Ατομική Διπλωματική Εργασία

ΜΕΛΕΤΗ, ΥΛΟΠΟΙΗΣΗ ΚΑΙ ΠΕΙΡΑΜΑΤΙΚΗ ΑΞΙΟΛΟΓΗΣΗ ΑΛΓΟΡΙΘΜΩΝ ΣΥΝΕΤΑΙΡΙΣΤΙΚΟΥ ΚΑΤΑΝΕΜΗΜΕΝΟΥ ΥΠΟΛΟΓΙΣΜΟΥ

Χατζηστασή Θεοφάνης

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάιος 2014

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΜΕΛΕΤΗ, ΥΛΟΠΟΙΗΣΗ ΚΑΙ ΠΕΙΡΑΜΑΤΙΚΗ ΑΞΙΟΛΟΓΗΣΗ ΑΛΓΟΡΙΘΜΩΝ ΣΥΝΕΤΕΡΙΣΤΙΚΟΥ ΚΑΤΑΝΕΜΗΜΕΝΟΥ ΥΠΟΛΟΓΙΣΜΟΥ

Χατζηστασή Θεοφάνης

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

ΕΥΧΑΡΙΣΤΙΕΣ

Θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή της Ατομικής Διπλωματικής Εργασίας, Δρ. Χρύση Γεωργίου, Αναπληρωτή Καθηγητή του τμήματος Πληροφορικής του Πανεπιστημίου Κύπρου, ο οποίος μου έδωσε την ευκαιρία να εργαστώ στην παρούσα διπλωματική εργασία. Ως ο επιβλέπων καθηγητής, μου έθεσε τις κατευθυντήριες γραμμές και μου πρόσφερε τα απαραίτητα εφόδια για την ολοκλήρωση της εργασίας αυτής, οδηγίες, συμβουλές και συνεχές στήριξη.

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

Σας Ευχαριστώ!

ΠΕΡΙΛΗΨΗ

Τόσο οι απαιτήσεις για πολύπλοκους υπολογισμούς, όσο και οι ανάγκες για επιτάχυνση έως προς την διεκπεραίωση των υπολογισμών αυτών, ειδικά στις περιπτώσεις όπου μεγάλες συλλογές από εργασίες πρέπει να εκτελεστούν, έχουν ωθήσει τα κατανεμημένα συστήματα υπολογιστών σε μια περίοδο εκρηκτικής εξέλιξης. Με τα διαθέσιμα δίκτυα υπερυπολογιστών να μας προσφέρουν ένα τεράστιο αριθμό υπολογιστών σε κατανεμημένο περιβάλλον μας δημιουργείται η ανάγκη για σχεδιασμό αλγορίθμων δυναμικού κατανεμημένου υπολογισμού. Τα μεγάλα σε έκταση κατανεμημένα συστήματα είναι εκ φύσεως δυναμικά και χαρακτηρίζονται από διαταραχές όπως σφάλματα και καταρρεύσεις υπολογιστών καθώς επίσης και προβλήματα αξιοπιστίας δικτύου. Κατά την μελέτη συνεταιριστικών αλγόριθμων, το πρόβλημα αυτό στην αφαιρετική του εκδοχή ονομάζεται *Do-All*, και ο στόχος του είναι να χρησιμοποιήσει *n* κατανεμημένες υπολογιστικές οντότητες για να εκτελέσουν *t* εργασίες στην παρουσία σφαλμάτων.

Στην παρούσα Ατομική Διπλωματική Εργασία παρουσιάζεται η μελέτη, υλοποίηση και πειραματική αξιολόγηση τριών αλγορίθμων δυναμικού κατανεμημένου υπολογισμού που επιλύουν το πρόβλημα *Do-All*. Οι συνεταιριστικοί αυτοί αλγόριθμοι είναι οι *AN, AR* και *Do-Um* όπως ορίζονται στα [2] [1] [3] αντίστοιχα. Αφού μελετήθηκαν και κατανοήθηκαν πλήρως, οι εν λόγω αλγόριθμοι υλοποιήθηκαν στη γλώσσα προγραμματισμού Java με τη χρήση της YALPS [4], μιας βιβλιοθήκης που υποβοηθά την υλοποίηση κατανεμημένων αλγορίθμων και την προσομοίωση ή εκτέλεσή τους σε πραγματικό υπολογιστικό περιβάλλον. Οι αλγόριθμοι σε πρώτο στάδιο εφαρμόστηκαν σε περιβάλλον προσομοίωσης με σκοπό να ελεγχθεί αυστηρά η ορθότητα υλοποίησης τους. Ακολούθως εφαρμόστηκαν στο PlanetLab [5], ένα πραγματικό Διαδικτυακό περιβάλλον κατανεμημένων υπολογιστών, ως ισχυρό επιχείρημα για να αξιολογηθεί και να επιβεβαιωθεί η πρακτικότητά τους.

Η εμπειρική αξιολόγηση των αλγορίθμων οδήγησε στο γενικό συμπέρασμα ότι η πειραματική τους απόδοση συνάδει με τη θεωρητική αξιολόγησή τους, στις περιπτώσεις όπου αυτή μας είναι διαθέσιμη.

ΠΙΝΑΚΑΣ ΠΕΡΙΕΧΟΜΕΝΩΝ

ΚΕΦΑΛΑΙΟ 1

Εισαγωγή

1.1 Κίνητρο

Κύριο πρόβλημα του κατανεμημένου υπολογισμού αποτελεί η συνεταιριστική εκτέλεση μεγάλων υπολογιστικών συνόλων εργασιών από κατανεμημένες υπολογιστικές οντότητες *(computing entities)*. Το σύστημα θα πρέπει να είναι ανεπτυγμένο έτσι ώστε να έχει την ικανότητα αντιμετωπίζει δυναμικές διαταραχές του μέσου επικοινωνίας, πιθανά σφάλματα των υπολογιστικών οντοτήτων ή και ακόμη επανεκκινήσεις οντοτήτων που κατέρρευσαν σε προηγούμενα στάδια του υπολογισμού. Με βάση τα προαναφερθέντα, τις τελευταίες δεκαετίες έχουν αφιερωθεί αρκετές μελέτες που στοχεύουν στην ανάπτυξη αλγοριθμικών λύσεων με ανοχή στα σφάλματα για ποικίλες εκδοχές τέτοιων συνεταιριστικών προβλημάτων [1-3, 5-8].

Στην συγκεκριμένη Διπλωματική εργασία θα μελετηθεί το γενικευμένο πρόβλημα *Do-All,* όπου χρησιμοποιώντας *n* κατανεμημένες υπολογιστικές οντότητες, σκοπός είναι να συνεργαστούν έτσι ώστε να εκτελέσουν στο σύνολο *t* εργασίες στην

παρουσία σφαλμάτων. Ο αριθμός *t* των εργασιών είναι προκαθορισμένος και γνωστός εκ των προτέρων σε κάθε υπολογιστική οντότητα.

Στα [1-3, 6-8] γίνεται περιγραφή αλγοριθμικών λύσεων οι οποίες επιλύουν διάφορες εκδοχές του προβλήματος *Do-All,* κάποιες εκ των οποίων προαπαιτούν ή όχι αξιοπιστία δικτύου και οι υπολογιστικές οντότητες που λαμβάνουν μέρος στον υπολογισμό μπορούν να υποκύψουν σε καταρρεύσεις ή και επανεκκινήσεις.

Οι αλγόριθμοι αυτοί όμως, αν και αναλύθηκαν θεωρητικά, εντούτοις δεν έχουν δοκιμαστεί σε πρακτικό επίπεδο και αυτό αποτέλεσε το βασικό κίνητρο που οδήγησε στην εκπόνηση της παρούσας εργασίας.

1.2 Στόχος Διπλωματικής Εργασίας

Η παρούσα Διπλωματική εργασία στοχεύει στην πειραματική αξιολόγηση των αλγορίθμων AN, AR και Do-Um [1-3] η οποία θα υποδείξει κατά πόσο η πειραματική τους απόδοση συνάδει με τη θεωρητική αξιολόγησή τους. Συγκεκριμένα οι αλγόριθμοι αυτοί έχουν υλοποιηθεί σε πρώτη φάση σε πλατφόρμα η οποία επιτρέπει την πειραματική εκτέλεση τους σε περιβάλλον προσομοίωσης για τον έλεγχο της ορθότητας τους. Στην συνέχεια, η πλατφόρμα μας παρείχε την δυνατότητα με κάποιες μετατροπές να εκτελέσουμε τους αλγόριθμους σε ένα πραγματικό διαδικτυακό περιβάλλον κατανεμημένων υπολογιστών. Μέσω των αποτελεσμάτων που προκύψαν από αυτές τις εκτελέσεις, μας ήταν πλέον εφικτή η μελέτη και η σύγκριση των μετρικών αξιολόγησης των αλγορίθμων σε πρακτικό επίπεδο.

Η υλοποίηση των αλγορίθμων έγινε σε γλώσσα προγραμματισμού JAVA με την χρήση της πλατφόρμας YALPS [4] του ινστιτούτου επιστήμης και τεχνολογίας INRIA [9]. Η εκτέλεση των αλγορίθμων έλαβε χώρο στο πραγματικό διαδικτυακό περιβάλλον κατανεμημένων υπολογιστών που μας προσφέρει το PlanetLab [5].

1.3 Μεθοδολογία Διπλωματικής Εργασίας

Για την επιτυχές ολοκλήρωση της εργασίας αυτής έγινε διαχωρισμός σε μικρά διακριτά στάδια. Πιο κάτω αναφέρεται η μεθοδολογία που έλαβε χώρο.

Σε αρχικό στάδιο έλαβε χώρο η μελέτη και εκμάθηση της πλατφόρμας YALPS. Εφόσον κατανοήθηκε πλήρως η δομή και οι απαιτήσεις της πλατφόρμας σε νοητικό επίπεδο, για προσωπική εξοικείωση αναπτύχθηκαν σε αυτή διάφορα απλά παραδείγματα που μου προτάθηκαν από τον επιβλέπων καθηγητή Δρ. Χρύση Γεωργίου. Αυτό, έτσι ώστε να έρθουν στην επιφάνεια νωρίς τυχόν προβλήματα τα οποία θα έπρεπε να επιλυθούν, καθώς επίσης να γίνει επικύρωση της νοητικής κατανόησης με την πλήρης πρακτική κατανόηση και ανάπτυξη αλγορίθμων στο συγκεκριμένο περιβάλλον. Αξίζει να αναφέρουμε ότι η βιβλιοθήκη YALPS βρίσκεται σε αρχικά στάδια (έκδοση 0.3 Alpha) και η ύπαρξη εγχειριδίου χρήσης είναι σε τυπικά επίπεδα. Επομένως, ήταν αναμενόμενο να προκύπτουν αρκετά προβλήματα κατανόησης τα οποία θα έπρεπε να αντιμετωπιστούν. Δεν γίνεται όμως αναφορά σε σπατάλη χρόνου, λόγο των πλεονεκτημάτων και δυνατοτήτων που μας παρέχει η συγκεκριμένη πλατφόρμα.

Εν συνεχεία, έγινε η εκμάθηση και εις βάθος κατανόηση των αλγορίθμων προς υλοποίηση *AN, AR* και *Do-Um* . Περισσότερη έμφαση σε αυτό το σημείο δόθηκε στην πλήρη κατανόηση της χρησιμότητας και την ακριβή λειτουργία των αλγορίθμων που προτάθηκαν, έτσι ώστε να αποφευχθούν πιθανές ασάφειες οι οποίες θα οδηγούσαν σε λάθος υλοποίηση και προφανώς λάθος αξιολόγηση.

Ακολούθως, έλαβε χώρο η υλοποίηση των αλγόριθμων *AN, AR* και *Do-Um* με την χρήση της βιβλιοθήκης YALPS και ο ενδελεχής έλεγχος επιτυχής υλοποίησης σε επίπεδο προσομοίωσης των προαναφερθέντων αλγόριθμων.

Σε επόμενο στάδιο, έγιναν οι απαραίτητες τροποποιήσεις στον κάθε ένα από τους αλγορίθμους ξεχωριστά έτσι ώστε να γίνει η μετάβαση εκτέλεσης τους από προσομοιωμένο σε πραγματικό διαδικτυακό περιβάλλον κατανεμημένων υπολογιστών.

Κατόπιν, βάσει των γνώσεων μας από την θεωρία και προηγούμενες πειραματικές αξιολογήσεις, προτάθηκαν τα διάφορα σενάρια τα οποία θα εκτελούσαμε για τον κάθε αλγόριθμο. Εφόσον εγκρίθηκαν από τον επιβλέπων καθηγητή πραγματοποιήθηκε η εκτέλεση των αλγορίθμων σε πραγματικό διαδικτυακό περιβάλλον κατανεμημένων υπολογιστών με διάφορες παραμέτρους για την ικανοποίηση των σεναρίων αυτών και έπειτα έγινε η συλλογή αποτελεσμάτων.

Συλλογή η οποία μας οδήγησε στο τελευταίο στάδιο, αυτό της πειραματικής αξιολόγησης και σύγκρισης των αναμενόμενων αποτελεσμάτων της θεωρίας με αυτά που προέκυψαν από την πειραματική αξιολόγηση.

1.4 Δομή της Διπλωματικής Εργασίας

Η παρούσα Ατομική Διπλωματική Εργασία αποτελείται συνολικά από επτά κεφάλαια. Με το πέρας του εισαγωγικού Κεφαλαίου 1, συναντάμε το Κεφάλαιο 2 όπου παρουσιάζεται το υπόβαθρο που χρειάζεται να έχει κανείς για να κατανοήσει καλύτερα το πρόβλημα *Do-All* και την υλοποίηση των δυναμικών κατανεμημένων αλγορίθμων που το επιλύουν. Συγκεκριμένα, ορίζονται οι σχετικές έννοιες και γίνεται μια περιεκτική αναφορά τόσο για τη βιβλιοθήκη YALPS που υποβοηθά στην υλοποίηση των αλγορίθμων όσο και για το διαδικτυακό κατανεμημένο περιβάλλον PlanetLab στο οποίο έλαβαν χώρο όλες οι πραγματικές εκτελέσεις των αλγορίθμων που αναπτύχθηκαν.

