

Ατομική Διπλωματική Εργασία

**ΥΛΟΠΟΙΗΣΗ ΚΑΙ ΠΕΙΡΑΜΑΤΙΚΗ ΑΞΙΟΛΟΓΗΣΗ ΤΟΥ
ΑΛΓΟΡΙΘΜΟΥ ΧΡΟΝΟΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ LIS ΜΕ
ΧΡΗΣΗ ΤΗΣ ΠΛΑΤΦΟΡΜΑΣ YALPS**

Θέα Τράντα

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ



ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάιος 2014

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΥΛΟΠΟΙΗΣΗ ΚΑΙ ΠΕΙΡΑΜΑΤΙΚΗ ΑΞΙΟΛΟΓΗΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ ΧΡΟΝΟΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ LIS ΜΕ ΧΡΗΣΗ ΤΗΣ ΠΛΑΤΦΟΡΜΑΣ YALPS

Θέα Τράντα

Επιβλέπων Καθηγητής
Χρύσης Γεωργίου

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

Μάιος 2014

Ευχαριστίες

Θα ήθελα να εκφράσω τις ευχαριστίες μου στον επιβλέποντα καθηγητή της διπλωματικής μου Δρ.Χρύση Γεωργίου, για τη συνεχή καθοδήγηση και βοήθεια που μου παρείχε σε όλα τα στάδια της δημιουργίας της.

Θα ήθελα επίσης να ευχαριστήσω την οικογένειά για τη διαρκή υποστήριξη, που επέτρεψε την επιτυχή διεκπεραίωση των σπουδών μου.

Περίληψη

Τα τελευταία χρόνια έχουμε δει μια δραματική αύξηση στη ζήτηση της επεξεργασίας εργασιών οι οποίες απαιτούν μεγάλη υπολογιστική δύναμη. Οι μονοπύρηντοι επεξεργαστές δεν είναι πλέον σε θέση να αντιμετωπίσουν τις υψηλές υπολογιστικές απαιτήσεις αυτών των εργασιών. Ως αποτέλεσμα, παράλληλες μηχανές βασισμένες στους πολυπύρηντους επεξεργαστές και πλατφόρμες υπερ-υπολογιστών έχουν γίνει εμφανή υπολογιστικά περιβάλλοντα. Ωστόσο, ο υπολογισμός σε αυτά τα περιβάλλοντα δημιουργεί πολλές προκλήσεις. Για παράδειγμα, μία πρόκληση είναι η συνεργασία καταναμημένων διεργασιών (nodes) ούτως ώστε να επιτευχθεί η εκτέλεση μεγάλων υπολογιστικών συνόλων εργασιών (tasks). Για να είναι αποτελεσματική αυτή η συνεργασιακή εκτέλεση εργασιών, πρέπει το σύστημα να είναι σχεδιασμένο έτσι ώστε να είναι ανθεκτικό σε σφάλματα δικτύου και διεργασιών. Για την επίτευξη αυτής της αποτελεσματικότητας, έχουν αφιερωθεί πολλές μελέτες τις τελευταίες δύο δεκαετίες στην ανάπτυξη αλγοριθμικών λύσεων ανεχτικών σε σφάλματα δικτύου και διεργασιών για ποικίλες εκδοχές τέτοιων προβλημάτων συνεργασίας.

Στην διπλωματική αυτή θα επικεντρωθούμε στην εργασία των Anta, Georgiou, Kowalski και Zavou με τίτλο «Online Parallel Scheduling of Non-uniform Tasks: Trading Failures For Energy». Σε αυτή την εργασία περιγράφονται αλγοριθμικές λύσεις οι οποίες επιλύουν μια εκδοχή του προβλήματος εκτέλεσης καταναμημένων μη-ομοιόμορφων εργασιών, όπου οι εργασίες εισάγονται δυναμικά στο σύστημα και οι διεργασίες υπόκεινται σε σφάλματα κατάρρευσης και επανεκκίνησης. Στην παρούσα διπλωματική παρουσιάζεται η υλοποίηση και η πειραματική αξιολόγηση του αλγορίθμου (n,β)-LIS που αναφέρεται στην εργασία, με σκοπό να διαφανεί κατά πόσο αυτός ο αλγόριθμος είναι αποδοτικός και στην πράξη, σε πραγματικό περιβάλλον.

Αφού μελετήθηκε σε βάθος, ο αλγόριθμος υλοποιήθηκε στη γλώσσα προγραμματισμού JAVA με τη χρήση της βιβλιοθήκης YALPS, η οποία υποβοηθά την υλοποίηση καταναμημένων αλγορίθμων. Ακολούθησε η εκτέλεσή του σε πραγματικό περιβάλλον με χρήση του PlanetLab, ένα Διαδικτυακό σύστημα, με διάφορες παραμέτρους για την εξαγωγή αποτελεσμάτων τα οποία χρησιμοποιήθηκαν για την πειραματική αξιολόγηση του αλγορίθμου.

Περιεχόμενα

Κεφάλαιο 1	Εισαγωγή.....	1
	1.1 Κίνητρο Ατομικής Διπλωματικής Εργασίας.....	1
	1.2 Στόχος Ατομικής Διπλωματικής Εργασίας.....	2
	1.3 Μεθοδολογία.....	3
	1.4 Οργάνωση της Εργασίας.....	4
Κεφάλαιο 2	Υπόβαθρο.....	5
	2.1 Προηγούμενες Εργασίες.....	5
	2.2 Εργασία των Anta, Georgiou, Kowalski και Zavou.....	7
	2.3 Η Βιβλιοθήκη YALPS.....	10
	2.3.1 Διεπαφές της Βιβλιοθήκης YALPS.....	12
	2.3.2 Ανταλλαγή Μηνυμάτων.....	15
	2.3.3 Βασικά Συστατική και Εκτέλεση.....	16
	2.4 Το Δικτυακό Σύστημα PlanetLab.....	21
	2.4.1 Γενικά.....	21
	2.4.2 Σύντομος Οδηγός Χρήσης του PlanetLab.....	22
Κεφάλαιο 3	Μοντέλο Υπολογισμού.....	25
	3.1 Κατανεμημένο Περιβάλλον.....	25
	3.2 Λειτουργίες.....	26
	3.3 Κύκλοι Επεξεργαστών.....	27
	3.4 Διάταξη Γεγονότων.....	27
	3.5 Εργασίες.....	28
	3.6 Αντίπαλος.....	29
	3.7 Μέτρα Αποδοτικότητας.....	29

Κεφάλαιο 4	Περιγραφή και Υλοποίηση Αλγορίθμου.....	31
4.1	Αλγόριθμος (n,β)-LIS.....	31
4.1.1	Ψευδοκώδικας και Περιγραφή Αλγορίθμου (n,β)-LIS.....	32
4.1.2	Λεπτομέρειες Υλοποίησης Αλγορίθμου (n,β)-LIS.....	33
Κεφάλαιο 5	Πειραματική Αξιολόγηση Αλγορίθμου.....	47
5.1	Μεθοδολογία.....	47
5.1.1	Πλατφόρμα Πειραματισμού.....	48
5.1.2	Μετρικές Αξιολόγησης Αλγορίθμου.....	48
5.1.3	Σενάρια.....	51
5.2	Ανάλυση Αποτελεσμάτων.....	55
5.2.1	Αλγόριθμος (n,β)-LIS χωρίς σφάλματα.....	55
5.2.2	Αλγόριθμος (n,β)-LIS με σφάλματα.....	70
5.2.3	Σύγκριση Αλγορίθμου (n,β)-LIS με και χωρίς σφάλματα.....	79
5.3	Συμπεράσματα.....	81
Κεφάλαιο 6	Συμπεράσματα.....	84
6.1	Γενικά Συμπεράσματα.....	84
6.2	Προβλήματα Υλοποίησης και Αντιμετώπιση.....	85
6.3	Μελλοντική Εργασία.....	86
Βιβλιογραφία		88
Παράρτημα Α		A-1
	Πηγαίος Κώδικας Αλγορίθμου (n,β)-LIS.....	A-1
Παράρτημα Β		B-1
	Αποτελέσματα Εκτελέσεων.....	B-1

Κεφάλαιο 1

Εισαγωγή

1.1 Κίνητρο Ατομικής Διπλωματικής Εργασίας	1
1.2 Στόχος Ατομικής Διπλωματικής Εργασίας	2
1.3 Μεθοδολογία	3
1.4 Οργάνωση της Εργασίας	4

1.1 Κίνητρο Ατομικής Διπλωματικής Εργασίας

Τα τελευταία χρόνια έχουμε δει μια δραματική αύξηση στη ζήτηση της επεξεργασίας εργασιών οι οποίες απαιτούν μεγάλη υπολογιστική δύναμη. Οι μονοπύρηντοι επεξεργαστές δεν είναι πλέον σε θέση να αντιμετωπίσουν τις υψηλές υπολογιστικές απαιτήσεις αυτών των εργασιών. Ως αποτέλεσμα, παράλληλες μηχανές βασισμένες στους πολypύρηντους επεξεργαστές και πλατφόρμες υπέρ-υπολογιστών έχουν γίνει εμφανή υπολογιστικά περιβάλλοντα. Ωστόσο, ο υπολογισμός σε αυτά τα περιβάλλοντα δημιουργεί πολλές προκλήσεις. Για παράδειγμα, μία πρόκληση είναι η συνεργασία κατανεμημένων διεργασιών (nodes) ούτως ώστε να επιτευχθεί η εκτέλεση μεγάλων υπολογιστικών συνόλων εργασιών (tasks). Για να είναι αποτελεσματική αυτή η συνεργασιακή εκτέλεση εργασιών, πρέπει το σύστημα να είναι σχεδιασμένο έτσι ώστε να είναι ανθεκτικό σε σφάλματα δικτύου και διεργασιών. Για την επίτευξη αυτής της αποτελεσματικότητας, έχουν αφιερωθεί πολλές μελέτες τις τελευταίες δύο δεκαετίες στην ανάπτυξη αλγοριθμικών λύσεων ανεχτικών σε σφάλματα δικτύου και διεργασιών για ποικίλες εκδοχές τέτοιων προβλημάτων συνεργασίας(π.χ. [1] [2]).

Προκειμένου να προσδιοριστεί η ανταλλαγή μεταξύ της αποδοτικότητας τέτοιων αλγορίθμων και ανοχή σε σφάλματα, έχουν γίνει πολλές έρευνες οι οποίες μελέτησαν το γενικευμένο πρόβλημα, όπου η διεργασίες συνεργάζονται για την εκτέλεση m ανεξάρτητων εργασιών στην παρουσία σφαλμάτων (π.χ. [2] [3] [4]). Στην διπλωματική αυτή θα επικεντρωθούμε στην εργασία των Anta, Georgiou, Kowalski και Zavou με τίτλο «Online Parallel Scheduling of Non-uniform Tasks: Trading Failures For Energy[5]. Σε αυτή την εργασία περιγράφονται αλγοριθμικές λύσεις οι οποίες επιλύουν μια εκδοχή του προβλήματος εκτέλεσης κατανεμημένων εργασιών, όπου οι εργασίες είναι μη-ομοιόμορφες και εισάγονται δυναμικά και συνεχόμενα στο σύστημα και οι διεργασίες υπόκεινται σε σφάλματα κατάρρευσης και επανεκκίνησης.

Βάσει των πιο πάνω τίθεται το ερώτημα κατά πόσο οι αλγόριθμοι είναι αποδοτικοί και στην πράξη.

1.2 Στόχος Ατομικής Διπλωματικής Εργασίας

Η παρούσα διπλωματική στοχεύει στην υλοποίηση και πειραματική αξιολόγηση ενός από τους αλγορίθμους που αναφέρονται στην εργασία [5], με σκοπό να διαφανεί κατά πόσο αυτός ο αλγόριθμος είναι αποδοτικός και στην πράξη. Συγκεκριμένα, υλοποιήθηκε ο αλγόριθμος (n,β) -LIS, τον οποίο θα παρουσιάσουμε στη συνέχεια.

Για την υλοποίηση του αλγορίθμου χρησιμοποιήθηκε η βιβλιοθήκη YALPS [6] του ινστιτούτου επιστήμης και τεχνολογίας INRIA [7], μία ανοικτού κώδικα βιβλιοθήκη της γλώσσας προγραμματισμού JAVA [8], η οποία έχει σχεδιαστεί ακριβώς για να διευκολύνει την ανάπτυξη κατανεμημένων εφαρμογών. Η πειραματική αξιολόγηση έγινε σε πραγματικό κατανεμημένο σύστημα και συγκεκριμένα στο PlanetLab [9], μια ανοικτή πλατφόρμα για την ανάπτυξη και τη χρήση υπηρεσιών δικτύου παγκόσμιας κλίμακας.

Μέσω των εξαγόμενων αποτελεσμάτων που πάρθηκαν με διαφορετικές παραμέτρους, έγινε περαιτέρω μελέτη της απόδοσης του αλγορίθμου σε πραγματικό σύστημα, επιρρεπή σε σφάλματα κατάρρευσης και επανεκκίνησης.

1.3 Μεθοδολογία

Για την επίτευξη των στόχων της διπλωματικής αυτής εργασίας ακολούθησα την πιο κάτω μεθοδολογία.

Αρχικά μελέτησα το άρθρο που προαναφέρθηκε καθώς και άλλα σχετικά άρθρα με σκοπό την ευρύτερη κατανόηση του γενικότερου προβλήματος.

Αφού κατανόησα τη χρησιμότητα του αλγορίθμου αλλά και την ακριβής λειτουργία του προχώρησα στο επόμενο στάδιο, την υλοποίηση του αλγορίθμου. Για να επιτευχθεί αυτό έπρεπε να μελετήσω την βιβλιοθήκη YALPS [6]. Για την εξοικείωση με τη βιβλιοθήκη αναπτύχθηκαν διάφορα απλά παραδείγματα. Πρέπει να αναφερθεί ότι η βιβλιοθήκη YALPS δεν βρίσκεται σε ολοκληρωμένο στάδιο και η ύπαρξη του εγχειριδίου χρήσης βρίσκεται σε τυπικά επίπεδα. Ως εκ τούτου έπρεπε να αντιμετωπιστούν ποικίλα προβλήματα τα οποία ήταν αναμενόμενο ότι θα εμφανίζονταν.

Μετά την εκμάθηση της βιβλιοθήκης, υλοποιήθηκε ο αλγόριθμος (n, β) -LIS τον οποίο θα αναλύσουμε εκτενέστερα στην συνέχεια της διπλωματικής αυτής εργασίας.

Ως τελικό στάδιο, πραγματοποιήθηκε η εκτέλεση του αλγορίθμου σε πραγματικό περιβάλλον, στο διαδικτυακό σύστημα PlanetLab [9], με διάφορες παραμέτρους για την εξαγωγή αποτελεσμάτων τα οποία χρησιμοποιήθηκαν για την πειραματική αξιολόγηση με απώτερο σκοπό να διαφανεί κατά πόσο ο αλγόριθμος είναι αποδοτικός και στην πράξη.

1.4 Οργάνωση της Εργασίας

Η εργασία αυτή αποτελείται από έξι κεφάλαια, εξαιρουμένων της Βιβλιογραφίας και των Παραρτημάτων.

Στο Κεφάλαιο 1 παρουσιάζεται μια εισαγωγή του γενικότερου προβλήματος αλλά και τα βήματα τα οποία ακολουθήθηκαν.

Στο Κεφάλαιο 2 παρουσιάζεται το υπόβαθρο που χρειάζεται να έχει κανείς για να κατανοήσει καλύτερα την αλγοριθμική λύση, την υλοποίηση της και την πειραματική της αξιολόγηση. Συγκεκριμένα, δίνονται κάποιες σχετικές έννοιες, η γενική περιγραφή της αλγοριθμικής λύσης καθώς και μια σύντομη αλλά περιεκτική αναφορά για τη βιβλιοθήκη YALPS που χρησιμοποιήθηκε στην υλοποίηση του αλγορίθμου και για την δικτυακή πλατφόρμα PlanetLab που χρησιμοποιήθηκε στην πειραματική αξιολόγηση του αλγορίθμου.

Στο Κεφάλαιο 3 αναφέρεται στο μοντέλο υπολογισμού της αλγοριθμικής λύσης.

Στο Κεφάλαιο 4 παρουσιάζεται ο αλγόριθμος (n, β) -LIS ο οποίος επιλύει το πρόβλημα του Χρονοπρογραμματισμού σε ασύγχρονα κατανεμημένα συστήματα. Δίνεται μια περιεκτική περιγραφή του αλγορίθμου και αναφέρονται επίσης λεπτομέρειες υλοποίησης και δομής του κώδικα που δημιουργήθηκε.

Στο Κεφάλαιο 5 παρουσιάζεται η διαδικασία της πειραματικής αξιολόγησης καθώς και η ανάλυση των αποτελεσμάτων.

Τέλος, η εργασία κλείνει με το Κεφάλαιο 6 όπου περιγράφονται και συνοψίζονται τα γενικά συμπεράσματα, αναφέρονται τυχόν δυσκολίες που είχαν παρουσιαστεί κατά την εκπόνηση της εργασίας και διατυπώνονται ιδέες για μελλοντική εργασία στο θέμα αυτό.

Κεφάλαιο 2

Υπόβαθρο

2.1 Προηγούμενες Εργασίες	5
2.2 Εργασία των Anta, Georgiou, Kowalski και Zavou	7
2.3 Η Βιβλιοθήκη YALPS	10
2.3.1 Διεπαφές της Βιβλιοθήκης YALPS	12
2.3.2 Ανταλλαγή Μηνυμάτων	15
2.3.3 Βασικά Συστατική και Εκτέλεση	16
2.4 Το Δικτυακό Σύστημα PlanetLab	21
2.4.1 Γενικά	21
2.4.2 Σύντομος Οδηγός Χρήσης του PlanetLab	22

2.1 Προηγούμενες Εργασίες

Πολλές έρευνες έχουν αφιερωθεί στο πρόβλημα του χρονοπρογραμματισμού εργασιών (Task scheduling problem) και κάθε έρευνα αντιμετωπίζει διαφορετικές προκλήσεις. Για παράδειγμα, πολλές έρευνες αντιμετωπίζουν το πρόβλημα της δυναμικής εισαγωγής εργασιών αλλά δεν συμπεριλαμβάνουν καθόλου σφάλματα (π.χ. [10]). Άλλες έρευνες αντιμετωπίζουν το πρόβλημα σε μία μηχανή (π.χ. [11]), με το μειονέκτημα ότι δεν γίνεται αξιοποίηση του παραλληλισμού (με την προϋπόθεση ότι οι εργασίες είναι ανεξάρτητες). Άλλες έρευνες θεωρούν αποτυχίες, αλλά υποθέτουν ότι οι εργασίες που πρέπει να εκτελεστούν είναι γνωστές εκ των προτέρων και ο αριθμός τους είναι περιορισμένος (π.χ. [2] [3]), ενώ υπάρχουν και έρευνες οι οποίες θεωρούν ότι οι εργασίες είναι ομοιόμορφες, έχουν δηλαδή τους ίδιους χρόνους επεξεργασίας (π.χ. [4]).

Πολλές έρευνες ασχολούνται με θέματα διατήρησης ενέργειας αλλά δεν θεωρούν σφάλματα (π.χ. [10]).

Σε αυτή την διπλωματική εργασία θα επικεντρωθούμε στο έργο των Anta, Georgiou, Kowalski και Zανου [5]. Σε αυτή την εργασία περιγράφονται αλγοριθμικές λύσεις οι οποίες επιλύουν μια εκδοχή του προβλήματος εκτέλεσης καταναμεμένων εργασιών, όπου οι εργασίες είναι μη-ομοιόμορφες (έχουν δηλαδή διαφορετικούς χρόνους εκτέλεσης) και εισάγονται δυναμικά και συνεχόμενα στο σύστημα και επιπλέον οι διεργασίες υπόκεινται σε σφάλματα κατάρρευσης και επανεκκίνησης.

Η εργασία με την πιο στενή σχέση με το πιο πάνω έργο είναι η εργασία των Γεωργίου και Kowalski [4] όπου και μελετήθηκε στην Διατριβή του Ανδρέα Ανδρέου [12]. Σε αυτή την εργασία, παρόμοια με αυτή που θα μελετήσουμε, θεωρείται ένα πρόβλημα εκτέλεσης εργασιών όπου οι εργασίες εισάγονται δυναμικά και συνεχώς στο σύστημα και οι διεργασίες υπόκεινται σε σφάλματα κατάρρευσης και επανεκκίνησης. Σε αντίθεση με την εργασία που θα μελετήσουμε, στην εργασία [4] ο υπολογισμός είναι σπασμένος σε συγχρονισμένους γύρους και θεωρείται η έννοια της ανταγωνιστικότητας με βάση τις ανά-γύρο εργασίες σε αναμονή (per-round pending-task competitiveness). Επιπλέον, οι εργασίες έχουν μοναδιαίο κόστος, δηλαδή μπορούν να εκτελεστούν σε ένα γύρο. Οι συγγραφείς θεωρούν αρχικά έναν κεντρικό χρονοδρομολογητή και στη συνέχεια δείχνουν πώς και κάτω από ποιες προϋποθέσεις μπορεί να υλοποιηθεί σε ένα σύστημα μεταβίβασης μηνυμάτων (message-passing system). Δείχνουν ότι και ακόμα και με ένα κεντρικό χρονοδρομολογητή, κανένας αλγόριθμος δεν μπορεί να είναι ανταγωνιστικός αν οι εργασίες είναι μη-ομοιόμορφες. Αυτό το αποτέλεσμα, είναι ουσιαστικά το κίνητρο για την εργασία [5] που θα μελετήσουμε στην παρούσα εργασία. Αποτελεί δηλαδή το κίνητρο για την εισαγωγή της έννοιας της επιτάχυνσης (speed-up) και της μελέτης των καταστάσεων κάτω από τις οποίες είναι εφικτή η ανταγωνιστικότητα. Περισσότερες πληροφορίες για την εργασία [4] των Γεωργίου και Kowalski μπορούν να ανακτηθούν από το [12].

Πρέπει επίσης, να επισημάνουμε ότι η έννοια της ανταγωνιστικότητας (competitiveness), δηλαδή η σύγκριση της απόδοσης ενός συγκεκριμένου αλγορίθμου με την απόδοση του

βέλτιστου αλγόριθμου ο οποίος γνωρίζει εκ των προτέρων το σενάριο εκτέλεσης (offline algorithm), εισήχθη από τους Sleator και Tarján [13].

2.2 Εργασία των Anta, Georgiou, Kowalski και Zavou

Στην εργασία των Anta, Georgiou, Kowalski και Zavou [5] όπου και θα επικεντρωθούμε στην διπλωματική αυτή μοντελοποιείται και μελετείται η επίδραση του παραλληλισμού και των σφαλμάτων κατάρρευσης και επανεκκίνησης στην ανταγωνιστικότητα του πιο κάτω συστήματος.

Σύστημα υπολογισμού: Θεωρείστε ένα σύστημα στο οποίο μη-ομοιόμορφες εργασίες (έχουν δηλαδή διαφορετικούς χρόνους εκτέλεσης) εισάγονται δυναμικά και συνεχώς στο σύστημα και πρέπει να εκτελεστούν από ένα σύνολο επεξεργαστών οι οποίοι είναι επιρρεπείς σε σφάλματα κατάρρευσης και επανεκκίνησης. Σε ένα περιβάλλον χωρίς σφάλματα, μία απλή Longest-in-System (Περισσότερο-στο-Σύστημα) πολιτική χρονοπρογραμματισμού ενισχυμένη με ένα μηχανισμό αποφυγής πλεονασμού (redundancy-avoidance mechanism), μπορεί να εγγυηθεί την βελτιστότητα σε μία μακροπρόθεσμη εκτέλεση.

Ωστόσο, με την παρουσία των σφαλμάτων, ο χρονοπρογραμματισμός εργασιών γίνεται ένα πολύ πιο δύσκολο έργο. Συγκεκριμένα, καμία παράλληλη ντετερμινιστική αλγοριθμική λύση δεν μπορεί να είναι ανταγωνιστική έναντι στην βέλτιστη λύση (offline algorithm), ακόμη και με ένα μόνο επεξεργαστή και με εργασίες που έχουν μόνο δύο διαφορετικούς χρόνους εκτέλεσης. Έχει διαπιστωθεί ότι όταν επιπρόσθετη ενέργεια παρέχεται στο σύστημα με τη μορφή της επιτάχυνσης του επεξεργαστή (speed-up), τα δεδομένα αλλάζουν. Συγκεκριμένα, έχουν εντοπιστεί από τους συγγραφείς τα κατώτατα όρια για την επιτάχυνση σύμφωνα με την οποία δεν μπορεί να επιτευχθεί ανταγωνιστικότητα με οποιοδήποτε ντετερμινιστικό αλγόριθμο, ενώ πάνω από αυτά τα όρια υφίστανται ανταγωνιστικοί αλγόριθμοι. Στην εργασία προτείνονται αλγόριθμοι που επιτυγχάνουν μικρούς φραγμένους ανταγωνιστικούς δείκτες όταν η επιτάχυνση είναι πάνω από το όριο.

Αποδοτικότητα Αλγορίθμων: Η αποδοτικότητα μετριέται σε σχέση με το μέγιστο κόστος που εκκρεμεί (pending-cost) σε οποιοδήποτε σημείο της εκτέλεσης, όπου το pending-cost είναι το άθροισμα των χρόνων εκτέλεσης των εργασιών που έχουν εισαχθεί στο σύστημα αλλά δεν έχουν εκτελεστεί ακόμα. Επίσης μετριέται ο μέγιστος αριθμός των εργασιών που εκκρεμούν (pending-tasks) σε οποιοδήποτε σημείο της εκτέλεσης. Αυτό δίνει τη δυνατότητα να δούμε το πρόβλημα ως ένα online πρόβλημα και να αναλύσουμε τους αλγόριθμους ως προς την ανταγωνιστικότητά τους, δηλαδή να συγκριθεί η αποδοτικότητα ενός συγκεκριμένου αλγορίθμου με την αποδοτικότητα του βέλτιστου αλγόριθμου ο οποίος γνωρίζει εκ των προτέρων για τις καταρρεύσεις, τις επανεκκινήσεις και εισαγωγές των εργασιών. Η πρώτη μετρική είναι χρήσιμη για την αξιολόγηση του εναπομείναντα χρόνου επεξεργασίας που απαιτείται από το σύστημα σε κάθε σημείο του υπολογισμού, ενώ ο δεύτερη μετρική είναι χρήσιμη για την αξιολόγηση του αριθμού των εργασιών που εκκρεμούν για να εκτελεστούν, ανεξάρτητα από το χρόνο επεξεργασίας που απαιτείται.

Συνοπτικά τα αποτελέσματα: Στην εργασία [5] αποδείχθηκε ότι κανένας παράλληλος αλγόριθμος για το πρόβλημα υπό μελέτη δεν είναι ανταγωνιστικός έναντι του βέλτιστου offline αλγόριθμου. Ωστόσο, μπορεί ένας αλγόριθμος να γίνει ανταγωνιστικός εάν εφαρμοστεί επιτάχυνση στους επεξεργαστές πάνω από κάποιο συγκεκριμένο όριο. Επιτάχυνση s σημαίνει ότι ένας επεξεργαστής μπορεί να εκτελέσει μία εργασία s φορές πιο γρήγορα από τον καθορισμένο χρόνο της εργασίας στο σύστημα και επομένως έχει νόημα μόνο όταν $s \geq 1$. Η επιτάχυνση έχει επιπτώσεις στην κατανάλωση ενέργειας του επεξεργαστή.

Η έρευνα στοχεύει στην ανάπτυξη ανταγωνιστικών αλγορίθμων που απαιτούν την μικρότερη δυνατή επιτάχυνση. Ως εκ τούτου, μία από τις βασικές προκλήσεις της έρευνας αυτής είναι να προσδιοριστούν τα όρια της επιτάχυνσης, σύμφωνα με την οποία δεν μπορεί να επιτευχθεί ανταγωνιστικότητα και πάνω από τα οποία μπορεί να επιτευχθεί. Το έργο των συγγραφέων μπορεί να θεωρηθεί ως η διερεύνηση των ανταλλαγών μεταξύ της γνώσης και της ενέργειας με την παρουσία σφαλμάτων: Πόση ενέργεια (με τη μορφή της επιτάχυνσης) χρειάζεται ένας ντετερμινιστικός αλγόριθμος χρονοπρογραμματισμού για να μπορεί να ανταγωνιστεί σε επίδοση τον βέλτιστο offline

αλγόριθμο που έχει πλήρη γνώση των σφαλμάτων και των εισαγωγών εργασιών; (Είναι κατανοητό ότι δεν υπάρχει τίποτα για να ερευνηθεί εάν η βέλτιστη λύση κάνει και αυτή χρήση της επιτάχυνσης). Τα αποτελέσματα της έρευνας συνοψίζονται στον πιο κάτω πίνακα (βλέπε Πίνακα 1.1).

Condition	Task costs	Task competitiveness	Cost competitiveness	Algorithm
$s < c_{max}/c_{min}$ and $s < \frac{\gamma c_{min} + c_{max}}{c_{max}}$	≥ 2	∞	∞	Any
$s \geq c_{max}/c_{min}$	Any	1	c_{max}/c_{min}	(n, β) -LIS
$\frac{\gamma c_{min} + c_{max}}{c_{max}} \leq s < c_{max}/c_{min}$	2	1	1	γn -Burst
$s \geq 7/2$	Finite	c_{max}/c_{min}	1	LAF

Πίνακας 1.1 Περίληψη των αποτελεσμάτων [5]

Σε αυτό το σημείο πρέπει να εξηγήσουμε την παράμετρο γ . Η παράμετρος γ αναπαριστά τον μικρότερο αριθμό c_{min} -εργασιών που μπορεί ένας αλγόριθμος να ολοκληρώσει χρησιμοποιώντας επιτάχυνση s , έτσι ώστε ο βέλτιστος offline αλγόριθμος δεν μπορεί να ολοκληρώσει περισσότερες εργασίες στον ίδιο χρόνο. Σημειώστε ότι το c_{min} είναι το χαμηλότερο όριο στο κόστος των εργασιών που εισάγονται στο σύστημα και το c_{max} είναι το ψηλότερο. Περισσότερες λεπτομέρειες για το μοντέλο υπολογισμού βρίσκονται στο επόμενο κεφάλαιο.

Στην έρευνα αυτή, προτάθηκαν δύο προϋποθέσεις, (α) $s < c_{max}/c_{min}$ και (β) $s < \frac{\gamma c_{min} + c_{max}}{c_{max}}$ και αποδείχθηκε ότι αν και οι δύο ισχύουν, τότε κανένας ντετερμινιστικός σειριακός ή παράλληλος αλγόριθμος δεν μπορεί να πετύχει ανταγωνιστικότητα όταν χρησιμοποιεί επιτάχυνση s .

