματος μπορεί να είναι ένα από τα έξι: `WRITE`, `READ`, `INFORM`, `WRITEACK`, `READACK`, `INFORACK`. Τα τρία τελευταία συναντώνται μόνο σε μηνύματα σταλμένα από τους εξυπηρετητές ως απαντήσεις των αιτήσεων των πελατών που περιλαμβάνουν ένα από τα τρία πρώτα αντίστοιχα. Ο εγγραφέας κάνει αιτήσεις `WRITE`, για να ενημερώσει την τιμή των αντιγράφων του αντικειμένου στους εξυπηρετητές. Μια εφαρμογή αναγνώστη κάνει μία από της δύο αιτήσεις `READ` και `INFORM` . Την πρώτη για να μάθει την τιμή του αντικειμένου και τη δεύτερη στην περίπτωση που κρίνει ότι χρειάζεται δεύτερο γύρο επικοινωνίας-βοηθά στην πιο γρήγορη μετάδοση της πιο πρόσφατης τιμής του αντικειμένου στο σύστημα. Κάθε μήνυμα τελειώνει με τον ειδικό χαρακτήρα `#` για να υποδηλώσει το τέλος του.

Οι εφαρμογές του εγγραφέα και του αναγνώστη λαμβάνουν σαν ορίσματα από τη γραμμή εντολών δύο αρχεία. Το ένα περιγράφει το σύστημα απαρτίας και το άλλο έχει πληροφορίες για τους εξυπηρετητές στο σύστημα. Πριν δούμε τη μορφή των δύο αρχείων να διευκρινίσουμε ότι δε γίνεται κάποιος έλεγχος ορθότητας των αρχείων και θεωρείται ότι δίνονται ορθά από το χρήστη.

Το αρχείο με τις πληροφορίες των εξυπηρετών περιλαμβάνει:

1) Στην πρώτη γραμμή είναι ο αριθμός των εξυπηρετητών στο σύστημα.
2) Ακολούθως σε κάθε γραμμή τη διεύθυνση (όνομα) του κάθε εξυπηρετητή και τη θύρα στην οποία αποδέχεται νέες συνδέσεις.

Το αρχείο με τις πληροφορίες του συστήματος απαρτίας αναπαριστάται ως παρακάτω. Για κάθε quorum έχει το χαρακτήρα Q και δίπλα τον αριθμό του quorum. Μετά ανοίγει παρένθεση και χωρισμένες με κόμμα είναι οι ταυτότητες των εξυπηρετητών του quorum και στο τέλος κλείνει η παρένθεση. Κατά αυτόν τον τρόπο κάτω από το quorum είναι και οι τομές του με τα υπόλοιπα quorums. Αλλά αντί το χαρακτήρα Q γράφει Int (intersection) και δίπλα ακολουθεί η ταυτότητα του quorum με τον οποίο τέμνεται. Οι εξυπηρετητές μέσα στην παρένθεση είναι οι εξυπηρετητές που ανήκουν στην τομής μεταξύ των δύο quorums.

Όπως βλέπουμε οι εφαρμογές πελάτη, αναγνώστη και εγγραφέα επεξεργάζονται τα ίδια αρχεία. Εκτός αυτού έχουν και άλλα κοινά στην λειτουργία τους και γι' αυτό και χρησιμοποιούν κάποιες κοινές συναρτήσεις. Μια από αυτές είναι αυτή που στέλνει μια αίτηση σε όλους τους εξυπηρετητές. Μια άλλη είναι αυτή που δέχεται απαντήσεις από τους εξυπηρετητές μέχρι να πάρει απαντήσεις από ένα ολόκληρο quorum. Για να το πετύχει αυτό χρησιμοποιεί έναν πίνακα ίσο με τον αριθμό των εξυπηρετητών, όπως το έχει διαβάσει από το αρχείο. Αρχικά αρχικοποιεί όλο τον πίνακα με τη σταθερά `NOT_ACK`. Όταν πάρει μια απάντηση από τον εξυπηρετητή με ταυτότητα id βάζει στη θέση id-1 την τιμή της σταθεράς `ACK`. Σε κάθε κύκλο ελέγχει αν έχουν συμπληρωθεί οι απαντήσεις από κάποιο quorum. Αυτό γίνεται διασχίζοντας τη δομή με το σύστημα quorum. Για κάθε quorum ελέγχει ένα τους εξυπηρετητές, μέσω του πίνακα που αναφέραμε, αν έχουν στείλει απάντηση. Αν βρει κάποιο εξυπηρετητή, στο quorum που ελέγχει, που δεν έχει στείλει απάντηση (δεν έχει την τιμή `ACK` στον αντίστοιχο πίνακα) τότε προχωρεί στην εξέταση του επόμενου quorum. Αν διασχίσει όλους τους εξυπηρετητές σε ένα quorum τότε σημαίνει ότι απάντησαν όλοι οι εξυπηρετητές αυτού του quorum, η αίτηση ολοκληρώθηκε και δεν δέχεται άλλες απαντήσεις.

3.5.1 Εφαρμογή Εξυπηρετητή (Server)

Ο ρόλος του εξυπηρετητή στο σύστημα είναι να κρατά ένα αντίγραφο του αντικειμένου και να απαντά στις αιτήσεις των πελατών. Σε κάθε απάντηση, κάποιας αίτησης, που στέλνει περιλαμβάνει την τιμή του αντικειμένου. Κάθε τιμή του αντικειμένου συνοδεύεται από μια μοναδική χρονοσφραγίδα, η οποία ενημερώνεται και αυτή με κάθε αλλαγή τιμής του αντικειμένου.

Ας δούμε τώρα πιο λεπτομερώς την υλοποίηση του εξυπηρετητή. Κατά την εκκίνηση της εφαρμογής, δέχεται σαν ορίσματα γραμμής εντολών την ταυτότητα του και τον αριθμό θύρας (port number) στον οποίο θα «ακούει» για νέες αιτήσεις. Καταλήγοντας μια εφαρμογή εξυπηρετητή ξεκινά τη λειτουργία του γράφοντας στη γραμμή εντολής:

```
> server [server id] [port number]
```

όπου `server` είναι το όνομα του εκτελέσιμου αρχείου.

Στην τοπική του μνήμη κρατά την τιμή του αντικειμένου (αρχικά -1), τη χρονοσφραγίδα (αρχικά 0) και την προηγούμενη τιμή του αντικειμένου. Να σημειώσουμε ότι ενώ η χρονοσφραγίδα *timestamp* αντιστοιχεί στην τιμή του αντικειμένου *value* , η χρονοσφραγίδα της προηγούμενης τιμής δεν είναι απαραίτητα timestamp-1. Αυτό λόγω της φύσης του δικτύου (ασυγχρονία, σφάλμα, καθυστερήσεις) ενδεχομένως να μην ενημερωθεί άμεσα για μια εγγραφή και να ενημερωθεί πρώτα με μια μεταγενέστερη εγγραφή. Επιπλέον κρατά ένα πίνακα *counter* με *MAX_READERS+1* θέσεις, ο οποίος φυλάει για κάθε διεργασία με ταυτότητα *id* στην *counter[id]* τον αριθμό αίτησης της διεργασίας όπως στάλθηκε στην τελευταία αίτηση από τη διεργασία *id*. Επειδή το δίκτυο είναι ασύγχρονο κάποια μηνύματα μπορεί να φτάσουν με λανθασμένη σειρά και με τη χρήση του πίνακα counter απορρίπτει τα πιο παλιά μηνύματα και επεξεργάζεται μόνο πιο πρόσφατα. Να ανακαλέσουμε ότι ο μετρητής των αιτήσεων αυξάνεται από τη διεργασία κατά ένα πριν στείλει την αίτηση. Έτσι το μόνο που αρκεί να κάνει ο εξυπηρετητής είναι να ελέγξει αν ο αριθμός αίτησης στο νέο μήνυμα που παραλήφθηκε είναι πιο μεγάλος από το *counter[id*]. Αν ναι τότε είναι πρόσφατη αίτηση και επεξεργάζεται αλλιώς απορρίπτεται.

Ειπώθηκε ότι ένας ταυτόχρονος εξυπηρετητής, όπως το δικό μας, αναθέτει το χειρισμό μιας νέας αίτησης σε ένα δεύτερο εξυπηρετητή. Στην περίπτωση μας ο δεύτερος εξυπηρετητής είναι μια νέα διεργασία παιδί του πρώτου που δημιουργείται μέσω της συνάρτησης *fork()* της C. Αυτή η συνάρτηση δημιουργεί μια νέα διεργασία στο σύστημα η οποία είναι παιδί της διεργασίας που κάλεσε τη συνάρτηση. Η διεργασία-παιδί κληρονομά από τον πατέρα όλες της μεταβλητές, αλλά μετά την κλήση της συνάρτησης συνεχίζει η καθεμιά ξεχωριστά και αυτόνομα χωρίς να μεταβάλλει η μια τις μεταβλητές της άλλης. Εξαιτίας αυτού χρειάζεται κάποιος τρόπος επικοινωνίας μεταξύ όλων των διεργασιών για να ξέρουν ανά πάσα στιγμή την πιο πρόσφατη τιμή του αντικειμένου που στάλθηκε στον εξυπηρετητή (όχι μια διεργασία μόνο). Επίσης υπάρχει η ίδια ανάγκη και για τις υπόλοιπες μεταβλητές που αναφέρθηκαν πιο πάνω. Η ενδοεπικοινωνία μεταξύ των διεργασιών του εξυπηρετητή επιτυγχάνεται με κοινά

αρχεία που είναι προσβάσιμα από όλες τις διεργασίες. Όταν μια από τις διεργασίες λάβει ένα νέο μήνυμα ενημερώνει τις τοπικές της μεταβλητές από τα κοινά αρχεία. Αν τα αρχεία δεν υπάρχουν τα δημιουργεί αποθηκεύοντας τις αρχικές τιμές των μεταβλητών. Με τον ίδιο τρόπο αποθηκεύει τις μεταβλητές αυτές, στα κοινά αρχεία, μετά την επεξεργασία ενός μηνύματος. Όταν μια διεργασία (αυτή που επικοινωνεί με τον εγγραφέα) πάρει το μήνυμα «FIN» τότε σβήνει όλα τα τοπικά αρχεία και τερματίζει την εκτέλεση της και την εκτέλεση των υπόλοιπων διεργασιών.

3.5.2 Εφαρμογή Εγγραφέα (Writer)

Στο σύστημα υπάρχει ένας μόνο εγγραφέας ο οποίος γράφει στα αντίγραφα του αντικειμένου. Για να εκτελέσει μια εγγραφή στέλνει μήνυμα `WRITE` σε όλους τους εξυπηρετητές. Μόλις πάρει απάντηση από ένα quorum τότε ολοκληρώνεται η εγγραφή. Η ιδιότητα τομής των quorum συστημάτων εγγυάται τη γνώση της νέας τιμής σε όλο το σύστημα. Μαζί με την τιμή του αντικειμένου υπάρχει και η αντίστοιχη χρονοσφραγίδα η οποία καθορίζει τη φυσική σειρά εγγραφής στο σύστημα, εφόσον υπάρχει μόνο ένας εγγραφέας.

Στη τοπική του μνήμη ο εγγραφέας, εκτός από την τιμή value και την αντίστοιχη χρονοσφραγίδα `timestamp`, κρατά και την προηγούμενη τιμή του αντικειμένου pvalue και τον μετρητή των αιτήσεων εγγραφής που έκανε `counter`. Οι πιο πάνω μεταβλητές είναι απαραίτητες για την εφαρμογή του αλγορίθμου SLIQ. Υπάρχουν και άλλες μεταβλητές που είναι απαραίτητες για την υλοποίηση του αλγορίθμου, όπως τη δομή που φυλάει το quorum σύστημα των εξυπηρετητών. Είναι αναγκαίο για τον εγγραφέα να γνωρίζει το quorum σύστημα εφόσον για κάθε αίτηση εγγραφής, η εγγραφή ολοκληρώνεται όταν πάρει απάντηση από ένα πλήρες quorum.

Μια σύνδεση TCP είπαμε ότι είναι μια σύνδεση αξιόπιστης αμφίδρομης επικοινωνίας μόνο μεταξύ ενός πελάτη και ενός εξυπηρετητή. Για αυτό το λόγο ο εγγραφέας εγκαθιδρύει διαφορετική υποδοχή ροής για επικοινωνία με τον κάθε εξυπηρετητή. Όλοι οι περιγραφείς των υποδοχών ροής (socket file descriptors) φυλάγονται σε ένα πίνακα

για άμεση πρόσβαση κατά την αποστολή μιας αίτησης και τη λήψη της αντίστοιχης απάντησης.

Το σύστημα απαρτίας και οι εξυπηρετητές του συστήματος περιγράφονται σε αρχεία τα οποία δέχεται ο εγγραφέας σαν ορίσματα στην γραμμή εντολών. Για να είμαστε πιο ακριβείς η εφαρμογή του εγγραφέα ξεκινά τη λειτουργία του γράφοντας στη γραμμή εντολής:

```
>writer [αρχείο με τους εξυπηρετητές] [αρχείο με το σύστημα απαρτίας]
```

όπου `writer` είναι το εκτελέσιμο αρχείο για τον εγγραφέα .

3.5.3 Εφαρμογή Αναγνώστη (Reader)

Στο σύστημα υπάρχουν πολλαπλοί αναγνώστες. Η υλοποίηση τους είναι παρόμοια με αυτή του εγγραφέα, με τη διαφορά ότι ο αναγνώστης μπορεί να εκτελέσουν δύο γύρους επικοινωνίας. Όταν σταλεί μια αίτηση ανάγνωσης `READ` σε όλους τους εξυπηρετητές, ο αναγνώστης περιμένει να πάρει απαντήσεις `READACK` από ένα quorum. Μετέπειτα εξετάζει την κατανομή που έχει η μέγιστη χρονοσφραγίδα μέσα στο σύστημα απαρτίας για να καθορίσει το quorum view. Τότε θα κρίνει αν χρειάζεται να εκτελέσει δεύτερο γύρο επικοινωνίας.

Στην τοπική του μνήμη όπως και οι εξυπηρετητές και εγγραφέας κρατά την τιμή του αντικειμένου, την πιο πρόσφατη χρονοσφραγίδα, την προηγούμενη τιμή του αντικειμένου και το μετρητή των αιτήσεων ανάγνωσης που έχει κάνει. Επιπλέον αποθηκεύει και τη μέγιστη χρονοσφραγίδα για τον προσδιορισμό του quorum view.

Το ίδιο με τον εγγραφέα, ο αναγνώστης πρέπει και αυτός να έχει γνώση του συστήματος απαρτίας και των εξυπηρετητών του συστήματος. Η γνώση αυτή λαμβάνεται με τον ίδιο τρόπο, μέσω αρχείων που παίρνονται σαν ορίσματα στη γραμμή

εντολών. Η εφαρμογή του αναγνώστη ξεκινά τη λειτουργία του γράφοντας στη γραμμή εντολής:

```
>reader [reader id] [αρχείο με τους εξυπηρετητές] [αρχείο με το σύστημα απαρτίας]
```

όπου `reader` το όνομα του εκτελέσιμου αρχείου του αναγνώστη.

Ο ψευδοκώδικας στο Σχήμα 3.5 περιγράφει τη συνάρτηση η οποία εντοπίζει ποιο quorum view αντιμετωπίζει ο αναγνώστης μετά την απάντηση ενός πλήρες quorum. Καταρχάς ελέγχει αν όλοι οι εξυπηρετητές αυτού του quorum έστειλαν τη μέγιστη χρονοσφραγίδα. Στην υλοποίηση αυτό γίνεται με τη βοήθεια της τομής για το σύστημα απαρτίας και του πίνακα που κρατά τις απαντήσεις των εξυπηρετητών. Αν βρει κάποιο εξυπηρετητή που δεν έστειλε τη μέγιστη χρονοσφραγίδα, τότε σημαίνει δεν είναι Quorum View 1 και σταματά τη διάσχιση του quorum. Μόλις τελειώσει η διάσχιση ελέγχεται αν έχει προσπελαθεί όλη η δομή (το i δείχνει στο `NULL`). Δεν έχει διασχίσει όλο το quorum στην περίπτωση που σταματά η διάσχιση, που βρίσκει μικρότερη χρονοσφραγίδα. Αλλιώς είναι Quorum View 1. με όμοιο τρόπο διασχίζει όλες τις τομές για να ελέγξει αν σε κάποια όλοι οι εξυπηρετητές έχουν τη μέγιστη σφραγίδα. Αν ναι τότε είναι Quorum View 3, αλλιώς είναι Quorum View 2.

```
1  Για κάθε στοιχείο i του quorum
2  Begin
3      if(serverAck[i.id-1].value != maxTS)
4          break
5  End
6
7  //Αν έχει διαπεράσει όλη τη λίστα, άρα όλοι οι εξυπηρετητές
8  //στο quorum έχουν το maxTs
9  if (i == NULL)
10  return Quorum View 1
11
12 Για κάθε τομή t με κάποιο quorum
13 Begin
14     Για κάθε στοιχείο i της τομής
15     Begin
16         if(serverAck[i.id-1].value != maxTs)
17             break
18     End
19     //Αν έχει διαπεράσει όλη τη λίστα, άρα όλοι οι εξυπηρετητές
20     //στην τομή έχουν το maxTs
21     if (i == NULL)
22         return Quorum View 3
23 End
24
25 //Αν δεν είναι Quorum View 1 ή 2 τότε είναι Quorum View 3
26   return Quorum View 2
```

Σχήμα 3.5: Ψευδοκώδικας για την εύρεση του Quorum View

3.6 Δυσκολίες και Παραδοχές

Αρχικά ο αλγόριθμος υλοποιήθηκε και ελέγχθηκε για την ορθότητα του στο τοπικό δίκτυο του Πανεπιστημίου Κύπρου. Αργότερα ελέγχθηκε αν η συμπεριφορά του αλγορίθμου ήταν ορθή και στο Planet Lab. Έτσι αποφεύχθηκε η αχρείαστη αποστολή μηνυμάτων, συνεπώς λιγότερος όγκος δεδομένων στο δίκτυο του Planet Lab, που πιθανώς να καθυστερούσε την παράδοση πακέτων άλλων εφαρμογών. Κατά τον τρόπο αυτό εξοικονομήθηκε και σημαντικός χρόνος από τη συνεχή ένωση με σταθμούς στο Planet Lab και αποστολή αρχείων προς τους σταθμούς αυτούς.