Στο Κεφάλαιο 3 παρουσιάζεται το μοντέλο υπολογισμού των αλγοριθμικών λύσεων που αναπτύξαμε. Πιο αναλυτικά γίνεται αναφορά στο κατανεμημένο περιβάλλον, στους γύρους, στις εργασίες, στον αντίπαλο, στις επανεκκινήσεις, στο εχθρικό σενάριο και στην ορθότητα ολοκλήρωσης των αλγορίθμων.

Εν συνεχεία, στο Κεφάλαιο 4, γίνεται παρουσίαση των αλγόριθμων AN, AR και Do-Um οι οποίοι επιλύουν το πρόβλημα Do-All σε σύγχρονα κατανεμημένα συστήματα. Δίνεται μια περιεκτική περιγραφή του κάθε αλγορίθμου και αναφέρονται

λεπτομέρειες δομής και υλοποίησης του κώδικα που δημιουργήθηκε για τον κάθε αλγόριθμο.

Ακολουθεί το Κεφάλαιο 5 όπου παρουσιάζεται η διαδικασία της πειραματικής αξιολόγησης του κάθε αλγορίθμου όταν αυτός εφαρμοστεί σε ένα διαδικτυακό περιβάλλον κατανεμημένων υπολογιστών, συγκεκριμένα στο PlanetLab. Στο τέλος παρατίθεται σχολιασμός των αποτελεσμάτων και εξάγονται συμπεράσματα που υποδεικνύουν σε ποιο βαθμό η θεωρητική ανάλυση του κάθε αλγορίθμου, εάν υπάρχει, συνάδει με την πειραματική.

Έπειτα, στο Κεφάλαιο 6 λαμβάνει χώρο η σύγκριση δύο αλγορίθμων. Συγκεκριμένα συγκρίνονται οι αλγόριθμοι AN και Do-UM. Η σύγκριση αυτή αφορά ένα σύνολο διαφόρων σεναρίων που εφαρμόστηκαν σε αυτούς. Οι μετρικές που λαμβάνουν χώρο στην κάθε σύγκριση είναι αυτή της πολυπλοκότητας εργασίας και πολυπλοκότητας μηνυμάτων.

Στο Κεφάλαιο 7 συνοψίζονται τα συμπεράσματα από την εφαρμογή των τριών αλγορίθμων σε πραγματικό κατανεμημένο περιβάλλον. Εν συνεχεία, γίνεται αναφορά σε τυχόν δυσκολίες που είχαν παρουσιαστεί κατά την εκπόνηση της εργασίας και διατυπώνονται ιδέες για μελλοντική εργασία από την εκμετάλλευση των συμπερασμάτων.

Η εργασία ολοκληρώνεται με τα παραρτήματα Α, Β, Γ και Δ. Στα αντίστοιχα πρώτα τρία παρουσιάζεται ο κώδικας υλοποίησης των αλγορίθμων AN, AR και Do-UM τόσο για πραγματικό περιβάλλον όσο για περιβάλλον προσομοίωσης. Τελικό παράρτημα είναι το Δ στο οποίο παρουσιάζονται μερικά παραδείγματα εκτέλεσης από τους τρεις αλγόριθμους.

Κεφάλαιο 2

Υπόβαθρο

2.1 Το Πρόβλημα Do-All

Η παρούσα εργασία ασχολείται με το πρόβλημα *Do-All*. Όπως έγινε αναφορά και σε προηγούμενο στάδιο της εργασίας το πρόβλημα *Do-All [1],* βάσει αυτού στην αφηρημένη του εκδοχή, στοχεύει στην συνεργασία *n* κατανεμημένων υπολογιστικών οντοτήτων, έτσι ώστε να εκτελέσουν στο σύνολο *t* εργασίες στην παρουσία σφαλμάτων.

Το πρόβλημα *Do-All* θεωρείτε λυμένο όταν όλες οι εργασίες *t* εκτελεστούν, υπό την προϋπόθεση ότι υπάρχει τουλάχιστον μία υπολογιστική οντότητα *n* ενεργή σε ολόκληρη την εκτέλεση.

Όπως θα αναλύσουμε και σε μελλοντικό στάδιο, το πρόβλημα *Do-All* έχει μελετηθεί σε διάφορα μοντέλα υπολογισμού κάτω από διάφορες προϋποθέσεις σχετικά με τον συγχρονισμό, σφάλματα, αποτυχίες/αστοχίες και αξιοπιστία δικτύου.

Όπως για κάθε αξιολόγηση κατανεμημένου αλγορίθμου, έτσι και για την αξιολόγηση των αλγορίθμων που επιλύουν το πρόβλημα Do-All, θα παρουσιάσουμε τις κύριες μετρικές που λαμβάνονται υπόψη. Συγκεκριμένα για την επίλυση του προβλήματος στο μοντέλο Ανταλλαγής Μηνυμάτων οι μετρικές αυτές είναι:

- Πολυπλοκότητα Εργασίας *(Work Complexity)* όπου καθορίζεται έως ο συνολικός αριθμός των βημάτων που έχουν εκτελεστεί από κάθε υπολογιστική οντότητα μέχρι την ολοκλήρωση της εκτέλεσης. Στην επίλυση του προβλήματος *Do-all* αυτό ισοδυναμεί με τον συνολικό αριθμό εκτελέσεων που έτυχε η κάθε εργασία.
- Πολυπλοκότητα Επικοινωνίας *(Communication Complexity)* όπου καθορίζεται έως ο συνολικός αριθμός μηνυμάτων σημείου προς σημείου που έχουν αποσταλεί στην διάρκεια μίας εκτέλεσης.
- Πολυπλοκότητα Χρόνου *(Time Complexity),* εδώ χρησιμοποιούμε την συμβατική έννοια του χρόνου έτσι ώστε να είμαστε σε θέση μετρήσουμε των αριθμών εντολών που εκτελεί κάθε επεξεργαστής, η σε σύγχρονα μοντέλα τον αριθμό επαναλήψεων του αλγόριθμου μέχρι την επίλυση του προβλήματος.

2.2 Προηγούμενες Εργασίες

Το πρόβλημα *Do-All* έχει μελετηθεί σε διάφορα μοντέλα υπολογισμού κάτω από διάφορες προϋποθέσεις σχετικά με τον συγχρονισμό, σφάλματα, αποτυχίες/αστοχίες και αξιοπιστία δικτύου. Βασικότερα μοντέλα υπολογισμού που λαμβάνει χώρο το πρόβλημα είναι τα ακόλουθα:

- Μοντέλο Μεταβίβασης Μηνυμάτων *(Message-Passing)* [1-3, 10] κατά το οποίο όλες οι υπολογιστικές οντότητες επικοινωνούν μεταξύ τους με ανταλλαγή μηνυμάτων για την διεκπεραίωση των εργασιών.

- Μοντέλο Κοινόχρηστης Μνήμης *(Shared-Memory)* [11-13] όπου κάθε υπολογιστική οντότητα έχει πρόσβαση σε μία κοινόχρηστη μνήμη στην οποία βρίσκεται και αναβαθμίζεται η απαιτούμενη γνώση. Η επικοινωνία εδώ επιτυγχάνεται με ανάγνωση και εγγραφή στην μνήμη.
- Μοντέλο Διαμερισμένων Δικτύων *(Partitionable Networks)* [14] βάση αυτού, ομάδες υπολογιστικών οντοτήτων αποσυνδέονται επικοινωνιακά μεταξύ τους και πιθανών επανασυνδέονται κατά την διάρκεια του κατανεμημένου υπολογισμού.
- Μοντέλο σε απουσία επικοινωνίας *(In Absence of Communication)* [15] όπου εξετάζεται κατά πόσο είναι εφικτό να υπάρξει συνεταιριστική εκτέλεση εργασιών στην περίπτωση όπου η επικοινωνία διακοπεί για μεγάλο χρονικό διάστημα.

Αλγόριθμοι οι οποίοι επιλύουν το πρόβλημα Do-All είχαν αρχικά προταθεί από τους Dwork, Halpern και Waarts [10], από τους De Prisco, Mayer και Yung [16], και από τούς Galil, Mayer και Yung [17]. Αυτοί όμως οι ντετερμινιστικοί αλγόριθμοι έχουν διατυπωθεί για μοντέλα τα οποία επιτρέπουν καταρρεύσεις επεξεργαστών αλλά όχι επανεκκινήσεις. Επίσης, ο σχεδιασμός των αλγορίθμων αυτών προνοεί σε κάθε βήμα να υπάρχει μόνο ένας συντονιστής, έτσι ώστε η πολυπλοκότητα επικοινωνίας να βρίσκεται σε χαμηλά επίπεδα.

Στην εργασία των Chlebus, De Prisco και Shvartsman [6], στην οποία βασίζονται οι δύο εκ των τριών αλγόριθμων που μελετάμε, AN και AR, αναπτύσσονται καινούργιοι αλγόριθμοι οι οποίοι εξυπηρετούν άλλες απαιτήσεις. Πιο αναλυτικά αυτοί οι αλγόριθμοι, σε αντίθεση με αυτούς που προαναφέρθηκαν, κάνουν χρήση της στρατηγικής πολλαπλών συντονιστών και προαπαιτούν αξιόπιστη πολυεκπομπή μηνυμάτων. Αυτό εξυπηρετεί έτσι ώστε να απλοποιήθει τόσο η ανάπτυξη αλγορίθμων για επεξεργαστές που δεν θα επανεκκινήσουν στο μέλλον όσο και η ανάπτυξη αλγοριθμικών λύσεων ικανών να μπορούν αποδοτικά να αντιμετωπίσουν τυχών επανεκκινήσεις επεξεργαστών.

Η εργασία των Davtyan, De Prisco, Georgiou και Shvartsman [3], είχε σκοπό την ανάπτυξη αλγόριθμου ικανού να επιλύει αποδοτικά το πρόβλημα Do-All στο μοντέλο

ανταλλαγής μηνυμάτων σημείο προς σημείο *(point-to-point message-passing)*. Συγκεκριμένα από αυτή την εργασία προέκυψε ο αλγόριθμος Do-Um, ο οποίος επιλύει το πρόβλημα Do-All με χρήση της στρατηγικής πολλαπλών συντονιστών και χωρίς την προϋπόθεση ύπαρξης αξιόπιστης πολλαπλής διανομής.

Σε αυτή την εργασία ασχολούμαστε αποκλειστικά με αλγορίθμους AN, AR και Do-Um που προκύπτουν από τις εργασίες [6] και [3] αντίστοιχα. Αυτοί οι αλγόριθμοι ικανοποιούν το μοντέλο Μεταβίβασης Μηνυμάτων και επιπρόσθετα ο αριθμός των εργασιών *t* θα είναι σταθερός και γνωστός εκ των προτέρων σε κάθε υπολογιστική οντότητα.

2.3 Η Βιβλιοθήκη YALPS

Η βιβλιοθήκη YALPS [4] είναι μια ανοικτού κώδικα βιβλιοθήκη της Java η οποία σχεδιάστηκε πρόσφατα από μια ομάδα ερευνητών του ιδρύματος INRIA [9] με απώτερο σκοπό τη διευκόλυνση της ανάπτυξης, της δοκιμής και του ελέγχου κατανεμημένων εφαρμογών.

Η βιβλιοθήκη περιλαμβάνει τις δύο κύριες πλατφόρμες υλοποίησης. Συγκεκριμένα τον προσομοιωτή *SimulatedNodeStarter* για εκτέλεση σε περιβάλλον προσομοίωσης και τον εκτελεστή *ExecutorNodeStarter* για εκτέλεση του αλγόριθμου σε πραγματικό διαδικτυακό κατανεμημένο περιβάλλον. Επομένως, κάθε εφαρμογή που αναπτύσσεται με τη χρήση της YALPS έχει την ικανότητα να τρέξει τόσο σε περιβάλλον προσομοίωσης όσο και σε πραγματικό περιβάλλον. Αυτό, είναι εφικτό προς τον χρήστη με ένα μικρό χρονικό κόστος που αφορά την τροποποίηση σημείων του κώδικα της εφαρμογής. Μ’ αυτόν τον τρόπο δίνεται η δυνατότητα στους ερευνητές, αφού δημιουργήσουν το πρωτότυπο μιας εφαρμογής, να πειραματιστούν και να δοκιμάσουν τα διάφορα σενάριά τους στον προσομοιωτή προτού αυτά εφαρμόστουν σε πραγματικό σύστημα.

Μία εφαρμογή ανεπτυγμένη στην βιβλιοθήκη YALPS είναι οργανωμένη από μια συλλογή εργασιών *Tasks* τις οποίες διαχειρίζεται ο διαχειριστής εργασιών *Task Manager*. Ο διαχειριστής είναι υπεύθυνος να καθορίζει με ποιο τρόπο και για πόση χρονική διάρκεια θα εκτελεστεί η κάθε μια από τις εργασίες. Συνολικά, διακρίνονται τρεις τρόποι εκτέλεσης για μια εργασία:

- Άμεση εκτέλεση, κατά την οποία η εργασία εκτελείται αμέσως μετά την καταχώρησή της.
- Αναβεβλημένη εκτέλεση, κατά την οποία η εργασία δεν εκτελείται άμεσα αλλά μετά από συγκεκριμένο χρονικό διάστημα.
- Περιοδική εκτέλεση, όπου η εργασία εκτελείται περιοδικά, δηλαδή ανά τακτά χρονικά διαστήματα.

Η βιβλιοθήκη YALPS μας προσφέρει επιπλέον τη δυνατότητα οργάνωσης των εργασιών σε ομάδες *TaskGroups* με σκοπό οι εργασίες να μπορούν να τυγχάνουν διαχείρισης σαν ομάδα αντί σαν ατομικές εργασίες. Αυτό μας εξυπηρετεί στην καλύτερη διαχείριση των εργασιών.