Επίσης, αποδείχθηκε ότι, για να πετύχουμε ανταγωνιστικότητα, είναι αρκετό να θέσουμε $s = c_{max}/c_{min}$ αν $c_{max}/c_{min} \in [1, \phi]$ και $s = 1 + \frac{\sqrt{1 - c_{min}/c_{max}}}{2}$ αλλιώς, όπου $\phi = \frac{1 + \sqrt{5}}{2}$.

Για την περίπτωση όπου η προϋπόθεση (α) δεν ισχύει, δηλαδή $s \geq c_{\max}/c_{\min}$, αναπτύχθηκε ο Αλγόριθμος (n,β)-LIS. Αποδείχθηκε ότι κάτω από αυτές τις συνθήκες, ο αλγόριθμος είναι 1-pending-task ανταγωνιστικός και $c_{\max}/c_{\min}$ -pending-cost ανταγωνιστικός, για παράμετρο $\beta \geq c_{\max}/c_{\min}$ και για οποιοδήποτε αριθμό επεξεργαστών n. Αυτά τα αποτελέσματα ισχύουν για οποιαδήποτε συλλογή από εργασίες (task) με κόστος στο εύρος $[c_{\min}, c_{\max}]$.

Τέλος, πρέπει να αναφερθεί ότι η μελέτη περιορίστηκε σε θεωρητικό επίπεδο και σκοπός της παρούσας Διπλωματικής Εργασίας είναι η επέκταση της μελέτης σε πρακτικό επίπεδο.

2.3 Η Βιβλιοθήκη YALPS

Η βιβλιοθήκη YALPS [6] είναι μία ανοικτού κώδικα βιβλιοθήκη της Java η οποία σχεδιάστηκε πρόσφατα από μια ομάδα ερευνητών του ερευνητικού ιδρύματος INRIA [7] με σκοπό να διευκολύνει το σχεδιασμό, την ανάπτυξη και τον έλεγχο κατανεμημένων εφαρμογών. Εφαρμογές οι οποίες αναπτύχθηκαν με χρήση της βιβλιοθήκης YALPS μπορούν να τρέξουν τόσο σε περιβάλλον προσομοίωσης όσο και σε πραγματικό περιβάλλον χωρίς καμία αλλαγή στον κώδικα της εφαρμογής. Η όλη προσαρμογή δε χρειάζεται μεταγλώττιση του πηγαίου κώδικα, πραγματοποιείται με απλά βήματα μέσω του αρχείου ρυθμίσεων για το οποίο θα μιλήσουμε εκτενέστερα στη συνέχεια αυτού του κεφαλαίου. Αυτό επιτυγχάνεται μέσω των δυο υλοποιήσεων που παρέχει η YALPS για τον διαχειριστή εργασιών, *SimulatedTaskManger* και *ExecutorTaskManager*, οι οποίες δίνουν τη δυνατότητα να τρέξει, είτε σε προσομοίωση είτε σε πραγματικό κατανεμημένο σύστημα αντίστοιχα. Αυτό είναι ένα πολύ σημαντικό πλεονέκτημα της βιβλιοθήκης γιατί με αυτό τον τρόπο δίνεται η δυνατότητα στους ερευνητές, αφού δημιουργήσουν το πρωτότυπο μιας εφαρμογής, να πειραματιστούν και να δοκιμάσουν τα διάφορα σενάρια τους στον προσομοιωτή προτού τα εφαρμόσουν σε πραγματικό σύστημα.

Μία εφαρμογή της βιβλιοθήκης YALPS είναι οργανωμένη σαν μία συλλογή από εργασίες¹ ‘Tasks’ τις οποίες διαχειρίζεται ο διαχειριστής εργασιών ‘Task Manager’. Ο διαχειριστής εργασιών επιτρέπει στις εργασίες να καθορίσουν τον τρόπο με τον οποίο θα εκτελεστούν. Υπάρχουν συνολικά τρεις τρόποι εκτέλεσης για μία εργασία: η άμεση εκτέλεση, η αναβεβλημένη εκτέλεση και η περιοδική εκτέλεση. Επίσης, για καλύτερη διαχείριση των εργασιών, η βιβλιοθήκη YALPS προσφέρει την επιπλέον δυνατότητα οργάνωσης των εργασιών σε ομάδες ‘TasksGroups’ έτσι ώστε οι εργασίες να μπορούν να τυγχάνουν διαχείρισης σαν ομάδα αντί σαν ατομικές εργασίες. Αυτό είναι σημαντικό διότι με μια απλή διαφοροποίηση σε μια ομάδα εργασιών μπορεί να αλλάξει τελείως η εκτέλεση της εφαρμογής.

Ένα άλλο στοιχείο της βιβλιοθήκης YALPS είναι ο τρόπος με τον οποίο επιτυγχάνεται η επικοινωνία μεταξύ των κατανεμημένων οντοτήτων στο σύστημα. Για το σκοπό αυτό υπάρχει το στρώμα επικοινωνίας της YALPS ‘YALPS CommunicationLayer’ για το οποίο υπάρχουν επίσης δύο υλοποιήσεις, μία υλοποίηση κατάλληλη για περιβάλλον προσομοίωσης και μία υλοποίηση κατάλληλη για εφαρμογή σε πραγματικό περιβάλλον. Για προσομοίωση υπάρχει μια ειδική υποδομή επικοινωνίας της βιβλιοθήκης YALPS ενώ για πραγματικό σύστημα τα μηνύματα της εφαρμογής δρομολογούνται μέσω των γνωστών πρωτοκόλλων UDP/TCP. Και στις δύο περιπτώσεις, το στρώμα επικοινωνίας της YALPS μας προσφέρει λειτουργίες για έλεγχο και αξιολόγηση των κατανεμημένων εφαρμογών, όπως είναι ο περιορισμός του εξερχόμενου εύρους ζώνης και η επιλογή για απώλεια μηνυμάτων κατά την εκτέλεση της εφαρμογής.

Οι εφαρμογές YALPS επικοινωνούν μέσω μηνυμάτων τα οποία απαιτείται να είναι σε σειριακή μορφή (serialized), δηλαδή μορφή κατάλληλη που να επιτρέπει τη μετάδοσή τους μέσω μιας σύνδεσης ενός δικτύου. Σε αυτή την εργασία, τα μηνύματα μετατρέπονται σε σειριακή μορφή αυτόματα μέσω της JAVA για να μπορέσουν να μεταδοθούν από τον ένα κόμβο στον άλλο μέσω του δικτύου, μέσω του πρωτοκόλλου TCP. Όταν το μήνυμα φτάσει στον παραλήπτη τότε μπορεί εύκολα να επανέλθει στην αρχική του μορφή.

¹ Διαφορετική έννοια από της εργασίες του προβλήματος εκτέλεσης εργασιών που μελετούμε

Υπάρχουν δύο τρόποι να επιτευχθεί αυτό σε μία εφαρμογή της YALPS. Ο ένας τρόπος είναι με την χρήση της πρότυπης μεθόδου σειριοποίησης της JAVA (κλάση `Serializable`), ενώ ο άλλος τρόπος είναι μέσω ενός πιο αποτελεσματικού μηχανισμού ο οποίος μπορεί να υλοποιηθεί μέσω της διεπαφής της βιβλιοθήκης YALPS και με βάση τις απαιτήσεις της εφαρμογής.

Στην παρούσα εργασία, έχοντας στη διάθεσή μας τη βιβλιοθήκη YALPS, η οποία μας παρέχει έτοιμα όλα τα απαραίτητα στοιχεία μίας κατανεμημένης εφαρμογής, υλοποιούνται δύο ασύγχρονοι κατανεμημένοι αλγόριθμοι οι οποίοι προσομοιώνονται και εφαρμόζονται σε πραγματικό κατανεμημένο σύστημα χωρίς να χρειάζεται να ανησυχούμε για τα πρακτικά θέματα δικτύων ή/και για άλλες τεχνικές δυσκολίες.

2.3.1 Διεπαφές της Βιβλιοθήκης YALPS

Διεπαφή NodeID

Οι κόμβοι μίας εφαρμογής YALPS δηλώνονται ως αντικείμενα της διεπαφής `NodeID`, ανεξάρτητα αν η εφαρμογή θα τρέξει σε περιβάλλον προσομοίωσης ή σε πραγματικό σύστημα. Στο παρασκήνιο όμως, υπάρχουν δύο κλάσεις που υλοποιούν τη διεπαφή `NodeID`, η κλάση `E_NodeID` και η κλάση `S_NodeID` οι οποίες χρησιμοποιούνται για την εκτέλεση της εφαρμογής σε πραγματικό σύστημα ή σε περιβάλλον προσομοίωσης, αντίστοιχα. Η επιλογή της υλοποίησης που θα εκτελεστεί καθορίζεται από το αρχείο διαμόρφωσης κατά την εκτέλεση της εφαρμογής.

Ένας κόμβος σε μία εφαρμογή της βιβλιοθήκης YALPS αρχικοποιείται με τον εξής τρόπο:

```
NodeID MyNode=cl.getNodeIdFromString(NodeName);
```

Όπου το αντικείμενο ‘cl’ αντιστοιχεί στο στρώμα επικοινωνίας το οποίο χρησιμοποιεί η εφαρμογή και η συνάρτηση ‘getNodeIdFromString(NodeName)’ επιστρέφει το μοναδικό αναγνωριστικό του κόμβου με το όνομα ‘NodeName’. Το μοναδικό αναγνωριστικό που επιστρέφει η συνάρτηση αποθηκεύεται στη μεταβλητή ‘MyNode’.

Διεπαφή YalpsNode

Η βασική αφαιρετική κλάση της βιβλιοθήκης YALPS είναι η κλάση AbstractYalpsNode η οποία υλοποιεί τη διεπαφή YalpsNode. Η AbstractYalpsNode απλοποιεί την κωδικοποίηση συστατικών μιας εφαρμογής τα οποία είναι απαραίτητα για όλες τις εφαρμογές της YALPS. Παραδείγματα τέτοιων συστατικών είναι η δήλωση ενός αντικειμένου ‘tm’ τύπου TaskManager το οποίο διαχειρίζεται τις εργασίες της εφαρμογής και η δήλωση ενός αντικειμένου ‘cl’ τύπου CommunicationLayer μέσω του οποίου θα επιτυγχάνεται η επικοινωνία μεταξύ των κόμβων της εφαρμογής.

Συνεπώς, μία κλάση σε κάθε εφαρμογή της YALPS μπορεί να επεκτείνει την κλάση AbstractYalpsNode με τον εξής τρόπο:

```
public class MyApplication extends AbstractYalpsNode{ }
```

Διεπαφή Task

Οι εργασίες (tasks) αποτελούν το ενεργό μέρος των εφαρμογών της YALPS. Μία εργασία είναι απλά ένα αντικείμενο που υλοποιεί τη διεπαφή ‘Task’. Για να ολοκληρωθεί η υλοποίηση της διεπαφής Task πρέπει να προστεθεί κώδικας στη μέθοδο run() του αντικειμένου ο οποίος θα περιγράφει το λειτουργικό μέρος της εργασίας. Το εργαλείο που χρησιμοποιούν οι εφαρμογές YALPS για να προγραμματίσουν τις εργασίες τους είναι το αντικείμενο τύπου ‘TaskManager’ που υπάρχει σε κάθε εφαρμογή. Επιπλέον, οι εργασίες μπορούν να ομαδοποιηθούν χρησιμοποιώντας το

μηχανισμό ομαδοποίησης εργασιών (TaskGroup) ούτως ώστε να τις διαχειρίζεται ο TaskManager' σαν ομάδα. Η δημιουργία μίας εργασίας γίνεται με τον εξής τρόπο:

```
class MyTask implements Task{
    public void run(){
        ...
    }
}
```

Υπάρχουν τρεις διαφορετικοί τύποι εργασιών, οι εργασίες που εκτελούνται αμέσως μετά την καταχώρησή τους, οι εργασίες που εκτελούνται σε συγκεκριμένο χρονικό διάστημα μετά την καταχώρησή τους και οι εργασίες που εκτελούνται περιοδικά. Η διεπαφή TaskManager' μας προσφέρει διάφορες μεθόδους για τον προγραμματισμό της εκτέλεσης των εργασιών.

- i. Άμεση εκτέλεση εργασιών: Η πιο κάτω μέθοδος καταχωρεί την εργασία t και τη συσχετίζει με την ομάδα εργασιών g. Αυτός ο τρόπος καταχώρησης της εργασίας t συνεπάγεται ότι η εργασία θα εκτελεστεί άμεσα.

```
void registerTask(Task t, TaskGroup g);
```

Η μέθοδος αυτή μπορεί να κληθεί από το διαχειριστή εργασιών 'tm' της εφαρμογής με τον εξής τρόπο, που αναφέρεται πιο κάτω, όπου tm.getSystemTaskGroup() θα επιστρέψει την προεπιλεγμένη ομάδα εργασιών του συστήματος.

```
tm.registerTask(new MyTask(), tm.getSystemTaskGroup());
```

- ii. Αναβιβλημένη εκτέλεση εργασιών: Αν θέλουμε μία εργασία να μην εκτελεστεί άμεσα αλλά μετά από κάποιο συγκεκριμένο χρονικό διάστημα, αυτό επιτυγχάνεται με τη μέθοδο:

```
void registerTask(Task t, TaskGroup g, long millis);
```

Η μέθοδος αυτή έχει την ίδια λειτουργία με την προηγούμενη, με τη διαφορά ότι έχει την επιπλέον παράμετρο 'long millis' η οποία προσδιορίζει ότι η εργασία θα εκτελεστεί μετά από χρονικό διάστημα 'millis' χιλιοστών του δευτερολέπτου.

- iii. Περιοδικές εργασίες: Οι εργασίες που εκτελούνται περιοδικά, δηλαδή ανά τακτά χρονικά διαστήματα. Για το σκοπό αυτό υπάρχει μέθοδος η οποία διαφέρει με τις προηγούμενες στο ότι καταχωρεί την εργασία 't' ως περιοδική εργασία (Periodic Task).

```
void registerPeriodicTask(PeriodicTask t, TaskGroup g, long millis);
```

Η 'PeriodicTask' είναι μία διεπαφή που εντάσσεται στη διεπαφή 'Task' και έχει κάποιες επιπλέον μεθόδους τις οποίες χρειάζονται αυτού του είδους οι εργασίες, όπως είναι η μέθοδος getPeriod(). Η μέθοδος αυτή επιστρέφει το χρονικό διάστημα στο οποίο η εργασία θα πρέπει να εκτελεστεί ξανά, κάτι το οποίο καθορίζει η παράμετρος 'long millis'.

2.3.2 Ανταλλαγή Μηνυμάτων

Αποστολή Μηνυμάτων

Η επικοινωνία μεταξύ των κόμβων μίας εφαρμογής YALPS γίνεται μέσω ενός αντικειμένου τύπου 'CommunicationLayer' το οποίο υπάρχει σε όλες τις εφαρμογές της YALPS. Το αντικείμενο αυτό έχει δύο διαφορετικές μεθόδους για αποστολή μηνυμάτων, χρησιμοποιώντας το πρωτόκολλο TCP ή UDP.

Η πιο κάτω μέθοδος στέλνει το μήνυμα ‘msg’ στον κόμβο ‘dest’ χρησιμοποιώντας το TCP πρωτόκολλο επικοινωνίας.

```
public void sendTCP(Serializable msg, NodeID dest) throws IOException;
```

Για την αποστολή του μηνύματος ‘msg’ στον κόμβο ‘dest’ χρησιμοποιώντας το UDP πρωτόκολλο επικοινωνίας χρησιμοποιούμε τη μέθοδο:

```
public void sendUDP(Serializable msg, NodeID dest) throws IOException;
```

Παραλαβή Μηνυμάτων

Ένας κόμβος σε μία εφαρμογή YALPS παραλαμβάνει όλα τα μηνύματα τα οποία προέρχονται από άλλους κόμβους της εφαρμογής και προορίζονται γι’ αυτόν. Αυτό επιτυγχάνεται με την υλοποίηση της πιο κάτω μεθόδου που ορίζεται στη διεπαφή ‘Reicever’.

```
public boolean receive(short header, Object msg, NodeID sender);
```

Η μέθοδος παραλαμβάνει το μήνυμα ‘msg’ με επικεφαλίδα ‘header’ από τον κόμβο ‘sender’.

2.3.3 Βασικά Συστατική και Εκτέλεση

Αρχικοποίηση του Κόμβου

Σημαντική μέθοδος της κλάσης ‘AbstractYalpsNode’ είναι η μέθοδος που επιτρέπει στον κόμβο να εκτελέσει όλες τις αρχικοποιήσεις που είναι απαραίτητες για την περαιτέρω λειτουργία της εφαρμογής. Οι αρχικοποιήσεις αυτές αφορούν συνήθως προγραμματισμό των εργασιών του κόμβου ή/και αποστολή κάποιων μηνυμάτων στους υπόλοιπους κόμβους της εφαρμογής.

```
public void startNode();
```

Η μέθοδος κληρονομείται και πάλι από την κλάση ‘AbstractYalpsNode’ και στην υλοποίησή της μπορούν να συμπεριληφθούν όλες οι αρχικοποιήσεις. Ένα παράδειγμα υλοποίησης της μεθόδου είναι το εξής:

```
public void startNode(){  
  
    tm.registerTask(new myTask(), tm.getSystemTaskGroup());  
  
}
```

Στην πιο πάνω υλοποίηση της μεθόδου ‘startNode()’ γίνεται προγραμματισμός των εργασιών του κόμβου.

Εκτελεστές μίας Εφαρμογής YALPS

Μετά την υλοποίηση κλάσης της διεπαφής ‘YalpsNode’ δημιουργείται η εφαρμογή YALPS η οποία μπορεί να εκτελεστεί είτε σε προσομοίωση είτε σε πραγματικό καταναμημένο σύστημα. Υπάρχουν δύο εκτελεστές της YALPS οι οποίοι χρησιμοποιούνται ανάλογα με τον τρόπο που θέλουμε να τρέξει η εφαρμογή.

Ο πιο κάτω εκτελεστής χρησιμοποιείται για να τρέξει η εφαρμογή σε πραγματικό σύστημα.

```
yalps.launchers.ExecutorNodeStarter
```

Ο πιο κάτω εκτελεστής χρησιμοποιείται για να τρέξει η εφαρμογή σε περιβάλλον προσομοίωσης.

```
yalps.launchers.SimulatedNodeStarter
```

Και οι δύο εκτελεστές παίρνουν ως είσοδο ένα αρχείο διαμόρφωσης, το οποίο προσδιορίζει ποιοι κόμβοι της εφαρμογής θα αρχικοποιηθούν και ακολούθως αναλαμβάνουν την εκτέλεσή τους.

Εκτέλεση εφαρμογής της βιβλιοθήκης YALPS

Ο εκτελεστής της YALPS που θα επιλεγεί για να εκτελέσει την εφαρμογή καθορίζει το περιβάλλον στο οποίο θα τρέξει, περιβάλλον προσομοίωσης ή πραγματικό σύστημα. Στην πρώτη περίπτωση χρησιμοποιείται ο εκτελεστής 'SimulatedNodeStarter' ενώ στη δεύτερη ο εκτελεστής 'ExecutorNodeStarter'.

Παράδειγμα εντολής για τρέξιμο της εφαρμογής σε περιβάλλον προσομοίωσης:

```
java -cp /yalps.jar launchers.SimulatedNodeStarter myExamplesim.conf
```

Παράδειγμα εντολής για τρέξιμο της εφαρμογής σε πραγματικό σύστημα:

```
java -cp /yalps.jar launchers.ExecutorNodeStarter myExampleexec.conf
```

Όπου myExample-sim.conf και myExample-exec.conf τα αντίστοιχα αρχεία διαμόρφωσης.

Αρχείο Διαμόρφωσης

Το αρχείο διαμόρφωσης (configuration file) της YALPS είναι ένα αρχείο ιδιοτήτων της JAVA το οποίο δηλώνει ένα σύνολο αντικειμένων με τις ιδιότητές τους. Το αρχείο διαμόρφωσης αποτελείται από δύο είδη δηλώσεων, μία για τις αρχικοποιήσεις αντικειμένων και μία για τη ρύθμιση των ιδιοτήτων τους.

i. Δηλώσεις Αρχικοποίησης Αντικειμένων

Οι δηλώσεις για αρχικοποίηση των αντικειμένων της εφαρμογής έχουν την ακόλουθη μορφή

```
<object name>.CLASS=<fully qualified class name>
```


Όπου ‘object name’ είναι το όνομα του αντικειμένου, ενώ ‘fully qualified class name’ είναι το απόλυτο όνομα της κλάσης του αντικειμένου. Με αυτό τον τρόπο αρχικοποιούμε αντικείμενα των οποίων ο κατασκευαστής δεν παίρνει παραμέτρους. Μπορούν όμως να καθοριστούν τυχόν ιδιότητες του αντικειμένου χρησιμοποιώντας δηλώσεις ρύθμισης ιδιοτήτων, όπως αυτές περιγράφονται πιο κάτω.

ii. Δηλώσεις Ρύθμισης Ιδιοτήτων

Οι δηλώσεις για ρύθμιση των ιδιοτήτων ενός αντικειμένου της εφαρμογής έχουν την ακόλουθη μορφή:

`<object name>.<property name>=<value>`

Όπου ‘object name’ το όνομα του αντικειμένου του οποίου θέλουμε να ρυθμίσουμε την ιδιότητα, ‘property name’ είναι το όνομα της ιδιότητας που θα ρυθμιστεί και ‘value’ η τιμή που θα πάρει η ιδιότητα.

Για να μπορεί να υπάρχει μία τέτοια δήλωση σε ένα αρχείο διαμόρφωσης πρέπει η κλάση του αντικειμένου ‘object name’ να περιέχει μία μέθοδο:

`void set+‘property name’();`

Όπου με το ‘+’ δηλώνεται η συνένωση των δύο συμβολοσειρών, ‘set’ και ‘property name’. Η παράμετρος της μεθόδου πρέπει να είναι συμβατή με τον τύπο της τιμής ‘value’. Ο εκτελεστής της YALPS περνά αυτόματα την τιμή ‘value’ σαν παράμετρο στην πιο πάνω μέθοδο μετατρέποντάς την στον κατάλληλο τύπο.

Ακολουθεί ένα παράδειγμα αρχείου διαμόρφωσης της YALPS:

```
node.CLASS = yalps.example.application.myApplication
node.nodeList=10
```

Για να είναι σωστό το πιο πάνω αρχείο διαμόρφωσης, θα πρέπει στην εφαρμογή myApplication να υπάρχει η μέθοδος:

```
void setNodeList(Type myParam) { }
```

Όπου ‘Type’ μπορεί να είναι οποιοσδήποτε τύπος (π.χ Integer, String) στον οποίο μπορεί να μετατραπεί η τιμή ‘10’.

Μία εφαρμογή η οποία προορίζεται για να τρέξει σε περιβάλλον προσομοίωσης ή σε πραγματικό σύστημα, και στις δύο περιπτώσεις το αρχείο διαμόρφωσης θα έχει την ίδια μορφή, με διαφορά στο σημείο που δηλώνονται οι κόμβοι της εφαρμογής.

Παράδειγμα δήλωσης των κόμβων εφαρμογής που θα τρέξει σε προσομοίωση:

```
node.nodeList=1;2;3;4;5;6;7;8;9;10
```

Παράδειγμα δήλωσης των κόμβων εφαρμογής που θα τρέξει σε πραγματικό σύστημα:

```
node.nodeList=hostname:port;hostname2:port2;....;
```

Οι κόμβοι στην προσομοίωση δεν είναι πραγματικοί και άρα δε χρειάζεται να δηλωθεί πραγματική διεύθυνση δικτύου ή πραγματικός αριθμός θύρας για τον κάθε κόμβο.

2.4 Το Δικτυακό Σύστημα PlanetLab

2.4.1 Γενικά

Το PlanetLab[9] είναι μια ανοικτή πλατφόρμα ιδανική για την ανάπτυξη και τη χρήση υπηρεσιών δικτύου παγκόσμιας κλίμακας. Είναι ένα γεωγραφικά καταναμημένο δίκτυο που ξεκίνησε αρχικά σαν μια ερευνητική κοινότητα με συνδέσεις μεταξύ διαφόρων πανεπιστημίων υπό την εποπτεία του Πανεπιστημίου Princeton. Είναι σχεδιασμένο έτσι ώστε να επιτρέπει στους επιστήμονες – απ’ όπου κι αν βρίσκονται – να δημιουργούν και να δοκιμάζουν νέου τύπου λογισμικά τα οποία δεν περιορίζονται σε έναν και μοναδικό υπολογιστή αλλά «τρέχουν» σε πολλούς υπολογιστές ταυτόχρονα χρησιμοποιώντας το παγκόσμιο δίκτυο σαν ένα μεγάλο υπολογιστή. Το PlanetLab, δηλαδή, δημιουργεί ένα μοναδικό περιβάλλον το οποίο επιτρέπει την πραγματοποίηση πειραμάτων σε παγκόσμια κλίμακα.

Αυτή τη στιγμή το PlanetLab αποτελείται από 1188 κόμβους σε 585 τοποθεσίες. Συνεργάζεται με το Παρατηρητήριο Abilene, ένα πρόγραμμα που υποστηρίζει την συλλογή πληροφοριών που αφορά την κίνηση δεδομένων στο Διαδίκτυο και που σχετίζεται με το δίκτυο Internet2. Με την συνεργασία αυτή το PlanetLab έχει πρόσβαση σε στοιχεία που αφορούν στατιστικά στοιχεία για την κυκλοφορία στους συνδέσμους του δικτύου, τη ροή πληροφοριών μέσα στο δίκτυο, τη δρομολόγηση δεδομένων, δηλαδή σε ποιο σημείο οδηγούνται τα πακέτα μέσα στο δίκτυο, τους δρομολογητές, την καθυστέρηση των δεδομένων, δηλαδή πόσο χρόνο χρειάζονται τα πακέτα για να φτάσουν στον προορισμό τους και το ρυθμό μετάδοσης των δεδομένων.

Στο φιλόδοξο αυτό πρόγραμμα λαμβάνουν μέρος τα μεγαλύτερα ινστιτούτα (εκπαιδευτικά κυρίως) ενώ στους χορηγούς του συμπεριλαμβάνονται δύο από τις μεγαλύτερες εταιρείες στον χώρο της τεχνολογίας, η Intel και η HP. Αυτή τη στιγμή υπάρχουν εκατοντάδες εφαρμογές που τρέχουν στο PlanetLab οι οποίες δημιουργήθηκαν από διάφορα πανεπιστήμια και άλλους οργανισμούς. Υπάρχουν όμως και απλές εφαρμογές που γίνονται από φοιτητές και μικρές ομάδες ερευνητών, οι οποίες μπορούν να χρησιμοποιηθούν από οποιονδήποτε, για ερευνητικούς και μη κερδοσκοπικούς σκοπούς.

Όραμα των δημιουργών του PlanetLab, οι οποίοι το αποκαλούν ως την «Νέα Γενεά του Διαδικτύου», είναι να πετύχουν μια αναβάθμιση του συστήματος, η οποία θα είναι ικανή να αντικαταστήσει ολόκληρο το διαδίκτυο.

Στην παρούσα διπλωματική εργασία χρησιμοποιούμε το PlanetLab EU (Europe) για την εφαρμογή και την πειραματική αξιολόγηση του Αλγορίθμου (n, β) -LIS. Από τον Φεβρουάριο του 2013, το PlanetLab EU αποτελείται από 349 κόμβους σε 156 τοποθεσίες.

Το PlanetLab είναι το καταλληλότερο περιβάλλον για την εφαρμογή και την πειραματική αξιολόγηση του αλγορίθμου (n, β) -LIS. Πρόκειται για ένα δικτυακό σύστημα που μας παρέχει ένα «πραγματικό» περιβάλλον όπου είναι από τη φύση του ασύγχρονο και ανά πάσα στιγμή μπορούν να συμβούν σφάλματα στους κόμβους και στις συνδέσεις. Είναι δηλαδή ιδανικό περιβάλλον για την πρακτική αξιολόγηση του Αλγορίθμου (n, β) -LIS για να μπορέσουμε να ελέγξουμε αν ο αλγόριθμος είναι σε θέση να αντιμετωπίσει αποδοτικά αυτές τις προκλήσεις.

2.4.2 Σύντομος Οδηγός Χρήσης του PlanetLab

Κάθε οργανισμός (site) ο οποίος συμμετέχει στο PlanetLab έχει στη διάθεσή του τουλάχιστον δύο μηχανές οι οποίες μπορούν να χρησιμοποιηθούν από χρήστες που ανήκουν σε άλλα sites. Για να μπορεί κάποιος να χρησιμοποιήσει το PlanetLab πρέπει πρώτα να γίνει μέλος του κάνοντας αίτηση εγγραφής σε κάποιο site. Αφού η αίτησή του εγκριθεί από τους PIs (Principal Investigators) του site, οι οποίοι είναι υπεύθυνοι για τη διαχείριση των νέων μελών του site και των διαθέσιμων πόρων (slices), πλέον ο αιτητής γίνεται επίσημο μέλος με προσωπικό λογαριασμό (account), αποκτά πρόσβαση στις μηχανές του PlanetLab και μπορεί να τρέξει ελεύθερα τις εφαρμογές του.

Οι πόροι του PlanetLab είναι οργανωμένοι σε μερίσματα (slices). Κάθε ακαδημαϊκός οργανισμός μπορεί να έχει μέχρι και 10 μερίσματα και κάθε μέρισμα έχει δικαίωμα χρήσης σε μέχρι 150 μηχανές. Ένα μέρισμα, δηλαδή, είναι μια συλλογή από εικονικές

μηχανές διαθέσιμες για μια υπηρεσία. Δεσμεύει ένα μέρος των πόρων (εύρος ζώνης δικτύου, μνήμη, χώρος δίσκου) το οποίο περιορίζεται ανάλογα με την κατάσταση δικτύου για κάθε χρήστη. Ο κάθε χρήστης μπορεί να δουλεύει προσωπικά και ανενόχλητα σε κάθε μηχανή μέσω του μερίσματός του χωρίς να επηρεάζει ή να επηρεάζεται από τους άλλους χρήστες που χρησιμοποιούν την ίδια μηχανή αλλά όλοι μπορούν να επηρεάζουν την απόδοση της μηχανής και του δικτύου. Αρχικά το μέρισμα είναι κενό χωρίς επεξεργαστές κειμένου ή κάποιο μεταγλωττιστή παρά μόνο τις βασικές εντολές UNIX. Η κάθε μηχανή μπορεί ανά πάσα στιγμή να μην είναι διαθέσιμη χωρίς να χαθούν οποιαδήποτε δεδομένα στο δίσκο και συμπεριφέρεται απρόσμενα ώστε να είναι μη αξιόπιστη.