Στο τοπικό δίκτυο του Πανεπιστημίου Κύπρου όμως δεν ήταν δυνατός ο έλεγχος του μοντέλου πελάτη-εξυπηρετητή και γενικότερα του αλγόριθμου σε διαφορετικές μηχανές παρά μόνο στην ίδια μηχανή. Για λόγους ασφαλείας οι μηχανές του πανεπιστημίου δεν δέχονται αιτήσεις για επικοινωνία στις υποδοχές τους από άλλους υπολογιστές. Εφόσον ο αλγόριθμος έτρεχε τοπικά στην ίδια μηχανή δεν υπήρχαν σε τόσο βαθμό οι συνθήκες του ασύγχρονου δικτύου στο οποίο θα δοκιμαζόταν αργότερα ο αλγόριθμος. Κατά συνέπεια δεν υπήρχαν τόσο ρεαλιστικές καθυστερήσεις στην παράδοση των μηνυμάτων με αποτέλεσμα κάποια σφάλματα στον αλγόριθμο να μη φανερωθούν.

Κεφάλαιο 4

Το Κατανεμημένο Δίκτυο Planet Lab

Το Planet Lab [15] είναι ένα παγκόσμιο ερευνητικό δίκτυο που υποστηρίζει την ανάπτυξη νέων υπηρεσιών δικτύων. Από την αρχή του 2003, περισσότεροι από 1.000 ερευνητές στα κορυφαία ακαδημαϊκά όργανα και τα βιομηχανικά ερευνητικά εργαστήρια έχουν χρησιμοποιήσει το Planet Lab για ανάπτυξη νέων τεχνολογιών για κατανεμημένη αποθήκευση, χαρτογράφηση δικτύων, peer-to-peer συστήματα, κατανεμημένους πίνακες κατακερματισμού και την επεξεργασία ερωτήσεων (queries). Το Πανεπιστήμιο Κύπρου συμμετέχει στο Planet Lab με δύο σταθμούς.

4.1 Κίνητρα

Τα τελευταία χρόνια παρατηρήθηκε μια ραγδαία εξέλιξη και χρήση του Διαδικτύου. Οι χρήστες του σε όλον τον κόσμο ξεπερνούν το ένα δισεκατομμύριο, και αυξάνεται όλο και πιο πολύ κάθε χρόνο. Το Διαδίκτυο αναπτύχθηκε ως ένα υπερεπίπεδο πάνω από το τηλεφωνικό δίκτυο. Μέχρι τότε το υπάρχον τηλεφωνικό δίκτυο και τα άλλα δίκτυα δεν ήταν εύκαμπτα και δύσκολο να αλλάξουν, αλλά υπήρχε ανάγκη να χτιστεί ένα εύρωστο και εύκαμπτο δίκτυο για ανταλλαγή δεδομένων. Παρά ταύτα μια ολοκαίνουργια υποδομή δικτύου επικοινωνίας ήταν ανέφικτη, πρακτικά και οικονομικά. Για να γεφυρωθούν τα δύο δίκτυα, εισάχθηκε η έννοια του υπερεπίπεδου, το χτίσιμο ενός νέου επιπέδου λειτουργικότητας πάνω από το τηλεφωνικό δίκτυο. Το νέο αυτό επίπεδο θα υπερίσχυε της υποδομής του τηλεφωνικού δικτύου. Η αρχιτεκτονική αυτή του

Διαδικτύου έλκυσε την ερευνητική κοινότητα, που ανέπτυξε πολύ γρήγορα καινοτομίες υπηρεσιών παγκοσμίας κλίμακας. Εντούτοις, στην εύκαμπτη αρχιτεκτονική του Διαδικτύου είναι πολύ εύκολη η διάδοση ιών και worms, η πρόσβαση σε προσωπικά δεδομένα και πολλά άλλα θέματα ασφαλείας που απασχολούν τους ερευνητές μέχρι σήμερα. Ακόμη αντιμετωπίζονται και άλλα πολλά προβλήματα. Ένα από αυτά είναι η κατάρρευση εξυπηρετητών από την ταυτόχρονη πρόσβαση μεγάλου αριθμού χρηστών, μεγαλύτερου του ορίου συνδέσεων που μπορεί ο εξυπηρετητής να χειριστεί. Να προσθέσουμε ότι σε συνδυασμό με τους περιορισμούς της αρχιτεκτονικής του, η εμπορική επιτυχία του Διαδικτύου το κάνει πολύ δύσκολο να αλλάξει. Υπάρχει η ανησυχία ότι η εισαγωγή νέων πρωτοκόλλων στο Διαδίκτυο θα επιφέρει δυσλειτουργίες και μείωση της απόδοσης του δικτύου. Συνεπώς η λύση είναι να δημιουργηθούν νέα πρωτόκολλα ως υπερεπίπεδα του παρών Διαδικτύου, για να έχουμε νέες τεχνολογίες και περισσότερη ασφάλεια. Αυτό ήταν ένα από τα κίνητρα για τη δημιουργία του κατανεμημένου δικτύου Planet Lab.

Επιπροσθέτως, υπήρχε ανάγκη από την ερευνητική κοινότητα να πειραματιστεί με παγκοσμίας κλίμακας υπηρεσίες, παρά μόνο με προσομοιώσεις και εργαστηριακές εξομοιώσεις. Το Planet Lab ήρθε για να ευκολύνει τα πράγματα και να προσφέρει αυτή τη δυνατότητα.

4.2 Η εξέλιξη

Το Μάρτιο του 2002 οι Larry Peterson (Princeton) και David Culler (UC Berkeley and Intel Research) διοργάνωσαν μια σύσκεψη από ερευνητές που ενδιαφέρονταν σε υπηρεσίες δικτύου παγκόσμιας κλίμακας και πρότειναν το Planet Lab. Οικοδεσπότης της σύσκεψης ήταν οι Intel Reasearch και Berkeley που προσέγγισαν 30 ερευνητές από τους MIT, Washington, Rice, Berkeley, Princeton, Columbia, Duke, CMU και Utah. Η David Tennenhouse (Intel Research) συμφώνησε να δώσει το κεφάλαιο του project για τους πρώτους 100 σταθμούς. Μέχρι τον Οκτώβριο του 2002 ολοκληρώθηκε η αρχική ανάπτυξη 100 σταθμών σε 42 sites και η έκδοση 1.0 του λογισμικού του Planet Lab. Τότε δημοσιεύτηκε και το πρώτο άρθρο ([14]) που περίγραφε το Planet Lab και τους

στόχους του. Το όραμα όπως περιγράφεται μέσα από το άρθρο είναι να φτάσουν τη συμμετοχή 1000 sites. Αργότερα, τον Ιούνιο του 2003, ανακοινώθηκε δημόσια από τους Intel, Princeton, UC Berkeley και το Πανεπιστήμιο της Washington οι προθέσεις τους για συμμετοχή ακαδημαϊκών και βιομηχανικών ιδρυμάτων έτσι ώστε να εξαπλωθεί η ανάπτυξη του Planet Lab. Οι προθέσεις αυτές πραγματοποιήθηκαν το Ιανουάριο του 2004 όπου και η λειτουργία του Planet Lab ανατέθηκε από την Intel στο Princeton. Σήμερα το Planet Lab αποτελείται από 1006 σταθμούς, σε 487 διαφορετικές τοποθεσίες παγκοσμίως, σε ακαδημαϊκά και βιομηχανικά ιδρύματα και συνεχίζει να εξαπλώνεται ολοένα σε περισσότερα ιδρύματα.

4.3 Η Λειτουργία του Planet Lab

Κάθε site που συμμετέχει στο Planet Lab διαθέτει τουλάχιστον δύο σταθμούς που μπορούν να χρησιμοποιηθούν από οποιοδήποτε χρήστη ανήκει σε άλλο site εκτός από αυτό. Για να γίνει κάποιος χρήστης του Planet Lab πρέπει να εγγραφεί και να εγκριθεί η αίτηση του για εγγραφή από τους PI (Principal Investigators) του site, αυτοί είναι υπεύθυνοι στον αντίστοιχο οργανισμό που έγινε η εγγραφή. Μόλις εγκριθεί μια εγγραφή μπορεί ο αιτούμενος να έχει πρόσβαση στους σταθμούς του Planet Lab και να αρχίσει να τρέχει τις εφαρμογές του.

Οι πόροι του Planet Lab μοιράζονται σε μερίσματα (slices) τα οποία παρουσιάζονται ως ένα δίκτυο από εικονικές μηχανές. Κάθε οργανισμός ανά πάσα στιγμή μπορεί να έχει το πολύ 10 μερίσματα. Κάθε μέρισμα μπορεί να έχει δικαίωμα χρήσης το πολύ μέχρι 32 σταθμούς, όπου ο καθένας μόνο ένα μέρος των πόρων (εύρος ζώνης δικτύου, μνήμη, χώρος δίσκου) κάθε σταθμού είναι δυνατόν να έχει στη διάθεση του. Αρχικά το μέρισμα είναι κενό, χωρίς επεξεργαστές κειμένου ή κάποιο μεταγλωττιστή παρά μόνο τις βασικές εντολές του UNIX. Από τη δημιουργία ενός μερίσματος υπάρχει κάποιο χρονικό περιθώριο χρήσης του και όταν ξεπεραστεί αυτό, λήγει αποδεσμεύοντας του πόρους του. Εν τούτοις μπορεί να ανανεωθεί απεριόριστες φορές. Ο κάθε χρήστης δουλεύει σε κάθε σταθμό ανεξάρτητα από τους άλλους χρήστες χωρίς να «βλέπονται» μεταξύ τους αλλά όλοι μπορούν επηρεάζουν την απόδοση του σταθμού και του

δικτύου. Ακόμη κακόβουλες υπηρεσίες ή λογισμικά που βρίθουν από λάθη μπορούν να επηρεάσουν την επικοινωνία και την επίδοση άλλων μερισμάτων, γι αυτό επιβάλλονται αυστηροί όροι και προϋποθέσεις για την παροχή ασφάλειας στο Planet Lab. Επιπλέον οι σταθμοί μπορεί να επανεκκινήσουν και να μην είναι διαθέσιμοι ανά πάσα στιγμή, χωρίς να χαθούν οποιαδήποτε δεδομένα στο δίσκο, αλλά συμπεριφέρονται απρόσμενα έτσι ώστε να είναι μη αξιόπιστοι.

Η πρόσβαση στους σταθμούς μπορεί να γίνει μέσω SSH για ασφαλή, κρυπτογραφημένη επικοινωνία με τους σταθμούς. Αρχικά πρέπει ο χρήστης να δημιουργήσει ένα ζεύγος κρυπτογραφημένων κλειδιών SSH, private και public, και να κάνει γνωστό το public κλειδί του στο μέρισμα του. Αυτό γίνεται μέσω του λογαριασμού του χρήστη από την επίσημη ιστοσελίδα του Planet Lab[15]. Η δημιουργία των δύο κλειδιών μπορεί να γίνει με την εκτέλεση της εντολής:

```
ssh-keygen -t rsa -f ~/.ssh/id_rsa
```

Ο κάθε χρήστης είναι υπεύθυνος για την προστασία του private κλειδιού του έτσι ώστε να μην παραπέσει σε λάθος χέρια. Αν παραπέσει και χρησιμοποιηθεί ο λογαριασμός του χρήστη σε μη επιτρεπτές λειτουργίες, τότε το site και οι PIs του site θεωρούνται νομικά υπεύθυνοι.

Εφόσον ο χρήστης έχει γνωστοποιήσει το public κλειδί του στο μέρισμα του μπορεί να ενωθεί σε κάποια μηχανή με την πιο κάτω εντολή:

```
ssh -l slice_name -i ~/.ssh/id_rsa node
```

όπου `slice_name` είναι το όνομα του μερίσματος του χρήστη και node το όνομα της μηχανής που επιθυμεί να συνδεθεί. Μετά την εκτέλεση της εντολής θα ζητηθεί από το χρήστη η κωδική φράση για να πιστοποιηθεί η ταυτότητα του.

Τελειώνοντας ο χρήστης έχει τη δυνατότητα μεταφοράς των αρχείων του σε μια μηχανή είτε μέσω κάποιου ειδικού λογισμικού (όπως winscp) ή με την εκτέλεση της εντολής:

```
scp –i ~/.ssh/id_rsa –r filename slice_name@node:
```

Η παραπάνω εντολή μεταφέρει το αρχείο με όνομα `filename`, από τον τρέχων κατάλογο, στον προσωπικό χώρο του μερίσματος `slice_name` στη μηχανή `node`, η οποία ανήκει στο μέρισμα αυτό. Για να ολοκληρωθεί η μεταφορά είναι απαραίτητη η πιστοποίηση της ταυτότητας του χρήστη μέσω της κωδικής φράσης.

Η πιο κάτω εντολή χρησιμοποιείται για τον ίδιο σκοπό:

```
rsync –a –e "ssh –l slice_name –i ~/.ssh/id_rsa" filename node
```

η διαφορά είναι ότι αντιγράφει μόνο τα αρχεία που δεν υπάρχουν ήδη στη μηχανή node ή έχουν αλλάξει από την τελευταία φορά που αντιγράφηκαν. Όπου είναι δυνατό συνιστάται η χρήση της παρά την προηγούμενη για αποφυγή όσο δυνατό περισσότερης αχρείαστης «κίνησης» στο δίκτυο

Κεφάλαιο 5

Πειραματική Αξιολόγηση

Σε αυτό το κεφάλαιο περιγράφονται και αναλύονται τα πειράματα της υλοποίησης του αλγορίθμου SLIQ στο ασύγχρονο δίκτυο Planet Lab.

5.1 Προετοιμασία και Μεθοδολογία

Αρχικά δοκιμάστηκε η ορθότητα της υλοποίησης του αλγορίθμου που περιγράφηκε στο Κεφάλαιο 3 σε ένα μικρό σύστημα απαρτίας 6 εξυπηρετητών. Αφού πήραμε μια ένδειξη της ορθότητας του, η υλοποίηση προσαρμόστηκε κατάλληλα για την διεξαγωγή των πειραμάτων. Συγκεκριμένα, αλλαγές έγιναν στην εφαρμογή του αναγνώστη και του εγγραφέα. Προστέθηκε επιλογή στο μενού επιλογής, με δυνατότητα επιλογής πόσες αιτήσεις θα στέλνει η διεργασία. Επιπλέον γίνεται και ο υπολογισμός του χρόνου για την ολοκλήρωση μιας λειτουργίας είτε εγγραφής είτε ανάγνωσης. Στην περίπτωση της λειτουργίας ανάγνωσης υπολογίζεται ακόμη και ο αριθμός των αιτήσεων που χρειάστηκαν δεύτερο γύρο επικοινωνίας. Επιπρόσθετα οι εφαρμογές του εξυπηρετητή, του εγγραφέα και του αναγνώστη, κατά την εκκίνηση τους παίρνουν τώρα και τις τιμές των cInt, wInt και rInt και αντίστοιχα. Τα δύο πρώτα είναι το χρονικό διάστημα μεταξύ δύο αιτήσεων, είτε γραφής είτε ανάγνωσης. Το τρίτο είναι το κάθε πόσο χρόνο θα καλείται μια συνάρτηση για τον εξυπηρετητή. Αυτή η συνάρτηση κάθε φορά που

καλείται έχει 5% πιθανότητα να τερματίσει τη λειτουργία του εξυπηρετητή. Κατά αυτόν τον τρόπο προσομοιώνονται καταρρεύσεις των εξυπηρετητών, εκτός των καταρρεύσεων που υπάρχουν στο δίκτυο Plane Lab εκ φύσεως του.

Το επόμενο βήμα ήταν να μετρηθεί η καθυστέρηση λειτουργίας του αλγορίθμου(operation latency). Με άλλα λόγια μετρήθηκε ο μέσος χρόνος που χρειάζεται για μια διεργασία ανάγνωσης και εγγραφής να στείλει μια αίτηση και να πάρει απάντηση από ένα ολόκληρο quorum. Εκτός από την καθυστέρηση της λειτουργίας μετρήθηκε και ο μέσος χρόνος για την αποστολή ενός μηνύματος ping και βρέθηκαν 0,38s και 177,8ms αντίστοιχα. Η διαφορά μεταξύ αυτών των δύο χρόνων δείχνει πόσο χρόνο χρειάζεται περίπου μια συγκεκριμένη λειτουργία εγγραφής ή ανάγνωσης για να ολοκληρωθεί σε ένα γύρο επικοινωνίας. Στη συνέχεια η καθυστέρηση λειτουργίας διπλασιάστηκε και προστέθηκε στα wInt και rInt. Επειδή το δίκτυο είναι ασύγχρονο, δύο διεργασίες που στέλνουν μια αίτηση την ίδια χρονική στιγμή το πιο πιθανό είναι να μην ολοκληρώσουν τη λειτουργία τους και στο ίδιο χρονικό διάστημα. Βάζοντας μια διεργασία να περιμένει το διπλάσιο του χρόνου καθυστέρησης λειτουργίας, δίνεται αρκετός χρόνος σε όλες τις διεργασίες να ολοκληρώσουν τη λειτουργία τους. Έτσι πριν κάνει την επόμενη αίτηση η διεργασία, οι υπόλοιπες διεργασίες, ακόμα και αυτές που χρειαστήκαν δύο γύρους επικοινωνίας, είναι πιο πιθανό να έχουν ολοκληρώσει τη λειτουργία τους. Να σημειωθεί ότι έστω και αν έχει περάσει ο χρόνος περιόδου για μια λειτουργία (wInt ή rInt), για λόγους διατήρησης της ατομικότητας του αντικειμένου, η επόμενη λειτουργία δεν ξεκινά αν δεν έχει ολοκληρωθεί η προηγούμενη.

Για τη διεξαγωγή των αποτελεσμάτων στο κάθε ένα από τα παρακάτω σενάρια οι διεργασίες του εγγραφέα και του αναγνώστη έκαναν 200 αιτήσεις εγγραφής και ανάγνωσης αντίστοιχα. Επιπλέον το κάθε σενάριο έτρεξε πέντε φορές και κάθε μια από τις πέντε φορές χρειάστηκε περίπου 30 λεπτά για να ολοκληρώσει τις 200 αιτήσεις. Από τα αποτελέσματα που παρθήκαν αφαιρέθηκε αυτό που χρειάστηκε τον περισσότερο χρόνο για την ολοκλήρωση μιας αίτησης και με τους περισσότερους γύρους επικοινωνίας (στην περίπτωση του αναγνώστη). Παρόμοια αφαιρέθηκε και αυτό με το μικρότερο χρόνο ολοκλήρωσης μιας αίτησης και με τους λιγότερους γύρους επικοινωνίας (στην περίπτωση του αναγνώστη). Τα τελικά αποτελέσματα είναι ο μέσος όρος των υπόλοιπων τριών που απέμειναν.