Σημαντικό στοιχείο της YALPS είναι ο τρόπος με τον οποίο είναι εφικτή η επικοινωνία. Για την πλήρη υποστήριξη της επικοινωνίας, η YALPS μας παρέχει το δικό της στρώμα επικοινωνίας, συγκεκριμένα το *YALPS CommunicationLayer*. Λόγο των δύο κύριων πλατφορμών εκτέλεσης, το *YALPS CommunicationLayer* είναι υλοποιημένο σε δυο εκδόσεις, η κάθε μία εκ των οποίων ικανοποιεί πλήρως την αντίστοιχη πλατφόρμα. Για την εκτέλεση σε πραγματικό κατανεμημένο περιβάλλον τα μηνύματα της εφαρμογής δρομολογούνται μέσω των πρωτοκόλλων *UDP/TCP*. Σε αντίθεση με την πραγματική εκτέλεση, κατά την προσομοίωση της εφαρμογής η επικοινωνία λαμβάνει χώρο σε μια εσωτερική υποδομή της YALPS. Και στις δύο περιπτώσεις όμως μας παρέχονται πλήρως οι λειτουργίες για έλεγχο και αξιολόγηση κατανεμημένων εφαρμογών, όπως είναι η επιλογή για απώλεια μνημάτων και ο περιορισμός του εξερχόμενου εύρους ζώνης κατά την εκτέλεση αυτών.

Τα μηνύματα που ανταλλάζονται μέσω των εφαρμογών της YALPS απαιτείτε να είναι σε σειριακή μορφή, δηλαδή, μηνύματα σε κατάλληλη μορφή που να τους

επιτρέπετε η αποθήκευση ή η μετάδοσή τους μέσω μιας σύνδεσης ενός δικτύου. Η σειριοποιήση μηνυμάτων είναι εφικτή είτε μέσω των βασικών κλάσεων που προσφέρει η JAVA είτε μέσω ενός πιο αποδοτικού τρόπου που προσφέρει η YALPS.

Με βάση τα προαναφερθέντα, έχοντας διαθέσιμη την βιβλιοθήκη YALPS για την ανάπτυξη κατανεμημένων εφαρμογών, μας παρέχονται έτοιμα όλα τα αναγκαία και απαραίτητα στοιχεία μίας κατανεμημένης εφαρμογής. Στην παρούσα εργασία χρησιμοποιείται η βιβλιοθήκη YALPS και με τη βοήθειά της υλοποιούνται τρεις συγχρονισμένοι αλγόριθμοι οι οποίοι προσομοιώνονται και κατόπιν εφαρμόζονται σε πραγματικό διαδικτυακό κατανεμημένο σύστημα.

2.3.1 Λειτουργίες της YALPS

Σε επόμενο στάδιο θα αναλύσουμε τις βασικές λειτουργίες της βιβλιοθήκης YALPS. Συγκεκριμένα θα γίνει αναφορά στις διάφορες διεπαφές που χρησιμοποιήσαμε, πώς έγινε εφικτή η αποστολή και λήψη μηνυμάτων και ποία ήταν τα βασικά συστατικά για την εκτέλεση μιας κατανεμημένης εφαρμογής.

2.3.1.1 Διεπαφές της YALPS

Διεπαφή NodeID

Η δήλωση ενός κόμβου της εφαρμογής γίνεται με την δήλωση του ως αντικείμενο της διεπαφής *NodeID* χωρίς αυτό να έχει αντίκτυπο στο εάν η εφαρμογή θα τρέξει σε πραγματικό ή περιβάλλον προσομοίωσης. Οι κλάσεις που υλοποιούν την διεπαφή αυτή είναι η *E_NodeID* και *S_NodeID* για το πραγματικό και προσομοίωσης περιβάλλον αντίστοιχα. Η επιλογή της κατάλληλης κλάσης εκ των δύο για την εκτέλεση της εφαρμογής καθορίζεται από το αρχείο διαμόρφωσης.

Η αρχικοποίηση ενός κόμβου με βάση την βιβλιοθήκη YALPS έχει την εξής δομή:

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

Όπου το αντικείμενο *cl* αντιστοιχεί στο υπόστρωμα επικοινωνίας που χρησιμοποιείται από την εφαρμογή. Η συνάρτηση *getNodeIdFromString(NodeName)* θα επιστρέψει το μοναδικό αναγνωριστικό του κόμβου με το όνομα *NodeName* και η τιμή της επιστροφής θα ανατεθεί στην μεταβλητή *MyNode*.

Διεπαφή YalpsNode

Η διεπαφή *YalpsNode* υλοποιείται από την βασική αφαιρετική κλάση της βιβλιοθήκης YALPS την *AbstractYalpsNode*. Η με την σειρά της απλοποιεί την κωδικοποίηση των βασικών συστατικών μίας εφαρμογής όπως είναι η δήλωση ενός αντικειμένου *tm* τύπου *TaskManager* για τον διαχειρισμό των εργασιών ή ενός αντικειμένου *cl* τύπου *CommunicationLayer* για την διασφάλιση επικοινωνίας.

Η επέκταση της μπορεί να γίνει εφικτή ως εξής:

```
public class MyApplication extends AbstractYalpsNode{
//insert code here
}
```

Διεπαφή Task

Κάθε εργασία στην εφαρμογή είναι ένα αντικείμενο που υλοποιεί την διεπαφή *Task*. Για να θεωρείται ορθή η υλοποίηση της διεπαφής *Task* θα πρέπει να προστεθεί κώδικας στην μέθοδο *run()* ο οποίος θα αντιπροσωπεύει το λειτουργικό μέρος της εργασίας. Η δημιουργία μίας εργασίας έχει ως εξής:

```
class MyTask implements Task{
        public void run(){
         //insert code here
        }
}
```

Όπως αναφέραμε και σε προηγούμενο στάδιο οι εργασίες διακρίνονται σε τρία είδη. Διεργασίες που εκτελούνται αμέσως μετά την καταχώρηση τους, σε συγκεκριμένα χρονικά διαστήματα και σε περιοδικές φάσης. Δομή και κλήση των εργασιών έχει ως εξής:

- Άμεση εκτέλεση: Βάση της πιο κάτω μεθόδου καταχωρείται η εργασία *t*, συσχετίζεται με την ομάδα εργασιών *g* και η εργασία εκτελείται αμέσως μετά την καταχώρησή της.

  void registerTask(Task t, TaskGroup g);

- Αναβεβλημένη εκτέλεση: Βάση της πιο κάτω μεθόδου καταχωρείται η εργασία *t*, συσχετίζεται με την ομάδα εργασιών *g* και η εργασία εκτελείται μετά το πέρας της χρονικής διάρκειας που καθορίζει η τρίτη παράμετρος *millis* σε χιλιοστά του δευτερολέπτου.

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

- Περιοδική εκτέλεση: Βάση της πιο κάτω μεθόδου καταχωρείται η εργασία *t*, συσχετίζεται με την ομάδα εργασιών *g* και η εργασία εκτελείται ανά τακτά χρονικά διαστήματα χρονικής διάρκειας που καθορίζει η τρίτη παράμετρος *millis* σε χιλιοστά του δευτερολέπτου.

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

2.3.1.2 Ανταλλαγή Μηνυμάτων

Αποστολή Μηνυμάτων

Όπως αναφέρθηκε και σε προηγούμενο στάδιο της εργασίας, η επικοινωνία μεταξύ των κόμβων μίας εφαρμογής γίνεται εφικτή μέσο του στρώματος επικοινωνίας που μας προσφέρει η YALPS και συγκεκριμένα, μέσο ενός αντικειμένου *cl* τύπου *CommunicationLayer*. Το αντικείμενο *cl* μας προσφέρει δύο διαφορετικές μεθόδους για αποστολή μηνυμάτων δίνοντας μας έτσι την επιλογή αποστολής αυτών μέσο πρωτοκόλλων *UDP* ή *TCP*, ανάλογα με την ανάγκη και τις απαιτήσεις της εφαρμογής.

- Αποστολή μηνύματος *msg* στον κόμβο *dest* με χρήση UDP πρωτοκόλλου:
  public void sendUDP(Serializable msg, NodeID dest) throws IOException;

- Αποστολή μηνύματος *msg* στον κόμβο *dest* με χρήση TCP πρωτοκόλλου:
  public void sendTCP(Serializable msg, NodeID dest) throws IOException;

Παραλαβή Μηνυμάτων

Η παραλαβή μηνυμάτων μεταξύ κόμβων της εφαρμογής επιτυγχάνεται με την υλοποίηση της μεθόδου που ορίζεται στην διεπαφή *Receiver*. Συγκεκριμένα παραλαμβάνεται το μήνυμα *msg* με επικεφαλίδα *header* που στάλθηκε από τον κόμβο *sender*. Σκοπός του χρήστη είναι να υλοποιήσει το σώμα αυτής της μεθόδου έτσι ώστε να μπορεί να διαχειρίζεται τα εισερχόμενα μηνύματα ανάλογα με τον τύπο τους.

```
public boolean receive(short header, Object msg, NodeID sender){
        //insert code here
}
```

2.3.1.3 Βασικά Συστατικά και Εκτέλεση

Αρχικοποίηση Κόμβου

Θεμελιώδης μέθοδος στην κλάση *YalpsNode* είναι η *StartNode*. Αυτή η μέθοδος επιτρέπει στον κόμβο να εκτελέσει όλες τις αρχικοποιήσεις που είναι απαραίτητες για την εκτέλεση της εφαρμογής. Πιο αναλυτικά, η μέθοδος αυτή περιλαμβάνει αρχικοποιήσεις τιμών, προγραμματισμό εργασιών και αποστολή μηνυμάτων σε άλλους κόμβους. Αυτό που απομένει είναι ο χρήστης με την σειρά του, ανάλογα με τις ανάγκες της εφαρμογής, να εισάγει τον κώδικα στον κύριο κορμό της μεθόδου αυτής.

```
public void startNode(){
    //insert code here
}
```

Εκτελεστές Εφαρμογής

Μετά την ολοκλήρωση της υλοποίησης της διεπαφής *YalpsNode,* είναι εφικτό να την εκτελέσουμε σαν εφαρμογή αρκεί να γίνει επίκληση σε ένα από τους δύο εκτελεστές που μας προσφέρει η YALPS. Συγκεκριμένα:

- Για την εκτέλεση της εφαρμογής σε περιβάλλον προσομοίωσης γίνεται η επίκληση του εκτελεστή

  yalps.launchers.SimulatedNodeStarter

- Για την εκτέλεση της εφαρμογής σε πραγματικό διαδικτυακό περιβάλλον κατανεμημένων υπολογιστών γίνεται η επίκληση του εκτελεστή

  yalps.launchers.ExecutorNodeStarter

Ο κάθε ένας από αυτούς παίρνει σαν είσοδο ένα αρχείο διαμόρφωσης που καθορίζει ποιοι κόμβοι της εφαρμογής YALPS θα λάβουν μέρος στον υπολογισμό και στην συνέχεια αναλαμβάνει την ορθή εκτέλεση τους.

Εκτέλεση Εφαρμογής

Κατά την εκτέλεση της εφαρμογής θα πρέπει να γίνει καθορισμός του εκτελεστή για το κατά επιθυμία περιβάλλον, πραγματικό ή προσομοίωσης.

Παράδειγμα επιλογής εκτελεστή για εκτέλεση σε πραγματικό περιβάλλον:

```
java -cp /MyApp.jar launchers.ExecutorNodeStarter nodeID.conf
```

Παράδειγμα επιλογής εκτελεστή για εκτέλεση σε περιβάλλον προσομοίωσης:

```
java -cp /MyApp.jar launchers.SimulatedNodeStarter nodeID.conf
```

Έχουμε επίσης την δυνατότητα, εάν γνωρίζουμε εκ των προτέρων ποιος εκτελεστής μας ικανοποιεί, να τον περιλάβουμε και να τον ενσωματώσουμε μέσα στο εκτελέσιμο .jar αρχείο. Σε αυτή την περίπτωση η ενσωμάτωση θα περιλαμβάνει ένα αποκλειστικό εκτελεστή και η εκτέλεση της εφαρμογής θα τροποποιηθεί ως εξής:

```
java –jar MyApp.jar nodeID.conf
```

Για όλα τα προαναφερθέντα, *nodID.conf* είναι το αρχείο διαμόρφωσης του κάθε κόμβου και *MyApp.jar* η εκτελέσιμη εφαρμογή.

2.3.1.4 Αρχείο Διαμόρφωσης

Το αρχείο διαμόρφωσης της YALPS είναι ένα αρχείο ιδιοτήτων της γλώσσας προγραμματισμού Java το οποίο δηλώνει ένα σύνολο αντικειμένων με τις ιδιότητες

τους. Ποίο αναλυτικά το αρχείο αποτελείται από δηλώσεις αρχικοποιήσεων αντικειμένων και ρύθμισης των ιδιοτήτων τους.

- Δηλώσεις Αρχικοποιήσεων

  Η μορφή για την δήλωση μιας αρχικοποίησης ενός αντικειμένου έχει ως εξής:

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

  κατά την οποία το *<object name>* είναι το όνομα του αντικειμένου και *<fully qualified class name>* είναι το απόλυτο όνομα της κλάσης του αντικειμένου.

- Δηλώσεις Ρυθμίσεων ιδιοτήτων

  Για να είναι εφικτή η ρύθμιση μίας ιδιότητας ενός αντικειμένου μέσο του αρχείου διαμόρφωσης θα πρέπει η κλάση του αντικειμένο *<object name>* να περιέχει μία μέθοδο της μορφής

  void set+'property_name'();

  όπου με το σύμβολο ¨+¨ υποδουλώνουμε την συνένωση των δύο συμβολοσειρών *set* και *property_name.*

  Οι δηλώσεις για την ρύθμιση των ιδιοτήτων ενός αντικειμένου της εφαρμογής στο αρχείο διαμόρφωσης έχουν ως εξής:

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

  όπου το *<object name>* είναι το όνομα του αντικειμένου που θέλουμε να ρυθμίσουμε την ιδιότητα, *<property name>* είναι το όνομα της ιδιότητας που επιθυμούμε να ρυθμίσουμε και *<value>* η τιμή που θα αποδώσουμε στην ιδιότητα.

Παράδειγμα Αρχείου διαμόρφωσης:

```
node.CLASS = yalps.example.application.myApplication
node.NodeList=6
```

Μέχρι αυτό το σημείο για να είναι ορθό το αρχείο διαμόρφωσης θα πρέπει στην εφαρμογή *myApplication* να υπάρχει ολοκληρωμένη η μέθοδος

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

, βάση αυτή, *Type* μπορεί να είναι οποιοσδήποτε τύπος (Integer, float, String, long, double) αρκεί να είναι επιτρεπτό να του ανατεθεί η τιμή που του αποδίδουμε μετά τον χαρακτήρα ¨=¨ .