Πρώτο βήμα του χρήστη είναι να αναθέσει μηχανές στο μέρισμά του δημιουργώντας το προσωπικό του δίκτυο για την εφαρμογή του. Από την στιγμή που ανατίθεται μια μηχανή από το PlanetLab χρειάζεται κάποιος χρόνος για να εγκριθεί από τον οργανισμό στον οποίο ανήκει η μηχανή, προτού ο χρήστης να μπορεί να την διαχειριστεί. Ο χρήστης δεν μπορεί να επιλέξει μηχανές οι οποίες ανήκουν στον οργανισμό του αφού είναι διαθέσιμες για άλλους χρήστες.

Για πρόσβαση σε οποιαδήποτε μηχανή πρέπει πρώτα να δημιουργηθεί ένα ζευγάρι κλειδιών μέσω SSH για σκοπούς επαλήθευσης της ταυτότητας. Το ζευγάρι αυτό αποτελείται από το Public Key και το Private Key. Το πρώτο γνωστοποιείται στο μέρισμα του χρήστη μέσω του λογαριασμού του από την επίσημη ιστοσελίδα του PlanetLab ενώ το δεύτερο δίνεται ως στοιχείο επαλήθευσης κατά τη διαδικασία σύνδεσης με την απομακρυσμένη μηχανή. Τα κλειδιά πρέπει να είναι κρυπτογραφημένα σύμφωνα με κωδική φράση που εισάγει ο χρήστης.

Η δημιουργία των δύο κλειδιών μπορεί να γίνει με την ακόλουθη εντολή:

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

Αφού γνωστοποιηθεί το Public Key στη σελίδα του PlanetLab [14], τότε ο χρήστης μπορεί να ενωθεί με κάποια μηχανή του PlanetLab εκτελώντας την εντολή:

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

Όπου είναι slice_name είναι το όνομα του μερίσματος του χρήστη το οποίο του έχει ανατεθεί από τον PI, id_rsa είναι το private κρυπτογραφημένο αρχείο που έχει ο χρήστης αποθηκευμένο στον τοπικό του χώρο και node_address είναι η διεύθυνση της μηχανής με την οποία θα γίνει η σύνδεση. Όλες οι διευθύνσεις των μηχανών βρίσκονται στην ιστοσελίδα του PlanetLab. Με την εκτέλεση της εντολής θα ζητηθεί από το χρήστη η κωδική φράση για πιστοποίηση της ταυτότητας του μερίσματος.

Αφού γίνει η σύνδεση με επιτυχία, πλέον ο χρήστης είναι έτοιμος να ξεκινήσει και να χρησιμοποιήσει τη μηχανή. Μπορεί να μεταφέρει είτε τα αρχεία του στο μέρισμα είτε απλά τα εκτελέσιμα αρχεία του στις μηχανές και να ξεκινήσει να τρέχει την εφαρμογή του. Αυτό μπορεί να γίνει μέσω κάποιου ειδικού λογισμικού (όπως winscp) με την παρακάτω εντολή:

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

Η οποία μεταφέρει το αρχείο με το όνομα filename από τον τρέχων κατάλογο στον προσωπικό χώρο του μερίσματος slice_name στη μηχανή node που ανήκει στο μέρισμα αυτό. Κατά την ολοκλήρωση της μεταφοράς ζητείται η κωδική φράση για πιστοποίηση της ταυτότητας του χρήστη.

Αναλυτικότερα εγχειρίδια χρήσης του PlanetLab μπορεί να βρει κάποιος στην επίσημη ιστοσελίδα του PlanetLab και μάλιστα με οπτικοακουστικά μέσα.

Κεφάλαιο 3

Μοντέλο Υπολογισμού

3.1 Κατανεμημένο Περιβάλλον	25
3.2 Λειτουργίες	26
3.3 Κύκλοι Επεξεργαστών	27
3.4 Διάταξη Γεγονότων	27
3.5 Εργασίες	28
3.6 Αντίπαλος	29
3.7 Μέτρα Αποδοτικότητας	29

3.1 Κατανεμημένο Περιβάλλον

Θεωρούμε ένα κατανεμημένο σύστημα το οποίο αποτελείται από n ασυγχρόνιστους ομοιογενείς επεξεργαστές/διεργασίες, επιρρεπείς σε σφάλματα, με μοναδικά αναγνωριστικά στο σύνολο $[n] = \{1, 2, \dots, n\}$.

Υποθέτουμε ότι οι διεργασίες² έχουν πρόσβαση σε ένα κοινόχρηστο αντικείμενο, το οποίο ονομάζεται Shared Repository ή Repository για συντομία. Αυτό το αντικείμενο αντιπροσωπεύει τη διασύνδεση του συστήματος που χρησιμοποιείται από τις διεργασίες για την υποβολή υπολογιστικών εργασιών και για την λήψη των ειδοποιήσεων σχετικά με τις εργασίες που έχουν εκτελεστεί. Το Repository δεν είναι χρονοπρογραμματιστής, αφού δεν παίρνει καθόλου αποφάσεις για την κατανομή των εργασιών στους επεξεργαστές. Οι αποφάσεις αυτές γίνονται από τους ίδιους του επεξεργαστές κατά την εκτέλεση του αλγορίθμου.

² Οι όροι «επεξεργαστής», «διεργασία» και «κόμβος» εκφράζουν το ίδιο νόημα.

Υποθέτουμε επίσης ότι κάθε διεργασία μπορεί να επικοινωνεί απευθείας με το Repository και τα μηνύματα δε χάνονται ή αλλοιώνονται κατά τη μεταφορά.

3.2 Λειτουργίες

Ο τύπος δεδομένων του Repository είναι μια σειρά, ένα set από εργασίες (το οποίο θα περιγραφεί αργότερα) που υποστηρίζει τρεις λειτουργίες: την εισαγωγή (inject), την παραλαβή (get), και την ενημέρωση (inform).

- i. Η λειτουργία inject εκτελείται στο Repository και προσθέτει εργασίες στο τρέχον σύνολο εργασιών, και όπως συζητείται παρακάτω, αυτή η λειτουργία ελέγχεται από έναν αντίπαλο. Οι άλλες δύο λειτουργίες εκτελούνται από τους επεξεργαστές/διεργασίες.
- ii. Με την εκτέλεση μιας λειτουργίας get, ένας επεξεργαστής αποκτά από το Repository το σύνολο των εκκρεμών εργασιών (Pending Tasks), δηλαδή τις εργασίες που έχουν εισαχθεί στο σύστημα, αλλά το Repository δεν έχει ενημερωθεί ακόμα ότι έχουν ολοκληρωθεί. Για την απλοποίηση του μοντέλου υποθέτουμε ότι αν δεν υπάρχουν εκκρεμείς εργασίες όταν εκτελεστεί η λειτουργία get, τότε μπλοκάρει μέχρι κάποια νέα εργασία να εισαχθεί στο σύστημα και στη συνέχεια επιστρέφει αμέσως το σύνολο των νέων εργασιών.
- iii. Με την ολοκλήρωση μίας εργασίας, ένας επεξεργαστής εκτελεί μια λειτουργία inform, η οποία ειδοποιεί το Repository για την ολοκλήρωση της εργασίας. Στη συνέχεια, το Repository αφαιρεί την εργασία αυτή από το σύνολο των εκκρεμών εργασιών. Αξίζει να σημειωθεί ότι αν ένας επεξεργαστής υποστεί σφάλμα κατάρρευσης, δεν θα ήταν χρήσιμο να ενημερώσει το Repository για την εργασία που έχει προγραμματίσει να εκτελέσει πριν πραγματικά να έχει ολοκληρώσει την εκτέλεσή της.

Κάθε λειτουργία που εκτελείται από έναν επεξεργαστή συνδέεται με μια χρονική στιγμή, εξαιρώντας την λειτουργία get που όπως αναφέραμε μπορεί να μπλοκάρει, και το αποτέλεσμα της λειτουργίας είναι στιγμιαίο (δηλαδή στο ίδιο σημείο χρόνου).

3.3 Κύκλοι Επεξεργαστών

Οι επεξεργαστές τρέχουν σε κύκλους πραγματικού χρόνου, που ελέγχονται από έναν αλγόριθμο. Κάθε κύκλος αποτελείται από μια λειτουργία get, μια ολοκλήρωση μιας εργασίας, και μια λειτουργία inform (αν η εργασία έχει ολοκληρωθεί). Υποθέτουμε ότι οι λειτουργίες get και inform καταναλώνουν αμελητέο χρόνο (εκτός και αν η λειτουργία get δεν βρει εργασία σε εκκρεμότητα και σε αυτή την περίπτωση μπλοκάρει αλλά επιστρέφει αμέσως όταν εισαχθεί νέα εργασία). Το υπολογιστικό τμήμα του κύκλου, το οποίο περιλαμβάνει την εκτέλεση μιας εργασίας, καταναλώνει το χρόνο που απαιτείται για την συγκεκριμένη εργασία να εκτελεστεί, ο οποίος διαιρείται με την επιτάχυνση $s \geq 1$.

Ένας κύκλος μπορεί να μην ολοκληρωθεί. Ένα σφάλμα κατάρρευσης μπορεί να διακόψει τον κύκλο ενός επεξεργαστή. Στη συνέχεια, όταν ο επεξεργαστής επανεκκινήσει, ξεκινά ένας νέος κύκλος.

3.4 Διάταξη γεγονότων

Λόγω της παράλληλης φύσης του συστήματος που υποθέσαμε, οι κύκλοι των επεξεργαστών μπορούν να επικαλύπτονται μεταξύ τους και με τις εισαγωγές των εργασιών (task injections). Ως εκ τούτου, προσδιορίζουμε την ακόλουθη διάταξη γεγονότων στο Repository σε χρόνο t : (α) πρώτα επεξεργάζονται οι λειτουργίες inform τις οποίες εκτελούν οι διεργασίες/πελάτες, (β) έπειτα γίνονται οι λειτουργίες εισαγωγής εργασιών και (γ) τελευταίες οι λειτουργίες get των επεξεργαστών. Αυτό σημαίνει ότι το σύνολο των εργασιών που εκκρεμούν που επέστρεψε μία λειτουργία get που εκτελέστηκε σε χρόνο t περιλαμβάνει, εκτός από τις παλιές εργασίες που δεν εκτελέστηκαν, τις εργασίες που εισάχθηκαν στο χρόνο t , και δεν περιλαμβάνει τις

εργασίες που ενημερώθηκε το Repository ότι εκτελέστηκαν στο χρόνο t . Πρέπει να αναφερθεί ότι αυτή η διάταξη γεγονότων γίνεται μόνο για την ευκολία της παρουσίασης και της αιτιολογίας και δεν επηρεάζει τη γενικότητα των αποτελεσμάτων.

3.5 Εργασίες

Κάθε εργασία αποτελείται από ένα μοναδικό αναγνωριστικό, το χρόνο άφιξής της (ο χρόνος που εισάχθηκε στο σύστημα με βάση το ρολόι του Repository) , καθώς και το κόστος της, το οποίο μετράται ως ο χρόνος που απαιτείται για να εκτελεστεί (χωρίς επιτάχυνση s). Υποθέτουμε ότι το c_{min} και c_{max} δηλώνουν το μικρότερο και το μεγαλύτερο, αντίστοιχα, κόστος που μπορεί να έχουν οι εργασίες. Αναφερόμαστε σε μια εργασία με κόστος $c \in [c_{min}, c_{max}]$ ως c -εργασία.

Υποθέτουμε ότι οι εργασίες είναι ατομικές σε σχέση με την ολοκλήρωσή τους: δηλαδή αν ένας επεξεργαστής σταματήσει να εκτελεί μια εργασία λόγω κατάρρευσης πριν από την ολοκλήρωση ολόκληρης της εργασίας, τότε δεν υπάρχει μερική πληροφορία την οποία μπορούν να μοιραστούν με το Repository, ούτε ο επεξεργαστής μπορεί να ξαναρχίσει την εκτέλεση της εργασίας από το σημείο που σταμάτησε. Πρέπει να σημειωθεί επίσης ότι αν ένας επεξεργαστής ολοκληρώσει μία εργασία, αλλά καταρρεύσει πριν εκτελέσει την λειτουργία `inform`, η εργασία αυτή δεν θεωρείται ολοκληρωμένη.

Τέλος, οι εργασίες θεωρούνται ότι είναι όμοιες, ανεξάρτητες και ταυτοδύναμες. Με την ομοιότητα εννοούμε ότι οι εργασίες απαιτούν τους ίδιους ή συγκρίσιμους πόρους για την εκτέλεσή τους. Με την ανεξαρτησία εννοούμε ότι η ολοκλήρωση των εργασιών δεν επηρεάζει οποιαδήποτε άλλη εργασία, καθώς και κάθε εργασία μπορεί να γίνει ταυτόχρονα με οποιαδήποτε άλλη εργασία. Με την ταυτοδυναμία εννοούμε ότι κάθε εργασία μπορεί να εκτελεστεί μία ή περισσότερες φορές παράγοντας το ίδιο αποτέλεσμα. Πολλές εφαρμογές που περιλαμβάνουν εργασίες με τέτοιες ιδιότητες που συζητήθηκαν στο [2] .

3.6 Αντίπαλος

Για την μοντελοποίηση σφαλμάτων και μελέτη χειρότερης περίπτωσης θεωρούμε την ύπαρξη ενός αντιπάλου/εχθρού (adversary) που μπορεί να προκαλέσει καταρρεύσεις και επανεκκινήσεις επεξεργαστών (processor crashes and restarts), καθώς και εισαγωγές εργασιών στο Repository. Ο αντίπαλος προσπαθεί να μειώσει όσο πιο πολύ μπορεί την απόδοση του αλγορίθμου.

Ορίζουμε ένα εχθρικό μοτίβο A ως μία συλλογή από καταρρεύσεις, επανεκκινήσεις και εισαγωγές εργασιών που προκαλούνται από τον αντίπαλο. Το κάθε γεγονός συσχετίζεται με την χρονική στιγμή που συμβαίνει.

Λέμε ότι ένας επεξεργαστής/διεργασία είναι «ζωντανός» στο χρονικό διάστημα $[t, t']$ αν ο επεξεργαστής λειτουργεί στη χρονική στιγμή t και δεν καταρρέει μέχρι τη χρονική στιγμή t' . Υποθέτουμε ότι ένας επεξεργαστής που έχει επανεκκινήσει έχει γνώση μόνο του αλγορίθμου που εκτελείται και της παραμέτρου n (αριθμός των επεξεργαστών). Έτσι, κατά την επανεκκίνηση ένας επεξεργαστής ξεκινά απλά ένα νέο κύκλο.

3.7 Μέτρα Αποδοτικότητας

Η αξιολόγηση των αλγορίθμων γίνεται χρησιμοποιώντας ως μέτρο το κόστος των εργασιών που εκκρεμούν (pending cost) το οποίο ορίζεται ως εξής: Λαμβάνοντας υπόψη ένα χρονικό σημείο $t \geq 0$ στην εκτέλεση ενός αλγορίθμου ALG υπό το εχθρικό μοτίβο A , ορίζουμε το κόστος των εργασιών που εκκρεμούν στο χρόνο t , $C_t(ALG, A)$, όπου είναι το άθροισμα των κοστών των εκκρεμών εργασιών στο Repository τη χρονική στιγμή t . Επιπλέον, δηλώνεται ο αριθμός των εκκρεμών εργασιών (pending tasks) στο Repository τη χρονική στιγμή t , $T_t(ALG, A)$.

Αφού θεωρούμε το πρόβλημα της επεξεργασίας εργασιών ως ένα online πρόβλημα, επιδιώκουμε ανταγωνιστική ανάλυση. Συγκεκριμένα, λέμε ότι ένας αλγόριθμος ALG είναι x -pending-cost ανταγωνιστικός αν $C_t(ALG, A) \leq x \cdot C_t(OPT, A) + \Delta$, για

οποιοδήποτε t και κάτω από ένα εχθρικό μοτίβο A . Το Δ μπορεί να είναι οποιαδήποτε έκφραση ανεξάρτητη του A . Το $C_t(\text{OPT}, A)$ είναι το μικρότερο pending-cost που επιτεύχθηκε από οποιοδήποτε βέλτιστο (offline) αλγόριθμο – ο οποίος γνωρίζει εκ των προτέρων το A και έχει απεριόριστη υπολογιστική δύναμη – στην χρονική στιγμή t της εκτέλεσής του κάτω από το εχθρικό μοτίβο A . Παρόμοια, λέμε ότι ένας αλγόριθμος ALG είναι x -pending-task ανταγωνιστικός αν $T_t(\text{ALG}, A) \leq x \cdot T_t(\text{OPT}, A) + \Delta$, όπου το $T_t(\text{OPT}, A)$ είναι ανάλογο του $C_t(\text{OPT}, A)$.

Κεφάλαιο 4

Περιγραφή και Υλοποίηση Αλγορίθμου

4.1 Αλγόριθμος (n,β)-LIS	31
4.1.1 Ψευδοκώδικας και Περιγραφή Αλγορίθμου (n,β)-LIS	32
4.1.2 Λεπτομέρειες Υλοποίησης Αλγορίθμου (n,β)-LIS	33

4.1 Αλγόριθμος (n,β)-LIS

Σε αυτό το κεφάλαιο παρουσιάζεται ο Αλγόριθμος (n,β)-LIS, ο οποίος ισορροπεί ανάμεσα σε δύο έννοιες: στον χρονοπρογραμματισμό εργασιών με βάση το χρόνο που βρίσκονται στο σύστημα (Longest-In-System scheduling) και στην αποφυγή του πλεονασμού. Πιο συγκεκριμένα, ο αλγόριθμος ο οποίος τρέχει σε κάθε επεξεργαστή, προσπαθεί να κάνει τον επεξεργαστή να εκτελέσει την εργασία που περιμένει για τον περισσότερο χρόνο στο σύστημα και δεν προκαλεί πλεονασμό στην εκτέλεση των εργασιών αν ο αριθμός των εργασιών που εκκρεμούν είναι αρκετά μεγάλος.

Οι εργασίες που εισάγονται στο σύστημα, είναι μη-ομοιόμορφες (έχουν δηλαδή διαφορετικούς χρόνους εκτέλεσης) και καταφθάνουν στο Repository δυναμικά και συνεχόμενα. Αξίζει επίσης να σημειωθεί ότι οι εισαγωγές εργασιών (task injections), έχουν ως αποτέλεσμα οι νέες εργασίες να μπαίνουν στο τέλος της λίστας Pending, δίνοντας έτσι προτεραιότητα στις πιο «παλιές» εργασίες, τις εργασίες δηλαδή που είναι περισσότερο χρόνο στο σύστημα.

Αποδείχθηκε ήδη στο [5] ότι ο αλγόριθμος για επιτάχυνση $s \geq c_{\max} / c_{\min}$ είναι σε θέση να ολοκληρώσει μία εργασία για κάθε εργασία που ολοκληρώνει ο βέλτιστος αλγόριθμος. Επιπρόσθετα, αποδείχθηκε ότι αν υπάρχουν σε κάθε χρονική στιγμή τουλάχιστον βn^2 εργασίες εν αναμονή, για $\beta \geq c_{\max} / c_{\min}$, δύο επεξεργαστές δεν θα

εκτελέσουν ποτέ την ίδια εργασία. Συνδυάζοντας αυτές τις δύο παρατηρήσεις αποδείχθηκε ότι ο αλγόριθμος (n,β) -LIS είναι 1-task-competitive.

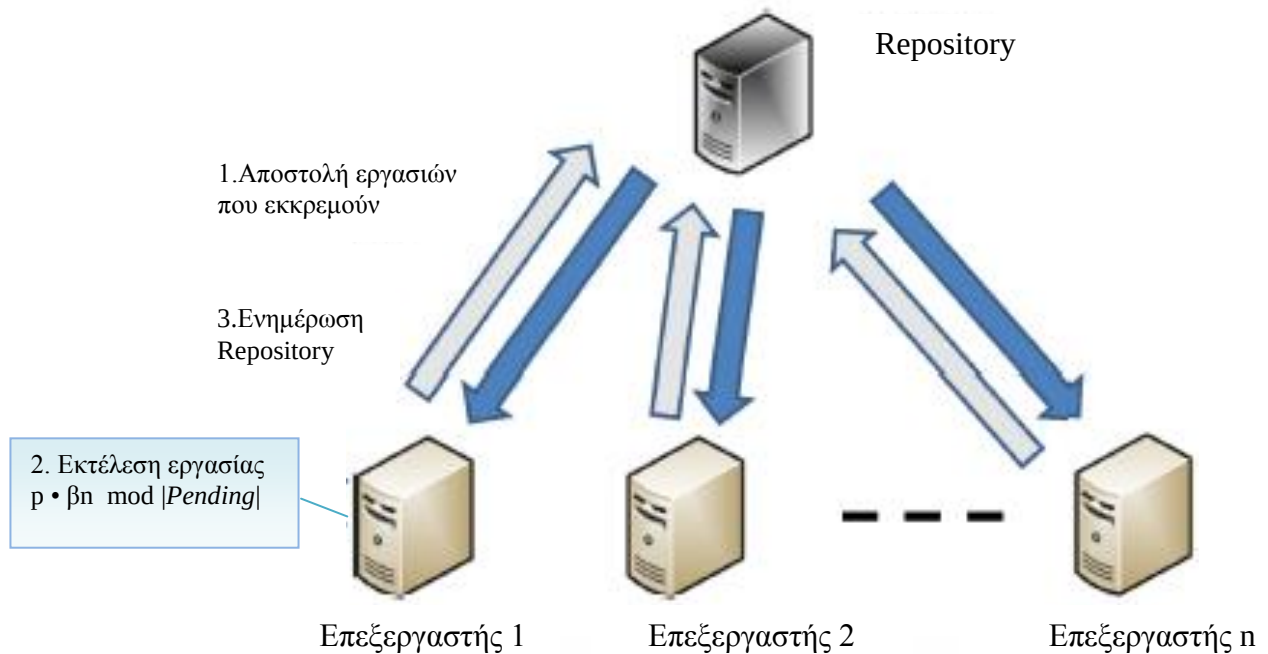
4.1.1 Ψευδοκώδικας και Περιγραφή Αλγορίθμου (n,β) -LIS

Algorithm (n,β) -LIS (for processor p)	
Repeat	//Upon awaking or restart, start here
Get from the Repository the set of pending tasks <i>Pending</i> ;	
Sort <i>Pending</i> by task arrival and ids/costs;	
If $ Pending \geq 1$	
then perform task with rank $p \cdot \beta n \bmod Pending $;	
Inform the Repository of the task performed.	

Όπως αναφέρθηκε στο υποκεφάλαιο 3.3, το σύστημα που υποθέσαμε είναι παράλληλο και οι επεξεργαστές τρέχουν σε κύκλους.

Η λειτουργία του αλγορίθμου (n,β) -LIS είναι η εξής: Ένας ολοκληρωμένος κύκλος επεξεργαστή αποτελείται από τρία βήματα. Αρχικά, στο πρώτο βήμα ο επεξεργαστής εκτελεί μία λειτουργία *get* και παίρνει από το Repository την λίστα με τις εργασίες που εκκρεμούν *Pending*. Έπειτα, μετά την παραλαβή της λίστας από το Repository, αν η λίστα δεν είναι κενή, υπάρχουν δηλαδή εργασίες που εκκρεμούν, τότε ο επεξεργαστής επιλέγει την εργασία που θα εκτελέσει με βάση την πολιτική LIS (Longest-In-System) και ακολούθως την εκτελεί. Σε αυτό το σημείο πρέπει να επισημανθεί ότι η εκτέλεση των εργασιών γίνεται εικονικά, δηλαδή δημιουργείται μία καθυστέρηση στον επεξεργαστή ανάλογα με το κόστος της εκάστοτε εργασίας που επιλέγει να εκτελέσει. Τέλος, ο επεξεργαστής εκτελεί μία λειτουργία *inform* για να ενημερώσει το Repository ότι εκτέλεσε την συγκεκριμένη εργασία.

Στο πιο κάτω σχήμα (Σχήμα 4.1) περιγράφεται με βήματα ένας ολοκληρωμένος κύκλος του «Επεξεργαστή 1».



Σχήμα 4.1: Περιγραφή λειτουργίας αλγορίθμου (n,β)-LIS

4.1.2 Λεπτομέρειες Υλοποίησης Αλγορίθμου (n,β)-LIS

Στο υποκεφάλαιο αυτό περιγράφονται αρχικά όλες οι κλάσεις οι οποίες είναι απαραίτητες για την ορθή λειτουργία του αλγορίθμου. Όλες οι κλάσεις υλοποιήθηκαν βάσει των προδιαγραφών του αλγορίθμου.

Ακολούθως, περιγράφεται η υλοποίηση του αλγορίθμου καθώς και μηχανισμοί οι οποίοι υλοποιήθηκαν για την ορθή λειτουργία του όπως η αποστολή και παραλαβή μηνυμάτων και η επιλογή και εκτέλεση εργασιών.

Κλάση Get_Message

Η κλάση Get_Message προσομοιώνει την λειτουργία get που εκτελούν οι επεξεργαστές όπως αυτή περιγράφηκε στο υποκεφάλαιο 3.2. Η κλάση υλοποιεί τη διεπαφή serializable η οποία προσφέρει τη δυνατότητα σειριοποίησης των δεδομένων. Η σειριοποίηση των δεδομένων είναι απαραίτητο χαρακτηριστικό για την αποστολή των αντικειμένων μέσω δικτύου.

```
public class Get_Message implements Serializable {  
    private static final long serialVersionUID = 1L;  
    int m_get;  
    Get_Message(int m_get) {  
        this.m_get = m_get;  
    }  
}
```

Το κάθε αντικείμενο τύπου Get_Message προσομοιώνει ένα μήνυμα get το οποίο στέλνει ένας κόμβος/επεξεργαστής στο Repository για να δηλώσει ότι θέλει να παραλάβει την λίστα με τις εργασίες που εκκρεμούν (Pending).

Κλάση List_Message

Τα αντικείμενα τύπου List_Message είναι στην ουσία τα μηνύματα που στέλνει το Repository ως απάντηση στους κόμβους/επεξεργαστές οι οποίοι εκτέλεσαν μία λειτουργία get, έστειλαν δηλαδή ένα μήνυμα τύπου Get_message στο Repository. Η κλάση υλοποιεί τη διεπαφή serializable η οποία προσφέρει τη δυνατότητα σειριοποίησης των δεδομένων. Η σειριοποίηση των δεδομένων είναι απαραίτητο χαρακτηριστικό για την αποστολή των αντικειμένων μέσω δικτύου.


```

public class List_Message implements Serializable {

    private static final long serialVersionUID = 1L;

    // ArrayList with Pending tasks
    public ArrayList<Integer> pending = new ArrayList<Integer>();

    // ArrayList with Duration
    public ArrayList<Integer> duration = new ArrayList<Integer>();

    List_Message(ArrayList<Integer> pending, ArrayList<Integer> duration){
        this.pending = (ArrayList<Integer>) pending.clone();
        this.duration = (ArrayList<Integer>) duration.clone();
    }
}

```

Οι επεξεργαστές αφού παραλάβουν το μήνυμα τύπου List_Message, εφαρμόζουν την πολιτική του αλγορίθμου με την οποία επιλέγουν ποια εργασία θα εκτελέσουν και έπειτα την εκτελούν.

Κλάση Inform_Message

Η κλάση Inform_Message προσομοιώνει την λειτουργία inform που εκτελούν οι κόμβοι/επεξεργαστές όπως αυτή περιγράφηκε στο υποκεφάλαιο 3.2. Τα αντικείμενα τύπου Inform_Message είναι στην ουσία τα μηνύματα που στέλνουν οι κόμβοι/επεξεργαστές στο Repository μετά την εκτέλεση της εργασίας, για να ενημερώσουν ότι η συγκεκριμένη εργασία, της οποίας το αναγνωριστικό (μεταβλητή task_id) περικλείεται στο μήνυμα, έχει εκτελεστεί. Η κλάση υλοποιεί τη διεπαφή serializable η οποία προσφέρει τη δυνατότητα σειριοποίησης των δεδομένων. Η σειριοποίηση των δεδομένων είναι απαραίτητο χαρακτηριστικό για την αποστολή των αντικειμένων μέσω δικτύου.

```

public class Inform_Message implements Serializable {

    private static final long serialVersionUID = 1L;

    int task_id;

    Inform_Message(int task) {

        this.task_id = task;

    }

}

```

Το Repository αφού παραλάβει ένα μήνυμα τύπου Inform_Message, τότε διαγράφει από την λίστα Pending την συγκεκριμένη εργασία.

Κλάση MyTCPBasedMsgSenderTask

Αυτή η κλάση είναι υπεύθυνη για την αποστολή των μηνυμάτων χρησιμοποιώντας το πρωτόκολλο TCP και για τον καθορισμό του τύπου του μηνύματος που θα σταλεί.

Όπως φαίνεται στο πιο κάτω κομμάτι κώδικα, η κλάση MyTCPBasedMsgSenderTask υλοποιεί τη διεπαφή Task και περιέχει τα εξής: (α) τον προορισμό του μηνύματος destination που είναι τύπου E_NodeID, (β) το στρώμα επικοινωνίας το οποίο είναι τύπου κατάλληλο για εκτέλεση σε πραγματικό σύστημα commLayer και (γ) τρία αντικείμενα τύπου Get_Message, Inform_Message και List_Message, τα οποία αρχικοποιούνται με null και χρησιμεύουν ως σημαφόροι για να αναγνωρίζουμε το είδος του μηνύματος που επιθυμούμε να αποσταλεί κάθε φορά.

```

public class MyTCPBasedMsgSenderTask implements Task {

    E_NodeID destination;
    RealWorldCommunicationLayer commLayer;
    Get_Message Get = null;
    Inform_Message Inform = null;
    List_Message List = null;
}

```

```

        . . .
    }

```

Υπάρχουν τρεις κατασκευαστές (constructors) για αυτή την κλάση όπως φαίνονται πιο κάτω.

```

public MyTCPBasedMsgSenderTask(Get_Message msg1, NodeID dest,
    CommunicationLayer comm) {
    this.Get = msg1;
    this.destination = (E_NodeID) dest;
    this.commLayer = (RealWorldCommunicationLayer) comm;
}

```

```

public MyTCPBasedMsgSenderTask(Inform_Message msg2, NodeID dest,
    CommunicationLayer comm) {
    this.Inform = msg2;
    this.destination = (E_NodeID) dest;
    this.commLayer = (RealWorldCommunicationLayer) comm;
}

```

```

public MyTCPBasedMsgSenderTask(List_Message msg3, NodeID dest,
    CommunicationLayer comm) {
    this.List = msg3;
    this.destination = (E_NodeID) dest;
    this.commLayer = (RealWorldCommunicationLayer) comm;
}

```

Κάθε φορά που επιθυμούμε να στείλουμε ένα μήνυμα χρησιμοποιώντας το πρωτόκολλο TCP, καλείται αυτή η κλάση και έπειτα καλείται ο ανάλογος κατασκευαστής με βάση τον τύπο του μηνύματος που επιθυμούμε να σταλεί.