5.2 Προβλήματα και Παραδοχές

Καθ' όλη τη διάρκεια εκτέλεσης των πειραμάτων παρατηρήθηκε η ασύγχρονη συμπεριφορά του δικτύου Planet Lab. Οι καθυστερήσεις για την παράδοση των μηνυμάτων δεν ήταν πάντοτε οι ίδιες και διάφεραν ανάλογα με την κατάσταση του δικτύου την τρέχων στιγμή. Η συμφόρηση του δικτύου έπαιζε σημαντικό ρόλο σε αυτό. Επίσης κάποιοι σταθμοί κάποτε ήταν πιο αργοί στη λειτουργία τους από κάποιους άλλους με αποτέλεσμα να καθυστερούν να απαντήσουν στις αιτήσεις που λάμβαναν. Έτσι οι χρόνοι ολοκλήρωσης των λειτουργιών δεν είχαν πάντα τις ίδιες τιμές κάθε φορά που έτρεχε ένα πείραμα.

Επιπλέον όσο αφορά το Planet Lab, ειπώθηκε ότι οι σταθμοί δεν είναι ούτε διαθέσιμοι συνεχώς ούτε λειτουργούν ορθά συνεχώς. Όταν ένας σταθμός γινόταν μη διαθέσιμος μπορεί να περνούσαν και μέρες ώσπου να γίνει ξανά διαθέσιμος σπαταλώντας χρόνο στην αποπεράτωση των πειραμάτων. Επίσης κάποιες φορές, σταθμοί που δεν λειτουργούσαν ορθά προκαλούσαν παράξενη συμπεριφορά στην εκτέλεση του αλγορίθμου στο σύστημα, όπως για παράδειγμα να προκαλούν segmentation fault σε σταθμούς που ήταν συνδεδεμένοι μαζί τους.

Κάτι άλλο που παρατηρήθηκε στον τρόπο λειτουργίας του δικτύου Planet Lab ήταν ότι κάποτε κάποιες μηχανές δεν ήταν προσβάσιμες για διάφορους λόγους, εκτός από το να μην είναι διαθέσιμοι όπως έχει προαναφερθεί. Για κάθε σταθμό του μερίσματος ενός χρήστη, κατά την πρώτη φορά που έχει πρόσβαση ο χρήστης σε αυτόν τον σταθμό φυλάγεται το αναγνωριστικό του σταθμού στο ~/.ssh/known_hosts. Το αναγνωριστικό ενός σταθμού δεν είναι το ίδιο, αλλά αλλάζει κατά διαστήματα για λόγους ασφαλείας. Έτσι όταν το αναγνωριστικό ενός σταθμού είχε αλλάξει ενώ στο ~/.ssh/known_hosts παρέμεινε το παλιό αναγνωριστικό, στην επόμενη προσπάθεια σύνδεσης με το σταθμό αυτό εμφανιζόταν το μήνυμα στο Σχήμα 5.1. Σε αυτή την περίπτωση έπρεπε να σβηστεί το παλιό αναγνωριστικό από το ~/.ssh/known_hosts για να γίνει επιτυχώς η σύνδεση στο σταθμό. Ακόμη όταν ένας σταθμός επανεκκινούσε κάποιες φορές εμφάνιζε το μήνυμα στο Σχήμα 5.2 κατά την προσπάθεια ένωσης με το σταθμό και για κάποιο χρονικό διάστημα δεν ήταν εφικτή η επιτυχής σύνδεση με αυτό το σταθμό.

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that the RSA host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
99:f7:b6:cc:ee:b5:c0:b2:b2:d2:69:4c:2a:f7:87:28.
Please contact your system administrator.
Add correct host key in /home/students/cs/2005/cs05ahl/.ssh/known_hosts to get rid of this message.
Offending key in /home/students/cs/2005/cs05ahl/.ssh/known_hosts:72
RSA host key for hptest-1.cs.princeton.edu has changed and you have requested strict checking.
Host key verification failed.
```

Σχήμα 5.1: Μήνυμα που εμφανίζεται όταν αλλάξει το μοναδικό αναγνωριστικό ενός σταθμού στο Planet Lab.