Ανάλογα με το περιβάλλον που θα τρέξει η εφαρμογή υπάρχουν κάποιες μικροδιαφορές στην συνέχεια του αρχείου διαμόρφωσης.

Η δήλωση των κόμβων της εφαρμογής που θα τρέξει σε προσομοίωση έχει ως

```
node.nodeList=1;2;3;4;5;6
```

ενώ η δήλωση για εκτέλεση σε πραγματικό κατανεμημένο περιβάλλον θα έχει την μορφή:

```
node.nodeList=hostname1:port1;hostname2:port2;....;
```

όπου hostnameNum είναι το μοναδική διαδικτυακή διεύθυνση *(IP)* της κάθε υπολογιστικής οντότητας και portNum η θύρα που θα λαμβάνει χώρο η επικοινωνία κάθε κόμβου.

Στο σημείο αυτό το αρχείο διαμόρφωσης για προσομοίωση έχει ολοκληρωθεί. Αυτό που υπολείπεται για το αρχείο διαμόρφωσης πραγματικού περιβάλλοντος είναι να τοποθετηθούν και οι εξής αναθέσεις:

```
executor.debug=true

executor.tokenbucket=true

executor.burstSize=1000000

executor.rnd='comm'

executor.lossRate=0
```

executor.maxQueueLength=10000000

executor.maxSendDelay=0

executor.uploadBandwidth=0

Αναθέσεις οι οποίες καθορίζουν άλλες σημαντικές παραμέτρους του εκτελεστή και οι τιμές τους μπορεί να αλλάξουν ανάλογα με τις ανάγκες της εφαρμογής.

2.4 Το Δικτυακό Σύστημα PlanetLab

2.4.1 Γενικά

Το *PlanetLab [5]* είναι μια ανοικτή πλατφόρμα για την ανάπτυξη και τη χρήση υπηρεσιών κατανεμημένου υπολογισμού παγκόσμιας κλίμακας. Είναι ένα γεωγραφικά κατανεμημένο δίκτυο που ξεκίνησε αρχικά σαν μια ερευνητική κοινότητα με συνδέσεις μεταξύ διαφόρων πανεπιστημίων υπό την εποπτεία του Πανεπιστημίου Princeton. Η ιδέα του σχεδιασμού του είναι τέτοια ώστε να επιτρέπει στους ερευνητές, ανεξαρτήτου τοποθεσίας, να υλοποιούν, να δοκιμάζουν και να αξιολογούν εφαρμογές κατανεμημένου υπολογισμού. Δηλαδή, εφαρμογές οι οποίες δεν περιορίζονται στην αξιοποίηση ενός και μοναδικού υπολογιστή αλλά σε πλήθος κατανεμημένων υπολογιστικών οντοτήτων ταυτόχρονα, χρησιμοποιώντας έτσι το παγκόσμιο δίκτυο σαν ένα μεγάλο υπολογιστή.

Από το 2014 το PlanetLab αποτελείται από 1188 κόμβους σε 585 τοποθεσίες. Το PlanetLab είναι διαχωρισμένο σε δύο υποσύνολα ανάλογα με την τοποθεσία των υπολογιστικών οντοτήτων που το απαρτίζουν. Συγκεκριμένα συγκροτείται από το PlanetLab Europe (PLE) για το σύνολο των υπολογιστικών οντοτήτων που γεωγραφικά ανήκουν στην Ευρώπη και το PlanetLab Central (PLC) για το σύνολο των υπολογιστικών οντοτήτων που ανήκουν στις υπόλοιπες γεωγραφικές τοποθεσίες. Σε

αυτό το πρόγραμμα λαμβάνουν μέρος τα μεγαλύτερα ινστιτούτα (εκπαιδευτικά κυρίως) ενώ στους χορηγούς του συμπεριλαμβάνονται δύο από τις μεγαλύτερες εταιρείες στον χώρο της τεχνολογίας της πληροφορικής, συγκεκριμένα η *Intel [18]* και η *HP [19]*. Αυτή τη στιγμή υπάρχουν χιλιάδες εφαρμογές που δημιουργήθηκαν από διάφορα πανεπιστήμια και οργανισμούς οι οποίες τρέχουν στο PlanetLab. Υπάρχουν όμως και εφαρμογές που αναπτύχθηκαν από μερίδες φοιτητών και ομάδες ερευνητών, οι οποίες είναι διαθέσιμες προς χρήση από οποιονδήποτε και εξυπηρετούν ερευνητικούς και μη κερδοσκοπικούς σκοπούς.

Βάση του ορισμού και των απαιτήσεων του προβλήματος *Do-All,* το PlanetLab είναι ένα κατάλληλο περιβάλλον για την εφαρμογή και λήψη μετρικών των αλγόριθμων που αναπτύξαμε *AN, AR* και *Do-Um*.

2.4.2 Σύντομος Οδηγός Χρήσης

Κάθε οργανισμός *(site)* που είναι ενεργός στο PlanetLab προσφέρει στην κοινότητα του PlanetLab τουλάχιστον δύο δικές του μηχανές οι οποίες θα μπορούν στο μέλλον να χρησιμοποιηθούν από χρήστες που ανήκουν σε άλλους οργανισμούς. Το τμήμα πληροφορικής του Πανεπιστημίου Κύπρου σαν μέλος υπάγεται στο σύνολο του PlanetLab Europe (PLE) λόγο και της γεωγραφικής του τοποθεσίας. Μέλος της κοινότητας αυτής γίνεται κάποιος μέσο αίτησης εγγραφής σε κάποιο οργανισμό. Εφόσον η αίτηση εγκριθεί τότε ο αιτητής γίνεται επίσημο μέλος με προσωπικό λογαριασμό, αποκτά πρόσβαση στις μηχανές του PlanetLab και του παρέχεται το δικαίωμα να αναπτύξει και να εκτελέσει τις εφαρμογές του χωρίς περιορισμούς.

Οι πόροι του PlanetLab είναι οργανωμένοι σε μερίσματα (slices). Κάθε οργανισμός μπορεί να έχει μέχρι και 10 μερίσματα και κάθε μέρισμα έχει δικαίωμα χρήσης μέχρι 150 μηχανές. Ένα μέρισμα είναι μια συλλογή από εικονικές μηχανές και δεσμεύει μέρος των πόρων (εύρος ζώνης δικτύου, μνήμη, χώρος δίσκου) το οποίο περιορίζεται ανάλογα με την κατάσταση δικτύου του κάθε χρήστη. Ο κάθε χρήστης μπορεί να δουλεύει ανενόχλητα σε κάθε μηχανή μέσω του μερίσματος του χωρίς να

επηρεάζει ή να επηρεάζεται από τους άλλους χρήστες που χρησιμοποιούν την ίδια μηχανή. Σε πρώτο στάδιο ο χρήστης καλείται να αναθέσει μηχανές στο μέρισμα του δημιουργώντας έτσι ένα μοναδικό κατανεμημένο δίκτυο για την εφαρμογή του.

Για πρόσβαση σε οποιαδήποτε μηχανή πρέπει πρώτα να δημιουργηθεί ένα ζευγάρι κλειδιών μέσω SSH για σκοπούς επικύρωσης και επαλήθευσης της ταυτότητας του. Το ζευγάρι αυτό απαρτίζουν το Public Key και το Private Key. Το Public Key γνωστοποιείται στο μέρισμα του χρήστη μέσω του λογαριασμού του από την επίσημη ιστοσελίδα του PlanetLab ενώ το Private Key δίνεται ως στοιχείο επαλήθευσης κατά τη διαδικασία σύνδεσης με την απομακρυσμένη μηχανή. Η δημιουργία των δύο κλειδιών γίνεται με την ακόλουθη εντολή:

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

Μετά την γνωστοποίηση του Public Key στη σελίδα του PlanetLab, τότε ο χρήστης μπορεί να ενωθεί με κάποια μηχανή του Planet Lab που ανήκει στο μέρισμα του εκτελώντας την εντολή:

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

όπου *slice_name* είναι το όνομα του μερίσματος του χρήστη, *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.

Εφόσον γίνει η σύνδεση με επιτυχία, πλέον ο χρήστης είναι να αξιοποιήσει τη μηχανή. Η μεταφορά αρχείων από και προς μία μηχανή του μερίσματος είναι εφικτή μέσο των παρακάτω εντολών:

```
Scp –i ~/.ssh/id_rsa -r file1 file2 slice_name@node:
```

```
scp -i ~/.ssh/id_rsa slice_name@node:filename1 ~/Location
```

Η πρώτη εντολή μεταφέρει τα αρχεία *file1 file2* από τον τρέχων κατάλογο στον προσωπικό χώρο του μερίσματος `slice_name` στη μηχανή `node` που ανήκει στο μέρισμα αυτό. Η δεύτερη με την σειρά της μεταφέρει το αρχείο *filename1* από

την μηχανή `node` που ανήκει στο μέρισμα `slice_name` στον κατάλογο `Location` του τρέχων υπολογιστή.

Αναλυτικότερα εγχειρίδια χρήσης του PlanetLab μπορεί να βρει κάποιος στην επίσημη ιστοσελίδα του Planet Lab [20].

Κεφάλαιο 3

Μοντέλο Υπολογισμού

3.1 Κατανεμημένο Περιβάλλον

Το κατανεμημένο περιβάλλον στο οποίο θα λάβει χώρο η εφαρμογή αποτελείται από *n* διαφορετικές υπολογιστικές οντότητες, η κάθε μία εκ των οποίον διακρίνεται από το μοναδικό αναγνωριστικό της. Συγκεκριμένα, εφόσον η εφαρμογή θα αξιολογηθεί σε πραγματικό κατανεμημένο περιβάλλον αυτό το αναγνωριστικό για τον κάθε κόμβο θα είναι η διαδικτυακή του διεύθυνση *(IP)*.

Οι κόμβοι έχουν πρόσβαση σε ένα κοινό ρολόι χρονισμού και η επικοινωνία λαμβάνει χώρο σε ένα πλήρως συνδεδεμένο μέσο επικοινωνίας στο οποίο δεν υπάρχει αλλοίωση ή/και απώλεια μηνυμάτων.

3.2 Γύροι

Για την ικανοποίηση της δομής αλγόριθμων, υποθέτουμε ότι ο κάθε γύρος διασπάτε σε τρία διαδοχικά στάδια. Συγκεκριμένα διασπάτε σε:

- *Λήψη Μηνυμάτων,* λαμβάνονται από ένα κόμβο τα μηνύματα τα οποία στάλθηκαν κατά τον προηγούμενο γύρο και προορίζονταν προς αυτόν.
- *Στάδιο Υπολογισμού,* όπου η κάθε υπολογιστική οντότητα εκτελεί το πολύ μία εργασία.
- *Αποστολή Μηνυμάτων,* κατά την οποία κάθε κόμβος αποστέλλει μηνύματα σε άλλους κόμβους.

Πιο αναλυτικά, βάση της ιδεολογίας τους, μια ολόκληρη επανάληψη των αλγόριθμων AN και AR αποτελείται από τρεις τέτοιους γύρους ενώ μια επανάληψη του αλγόριθμου Do-Um προνοεί δύο τέτοιους γύρους.

3.3 Εργασίες

Κάθε εργασία αποτελείται από ένα μοναδικό θετικό αριθμό ο οποίος σηματοδοτεί το μοναδικό αναγνωριστικό της εργασίας αυτής. Αξίζει να αναφέρουμε πως στην δική μας υλοποίηση οι εργασίες είναι εικονικές *(virtual or dummy tasks)*, δηλαδή δεν έχουν κάποιο σώμα εκτέλεσης και η ανάθεση τους σε κάποιο κόμβο σηματοδοτεί και την επιτυχές ολοκλήρωση της εκτέλεσης τους. Σε μία πραγματική ανάγκη επίλυσης του προβλήματος Do-All, αυτές οι εικονικές εργασίες αντικαθιστώνται από τις ανάλογες ρεαλιστικές εργασίες που προκύπτουν από τις ανάγκες του κάθε προβλήματος.

3.4 Αντίπαλος

Για την μοντελοποίηση σφαλμάτων και μελέτη χειρότερης περίπτωσης, θεωρούμε την ύπαρξη ενός αντιπάλου που προκαλεί σφάλματα στην πρόθεση του να μειώσει την απόδοση του αλγόριθμου. Συγκεκριμένα, ο αντίπαλος μπορεί να προκαλέσει καταρρεύσεις και επανεκκινήσεις υπολογιστικών οντοτήτων ανάλογα με τον αλγόριθμο που εκτελούμε.

3.5 Επανεκκίνηση Υπολογιστικών Οντοτήτων

Βάση των απαιτήσεων των αλγόριθμων μας, όταν ένας κόμβος επανεκκινήσει τότε η μόνη γνώση που τον διακατέχει είναι ο ίδιος ο αλγόριθμος και τα μοναδικά αναγνωριστικά των άλλων οντοτήτων που ξεκίνησαν μαζί την εκτέλεση της εφαρμογής. Επομένως, καμία άλλη πληροφορία για τις τρέχων ολοκληρωμένες εργασίες και για το ποίοι άλλοι κόμβοι είναι ενεργοί – ζωντανοί δεν παρέχεται αρχικά στον κόμβο που επανεκκίνει.

Αλγοριθμικά, ένας κόμβος που επανεκκίνει αποστέλλει μηνύματα επανεκκίνησης σε όλους τους γνωστούς προς αυτόν κόμβους κατά την πρώτη φάση του γύρου. Με το πέρας του γύρου αυτού θα έχει ήδη λάβει όλες τις απαραίτητες πληροφορίες από τους ενεργούς κόμβους έτσι ώστε να θεωρηθεί ενεργός, να αναβαθμίσει την ολική του γνώση και να λάβει θέση στον υπολογισμό.

3.6 Εχθρικό Σενάριο

Βάση των απαιτήσεων, των αναγκών και της δομής των αλγορίθμων AN και Do-Um ένα εχθρικό μοτίβο είναι αποδεκτό εάν σε κάθε γύρο υπάρχει τουλάχιστον μία ενεργή – ζωντανή υπολογιστική οντότητα.