Το ενεργό μέρος αυτής της κλάσης είναι η μέθοδος run(). Σε αυτή τη μέθοδο ελέγχονται οι τιμές των τριών σημαφόρων που προαναφέραμε και ανάλογα με το ποιος δεν είναι null στέλνεται το αντίστοιχο μήνυμα στον προορισμό destination.

```

@Override
public void run() {
    if (Get != null) { // msg to send is Serializable

        try {
            commLayer.sendTCP(Get, destination);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    if (Inform != null) { // msg to send is Serializable

        try {
            commLayer.sendTCP(Inform, destination);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    if (List != null) { // msg to send is Serializable

        try {
            commLayer.sendTCP(List, destination);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

Κλάση MyLIS – Υλοποίηση Αλγορίθμου

Με αυτή την κλάση υλοποιήθηκε ο αλγόριθμος (n, β) -LIS. Αρχικά πρέπει να αναφέρουμε ότι λόγω φιλοσοφίας της βιβλιοθήκης YALPS έπρεπε μία

διεργασία/κόμβος να έχει το ρόλο του Repository. Έτσι μία προκαθορισμένη διεργασία εκτελεί το μέρος του Repository σε αντίθεση με όλες τις υπόλοιπες που εκτελούν το μέρος του κόμβου/επεξεργαστή που εκτελεί εργασίες. Ο διαχωρισμός αυτός έγινε όπως φαίνεται πιο κάτω.

```
@Override
    public void startNode() {
        try {
            if(getNodeId().toString().equals("[194.199.68.166,51880]"))
                initialize_repository();
            else
                initialize_client();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

Στην μέθοδο startNode(), όπως αναφέρθηκε και στο υποκεφάλαιο 2.3.3, όπου γίνεται η αρχικοποίηση του κόμβου, με έναν έλεγχο καθορίζουμε τον κόμβο με αναγνωριστικό [194.199.68.166,51880] (όπου 194.199.68.166 το IP Address του κόμβου και 51880 το port αφού μιλάμε για κόμβους πραγματικού συστήματος) ως το Repository και τους υπόλοιπους ως Clients (κόμβοι δηλαδή που έχουν τον ρόλο των πελατών, των επεξεργαστών που εκτελούν εργασίες).

Στις μεθόδους initialize_repository() και initialize_client() γίνεται προγραμματισμός των εργασιών του κάθε κόμβου για να ξεκινήσουν ομαλά την λειτουργία τους. Οι μέθοδοι αυτοί παρουσιάζονται πιο κάτω.

```
public void initialize_repository() throws IOException {
    MyRepository MyTask1 = new MyRepository();
    tm.registerTask(MyTask1, tm.getSystemTaskGroup());
}
```

Στην πιο πάνω μέθοδο, δημιουργείται μία διεργασία MyTask1 η οποία είναι τύπου MyRepository και αναπαριστά το Repository. Για την κλάση MyRepository θα

μιλήσουμε εκτενέστερα στην συνέχεια του υποκεφαλαίου αυτού. Έπειτα, η διεργασία αυτή καταχωρείται για άμεση εκτέλεση μέσω της εντολής `tm.registerTask(MyTask1, tm.getSystemTaskGroup())` την οποία και αναλύσαμε στο υποκεφάλαιο 2.3.1.

```
public void initialize_client() throws IOException {  
    MyClients MyTask2 = new MyClients();  
    rand = new Random();  
    timeDelay = rand.nextInt(1000) + 1000;  
    tm.registerTask(MyTask2, tm.getSystemTaskGroup(), timeDelay);  
}
```

Στην πιο πάνω μέθοδο, δημιουργείται μία διεργασία `MyTask2` η οποία είναι τύπου `MyClients` και αναπαριστά έναν κόμβο/πελάτη του συστήματος. Για την κλάση `MyClients` θα μιλήσουμε εκτενέστερα στην συνέχεια του υποκεφαλαίου αυτού. Η διαφορά με την προηγούμενη μέθοδο είναι ότι τώρα προσθέτουμε μία καθυστέρηση στις διεργασίες αυτές. Αυτό το επιτυγχάνουμε καταχωρώντας την διεργασία για αναβεβλημένη εκτέλεση. Γίνεται μέσω της εντολής `tm.registerTask(MyTask2, tm.getSystemTaskGroup(), timeDelay)` την οποία και αναλύσαμε στο υποκεφάλαιο 2.3.1. Η διεργασία δηλαδή θα εκτελεστεί σε τόσο χρόνο από την καταχώρησή της όσο είναι το `timeDelay` (σε milliseconds) το οποίο παράγεται τυχαία την στιγμή της εκτέλεσης. Ο λόγος που γίνεται αυτό είναι για να έχουμε την έννοια του ασυγχρονισμού, δηλαδή οι κόμβοι/πελάτες μπορούν να εκκινήσουν σε διαφορετική χρονική στιγμή.

Κλάση `MyRepository`

Το επόμενο κύριο μέρος του αλγορίθμου είναι η «ενεργή» μέθοδος. Μια εργασία αποτελεί το «ενεργό» μέρος της εφαρμογής YALPS. Πρόκειται ουσιαστικά για ένα αντικείμενο που υλοποιεί τη διεπαφή `Task`. Για την ολοκλήρωση της υλοποίησης της διεπαφής αυτής πρέπει να προστεθεί κώδικας στη “ενεργή” μέθοδο `run()` του αντικειμένου ο οποίος θα περιγράφει τη λειτουργία της εργασίας. Και σε αυτό το μέρος ήταν αναγκαία η διαφορετική υλοποίηση για το `Repository`.

Πιο κάτω φαίνονται τα κύρια σημεία του κώδικα που υλοποιεί την λειτουργία του Repository.

```
public class MyRepository implements Task {
    . . .
    @Override
    public void run() {
        // Create randomly new tasks and add them to the list
        Pending = Random_Tasks(Pending, 1000, 1200);
        . . .
        // If am not done with the rounds
        if (counter < rounds) {
            // run repository again
            tm.registerTask(this, tm.getSystemTaskGroup(), 15000);
        } else {
            // If I completed the rounds
            end_time = System.currentTimeMillis() - start_time;
            System.out.println("TIME FOR NODE " + getNodeId() + "
                : "+ end_time);
        }
    }
}
```

Η κλάση MyRepository υλοποιεί τη διεπαφή Task και προσομοιώνει την λειτουργία του Repository. Το «ενεργό» μέρος του Repository είναι υπεύθυνο για την εισαγωγή των εργασιών και την ανανέωση της λίστας Pending. Αυτό επαναλαμβάνεται σε τακτά χρονικά διαστήματα και μέχρι να τελειώσουν οι κύκλοι, αριθμός τον οποίο εμείς αναθέτουμε.

Κλάση MyClients

Όπως, προαναφέραμε, λόγω της διαφορετικής λειτουργίας των υπολοίπων κόμβων/επεξεργαστών, χρειάστηκε να δημιουργηθεί μία άλλη κλάση, η κλάση

MyClients που να προσομοιώνει αυτή την λειτουργία. Πιο κάτω φαίνονται τα κύρια σημεία του κώδικα που υλοποιεί την λειτουργία των επεξεργαστών (Clients).

```
public class MyClients implements Task {

    MyClients() {
    };

    @Override
    public void run() {
        . . .
        // create a Get message and send it to the Repository
        Get_Message m = new Get_Message(id);
        for (E_NodeID neighbor : DestinationList) {
            if (neighbor.getNodeId().toString()
                .equals("[194.199.68.166,51880]")) {
                System.out.println("Node " + getNodeId()
                    + " sending message Get to Repository");
                MyTCPBasedMsgSenderTask myTask1 = new
                    MyTCPBasedMsgSenderTask(m, neighbor, cl);
                tm.registerTask(myTask1, tm.getSystemTaskGroup(), 0);
            }
        }
    }
}
```

Η κλάση MyClients υλοποιεί τη διεπαφή Task. Το «ενεργό» μέρος των κόμβων Clients είναι υπεύθυνο για την δημιουργία και την αποστολή μηνύματος get στο Repository ούτως ώστε να λάβει την λίστα Pending. Αυτό θα επαναληφθεί μόλις ο Client παραλάβει την λίστα Pending , εκτελέσει την εργασία που επέλεξε και ενημερώσει το Repository.

Μέθοδος receive()

Το τελευταίο κύριο μέρος του αλγορίθμου είναι η «παθητική» μέθοδος receive(). Η μέθοδος receive() ορίζεται στη διεπαφή Receiver και μέσω αυτής οι κόμβοι της

εφαρμογής παραλαμβάνουν όλα τα μηνύματα που προέρχονται από άλλους κόμβους και προορίζονται γι' αυτούς. Ενεργοποιείται αμέσως μόλις παραληφθεί ένα μήνυμα με κάποια επικεφαλίδα από τον κόμβο – αποστολέα και επιστρέφει true αν το μήνυμα έχει παραληφθεί με επιτυχία και false στην αντίθετη περίπτωση.

Η «παθητική» μέθοδος receive() ορίζεται ως εξής:

```
public boolean receive(short header, Object msg, NodeID sender) {  
}
```

Ένας κόμβος p εισέρχεται αυτόματα στην παθητική μέθοδο όταν παραλάβει κάποιο μήνυμα. Βάσει του αναγνωριστικού του μπορούμε να διακρίνουμε αν ο κόμβος είναι το Repository ή είναι κάποιος από τους Clients.

Πιο κάτω φαίνεται το κομμάτι κώδικα που εκτελεί το Repository όταν παραλάβει ένα μήνυμα.

```
if (getNodeId().toString().equals("[194.199.68.166,51880]")) {  
    // If i received a Get message  
    if (msg instanceof Get_Message) {  
        // Create message with the Pending list and send it to the node  
        List_Message m1 = new List_Message(Pending, Duration);  
        MyTCPBasedMsgSenderTask myTask1 = new MyTCPBasedMsgSenderTask(  
            m1, sender, cl);  
        tm.registerTask(myTask1, tm.getSystemTaskGroup(), 0);  
    }  
    // If i received an Inform message  
    else if (msg instanceof Inform_Message) {  
        Inform_Message m1 = (Inform_Message) msg;  
        // Remove the task that Node performed  
        for (int counter = 0; counter < Pending.size(); counter++) {  
            if (Pending.get(counter) == m1.task) {  
                // track the task in the list  
                Pending.remove(counter); // delete the task  
                Duration.remove(counter); // and its duration  
            }  
        }  
    }  
}
```

```

    }
}

```

Όπως φαίνεται και πιο πάνω, γίνεται έλεγχος του μηνύματος που παρέλαβε το Repository για να καθοριστεί ο τύπος του. Αν το μήνυμα είναι τύπου Get_Message τότε δημιουργείται ένα μήνυμα τύπου List_Message και στέλνεται στον κόμβο ο οποίος έστειλε εξ αρχής το μήνυμα get. Αν το μήνυμα είναι τύπου Inform_Message, δηλαδή είναι ενημέρωση ότι κάποια εργασία ολοκληρώθηκε, τότε γίνεται εντοπισμός της εργασίας μέσα στην λίστα Pending και έπειτα η συγκεκριμένη εργασία διαγράφεται.

Πιο κάτω φαίνεται το κομμάτι κώδικα που εκτελεί ένας Client όταν παραλάβει ένα μήνυμα.

```

if (!(getNodeId().toString().equals("[130.37.193.141,51881]"))) {
    if (counter2 < rounds) {
        // if i'm not done with the rounds
        List_Message l = (List_Message) msg;
        // if there are tasks in the list
        if (l.pending.size() >= 1) {
            // choose which task to perform
            rank = (((int) (id * b * n)) % l.pending.size());
            System.out.println("Node " + getNodeId()+ " will perform
            task "+ l.pending.get(rank)+ " with duration: "+
            l.duration.get(rank));
            // Delay until i perform the task
            try {
                Thread.currentThread().sleep((int)
                (l.duration.get(rank) / speedup));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            // Create an Inform msg and send it to the Repository
            Inform_Message inform = new
            Inform_Message(l.pending.get(rank));
            MyTCPBasedMsgSenderTask myTask1 = new
            MyTCPBasedMsgSenderTask(inform, sender, cl);

```

```

        tm.registerTask(myTask1,tm.getSystemTaskGroup(), 0);
    }else {
        // The pending list is empty
        System.out.println("The Pending list is empty!!");
    }
    // Run the client again
    MyClients MyTask2 = new MyClients();
    tm.registerTask(MyTask2, tm.getSystemTaskGroup(), 0);
    counter2++; // increase the number of rounds
}
if (counter2 == rounds) {
    // I finished the rounds
    end_time = System.currentTimeMillis() - start_time;
    System.out.println("NODE " + getId() + "finished");
    System.out.println("TIME FOR NODE " + getId() + " : "+
        end_time);
}
}

```

Ένας Client, αφού παραλάβει ένα μήνυμα, το οποίο αναμφίβολα είναι τύπου List_Message, ελέγχει αν η λίστα Pending η οποία περιέχεται στο μήνυμα είναι κενή. Αν η λίστα είναι κενή τότε εμφανίζεται κατάλληλο μήνυμα και προχωρά στον επόμενο κύκλο. Αν δεν είναι, τότε επιλέγει με βάση την πολιτική του αλγορίθμου ποια εργασία θα εκτελέσει.

Η πολιτική αυτή εκφράζεται ως εξής:

$$\text{rank} = (((\text{int}) (\text{id} * \text{b} * \text{n})) \% \text{l.pending.size()});$$

Έπειτα, δημιουργείται μία καθυστέρηση, ίση με το κόστος της εργασίας που επέλεξε να εκτελέσει διαιρούμενο με το speedup, ούτως ώστε να αναπαρασταθεί εικονικά η εκτέλεση της εργασίας. Αυτό επιτυγχάνεται με την εξής εντολή:

```
Thread.currentThread().sleep((int)(l.duration.get(rank) / speedup));
```

Στη συνέχεια, δημιουργείται ένα μήνυμα τύπου `Inform_Message`, το οποίο περιέχει το αναγνωριστικό της εργασίας που μόλις εκτέλεσε ο `Client`, και αποστέλλεται στο `Repository`. Αν δεν τελείωσε ο επιθυμητός αριθμός κύκλων, τότε ξανακαλείται η διεργασία των `Clients` ούτως ώστε να δημιουργηθεί ένας νέος κύκλος, δηλαδή να ξανασταλθεί μήνυμα `get` προς το `Repository`.

Κεφάλαιο 5

Πειραματική Αξιολόγηση Αλγορίθμου

5.1 Μεθοδολογία	47
5.1.1 Πλατφόρμα Πειραματισμού	48
5.1.2 Μετρικές Αξιολόγησης Αλγορίθμου	48
5.1.3 Σενάρια	51
5.2 Ανάλυση Αποτελεσμάτων	55
5.2.1 Αλγόριθμος (n,β)-LIS χωρίς σφάλματα	55
5.2.2 Αλγόριθμος (n,β)-LIS με σφάλματα	70
5.2.3 Σύγκριση Αλγορίθμου (n,β)-LIS με και χωρίς σφάλματα	79
5.3 Συμπεράσματα	81

5.1 Μεθοδολογία

Μετά τη φάση της υλοποίησης του αλγορίθμου ακολουθεί η πειραματική αξιολόγησή του. Απώτερος στόχος της αξιολόγησης αυτής είναι να διαφανεί σε ποιο βαθμό ο αλγόριθμος είναι αποτελεσματικός, αποδοτικός και ανεκτικός σε σφάλματα όχι μόνο στη θεωρία αλλά και στην πράξη.

Στο κεφάλαιο της Πειραματικής Αξιολόγησης Αλγορίθμου παρουσιάζεται η όλη διαδικασία που ακολουθήθηκε για την εξαγωγή των αποτελεσμάτων. Γίνετε αναλυτική περιγραφή όλων των διαδικαστικών βημάτων από την επιλογή των μετρικών αξιολόγησης και την δημιουργία των σεναρίων εκτέλεσης μέχρι και την παρουσίαση των αποτελεσμάτων.

5.1.1 Πλατφόρμα Πειραματισμού

Η συγγραφή του κώδικα πραγματοποιήθηκε στην Java 1.6.0_30 και χρησιμοποιήθηκε η Βιβλιοθήκη YALPS με έκδοση 0.3 Alpha σε λειτουργικό σύστημα Windows 7 x32.

Η πειραματική διαδικασία εκτελέστηκε σε πραγματικό σύστημα, στο διαδικτυακό σύστημα PlanetLab [9] όπου χρησιμοποιήθηκαν μηχανές διαφόρων ταχυτήτων και μνήμης RAM.

5.1.2 Μετρικές Αξιολόγησης Αλγορίθμου

Για μια πιο σωστή και μεθοδική πειραματική ανάλυση πρώτα καθορίζονται με ακρίβεια και σαφήνεια οι μετρικές αξιολόγησης που θα χρησιμοποιηθούν στη μελέτη των αποτελεσμάτων της εφαρμογής του αλγορίθμου.

Για την πειραματική μελέτη του αλγορίθμου χρησιμοποιήθηκαν οι εξής μετρικές αξιολόγησης:

- Χρόνος Διεκπεραίωσης: Ο συνολικός χρόνος σε χιλιοστά δευτερόλεπτων (milliseconds) που χρειάστηκε για την εκτέλεση του κάθε σεναρίου.
- Αριθμός μηνυμάτων: Ο συνολικός αριθμός μηνυμάτων που κινήθηκε στο δίκτυο από και προς τις διεργασίες.
- Pending Tasks: Ο συνολικός αριθμός εργασιών εν αναμονή στο Repository σε μία χρονική στιγμή t όπως ορίζεται στο υποκεφάλαιο 3.7.
- Pending Cost: Το συνολικό κόστος των εργασιών εν αναμονή στο Repository σε μία χρονική στιγμή t όπως ορίζεται στο υποκεφάλαιο 3.7.

Χρόνος Διεκπεραίωσης

Η πρώτη μετρική, αφορά το χρόνο ολοκλήρωσης της εκτέλεσης του αλγορίθμου. Συγκεκριμένα, μετριέται ο χρόνος ολοκλήρωσης της κάθε διεργασίας ξεχωριστά και από τους χρόνους αυτούς λαμβάνεται ο μέγιστος ως χρόνος ολοκλήρωσης της εκτέλεσης του αλγορίθμου. Η καταμέτρηση γίνεται για κάθε διεργασία ξεχωριστά, από την αρχή της εκτέλεσής της μέχρι το σημείο όπου ολοκληρώνεται και ο τελευταίος κύκλος της διεργασίας.

Ένας απλός ψευδοκώδικας για την υλοποίηση του μετρητή του χρόνου ολοκλήρωσης για μια διεργασία p είναι ο εξής:

Για κάθε διεργασία υπάρχουν δύο μεταβλητές `start_time` και `end_time`. Τότε με την εκκίνηση της διεργασίας:

```
start_time = System.currentTimeMillis();
```

Με το τέλος της διεργασίας τότε:

```
end_time = System.currentTimeMillis() - start_time;
```

Αφού καταμετρηθεί ο χρόνος ολοκλήρωσης της κάθε διεργασίας, στη συνέχεια καταγράφεται ο μέγιστος από αυτούς χρόνους ως εξής:

$$\text{time_complexity} = \max_{i \in n}(\text{elapsed_time}_i)$$

Αριθμός μηνυμάτων

Η δεύτερη μετρική αφορά το συνολικό αριθμό μηνυμάτων που αποστέλλονται από τις διεργασίες καθ' όλη τη διάρκεια της εκτέλεσης. Ένας απλός ψευδοκώδικας για την υλοποίηση του μετρητή μηνυμάτων για μια διεργασία p είναι ο εξής:

Για κάθε διεργασία έστω υπάρχει μια ακέραια μεταβλητή `msg_counter`. Τότε

```
msg_counter=0;
for each msg sent{
    msg_counter++;
}
```

Αφού καταμετρηθεί ο αριθμός των μηνυμάτων που αποστέλλει η κάθε διεργασία, στη συνέχεια καταγράφεται το άθροισμα όλων αυτών των μηνυμάτων ως εξής:

$$\text{msg_complexity} = \sum_{i \in n} \text{msg_counter}_i$$

Pending Tasks

Η τρίτη μετρική αφορά τον συνολικό αριθμό των εν αναμονή εργασιών στο Repository. Συγκεκριμένα, για τον σκοπό καταμέτρησης του αριθμού των εργασιών που εκκρεμούν, δημιουργήθηκε μία νέα διεργασία, η οποία ανά τακτά χρονικά διαστήματα λάμβανε από το Repository την λίστα Pending και αποθήκευε τον αριθμό των εργασιών που εκκρεμούν. Αυτή η διεργασία εκτελείτο κάθε 14000 milliseconds. Η επιλογή αυτού του διαστήματος προέκυψε λόγω του ότι στο Repository εργασίες εισάγονται κάθε 15000 milliseconds, άρα ήταν πιο ορθό να γίνεται η μέτρηση των εργασιών εν αναμονή πριν να εισαχθούν οι καινούριες εργασίες στο σύστημα αφού διαφορετικά θα παρατηρούσαμε πολύ υψηλούς αριθμούς. Έπειτα, με το τέλος της εκτέλεσης του αλγορίθμου, ως αριθμός των εργασιών που εκκρεμούν υπολογίζεται ο μέσος όρος όλων των αριθμών που πήραμε.

Pending Cost

Τέλος, η τέταρτη μετρική αφορά το συνολικό κόστος των εν αναμονή εργασιών στο Repository. Συγκεκριμένα, για τον σκοπό καταμέτρησης του κόστους των εργασιών που εκκρεμούν, η ίδια διεργασία η οποία καταμετρούσε το συνολικό αριθμό των εργασιών που εκκρεμούν όπως περιγράφηκε προηγουμένως, καταμετρεί και το κόστος όλων των εργασιών που εκκρεμούν. Έπειτα, με το τέλος της εκτέλεσης του αλγορίθμου, ως συνολικό κόστος των εργασιών που εκκρεμούν υπολογίζεται ο μέσος όρος όλων των αριθμών που πήραμε.

5.1.3 Σενάρια

Κατά την πειραματική αξιολόγηση του αλγορίθμου, στόχος ήταν να δοκιμαστεί ο αλγόριθμος σε πραγματικό περιβάλλον με τη χρήση «ρεαλιστικών» σεναρίων, παρέχοντας στο τέλος αποτελέσματα τα οποία να έχουν αντίκρισμα στην πραγματικότητα.

Για το λόγο αυτό έχουν αποφευχθεί σενάρια στα οποία ο αλγόριθμος θα υπολειτουργούσε τον περισσότερο χρόνο ή θα ήταν αδύνατο λόγω υπερφόρτωσης του συστήματος να αντεπεξέλθει.

Για την εκτέλεση του αλγορίθμου και τη δημιουργία των σεναρίων ορίστηκαν οι εξής παράμετροι:

- i. Αριθμός των κόμβων
- ii. Επιτάχυνση - Speedup
- iii. Πιθανότητα ο κόμβος να παραμείνει ενεργός
- iv. Πιθανότητα να επανέλθει πίσω ένας μη ενεργός κόμβος

Επίσης σημαντικός παράγοντας είναι ο αριθμός των νέων εργασιών που προστίθενται στο Repository ανά τακτά χρονικά διαστήματα. Για την αποφυγή σεναρίων που θα υπερφόρτωναν το σύστημα όπως προαναφέραμε, επιλέχθηκε αυτός ο αριθμός των νέων εργασιών που προστίθενται στο σύστημα να είναι ένας τυχαίος αριθμός στο εύρος [1000..1200].

Επιπρόσθετα, πρέπει να σημειωθεί ότι για την δίκαιη παραγωγή των αποτελεσμάτων, δεδομένου ότι οι επεξεργαστές στο διαδικτυακό σύστημα PlanetLab έχουν διαφορετικές ταχύτητες, δεν τέθηκε χρονικό περιθώριο στην εκτέλεσή των διεργασιών αλλά αριθμός κύκλων. Μετά από αρκετές προκαταρκτικές εκτελέσεις επιλέχθηκε η κάθε διεργασία να εκτελέσει είκοσι (20) κύκλους. Η επιλογή αυτού του αριθμού προέκυψε από το γεγονός ότι για πολύ μεγάλο αριθμό κύκλων το σύστημα υπερφορτωνόταν και συγκεκριμένα το Repository αφού αποθηκεύει μεγάλο αριθμό εργασιών και σε κάθε κύκλο προστίθενται ακόμα περισσότερες.

Πιο κάτω, (Πίνακας 5.1) φαίνονται τα σενάρια που εκτελέστηκαν για τον αλγόριθμο (n,β)-LIS χωρίς σφάλματα. Το κάθε ένα από τα τέσσερα σενάρια εκτελέστηκε για 32,50,64,80,100 και 128 κόμβους.

Σενάριο	Επιτάχυνση - Speedup	cmin	cmax
1	s=1	12000	16000
2	s=cmax/cmin	12000	16000
3	$s=1 + \sqrt{(1- cmin/cmax)}$	10000	17000
4	$s=1 + \sqrt{(1- cmin/cmax)}$	10000	21000

Πίνακας 5.1: Σενάρια εκτέλεσης αλγορίθμου (n,β)-LIS

Να θυμηθούμε ότι οι αριθμοί cmin και cmax δηλώνουν το μικρότερο και το μεγαλύτερο, αντίστοιχα, κόστος που μπορεί να έχουν οι εργασίες.

Αξιίζει να σημειωθεί ότι οι τιμές για την επιτάχυνση και τα c_{min} , c_{max} προέκυψαν από την θεωρητική ανάλυση που προηγήθηκε όπως αναφέραμε στο [5] (υποκεφάλαιο 2.2).

Στον επόμενο πίνακα (Πίνακας 5.2) φαίνονται τα σενάρια που εκτελέστηκαν για τον αλγόριθμο (n,β)-LIS με την παρουσία σφαλμάτων κατάρρευσης και επανεκκίνησης.

Πρέπει να αναφέρουμε ότι πιθανότητα κατάρρευσης υπάρχει μόνο για τους κόμβους Clients και όχι για το Repository αφού αυτό θα είχε ως αποτέλεσμα την κατάρρευση του συστήματος (single point of failure).

Σενάριο	Επιτάχυνση - Speedup	c_{min}	c_{max}	Πιθανότητα ο κόμβος να παραμείνει ενεργός	Πιθανότητα να επανέλθει πίσω ένας μη ενεργός κόμβος
1	$s=1$	12000	16000	50%	50%
2	$s=1$	12000	16000	70%	50%
3	$s=1$	12000	16000	90%	50%
4	$s=1$	12000	16000	95%	50%
5	$s=c_{max}/c_{min}$	12000	16000	50%	50%
6	$s=c_{max}/c_{min}$	12000	16000	70%	50%
7	$s=c_{max}/c_{min}$	12000	16000	90%	50%
8	$s=c_{max}/c_{min}$	12000	16000	95%	50%
9	$s=1+\sqrt{(1-c_{min}/c_{max})}$	10000	17000	50%	50%
10	$s=1+\sqrt{(1-c_{min}/c_{max})}$	10000	17000	70%	50%
11	$s=1+\sqrt{(1-c_{min}/c_{max})}$	10000	17000	90%	50%
12	$s=1+\sqrt{(1-c_{min}/c_{max})}$	10000	17000	95%	50%
13	$s=1+\sqrt{(1-c_{min}/c_{max})}$	10000	21000	50%	50%
14	$s=1+\sqrt{(1-c_{min}/c_{max})}$	10000	21000	70%	50%
15	$s=1+\sqrt{(1-c_{min}/c_{max})}$	10000	21000	90%	50%
16	$s=1+\sqrt{(1-c_{min}/c_{max})}$	10000	21000	95%	50%

Πίνακας 5.2: Σενάρια εκτέλεσης αλγορίθμου (n,β)-LIS με σφάλματα

Για την εκτέλεση αυτών των σεναρίων επιλέχθηκε ο αριθμός των κόμβων να είναι ίσος με 64 λόγω του ότι είναι ένας ικανοποιητικός αριθμός κόμβων, ούτε αρκετά μικρός ούτε αρκετά μεγάλος και σε προκαταρκτικές εκτελέσεις παρουσίασε τα πιο «λογικά» αποτελέσματα.

Στο επόμενο υποκεφάλαιο παρουσιάζονται τα αποτελέσματα των πιο πάνω σεναρίων και γίνεται η ανάλυση των αποτελεσμάτων αυτών.

Ο μηχανισμός που χρησιμοποιήθηκε για τη δυναμική εισαγωγή εργασιών στο σύστημα είναι σχετικά απλός καθώς αποφασίζεται τυχαία ένας x αριθμός νέων εργασιών στο πεδίο [1000,1200] και ακολούθως εισάγονται στο σύστημα. Πιο κάτω φαίνεται το τμήμα κώδικα που εισάγει τις εργασίες στο Repository.

```
int number; // number of new tasks to add
Random random = new Random();
// get the range, casting to long to avoid overflow problems
long range = (long) max - (long) min + 1;
// compute a fraction of the range,  $0 \leq \text{frac} < \text{range}$ 
long fraction = (long) (range * random.nextDouble());
number = (int) (fraction + min);
// append these tasks to the list
for (int i = 0; i < number; i++) {
    L.add(task_id);
    // increase the task_id to get a unique id task every time
    task_id++;
}
```

Παρόμοιος μηχανισμός χρησιμοποιείται και για την παραγωγή των κοστών των εργασιών καθώς αποφασίζεται και πάλι τυχαία ένας x αριθμός που αναπαριστά το κόστος στο πεδίο [cmin,cmax].

Στην ίδια λογική βασίζεται και ο μηχανισμός ο οποίος αποφασίζει ποια διεργασία θα καταρρεύσει ή θα επανεκκινήσει εάν έχει ήδη καταρρεύσει.

Πιο κάτω φαίνεται το τμήμα κώδικα από το μηχανισμό κατάρρευσης και επανεκκίνησης διεργασιών του αλγόριθμου.

```
if (nodeIsDown == false && NodeGoesDown(StayUpChance) == true) {
    nodeIsDown = true;
    . . .
}else if (nodeIsDown == true && NodeRestarts(RestartChance) == true) {
    nodeIsDown = false;
    . . .
}
```

5.2 Ανάλυση Αποτελεσμάτων

Στο παρόν υποκεφάλαιο παρουσιάζονται τα αποτελέσματα της πειραματικής εκτέλεσης του αλγορίθμου και έπειτα σχολιάζονται και αναλύονται.