```
Enter passphrase for key '/home/students/cs/2005/cs05ahl/.ssh/id_rsa':
Last login: Wed May 20 13:04:31 2009 from 194.42.18.191
 *** planetlab1.pop-mg.rnp.br: cyprus_SLIQ has not been started yet, please chec
k back later ***
Connection to planetlab1.pop-mg.rnp.br closed.
```

Σχήμα 5.2: Μήνυμα που εμφανίζεται κατά την επανεκκίνηση ενός σταθμού.

Τέλος, ένα συχνό πρόβλημα που παρατηρήθηκε ήταν όταν ο αριθμός των διεργασιών στο σύστημα ήταν μεγάλος. Σε αυτή την περίπτωση οι εξυπηρετητές σταματούσαν να απαντούν σε ένα σύνολο των αναγνωστών ή και τον εγγραφέα ή ακόμη και σε όλες τις διεργασίες του συστήματος. Αυτό πιθανώς να οφειλόταν στο γεγονός ότι η εκτέλεση τους προγράμματος χρειαζόταν μεγάλο ποσοστό μνήμης της μηχανής, το οποίο δεν ήταν διαθέσιμο εκείνη τη στιγμή.

5.3 Σενάρια

Τα σενάρια τα οποία χρησιμοποιήθηκαν για την αξιολόγηση της συμπεριφοράς της υλοποίησης του αλγορίθμου SLIQ, είναι παρόμοια με αυτά που χρησιμοποιήθηκαν στις προσομοιώσεις του αλγορίθμου στο [6]. Στα παρακάτω σενάρια λαμβάνονται υπόψη τρεις διαφορετικές περιπτώσεις για τα rInt και wInt που αναφέρθηκαν πιο πάνω:

α) rInt < wInt: πιο συχνές λειτουργίες ανάγνωσης παρά λειτουργίες εγγραφής.

β) rInt = wInt: οι λειτουργίες ανάγνωσης και οι λειτουργίες για εγγραφή γίνονται σχεδόν ταυτόχρονα.

γ) rInt > wInt: πιο συχνές λειτουργίες για εγγραφή παρά λειτουργίες ανάγνωσης.

Στα σενάρια που πραγματοποιήθηκαν η τιμή του rInt είναι πάντα σταθερή και ίση με 4,3s. Οι τρεις πιο πάνω περιπτώσεις επιτεύχθηκαν μεταβάλλοντας τη τιμή του wInt ως εξής:

α) rInt < wInt: wInt = 6,3s.

β) rInt = wInt: wInt = 4,3s.

γ) rInt > wInt: wInt = 2,3s.

Στις πιο πάνω τιμές των rInt και wInt δεν περιλαμβάνεται το διπλάσιο του χρόνου καθυστέρησης της λειτουργίας που προαναφέρθηκε. Η τιμή για την καθυστέρηση λειτουργίας στα σενάρια που πραγματοποιήθηκαν είναι ίση με 3 δευτερόλεπτα, αρκετά μεγάλη για να ολοκληρώσουν όλες οι διεργασίες τη λειτουργία τους.

Όσο αφορά τα wInt και rInt λαμβάνονται υπόψη ακόμα δύο περιπτώσεις:

α) Εκτελέσεις με σταθερά χρονικά διαστήματα: όπου δύο συνεχόμενες λειτουργίες εγγραφής και ανάγνωσης διαφέρουν τουλάχιστον wInt και rInt αντίστοιχα.

β) Εκτελέσεις με μεταβλητά χρονικά διαστήματα: όπου δύο συνεχόμενες λειτουργίες εγγραφής και ανάγνωσης διαφέρουν τουλάχιστον ένα τυχαίο χρονικό διάστημα μεταξύ [operTime-wInt] και [operTime-rInt] αντίστοιχα.

Όπου operTime είναι η καθυστέρηση λειτουργίας και για τις δύο περιπτώσεις.

Σενάριο 1

Σε αυτό το σενάριο εξετάζεται η απόδοση της υλοποίησης του αλγορίθμου κάτω από παρόμοιες συνθήκες (quorums, σφάλματα) αλλά με διαφορετικό αριθμό αναγνωστών R, $R \in [10,20,40,80]$. Ο αλγόριθμος εξετάζεται σε τρία διαφορετικά συστήματα απαρτίας, το matrix, το crumbling walls και το majorities. Στα δύο πρώτα ο αριθμός των εξυπηρετητών που χρησιμοποιήθηκε είναι ίσος με 25 και στο τελευταίο είναι ίσος με 10. Επιπλέον η τιμή του cInt σε αυτό το σενάριο είναι ίση με 0, δηλαδή δεν προσομοιώνονται σφάλματα.

Σενάριο 2

Σε αυτό το σενάριο εξετάζεται η απόδοση του αλγορίθμου κάτω από τα συστήματα απαρτίας matrix και crumbling walls αλλά με διαφορετικό αριθμό αντιγράφων του αντικειμένου, δηλαδή των εξυπηρετητών S, S $\in$ [12,25,49]. Επιπλέον μεταβάλλεται όπως και στο πρώτο σενάριο ο αριθμός των αναγνωστών R, R$\in$[10,20,40,80]. Τέλος η τιμή του cInt σε αυτό το σενάριο είναι ίση με 0, δηλαδή δεν προσομοιώνονται σφάλματα.

Σενάριο 3

Σε αυτό το σενάριο εξετάζεται η απόδοση του αλγορίθμου στο σύστημα απαρτίας matrix, με διαφορετική τιμή της μεταβλητής cInt, cInt $\in$ [20,30,40]. Επιπλέον μεταβάλλεται όπως και στα προηγούμενα σενάρια ο αριθμός των αναγνωστών R, R$\in$[10,20,40,80].

5.4 Αποτελέσματα

Σενάριο 1

Σε αυτό το σενάριο αναμένεται ο χρόνος ολοκλήρωσης μιας λειτουργίας είτε ανάγνωσης είτε εγγραφής να αυξάνεται όσο αυξάνεται και ο αριθμός των αναγνωστών στο δίκτυο. Αυτό διότι όσο αυξάνεται ο αριθμός των αναγνωστών αυξάνεται και ο αριθμός των μηνυμάτων που στέλνονται στο δίκτυο, έτσι υπάρχει μεγαλύτερη συμφόρηση και τα μηνύματα καθυστερούν να φτάσουν στον προορισμό τους με αποτέλεσμα μια λειτουργία να καθυστερεί να ολοκληρωθεί. Επιπρόσθετα οι εξυπηρετητές δέχονται περισσότερο φόρτο αιτήσεων με αποτέλεσμα να καθυστερούν περισσότερο έως ότου να απαντήσουν σε όλες τις αιτήσεις.

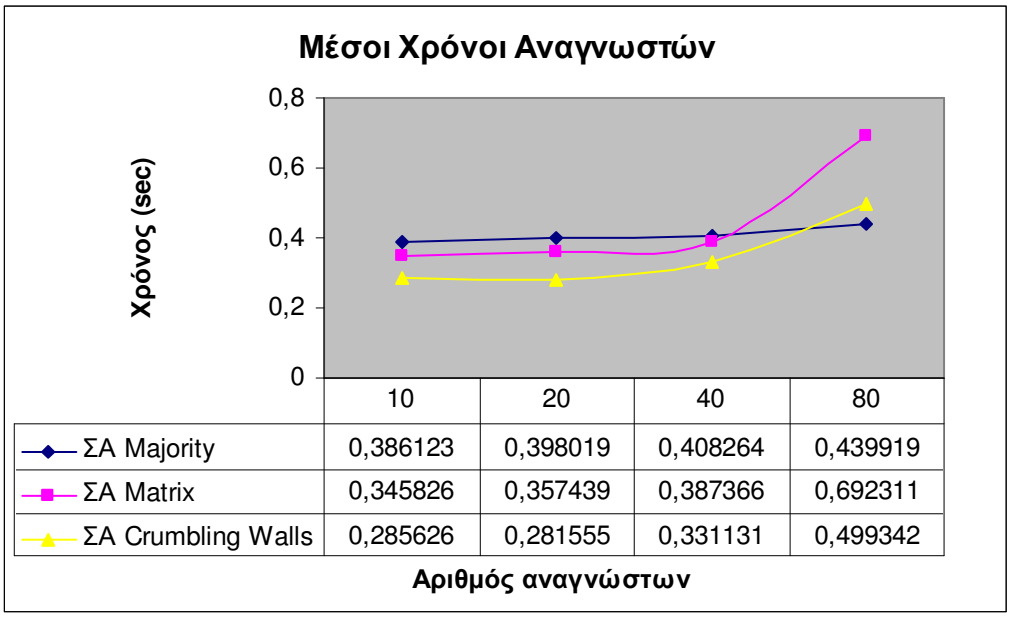
Στην περίπτωση σταθερών χρονικών διαστημάτων που rInt = wInt αναμένεται το ποσοστό δεύτερων γύρων επικοινωνίας, στους αναγνώστες, να είναι μεγαλύτερο από τις υπόλοιπες περιπτώσεις. Είναι πιο πιθανό σε αυτή την περίπτωση να γίνει κάποια

αίτηση ανάγνωσης καθώς γίνεται και μια νέα εγγραφή, εφόσον γίνονται σχεδόν ταυτόχρονα. Στην περίπτωση τυχαίων χρονικών διαστημάτων δεν αναμένεται τόσο αυξημένος ο αριθμός των δεύτερων γύρων επικοινωνίας εφόσον δε γίνονται ακριβώς ταυτόχρονα μια αίτηση εγγραφής και μια αίτηση ανάγνωσης. Αυτό το ποσοστό αναμένεται να μειωθεί ακόμα περισσότερο στις περιπτώσεις όπου rInt≠wInt, δηλαδή στις περιπτώσεις όπου rInt>wInt και rInt<wInt. Τα ίδια αναμένονται όσο αφορά το μέσο χρόνο ολοκλήρωσης μιας λειτουργίας εφόσον όσο πιο πολλές αιτήσεις γίνονται «ταυτόχρονα» τόσο πιο πολύ καθυστερούν οι λειτουργίες να ολοκληρωθούν για τους ίδιους λόγους που αναφέρθηκαν πιο πάνω.

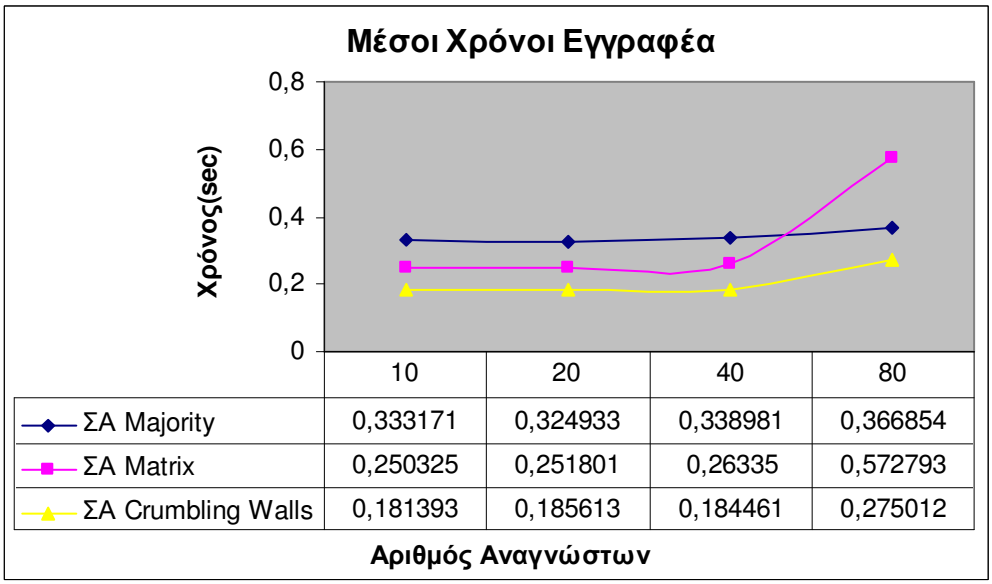
Τέλος αναμένεται ο χρόνος για την ολοκλήρωση μιας λειτουργίας εγγραφής να είναι μικρότερος από το χρόνο για την ολοκλήρωση μιας λειτουργίας ανάγνωσης. Αυτό γιατί κάποιες λειτουργίες ανάγνωσης χρειάζονται δύο γύρους επικοινωνίας για να ολοκληρωθούν ενώ οι λειτουργίες εγγραφής ολοκληρώνονται πάντα στον πρώτο γύρο επικοινωνίας.

Όπως φαίνεται μέσα από τα Γράφημα 5.1 και Γράφημα 5.2, ο χρόνος για την ολοκλήρωση μιας λειτουργίας εγγραφής ή ανάγνωσης αυξάνεται όσο αυξάνεται ο αριθμός των αναγνωστών όπως ήταν αναμενόμενο. Μέσα από αυτά τα γραφήματα βλέπουμε ότι οι μέσοι χρόνοι αυξάνονται εκθετικά στα crumbling walls και matrix συστήματα απαρτίας ενώ στο majority γραμμικά. Επίσης αρχικά ο μέσος χρόνος ολοκλήρωσης μιας λειτουργίας είναι μεγαλύτερος στα majority και matrix συστήματα απαρτίας από ότι στο crumbling walls. Για R=80, τα τελευταία δύο έχουν μεγαλύτερους χρόνους ολοκλήρωσης μιας λειτουργίας από το majority σύστημα απαρτίας στους αναγνώστες ενώ στον εγγραφέα το matrix σύστημα απαρτίας έχει μεγαλύτερους χρόνους για την ολοκλήρωση μιας λειτουργίας από τα υπόλοιπα. Επίσης παρατηρούμε ότι ως R=40 τα matrix και crumbling walls συστήματα απαρτίας διατηρούν σταθερή διαφορά περίπου 0,06 δευτερόλεπτα ενώ για R=80 η διαφορά αυτή αυξάνεται σε 2 περίπου δευτερόλεπτα στους αναγνώστες στον εγγραφέα περίπου 3 δευτερόλεπτα. Ήταν αναμενόμενο το σύστημα απαρτίας majority να παρουσιάσει χειρότερους χρόνους από τα άλλα δύο λόγω της μεγάλης τομής μεταξύ των quorums του. Σε αντίθεση το σύστημα απαρτίας crumbling walls ήταν αναμενόμενο για να έχει καλύτερους χρόνους λόγω του μεταβλητού μεγέθους των quorums του και των μεταξύ τους τομών.

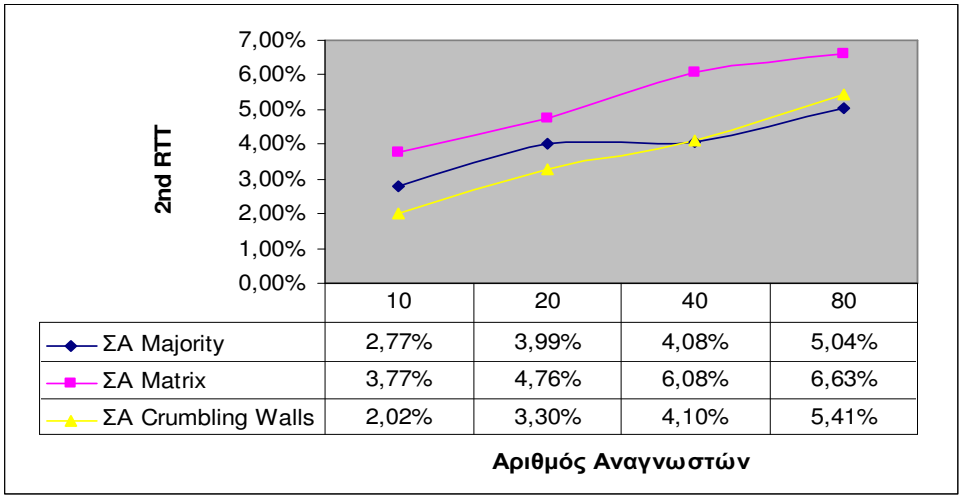
Στο Γράφημα 5.3 παρατηρείται, ότι το ποσοστό των δεύτερων γύρων επικοινωνίας αυξάνεται καθώς αυξάνεται και αριθμός των αναγνωστών στο σύστημα. Όπως αναφέρθηκε, λόγω της αύξησης των μηνυμάτων προκαλούνται καθυστερήσεις στο δίκτυο με αποτέλεσμα οι λειτουργίες να καθυστερούν να ολοκληρωθούν. Συνεπώς μια λειτουργία ανάγνωσης έχει περισσότερες πιθανότητες να συμπέσει τη στιγμή που πραγματοποιείται και μια λειτουργία εγγραφής και να εκτελέσει δεύτερο γύρο επικοινωνίας. Ακόμη όπως αναμενόταν το ποσοστό των δεύτερων γύρων επικοινωνίας είναι μεγαλύτερο στην περίπτωση των σταθερών χρονικών διαστημάτων όπου rInt=wInt και μικρότερο στις άλλες περιπτώσεις όπως φαίνεται στο Γράφημα 5.4. Επιπρόσθετα στο Γράφημα 5.4 παρατηρούμε ότι το ποσοστό αυτό δεν είναι πάντα πιο μεγάλο στην περίπτωση των μεταβλητών χρονικών διαστημάτων όπου rInt=wInt από τις υπόλοιπες περιπτώσεις όπως αναμενόταν λόγω της ασυγχρονίας του δικτύου. Τέλος στο Γράφημα 5.3 παρατηρείται ότι το matrix σύστημα απαρτίας εκτελεί μεγαλύτερο ποσοστό δεύτερων γύρων επικοινωνίας από τα υπόλοιπα, κάτι που αναμενόταν για το majority σύστημα απαρτίας λόγω της μεγάλης τομής μεταξύ των quorums του. Παρόλα αυτά αν δούμε στο Γράφημα 5.4 πιο λεπτομερώς και τα τρία συστήματα απαρτίας, όλα εκτελούν ένα ποσοστό περίπου μεταξύ 3-6% δεύτερων γύρων επικοινωνίας καθώς αυξάνεται ο αριθμός των αναγνωστών, εκτός στις περιπτώσεις που wInt=rInt. Σε αυτές τις περιπτώσεις τα majority και crumbling walls συστήματα απαρτίας φτάνουν περίπου έως το 8% και 12% αντίστοιχα. Εντούτοις τα ποσοστά αυτά είναι σχετικά με πόσο «ταυτόχρονα» εκτελούνται οι λειτουργίες και σε μια άλλη εκτέλεση του σεναρίου να δώσουν διαφορετικά αποτελέσματα. Όμως όπως αναμενόταν αυτά τα δύο συστήματα εκτελούν περισσότερους γύρους επικοινωνίας από το crumbling walls.