Επιπρόσθετα με την προϋπόθεση αυτή, για τον αλγόριθμο AR, ο οποίος είναι ικανός να διαχειρίζεται και τυχών επανεκκινήσεις υπολογιστικών οντοτήτων κατά τον υπολογισμό, το εχθρικό σενάριο είναι πιο αυστηρό. Συγκεκριμένα, ο αλγόριθμος AR είναι 26-περιορισμένος. Με τον όρο κ-περιορισμένος υπονοούμε το πρότυπο καταρρεύσεων κατά το οποίο υπάρχει τουλάχιστον ένας ζωντανός επεξεργαστής σε κ ακολουθιακά βήματα του αλγορίθμου. Βάσει του ορισμού, θα πρέπει σε σχεδόν 3 ολόκληρους γύρους του αλγόριθμου AR να υπάρχει τουλάχιστον ένας ζωντανός επεξεργαστής (όπως θα διατυπωθεί και σε επόμενο κεφάλαιο κάθε γύρος του AR αποτελείται από 9 βήματα).

3.7 Ορθότητα και Ολοκλήρωση

Ανάλογα με τον αλγόριθμο που αξιολογούμε οι ιδιότητες/εγγυήσεις που παρέχονται διαφέρουν. Συγκεκριμένα αυτοί οι αλγόριθμοι είναι ορθοί εάν για κάθε εκτέλεση τους σε ένα αποδεκτό εχθρικό μοτίβο (κεφ. 3.6) μετά από ένα πεπερασμένο αριθμό βημάτων η εκτέλεση τους θα τερματίσει και όλες οι εργασίες θα έχουν εκπληρωθεί.

Σε αντίθεση με τον αλγόριθμο Do-Um οι αλγόριθμοι AN και AR επιλύουν το πρόβλημα Do-All κάτω υπό την προϋπόθεση ύπαρξης δικτύου με αξιόπιστη πολυεκπομπή. Επίσης ο αλγόριθμος AR είναι ο μόνος εκ των τριών αλγόριθμων που μελετάμε, που επιτρέπει επανεκκινήσεις υπολογιστικών οντοτήτων κατά τον υπολογισμό.

Κεφάλαιο 4

Περιγραφή και Υλοποίηση Αλγορίθμων

4.1 Υλοποίηση Αλγορίθμων με Χρήση YALPS

Και οι τρεις αλγόριθμοι, AN, AR και Do-Um, υλοποιούνται πλήρως με τη βοήθεια της βιβλιοθήκης YALPS σε γλώσσα προγραμματισμού Java μέσο αρκετών έτοιμων υλοποιημένων μεθόδων και διαδικασιών που μας παρέχει για την ανάπτυξη μιας κατανεμημένης εφαρμογής (κεφ. 2.4). Η λειτουργία των αλγορίθμων αυτών βασίζεται σε τρία κύρια βήματα: Βήμα «παραλαβής» όπου ένας κόμβος λαμβάνει ένα σύνολο

μηνυμάτων και κάνει αναβάθμιση της γνώσης του. Το βήμα «υπολογισμού» όπου κάθε κόμβος επιλέγει να εκτελέσει κάποια εργασία με βάση κάποιο κανόνα κατανομής εργασιών *(Load Balancing Rule)*. Τέλος, το βήμα «αποστολής» κατά το οποίο ένας κόμβος ενημερώνει τους αρμόδιους κόμβους για την διεκπεραίωση της εργασίας που ανέλαβε μέσο αποστολής μηνυμάτων. Τα τρία βήματα μαζί αποτελούν ένα ολόκληρο στάδιο, και ένας πεπερασμένος αριθμός σταδίων, ανάλογα με τον αλγόριθμο, αποτελούν μία φάση. Για να γίνει αυτό εφικτό, ανάλογα με τον αλγόριθμο και τον αριθμό των σταδίων που τον απαρτίζουν, μία ολόκληρη φάση του αλγορίθμου για να ολοκληρωθεί μπορεί να χρειαστεί να αναπαρασταθεί με περισσότερους από ένα συγχρονισμένους περιοδικούς γύρους της YALPS.

Σε αρχικό στάδιο δηλώνονται και αρχικοποιούνται οι n κόμβοι που απαρτίζουν το σύστημα. Όπως αναφέραμε και στο υποκεφάλαιο 2.4 οι κόμβοι στη βιβλιοθήκη YALPS δηλώνονται ως αντικείμενα της διεπαφής NodeID και αρχικοποιούνται στη μέθοδο `startNode()` η οποία κληρονομείται από την κλάση `AbstractYalpsNode`.

Στη συνέχεια καθορίζεται η μορφή των μηνυμάτων που θα διακινούνται καθώς και το υπόστρωμα επικοινωνίας μεταξύ των κόμβων. Βάση των απαιτήσεων των αλγορίθμων μας η επικοινωνία μεταξύ κόμβων γίνεται εφικτή με αποστολή και λήψη μηνυμάτων με χρήση του πρωτοκόλλου TCP. Αυτό επιτυγχάνεται με την χρήση της μεθόδου `sendTCP()` η οποία παίρνει σαν παράμετρο το μήνυμα *msg* που πρέπει να αποσταλεί και τον κόμβο – παραλήπτη *DestinationNode* στον οποίο πρέπει να παραδοθεί το μήνυμα και το αποστέλλει σ’ αυτόν βάση του TCP πρωτοκόλλου.

Αυτό που απομένει είναι η υλοποίηση των δύο βασικών μεθόδων της κλάσης `AbstractYalpsNode`, συγκεκριμένα την δομή του βασικού κορμού του αλγόριθμου που αναπτύσσουμε στην μέθοδο `run()` και της μεθόδου που διαχειρίζεται την παραλαβή μηνυμάτων `receive()`.

Εφόσον υλοποιηθούν και οι δύο αυτές μέθοδοι δημιουργείται το κατάλληλο αρχείο διαμόρφωσης η δομή του οποίου περιγράφεται αναλυτικά στο υποκεφάλαιο 2.4.1.4 .

Με την βοήθεια της προγραμματιστικής εφαρμογής Eclipse Juno [21], η οποία μας παρείχε την δυνατότητα να ενσωματώσουμε τον εκτελεστή launchers.ExecutorNodeStarter ή launchers.SimulatedNodeStarter μέσα στο εκτελέσιμο .jar αρχείο, η κλήση της εφαρμογής τροποποιείται και έχει ως εξής:

java –jar MyApp.jar nodeID.conf

όπου MyApp.jar το εκτελέσιμο αρχείο και nodeID το αρχείο διαμόρφωσης του ID–οστού κόμβου.

4.2 Αλγόριθμος AN

Στο συγκεκριμένο υποκεφάλαιο αρχικά θα γίνει αρχικά μια σύντομη και συγχρόνως περιεκτική περιγραφή του αλγορίθμου AN, του πρώτου αλγορίθμου με τον οποίο ασχολείται η παρούσα εργασία και στη συνέχεια θα παρουσιαστούν και θα αναλυθούν σημαντικές λεπτομέρειες σε θέματα υλοποίησης του αλγορίθμου αυτού.

4.2.1 Περιγραφή Αλγορίθμου AN

Ο AN είναι ένας αλγόριθμος που επιλύει το πρόβλημα *Do-All (βλ. Υποκεφάλαιο 2.1)* και ανήκει στο Μοντέλο Μεταβίβασης Μηνυμάτων [1-3, 10]. Με βάση το μοντέλο αυτό, όλες οι υπολογιστικές οντότητες επικοινωνούν μεταξύ τους με ανταλλαγή μηνυμάτων για την διεκπεραίωση των εργασιών. Ο αλγόριθμος AN συνδυάζει τοπικές δομές δεδομένων και την τεχνική πολλαπλών συντονιστών όπως περιγράφονται πλήρες στην ενότητα 3 του [2].

Σύμφωνα με την τεχνική πολλαπλών συντονιστών, χρησιμοποιείται ένα επιθετικό μοτίβο συντονιστών το οποίο επιτρέπει σε πολλαπλούς επεξεργαστές να συμπεριφέρονται σαν συντονιστές ταυτόχρονα. Ο αριθμός των επεξεργαστών που συμπεριφέρονται ως συντονιστές προσαρμόζεται αναλόγως της περίστασης.

Συγκεκριμένα, όταν οι καταρρεύσεις υπολογιστών διαταράσσουν την πρόοδο του υπολογισμού, τότε ο αριθμός των συντονιστών αυξάνεται. Αντιθέτως όταν οι καταρρεύσεις υποχωρήσουν διορίζεται σαν συντονιστής μόνο ένας ενεργός επεξεργαστής. Αυτή η τεχνική είναι αποδοτική όταν υπάρχει αξιόπιστη πολυεκπομπή, πιο αναλυτικά όταν ένα μήνυμα περιλαμβάνει πολλούς αποδέκτες τότε το δίκτυο μας δίδει την εγγύηση ότι όλοι θα το παραλάβουν. Τυχών απώλειες και αλλοιώσεις μηνυμάτων δεν υφίστανται. Βάση των προαναφερθέντων, ο αλγόριθμος AN τρέχει σε συγχρονισμένο μοντέλο και προϋποθέτει αξιόπιστη πολυεκπομπή.

Έχει αποδειχθεί ότι ο αλγόριθμος AN είναι ανεκτικός σε f καταρρεύσεις επεξεργαστών (f < p). Έχει πολυπλοκότητα Εργασίας S όπου S = O$\big((n + p\ log\ p/\ log\ log\ p)\log \mathrm{f}\big)$ και πολυπλοκότητα Μηνυμάτων M όπου M = O$(n + p\ log\ p/\ log\ log\ p\ +\ fp)$. Ο αλγόριθμος AN αποτελεί και την βάση του αλγορίθμου AR ο οποίος επιτρέπει και επανεκκινήσεις επεξεργαστών μετά από σφάλματα.

4.2.2 Δομές Δεδομένων Και Φάσεις AN

Συντονιστές και Εργάτες

Ο αλγόριθμος AN εκτελείτε μέσα σε ένα επαναληπτικό βρόγχο ως ότου να εκτελεστούν όλες οι εργασίες. Μια επανάληψη του βρόγχου ονομάζεται φάση. Μια φάση αποτελείται από τρία διαδοχικά στάδια. Κάθε στάδιο απαρτίζεται από τρία βήματα, άρα μια φάση συνολικά αποτελείται από 9 βήματα. Σε κάθε στάδιο το πρώτο βήμα χρησιμοποιήστε για παραλαβή μηνυμάτων, το δεύτερο βήμα για εκτέλεση τοπικών υπολογισμών και το τρίτο για αποστολή μηνυμάτων.

Ο κάθε επεξεργαστής μπορεί να είναι συντονιστής σε κάποια φάση και όλοι οι επεξεργαστές είναι εργάτες σε κάθε φάση. Ο συντονιστής είναι υπεύθυνος για την καταγραφή της προόδου καθώς οι εργάτες εκτελούν μία εργασία και την αναφέρουν σε αυτόν. Εάν τουλάχιστον ένας επεξεργαστής ενεργεί σαν συντονιστής καθ' όλη την διάρκεια μίας φάσης τότε η φάση αυτή ονομάζεται *attended* αλλιώς *unattended*. Ο

αριθμός των επεξεργαστών που θα θεωρηθούν σαν συντονιστές καθορίζονται από την Αρχή Martingale. Συγκεκριμένα αν κανένας από τους αναμενόμενους coordinators δεν επιζήσει σε μία φάση τότε διπλασιάζεται ο αριθμός των coordinators στην επόμενη φάση. Όταν έστω και ένας από τους coordinators επιζήσει σε μια ολόκληρη φάση τότε ο αριθμός των coordinators μειώνεται σε 1.

Local Views

Οι επεξεργαστές υποθέτουν τον ρόλο ενός coordinator βασισμένοι στην τοπική τους γνώση *(Loal View).* Ο συμβολισμός του Local View είναι $L_{\ell w}$, όπου ℓ είναι η φάση και *w* το μοναδικό αναγνωριστικό του κάθε εργάτη. Κατά τον υπολογισμό κάθε worker *w* διαχειρίζεται μια λίστα $Lw = <q1, q2 .. qn>$ με τους υποτιθέμενους ζωντανούς επεξεργαστές. Στην συγκεκριμένη λίστα *L* οι επεξεργαστές είναι χωρισμένοι σε στρώματα. $Lw = <\Lambda^0, \Lambda^1 ... >$ όπου σε κάθε στρώμα *i* έχουμε διπλάσιους επεξεργαστές από το προηγούμενο. Εξαίρεση αποτελεί το τελευταίο επίπεδο το οποίο μπορεί να έχει λιγότερους επεξεργαστές από το $i - 1$ επίπεδο, αυτό οφείλεται στα χαρακτηριστικά της δεντρικής δομής που χρησιμοποιεί ο αλγόριθμος.

Κανόνας ενημέρωσης Local View

Το Local View ενημερώνεται ανάλογα με το είδος της φάσης ℓ.

- Περίπτωση όπου η φάση ℓ είναι attended: Ο επεξεργαστής *w* λαμβάνει μηνύματα περίληψης από κάποιους συντονιστές από το επίπεδο Λ^0. Τότε υπολογίζει το σύνολο *Pw* όπως περιγράφεται στο στάδιο 3 του αλγορίθμου. Το Local View $L_{\ell+1,w}$ του *w* θα έχει δενδρική δομή περιλαμβάνοντας όλους τους ενεργούς επεξεργαστές ταξινομημένους βάση του αναγνωριστικού τους.

- Περίπτωση όπου η φάση ℓ είναι unattended: Το Local View του επεξεργαστή *w* για την φάση $\ell + 1$ είναι $L_{\ell+1,w} = <\Lambda^1 ... \Lambda^j >$

Δομή της φάσης και δέσμευση εργασίας

Γενική γνώση κάθε επεξεργαστή:

- *T* – Σύνολο όλων των εργασιών που πρέπει να γίνουν

Κάθε επεξεργαστής w διαχειρίζεται:

- *Dw* - Την τοπική πληροφορία για το σύνολο εργασιών που έγιναν.
- *Pw* – Σύνολο ζωντανών επεξεργαστών που γνωρίζει.
- *Uw* - Εργασίες που είναι άγνωστη κατάσταση τους στον w.