5.2.1 Αλγόριθμος (n,β)-LIS χωρίς σφάλματα

Αρχικά ο αλγόριθμος (n,β)-LIS χωρίς την παρουσία σφαλμάτων κατάρρευσης και επανεκκίνησης εκτελέστηκε συνολικά εικοσιτέσσερις (24) φορές, με διαφορές στην επιτάχυνση s , στα $cmin, cmax$ και στον αριθμό των κόμβων όπως περιγράφονται και στο υποκεφάλαιο 5.1.3. Συγκεκριμένα, το κάθε ένα από τα τέσσερα (4) σενάρια εκτελέστηκε έξι (6) φορές, για αριθμό κόμβων 32,50,64,80,100 και 128.

Σενάριο 1

$s=1$		$cmin=12000$		$cmax=16000$	
Number of Nodes	Total Execution Time	Number of Msgs	Pending Tasks	Pending Cost	
32	285291	1920	12379	1,73E+15	
50	373283	3000	11873	1,66E+15	
64	485207	3840	4928	7,74E+14	
80	544760	4800	4330	6,50E+14	
100	577427	6000	4193	5,84E+14	
128	1434452	7680	3636	5,09E+14	

Πίνακας 5.3: Αποτελέσματα εκτελέσεων σεναρίου 1 αλγορίθμου (n,β)-LIS χωρίς σφάλματα

Σενάριο 2

$s=c_{\max}/c_{\min}$		$c_{\min}=12000$		$c_{\max}=16000$
Number of Nodes	Total Execution Time	Number of Msgs	Pending Tasks	Pending Cost
32	285252	1920	5034	6,39E+14
50	356990	3000	4562	5,94E+14
64	329037	3840	4242	5,71E+14
80	471166	4800	4086	4,27E+14
100	553033	6000	3050	4,12E+14
128	550421	7680	2941	7,05E+13

Πίνακας 5.4: Αποτελέσματα εκτελέσεων σεναρίου 2 αλγορίθμου (n,β)-LIS χωρίς σφάλματα

Σενάριο 3

$s=1+\sqrt{1-c_{\min}/c_{\max}}$		$c_{\min}=10000$		$c_{\max}=17000$
Number of Nodes	Total Execution Time	Number of Msgs	Pending Tasks	Pending Cost
32	164289	1920	4406	6,90E+14
50	199892	3000	4463	5,95E+14
64	285222	3840	3992	5,21E+14
80	285266	4800	3858	5,03E+14
100	365307	6000	3720	3,98E+14
128	597562	7680	2839	5,59E+13

Πίνακας 5.5: Αποτελέσματα εκτελέσεων σεναρίου 3 αλγορίθμου (n,β)-LIS χωρίς σφάλματα

Σενάριο 4

$s=1+\sqrt{1-c_{\min}/c_{\max}}$		$c_{\min}=10000$		$c_{\max}=21000$
Number of Nodes	Total Execution Time	Number of Msgs	Pending Tasks	Pending Cost
32	158076	1920	3888	6,00E+14
50	184907	3000	3827	5,96E+14
64	255210	3840	3361	5,09E+14
80	275478	4800	3284	4,07E+14
100	307347	6000	3172	4,29E+13
128	364796	7680	3005	5,19E+11

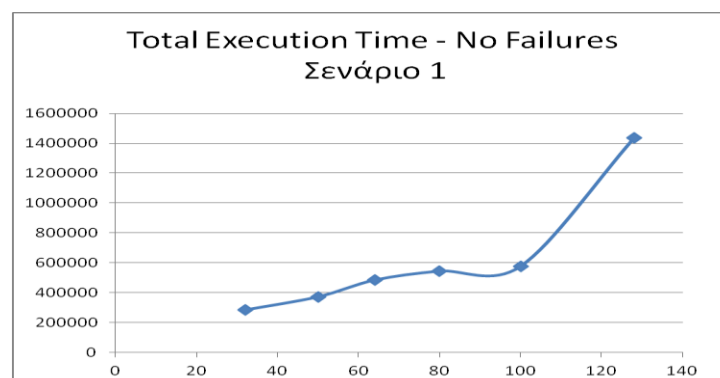
Πίνακας 5.6: Αποτελέσματα εκτελέσεων σεναρίου 4 αλγορίθμου (n,β)-LIS χωρίς σφάλματα

Στους πιο πάνω πίνακες (Πίνακες 5.3, 5.4, 5.5 και 5.6) εμφανίζονται τα αποτελέσματα των σεναρίων όπως έχουν εξαχθεί από τα αρχεία ελέγχου. Εύκολα γίνεται αντιληπτό ότι για όλα τα σενάρια όσο μεγαλώνει ο αριθμός των κόμβων στο σύστημα έχουμε αύξηση του χρόνου διεκπεραίωσης και του αριθμού των μηνυμάτων και μείωση των εργασιών που εκκρεμούν και του κόστους τους.

Χρόνος Διεκπεραίωσης

Αρχικά θα αναλύσουμε τα αποτελέσματα που πήραμε για την πρώτη μετρική, τον Χρόνο Διεκπεραίωσης. Οι γραφικές παραστάσεις δεν αναμένουμε να έχουν κάποια σταθερή μορφή. Αυτό προκύπτει από το γεγονός ότι οι εργασίες που εκτελούνται δεν έχουν τους ίδιους χρόνους εκτέλεσης και οι επεξεργαστές έχουν διαφορετικές ταχύτητες και επομένως ο κάθε επεξεργαστής ολοκληρώνει τις εκτελέσεις των εργασιών του σε διαφορετικό χρόνο από ότι οι υπόλοιποι. Πρέπει επίσης, να ληφθεί υπόψη ότι για τον χρόνο ολοκλήρωσης του κάθε σεναρίου σημαντικός παράγοντας είναι ο χρόνος που χρειάζεται για να ανταποκριθεί το Repository και να λάβει ο κάθε Client την λίστα Pending ούτως ώστε να επιλέξει και να ξεκινήσει να εκτελεί την εργασία του.

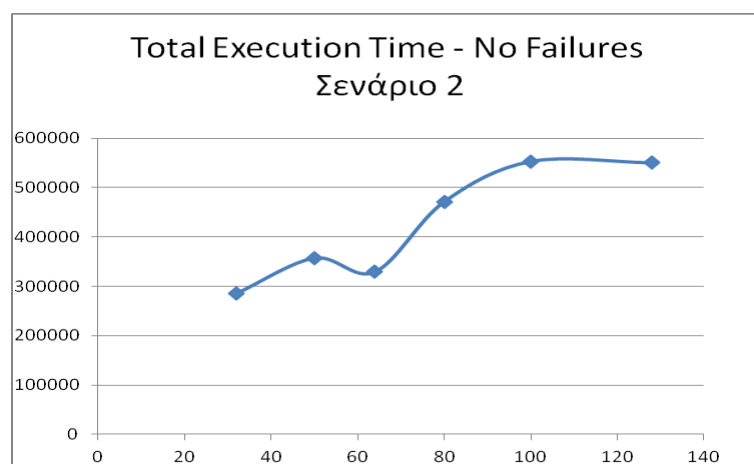
Θα μελετήσουμε το κάθε ένα από τα τέσσερα σενάρια ξεχωριστά και έπειτα όλα μαζί. Στις πιο κάτω γραφικές παραστάσεις φαίνονται οι χρόνοι διεκπεραίωσης για τα σενάρια του αλγορίθμου χωρίς σφάλματα. Στον άξονα χ φαίνονται οι αριθμοί των κόμβων οι οποίοι όπως προαναφέραμε είναι 32,50,64,80,100 και 128 ενώ στον άξονα ψ φαίνονται οι χρόνοι διεκπεραίωσης σε milliseconds.



Σχήμα 5.1: Συνολικός Χρόνος Διεκπεραίωσης για Σενάριο 1 χωρίς σφάλματα

Στην πιο πάνω γραφική παράσταση (Σχήμα 5.1) φαίνονται οι χρόνοι διεκπεραίωσης για το Σενάριο 1 του αλγορίθμου χωρίς σφάλματα όπου δεν υπάρχει επιτάχυνση ($s=1$ και $cmin=12000$, $cmax=16000$). Παρατηρούμε ότι όσο αυξάνεται ο αριθμός των κόμβων που βρίσκονται στο σύστημα, αυξάνεται και ο χρόνος διεκπεραίωσης του σεναρίου. Τα αποτελέσματα ήταν αναμενόμενα αφού όσο αυξάνονται οι κόμβοι, αυξάνονται και τα μηνύματα τα οποία διακινούνται στο δίκτυο, και συγκεκριμένα από και προς το Repository. Άρα το Repository χρειάζεται περισσότερο χρόνο να επεξεργαστεί και να ικανοποιήσει όλα τα αιτήματα και να ανταποκριθεί, με αποτέλεσμα ο κάθε κόμβος Client του συστήματος να περιμένει περισσότερο χρόνο μέχρι να λάβει τα απαραίτητα μηνύματα και να ολοκληρώσει τους κύκλους του.

Όπως αναμέναμε, η μορφή της γραφικής παράστασης δεν είναι σταθερή. Αρχικά, παρατηρείται μια λογαριθμική αύξηση του χρόνου όσο αυξάνονται οι κόμβοι, καθώς όμως μεγαλώνει πολύ ο αριθμός των κόμβων, και συγκεκριμένα για αριθμό κόμβων ίσο με 128, παρατηρείται μία απότομη αύξηση στον χρόνο διεκπεραίωσης. Αυτό το αποδίδουμε στο γεγονός ότι το Repository σε αυτή την περίπτωση λαμβάνει πολύ μεγάλο αριθμό μηνυμάτων από τους Clients, με λίγα λόγια υπερφορτώνεται και χρειάζεται πολύ περισσότερο για να επεξεργαστεί και να ανταποκριθεί στα αιτήματα των Clients με αποτέλεσμα αυτό να καθυστερεί τον συνολικό χρόνο εκτέλεσης του αλγορίθμου.

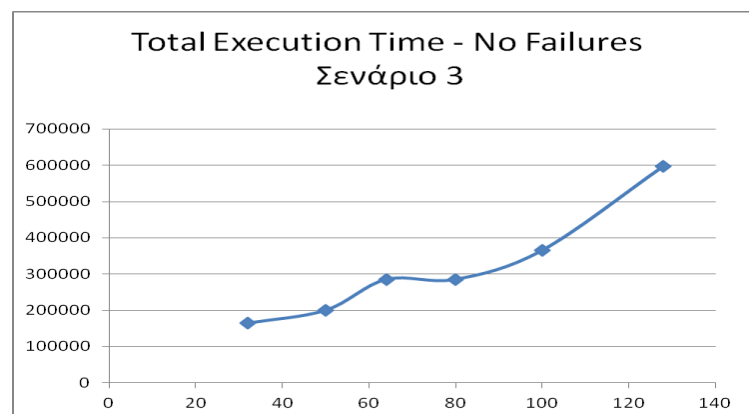


Σχήμα 5.2: Συνολικός Χρόνος Διεκπεραίωσης για Σενάριο 2 χωρίς σφάλματα

Στην πιο πάνω γραφική παράσταση (Σχήμα 5.2) φαίνονται οι χρόνοι διεκπεραίωσης για το Σενάριο 2 του αλγορίθμου χωρίς σφάλματα όπου η επιτάχυνση είναι ίση με $s=c_{max}/c_{min}$ και $c_{min}=12000$, $c_{max}=16000$. Παρατηρούμε και εδώ ότι όσο αυξάνεται ο αριθμός των κόμβων που βρίσκονται στο σύστημα, αυξάνεται και ο χρόνος διεκπεραίωσης του σεναρίου για τους ίδιους λόγους που προαναφέρθηκαν.

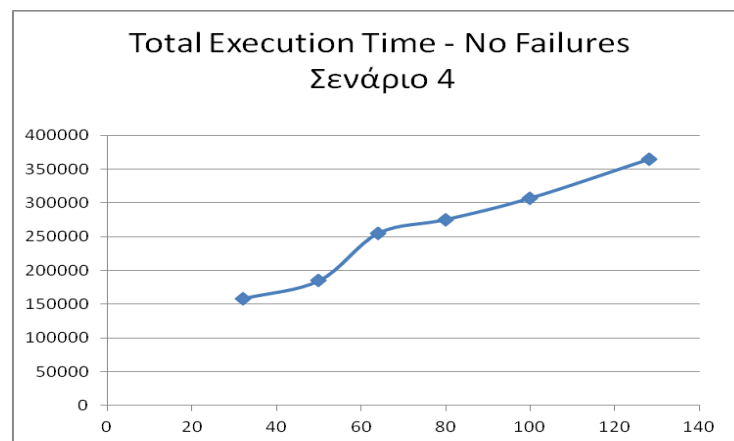
Σε αυτό το σενάριο όπως και στο προηγούμενο, παρατηρείται η αναμενόμενη μη σταθερή αύξηση της γραφικής παράστασης. Παρατηρείται μια λογαριθμική αύξηση του χρόνου όσο αυξάνονται οι κόμβοι, με εξαίρεση το τρίτο σημείο (όπου οι κόμβοι είναι 64) στο οποίο παρατηρείται μία απότομη μείωση στον χρόνο διεκπεραίωσης. Την απόκλιση αυτού του σημείου την αποδίδουμε στο γεγονός ότι σε αυτή την περίπτωση πιθανό να εκτελέστηκαν από τους επεξεργαστές περισσότερες εργασίες με κόστος κοντινό στο c_{min} (αφού όπως προαναφέραμε το κόστος της κάθε εργασίας είναι ένας τυχαίος αριθμός στο εύρος $[c_{min}, c_{max}]$), με αποτέλεσμα η εκτέλεση του συγκεκριμένου σεναρίου να διήρκεσε λιγότερο χρόνο.

Στην επόμενη γραφική παράσταση (Σχήμα 5.3) φαίνονται οι χρόνοι διεκπεραίωσης για το Σενάριο 3 του αλγορίθμου χωρίς σφάλματα όπου η επιτάχυνση είναι ίση με $s=1+\sqrt{1-c_{min}/c_{max}}$ και $c_{min}=10000$, $c_{max}=17000$. Παρατηρείται και εδώ ότι όσο αυξάνεται ο αριθμός των κόμβων που βρίσκονται στο σύστημα, αυξάνεται και ο χρόνος διεκπεραίωσης του σεναρίου για τους ίδιους λόγους που προαναφέρθηκαν.



Σχήμα 5.3: Συνολικός Χρόνος Διεκπεραίωσης για Σενάριο 3 χωρίς σφάλματα

Στο Σενάριο 3 η αύξηση του χρόνου που παρατηρείται είναι πιο ομαλή από τα προηγούμενα δύο σενάρια, δεν υπάρχουν μεγάλες διακυμάνσεις στις τιμές αλλά ωστόσο ο ρυθμός αύξησης παρατηρείται και εδώ να είναι λογαριθμικός.

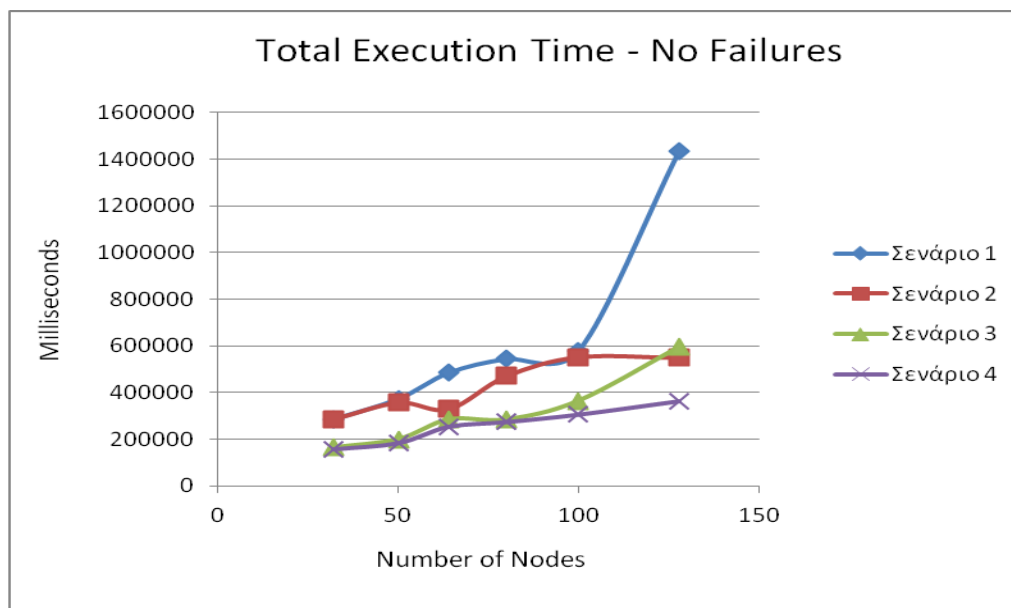


Σχήμα 5.4: Συνολικός Χρόνος Διεκπεραίωσης για Σενάριο 4 χωρίς σφάλματα

Στην πιο πάνω γραφική παράσταση (Σχήμα 5.4) φαίνονται οι χρόνοι διεκπεραίωσης για το Σενάριο 4 του αλγορίθμου χωρίς σφάλματα όπου η επιτάχυνση είναι ίση με $s=1+\sqrt{1-c_{min}/c_{max}}$ και $c_{min}=10000$, $c_{max}=21000$. Παρατηρείται και εδώ ότι όσο αυξάνεται ο αριθμός των κόμβων που βρίσκονται στο σύστημα, αυξάνεται και ο χρόνος διεκπεραίωσης του σεναρίου για τους ίδιους λόγους που προαναφέρθηκαν.

Στο Σενάριο 4 και πάλι η αύξηση του χρόνου που παρατηρείται είναι ομαλή χωρίς μεγάλες διακυμάνσεις στις τιμές και ο ρυθμός αύξησης είναι λογαριθμικός.

Στην επόμενη γραφική παράσταση (Σχήμα 5.5) φαίνονται οι χρόνοι διεκπεραίωσης και για τα σενάρια του αλγορίθμου χωρίς σφάλματα. Στον άξονα x φαίνονται οι αριθμοί των κόμβων οι οποίοι όπως προαναφέραμε είναι 32,50,64,80,100 και 128. Στον άξονα y φαίνονται οι χρόνοι διεκπεραίωσης σε milliseconds.

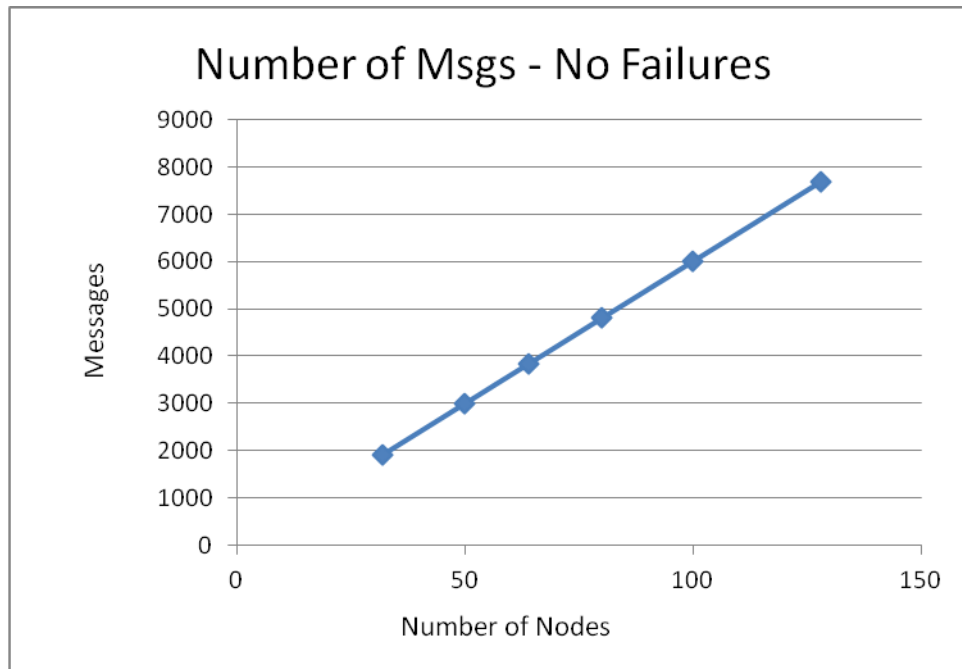


Σχήμα 5.5: Συνολικός Χρόνος Διεκπεραίωσης αλγορίθμου (n,β)-LIS χωρίς σφάλματα

Από αυτή την γραφική παράσταση, παρατηρείται ότι στην εκτέλεση του σεναρίου 1 όπου δεν υπάρχει επιτάχυνση ($s=1$), παρατηρείται ότι οι χρόνοι διεκπεραίωσης είναι μεγαλύτεροι από τα υπόλοιπα σενάρια και ο αλγόριθμος δεν είναι πολύ αποδοτικός. Αυτό συμβαίνει λόγω του ότι οι κόμβοι Clients χρειάζονται χρόνο όσο είναι το κόστος κάθε εργασίας για να εκτελέσουν την εργασία, σε αντίθεση με τα υπόλοιπα στα οποία το κόστος αυτό διαιρείται με την επιτάχυνση. Το σενάριο 4, το οποίο έχει την πιο μεγάλη επιτάχυνση, φαίνεται να είναι το πιο αποδοτικό και από τα τέσσερα σενάρια αφού έχει τους μικρότερους χρόνους διεκπεραίωσης για κάθε αριθμό κόμβων.

Συνολικός Αριθμός Μηνυμάτων

Συνεχίζοντας, θα αναλύσουμε τα αποτελέσματα που πήραμε για την δεύτερη μετρική αξιολόγησης, τον Συνολικό Αριθμό Μηνυμάτων. Η γραφική παράσταση αναμένουμε να έχει γραμμική μορφή. Αυτό προκύπτει γιατί στο σύστημα δεν υπάρχουν απώλειες μηνυμάτων και είναι σταθερός ο αριθμός των μηνυμάτων που αποστέλλει η κάθε διεργασία ούτως ώστε να ολοκληρώσει τις λειτουργίες της.



Σχήμα 5.6: Συνολικός Αριθμός Μηνυμάτων αλγορίθμου (n,β) -LIS χωρίς σφάλματα

Στην πιο πάνω γραφική παράσταση (Σχήμα 5.6) φαίνεται ο συνολικός αριθμός μηνυμάτων που στάλθηκαν συνολικά στο δίκτυο για τα σενάρια του αλγορίθμου χωρίς σφάλματα. Στον άξονα x φαίνονται οι αριθμοί των κόμβων οι οποίοι όπως προαναφέραμε είναι 32,50,64,80,100 και 128. Στον άξονα y φαίνονται οι συνολικοί αριθμοί των μηνυμάτων.

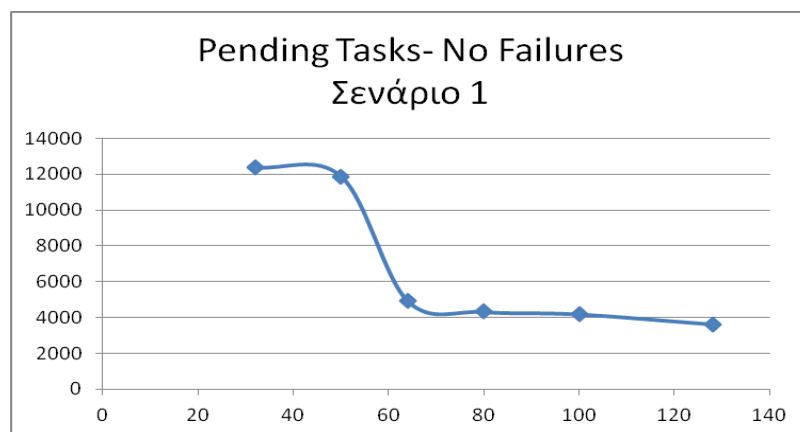
Η γραφική παράσταση έχει γραμμική μορφή. Ο αριθμός των μηνυμάτων αυξάνεται γραμμικά ως προς τον αριθμό των κόμβων του συστήματος και είναι ο ίδιος για όλα τα σενάρια. Επομένως, τα αποτελέσματα συμπίπτουν και επαληθεύουν τα αναμενόμενα αποτελέσματα.

Pending Tasks και Pending Cost

Προχωρούμε στις δύο τελευταίες μετρικές αξιολόγησης, Pending Tasks και Pending Cost. Οι γραφικές παραστάσεις δεν αναμένουμε να έχουν κάποια σταθερή μορφή. Αυτό προκύπτει από το γεγονός ότι οι εργασίες που εκτελούνται δεν έχουν τους ίδιους χρόνους εκτέλεσης και οι επεξεργαστές έχουν διαφορετικές ταχύτητες και επομένως

στις χρονικές στιγμές όπου πάρθηκαν οι μετρήσεις των εν αναμονή εργασιών και του κόστους τους, ο αριθμός των εργασιών που έχει ήδη ολοκληρώσει ο κάθε επεξεργαστής είναι διαφορετικός και όχι σταθερός. Θα μελετήσουμε το κάθε ένα από τα τέσσερα σενάρια ξεχωριστά και έπειτα όλα μαζί.

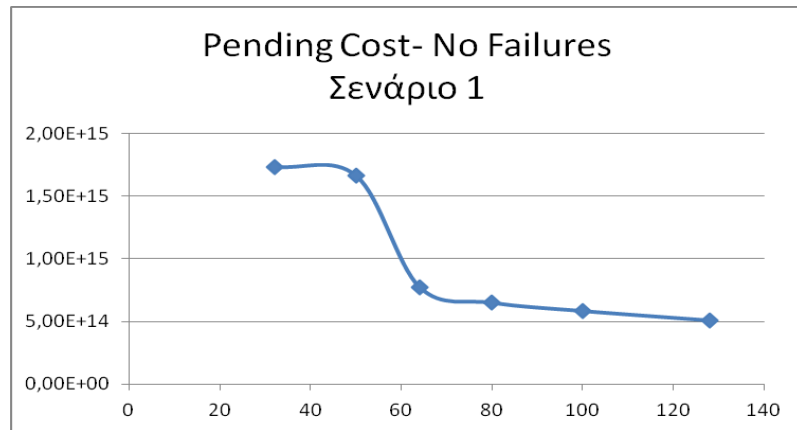
Στα ζεύγη γραφικών παραστάσεων που ακολουθούν, φαίνονται αρχικά οι συνολικοί αριθμοί των εργασιών εν αναμονή και έπειτα φαίνεται ο συνολικός αριθμός του κόστους των εργασιών αυτών για τα σενάρια του αλγορίθμου χωρίς σφάλματα. Στον άξονα χ φαίνονται οι αριθμοί των κόμβων οι οποίοι όπως προαναφέραμε είναι 32,50,64,80,100 και 128 ενώ στον άξονα ψ για την πρώτη γραφική παράσταση φαίνονται οι συνολικοί αριθμοί των Pending Tasks και για την δεύτερη φαίνονται οι συνολικοί αριθμοί του Pending Cost .



Σχήμα 5.7: Συνολικός Αριθμός Pending Tasks για Σενάριο 1 χωρίς σφάλματα

Στην πιο πάνω γραφική παράσταση (Σχήμα 5.7) φαίνεται ο συνολικός αριθμός των Pending Tasks για το Σενάριο 1 του αλγορίθμου χωρίς σφάλματα όπου δεν υπάρχει επιτάχυνση ($s=1$ και $cmin=12000$, $cmax=16000$).

Ενώ στην πιο κάτω γραφική παράσταση (Σχήμα 5.8) φαίνεται ο συνολικός αριθμός του Pending Cost για το Σενάριο 1 του αλγορίθμου χωρίς σφάλματα.

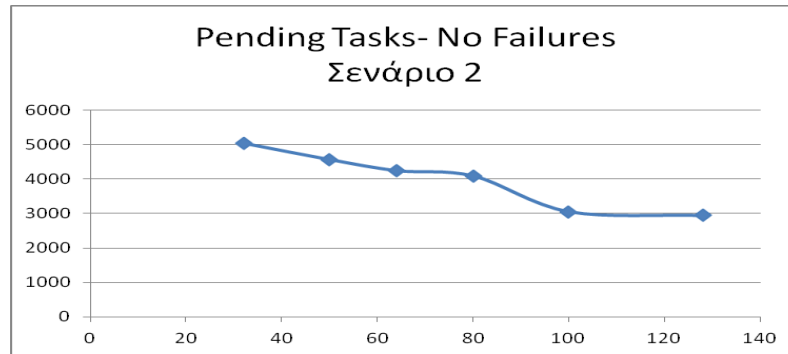


Σχήμα 5.8: Συνολικός Αριθμός Pending Cost για Σενάριο 1 χωρίς σφάλματα

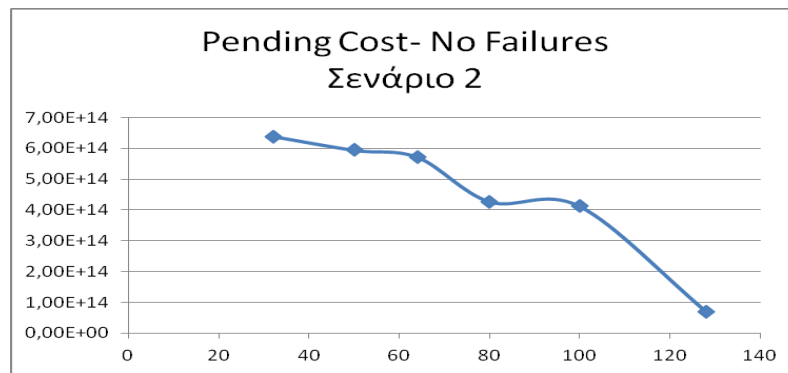
Ο αριθμός των Pending Tasks και ως αποτέλεσμα και του Pending Cost, παρατηρούμε ότι μειώνεται καθώς αυξάνεται ο αριθμός των κόμβων. Αυτό ήταν αναμενόμενο αφού όσο περισσότεροι κόμβοι Clients υπάρχουν στο σύστημα, τόσες περισσότερες εργασίες εκτελούνται και δεδομένου ότι ο αριθμός των εργασιών που προστίθενται είναι ίδιος για όλα τα σενάρια (όπως προαναφέρθηκε προστίθεται κάθε φορά ένας τυχαίος αριθμός εργασιών στο διάστημα [1000..1200]) άρα ο αριθμός των εργασιών που εισάχθηκαν στο σύστημα αλλά ακόμα δεν εκτελέστηκαν είναι λογικό να μειώνεται.