Γράφημα 5.1 Ο μέσος χρόνος των λειτουργιών ανάγνωσης όσο αυξάνεται ο αριθμός των αναγνωστών στα τρία Συστήματα Απαρτίας.

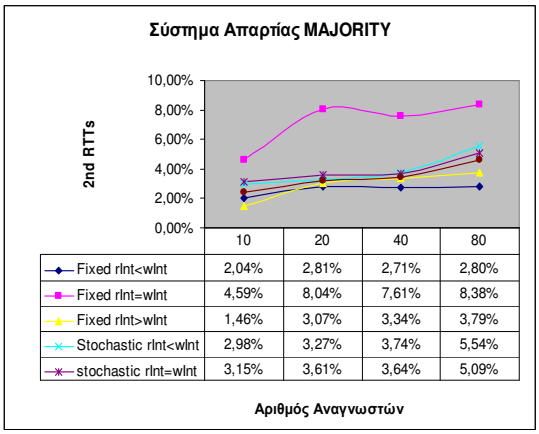


Γράφημα 5.2 Ο μέσος χρόνος των λειτουργιών εγγραφής όσο αυξάνεται ο αριθμός των αναγνωστών στα τρία Συστήματα Απαρτίας.

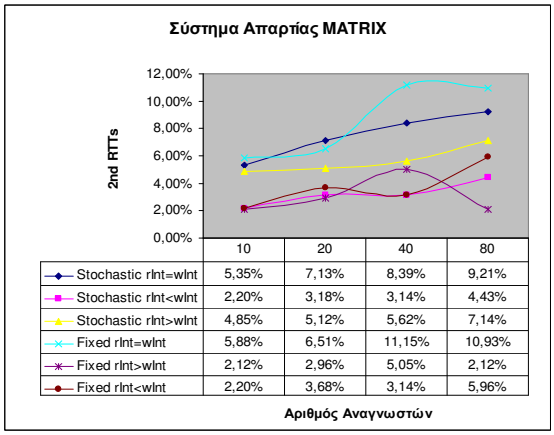


	10	20	40	80
ΣΑ Majority	2,77%	3,99%	4,08%	5,04%
ΣΑ Matrix	3,77%	4,76%	6,08%	6,63%
ΣΑ Crumbling Walls	2,02%	3,30%	4,10%	5,41%

Γράφημα 5.3 Το ποσοστό των δεύτερων γύρων επικοινωνίας όσο αυξάνεται ο αριθμός των αναγνωστών στα τρία Συστήματα Απαρτίας.



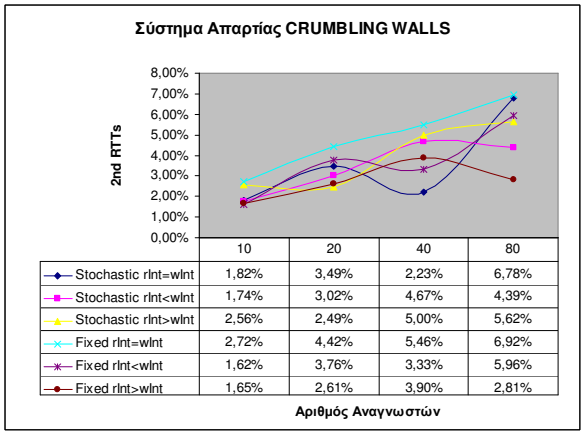
	10	20	40	80
Fixed rInt<wInt	2,04%	2,81%	2,71%	2,80%
Fixed rInt=wInt	4,59%	8,04%	7,61%	8,38%
Fixed rInt>wInt	1,46%	3,07%	3,34%	3,79%
Stochastic rInt<wInt	2,98%	3,27%	3,74%	5,54%
stochastic rInt=wInt	3,15%	3,61%	3,64%	5,09%



	10	20	40	80
Stochastic rInt=wInt	5,35%	7,13%	8,39%	9,21%
Stochastic rInt<wInt	2,20%	3,18%	3,14%	4,43%
Stochastic rInt>wInt	4,85%	5,12%	5,62%	7,14%
Fixed rInt=wInt	5,88%	6,51%	11,15%	10,93%
Fixed rInt>wInt	2,12%	2,96%	5,05%	2,12%
Fixed rInt<wInt	2,20%	3,68%	3,14%	5,96%

(α)

(β)

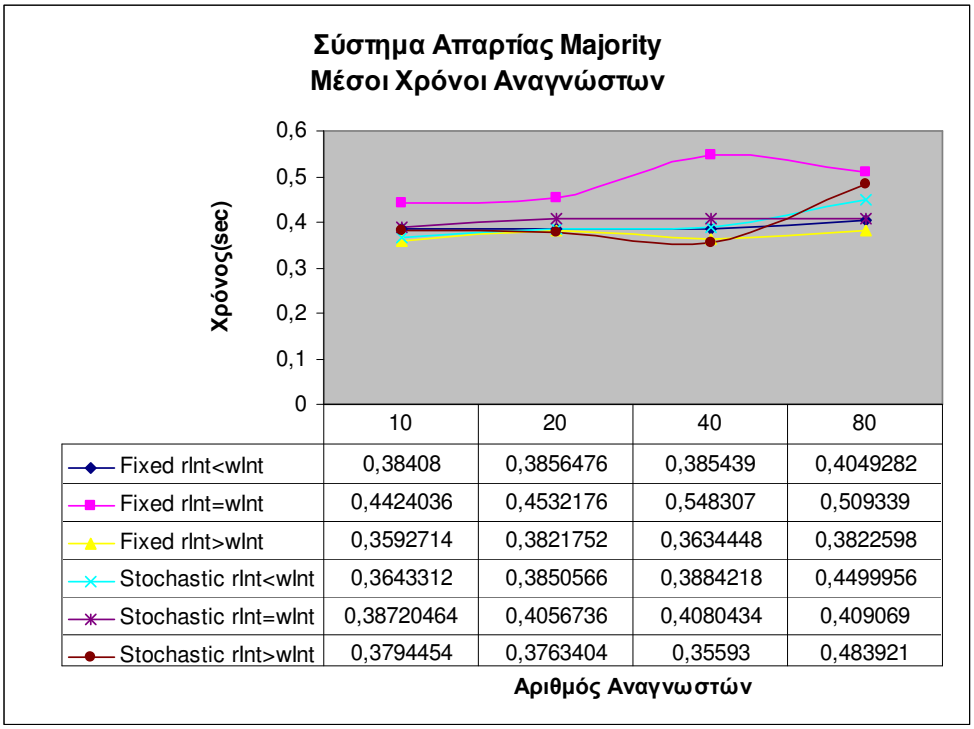


	10	20	40	80
Stochastic rInt=wInt	1,82%	3,49%	2,23%	6,78%
Stochastic rInt<wInt	1,74%	3,02%	4,67%	4,39%
Stochastic rInt>wInt	2,56%	2,49%	5,00%	5,62%
Fixed rInt=wInt	2,72%	4,42%	5,46%	6,92%
Fixed rInt<wInt	1,62%	3,76%	3,33%	5,96%
Fixed rInt>wInt	1,65%	2,61%	3,90%	2,81%

(γ)

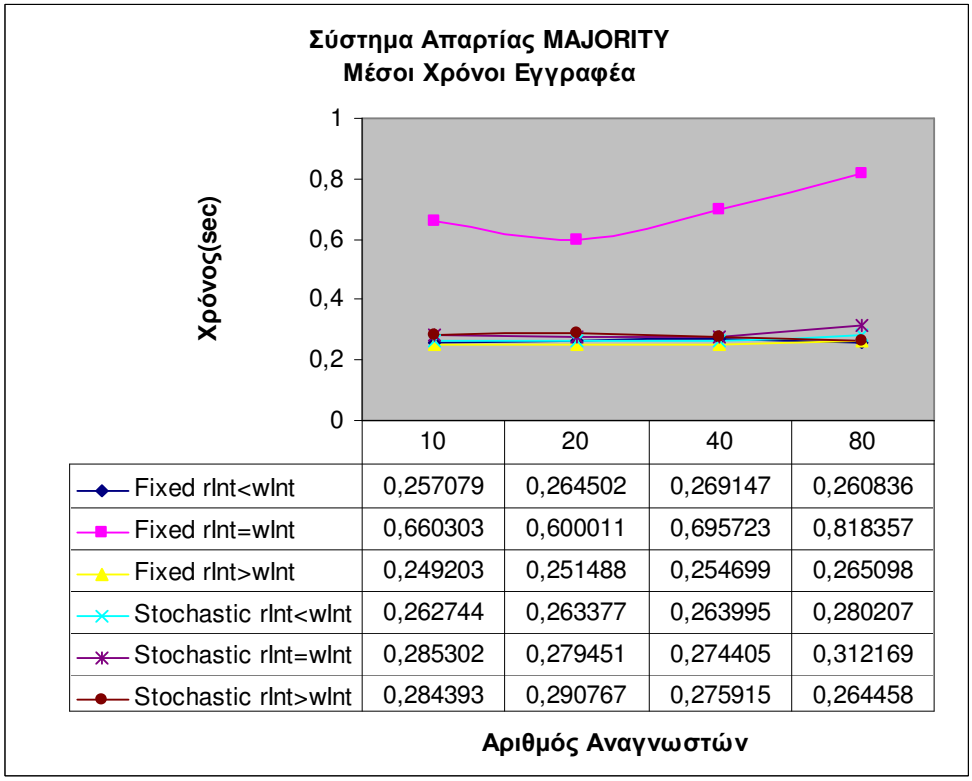
Γράφημα 5.4 Το ποσοστό των δεύτερων γύρων επικοινωνίας όσο αυξάνεται ο αριθμός των αναγνωστών αναλυτικά για τα τρία Συστήματα Απαρτίας.
(α)Majority (β)Matrix (γ) Crumbling Walls

Τελειώνοντας να πούμε ότι σε αυτό το σενάριο παρατηρήθηκε ότι ο χρόνος για την ολοκλήρωση μιας λειτουργίας ανάγνωσης δεν είναι πάντοτε μεγαλύτερος από το χρόνο ολοκλήρωσης μιας λειτουργίας εγγραφής. Όπως για παράδειγμα στο Γράφημα 5.5 και Γράφημα 5.6 όπου στην περίπτωση σταθερών χρονικών διαστημάτων που rInt=wInt, ο χρόνος της λειτουργίας εγγραφής είναι μεγαλύτερος από αυτόν της λειτουργίας ανάγνωσης στο σύστημα απαρτίας majority. Πιθανώς σε αυτή την περίπτωση οι αιτήσεις από τον εγγραφέα να συναντούσαν περισσότερη συμφόρηση μέσα στο δίκτυο. Εκτός αυτής της περίπτωσης, στις πλείστες περιπτώσεις ο μέσος χρόνος των λειτουργιών ανάγνωσης είναι μεγαλύτερος από το μέσο χρόνο λειτουργιών εγγραφής όπως αναμενόταν. Τα γραφήματα 5.7-5.9 συνοψίζουν τους μέσους χρόνους των λειτουργιών εγγραφής και ανάγνωσης για τα τρία συστήματα απαρτίας όπου φαίνεται το πιο πάνω.



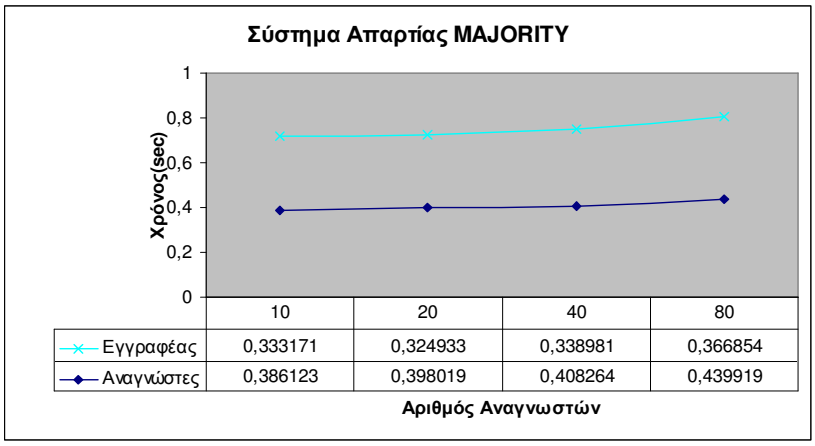
	10	20	40	80
Fixed rInt<wInt	0,38408	0,3856476	0,385439	0,4049282
Fixed rInt=wInt	0,4424036	0,4532176	0,548307	0,509339
Fixed rInt>wInt	0,3592714	0,3821752	0,3634448	0,3822598
Stochastic rInt<wInt	0,3643312	0,3850566	0,3884218	0,4499956
Stochastic rInt=wInt	0,38720464	0,4056736	0,4080434	0,409069
Stochastic rInt>wInt	0,3794454	0,3763404	0,35593	0,483921

Γράφημα 5.5 Ο μέσος χρόνος των λειτουργιών ανάγνωσης όσο αυξάνεται ο αριθμός των αναγνωστών σε Majority σύστημα απαρτίας.

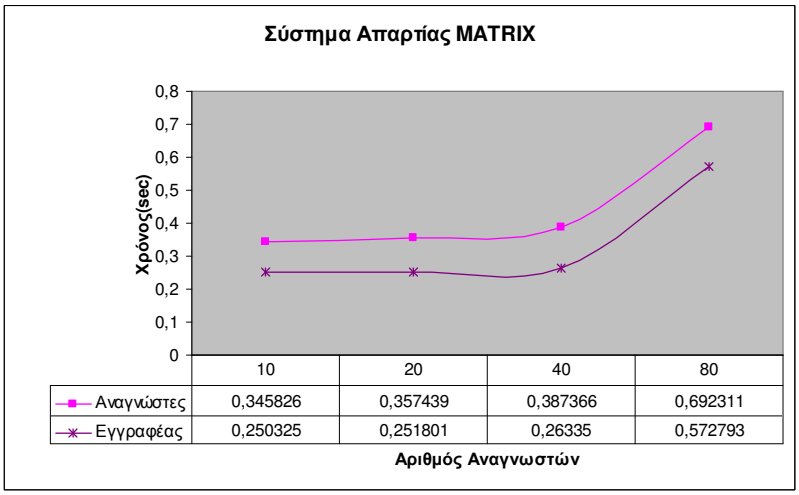


	10	20	40	80
Fixed rInt<wInt	0,257079	0,264502	0,269147	0,260836
Fixed rInt=wInt	0,660303	0,600011	0,695723	0,818357
Fixed rInt>wInt	0,249203	0,251488	0,254699	0,265098
Stochastic rInt<wInt	0,262744	0,263377	0,263995	0,280207
Stochastic rInt=wInt	0,285302	0,279451	0,274405	0,312169
Stochastic rInt>wInt	0,284393	0,290767	0,275915	0,264458

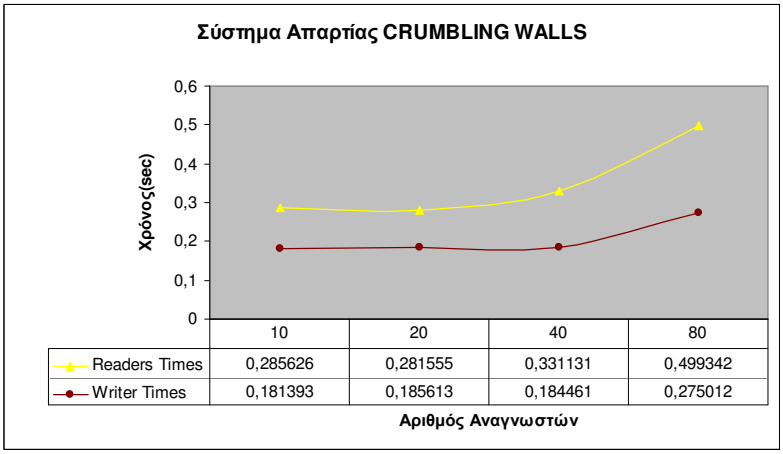
Γράφημα 5.6 Ο μέσος χρόνος των λειτουργιών εγγραφής όσο αυξάνεται ο αριθμός των αναγνωστών σε Majority σύστημα απαρτίας.



Γράφημα 5.7 Ο μέσος χρόνος των λειτουργιών εγγραφής και ανάγνωσης όσο αυξάνεται ο αριθμός των αναγνωστών σε Majority σύστημα απαρτίας.



Γράφημα 5.8 Ο μέσος χρόνος των λειτουργιών εγγραφής και ανάγνωσης όσο αυξάνεται ο αριθμός των αναγνωστών σε Matrix σύστημα απαρτίας.

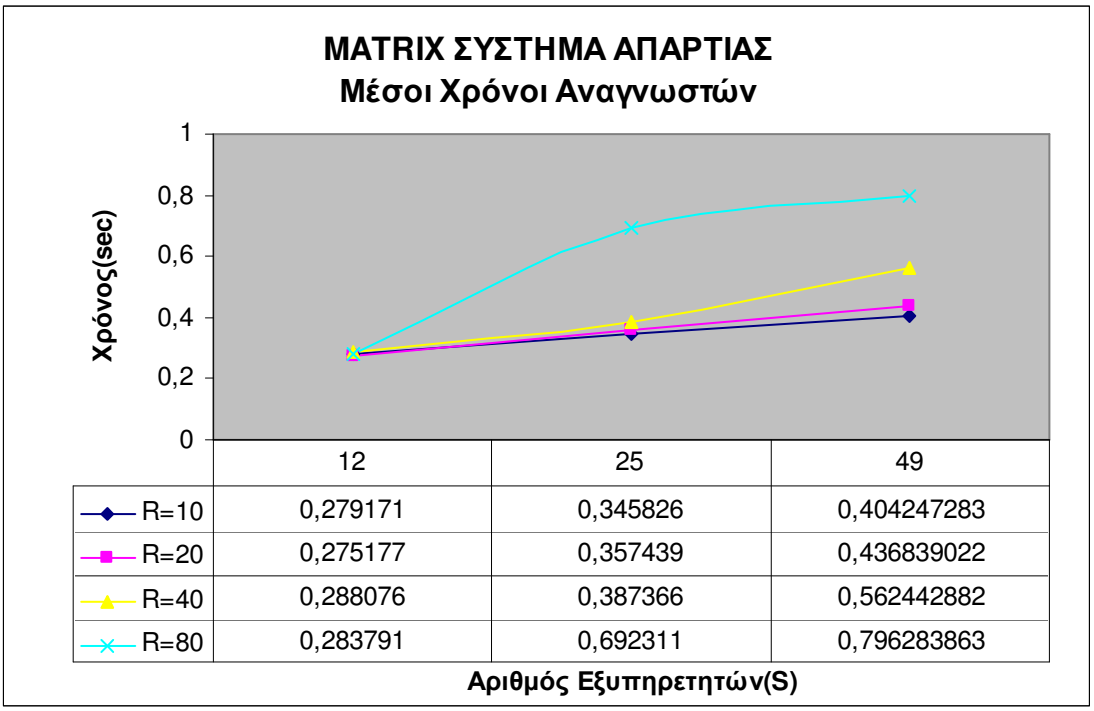


Γράφημα 5.9 Ο μέσος χρόνος των λειτουργιών εγγραφής και ανάγνωσης όσο αυξάνεται ο αριθμός των αναγνωστών σε Crumbling Walls σύστημα απαρτίας.

Σενάριο 2

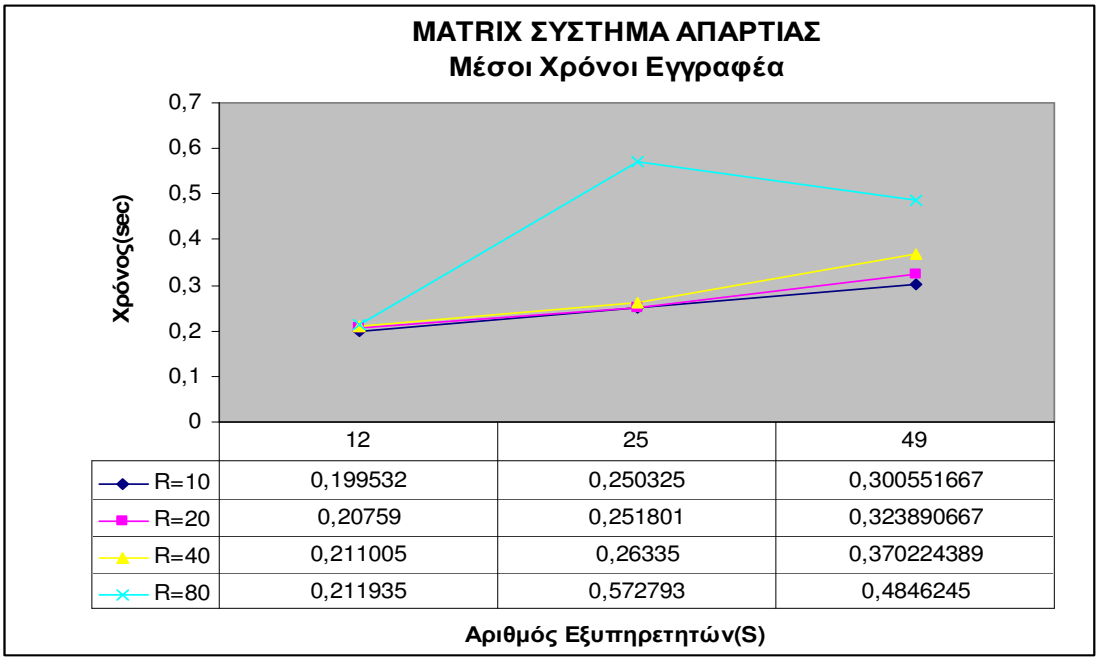
Αυτό που αναμένεται σε αυτό το σενάριο είναι ότι καθώς αυξάνεται ο αριθμός των αντιγράφων του αντικειμένου, των εξυπηρετητών, να αυξάνεται και ο μέσος χρόνος ολοκλήρωσης των λειτουργιών ανάγνωσης και εγγραφής. Αυτό για τον ίδιο λόγο που αυξανόταν ο μέσος χρόνος των λειτουργιών στο Σενάριο 1 καθώς αυξανόταν ο αριθμός των αναγνωστών. Οι αναγνώστες στέλνουν μηνύματα σε περισσότερους εξυπηρετητές και η αύξηση του όγκου των μηνυμάτων στο δίκτυο προκαλεί μεγαλύτερες καθυστερήσεις με αποτέλεσμα την καθυστέρηση της ολοκλήρωσης των λειτουργιών. Πράγματι τα αποτελέσματα που πάρθηκαν έδειξαν το πιο πάνω. Το Γράφημα 5.10 συνοψίζει τα αποτελέσματα του σεναρίου αυτού για το σύστημα απαρτίας matrix. Παρά ταύτα στο Γράφημα 5.11, που συνοψίζει τα αποτελέσματα για σύστημα απαρτίας Crumbling Walls, βλέπουμε ότι το S=49 αποτελεί εξαίρεση. Αυτό πιθανώς να οφείλεται στην κατάσταση του δικτύου Planet Lab όταν εξετάστηκε αυτή η περίπτωση. Πιθανώς οι σταθμοί να είχαν μεγάλο φόρτο εργασίας και να ήταν πιο αργοί ή και να υπήρχε μεγαλύτερη συμφόρηση στο δίκτυο.

Όσο αφορά το ποσοστό των δεύτερων γύρων επικοινωνίας, σε αυτό το σενάριο παρατηρούμε όπως και στο σενάριο 1 να αυξάνεται όσο αυξάνεται ο αριθμός των εξυπηρετητών για τους ίδιους λόγους (Γράφημα 5.12).



	12	25	49
R=10	0,279171	0,345826	0,404247283
R=20	0,275177	0,357439	0,436839022
R=40	0,288076	0,387366	0,562442882
R=80	0,283791	0,692311	0,796283863

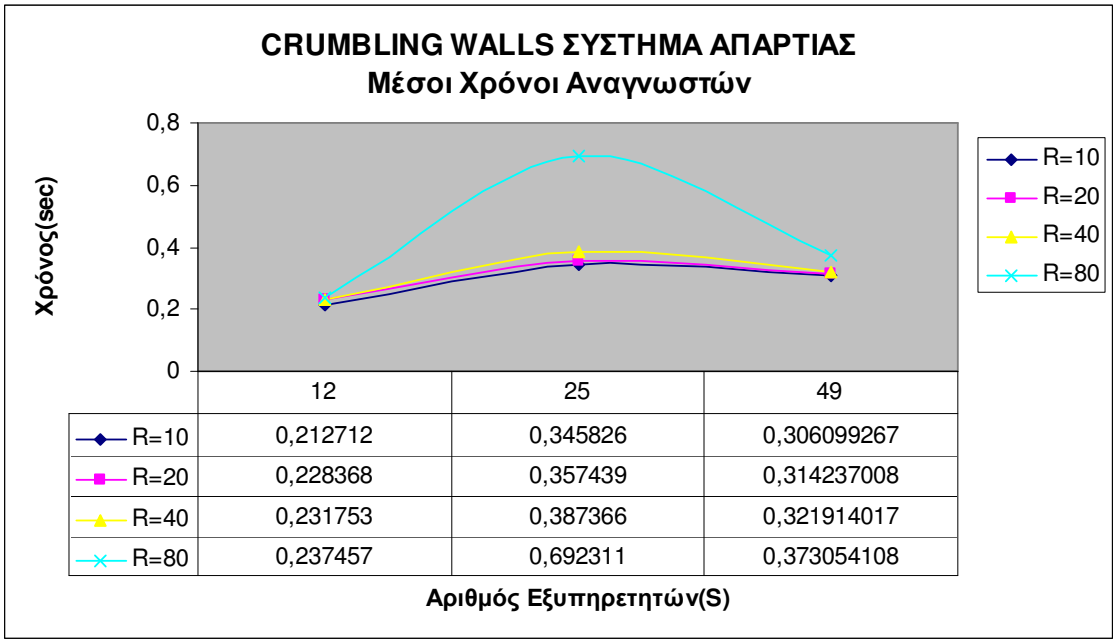
(α)



	12	25	49
R=10	0,199532	0,250325	0,300551667
R=20	0,20759	0,251801	0,323890667
R=40	0,211005	0,26335	0,370224389
R=80	0,211935	0,572793	0,4846245

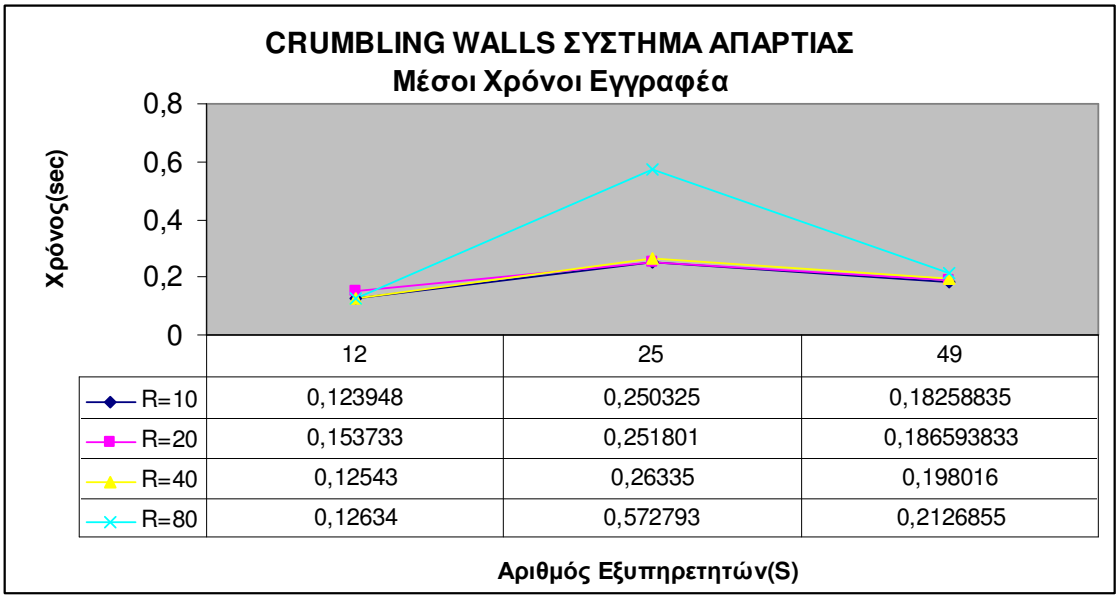
(β)

Γράφημα 5.10 Ο μέσος χρόνος των λειτουργιών εγγραφής(β) και ανάγνωσης(α) όσο αυξάνεται ο αριθμός των εξυπηρετητών σε Matrix σύστημα απαρτίας, για R=10,20,40,80.



	12	25	49
R=10	0,212712	0,345826	0,306099267
R=20	0,228368	0,357439	0,314237008
R=40	0,231753	0,387366	0,321914017
R=80	0,237457	0,692311	0,373054108

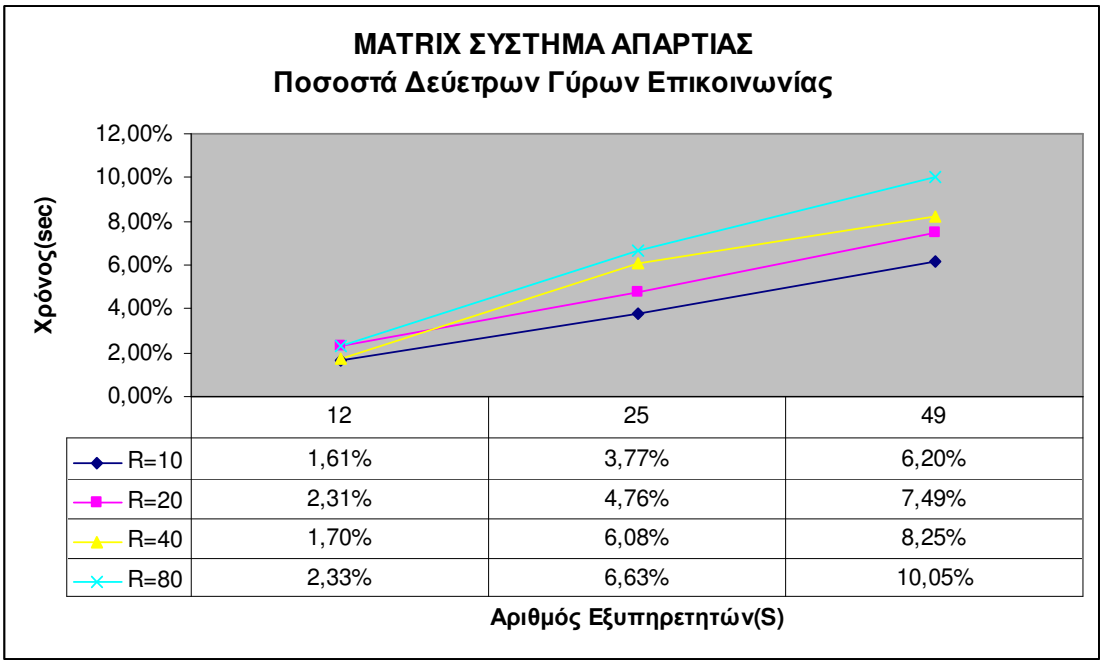
(α)



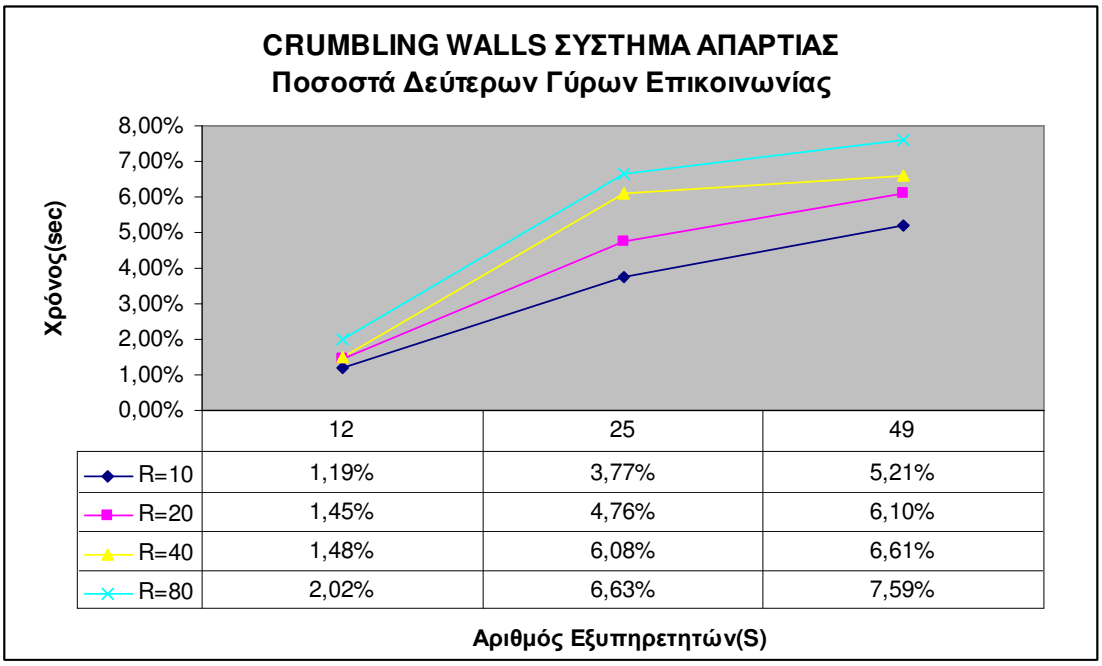
	12	25	49
R=10	0,123948	0,250325	0,18258835
R=20	0,153733	0,251801	0,186593833
R=40	0,12543	0,26335	0,198016
R=80	0,12634	0,572793	0,2126855

(β)

Γράφημα 5.11 Ο μέσος χρόνος των λειτουργιών εγγραφής(β) και ανάγνωσης(α) όσο αυξάνεται ο αριθμός των εξυπηρετητών σε Crumbling Walls σύστημα απαρτίας, για R=10,20,40,80.



(α)



(β)

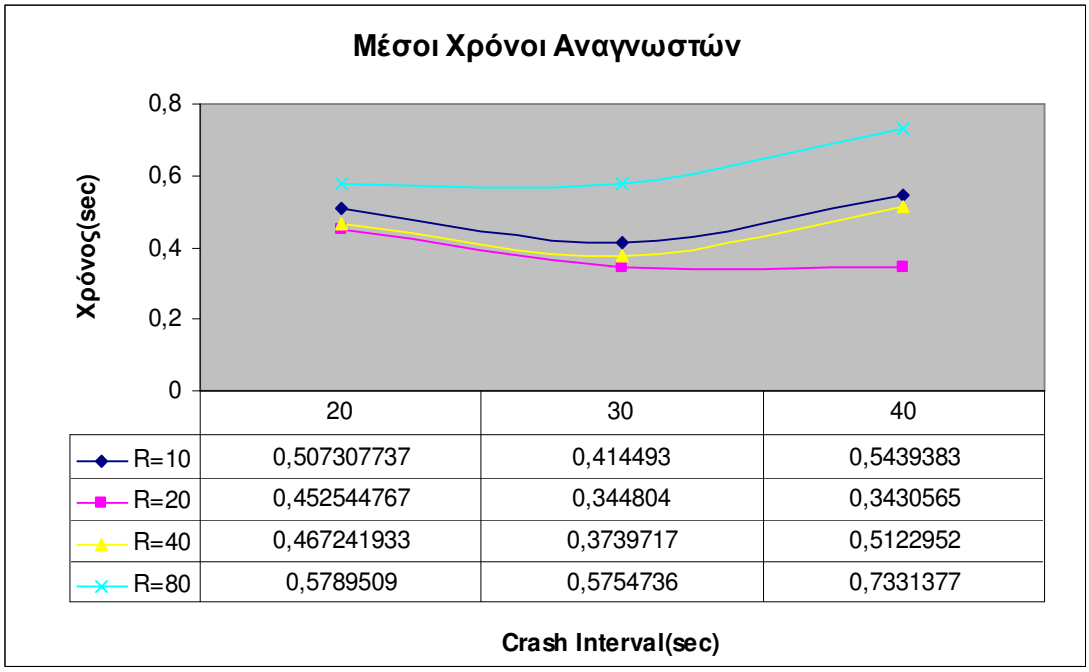
Γράφημα 5.12: Ποσοστά δεύτερων γύρων επικοινωνίας των αναγνωστών. (α) Για Matrix Σύστημα Απαρτίας (β) Για Crumbling Walls Σύστημα Απαρτίας

Σενάριο 3

Σε αυτό το σενάριο προσομοιώνονται σφάλματα στην υλοποίηση του αλγορίθμου, καθορίζοντας τη συχνότητα των σφαλμάτων με τη μεταβλητή cInt. Αναμένεται όσο πιο μικρή είναι η τιμή αυτής της μεταβλητής τόσο πιο γρήγορα το σύστημα απαρτίας να μικραίνει (μεγαλώνει ο αριθμός των εξυπηρετητών που καταρρέουν) και να συγκλίνει προς το non-faulty quorum, δηλαδή το quorum που προϋποθέτουμε ότι δε θα έχει ούτε ένα εξυπηρετητή επιρρεπή σε σφάλματα(non-faulty). Επιπλέον αναμένεται όσο μειώνεται ο αριθμός των εξυπηρετητών στο σύστημα να μειώνεται και το ποσοστό των δεύτερων γύρων επικοινωνίας, συνεπώς και του μέσου χρόνου ολοκλήρωσης των λειτουργιών ανάγνωσης. Αυτό επειδή οι τομές στα quorums θα μικραίνουν έτσι θα συμβαίνουν σπανιότερα δεύτεροι γύροι επικοινωνίας.

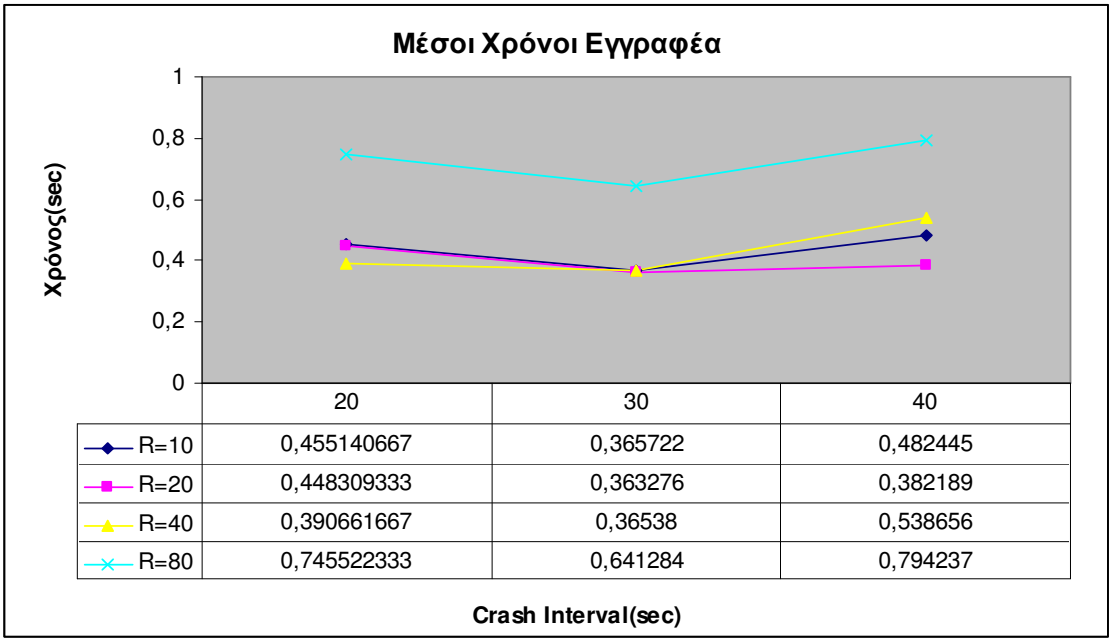
Όντως κατά τη διάρκεια εκτέλεσης του σεναρίου παρατηρήθηκε ότι όσο πιο μικρό ήταν το cInt τόσο πιο γρήγορα σύγκλινε στο quorum το οποίο δεν είναι επιρρεπή στα σφάλματα. Συγκεκριμένα για cInt=10 σχεδόν όλες τις φορές που εκτελέστηκε, στο τέλος της ολοκλήρωσης της εκτέλεσης παρατηρήθηκε να έχει παραμείνει μόνο το quorum που δεν ήταν επιρρεπή στα σφάλματα ενώ όλοι οι υπόλοιποι εξυπηρετητές να έχουν καταρρεύσει. Για cInt=30 και cInt=40 αυτό δε συνέβηκε σχεδόν καθόλου.