Ο Υπολογισμός του Uw είναι ίσος με $Uw = T \setminus Dw$. Δηλαδή όλες οι εργασίες που βρίσκονται στο *T* - και όχι στο *Dw* - γνωστές εργασίες που ολοκληρώθηκαν.

Ο υπολογισμός ξεκινάει στην φάση 0 και ο κάθε κόμβος *q* έχει όλους τους επεξεργαστές στο $L_{0,q}$ και άδειο το $D_{0,q}$. Στην αρχή της φάσης ℓ κάθε εργάτης *w* εκτελεί μία εργασία σύμφωνα με το *Local View* $L_{\ell,w}$ και την γνώση του για τις άγνωστες εργασίες $U_{\ell,w}$ χρησιμοποιώντας τον κανόνα *Load Balancing*. Βάση του κανόνα αυτού ο worker w εκτελεί την εργασία *t* της οποίας η θέση είναι $(\iota \bmod |U_{\ell w}|)^{\text{οστή}}$.

Διευκρίνηση:

- $| U_{\ell,w} |$ είναι το μέτρο των Unaccound Tasks για την φάση ℓ στον worker *w*
- ι είναι η θέση του worker στο Local view $L_{\ell,w}$

Στην συνέχεια οι εργάτες αποστέλλουν αναφορά εκτέλεσης μίας εργασίας *t* σε όλους τους συντονιστές σύμφωνα και πάλι με το Local View τους για την φάση ℓ.

4.2.3 Λεπτομέρειες Υλοποίησης AN

Πιο κάτω μας παρουσιάζεται ο ψευδοκώδικας βάση του οποίου έγινε η υλοποίηση του αλγορίθμου AN (Εικόνα 4.1) και τα στάδια που απαρτίζουν μία ολοκληρωμένη φάση ℓ (Εικόνα 4.2).

Phase ℓ of algorithm *AN*:

STAGE 1.

RECEIVE: The receive substage is not used.

COMPUTE: Processor w performs task z according to the load balancing rule.

SEND: Processor w sends `report`(z) to all coordinators, that is, to all processors in layer Λ^0 of its local view $L_{\ell,w}$.

STAGE 2.

RECEIVE: : If processor w is coordinator c, it gathers `report` messages. For any coordinator c, let $Z = \{z^{r_1}, \ldots, z^{r_k}\}$ be the set of TIDs received from processors $R = \{r_1, \ldots, r_k\}$, respectively.

COMPUTE: Coordinator c sets $D_c \leftarrow D_c \cup Z$ and $P_c \leftarrow R$.

SEND: Coordinator c multicasts `summary`(D_c, P_c) to processors in P_c.

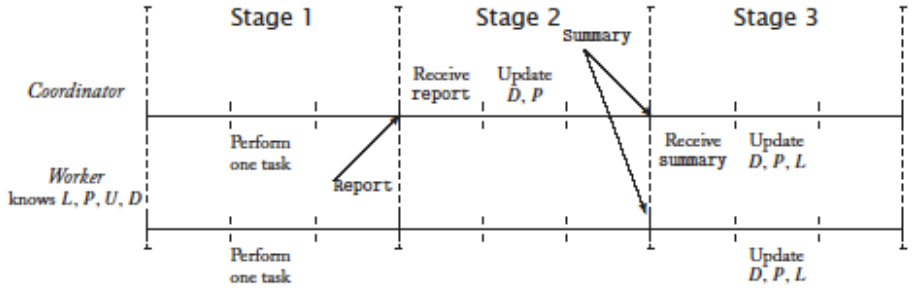
STAGE 3.

RECEIVE: Processor w receives some set {`summary`(D^i, P^i)} of messages (we will see in the analysis that these messages are in fact identical).

COMPUTE: Processor w sets $D_w \leftarrow D^i$ and $P_w \leftarrow P^i$ for an arbitrary i and updates its local view $L_{\ell,w}$ (the update rule is described in detail in the text).

SEND: The send substage is not used.

Εικόνα 4.1 Ψευδοκώδικας Αλγόριθμου AN για ένα εργάτη w σε μια φάση ℓ [1]



Εικόνα 4.2 Σχηματική Αναπαράσταση μιας φάσης ℓ του αλγόριθμου AN [1]

Βάση του αλγορίθμου και της σχηματικής αναπαράστασης σε πρώτη φάση υλοποιήθηκαν πλήρως μέσα στην εφαρμογή μας οι δύο κλάσεις μηνυμάτων, η κάθε

μία εκ των οποίων αναπαριστά το μήνυμα αναφοράς - *ReportMessage* και περίληψης - *SummaryMessage* αντίστοιχα. Η υλοποίηση της κλάσης *ReportMessage* έχει ως εξής:

```
class ReportMessage implements Serializable{

	private static final long serialVersionUID = 1L;

	// ArrayList with Pending Done Work
	ArrayList<String> pending = new ArrayList<String>();

	@SuppressWarnings("unchecked")
	ReportMessage (ArrayList<String> value){
		this.pending = (ArrayList<String>) value.clone();
	}
}
```

όπου μία λίστα *pending* τύπου String θα κρατάει μέσα της την δουλεία που εκκρεμεί, δηλαδή που έγινε από τον επεξεργαστή αλλά ακόμη να αναφερθεί, και ο ρόλος του κατασκευαστή είναι κατά την κλήση του να αντιγράψει την παράμετρο που του αποστέλλεται στην λίστα αυτή. Ομοιόμορφα έχει αναπτυχθεί και η κλάση SummaryMessage με την διαφορά ότι αυτή η κλάση διαχειρίζεται δύο λίστες *Dw* και *Pw* τύπου *String* και *NodeID* αντίστοιχα. Η πρώτη υπονοεί την δουλεία που διεκπεραιώθηκε μέχρι τώρα και η δεύτερη το σύνολο των μοναδικών αναγνωριστικών των ενεργών επεξεργαστών.

Ο αριθμός των εργασιών καθώς και το σύνολο των επεξεργαστών που θα λάβουν μέρος στο κάθε σενάριο στην εκτέλεση της εφαρμογής για δική μας ευκολία καθορίζεται μέσο του αρχείου διαμόρφωσης. Κατά τον πρώτο γύρο γίνεται η αρχικοποίηση των βασικών συστατικών και μεταβλητών για ορθή εκτέλεση του αλγορίθμου. Συγκεκριμένα, όπως παρατηρούμε και στο πιο κάτω κομμάτι κώδικα του AN, αρχικοποιείται το μέγεθος ενός πίνακα δυαδικών τιμών ίσο με τον αριθμό των επεξεργαστών που θα λάβουν μέρος στην εκτέλεση, ο κόμβος με id=1 θέτει τον εαυτό του σαν συντονιστή, γίνονται γνωστοί όλοι οι επεξεργαστές του υπολογισμού σε κάθε επεξεργαστή, το σύνολο των tasks και utasks γίνεται ίσο με όλες τις εργασίες και το σύνολο των ολοκληρωμένων εργασιών θα γίνει ίσο με το κενό σύνολο.

```
if (round == 0) {

     // Initialize the size of our boolean table
     mark_alive = new boolean [nodes.size()];
     // Set Node1 as the coordinator
     if (getNodeId().toString().equals("Node1")) {
          cordinator = true; }

     // Initialiaze all known processors
     for (NodeID neighbor : nodes) {
          aliveproc.add(neighbor); }

     // Initialiaze all tasks to be done
     // Initialiaze all Unaccound tasks to be done
     // Donetasks will be empty
     // Unaccounted tasks same as tasks
     for (int i = 1; i <= no_tasks; i++) {
          tasks.add("task" + i);
          utasks.add("task" + i); }

     donetasks.clear();
}
```

Ακολούθως, μετά τον γύρο της αρχικοποίησης ξεκινάει η υλοποίηση του πρώτου σταδίου του αλγορίθμου μας. Βάση αυτού ένας ενεργός επεξεργαστής θα αναλάβει μία εργασία με χρήση του *Load Balancing Rule*, θα δημιουργήσει ένα μήνυμα αναφοράς και να το αποστείλει σε όλους τους συντονιστές σύμφωνα με το τρέχων Local View του όπως επεξηγείται και στο υποκεφάλαιο 4.2 . Παρουσιάζουμε τον κώδικα που εξυπηρετεί το πρώτο στάδιο του αλγορίθμου.

```
//Get our position from local view.
for(int i=0; i<aliveproc.size(); i++){

if(aliveproc.get(i).toString().equals(getNodeId().toString(
))){
     l_pos=i;
     break;
     }
}

     //Take the job with Load Balance rule.
     //We must remove it from unaccounted tasks
     //and add it in pending tasks list.

     job = utasks.get(l_pos % utasks.size());
```

```
        utasks.remove(l_pos % utasks.size());
        p_tasks.add(job);

// Create and send Report message to all
// Coordinators corresponding to our local view.
ReportMessage ms = new ReportMessage(p_tasks);


for(int i=0; i<Math.pow(2, level) && i<aliveproc.size();
i++){
        try {
                getCommLayer().sendTCP(ms, aliveproc.get(i));
                total_msgs++;

        } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
                }
        }
```

Εφόσον οι συντονιστές έλαβαν όλα τα μηνύματα αναφοράς και τα επεξεργάστηκαν ανακτώντας από αυτά τις αναγκαίες πληροφορίες όπως θα δούμε σε μετέπειτα στάδιο, λαμβάνει χώρο το δεύτερο στάδιο του αλγορίθμου AN. Κατ' αυτό, ένας συντονιστής δημιουργεί μήνυμα περίληψης το οποίο περιέχει όλη την διεκπεραιωμένη εργασία *donetasks* μέχρι την τρέχον στιγμή και το σύνολο των ενεργών επεξεργαστών *p* που έλαβε μηνύματα αναφοράς. Μετά την δημιουργία αυτό το μήνυμα αποστέλλεται σε όλους τους ενεργούς προς αυτόν επεξεργαστές για να αναβαθμίσουν την τοπική τους γνώση στο επόμενο στάδιο.

```
        if(cordinator){

        //create messages with alive proc and done work
        SummaryMessage m3 = new SummaryMessage(donetasks,p);

                //Send the Summary message to all alive nodes
                for(int i=0; i<p.size(); i++){

                        try {
                        getCommLayer().sendTCP(m3, p.get(i));
                        total_msgs++;
                        } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                        }
                }
```

```
        //empty the alive nodes list
        p.clear();
        cordinator=false;
    }
```

Σε τρίτο στάδιο του αλγορίθμου AN αυτό που απομένει είναι να γίνει λήψη μηνυμάτων περίληψης από τους συντονιστές και βάση αυτής της λήψης ο κάθε επεξεργαστής να αναβαθμίσει την γνώση του για τον αριθμό των ολοκληρωμένων εργασιών, το σύνολο των ενεργών επεξεργαστών και την δομή του Local View του (κανόνας ενημέρωσης Local View). Εάν δεν γίνει καμία λήψη μηνύματος περίληψης από τους συντονιστές τότε η φάση θεωρείται *unattended*, ο κάθε επεξεργαστής διαγράφει τους τρέχων σύμφωνα με το Local View του συντονιστές για αυτή την φάση και αυξάνει το επίπεδο των απαιτούμενων συντονιστών.

```
    if(!attended){

    //Delete supposed coordinator for this
    //current phase according to our Local View.

        for(int i=(int) Math.pow(2, level) -1; (i>=0 &&
        i<aliveproc.size());  i--){
            aliveproc.remove(i);
        }

        //Increase our coordinators level.
        level++;
        System.out.println("Phase was Unattended);

    }else{

        //phase is Attended.
        //We must empty our pending tasks list.
        System.out.println("** Phase was Attended **");
        p_tasks.clear();
    }
```

Τελικό στάδιο της ανάλυσης είναι η παρουσίαση του κώδικα για την διαχείριση των εισερχόμενων μηνυμάτων, είτε αυτά ήταν τύπου αναφοράς - *ReportMessge* είτε τύπου περίληψης - *SummaryMessage*. Η κάθε μία από τις περιπτώσεις απαιτούσε διαφορετική διαχείριση. Σε πρώτο στάδιο παρουσιάζεται ο κώδικας για την λήψη μηνύματος αναφοράς.

```
if (msg instanceof ReportMessage) {

     cordinator=true;
     ReportMessage m=(ReportMessage)msg;

     for(int i=0; i<m.pending.size(); i++){

          if(!donetasks.contains(m.pending.get(i)))
               donetasks.add(m.pending.get(i));
          }

     p.add(sender);
}
```

Όπως κατανοούμε και από τα σχόλια του κώδικα, λαμβάνοντας τέτοιου είδους μήνυμα θα πρέπει να ενεργήσουμε σαν συντονιστές, δηλαδή να συμπληρώσουμε την ολοκληρωμένη δουλεία που αποστέλλει ο κάθε επεξεργαστής και να τοποθετήσουμε τον αποστολέα στους ενεργούς προς τον συντονιστή επεξεργαστές της τρέχων φάσης. Σε δεύτερο στάδιο παρουσιάζουμε τον κώδικα για την διαχείριση των εισερχομένων μηνυμάτων περίληψης.

```
if (msg instanceof SummaryMessage){

     attended=true;
     level=0;

     //Get the message to a handler
     SummaryMessage m =(SummaryMessage)msg;

     // Update our knowledge for Done tasks
     // and Utasks. Unaccounted = Tasks - Done
     for(int i=0; i<m.Dw.size(); i++){

          if(!donetasks.contains(m.Dw.get(i)))
               donetasks.add(m.Dw.get(i));

          if(utasks.contains(m.Dw.get(i)))
               utasks.remove(m.Dw.get(i));
          }


     //Initialize all processors as dead.
     //At next mark alive ones one by one.
     for(int k=0; k<mark_alive.length; k++)
          mark_alive[k]=false;
```

```
            //mark all alive processors at boolean
            //mark_alive table while reading them.

            for(int i=0; i<m.Pw.size(); i++){
                  int node_pos = Integer.valueOf
            (m.Pw.get(i).toString().replace("Node", ""))-1;

                  mark_alive [node_pos] = true;
            }


            //after marking all alive we have to add them
            //sorted to the aliveproc list.
            int pos=0;

            for (NodeID neighbor : nodes) {

            //if it's alive we add him to the list
            if(mark_alive[pos]==true){
                  if(!aliveproc.contains(neighbor))
                        aliveproc.add(neighbor);
                  }

                  pos++;
            }

      }
```

Συνοπτικά, αυτό που εφαρμόζεται κατά την λήψη μηνύματος περίληψης είναι να θεωρήσουμε την φάση ως *attended* (κάποιος συντονιστής είναι ενεργός για να λάβουμε αυτό το μήνυμα). Ακολούθως, να θέσουμε το επίπεδο ίσο με μηδέν, να αναβαθμίσουμε την γνώση μας για τις ολοκληρωμένες η όχι εργασίες και να γεμίσουμε την λίστα με το σύνολο των ενεργών επεξεργαστών βάση της πληροφορίας του μηνύματος.