Όπως αναμέναμε, η μορφή των γραφικών παραστάσεων δεν είναι σταθερή. Αρχικά, παρατηρείται μια μικρή μείωση του αριθμού των εν αναμονή εργασιών και του κόστους τους (σημεία 1 και 2 όπου οι κόμβοι είναι 32 και 50 αντίστοιχα) ενώ όσο αυξάνονται οι κόμβοι παρατηρείται μεγάλη μείωση των τιμών. Συγκεκριμένα, απότομη πτώση υπάρχει από το δεύτερο στο τρίτο σημείο των γραφικών παραστάσεων (50 και 64 κόμβοι). Αυτή η μεγάλη πτώση δεν ήταν αναμενόμενη και ίσως να οφείλεται στο τυχαία επιλεγμένο κόστος των εργασιών και στην ασύγχρονη φύση του συστήματος. Δηλαδή, μέχρι τις χρονικές στιγμές που καταμετρήθηκαν οι τιμές των εκκρεμών εργασιών και του κόστους τους, στην εκτέλεση του σεναρίου με αριθμό κόμβων 50 φαίνεται να εκτελέστηκαν λιγότερες εργασίες από το αναμενόμενο, λόγω του μεγαλύτερου κόστους τους ενώ στην εκτέλεση με 60 κόμβους να εκτελέστηκαν περισσότερες εργασίες λόγω του μικρότερου κόστους τους.

Έπειτα, βλέπουμε ότι η μείωση των τιμών είναι ομαλή.



Σχήμα 5.9: Συνολικός Αριθμός Pending Tasks για Σενάριο 2 χωρίς σφάλματα



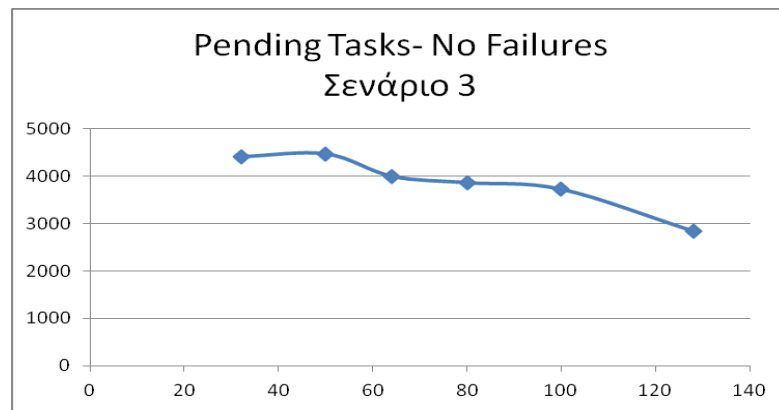
Σχήμα 5.10: Συνολικός Αριθμός Pending Cost για Σενάριο 2 χωρίς σφάλματα

Στις πιο πάνω γραφικές παραστάσεις (Σχήμα 5.9 και Σχήμα 5.10) φαίνονται ο συνολικός αριθμός των Pending Tasks και ο συνολικός αριθμός του Pending Cost για το Σενάριο 2 του αλγορίθμου χωρίς σφάλματα όπου η επιτάχυνση είναι ίση με $s=c_{max}/c_{min}$ και $c_{min}=12000$, $c_{max}=16000$. Παρατηρούμε και εδώ, όπως και στο προηγούμενο σενάριο, ότι όσο αυξάνεται ο αριθμός των κόμβων που βρίσκονται στο σύστημα, ο αριθμός των Pending Tasks και ως αποτέλεσμα και του Pending Cost, μειώνεται για τον ίδιο λόγο που προαναφέρθηκε.

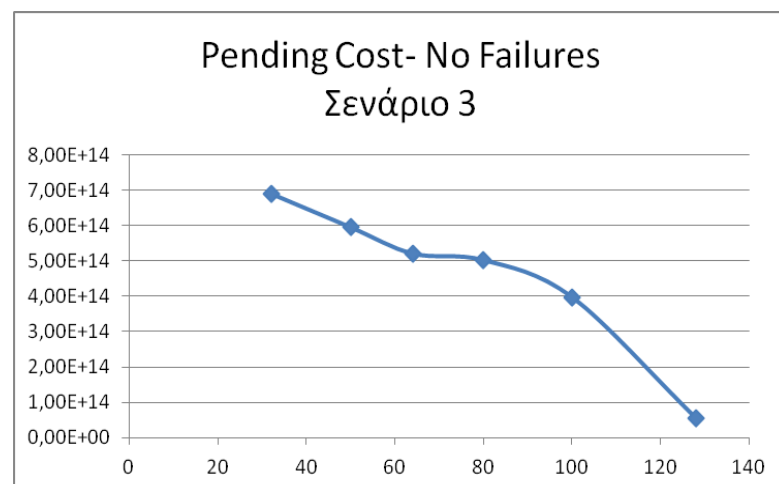
Παρατηρείται και σε αυτές τις γραφικές παραστάσεις μια μη σταθερή μείωση των τιμών καθώς μεγαλώνουν οι κόμβοι. Η μείωση στην γραφική παράσταση των Pending Tasks όμως παρουσιάζεται σε αυτό το σενάριο να είναι πιο ομαλή. Ωστόσο, στην γραφική παράσταση του Pending Cost παρατηρείται μία απότομη μείωση στην εκτέλεση όπου οι κόμβοι είναι 128 και αυτό αποδίδεται τόσο στην κλίμακα των

γραφικών παραστάσεων όσο και στους μεγάλους αριθμούς που αντιστοιχούν σε κάθε σημείο.

Στις επόμενες δύο γραφικές παραστάσεις (Σχήμα 5.11 και Σχήμα 5.12) φαίνονται ο συνολικός αριθμός των Pending Tasks και ο συνολικός αριθμός του Pending Cost για το Σενάριο 3 του αλγορίθμου χωρίς σφάλματα όπου η επιτάχυνση είναι ίση με $s=1+\sqrt{1-c_{\min}/c_{\max}}$ και $c_{\min}=10000$, $c_{\max}=17000$. Όμοια με τα προηγούμενα δύο σενάρια, παρατηρείται ότι όσο αυξάνεται ο αριθμός των κόμβων που βρίσκονται στο σύστημα, ο αριθμός των Pending Tasks και ως αποτέλεσμα και του Pending Cost, μειώνεται για τον ίδιο λόγο που προαναφέρθηκε.



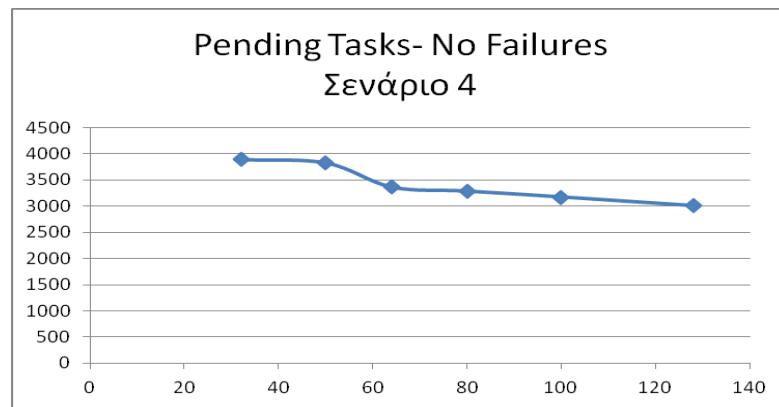
Σχήμα 5.11: Συνολικός Αριθμός Pending Tasks για Σενάριο 3 χωρίς σφάλματα



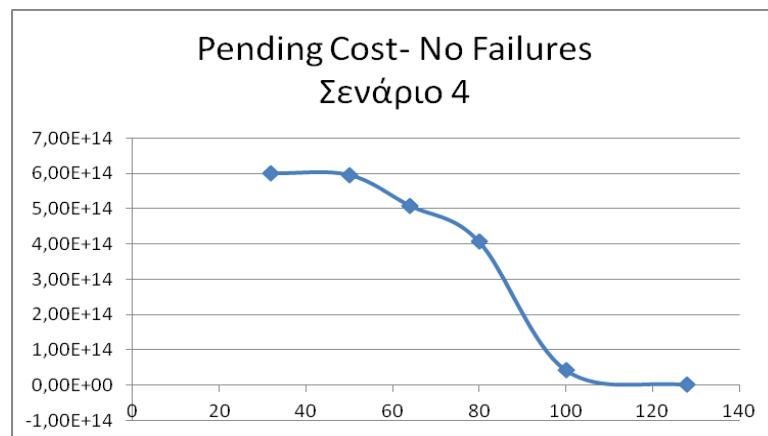
Σχήμα 5.12: Συνολικός Αριθμός Pending Cost για Σενάριο 3 χωρίς σφάλματα

Παρατηρούμε και σε αυτό το σενάριο, ότι η μείωση των τιμών του Pending Tasks είναι ομαλή αντίθετα με το Σενάριο 1 όπου υπήρχαν σημεία με μεγάλη διακύμανση. Επίσης στην γραφική παράσταση του Pending Cost παρατηρείται και πάλι μία απότομη μείωση στην εκτέλεση όπου οι κόμβοι είναι 128 για τους ίδιους λόγους που προαναφέρθηκαν.

Στις επόμενες δύο γραφικές παραστάσεις (Σχήμα 5.13 και Σχήμα 5.14) φαίνονται ο συνολικός αριθμός των Pending Tasks και ο συνολικός αριθμός του Pending Cost για το Σενάριο 4 του αλγορίθμου χωρίς σφάλματα όπου η επιτάχυνση είναι ίση με $s=1+\sqrt{1-c_{min}/c_{max}}$ και $c_{min}=10000$, $c_{max}=21000$. Και σε αυτό το σενάριο, όμοια με τα προηγούμενα σενάρια, παρατηρείται ότι όσο αυξάνεται ο αριθμός των κόμβων που βρίσκονται στο σύστημα, ο αριθμός των Pending Tasks και ως αποτέλεσμα και του Pending Cost, μειώνεται για τον ίδιο λόγο που προαναφέρθηκε.



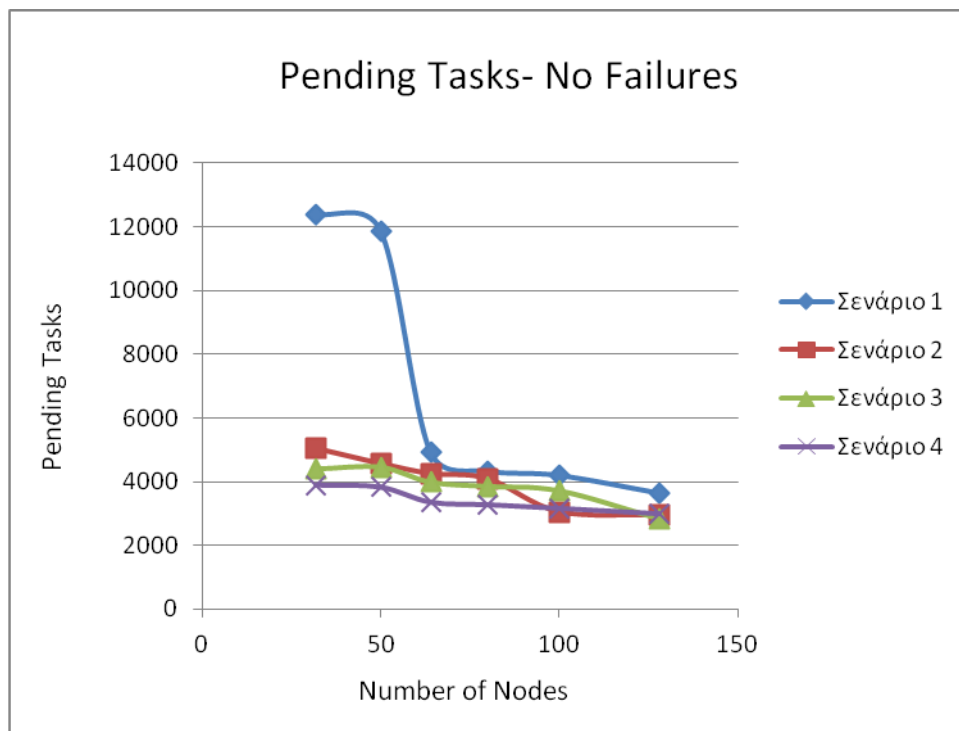
Σχήμα 5.13: Συνολικός Αριθμός Pending Tasks για Σενάριο 4 χωρίς σφάλματα



Σχήμα 5.14: Συνολικός Αριθμός Pending Cost για Σενάριο 4 χωρίς σφάλματα

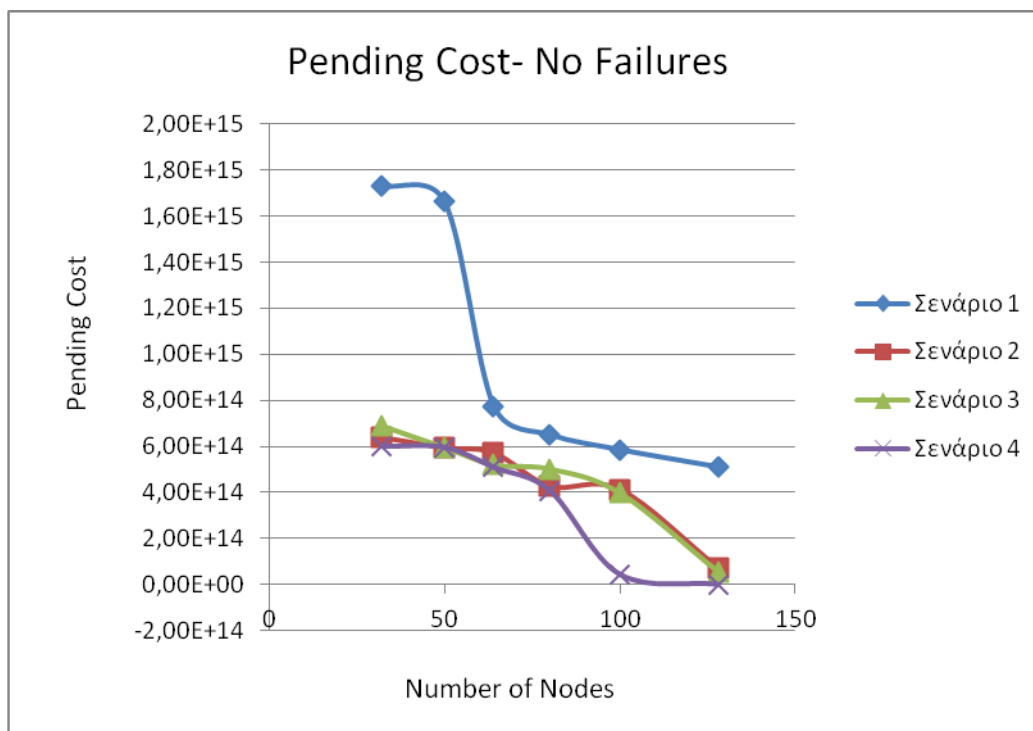
Παρατηρούμε και σε αυτό το σενάριο, όπως και στο προηγούμενο ότι η μείωση των τιμών του Pending Tasks είναι ομαλή ενώ στην γραφική παράσταση του Pending Cost παρατηρείται και πάλι μία απότομη μείωση στην εκτέλεση όπου οι κόμβοι είναι 128 για τους ίδιους λόγους που προαναφέρθηκαν.

Στις επόμενες δύο γραφικές παραστάσεις (Σχήμα 5.15 και Σχήμα 5.16) φαίνονται ο συνολικός αριθμός των Pending Tasks και ο συνολικός αριθμός του Pending Cost και για τα τέσσερα σενάρια του αλγορίθμου χωρίς σφάλματα.



Σχήμα 5.15: Συνολικός Αριθμός Pending Tasks αλγορίθμου (n,β)-LIS χωρίς σφάλματα

Στην πιο πάνω γραφική παράσταση (Σχήμα 5.15) φαίνεται ο συνολικός αριθμός των Pending Tasks για τα σενάρια του αλγορίθμου χωρίς σφάλματα. Στον άξονα x φαίνονται οι αριθμοί των κόμβων οι οποίοι όπως προαναφέραμε είναι 32,50,64,80,100 και 128. Στον άξονα y φαίνονται οι συνολικοί αριθμοί των Pending Tasks.



Σχήμα 5.16: Συνολικός Αριθμός Pending Cost αλγορίθμου (n,β)-LIS χωρίς σφάλματα

Ενώ στην πιο πάνω γραφική παράσταση (Σχήμα 5.16) φαίνεται ο συνολικός αριθμός του Pending Cost για τα σενάρια του αλγορίθμου χωρίς σφάλματα. Στον άξονα χ φαίνονται οι αριθμοί των κόμβων οι οποίοι όπως προαναφέραμε είναι 32,50,64,80,100 και 128. Στον άξονα ψ φαίνονται οι συνολικοί αριθμοί του Pending Cost.

Όπως φαίνεται στις δύο πιο πάνω γραφικές παραστάσεις για το Σενάριο 1 όπου δεν υπάρχει επιτάχυνση ο αλγόριθμος δεν αποδίδει καλά σε σχέση με τα υπόλοιπα τρία σενάρια ειδικά για μικρό αριθμό κόμβων. Όσο αυξάνεται η επιτάχυνση, τόσο καλύτερα είναι τα αποτελέσματα. Στα σενάρια 2 και 3 ο αλγόριθμος παρουσιάζει καλά αποτελέσματα με το σενάριο 4 να είναι το πιο αποδοτικό.

Αυτό ήταν αναμενόμενο αφού η καλή απόδοση του αλγορίθμου, όπως αποδείχθηκε και στο [5], βασίζεται στην ύπαρξη της επιτάχυνσης και για αυτό το λόγο ήταν αναμενόμενη η μειονεκτική θέση του συστήματος στο σενάριο 1 όπου δεν υπάρχει επιτάχυνση.

Δεδομένου ότι στο Σενάριο 1 οι Clients χρειάζονται χρόνο όσο είναι το κόστος κάθε εργασίας για να εκτελέσουν την εργασία, προκύπτει ότι μειονεκτεί έναντι στα υπόλοιπα στα οποία το κόστος αυτό διαιρείται με την επιτάχυνση. Αυτό συμβαίνει γιατί οι μετρήσεις για όλα τα σενάρια γίνονται στα ίδια χρονικά διαστήματα και στα σενάρια 2,3 και 4 οι Clients είναι σε θέση να εκτελέσουν περισσότερες εργασίες στον ίδιο χρόνο λόγω της επιτάχυνσης. Για τον ίδιο ακριβώς λόγο το σενάριο 4, το οποίο έχει την πιο μεγάλη επιτάχυνση, φαίνεται να είναι το πιο αποδοτικό και από τα τέσσερα σενάρια αφού έχει τις μικρότερες τιμές για κάθε αριθμό κόμβων.

5.2.2 Αλγόριθμος (n,β)-LIS με σφάλματα

Στη συνέχεια, ο αλγόριθμος (n,β)-LIS εκτελέστηκε με την παρουσία σφαλμάτων κατάρρευσης και επανεκκίνησης. Συνολικά εκτελέστηκε δεκαέξι (16) φορές, με διαφορές στην επιτάχυνση s , στα c_{min}, c_{max} και στην πιθανότητα κατάρρευσης κόμβων όπως περιγράφονται και στο υποκεφάλαιο 5.1.3. Συγκεκριμένα, εκτελέστηκαν δεκαέξι σενάρια για αριθμό κόμβων 64.

Σενάριο	Total Execution Time	Number of Msgs	Pending Tasks	Pending Cost	Total Execution Time
1	533781	2481	5523	6,53E+14	533781
2	429626	2743	4667	6,10E+14	429626
3	308025	3256	4572	7,73E+13	308025
4	288869	3467	4359	6,40E+13	288869
5	549883	2018	6164	8,63E+14	549883
6	339660	2890	5563	7,79E+14	309660
7	285106	3019	4467	6,06E+14	285106
8	285117	3500	4326	6,25E+13	285117
9	330839	2453	4398	5,62E+14	330839
10	305823	2789	4163	5,21E+14	305823
11	285109	3245	3872	5,23E+13	285109

12	292869	3654	3852	5,93E+12	292869
13	355220	2242	4487	8,53E+14	355220
14	343150	2678	3828	5,21E+14	353150
15	274150	3198	3355	4,48E+14	274150
16	262454	3568	2890	5,94E+13	262454

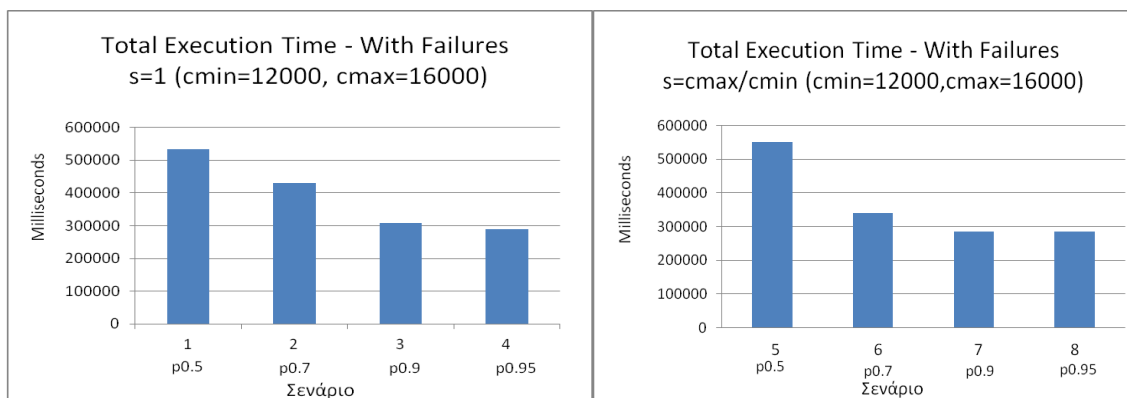
Πίνακας 5.7: Αποτελέσματα εκτελέσεων αλγορίθμου (n,β)-LIS με σφάλματα

Στον πιο πάνω πίνακα (Πίνακας 5.7) εμφανίζονται τα αποτελέσματα των σεναρίων του αλγορίθμου όπως έχουν εξαχθεί από τα αρχεία ελέγχου. Εύκολα γίνεται αντιληπτό ότι κατά τις εκτελέσεις των σεναρίων 1,5,9 και 13 όπου η πιθανότητα να παραμείνει ενεργός ένας κόμβος ήταν σχετικά μικρή (50%), το σύστημα δεν απέδιδε ικανοποιητικά.

Στις εκτελέσεις των σεναρίων 3,7,11 και 15 με πιθανότητα 90% να παραμείνει ζωντανός ένας κόμβος και στις εκτελέσεις των σεναρίων 4,8,12 και 16 με πιθανότητα 95%, παρατηρούμε ότι τα αποτελέσματα βρίσκονται σε πολύ καλά επίπεδα.

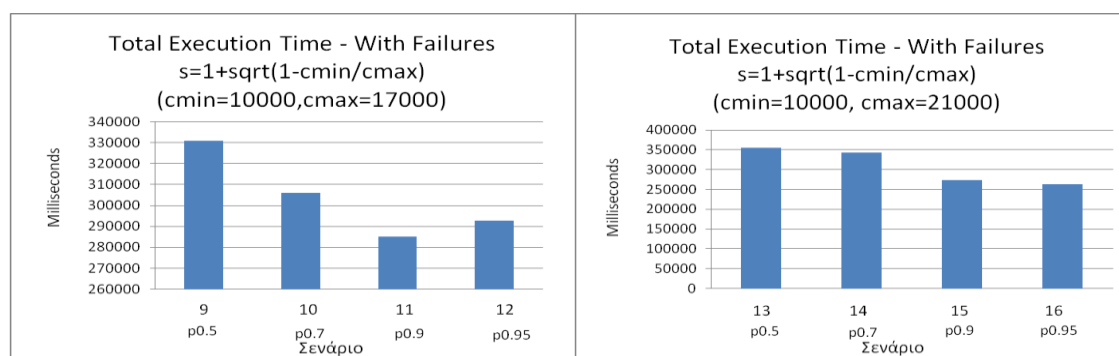
Χρόνος Διεκπεραίωσης

Πιο κάτω (Σχήματα 5.17, 5.18, 5.19 και 5.20) στις γραφικές παραστάσεις φαίνονται οι συνολικοί χρόνοι διεκπεραίωσης για τα σενάρια του αλγορίθμου με την παρουσία σφαλμάτων κατάρρευσης και επανεκκίνησης. Τα σενάρια κατηγοριοποιήθηκαν ανά τετράδες με βάση την επιτάχυνση. Στον άξονα χ φαίνονται οι αριθμοί των διάφορων σεναρίων ενώ στον άξονα ψ φαίνονται οι χρόνοι διεκπεραίωσης σε milliseconds.



Σχήμα 5.17: Συνολικός Χρόνος Διεκπεραίωσης αλγορίθμου (n,β)-LIS με σφάλματα για σενάρια 1,2,3 και 4

Σχήμα 5.18: Συνολικός Χρόνος Διεκπεραίωσης αλγορίθμου (n,β)-LIS με σφάλματα για τα σενάρια 5,6,7 και 8



Σχήμα 5.19: Συνολικός Χρόνος Διεκπεραίωσης αλγορίθμου (n,β)-LIS με σφάλματα για τα σενάρια 9,10,11 και 12

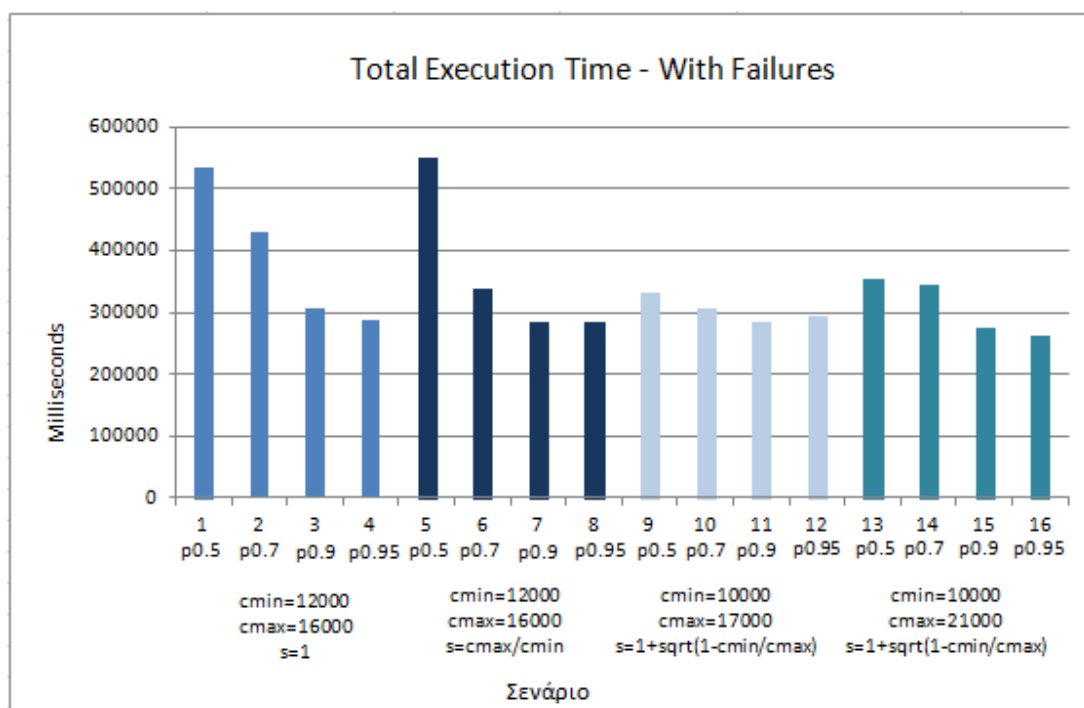
Σχήμα 5.20: Συνολικός Χρόνος Διεκπεραίωσης αλγορίθμου (n,β)-LIS με σφάλματα για τα σενάρια 13,14,15 και 16

Ο χρόνος διεκπεραίωσης στα σενάρια με μεγάλη πιθανότητα κατάρρευσης κόμβου, δηλαδή στα σενάρια όπου η πιθανότητα ένας κόμβος να παραμείνει ενεργός είναι 50% (σενάρια 1,5,9 και 13), αυξάνεται σημαντικά. Αυτό το αποδίδουμε στο ότι περισσότεροι κόμβοι καταρρέουν και επανεκκινούν με αποτέλεσμα να καταναλώνεται περισσότερος χρόνος, ειδικά όταν κάποιοι κόμβοι επανεκκινούν προς το τέλος των κύκλων των υπολοίπων επεξεργαστών και έτσι καθυστερούν τον συνολικό χρόνο εκτέλεσης του αλγορίθμου. Τα ίδια, ελαφρώς βελτιωμένα αποτελέσματα παρουσίασαν και τα σενάρια με πιθανότητα να παραμείνει ενεργός ένας κόμβος 70% (σενάρια 2,6, 10 και 14). Καθώς όμως η πιθανότητα αυτή μεγαλώνει, φτάνει δηλαδή στο 90% και 95% τότε παρατηρούμε πολύ βελτιωμένα αποτελέσματα. Ο χρόνος διεκπεραίωσης για τα σενάρια 3,4,7,8,11,12,15 και 16 φτάνει μέχρι και στο μισό από ότι στα προαναφερθέντα

σενάρια. Σε αυτά τα σενάρια ο αλγόριθμος φαίνεται να αποδίδει καλά ακόμη και στην παρουσία σφαλμάτων.

Και για τις τέσσερεις γραφικές παραστάσεις παρατηρείται μείωση του χρόνου διεκπεραίωσης καθώς αυξάνεται η πιθανότητα να παραμείνει ενεργός ένας κόμβος όπως προαναφέραμε. Εξαιρεση όμως αποτελεί το Σενάριο 11. Στο σενάριο αυτό, όπου οι κόμβοι μένουν ενεργοί με πιθανότητα 90% , βλέπουμε ότι ο χρόνος διεκπεραίωσης είναι πιο χαμηλός από ότι στο Σενάριο 12 όπου η πιθανότητα αυτή είναι 95%. Αυτό το αποδίδουμε στο γεγονός ότι όπως είπαμε οι κόμβοι καταρρέουν βάση κάποιας πιθανότητας και η πιθανότητα στα σενάρια 11 και 12 είναι πολύ κοντινή (90% και 95%) οπότε λογικό θα ήταν στο Σενάριο 12 να συνέβηκαν περισσότερα σφάλματα κατάρρευσης και επανεκκίνησης, με αποτέλεσμα να παρατηρούμε μεγαλύτερο χρόνο διεκπεραίωσης. Ακόμα μία εξήγηση για αυτή την εξαιρεση αποτελεί και το τυχαίο κόστος που έχουν οι εργασίες, δηλαδή στο Σενάριο 11 πιθανό θα ήταν να εκτελέστηκαν εργασίες με μικρότερα κόστη από ότι στο Σενάριο 12 με αποτέλεσμα να ολοκληρωθεί πιο γρήγορα η εκτέλεση του αλγορίθμου.

Στην συνέχεια, βλέπουμε στην πιο κάτω γραφική παράσταση (Σχήμα 5.21) τον Χρόνο Διεκπεραίωσης για τα 16 σενάρια που εκτελέστηκαν με την παρουσία σφαλμάτων.