Στο Γράφημα 5.13 παρατηρείτε όπως αναμενόταν να αυξάνεται ο μέσος χρόνος των λειτουργιών όσο αυξάνεται το cInt. Όπως είπαμε όσο αυξάνεται το cInt οι εξυπηρετητές καταρρέουν με μικρότερους ρυθμούς. Έτσι όπως είδαμε και από τα προηγούμενα σενάρια, όσο πιο μεγάλος είναι ο αριθμός των επεξεργαστών τόσο πιο μεγάλοι είναι και οι μέσοι χρόνοι ολοκλήρωσης των λειτουργιών ανάγνωσης και εγγραφής. Παρατηρούμε ότι το cInt=40 και R=80 αποτελεί εξαίρεση.



	20	30	40
R=10	0,507307737	0,414493	0,5439383
R=20	0,452544767	0,344804	0,3430565
R=40	0,467241933	0,3739717	0,5122952
R=80	0,5789509	0,5754736	0,7331377

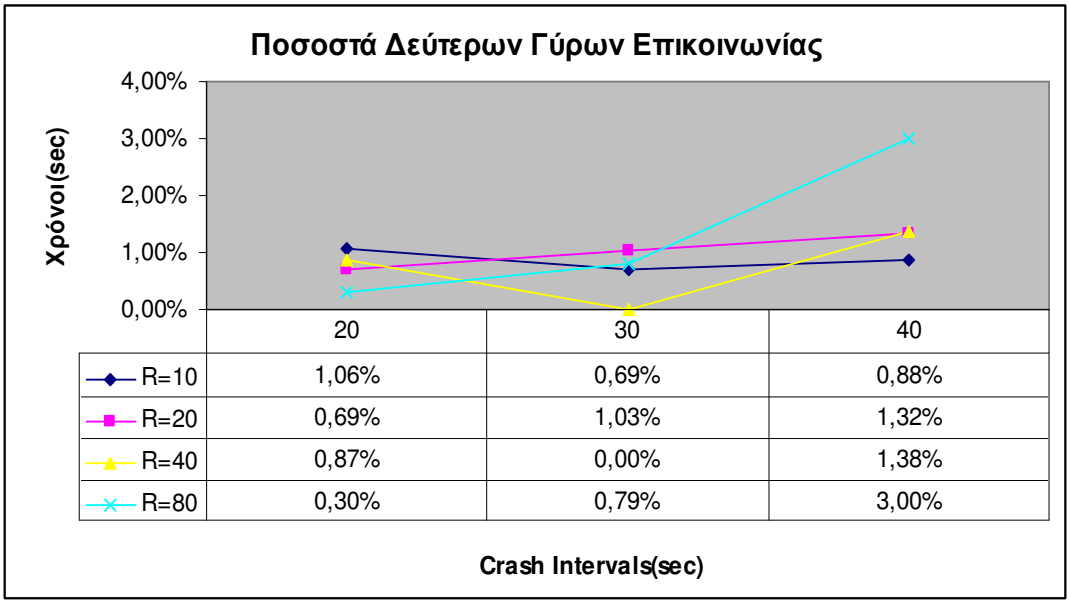
(α)



	20	30	40
R=10	0,455140667	0,365722	0,482445
R=20	0,448309333	0,363276	0,382189
R=40	0,390661667	0,36538	0,538656
R=80	0,745522333	0,641284	0,794237

(β)

Γράφημα 5.13: Μέσοι χρόνοι λειτουργιών ανάγνωσης (α) και εγγραφής (β) καθώς αυξάνεται η τιμή του cInt.



	20	30	40
R=10	1,06%	0,69%	0,88%
R=20	0,69%	1,03%	1,32%
R=40	0,87%	0,00%	1,38%
R=80	0,30%	0,79%	3,00%

Γράφημα 5.14: Ποσοστά δεύτερων γύρων επικοινωνίας καθώς αυξάνεται η τιμή του cInt.

5.5 Παρατηρήσεις και Συμπεράσματα

Κατά τη διάρκεια διεξαγωγής των πειραμάτων παρατηρήθηκαν πολλά από τα χαρακτηριστικά του ερευνητικού δικτύου Planet Lab, χαρακτηριστικά οποιουδήποτε ρεαλιστικού ασύγχρονου δικτύου. Συγκεκριμένα παρατηρήθηκε κατάρρευση κάποιων από των μηχανών, μηχανές που σταματούσαν να ανταποκρίνονται και η συμφόρηση στο δίκτυο που δεν ήταν πάντοτε η ίδια. Το τελευταίο παρατηρήσαμε να επηρεάζει τις μετρήσεις που κάναμε με αποτέλεσμα να μην έχουμε πάντοτε τα αναμενόμενα αποτελέσματα.

Το ποσοστό των λειτουργιών που απαιτούν δεύτερο γύρο επικοινωνίας δε ξεπερνά το 12%, στη χειρότερη περίπτωση σταθερών χρονικών διαστημάτων που rInt = wInt. Στις υπόλοιπες περιπτώσεις το ποσοστό αυτό δε ξεπερνά το 6%. Σε σύγκριση με τα αποτελέσματα των προσομοιώσεων στο [6], που στη χειρότερη περίπτωση το αντίστοιχο ποσοστό των λειτουργιών που χρειάζονται δεύτερο γύρο επικοινωνίας είναι πάνω από 85% και στις υπόλοιπες περιπτώσεις δε ξεπερνά το 12%, είναι πολύ ικανοποιητική η συμπεριφορά του αλγορίθμου σε πραγματικό, ασύγχρονο δίκτυο. Η μεγάλη διαφορά των αποτελεσμάτων στη χειρότερη περίπτωση όπου οι λειτουργίες ανάγνωσης εκτελούνται «ταυτόχρονα» με τις λειτουργίες εγγραφής οφείλεται στην

ασυγχρονία του δικτύου. Λόγω της ασυγχρονίας του δικτύου είναι πιο απίθανο να εκτελεστούν εντελώς ταυτόχρονα δύο ή περισσότερες λειτουργίες, σε αντίθεση με τις προσομοιώσεις που η ταυτοχρονία είναι πιθανή.

Κεφάλαιο 6

Συμπεράσματα

6.1 Γενικά Συμπεράσματα

Σε αυτή τη διπλωματική εργασία υλοποιήθηκε και αξιολογήθηκε ο ασθενές-ημιγρήγορος αλγόριθμος γραφής/ανάγνωσης ατομικών αντικειμένων όπως περιγράφεται στο [6].

Όπως παρατηρήθηκε από τα αποτελέσματα του προηγούμενου κεφαλαίου οι αργές λειτουργίες ανάγνωσης, που χρειάζονται δεύτερο γύρο επικοινωνίας για να ολοκληρωθούν δεν ξεπερνούν το 12% στη χειρότερη περίπτωση ενώ στις υπόλοιπες περιπτώσεις δεν ξεπερνούν το 6%. Εντούτοις δεν υπάρχει καμιά εγγύηση στον αριθμό των αργών λειτουργιών ανάγνωσης και είναι δυνατό να διαφέρουν ανάλογα με την κατάσταση του δικτύου και να δώσουν καλύτερα ή χειρότερα αποτελέσματα. Παρόλα αυτά τα αποτελέσματα αυτά είναι πολύ πιο καλύτερα από τις προσομοιώσεις του αλγορίθμου που έδωσαν πάνω από πάνω από 85% στη χειρότερη περίπτωση και στις υπόλοιπες περιπτώσεις δεν ξεπερνά το 12%. Η μεγάλη διαφορά των αποτελεσμάτων, για τη χειρότερη περίπτωση, οφείλεται στην ασυγχρονία του δικτύου Planet Lab όπου οι λειτουργιές δεν είναι εφικτό να εκτελεστούν ακριβώς ταυτόχρονα όπως στις προσομοιώσεις. Άρα βασικά, οι προσομοιώσεις μας δίνουν τα χειρότερα αποτελέσματα που μπορεί να έχει ο αλγόριθμος SLIQ και τα οποία έχουν μηδαμινές πιθανότητες να συμβούν σε ρεαλιστικό δίκτυο. Συνεπώς ο αλγόριθμος SLIQ σε πραγματικό δίκτυο είναι πολύ αποδοτικός λόγω του πολύ μικρού αριθμού αργών αναγνώσεων που είδαμε

στο προηγούμενο κεφάλαιο. Τέλος, βάση αυτής της εργασίας συμπεράνουμε ότι ο αλγόριθμος SLIQ σε πρακτικό περιβάλλον συμπεριφέρεται ως γρήγορος παρά ως ασθενές-ημιγρήγορος.

6.2 Μελλοντική Εργασία

Η υλοποίηση που έγινε για τον αλγόριθμο δεν είναι η βέλτιστη δυνατή και υπάρχουν πολλά περιθώρια για βελτίωση καθώς και επέκτασης της. Για να γίνουμε πιο σαφής η υλοποίηση του εξυπηρετητή έχει υλοποιηθεί έτσι ώστε κάθε νέα αίτηση να την αναλαμβάνει μια νέα διεργασία. Μπορεί να υλοποιηθεί έτσι ώστε να την αναλαμβάνει ένα νέο νήμα (thread). Η εναλλαγή των νημάτων είναι πέντε φορές πιο γρήγορη από την εναλλαγή των διεργασιών και κοστίζει λιγότερο η δημιουργία ενός νήματος από τη δημιουργία μιας νέας διεργασίας. Εφόσον ο μέγιστος αριθμός των αναγνωστών είναι γνωστός μπορεί να γίνει χρήση thread pools που κοστίζουν ακόμη λιγότερο, επειδή δε γίνεται δημιουργία και καταστροφή των νημάτων κάθε φορά. Με τη χρήση νημάτων δε θα είναι απαραίτητη η επικοινωνία των διεργασιών μέσω κοινών αρχείων, η οποία δεν είναι αποδοτική. Παρόλα αυτά στην υλοποίηση με νήματα είναι απαραίτητος κάποιος μηχανισμός για την προστασία των κοινών πόρων. Επιπρόσθετα η υλοποίηση δεν είναι ανθρωποκεντρική, δηλαδή η διαπροσωπεία της εφαρμογής δεν είναι φιλική ως προς την αλληλεπίδραση με το χρήστη και υπάρχουν πολλά περιθώρια βελτίωσης σε αυτό τον τομέα.

Τέλος η αξιολόγηση του αλγορίθμου έγινε εξετάζοντας την απόδοση του σε τρία διαφορετικά συστήματα απαρτίας. Για αυτά τα συστήματα απαρτίας εξετάσαμε την απόδοση του αλγορίθμου μεταβάλλοντας τον αριθμό των αναγνωστών και τον αριθμό των αντιγράφων του αντικειμένου στο σύστημα. Θα μπορούσε να γίνει αξιολόγηση της συμπεριφοράς του αλγορίθμου κάτω από διαφορετικές συνθήκες μεταβάλλοντας άλλες παραμέτρους που δεν εξετάστηκαν σε αυτή την εργασία για να εγγυηθούν λιγότερες «αργές» λειτουργίες ανάγνωσης.

Βιβλιογραφία

[1] H. Attiya, A. Bar-Noy και D.Dolev. “Sharing memory robusty in message passing systems”. Journal of the ACM, 42(1):124-142,1996

[2] P. Dutta , R. Guerraoui, R. R. Levy, A. Chakraborty, “How fast can a distributed atomic read be?”, In Proceeding of the 23rd ACM symposium of Principles of Distributed Computing (PODC) , pages 236-24, 2004

[3] C. Georgiou, P. M. Musial, and A. A. Shvartsman. Developing a consistent domain-oriented distributed object service.
In Proceedings of the 4th IEEE International Symposium on Network Computing and Applications (NCA), pages 149–
158, 2005.

[4] C. Georgiou, P. M. Musial, and A. A. Shvartsman. Long-lived RAMBO: Trading knowledge for communication. Theoretical Computer Science, 383(1):59–85, 2007.

[5] Chryssis Georgiou, Nicolas C. Nicolaou, Alexander A. Shvasrtsman, “Fault-Tolerant SemiFast Implementations of Atomic Read/Write Registers”. In Proceedings of the 18th annual ACM symposium on Parallelism in Algorithms and Architectures (SPAA 2006), pages 281-290, 2006.

[6] Chryssis Georgiou, Nicolas C. Nicolaou, Alex A. Shvartsman “On the Robustness of (Semi)Fast Quorum-Based Implementations of Atomic Shared Memory”. In Proceedings of the 22nd International Symposium on Distributed Computing (DISC 2008), pages 289-304, Arcachon, France, 2008.

[7] S. Gilbert, N. Lynch, and A. Shvartsman. RAMBO II: Rapidly reconfigurable atomic memory for dynamic networks. In Proceedings of International Conference on Dependable Systems and Networks (DSN), pages 259–268, 2003.

[8] Maurice Herlihy and Jeannette Wing. Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Language and System, July 1992.

[9] Ajay D. Kshemkalyani, Mukesh Singhal, "Distributed Computing Principles, Algorithms, and Systems", First Edition, Cambridge University Press, 2008

[10] Don Libes, "Exploring Expect", O'Reilly & Associates, 1994.

[11] N. Lynch, "Distributed Algorithms", Morgan Kaufmann Publishers, 1996.

[12] N. Lynch and A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In Proceedings of 16th International Symposium on Distributed Computing (DISC), pages 173–190, 2002.

[13] D.Peleg και A.Wool, "Crumbling walls: A class of high availability quorum system". In Proceedings of 14th ACM Symposium on Principles of Distributed Computing (PODC), pages 120-129, 1995.

[14] L.Peterson, T.Anderson, DCuller, T. Roscoe, "A Blueprint for Introducing Disruptive Technology into the Internet", pages 59-64, ACM SIGCOMM, Volume 33, Issue 1, 2003.

[15] Planet Lab Site, http://www.planet-lab.org

[16] RFC 793, http://www.rfc-editor.org/rfc/rfc793.txt.

[17] W.Richard Stevens, "TCP/IP Illustrated, Volume I", Addison Wesley Longman, 1994.

Παράρτημα Α