Ο πλήρης με επαρκές σχόλια κώδικας για τον αλγόριθμο AN επισυνάπτεται στο τέλος της εργασίας στο Παράρτημα Α.

4.3 Αλγόριθμος AR

Στο συγκεκριμένο υποκεφάλαιο αρχικά θα γίνει αρχικά μια σύντομη και συγχρόνως περιεκτική περιγραφή του αλγορίθμου AR, του δεύτερου αλγορίθμου με τον οποίο ασχολείται η παρούσα εργασία και στη συνέχεια θα παρουσιαστούν και θα αναλυθούν σημαντικές λεπτομέρειες σε θέματα υλοποίησης του αλγορίθμου αυτού.

4.3.1 Περιγραφή Αλγορίθμου AR

Βασισμένος στην ιδέα του αλγόριθμου AN, ο AR είναι και αυτός ένας αλγόριθμος που επιλύει το πρόβλημα *Do-All (βλ. Υποκεφάλαιο 2.1)* και ανήκει στο Μοντέλο Μεταβίβασης Μηνυμάτων [1-3, 10]. Όπως αναφέραμε και σε προηγούμενο στάδιο της εργασίας (*βλ. Υποκεφάλαιο 4.2*), βάση το μοντέλο αυτό, όλες οι υπολογιστικές οντότητες επικοινωνούν μεταξύ τους με ανταλλαγή μηνυμάτων για την διεκπαιρέωση των εργασιών. Ο αλγόριθμος AR συνδυάζει όπως και ο AN τοπικές δομές δεδομένων και την τεχνική πολλαπλών συντονιστών όπως περιγράφονται πλήρες στην ενότητα 3 του [2].

Ο αλγόριθμος AR είναι μία επέκταση του αλγόριθμου AN στην οποία προστέθηκαν επιπλέον είδη μηνυμάτων έτσι ώστε να είναι ικανός να διαχειρίζεται τυχών επανεκκινήσεις υπολογιστικών οντοτήτων κατά τον υπολογισμό. Σε απουσία επανεκκινήσεων ο αλγόριθμος AR συμπεριφέρεται πανομοιότυπα όπως και ο αλγόριθμος AN.

Έχει αποδειχθεί ότι ο αλγόριθμος AR είναι ανεκτικός σε f καταρρεύσεις επεξεργαστών (f < p) και ανεκτικός σε πρότυπα καταρρεύσεων/επανεκκινήσεων 26-περιορισμένα. Με τον όρο κ-περιορισμένα υπονοούμε το πρότυπο καταρρεύσεων κατά το οποίο υπάρχει τουλάχιστον ένας ζωντανός επεξεργαστής σε κ ακολουθιακά βήματα του αλγορίθμου. Ο αλγόριθμος AR είναι Έχει πολυπλοκότητα Εργασίας *S* όπου S = $O\big((n + p\ log\ p + f) \cdot \min\{\log p\ , \log f\ \big)$ και πολυπλοκότητα Μηνυμάτων *M* όπου M = $O(n + p\ log\ p + fp)$.

4.3.2 Δομές Δεδομένων Και Φάσεις AR

Συντονιστές και Εργάτες

Ομοίως με τον αλγόριθμο AN, έτσι και ο AR εκτελείτε μέσα σε ένα επαναληπτικό βρόγχο ως ότου να εκτελεστούν όλες οι εργασίες. Μια επανάληψη του βρόγχου ονομάζεται φάση. Μια φάση αποτελείται από τρία διαδοχικά στάδια. Κάθε στάδιο απαρτίζεται από τρία βήματα, άρα μια φάση συνολικά αποτελείται από 9 βήματα. Σε κάθε στάδιο το πρώτο βήμα χρησιμοποιήστε για παραλαβή μηνυμάτων, το δεύτερο βήμα για εκτέλεση τοπικών υπολογισμών και το τρίτο για αποστολή μηνυμάτων.

Ο κάθε επεξεργαστής μπορεί να είναι συντονιστής σε κάποια φάση και όλοι οι επεξεργαστές είναι εργάτες σε κάθε φάση. Ο συντονιστής είναι υπεύθυνος για την καταγραφή της προόδου καθώς οι εργάτες εκτελούν μία εργασία και την αναφέρουν σε αυτόν. Εάν τουλάχιστον ένας επεξεργαστής ενεργεί σαν συντονιστής καθ' όλη την διάρκεια μίας φάσης τότε η φάση αυτή ονομάζεται *attended* αλλιώς *unattended*. Ο αριθμός των επεξεργαστών που θα θεωρηθούν σαν συντονιστές καθορίζονται από την Αρχή Martingale. Συγκεκριμένα αν κανένας από τους αναμενόμενους συντονιστές δεν επιζήσει σε μία φάση τότε διπλασιάζεται ο αριθμός αυτών στην επόμενη φάση. Όταν έστω και ένας από τους coordinators επιζήσει σε μια ολόκληρη φάση τότε ο αριθμός των coordinators μειώνεται σε 1.

Local Views

Οι επεξεργαστές υποθέτουν τον ρόλο ενός coordinator βασισμένοι στην τοπική τους γνώση *(Loal View).* Ο συμβολισμός του Local View είναι $L_{\ell w}$, όπου ℓ είναι η φάση και *w* το μοναδικό αναγνωριστικό του κάθε εργάτη. Κατά τον υπολογισμό κάθε worker *w* διαχειρίζεται μια λίστα *Lw = <q1, q2 .. qn>* με τους υποτιθέμενους ζωντανούς επεξεργαστές. Στην συγκεκριμένη λίστα *L* οι επεξεργαστές είναι χωρισμένοι σε στρώματα. $Lw = <\Lambda^0, \Lambda^1 ... >$ όπου σε κάθε στρώμα *i* έχουμε διπλάσιους επεξεργαστές από το προηγούμενο. Εξαίρεση αποτελεί το τελευταίο επίπεδο το οποίο μπορεί να έχει λιγότερους επεξεργαστές από το *i – 1* επίπεδο, αυτό οφείλεται στα

χαρακτηριστικά της δεντρικής δομής που χρησιμοποιεί ο αλγόριθμος. Σημαντική διαφορά από τον αλγόριθμο AN είναι το γεγονός ότι οι επεξεργαστές μέσα στο *Local View* δεν εμφανίζονται κατ' ανάγκη με την σειρά των αναγνωριστικών τους. Αυτό οφείλεται στο ότι επεξεργαστές που θα επανεκκινήσουν σε κάποια δεδομένη στιγμή τοποθετούνται ταξινομημένα στο τέλος του Local View σε σχέση με όλους τους άλλους επανεκκινημένους επεξεργαστές της δεδομένης φάσης.

Κανόνας ενημέρωσης Local View

Το Local View ενημερώνεται ανάλογα με το είδος της φάσης ℓ.

- Φάση ℓ είναι attended: Έστω R^{ℓ} το σύνολο των επεξεργαστών που επανεκκινήσαν. Εφόσον η φάση είναι attended τότε μηνύματα περίληψης από τους συντονιστές παραλήφθηκαν από όλους τους επεξεργαστές. Τότε ο κάθε επεξεργαστής *w* υπολογίζει το σύνολο *Pw* όπως περιγράφεται στο στάδιο 3 του αλγορίθμου. Το Local View για την επόμενη φάση $L_{\ell+1,w}$ του *w* θα έχει δενδρική δομή περιλαμβάνοντας όλους τους ενεργούς και επανεκκινημένους επεξεργαστές ταξινομημένους βάση του αναγνωριστικού τους, $Pw \cup R^{\ell}$.

- Φάση ℓ είναι unattended: Έστω R^{ℓ} το σύνολο των επεξεργαστών που επανεκκινήσαν. Έστω R' το σύνολο των επεξεργαστών R^{ℓ} οι οποίοι δεν βρίσκονται μέσα στο Local View $L_{\ell w}$. Έστω $<R^{\ell}>$ το σύνολο των επεξεργαστών στο R^{ℓ} ταξινομημένοι βάση του αναγνωριστικού τους. Τότε το Local View για την επόμενη φάση $L_{\ell+1,w}$ θα είναι $<\Lambda^1 \dots \Lambda^j> \oplus <R'>$. Ο τελεστής $\oplus$ τοποθετεί το σύνολο των επεξεργαστών $<R'>$ ταξινομημένους στο τελευταίο επίπεδο Λ^j .

Δομή της φάσης και δέσμευση εργασίας

Η Γενική δομή κάθε φάσης του αλγορίθμου είναι η ίδια με αυτή του AN. Οι κύριες διαφορές έχουν ως εξής:

- Εάν ένας επεξεργαστής επανεκκινήσει τότε ενημερώνει όλους τους επεξεργαστές για αυτό το γεγονός.
- Κάθε επεξεργαστής που κάνει επανεκκίνηση σε μία δεδομένη φάση δεν θεωρείται διαθέσιμος εφόσον υπάρχει πιθανότητα να μην έχει πλήρως ενημερωμένη γνώση για τον τρέχων υπολογισμό.
- Βασισμένοι στα μηνύματα επανεκκίνησης, αυτοί οι επεξεργαστές ενσωματώνονται στην δομή του Local View του κάθε ενεργού επεξεργαστή και γίνονται διαθέσιμοι για τον υπολογισμό από την επόμενη φάση.

4.3.3 Λεπτομέρειες Υλοποίησης AR

Πιο κάτω μας παρουσιάζεται ο ψευδοκώδικας βάση του οποίου έγινε η υλοποίηση του αλγ1ορίθμου AN (Εικόνα 4.3) και τα στάδια που απαρτίζουν μία ολοκληρωμένη φάση ℓ (Εικόνα 4.4).

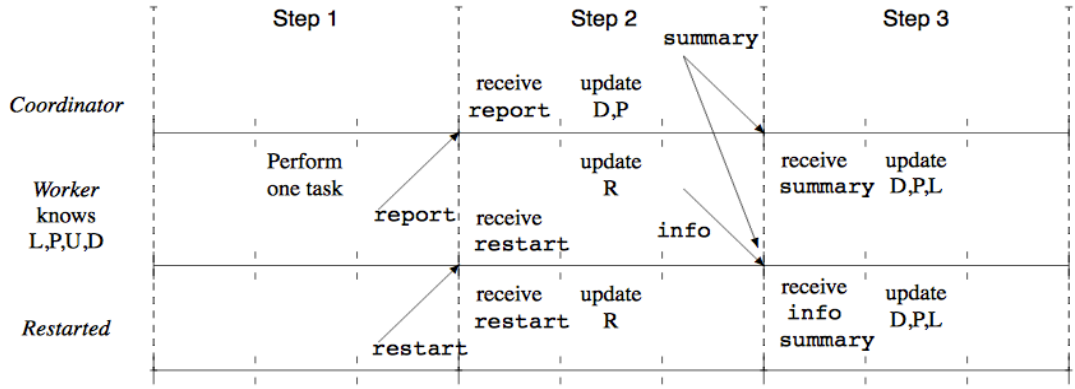
Phase ℓ of algorithm AR:

STAGE 1. RECEIVE: The receive substage is not used.
COMPUTE: In the compute substage any processor w performs a specific task z according to the load balancing rule.
SEND: In the send substage w sends a `report`(z) to any coordinator, that is, to any processor in the first layer of $L_{\ell,w}$.
Any restarted processor q broadcasts the `restart`(q) *message informing all live processors of its restart.*

STAGE 2. RECEIVE: In the receive substage the coordinators gather `report` messages *and all processors gather* `restart` *messages. Let R be the set of processors that sent a* `restart` *message.* For any coordinator c, let $z_c^1, ..., z_c^{k_c}$ be the set of TIDs received in `report` messages.
COMPUTE: In the compute substage c sets $D_c \leftarrow D_c \cup \bigcup_{i=1}^{k_c} \{z_c^i\}$ and P_c to the set of processors from which c received `report` messages.
SEND: In the send substage, coordinator c multicasts the message `summary`(D_c, P_c) to the processors in P_c *and R. Any processor in P_c sends the message* `info`(U_ℓ, L_ℓ) *to processors in R.*

STAGE 3. RECEIVE: *In the receive substage processors in R receive* `info`(U_ℓ, L_ℓ) *messages and processors in P_c and R receive* `summary`(D_c, P_c) *messages.*
COMPUTE: *In the compute substage, a restarted processor q sets $L_{\ell,q} \leftarrow L_\ell$ and $U_{\ell,q} \leftarrow U_\ell$.* Let $(D_w^1, P_w^1), ..., (D_w^{k_w}, P_w^{k_w})$ be the sets received in `summary` messages by processor w. Processor w sets $D_w \leftarrow D_w^i$ and $P_w \leftarrow P_w^i$ for an arbitrary $i \in 1, ..., k_w$ and updates its local view $L_{\ell,w}$ *as described below.*
SEND: The send substage is not used.