Σχήμα 5.21: Συνολικός Χρόνος Διεκπεραίωσης αλγορίθμου (n.β)-LIS με σφάλματα για όλα τα σενάρια

Στον άξονα χ φαίνονται οι αριθμοί των διάφορων σεναρίων ενώ στον άξονα ψ φαίνονται οι χρόνοι διεκπεραίωσης σε milliseconds. Τα σενάρια είναι κατηγοριοποιημένα ανά τετράδες με βάση την επιτάχυνση.

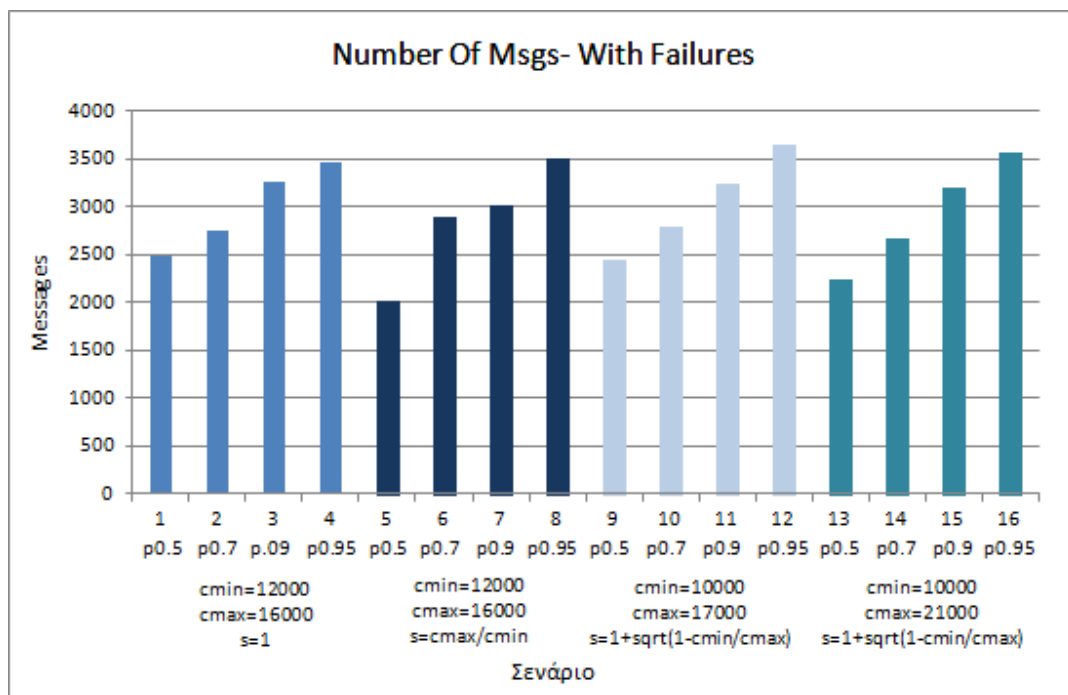
Από αυτή την γραφική παράσταση απορρέουν σημαντικές πληροφορίες σχετικά με την επίδραση της επιτάχυνσης και των σφαλμάτων κατάρρευσης και επανεκκίνησης στον χρόνο διεκπεραίωσης του αλγορίθμου. Γενική παρατήρηση αποτελεί το γεγονός ότι όσο αυξάνεται η επιτάχυνση μειώνεται ο χρόνος διεκπεραίωσης του αλγορίθμου. Παρατηρούμε επίσης ότι καθώς αυξάνεται η επιτάχυνση, τα σφάλματα φαίνεται να μην επηρεάζουν πολύ τον χρόνο του αλγορίθμου. Αυτό προκύπτει από το γεγονός ότι στα σενάρια 1 μέχρι 4, όπου δεν υπάρχει επιτάχυνση και στα σενάρια 5 μέχρι 8 όπου η επιτάχυνση είναι $s=c_{max}/c_{min}$ ($c_{min}=12000$, $c_{max}=16000$) παρατηρείται μεγάλη διακύμανση ανάμεσα στους χρόνους των σεναρίων όσο αυξάνεται η πιθανότητα να μείνει ενεργός ένας κόμβος. Αντιθέτως, στα σενάρια όπου η επιτάχυνση μεγαλώνει, δηλαδή στα σενάρια 9 μέχρι 12 όπου η επιτάχυνση είναι $s=1+\sqrt{1-c_{min}/c_{max}}$ ($c_{min}=10000$, $c_{max}=17000$) και στα σενάρια 13 μέχρι 16 όπου η επιτάχυνση είναι $s=1+\sqrt{1-c_{min}/c_{max}}$ και $c_{min}=10000$, $c_{max}=21000$, η διακύμανση αυτή δεν φαίνεται να είναι αρκετά μεγάλη.

Αυτό αποδίδεται στο γεγονός ότι ο χρόνος που χρειάζεται ένας επεξεργαστής για να εκτελέσει μία εργασία στα τελευταία σενάρια (σενάρια 9 μέχρι 16) διαιρείται με ένα μεγάλο αριθμό, τον αριθμό της επιτάχυνσης και έτσι ο χρόνος διεκπεραίωσης των σεναρίων αυτών είναι κατά πολύ μικρότερος ακόμα και στην παρουσία σφαλμάτων. Από ένα σημείο και μετά, βλέπουμε ότι λόγω της επιτάχυνσης ο αλγόριθμος αποδίδει πολύ καλά ακόμα και στην παρουσία σφαλμάτων.

Τέλος, παρατηρούμε ότι στο Σενάριο 16, όπου υπάρχει ο συνδυασμός της μεγαλύτερης επιτάχυνσης και της μεγαλύτερης πιθανότητας να παραμείνει ενεργός ένας κόμβος, έχουμε τα καλύτερα αποτελέσματα. Ο χρόνος διεκπεραίωσης σε αυτό το σενάριο είναι ο πιο μικρός από όλα τα σενάρια.

Συνολικός Αριθμός Μηνυμάτων

Προχωρώντας, θα αναλύσουμε τις τιμές που πήραμε για την δεύτερη μετρική αξιολόγησης, τον Συνολικό Αριθμό Μηνυμάτων που στάλθηκαν από όλες τις διεργασίες. Στην πιο κάτω γραφική παράσταση (Σχήμα 5.22) φαίνεται ο συνολικός αριθμός μηνυμάτων που στάλθηκαν συνολικά στο δίκτυο για τα σενάρια του αλγορίθμου με σφάλματα κατάρρευσης και επανεκκίνησης. Στον άξονα χ φαίνονται οι αριθμοί των σεναρίων ενώ στον άξονα ψ φαίνονται οι συνολικοί αριθμοί των μηνυμάτων. Τα σενάρια και εδώ κατηγοριοποιήθηκαν ανά τετράδες με βάση την επιτάχυνση.



Σχήμα 5.22: Συνολικός Αριθμός Μηνυμάτων αλγορίθμου (n,β)-LIS με σφάλματα

Όπως παρατηρούμε, ο αριθμός των μηνυμάτων που κινήθηκαν στο σύστημα ήταν ο αναμενόμενος καθώς ο αριθμός των μηνυμάτων που αποστέλλει η κάθε διεργασία είναι σταθερός ούτως ώστε να ολοκληρώσει τις λειτουργίες της. Στο πιο πάνω σχήμα (Σχήμα 5.22) βλέπουμε ότι κατά τις εκτελέσεις των σεναρίων με μεγάλη πιθανότητα κατάρρευσης ο αριθμός των μηνυμάτων που κινήθηκαν στο δίκτυο από και προς το

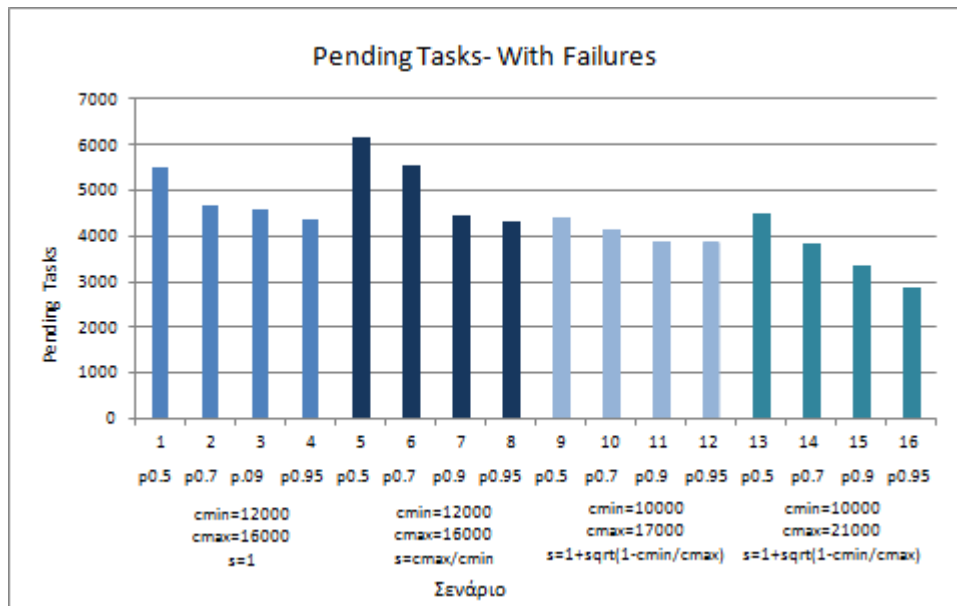
Repository είναι μειωμένος. Αυτό προκύπτει από το γεγονός ότι πολλοί κόμβοι Clients δεν ήταν ενεργοί ούτως ώστε να στείλουν μηνύματα και να λάβουν απάντηση από το Repository. Αντίθετα, στα σενάρια με ελάχιστη πιθανότητα κατάρρευσης, αποστάληκαν σχεδόν όσα μηνύματα αποστάληκαν και στα σενάρια που εκτελέστηκαν χωρίς σφάλματα (Σχήμα 5.6) αφού η πλειονότητα των επεξεργαστών ήταν ενεργή και έστελνε μηνύματα.

Σημαντική επίσης παρατήρηση αποτελεί το γεγονός ότι, για αυτή την μετρική αξιολόγησης, τον Συνολικό Αριθμό Μηνυμάτων, δεν φάνηκε να έχει σημαντική επίπτωση η επιτάχυνση η οποία ήταν διαφορετική σε κάποια σενάρια όπως προαναφέρθηκε στο υποκεφάλαιο 5.1.3 (Πίνακας 5.2). Αυτό έρχεται σε αντίθεση με την προηγούμενη μετρική αξιολόγησης, τον Χρόνο διεκπεραίωσης στον οποίο όπως προαναφέρθηκε η επιτάχυνση αποτέλεσε σημαντικό παράγοντα. Σημαντικός παράγοντας όμως για τον Αριθμό Μηνυμάτων όπως προαναφέρθηκε είναι η πιθανότητα κατάρρευσης κόμβων.

Αυτό προκύπτει από το γεγονός ότι, ανεξάρτητα από την επιτάχυνση, ο κάθε επεξεργαστής για το διάστημα που είναι ενεργός θα εκτελεί τις λειτουργίες του μέχρι να ολοκληρώσει τους κύκλους του και θα στέλνει σε κάθε κύκλο τα απαραίτητα μηνύματα και θα λαμβάνει απαντήσεις από το Repository. Δε επηρεάζεται ο αριθμός των μηνυμάτων από τον χρόνο ολοκλήρωσης της κάθε εργασίας για αυτό και δεν βλέπουμε να έχει κάποια επίπτωση η επιτάχυνση, παρά μόνο η πιθανότητα κατάρρευσης κόμβων για τους λόγους που προαναφέρθηκαν.

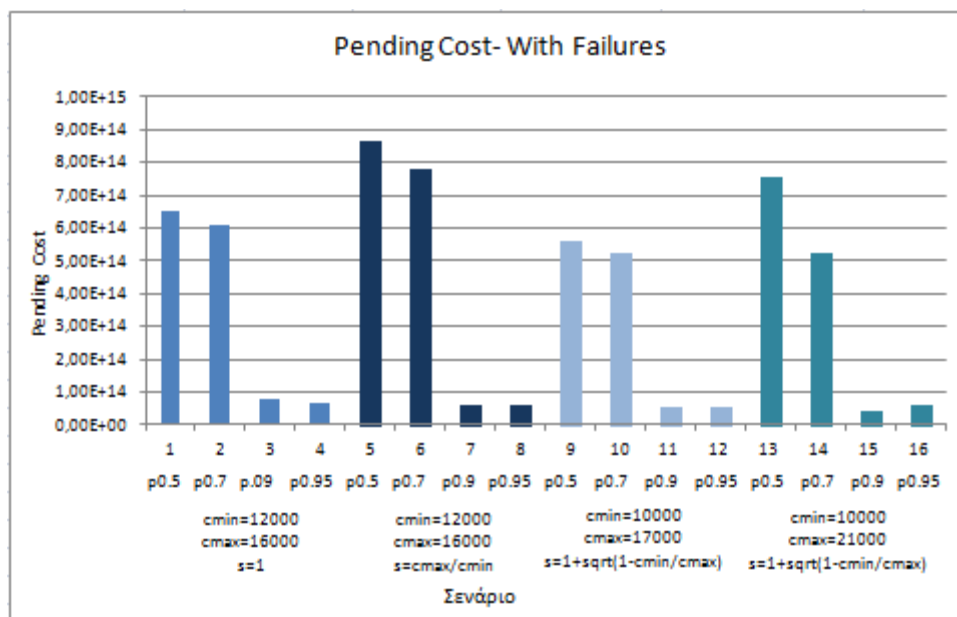
Pending Tasks και Pending Cost

Φτάνουμε τώρα στις δύο τελευταίες μετρικές αξιολόγησης, Pending Tasks και Pending Cost. Στην πιο κάτω γραφική παράσταση (Σχήμα 5.23) φαίνεται ο συνολικός αριθμός των Pending Tasks για τα σενάρια του αλγορίθμου με σφάλματα κατάρρευσης και επανεκκίνησης. Στον άξονα χ φαίνονται οι αριθμοί των σεναρίων ενώ στον άξονα ψ φαίνονται οι συνολικοί αριθμοί των Pending Tasks.



Σχήμα 5.23: Συνολικός Αριθμός Pending Tasks αλγορίθμου (n,β)-LIS με σφάλματα

Ενώ στην πιο κάτω γραφική παράσταση (Σχήμα 5.24) φαίνεται ο συνολικός αριθμός του Pending Cost για τα σενάρια του αλγορίθμου με σφάλματα κατάρρευσης και επανεκκίνησης. Στον άξονα χ φαίνονται οι αριθμοί των σεναρίων ενώ στον άξονα ψ φαίνονται οι συνολικοί αριθμοί του Pending Cost.



Σχήμα 5.24: Συνολικός Αριθμός Pending Cost αλγορίθμου (n,β)-LIS με σφάλματα

Και για αυτές τις δύο μετρικές τα σενάρια κατηγοριοποιήθηκαν ανά τετράδες με βάση την επιτάχυνση.

Ο αριθμός των Pending Tasks και ως αποτέλεσμα και του Pending Cost, παρατηρούμε ότι στα σενάρια όπου ένας κόμβος παραμένει ζωντανός με πιθανότητα 90% (σενάρια 3,7,11 και 15) και 95% (σενάρια 4,8,12 και 16) ο αλγόριθμος έχει πολύ καλή απόδοση. Αντιθέτως στα σενάρια όπου η πιθανότητα βρίσκεται στο 50% και 70% ο αλγόριθμος δεν αποδίδει ικανοποιητικά. Η μειονεκτική θέση του συστήματος σε αυτά τα σενάρια ήταν αναμενόμενη. Αυτό προκύπτει από το γεγονός ότι πολλοί κόμβοι Clients σε αυτά τα σενάρια δεν είναι ενεργοί κατά την διάρκεια των κύκλων που εκτελείται ο αλγόριθμος και δεν εκτελούν εργασίες, άρα είναι φυσικό επακόλουθο ο αριθμός των Pending Tasks και του Pending Cost να παρατηρείται αυξημένος.

Βλέπουμε επίσης, μία απότομη πτώση στην γραφική παράσταση του Pending Cost για τα σενάρια με πιθανότητα να παραμείνει ενεργός ένας κόμβος 90% και 95%. Ήταν αναμενόμενη η μείωση του αριθμού του Pending Cost για τον λόγο που προαναφέραμε, όμως όχι μία τόσο μεγάλη πτώση. Αυτό ίσως να οφείλεται στο τυχαίο κόστος που έχουν οι εργασίες. Σε αυτές τις εκτελέσεις πιθανό να εκτελέστηκαν από τους επεξεργαστές περισσότερες εργασίες με κόστος κοντινό στο c_{max} (αφού όπως προαναφέραμε το κόστος της κάθε εργασίας είναι ένας τυχαίος αριθμός στο εύρος $[c_{min}, c_{max}]$), και σε συνδυασμό με την μεγάλη πιθανότητα να παραμείνουν ενεργοί οι κόμβοι έχουμε ως αποτέλεσμα το συνολικό κόστος των εργασιών που εκκρεμούν να μειώνεται κατά πολύ.

Επίσης, ακόμα μία παρατήρηση αποτελεί το γεγονός ότι για τις αυτές τις μετρικές (Pending Tasks και Pending Cost) παρατηρείται αισθητή βελτίωση για τα σενάρια όπου η επιτάχυνση είναι μεγαλύτερη. Ο αριθμός των Pending Tasks και ως αποτέλεσμα και του Pending Cost, παρατηρούμε ότι μειώνεται καθώς αυξάνεται η επιτάχυνση. Αυτό προκύπτει από το γεγονός ότι όσο μεγαλύτερη είναι η επιτάχυνση, τόσο πιο γρήγορα εκτελεί ένας επεξεργαστής μία εργασία και άρα μέχρι το χρονικό σημείο που γίνεται η καταμέτρηση, περισσότερες εργασίες έχουν εκτελεστεί. Άρα δεδομένου ότι ο αριθμός των εργασιών που προστίθενται είναι ίδιος για όλα τα σενάρια (όπως προαναφέρθηκε

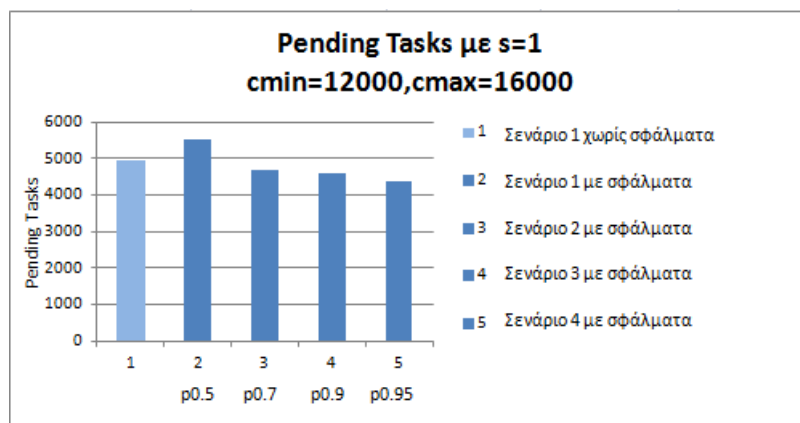
προστίθεται κάθε φορά ένας τυχαίος αριθμός εργασιών στο διάστημα [1000..1200]) ο αριθμός των εργασιών που εισάχθηκαν στο σύστημα αλλά ακόμα δεν εκτελέστηκαν είναι λογικό να μειώνεται.

Τέλος, στο σενάριο 16, όπου υπάρχει ο συνδυασμός της μεγαλύτερης επιτάχυνσης και μεγαλύτερης πιθανότητας να παραμείνει ενεργός ένας κόμβος, παρατηρούμε τα πιο καλά αποτελέσματα. Σε αυτό το σενάριο ο αλγόριθμος έχει πολύ καλή απόδοση.

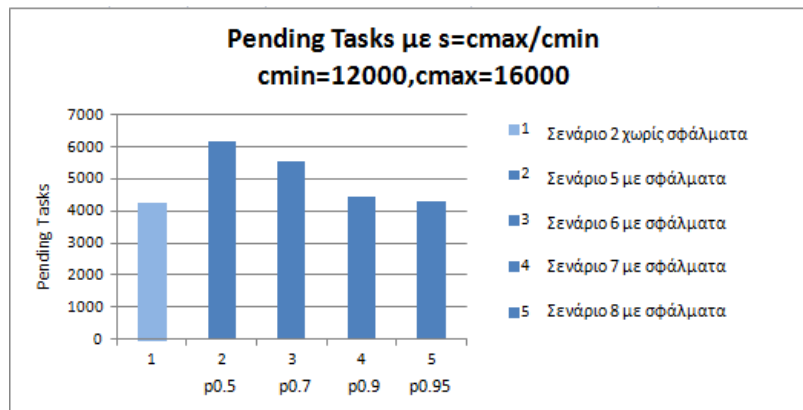
5.2.3 Σύγκριση Αλγορίθμου (n,β)-LIS με και χωρίς σφάλματα

Στο υποκεφάλαιο αυτό γίνεται σύγκριση της εφαρμογής του Αλγορίθμου (n,β)-LIS με και χωρίς σφάλματα κατάρρευσης και επανεκκίνησης βάσει των αποτελεσμάτων που παρουσιάστηκαν στα δύο προηγούμενα υποκεφάλαια.

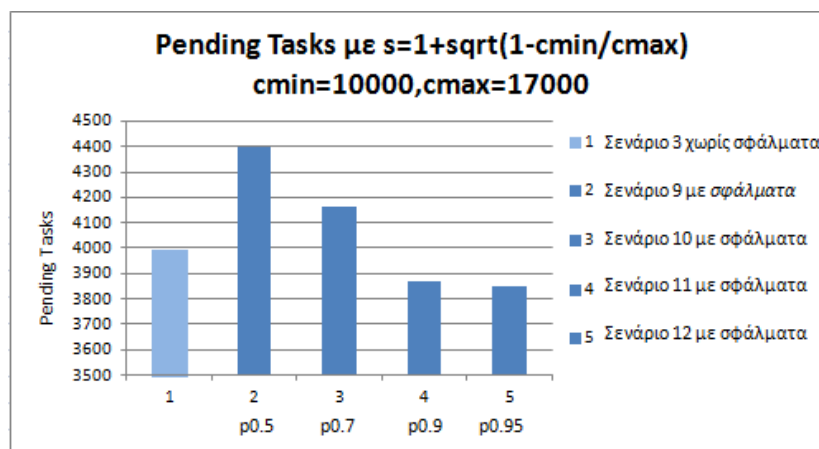
Πήραμε τα αποτελέσματα των σεναρίων 1,2,3 και 4 χωρίς σφάλματα (Πίνακες 5.3, 5.4, 5.5 και 5.6) και συγκεκριμένα τις τιμές που πήραμε για 64 κόμβους και τα συγκρίναμε με τις τιμές που πήραμε για τα 16 σενάρια με την παρουσία σφαλμάτων (Πίνακας 5.7). Τα αποτελέσματα κατηγοριοποιήθηκαν με βάση την επιτάχυνση. Οι γραφικές που προέκυψαν παρουσιάζονται πιο κάτω.



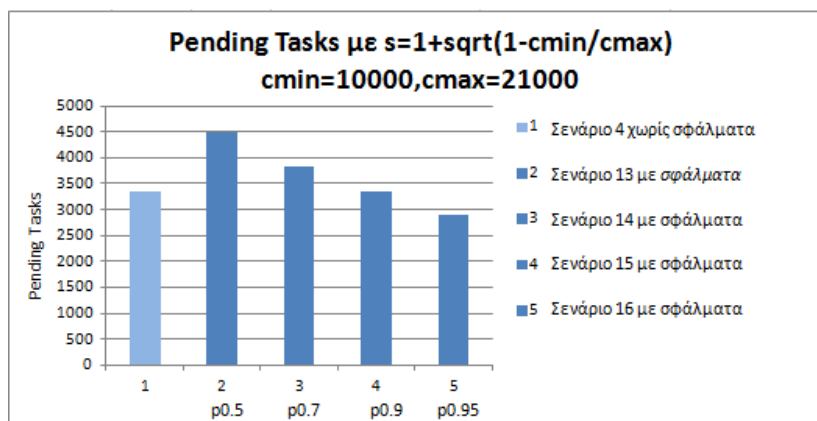
Σχήμα 5.25: Συνολικός Αριθμός Pending Task αλγορίθμου (n,β)-LIS



Σχήμα 5.26: Συνολικός Αριθμός Pending Task αλγορίθμου (n,β)-LIS



Σχήμα 5.27: Συνολικός Αριθμός Pending Task αλγορίθμου (n,β)-LIS



Σχήμα 5.28: Συνολικός Αριθμός Pending Task αλγορίθμου (n,β)-LIS

Στις πιο πάνω γραφικές παραστάσεις (Σχήματα 5.25, 5.26, 5.27 και 5.28) φαίνεται ο συνολικός αριθμός των Pending Tasks για τα σενάρια του αλγορίθμου με και χωρίς σφάλματα κατάρρευσης και επανεκκίνησης για 64 κόμβους. Σε κάθε μία από τις

γραφικές παραστάσεις στον άξονα χ φαίνονται οι αριθμοί των σεναρίων, αρχίζοντας από το σενάριο χωρίς σφάλματα και συνεχίζοντας με τα 4 σενάρια με σφάλματα. Στον άξονα ψ φαίνονται οι συνολικοί αριθμοί των Pending Tasks.

Αυτό που παρατηρούμε από αυτές τις γραφικές παραστάσεις είναι ότι στα σενάρια χωρίς την ύπαρξη σφαλμάτων οι τιμές που πήραμε από την πειραματική εκτέλεση είναι κοντινές με τις τιμές που πήραμε για τα σενάρια με σφάλματα όταν οι πιθανότητες να παραμείνει ενεργός ένας κόμβος είναι υψηλές (90% και 95%). Αυτό ισχύει για όλες τις τιμές της επιτάχυνσης.

Τα σενάρια στα οποία η πιθανότητα να παραμείνει ένας κόμβος ενεργός είναι 90% και 95% μπορούν να θεωρηθούν ως τα πιο ρεαλιστικά. Ο αλγόριθμος χωρίς την παρουσία σφαλμάτων παρουσιάζει παρόμοιες, σχεδόν τις ίδιες επιδόσεις με αυτά τα σενάρια. Σε γενικές γραμμές, μπορούμε να πούμε ότι ο αλγόριθμος είναι ανεκτικός σε σφάλματα κατάρρευσης και αποδίδει τόσο καλά στην παρουσία σφαλμάτων όσο και στην απουσία τους.

5.3 Συμπεράσματα

Μέσω των αναλύσεων των πειραματικών εκτελέσεων και ακολούθως της σύγκρισης του Αλγορίθμου (n,β) -LIS με και χωρίς σφάλματα, μπορούμε να συμπεράνουμε ότι ο Αλγόριθμος (n,β) -LIS μπορεί να αποδίδει καλά κάτω από οποιοδήποτε ρεαλιστικό σενάριο.

Συνοπτικά, οι παρατηρήσεις που προέκυψαν από την εκτέλεση του αλγορίθμου χωρίς σφάλματα είναι οι εξής:

Παρατηρήσαμε ότι όσο αυξάνεται ο αριθμός των κόμβων που βρίσκονται στο σύστημα, αυξάνεται και ο χρόνος διεκπεραίωσης του κάθε σεναρίου. Επίσης παρατηρήσαμε ότι ο αριθμός των μηνυμάτων αυξάνεται γραμμικά ως προς τον αριθμό των κόμβων του συστήματος και είναι ο ίδιος για όλα τα σενάρια. Και τέλος, παρατηρήσαμε ότι ο αριθμός των Pending Tasks και ως αποτέλεσμα και του Pending Cost, μειώνεται καθώς αυξάνεται ο αριθμός των κόμβων.

Για τις εκτελέσεις των σεναρίων με σφάλματα κατάρρευσης και επανεκκίνησης προέκυψαν οι εξής παρατηρήσεις:

Για τις γραφικές παραστάσεις του χρόνου διεκπεραίωσης παρατηρείται μείωση του χρόνου καθώς αυξάνεται η πιθανότητα να παραμείνει ενεργός ένας κόμβος. Ακόμα μία παρατήρηση αποτελεί το γεγονός ότι όσο αυξάνεται η επιτάχυνση μειώνεται ο χρόνος διεκπεραίωσης του αλγορίθμου. Επίσης καθώς αυξάνεται η επιτάχυνση, τα σφάλματα φαίνεται να μην επηρεάζουν πολύ τον χρόνο του αλγορίθμου. Στη συνέχεια παρατηρήθηκε ότι κατά τις εκτελέσεις των σεναρίων με μεγάλη πιθανότητα κατάρρευσης ο αριθμός των μηνυμάτων που κινήθηκαν στο δίκτυο από και προς το Repository είναι μειωμένος ενώ αντίθετα για τα σενάρια με ελάχιστη πιθανότητα κατάρρευσης, αποστάληκαν σχεδόν όσα μηνύματα αποστάληκαν και στα σενάρια που εκτελέστηκαν χωρίς σφάλματα. Σημαντική επίσης παρατήρηση αποτελεί το γεγονός ότι για τον Συνολικό Αριθμό Μηνυμάτων, δεν φάνηκε να έχει σημαντική επίπτωση η επιτάχυνση η οποία ήταν διαφορετική σε κάποια σενάρια. Προχωρώντας, ο αριθμός των Pending Tasks και ως αποτέλεσμα και του Pending Cost, παρατηρούμε ότι στα σενάρια όπου ένας κόμβος παραμένει ζωντανός με πιθανότητα 90% και 95% ο αλγόριθμος έχει πολύ καλή απόδοση. Αντιθέτως στα σενάρια όπου η πιθανότητα βρίσκεται στο 50% και 70% ο αλγόριθμος δεν αποδίδει ικανοποιητικά. Και τέλος, ακόμα μία παρατήρηση αποτελεί το γεγονός ότι για τις αυτές τις μετρικές (Pending Tasks και Pending Cost) παρατηρείται αισθητή βελτίωση για τα σενάρια όπου η επιτάχυνση είναι μεγαλύτερη. Ο αριθμός των Pending Tasks και ως αποτέλεσμα και του Pending Cost, παρατηρούμε ότι μειώνεται καθώς αυξάνεται η επιτάχυνση.

Στο τελευταίο στάδιο της Πειραματικής Αξιολόγησης όπου συγκρίθηκε ο αλγόριθμος με και χωρίς σφάλματα προέκυψε η εξής παρατήρηση:

Ο αλγόριθμος χωρίς την παρουσία σφαλμάτων παρουσιάζει παρόμοιες και σχεδόν τις ίδιες επιδόσεις με τα σενάρια στα οποία η πιθανότητα να παραμείνει ένας κόμβος ενεργός είναι 90% και 95% τα οποία και μπορούν να θεωρηθούν ως τα πιο ρεαλιστικά. Σε γενικές γραμμές, μπορούμε να πούμε ότι ο αλγόριθμος είναι ανεκτικός σε σφάλματα

κατάρρευσης και αποδίδει τόσο καλά στην παρουσία σφαλμάτων όσο και στην απουσία τους.