```
/*********************************************************************/
/*                                                                   */
/*Onoma: Antigoni Hadjidemetriou                                     */
/*Date: 01/06/2009                                                   */
/*Filename: server.c                                                 */
/*                                                                   */
/*********************************************************************/

/*********************************************************************/
/*                              LIBRARIES                            */
/*********************************************************************/
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/time.h>
#include <stdlib.h>
#include <memory.h>
#include <string.h>
#include <sys/wait.h>


/*********************************************************************/
/*                        CONSTANT VARIABLES                         */
/*********************************************************************/
#define MAX_PENDING 6     //Maximum length of pending connections in
                            the queue
#define MAX_READERS 100   //Maximum number of readers in the system


/*********************************************************************/
/*                      STRUCTURE DEFINITIONS                        */
/*********************************************************************/
//Structure of the messages exchange in the system
struct msg
{
  char* type;
  int ts;
  int value;
  int pvalue;
  int counter;
  int process_id;
};


/*********************************************************************/
/*                      FUNCTION PPROTOTYPES                         */
/*********************************************************************/
void sigchld_handler(int s);
```

```c
/*****************************************************************/
/*                              MAIN                             */
/*****************************************************************/
int main(int argc,char *argv[])
{
  int socketFd;                 //Socket file descriptor
  int newSocket;                //Socket file descriptor of a new
                                  connection
  int addrLen;                  //Address size
  int yes = 1;                  //For setsockopt()
  int msgLen;                   //Message size
  int id;                       //Server Id
  int port;                     //Port number that server waits for new
                                  connections
  int timestamp = 0;            //Timestamp of the atomic object
                                  replica
  int value = -1;               //Value of the atomic object replica
  int pvalue = -1;              //Previous value of the atomic object
                                  replica
  int counter[MAX_READERS+1];   //Table with processes last request ids
  int i;                        //Counter
  int retValue;                 //Functions return value
  char buf[256];                //Message contents
  char command[30];             //Use to remove temporary files
  char tempFile[30];            //Temporary filename
  FILE* fd;                     //Use in read and write of temporary
                                  files
  struct msg reply;             //Received message
  struct sockaddr_in server;    //Server information
  struct sockaddr_in client;    //New connection information
  struct sigaction sa;          //Use to install signal handler
  struct hostent *rem;          //Use to learn the name of the remote
                                  host

  srand(time(NULL));
  reply.type = (char*) malloc(10* sizeof(char));


  //Check of input arguments
  if(argc!=3)
    {
      printf("\nUsage: %s [server ID] [port number]\n", argv[0]);
      exit(-1);
    }

  id = atoi(argv[1]);
  port = atoi(argv[2]);


 //Installs signal handler of signal SIGCHLD
  sa.sa_handler = sigchld_handler;
  sigemptyset(&sa.sa_mask);
  sa.sa_flags = SA_RESTART;

  if (sigaction(SIGCHLD, &sa, NULL) == -1)
    {
      perror("sigaction()");
      exit(1);
    }
```

```
//Creates the socket
socketFd = socket(PF_INET, SOCK_STREAM, 0);
if(socketFd < 0)
  {
    perror("\nsocket()");
    exit(-1);
  }


//Allow to reuse the port
if (setsockopt(socketFd, SOL_SOCKET, SO_REUSEADDR, &yes,
    sizeof(int)) == -1)
  {
    perror("\nsetsockopt\n");
    close(socketFd);
    exit(-1);
  }


//Bind Socket to a Port
server.sin_family = AF_INET;
server.sin_port = htons(port);
server.sin_addr.s_addr = htonl(INADDR_ANY);
memset( &(server.sin_zero), '\0' , sizeof(server.sin_zero));

if(bind(socketFd, (struct sockaddr *) &server, sizeof(server)) < 0)
  {
    perror("\nbind()\n");
    close(socketFd);
    exit(-1);
  }


//Set Socket to Listen
if(listen(socketFd, MAX_PENDING) < 0)
  {
    perror("\nlisten()");
    close(socketFd);
    exit(-1);
  }


while(1)
  {
    //Accepts New Connection
    addrLen = sizeof(client);
    newSocket = accept(socketFd, (struct sockaddr *) &client,
                       &addrLen);
    if(newSocket < 0)
    {
      perror("\naccept()\n");
      close(socketFd);
      exit(-1);
    }

    if((rem = gethostbyaddr((char*) &client.sin_addr.s_addr,
       sizeof(client.sin_addr.s_addr),client.sin_family))==NULL)
    {
      perror("gethostbyaddr");
```

```
  close(newSocket);
  close(socketFd);
  exit(-1);
}
printf("Accepted connection from %s\n\n",rem->h_name);

switch(fork())
{
case -1:
  perror("\nfork() failed");
  close(newSocket);
  close(socketFd);
  exit(-1);
  break;

//Child process
case 0:
  while(1)
    {
      //Read Message
      bzero(buf, sizeof (buf));
      if(recv(newSocket, buf, sizeof(buf), 0) < 0)
      {
        perror("recv()");
        close(newSocket);
        close(socketFd);
        exit(-1);
      }

      printf("\nMessage \"%s\" received from %s", buf,
              rem->h_name);


      //Algorithm termination
      if(!(strcmp(buf,"FIN")))
      {
        //Remove temprorary files
        bzero(command,sizeof(command));
        sprintf(command,"rm server%dcounter.txt",port);
        system(command);

        bzero(command,sizeof(command));
        sprintf(command,"rm server%d.txt",port);
        system(command);

        kill(0, SIGKILL);
      }

      //Reader exits
      else if(!strcmp(buf,"Reader Exiting..."))
      {
        printf("\n\nReader %s is exiting...\n\n", rem->h_name);
        break;
      }
      else
      {
        //Retrieve data from temporary files
        sprintf(tempFile, "server%dcounter.txt", port);
        if(!(fd = fopen(tempFile, "r")))
          {
```

```
    fd = fopen(tempFile, "w");
    for(i=0; i<=MAX_READERS; i++)
    {
      counter[i] = 0;
      fprintf(fd, "0\n");
    }
  }
else
  {
    for(i=0; !feof(fd); i++)
    fscanf(fd, "%d\n", &counter[i]);
  }
fclose(fd);


sprintf(tempFile, "server%d.txt", port);
if(!(fd = fopen(tempFile, "r")))
  {
    fd = fopen(tempFile, "w");
    fprintf(fd, "0\t0\t0\n");
    timestamp = 0;
    value = -1;
    pvalue = -1;
  }
else
  {
    fscanf(fd, "%d\t%d\t%d\n", &timestamp, &value,
           &pvalue);
    fclose(fd);
  }


//Convert message to its tokens
reply.type = strdup(strtok(buf, ","));
reply.ts = atoi(strtok(NULL, ","));
reply.value = atoi(strtok(NULL, ","));
reply.pvalue = atoi(strtok(NULL, ","));
reply.counter = atoi(strtok(NULL, ","));
reply.process_id = atoi(strtok(NULL, "#"));


if( reply.counter > counter[reply.process_id])
  //A new request from the process
  {
    counter[reply.process_id] =  reply.counter;
    if(reply.ts > timestamp)
    {
      value =  reply.value;
      pvalue = reply.pvalue;
      timestamp = reply.ts;
    }
  }


//Reply to the client
bzero(buf, sizeof (buf));
if(strcmp(reply.type, "WRITE")==0)
  sprintf(buf, "WRITEACK,%d,%d,%d,%d,%d#", timestamp,
          value, pvalue, counter[reply.process_id],id);
else if(strcmp(reply.type, "READ")==0)
```

```
                sprintf(buf, "READACK,%d,%d,%d,%d,%d#", timestamp,
                        value, pvalue, counter[reply.process_id],id);
              else if(!(strcmp(reply.type, "INFORM")))
                sprintf(buf, "INFORMACK,%d,%d,%d,%d,%d#", timestamp,
                        value, pvalue, counter[reply.process_id],id);

               msgLen = strlen(buf);
               retValue = send(newSocket, buf, msgLen, 0);
               if(retValue != msgLen)
                 {
                  printf("\nsend() sent a different number of bytes
                          than expected\n");
                  exit(-1);
                 }
               else if(retValue < 0)
                  {
                  perror("\nsend()");
                  exit(-1);
                  }
               else
                 printf("\nAcknowledgment \"%s\" sended to %s\n", buf,
                        rem->h_name);


               //Save data to temporary files
               sprintf(tempFile, "server%dcounter.txt", port);
               if(!(fd = fopen(tempFile,"w")))
                 {
                  printf("Unable to open file %s\nExiting...\n\n",
                          tempFile);
                  exit(-1);
                 }

               for(i=0; i<=MAX_READERS; i++)
                 fprintf(fd, "%d\n", counter[i]);

               fclose(fd);


               sprintf(tempFile, "server%d.txt", port);
               if(!(fd = fopen(tempFile, "w")))
                 {
                  printf("Unable to open file %s\nExiting...\n\n",
                          tempFile);
                  exit(-1);
                 }
               fprintf(fd,"%d\t%d\t%d\n",timestamp, value, pvalue);
               fclose(fd);

            }//else
          }//while(1)
        break;
      }//switch(fork())
    }//while(1)

  return 0;
}
```

```c
/*****************************************************************/
/*                Function that reaps all dead processes         */
/*****************************************************************/
void sigchld_handler(int s)
{
  while(waitpid(-1, NULL, WNOHANG) > 0);
}
```

```
/*************************************************************************/
/*                                                                       */
/*Onoma: Antigoni Hadjidemetriou                                         */
/*Date: 01/06/2009                                                       */
/*Filename: writer.c                                                     */
/*                                                                       */
/*************************************************************************/

/*************************************************************************/
/*                              LIBRARIES                                */
/*************************************************************************/
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/time.h>
#include <stdlib.h>
#include <memory.h>
#include <string.h>
#include <sys/wait.h>


/*************************************************************************/
/*                         STRUCTURE DEFINITIONS                         */
/*************************************************************************/
//List with servers
struct node
{
  int serverId;           //Server Id
  struct node* next;      //Next Server
};

//Quorum Structure
struct quorum
{
  int quorumId;              //Quorum ID
  struct node* servers ;     //Servers in the quorum
  struct quorum* next;       //Next quorum of quorum system
};

//Structure of the messages exchange in the system
struct msg
{
  char* type;
  int ts;
  int value;
  int pvalue;
  int counter;
  int process_id;
};
```

```
/*****************************************************************/
/*                          GLOBAL VARIABLES                     */
/*****************************************************************/
int wCounter = 0;                    //Write operations' counter
int* servAck;                        //Set of servers respond with ack
int* socket_fd;                      //Socket Descriptors
fd_set readfds;                      //Set of socket descriptors use in
                                       select
fd_set crashfds;                     //Crashed servers
struct quorum* quorumSystem = NULL; //System's Quorum System
struct sockaddr_in *server;          //Servers' information


/*****************************************************************/
/*                       FUNCTION PROTOTYPES                     */
/*****************************************************************/
void create_quorum_system(FILE* fin);
void send_msg(char* buf, int numServers);
void receive_complete_quorum(int numServers);
void clearQuorums();
void serverAck(int id);
int isQuorumCompl();


/*****************************************************************/
/*                              MAIN                             */
/*****************************************************************/
int main(int argc,char *argv[])
{
  int timestamp = 0;            //Atomic object's timestamp
  int value = -999;             //Atomic object's value
  int pvalue = 0;               //Atomic object's previous value
  int numServers;               //Number of servers in the quorum
                                  system
  int port;                     //Use to read server's port from file
  int i;                        //Counter
  int choice;                   //User's choice in menu
  char buf[300];                //Message contents
  char servAddres[100];         //Use to read server's  name from file
  FILE* fin;                    //File descriptor of the file
                                  containing servers' information
  struct hostent *rem;          //Use to find server's address


  srand(time(NULL));

  //Check of input arguments
  if(argc!=3)
    {
      printf("\nUsage: argv[0] [servers filename] [quorum system
              filename]\n");
      exit(-1);
    }

  //Open servers file
  if(!(fin = fopen(argv[1],"r")))
    {
      perror("\nfopen()");
      exit(-1);
    }
```

```
//Read number of servers in the first line
fscanf(fin, "%d", &numServers);


//Create tables
servAck = (int*) malloc(numServers * sizeof(int));
socket_fd = (int*) malloc(numServers * sizeof(int));
server = (struct sockaddr_in*) malloc(numServers * sizeof(struct
          sockaddr_in));

//Initialization of file descriptor set
FD_ZERO(&readfds);
FD_ZERO(&crashfds);

//Read in each line server name and server port
for(fscanf(fin,"%s%d", servAddres,&port), i=0; !feof(fin),
    i<numServers; fscanf(fin,"%s%d", servAddres,&port), i++)
  {
    //Creates ith socket
     socket_fd[i] = socket(PF_INET,SOCK_STREAM, 0);
     if(socket_fd[i] < 0){ perror("\nsocket() failed"); exit(-1); }
     if((rem=gethostbyname(servAddres))==NULL)
     {herror("\ngethostbyname() failed"); exit(-1);}

     server[i].sin_family = AF_INET;
     server[i].sin_port = htons(port);
     bcopy((char*)rem->h_addr,(char*) &server[i].sin_addr,
           rem->h_length);
     memset(&(server[i].sin_zero), '\0',sizeof(server[i].sin_zero));

     //Connects with Server
     if((connect(socket_fd[i], (struct sockaddr *) &server[i],
         sizeof(server[i]))<0))
     {
       perror("\nconnect()");
       exit(-1);
     }

     printf("\nConnection with %s in port %d is established!!!\n",
             servAddres, port);

     FD_SET(socket_fd[i], &readfds);
  }
fclose(fin);


//Opens quorum system file
if(!(fin = fopen(argv[2],"r")))
  {
    perror("\nfopen()");
    exit(-1);
  }

//Creates quorum system structure
create_quorum_system(fin);
fclose(fin);
```

```
  while(1)
    {

      printf("\n\n********************MENOU************************");
      printf("\n1. Send write message to all servers.");
      printf("\n2. Exit.");
      printf("\n\nChoice--->");
      scanf("%d", &choice);

      if(choice == 1)
      {
        //Sends the atomic object
        bzero(buf, sizeof(buf));
        pvalue = value;
        value = rand() % 1000;
        timestamp++;
        wCounter++;

        sprintf(buf, "WRITE,%d,%d,%d,%d,0#", timestamp, value, pvalue,
                wCounter);
        send_msg(buf, numServers);

        clearQuorums(numServers);
        receive_complete_quorum(numServers);
      }
      else if(choice == 2)
      {
        //Sends algorithm termination message
        bzero(buf, sizeof(buf));
        strcpy(buf, "FIN");
        send_msg(buf, numServers);
        break;
      }
      else
      printf("\n\nWrong Choice!!!\n");
    }

  return 0;
}


/*****************************************************************/
/* Function that reads quorums system's information from file and   */
/*               creates the quorums system structure              */
/*****************************************************************/
void  create_quorum_system(FILE* fin)
{
  int id;
  char line[1024];
  char c;
  char *dummy = NULL;
  struct quorum* temp;
  struct node* tmp;


  for(fscanf(fin,"%c", &c); !feof(fin); fscanf(fin,"%c", &c))
    {

      if(c=='\n' || c=='\t' || c==' ')
      continue;
```

```
        if(c!='Q' && c!='q')
        {
          printf("\nError in reading the file with the quorum system
                    information!\n");
          printf("Expecting Q/q\n");
          exit(-1);
        }

        temp = (struct quorum*) malloc(sizeof(struct quorum));
        temp->servers = NULL;
        temp->next = quorumSystem;

        quorumSystem = temp;

        fscanf(fin, "%d", &id);
        temp->quorumId = id;

        //Reads the character (
        fscanf(fin, "%c", &c);
        fscanf(fin, "%s", line);


        for(dummy = strtok(line, ","); dummy[strlen(dummy)-1]!=')';
            dummy = strtok(NULL, ","))
        {
          tmp = (struct node*) malloc(sizeof(struct node));
          tmp->serverId = atoi(dummy);
          tmp->next = temp->servers;
          temp->servers = tmp;
        }


        dummy = strtok(dummy, ")");
        tmp = (struct node*) malloc(sizeof(struct node));
        tmp->serverId = atoi(dummy);
        tmp->next = temp->servers;
        temp->servers = tmp;

        //Reads the intersections
        for(fscanf(fin, "%c%c", &c, &c); (c=='I' || c=='i') &&
            !feof(fin); fscanf(fin, "%c%c", &c, &c))
        fscanf(fin, "%s", line);

        if(feof(fin))
        break;
        else
        fseek(fin, -1, SEEK_CUR);
    }
}


/*****************************************************************/
/*               Sends the message "buf" to all servers          */
/*****************************************************************/
void send_msg(char* buf, int numServers)
{
  int msg_len, i, retValue;

  msg_len = strlen(buf);
```

```
  for(i=0; i<numServers; i++)
    {
      if(!FD_ISSET(socket_fd[i], &crashfds))
        {
            retValue = send(socket_fd[i], buf, msg_len, 0);
            if( retValue != msg_len)
            {
                  printf("\nsend() sent a different number of bytes
                          than expected");
                  exit(-1);
            }
            else if(retValue < 0)
            {
                  perror("send()");
                  exit(-1);
            }
            else
            printf("\nMessage '%s' sent to %s", buf,
                    *gethostbyaddr((char*) &server[i].sin_addr,
                     sizeof(server[i].sin_addr),AF_INET));
            }
      }
}


/*********************************************************************/
/*        Receive messages until a whole quorum is completed         */
/*********************************************************************/
void receive_complete_quorum(int numServers)
{
  int i;
  char buf[300];
  struct msg reply;
  struct timeval tv;

  printf("\n");
  reply.type = (char*) malloc(8 * sizeof(char));
  do
    {

      //Wait 0 seconds
      tv.tv_sec = 0;
      tv.tv_usec = 0;


      if(select(FD_SETSIZE, &readfds, NULL, NULL, &tv) < 0)
      {
        perror("select()");
        exit(-1);
      }

      for(i=0; i<numServers; i++)
      {
        if(FD_ISSET(socket_fd[i], &readfds))
          {
            //Receiving message
            bzero(buf,sizeof (buf));
            if(recv(socket_fd[i], &buf, sizeof(buf),0) < 0)
            {
```

```
            perror("\nrecv() failed");
            exit(-1);
          }

          if(!strlen(buf))
          {
            FD_CLR(socket_fd[i], &readfds);
            FD_SET(socket_fd[i], &crashfds);
            continue;
          }

          printf("\nAcknowlegment %s received from %s", buf,
                   *gethostbyaddr((char*) &server[i].sin_addr,
                   sizeof(server[i].sin_addr),AF_INET));

          reply.type = strdup(strtok(buf, ","));
          reply.ts = atoi(strtok(NULL, ","));
          reply.value =  atoi(strtok(NULL, ","));
          reply.pvalue =  atoi(strtok(NULL, ","));
          reply.counter = atoi(strtok(NULL, ","));
          reply.process_id = atoi(strtok(NULL, "#"));

          if( reply.counter == wCounter )
                serverAck(reply.process_id);
        }

      //Add again to the set of quorums
      else if(!FD_ISSET(socket_fd[i], &crashfds))
        FD_SET(socket_fd[i], &readfds);
    }

  }while(!isQuorumCompl());

  free(reply.type);

}


/*********************************************************************/
/*         Clears the servers who sent a WRITEACK message            */
/*********************************************************************/
void clearQuorums(int numServers)
{
  int i;

  for(i=0; i<numServers; i++)
    servAck[i] = 0;

}


/*********************************************************************/
/*   Add the server in the set of servers who reply with WRITEACK    */
/*********************************************************************/
void serverAck(int id)
{
  servAck[id-1] = 1;
}
```

```
/*********************************************************************/
/*    Checks if a whole quorum responded with WRITEACK messages      */
/*********************************************************************/
int isQuorumCompl()
{
  struct quorum* temp;
  struct node* tmp;

  temp = quorumSystem;

  for(temp = quorumSystem; temp!=NULL; temp = temp->next)
    {
      for(tmp = temp->servers; tmp!=NULL; tmp = tmp->next)
      if(servAck[(tmp->serverId)-1] == 0)
        break;

      if(tmp == NULL)
      return 1;
    }

  return 0;
}
```

```
/*************************************************************************/
/*                                                                       */
/*Onoma: Antigoni Hadjidimitriou                                         */
/*Date: 01/06/2009                                                       */
/*Filename: reader.c                                                     */
/*                                                                       */
/*************************************************************************/

/*************************************************************************/
/*                              LIBRARIES                                */
/*************************************************************************/
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/time.h>
#include <stdlib.h>
#include <memory.h>
#include <string.h>
#include <sys/wait.h>


/*************************************************************************/
/*                         STRUCTURE DEFINITIONS                         */
/*************************************************************************/
//List with servers
struct node
{
  int serverId;           //Server Id
  struct node* next;      //Next Server
};

//List with intersected nodes
struct intersectionNode
{
  int intQuorum;                  //Intersected quorum id
  struct intersectionNode* next;  //Next intersection
  struct node * servers;          //Servers of the quorum who belongs
                                    in this intersection
};

//Quorum Structure
struct quorum
{
  int quorumId;                           //Quorum ID
  struct node* servers ;                  //Servers in the quorum
  struct intersectionNode* intersection;  //Quorum's intersections
                                            with other quorums
  struct quorum* next;                    //Next quorum of quorum
                                            system
};
```

```
struct serverNode
{
  int ack;       //Defines if an ack is received
  int ts;        //The timestamp in the ack message
};


//Structure of the messages exchanged in the system
struct msg
{
  char* type;
  int ts;
  int value;
  int pvalue;
  int counter;
  int process_id;
};



/*****************************************************************/
/*                          GLOBAL VARIABLES                     */
/*****************************************************************/
int rCounter = 0;                   //Read operation's counter
int timestamp = 0;                  //Atomic object's timestamp
int value = -1;                     //Atomic object's value
int pvalue = -1;                    //Atomic object's previous value
int* socket_fd;                     //Socket descriptors
struct quorum* quorumSystem = NULL; //System's quorum system
struct serverNode* servers;         //Set of servers respond with ack
struct sockaddr_in *server;         //Servers' information
fd_set readfds;                     //Set of socket descriptors used
                                      in select for read
fd_set crashfds;                    //Set of crashed servers


/*****************************************************************/
/*                       FUNCTION PROTOTYPES                     */
/*****************************************************************/
void send_msg(char* buf, int numServers);
struct quorum* receive_complete_quorum(int numServers, int* maxTs,
int* object, int* pobject);
void create_quorum_system(FILE* fin);
void clearQuorums();
void serverAck(int id, int ts);
int isQuorumCompl(struct quorum** q);
int find_quorum_view(struct quorum* q, int maxTs);



/*****************************************************************/
/*                              MAIN                             */
/*****************************************************************/
int main(int argc,char *argv[])
{
  int id;                           //Reader Id
  int port;                         //Use to read server's port from
                                      file
  int numServers;                   //Number of servers in the quorum
                                      system
```

```
int i;                              //Counter
int choice;                         //User's choice in menu
int maxTs = -1;                     //Maximum timestamp the reader has
                                      witnessed
int retValue;                       //The value of the atomic object
                                      after a read is completed
int retPvalue;                      //The previous value of the atomic
                                      object after a read is completed
char buf[256];                      //Message contents
char servAddres[100];               //Use to read server's name from
                                      file
FILE* fin;                          //File descriptor of the file
                                      containing servers information
struct hostent *rem;                //Use to find server's address
struct quorum* completeQ;           //The quorum which respond to
                                      reader


srand(time(NULL));

//Check of input arguments
if(argc!=4)
  {
    printf("\nUsage: argv[0] [reader ID] [servers filename] [quorum
            system filename]\n");
    exit(-1);
  }

id = atoi(argv[1]);

//Opens servers file
if(!(fin = fopen(argv[2],"r")))
  {
    perror("\nfopen()");
    exit(-1);
  }

//Read number of servers in the first line
fscanf(fin, "%d", &numServers);

//Create tables
servers = (struct serverNode*) malloc(numServers *
           sizeof(struct serverNode));
socket_fd = (int*) malloc(numServers * sizeof(int));
server = (struct sockaddr_in*) malloc(numServers *
          sizeof(struct sockaddr_in));

//Initialization of file descriptor set
FD_ZERO(&readfds);
FD_ZERO(&crashfds);

 //Read in each line server name and server port
for(fscanf(fin,"%s%d", servAddres,&port), i=0; !feof(fin),
    i<numServers; fscanf(fin,"%s%d", servAddres,&port), i++)
  {
    //Creates ith socket
    socket_fd[i] = socket(PF_INET,SOCK_STREAM, 0);
    if(socket_fd[i] < 0){perror("\nsocket() failed");  exit(-1);}
    if((rem=gethostbyname(servAddres))==NULL)
    {herror("\ngethostbyname() failed"); exit(-1);}
```

```
    server[i].sin_family = AF_INET;
    server[i].sin_port = htons(port);
    bcopy((char*)rem->h_addr,(char*) &server[i].sin_addr,
          rem->h_length);
    memset(&(server[i].sin_zero), '\0', sizeof(server[i].sin_zero));

    //Connects with Server
    if((connect(socket_fd[i], (struct sockaddr *) &server[i],
       sizeof(server[i]))<0))
    {
      perror("\n\nconnect() failed");
      exit(-1);
    }
    printf("\nConnection with %s in port %d established!!!\n",
           servAddres, port);

    FD_SET(socket_fd[i], &readfds);
  }
  fclose(fin);


//Opens quorum system file
if(!(fin = fopen(argv[3],"r")))
  {
    perror("\nfopen()");
    exit(-1);
  }

//Creates quorum system structure
create_quorum_system(fin);
fclose(fin);

while(1)
  {

    printf("\n\n********************MENOU************************");
    printf("\n1. Send read message to all servers.");
    printf("\n2. Exit.");
    printf("\n\nChoice--->");
    scanf("%d", &choice);


    if(choice == 1)
      {
      //Sends the atomic object
      bzero(buf, sizeof(buf));
      rCounter++;
    sprintf(buf, "READ,%d,%d,%d,%d,%d#", timestamp, value, pvalue,
            rCounter, id);
      send_msg(buf, numServers);

      clearQuorums(numServers);
      completeQ = receive_complete_quorum(numServers, &maxTs,
                  &retValue, &retPvalue);
      qv = find_quorum_view(completeQ, maxTs);

      if(qv == 1)
     //Quorum View 1
          {
```

```
            timestamp = maxTs;
            value = retValue;
            pvalue = retPvalue;
          }
        else if (qv==2)
        //Quorum View 2
          {
            timestamp = maxTs - 1;
            value = retPvalue;
            pvalue = retPvalue;
          }
         else
         //Quorum View 3
          {
            timestamp = maxTs;
            value = retValue;
            pvalue = retPvalue;

            //Sends the atomic object
            bzero(buf, sizeof(buf));
            rCounter++;
            sprintf(buf, "INFORM,%d,%d,%d,%d,%d#", timestamp, value,
                    pvalue, rCounter, id);
            send_msg(buf, numServers);

            clearQuorums(numServers);
            completeQ = receive_complete_quorum(numServers, &maxTs,
                        &retValue, &retPvalue);
          }
      }
      else if(choice == 2)
      {
        //Sends the atomic object
        bzero(buf, sizeof(buf));
        sprintf(buf, "Reader Exiting...");
        send_msg(buf, numServers);
        break;
      }
      else
            printf("\n\nWrong Choice!!!\n");
    }

  return 0;
}


/*********************************************************************/
/*  Function that reads quorums system's information from file and  */
/*            creates the quorum's system structure                  */
/*********************************************************************/

void  create_quorum_system(FILE* fin)
{
  int id;
  char line[1024];
  char c;
  char *dummy = NULL;
  struct quorum* temp;
  struct node* tmp;
  struct intersectionNode* tmp2;
```

```
for(fscanf(fin,"%c", &c); !feof(fin); fscanf(fin,"%c", &c))
  {

    if(c=='\n' || c=='\t' || c==' ')
      continue;

    if(c!='Q' && c!='q')
    {
      printf("\nError in reading the file with the quorum system
              information!\n");
      printf("Expecting Q\n");
      exit(-1);
    }


     temp = (struct quorum*) malloc(sizeof(struct quorum));
     temp->servers = NULL;
     temp->intersection = NULL;
     temp->next = quorumSystem;

     quorumSystem = temp;

     fscanf(fin, "%d", &id);
     temp->quorumId = id;

     //Reads the character(
     fscanf(fin, "%c", &c);
     fscanf(fin, "%s", line);


     for(dummy = strtok(line, ","); dummy[strlen(dummy)-1]!=')';
         dummy = strtok(NULL, ","))
     {
       tmp = (struct node*) malloc(sizeof(struct node));
       tmp->serverId = atoi(dummy);
       tmp->next = temp->servers;
       temp->servers = tmp;
     }

     dummy = strtok(dummy, ")");
     tmp = (struct node*) malloc(sizeof(struct node));
     tmp->serverId = atoi(dummy);
     tmp->next = temp->servers;
     temp->servers = tmp;

     //Reads the intersections with the quorum
     for( fscanf(fin, "%c%c", &c,&c); c=='I'; fscanf(fin, "%c%c",
          &c,&c))
     {
       fscanf(fin, "%c%c%d%c", &c,&c,&id,&c);

       tmp2 = (struct intersectionNode*) malloc(sizeof(struct
               intersectionNode));
       tmp2->intQuorum = id;
       tmp2->next = temp->intersection;
       tmp2->servers = NULL;
       temp->intersection = tmp2;
```

```
            fscanf(fin, "%s", line);

            for(dummy = strtok(line, ","); dummy[strlen(dummy)-1]!=')';
                dummy = strtok(NULL, ","))
              {

                tmp = (struct node*) malloc(sizeof(struct node));
                tmp->serverId = atoi(dummy);
                tmp->next = tmp2->servers;
                tmp2->servers = tmp;
              }

            dummy = strtok(dummy, ")");
            tmp = (struct node*) malloc(sizeof(struct node));
            tmp->serverId = atoi(dummy);
            tmp->next = tmp2->servers;
            tmp2->servers = tmp;

         }

        if(feof(fin))
          break;
        else
          fseek(fin, -1, SEEK_CUR);
      }
}


/*********************************************************************/
/*            Sends the message "buf" to all servers                 */
/*********************************************************************/
void send_msg(char* buf, int numServers)
{
  int msg_len, i, retValue;

  msg_len = strlen(buf);

  for(i=0; i<numServers; i++)
    {
      if(!FD_ISSET(socket_fd[i], &crashfds))
        {
            retValue = send(socket_fd[i], buf, msg_len, 0);
            if(retValue != msg_len)
            {
                  printf("\nsend() sent a different number of bytes
                          than expected");
                  exit(-1);
            }
            else if (retValue<0)
            {
                  perror("\nsend()");
            }
            else
                  printf("\nMessage '%s' sent to %s", buf,
                          *gethostbyaddr((char*) &server[i].sin_addr,
                           sizeof(server[i].sin_addr),AF_INET));
            }
      }
}
```

```
/*********************************************************************/
/*     Receive messages until a whole quorum is completed            */
/*********************************************************************/
struct quorum* receive_complete_quorum(int numServers, int* maxTs,
int* object, int* pobject)
{
  int i;
  char buf[1024];
  struct timeval tv;
  struct msg reply;
  struct quorum* compQ;

  reply.type = (char*) malloc(10* sizeof(char));

  printf("\n");
  do
    {
      //Wait 0 seconds
      tv.tv_sec = 0;
      tv.tv_usec = 0;

      select(FD_SETSIZE, &readfds, NULL, NULL, &tv);

      for(i=0; i<numServers; i++)
      {
        if(FD_ISSET(socket_fd[i], &readfds))
          {
            //Receiving message
            bzero(buf, sizeof (buf));
            if(recv(socket_fd[i], &buf, sizeof(buf), 0) < 0)
            {
              perror("\nrecv() failed");
              exit(-1);
            }

            if(!strlen(buf))
            {
              FD_CLR(socket_fd[i], &readfds);
              FD_SET(socket_fd[i], &crashfds);
              continue;
            }

            printf("\nAcknowlegment %s received from %s", buf,
                     *gethostbyaddr((char*) &server[i].sin_addr,
                     sizeof(server[i].sin_addr),AF_INET));
            reply.type = strdup(strtok(buf, ","));
            reply.ts = atoi(strtok(NULL, ","));
            reply.value =  atoi(strtok(NULL, ","));
            reply.pvalue =  atoi(strtok(NULL, ","));
            reply.counter = atoi(strtok(NULL, ","));
            reply.process_id = atoi(strtok(NULL, "#"));

            //add to the set of quorums
            if(reply.counter == rCounter)
            {
              serverAck(reply.process_id, reply.ts);
              if(reply.ts > *maxTs)
                {
                  *maxTs = reply.ts;
                  *object = reply.value;
```

```
                *pobject = reply.pvalue;
              }
            }

          //Add again to the set of quorums
          else if(!FD_ISSET(socket_fd[i], &crashfds))
            FD_SET(socket_fd[i], &readfds);
        }

      }while(!isQuorumCompl(&compQ));

    free(reply.type);

    return compQ;
}


/*********************************************************************/
/*            Clears the servers who send a READACK message          */
/*********************************************************************/
void clearQuorums(int numServers)
{
    int i;

    for(i=0; i<numServers; i++)
      {
        servers[i].ack = 0;
        servers[i].ts = -1;
      }
}


/*********************************************************************/
/*  Add the server in the set of servers who reply with WRITEACK     */
/*********************************************************************/
void serverAck(int id, int ts)
{
    servers[id-1].ack = 1;
    servers[id-1].ts = ts;
}


/*********************************************************************/
/*     Checks if a whole quorum responded with READACK replies       */
/*********************************************************************/
int isQuorumCompl(struct quorum** q)
{
    struct quorum* temp;
    struct node* tmp;

    temp = quorumSystem;

    for(temp = quorumSystem; temp!=NULL; temp = temp->next)
      {
        for(tmp = temp->servers; tmp!=NULL; tmp = tmp->next)
        if(servers[(tmp->serverId)-1].ack == 0)
          break;

        if(tmp == NULL)
        {
```

```
        *q = temp;
        return 1;
      }
    }

  return 0;
}


/*********************************************************************/
/*                      Finds the quorum view                        */
/*********************************************************************/
int find_quorum_view(struct quorum* q, int maxTs)
{
  struct node* tmp;
  struct intersectionNode* tmp2;

  //Quorum View 1
  for(tmp = q->servers; tmp!=NULL && servers[(tmp->serverId)-
1].ts==maxTs; tmp = tmp->next);
  if(tmp == NULL)
    return 1;

  //Quorum View 3
  for(tmp2 = q->intersection;  tmp2!=NULL; tmp2 = tmp2->next)
    {
      for(tmp = tmp2->servers; tmp!=NULL && servers[(tmp->serverId)-
1].ts==maxTs; tmp=tmp->next);
      if(tmp == NULL)
      return 3;
    }

  //Quorum View 2
  return 2;
}
```