Ωστόσο, η αποτελεσματικότητα και η βελτιστότητα αλγορίθμου όπως προαναφέρθηκε, προϋποθέτει την ύπαρξη της επιτάχυνσης, άρα μεγάλο υπολογιστικό κόστος. Εύκολα συμπεραίνουμε ότι η ύπαρξη του κοινόχρηστου Repository, μειώνει κατά πολύ το υπολογιστικό κόστος και το κόστος μεταφοράς πληροφοριών αφού οι κόμβοι δεν επικοινωνούν μεταξύ τους παρά μόνο με το Repository. Δεν πρέπει όμως να παραληφθεί ότι το Repository αποτελεί μοναδικό σημείο αποτυχίας (Single Point Of Failure).

Τα γενικά συμπεράσματα που προέκυψαν μέσω της πειραματικής αξιολόγησης είναι ότι ο αλγόριθμος είναι αποτελεσματικός, αποδοτικός και ανεκτικός σε σφάλματα όχι μόνο στη θεωρία αλλά και στην πράξη.

Κεφάλαιο 6

Συμπεράσματα

6.1 Γενικά συμπεράσματα	84
6.2 Προβλήματα Υλοποίησης και Αντιμετώπιση	85
6.3 Μελλοντική Εργασία	86

6.1 Γενικά συμπεράσματα

Το πρόβλημα της συνεργασίας κατανεμημένων διεργασιών ούτως ώστε να επιτευχθεί η εκτέλεση μεγάλων υπολογιστικών συνόλων εργασιών, έχει απασχολήσει πολλούς ερευνητές οι οποίοι προσπάθησαν να κατασκευάσουν διάφορους αλγόριθμους που να επιλύουν το πρόβλημα και ταυτόχρονα να είναι αποδοτικοί. Για να είναι αποτελεσματική αυτή η συνεργασιακή εκτέλεση εργασιών πρέπει το σύστημα να είναι σχεδιασμένο έτσι ώστε να είναι ανθεκτικό σε σφάλματα δικτύου και διεργασιών.

Στην εργασία αυτή υλοποιήθηκε και αξιολογήθηκε ο Αλγόριθμος (n,β) -LIS ο οποίος επιλύει μια εκδοχή του προβλήματος εκτέλεσης κατανεμημένων μη-ομοιόμορφων εργασιών, όπου οι εργασίες εισάγονται δυναμικά στο σύστημα και οι διεργασίες υπόκεινται σε σφάλματα κατάρρευσης και επανεκκίνησης.

Εκτός της γενικής αντίληψης γύρω από τον αλγόριθμο που αποκτήθηκε και της κατανόησης του γενικότερου προβλήματος, στην διπλωματική αυτή εκπληρώθηκε ο στόχος της εργασίας, ο οποίος ήταν να διερευνήσει κατά πόσο ο Αλγόριθμος (n,β) -LIS είναι αποτελεσματικός, αποδοτικός και ανεκτικός σε σφάλματα όχι μόνο στη θεωρία αλλά και στην πράξη. Επιπλέον η πειραματική αξιολόγηση μας έδωσε τη δυνατότητα

να μελετήσουμε τη συμπεριφορά του αλγορίθμου κάτω από παραμέτρους που δεν ήταν εφικτό να μελετηθούν με την θεωρητική ανάλυση.

Καταλήξαμε στο συμπέρασμα ότι ο αλγόριθμος είναι αποτελεσματικός, αποδοτικός και ανεκτικός σε σφάλματα όχι μόνο στη θεωρία αλλά και στην πράξη.

Καθώς όμως οι παράμετροι του κάθε αλγορίθμου επιλέγονται πάντα ανάλογα με τις προδιαγραφές και τους περιορισμούς του συστήματος που θα εφαρμοσθεί, από την πειραματική αξιολόγηση μπορεί να διαφανεί η ανταλλαγή μεταξύ της ενέργειας (με τη μορφή της επιτάχυνσης των επεξεργασιών), της βελτιστότητας και της ανοχής σφαλμάτων του συστήματος. Κανένας αλγόριθμος δεν μπορεί να προσφέρει τα πάντα χωρίς το ανάλογο κόστος.

Με λίγα λόγια, η πειραματική αξιολόγηση αναδεικνύει τα ανταλλάγματα μεταξύ απόδοσης, υπολογιστικού κόστους και παρουσίας σφαλμάτων, γεγονός που επιτρέπει την επιλογή των κατάλληλων παραμέτρων σε κάθε εφαρμογή.

6.2 Προβλήματα Υλοποίησης και Αντιμετώπιση

Στα πλαίσια της εκπόνησης της εργασίας αυτής παρουσιάστηκαν διάφορες δυσκολίες τόσο στην υλοποίηση του αλγορίθμου με χρήση της βιβλιοθήκης YALPS όσο και στην προσομοίωση και εφαρμογή του αλγορίθμου στο δικτυακό σύστημα PlanetLab οι οποίες ξεπεράστηκαν αλλά χρειάστηκαν αρκετό χρόνο.

Οι πρώτες δυσκολίες που αντιμετώπισα ήταν στην αυτοδίδακτη εκμάθηση της βιβλιοθήκης YALPS λόγω της αρχικής εκδοχής που βρίσκεται ακόμη η βιβλιοθήκη. Πρέπει να αναφέρουμε ότι η τεκμηρίωση που υπάρχει για τη βιβλιοθήκη είναι πρόχειρη και ελάχιστη, μέρη της οποίας βρίσκονται και στη διπλωματική αυτή. Η βιβλιοθήκη YALPS περιέχει εργαλεία τα οποία δίνουν τη δυνατότητα δημιουργίας ενός ολοκληρωμένου κατανεμημένου συστήματος, αλλά λόγω έλλειψης τεκμηρίωσης είναι πολύ δύσκολο να εφαρμοστούν.

Το επόμενο πρόβλημα που αντιμετώπισα ήταν κατά την εφαρμογή και πειραματική αξιολόγηση του αλγορίθμου στο PlanetLab. Όταν έπρεπε να τρέξει ο αλγόριθμος στο PlanetLab παρουσιάστηκε ένα πρόβλημα με την παραλαβή μηνυμάτων. Ενώ οι κόμβοι εγκαθίδρυναν ορθά τη σύνδεσή τους με το Repository και έστελλαν κανονικά τα μηνύματα προς αυτό, το Repository δεν έδειχνε να λάμβανε τα μηνύματα αυτά. Γι' αυτό το λόγο, φοιτητής του τμήματος της Πληροφορικής του Πανεπιστημίου Κύπρου, ονόματι Θεοφάνης Χατζηστασή, επικοινωνήσε και ζήτησε βοήθεια από ένα μέλος της ερευνητικής ομάδας του ιδρύματος INRIA [7]. Το πρόβλημα τελικά επιλύθηκε όταν το μέλος αυτό έστειλε τη νέα εκδοχή της βιβλιοθήκης YALPS.

Ένα άλλο πρόβλημα που προέκυψε ήταν ότι λόγω της φύσης του αλγορίθμου, το Repository έπρεπε να αποθηκεύει μεγάλο όγκο δεδομένων, την λίστα με τις εργασίες που εκκρεμούν και τα κόστη τους. Έτσι, αν επιλεγόταν σε κάθε κύκλο να προσθέτεται πολύ μεγάλος αριθμός εργασιών τότε με το πέρας του αλγορίθμου το Repository αντιμετώπιζε προβλήματα μνήμης. Αυτό επιλύθηκε με την επιλογή να προσθέτονται σε κάθε κύκλο εργασίες στο εύρος [1000..1200]. Επίσης, λόγω του ότι όλοι οι κόμβοι Clients επικοινωνούσαν αποκλειστικά με το Repository, ο μεγαλύτερος αριθμός κόμβων που έγινε κατορθωτό να χρησιμοποιηθούν ήταν 128, αφού για μεγαλύτερο αριθμό κόμβων το Repository δεν μπορούσε να ανταπεξέλθει στα συνεχόμενα και πολλαπλά μηνύματα που λάμβανε. Όμως παρόλο που η αξιολόγηση του αλγορίθμου έγινε κάτω από τέτοιους περιορισμούς, εντούτοις δεν εμπόδιζε καθόλου την εξαγωγή συμπερασμάτων ως προς την απόδοσή του. Βέβαια, αν μπορούσαν να χρησιμοποιηθούν περισσότεροι κόμβοι και με περισσότερες εργασίες, σίγουρα τα αποτελέσματα της αξιολόγησης θα ήταν πιο ακριβή και πιο γενικά.

6.3 Μελλοντική Εργασία

Οι στόχοι της εργασίας γενικά έχουν επιτευχθεί. Όπως αναφέρθηκε πιο πάνω ο μόνος περιορισμός ήταν στον αριθμό των κόμβων που ήταν δυνατό να χρησιμοποιηθούν για την αξιολόγηση του αλγορίθμου. Ο μέγιστος αριθμός κόμβων που χρησιμοποιήθηκαν για την πειραματική αξιολόγηση ήταν 128. Μελλοντική εργασία μπορεί να εφαρμόσει

και να αξιολογήσει τον αλγόριθμο για μεγαλύτερο αριθμό κόμβων ώστε να, μελετηθεί ο αλγόριθμος και σε μεγαλύτερο δίκτυο.

Για την πειραματική αξιολόγηση του αλγορίθμου χρησιμοποιήθηκαν μετρικές αξιολόγησης όπως ο χρόνος διεκπεραίωσης, ο αριθμός μηνυμάτων και ο αριθμός των Pending Tasks και Pending Cost. Είναι δυνατό σε μετέπειτα στάδιο να αξιολογηθεί ο αλγόριθμος χρησιμοποιώντας περισσότερες μετρικές όπως είναι ο χρόνος που παρέμειναν ανενεργοί κάποιοι κόμβοι στην παρουσία σφαλμάτων, οι κύκλοι επεξεργαστών όπου επεξεργαστές παρέμειναν ανενεργοί ή ακόμα και το ποσοστό του πλεονασμού, δηλαδή ο αριθμός των εργασιών που εκτελέστηκε από περισσότερους από ένα κόμβους. Θα ήταν μάλιστα ενδιαφέρον να μελετηθεί κατά πόσο μπορεί να μειωθεί το υπολογιστικό κόστος του αλγορίθμου χωρίς να επηρεαστεί η ορθότητά του.

Τέλος, στην εργασία στην οποία βασίστηκε η διπλωματική αυτή [5], αναφέρεται επίσης ακόμα ένας αλγόριθμος που επιλύει το πρόβλημα το οποίο μελετήσαμε. Θα μπορούσε μελλοντικά να μελετηθεί και να υλοποιηθεί και αυτός ο αλγόριθμος ούτως ώστε να γίνει σύγκριση των δύο αλγορίθμων και να διαφανεί υπό ποιες συνθήκες είναι κατάλληλη χρήση του καθενός.

Βιβλιογραφία

- [1] Y. Emek, M. M. Halldorsson, Y. Mansour, B. Patt-Shamir, J. Radhakrishnan and D. Rawitz, “Online set packing and competitive scheduling of multi-part tasks”. In Proceedings of the 29th Annual ACM Symposium of Principles of Distributed Computing, New York, NY, USA, 2010.
- [2] C. Georgiou and A. Shvartsman, Do-All Computing in Distributed Systems: Cooperation in the Presence of Adversity, Springer, 2008.
- [3] P. C. Kanellakis and A. A. Shvartsman, Fault-Tolerant Parallel Computation, Norwell, MA, USA: Kluwer Academic Publishers, 1997.
- [4] C. Georgiou and D. R. Kowalski, “Performing dynamically injected tasks on Processes prone to crashes and restarts”. Proceedings of the 25th international conference on Distributed Computing, Berlin, Heidelberg, 2011.
- [5] A.F. Anta, C. Georgiou, D.R. Kowalski, and E. Zavou, “Online parallel scheduling of non-uniform tasks: trading failures for energy” Berlin, Heidelberg, 2013.
- [6] “YALPS: Your Applications Like Planetlab & Simulation,” INRIA, [Online]. Available: <http://yalps.gforge.inria.fr/>
- [7] “INRIA - Inventors for the digital world,” [Online]. Available: <http://www.inria.fr>
- [8] “Java Website,” Oracle, [Online]. Available: <http://www.java.com>
- [9] Πλατφόρμα PlanetLab, [Online]. Available: <http://www.planet-lab.eu/>
- [10] H. L. Chan, J. Edmonds, and K. Pruhs. Speed scaling of processes with arbitrary speedup curves on a multiprocessor. In Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2009), pages 1–10, 2009.
- [11] S. Albers and A. Antoniadis. Race to idle: New algorithms for speed scaling with a sleep state. In Proceedings of the 23rd ACM-SIAM Symposium on Discrete Algorithms (SODA 2012), pages 1266–1285, 2012.

- [12] Α. Ανδρέου, « Υλοποίηση και Πειραματική Αξιολόγηση Εργασιοκεντρικών Κατανεμημένων Αλγορίθμων στην Πλατφόρμα Yalps», Μεταπτυχιακή Διατριβή, Τμήμα Πληροφορικής, Πανεπιστήμιο Κύπρου, 2012.
- [13] D. D. Sleator and R. E. Tarjan, “Amortized efficiency of list update and paging rules,” Commun. ACM, vol. 28, no. 2, pp. 202-208, 1985.

Παράρτημα Α

Πηγαίος Κώδικας Αλγορίθμου (n,β)-LIS

```
import java.io.IOException;
import java.net.UnknownHostException;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.Random;
import java.util.Set;
import yalps.NodeID;
import yalps.Task;
import yalps.executor.E_NodeID;
import yalps.launchers.AbstractYalpsNode;

public class MyLIS extends AbstractYalpsNode {
    Random rand;
    public int id; // the id of node
    public int counter1 = 0; // counter for measurements rounds
    public int counter2 = 0; // counter for clients rounds
    public int counter3 = 0; // counter for repository rounds
    public int timeDelay = 0; // random delay
    public int task_id = 1; // the last unique id given to a task
    public int cmin = 10000; // minimum duration of a task
    public int cmax = 16000; // maximum duration of a task
    public float b = (float) cmin / cmax; // b = cmin / cmax
    public int rank = 0; // ranking the tasks
    public int n = 32; // number of nodes(only clients)
    public int old = 0; // calculation of new tasks' duration
    public double speedup = 1 + Math.sqrt(1 - cmin / cmax); // speedup
    public boolean nodeIsDown = false;
```



```

public int RestartChance = 50;
public int StayUpChance = 95;
// calculation of time
public static long start_time = System.currentTimeMillis();
public long end_time = 0;
// array of destination nodes
protected ArrayList<E_NodeID> DestinationList = new
ArrayList<E_NodeID>();
public int my_id; // getting the node id from conf file
public int my_port; // getting the node port from conf file
public int rounds = 20; // rounds of clients
public int rounds2 = 20; // rounds of repository
public int succesful_tasks = 0; // counter of succesful tasks
// Arraylist with pending tasks
public ArrayList<Integer> Pending = new ArrayList<Integer>();
// Arraylist with pending tasks durations
public ArrayList<Integer> Duration = new ArrayList<Integer>();
public int task_counter = 0; // counter for tasks that a client performed
// counter for total duration of tasks that client performed
public int duration_counter = 0;
public int msg_counter = 0; // counter for msgs that one node sent
public int pending_cost = 0; // counter for the pending cost
// Arraylist with Pending tasks for measurements
public ArrayList<Integer> Pending_Tasks = new ArrayList<Integer>();
// Arraylist with Pending cost for measurements
public ArrayList<Integer> Pending_Cost = new ArrayList<Integer>();

// method that decides if the node is going down based on a random chance
public boolean NodeGoesDown(int StayUpChance) {
    int value;
    boolean answer = false;
    Random random = new Random();

```

```

        // get the range, casting to long to avoid overflow problems
        long range = (long) cmax - (long) cmin + 1;
        // compute a fraction of the range, 0 <= frac < range
        long fraction = (long) (range * random.nextDouble());
        value = (int) (fraction + cmin);

        if (value < StayUpChance)
            answer = true;
        return answer;
    }

// method that decides if the node is restarting based on a random chance
public boolean NodeRestarts(int RestartChance) {
    int value;
    boolean answer = false;
    Random random = new Random();
    // get the range, casting to long to avoid overflow problems
    long range = (long) cmax - (long) cmin + 1;
    // compute a fraction of the range, 0 <= frac < range
    long fraction = (long) (range * random.nextDouble());
    value = (int) (fraction + cmin);

    if (value < RestartChance)
        answer = true;
    return answer;
}

// method that generates a random duration for one task between [cmin,cmax]
public int Random_duration(int cmin, int cmax) {
    int value;
    Random random = new Random();
    // get the range, casting to long to avoid overflow problems

```

```

        long range = (long) cmax - (long) cmin + 1;
        // compute a fraction of the range, 0 <= frac < range
        long fraction = (long) (range * random.nextDouble());
        value = (int) (fraction + cmin);
        return value;
    }

    // method that generates a random number of tasks between [min,max]
    public ArrayList<Integer> Random_Tasks(ArrayList<Integer> L, int min,
        int max) {
        int number; // number of new tasks to add
        Random random = new Random();
        // get the range, casting to long to avoid overflow problems
        long range = (long) max - (long) min + 1;
        // compute a fraction of the range, 0 <= frac < range
        long fraction = (long) (range * random.nextDouble());
        number = (int) (fraction + min);
        // adding the tasks to the list
        for (int i = 0; i < number; i++) {
            L.add(task_id);
            // increase the task_id to get a unique id task every time
            task_id++;
        }
        return L;
    }

    // initializing the repository
    public void initialize_repository() throws IOException {
        getOut().println("INITIALIZING REPOSITORY\n");
        getOut().println("-----" + '\n');
        MyRepository MyTask1 = new MyRepository();
        tm.registerTask(MyTask1, tm.getSystemTaskGroup(), 0);
    }
}

```

```

// initializing the clients with a random delay
public void initialize_client() throws IOException {
    getOut().println("INITIALIZING CLIENT\n");
    getOut().println("-----" + '\n');
    MyClients MyTask2 = new MyClients();
    rand = new Random();
    timeDelay = rand.nextInt(1000) + 2000;
    tm.registerTask(MyTask2, tm.getSystemTaskGroup(), timeDelay);
}

// initializing the task that keeps the measurements
public void initialize_measurements() throws IOException {
    getOut().println("MEASUREMENTS\n");
    getOut().println("-----" + '\n');
    MyMeasurements myTask3 = new MyMeasurements();
    tm.registerTask(myTask3, tm.getSystemTaskGroup(), 200);
}

// starting the nodes
@Override
public void startNode() {
    try {
        if (getNodeId().toString().equals("[194.199.68.166,51880]"))
            initialize_repository();
        else if (getNodeId().toString().equals("[130.37.193.141,51881]"))
            initialize_measurements();
        else
            initialize_client();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

```

}

// Receiving a message
@SuppressWarnings("static-access")
public boolean receive(short header, Object msg, NodeID sender) {
    // If i am the Repository
    if (getNodeId().toString().equals("[194.199.68.166,51880]")) {
        // If i received a Get message
        if (msg instanceof Get_Message) {
            getOut().println("Repository received msg Get from Node "+ sender);
            // Create message with the Pending list and send it to the node
            List_Message ml = new List_Message(Pending, Duration);
            getOut().println("Sending the Pending List to Node " + sender);
            MyTCPBasedMsgSenderTask myTask1 = new
            MyTCPBasedMsgSenderTask(ml, sender, cl);
            tm.registerTask(myTask1, tm.getSystemTaskGroup(), 0);
            // increase the number of msgs that repository sent
            msg_counter++;
            // If i received an Inform message
        } else if (msg instanceof Inform_Message) {
            Inform_Message m1 = (Inform_Message) msg;
            // Remove the task that Node performed
            for (int counter = 0; counter < Pending.size(); counter++) {
                if (Pending.get(counter) == m1.task) {
                    // track the task in the list
                    getOut().println("Removed task " + m1.task+ " from Pending
                    list");
                    Pending.remove(counter); // delete the task
                    Duration.remove(counter); // and its duration
                    succesful_tasks++;
                }
            }
        }
    }
}

```

```

    }

    // If i am a Client and i received a List Message
} else if (!(getNodeId().toString().equals("[130.37.193.141,51881]"))
&& !(getNodeId().toString().equals("[194.199.68.166,51880]"))) {
    if (nodeIsDown == true && NodeRestarts(RestartChance) == true) {
        nodeIsDown = false;
    } else if (nodeIsDown == false && NodeGoesDown(StayUpChance) == false) {
        if (counter2 < rounds) {
            List_Message l = (List_Message) msg;
            getOut().println("Node " + getNodeId()+ " received the Pending List
                from Repository ");
            id = my_id;
            // if there are tasks in the list
            if (l.pending.size() >= 1) {
                // choose which task i will perform
                rank = (((int) (id * b * n)) % l.pending.size());
                getOut().println("Node " + getNodeId()+ " will perform task " +
                    l.pending.get(rank)+ " with duration: " + l.duration.get(rank));
                // Delay until i perform the task
                try {
                    Thread.currentThread().sleep((int) (l.duration.get(rank) / speedup));
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                task_counter++; // increase the task counter
                // and increase the total duration
                duration_counter += l.duration.get(rank);

                // Create an Inform msg and send it to the Repository
                Inform_Message inform = new Inform_Message(l.pending.get(rank));
                getOut().println("Node "+ getNodeId()+ " sending Inform Message to

```

```

Repository");
MyTCPPBasedMsgSenderTask myTask1 = new
MyTCPPBasedMsgSenderTask(inform, sender, cl);
tm.registerTask(myTask1, tm.getSystemTaskGroup(), 0);
msg_counter++; // increase the number of msgs sent
} else {
    // The pending list is empty
    getOut().println("Pending list is empty!!");
}

// Run the client again
MyClients MyTask2 = new MyClients();
tm.registerTask(MyTask2, tm.getSystemTaskGroup(), 0);
counter2++; // increase the number of rounds
if (counter2 == rounds) {
    // I finished the rounds
    end_time = System.currentTimeMillis() - start_time;
    getOut().println();
    getOut().println("NODE " + getId() + "finished");
    getOut().println("XRONOS gia NODE " + getId() + " : "+
    end_time);
    getOut().println("NODE " + getId()+ "totally performed tasks "+
    task_counter);
    getOut().println("NODE " + getId()+ " perfomed tasks with totally
    cost "+ duration_counter);
    getOut().println("NODE " + getId()+ " totally sent messages "+
    msg_counter);
    getOut().println();
}
}
} else
    // i'm dead in this round

```

```

        counter2++; //increase the number of rounds
    }

    // Measurements
    if (getNodeId().toString().equals("[130.37.193.141,51881]")) {
        // If i am not done with the rounds
        if (counter1 < rounds) {
            List_Message l1 = (List_Message) msg;
            // add the pending tasks to the list
            Pending_Tasks.add(l1.pending.size());
            pending_cost = 0;
            // calculate the pending cost
            for (int i = 0; i < l1.pending.size(); i++) {
                pending_cost += l1.duration.get(i);
            }
            Pending_Cost.add(pending_cost);
            counter1++;

            // run the measurements again
            MyMeasurements MyTask3 = new MyMeasurements();
            rand = new Random();
            timeDelay = 14000;
            tm.registerTask(MyTask3, tm.getSystemTaskGroup(), timeDelay);

            if (counter1 == rounds) { // if i completed the rounds
                for (int i = 0; i < Pending_Tasks.size(); i++) {
                    getOut().print("Pending Tasks: ");
                    getOut().println(Pending_Tasks.get(i));
                    getOut().print("Pending Cost: ");
                    getOut().println(Pending_Cost.get(i));
                    getOut().println();
                }
            }
        }
    }

```



```

        }
    }
}

return true;
}

// The Clients
public class MyClients implements Task {

    MyClients() {
    };

    @Override
    public void run() {
        if (nodeIsDown == true && NodeRestarts(RestartChance) == true) {
            nodeIsDown = false;
        } else if (nodeIsDown == false && NodeGoesDown(StayUpChance) ==
false) {
            // create a Get message and send it to the Repository
            Get_Message m = new Get_Message(id);
            for (E_NodeID neighbor : DestinationList) {
                if (neighbor.getNodeId().toString()
.equals("[194.199.68.166,51880]")) {
                    getOut().println("Node " + getNodeId()
+ " sending message Get to Repository");
                    MyTCPBasedMsgSenderTask myTask1 = new
MyTCPBasedMsgSenderTask(m, neighbor, cl);
                    tm.registerTask(myTask1,
tm.getSystemTaskGroup(), 0);
                    msg_counter++;
                }
            }
        }
    }
}

```

```

    }

    }

}

```

// The Repository

public class MyRepository implements Task {

```

    MyRepository() {
    };

```

@Override

```

public void run() {
    // Create randomly new tasks and add them to the list
    old = task_id;
    getOut().println("REPOSITORY ADDS NEW TASKS TO LIST");
    Pending = Random_Tasks(Pending, 1000, 1200);
    // Create random durations
    for (int i = 0; i < task_id - old; i++) {
        Duration.add(Random_duration(cmin, cmax));
    }
    counter3++;
    // If am not done with the rounds
    if (counter3 < rounds2) {
        // run repository again
        tm.registerTask(this, tm.getSystemTaskGroup(), 15000);
    } else {
        // If i completed the rounds
        end_time = System.currentTimeMillis() - start_time;
        getOut().println("TIME FOR NODE " + getNodeId() + " :
" + end_time);
    }
}

```

```

        getOut().println();
        getOut().println("NODE " + getNodeId() + " finished!!");
        getOut().println("NODE " + getNodeId() + "totally sent
        messages "+ msg_counter);
        getOut().println("Succesful Tasks: " + succesful_tasks);
        getOut().println();
    }
}
}

```

// Measurements

```

public class MyMeasurements implements Task {

    MyMeasurements() {
    };

    @Override
    public void run() {
        // Create a Get message and send it to the Repository
        Get_Message m = new Get_Message(id);
        for (E_NodeID neighbor : DestinationList) {
            if (neighbor.getNodeId().toString()
                .equals("[194.199.68.166,51880]")) {
                MyTCPBasedMsgSenderTask myTask1 = new
                MyTCPBasedMsgSenderTask(m, neighbor, cl);
                tm.registerTask(myTask1,
                tm.getSystemTaskGroup(), 0);
            }
        }
    }
}
}

```

```

// The set of nodes
private Set<NodeID> nodes = new HashSet<NodeID>();

// The nodelist
public void setNodeList(String nodeList) {
    String[] nodeStrings = nodeList.split(";");
    for (String node : nodeStrings) {
        try {
            nodes.add(cl.getNodeIdFromString(node));
        } catch (UnknownHostException e) {
            e.printStackTrace();
        }
    }
}

// The destination list
public void setDestination(String destinationNode) {
    try {

        String[] nodeStrings = destinationNode.split(";");
        for (String node : nodeStrings)
            DestinationList.add((E_NodeID)
                cl.getNodeIdFromString(node));

    } catch (UnknownHostException e) {
        e.printStackTrace();
    }
}

// Getting the id from conf file
public void setMyid(int id) {
    System.out.println("My ID is: " + id);
}

```

```

        my_id = id;
    }

    // Getting the port from conf file
    public void setMyport(int port) {
        System.out.println("My port is: " + port);
        my_port = port;
    }
}

public class Get_Message implements Serializable {

    private static final long serialVersionUID = 1L;

    int m_get;

    Get_Message(int m_get) {

        this.m_get = m_get;
    }
}

public class Inform_Message implements Serializable {

    private static final long serialVersionUID = 1L;

    int task;

    Inform_Message(int task) {

        this.task = task;
    }
}

public class List_Message implements Serializable {

    private static final long serialVersionUID = 1L;

    // ArrayList with Pending tasks
    public ArrayList<Integer> pending = new ArrayList<Integer>();

    // ArrayList with Duration
    public ArrayList<Integer> duration = new ArrayList<Integer>();
}

```

```

        @SuppressWarnings("unchecked")
        List_Message(ArrayList<Integer> pending, ArrayList<Integer> duration) {
            this.pending = (ArrayList<Integer>) pending.clone();
            this.duration = (ArrayList<Integer>) duration.clone();
        }
    }
}

```

```

public class MyLIS_starter {
    @SuppressWarnings("static-access")
    public static void main(String[] args) {
        ExecutorNodeStarter starter = new ExecutorNodeStarter();
        starter.start(args[0]);
    }
}

```

```

public class MyTCPBasedMsgSenderTask implements Task {
    E_NodeID destination;
    RealWorldCommunicationLayer commLayer;
    Get_Message Get = null;
    Inform_Message Inform = null;
    List_Message List = null;

    public MyTCPBasedMsgSenderTask(Get_Message msg1, NodeID dest,
        CommunicationLayer comm) {
        this.Get = msg1;
        this.destination = (E_NodeID) dest;
        this.commLayer = (RealWorldCommunicationLayer) comm;
    }
}

```

```

public MyTCPBasedMsgSenderTask(Inform_Message msg2, NodeID dest,
    CommunicationLayer comm) {
    this.Inform = msg2;
    this.destination = (E_NodeID) dest;
    this.commLayer = (RealWorldCommunicationLayer) comm;
}

```

```

public MyTCPBasedMsgSenderTask(List_Message msg2, NodeID dest,
    CommunicationLayer comm) {
    this.List = msg2;
    this.destination = (E_NodeID) dest;
    this.commLayer = (RealWorldCommunicationLayer) comm;
}

```

```

@Override
public void run() {
    if (Get != null) { // msg to send is Serializable
        try {
            commLayer.sendTCP(Get, destination);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    if (Inform != null) { // msg to send is Serializable
        try {
            commLayer.sendTCP(Inform, destination);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    if (List != null) { // msg to send is Serializable

```

```
        try {  
            commLayer.sendTCP(List, destination);  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
  
}  
  
}
```


Παράρτημα Β

Αποτελέσματα Εκτελέσεων

Τα αρχεία των αποτελεσμάτων βρίσκονται στον ψηφιακό δίσκο που συνοδεύει την διπλωματική αυτή. Κάτω από τον φάκελο «Analysis» βρίσκεται το αρχείο τύπου Excel το οποίο περιέχει τα αποτελέσματα των εκτελέσεων και τις γραφικές που χρησιμοποιήθηκαν για την πειραματική αξιολόγηση.

Επίσης, μερικά δείγματα από τα αρχεία διαμόρφωσης βρίσκονται κάτω από τον φάκελο «Configuration Files» ενώ δείγματα από τα αρχεία καταμέτρησης βρίσκονται κάτω από το φάκελο «Log Files».