Εικόνα 4.3 Ψευδοκώδικας Αλγόριθμου AR για ένα εργάτη w σε μια φάση ℓ [1]



Εικόνα 4.4 Σχηματική Αναπαράσταση μιας φάσης ℓ του αλγόριθμου AR [1]

Βάση του αλγορίθμου AR και της σχηματικής αναπαράστασης σε πρώτη φάση υλοποιήθηκαν πλήρως μέσα στην εφαρμογή μας οι τέσσερις κλάσεις μηνυμάτων, η κάθε μία εκ των οποίων αναπαριστά το μήνυμα αναφοράς - *ReportMessage*, περίληψης - *SummaryMessage,* μήνυμα πληροφορίας - *InformationMessage* και μήνυμα επανεκκίνησης - *RestartMessage*. Η υλοποίηση του *ReportMessage* έχει ως εξής:

```
class ReportMessage implements Serializable{

	private static final long serialVersionUID = 1L;

	// ArrayList with Pending Done Work
	ArrayList<String> pending = new ArrayList<String>();

	@SuppressWarnings("unchecked")
	ReportMessage (ArrayList<String> value){
		this.pending = value.clone();
	}
}
```

όπου μία λίστα *pending* τύπου String θα κρατάει μέσα της την δουλεία που εκκρεμεί, δηλαδή που έγινε από τον επεξεργαστή αλλά ακόμη να αναφερθεί. Ο ρόλος του κατασκευαστή είναι κατά την κλήση του να αντιγράψει την παράμετρο που του αποστέλλεται στην λίστα αυτή. Ομοιόμορφα έχουν υλοποιηθεί και οι κλάσεις SummaryMessage, InformationMessage και RestartMessage η κάθε μία εκ των οποίων εξυπηρετεί τον δικό της σκοπό βάση του αλγορίθμου.

Οι αρχικοποιήσεις που λαμβάνουν χώρο είναι ο ίδιες με αυτές του αλγορίθμου AN (βλέπε υποκεφάλαιο 4.2) και επιπρόσθετα αρχικοποιείται και ένας δυαδικός πίνακας για το πλήθος των επεξεργαστών που επανεκκινήσαν κάθε θέση του οποίου είναι ίση με την τιμή false. Ακολούθως, μετά τον γύρο της αρχικοποίησης ξεκινάει η υλοποίηση του πρώτου σταδίου του αλγορίθμου μας. Βάση αυτού ένας ενεργός επεξεργαστής θα αναλάβει μία εργασία με χρήση του *Load Balancing Rule*, θα δημιουργήσει ένα μήνυμα αναφοράς και να το αποστείλει σε όλους τους συντονιστές σύμφωνα με το τρέχων Local View του όπως επεξηγείται και στο υποκεφάλαιο 4.3. Σε αυτό το γύρο ένας επεξεργαστής που επανεκκίνει θα αποστείλει μήνυμα επανεκκίνησης σε όλους τους επεξεργαστές. Παρουσιάζουμε το επιπρόσθετο κομμάτι κώδικα από τον αλγόριθμο AN που εξυπηρετεί το πρώτο στάδιο του αλγορίθμου AR.

```
if(restarted){

        restarted.clear();
        aliveproc.clear();

        //Create and send Restart message to all nodes
        RestartMessage m = new RestartMessage();

        for (NodeID neighbor : nodes) {
            try {
                getCommLayer().sendTCP(m,neighbor);
                total_msgs++;
            } catch (IOException e) {
            // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }

}
```

Εφόσον οι συντονιστές έλαβαν όλα τα μηνύματα αναφοράς και όλοι επεξεργαστές έλαβαν μηνύματα επανεκκίνησης, τα επεξεργάστηκαν ανακτώντας από αυτά τις αναγκαίες πληροφορίες όπως θα δούμε σε μετέπειτα στάδιο, λαμβάνει χώρο το δεύτερο στάδιο του αλγορίθμου AR. Σύμφωνα με αυτό ένας συντονιστής δημιουργεί μήνυμα περίληψης και το αποστέλλει προς όλους τους επεξεργαστές, και

ο κάθε εργάτης δημιουργεί μήνυμα πληροφορίας που το αποστέλλει σε κάθε επεξεργαστή που γνωρίζει ότι επανεκκίνησε κατά την αρχή της φάσης τρέχων φάσης.

```
if(cordinator){

      //create message with alive processors and done work
      SummaryMessage m3 = new SummaryMessage(donetasks,p);

      //Send the Summary msg to all alive nodes
      for(int i=0; i<p.size(); i++){
            try {
                  getCommLayer().sendTCP(m3, p.get(i));
                  total_msgs++;
            } catch (IOException e) {
                  e.printStackTrace();
            }
      }

      //Send the Summary msg to all restarted nodes
      for(int i=0; i<restarted.size(); i++){
            try {
            getCommLayer().sendTCP(m3, restarted.get(i));
                  total_msgs++;
            } catch (IOException e) {
                  e.printStackTrace();
            }
      }
}

//create message with info (Utasks and Local View)
//Send the Info message messages to all restarted nodes
InfoMessage ms = new InfoMessage(utasks,aliveproc);

for(int i=0; i<restarted.size(); i++){
      try {
      getCommLayer().sendTCP(ms, restarted.get(i));
            total_msgs++;
      } catch (IOException e) {
            e.printStackTrace();
      }
}
```

Για το τρίτο και τελικό στάδιο του αλγορίθμου AR αυτό που απομένει είναι να γίνει λήψη μηνυμάτων περίληψης από τους συντονιστές. Βάση αυτής της λήψης ο κάθε επεξεργαστής θα αναβαθμίσει την τοπική του γνώση. Εκτός από τα μηνύματα των συντονιστών οι επεξεργαστές που επανεκκινήσαν θα λάβουν και μηνύματα

πληροφορίας. Εάν δεν γίνει καμία λήψη μηνύματος περίληψης τότε η φάση θα θεωρηθεί *unattended*. Ο κάθε επεξεργαστής αναβαθμίζει την τοπική του γνώση και το Local View βάση του κανόνα που περιγράψαμε στο υποκεφάλαιο 4.3.2. Πιο κάτω παρουσιάζεται ο κώδικας που διαχειρίζεται τον κανόνα ενημέρωσης του Local View σε περίπτωση της φάσης που είναι unattended.

```
if(!attended){

      for(int i=(int) Math.pow(2, level) -1; (i>=0 &&
      i<aliveproc.size()); i--){
            aliveproc.remove(i);}

      // Add Restarted processors sorted by their ID's
      // at the end of local view
      // Initialize all processors as not restarted.
      // At next mark restarted ones one by one.
      for(int k=0; k<mark_restarted.length; k++)
            mark_restarted[k]=false;

      for(int i=0; i<restarted.size(); i++){

            int node_pos = Integer.valueOf(restarted.get(i).
            toString().replace("Node", ""))-1;
            mark_restarted [node_pos] = true;}

      int pos=0;
      for (NodeID neighbor : nodes) {
            if(mark_restarted[pos]==true){
                        if(!aliveproc.contains(neighbor))
                        aliveproc.add(neighbor);
            }
            pos++;
      }
   }
```

Τελικό στάδιο της ανάλυσης είναι η παρουσίαση του κώδικα για την διαχείριση των εισερχόμενων μηνυμάτων. Η διαχείριση μηνυμάτων τύπου αναφοράς *(ReportMessge)* και τύπου περίληψης *(SummaryMessage)* παραμένει η ίδια όπως και του αλγορίθμου AN. Σε πρώτο στάδιο παρουσιάζεται ο κώδικας για την λήψη μηνύματος πληροφορίας *(InformationMessage)*. Κάποιος που λαμβάνει αυτού του

είδους μήνυμα γνωρίζει ότι είναι επεξεργαστής που επανεκκίνησε, άρα θα πρέπει να ενημερώσει το Local View του και το σύνολο των ολοκληρωμένων εργασιών σύμφωνα με τις πληροφορίες που του παρέχει το μήνυμα πληροφορίας.

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

if (msg instanceof InfoMessage){

     /*
      * Since we get this kind of a message
      * we know that we are a restarted processor.
      * Hence we have to update our Local view
      * and Unaccounted tasks
      */

     InfoMessage m = (InfoMessage) msg;

     for(int i=0; i<m.Lw.size(); i++){
          if(!aliveproc.contains(m.Lw.get(i)))
               aliveproc.add(m.Lw.get(i));}

     for(int i=0; i<m.Uw.size(); i++){
          if(!utasks.contains(m.Uw.get(i)))
               utasks.add(m.Uw.get(i));}
          alive=true
     }
```

Σε δεύτερο και τελικό στάδιο παρουσιάζουμε τον κώδικα για την διαχείριση των εισερχομένων μηνυμάτων επανεκκίνησης *(RestartMessage)*. Η μόνη ενέργεια που λαμβάνει χώρο είναι η εισαγωγή του αποστολέα στην λίστα με τους επεξεργαστές που επανεκκινήσαν.

```
     if (msg instanceof RestartMessage){

               restarted.add(sender);
     }
```

Ο πλήρης με επαρκές σχόλια κώδικας για τον αλγόριθμο AR επισυνάπτεται στο τέλος της εργασίας στο Παράρτημα Β.

4.4 Αλγόριθμος Do-UM

Στο συγκεκριμένο υποκεφάλαιο αρχικά θα γίνει αρχικά μια σύντομη και συγχρόνως περιεκτική περιγραφή του αλγορίθμου Do-Um, του τρίτου και τελευταίου αλγορίθμου με τον οποίο ασχολείται η παρούσα εργασία και στη συνέχεια θα παρουσιαστούν και θα αναλυθούν σημαντικές λεπτομέρειες σε θέματα υλοποίησης του αλγορίθμου αυτού.

4.4.1 Περιγραφή Αλγορίθμου Do-UM

Ο αλγόριθμος *Do-Um* είναι και αυτός ένας αλγόριθμος που επιλύει το πρόβλημα *Do-All (βλ. Υποκεφάλαιο 2.1)* και ανήκει στο Μοντέλο Μεταβίβασης Μηνυμάτων [1-3, 10]. Ο αλγόριθμος Do-Um συνδυάζει όπως και οι AR - AN τοπικές δομές δεδομένων και κάνει χρήση την τεχνική πολλαπλών συντονιστών όπως περιγράφονται πλήρες στην ενότητα 3 του [2].

Ο αλγόριθμος *Do-Um* σε αντίθεση με τους δύο προηγούμενους αλγόριθμους που μελετήθηκαν επιλύει το πρόβλημα *Do-All* χωρίς την απαίτηση δικτύου με αξιόπιστη πολυεκπομπή.

Έχει αποδειχθεί ότι ο αλγόριθμος *Do-Um* είναι ανεκτικός σε f καταρρεύσεις επεξεργαστών ($f < p$). Η πολυπλοκότητα εργασίας S του αλγορίθμου αυτού είναι αρκετά αποδοτική όμως η πολυπλοκότητα μηνυμάτων του M είναι απαγορευτική.

4.4.2 Δομές Δεδομένων Και Φάσεις Do-UM

Συντονιστές και Εργάτες

Ο αλγόριθμος Do-Um εκτελείτε μέσα σε ένα επαναληπτικό βρόγχο ως ότου να εκτελεστούν όλες οι εργασίες. Μια επανάληψη του βρόγχου αποτελείται από δύο διαδοχικούς γύρους. Συγκεκριμένα τον γύρο Συλλογής *(Collect Round)* και τον γύρο Διάδοσης *(Disseminate Round)*. Κάθε γύρος απαρτίζεται από τρία βήματα*,* άρα μια επανάληψη συνολικά αποτελείται από 6 βήματα*.* Σε κάθε γύρο το πρώτο βήμα χρησιμοποιήστε για αποστολή μηνυμάτων*,* το δεύτερο βήμα για παραλαβή μηνυμάτων και το τρίτο βήμα για εκτέλεση τοπικών υπολογισμών.

Ο κάθε επεξεργαστής μπορεί να είναι συντονιστής σε κάποια φάση και όλοι οι επεξεργαστές είναι εργάτες σε κάθε φάση. Ο συντονιστής είναι υπεύθυνος για την καταγραφή της προόδου καθώς οι εργάτες εκτελούν μία εργασία και την αναφέρουν σε αυτόν. Ένας επεξεργαστής ενεργεί σαν συντονιστής εάν λάβει μήνυμα αναφοράς ή όταν πιστεύει βάση της γνώσης του ότι πρέπει να ενεργεί σαν συντονιστής. Ο αριθμός των επεξεργαστών που θα θεωρηθούν σαν συντονιστές καθορίζονται από την Αρχή Martingale*.* Συγκεκριμένα αν κανένας από τους αναμενόμενους coordinators δεν επιζήσει σε μία φάση τότε διπλασιάζεται ο αριθμός των coordinators στην επόμενη φάση. Όταν έστω και ένας από τους συντονιστές επιζήσει σε μια ολόκληρη φάση τότε ο αριθμός αυτών μειώνεται σε 1. Εάν ένας επεξεργαστής λάβει τουλάχιστον ένα μήνυμα περίληψης τότε θεωρεί την φάση *attended* και σε αντίθετη περίπτωση *unattended*.

Local Views

Έστω *P* το σύνολο όλων των επεξεργαστών και *F* το σύνολο των εσφαλμένων επεξεργαστών. Κατά τον υπολογισμό κάθε επεξεργαστής διαχειρίζεται μια λίστα *L = <q1, q2 .. qn>* με τους υποτιθέμενους ζωντανούς επεξεργαστές. Ο υπολογισμός του L γίνεται βάση του P - F. Στην συγκεκριμένη λίστα *L* οι επεξεργαστές είναι χωρισμένοι σε στρώματα *(Layers)*. *Lw = <*Λ^0 *,* Λ^1 *... >* όπου σε κάθε στρώμα *i* έχουμε

διπλάσιους επεξεργαστές από το προηγούμενο. Εξαίρεση αποτελεί το τελευταίο επίπεδο το οποίο μπορεί να έχει λιγότερους επεξεργαστές από το *i – 1* επίπεδο, αυτό οφείλεται στα χαρακτηριστικά της δεντρικής δομής που χρησιμοποιεί ο αλγόριθμος.

Δομή της φάσης και δέσμευση εργασίας

Γενική γνώση κάθε επεξεργαστή:

- *T* – Σύνολο όλων των εργασιών που πρέπει να γίνουν.
- *P* – Σύνολο όλων των επεξεργαστών.

Κάθε επεξεργαστής διαχειρίζεται:

- *D* - Την τοπική πληροφορία για το σύνολο εργασιών που έγιναν.
- *F* – Σύνολο των εσφαλμένων επεξεργαστών που γνωρίζει.
- *L* - Η δομή των συντονιστών.
- *level* – το τρέχων επίπεδο στο L.
- *C* – τελευταίοι ενεργοί συντονιστές.
- *R* – Σύνολο των *Report Messages* που παραλήφθηκαν.
- *S* – Σύνολο των *Summary Messages* που παραλήφθηκαν.

Γύρος Συλλογής (Collect Round) – Κάθε επεξεργαστής στέλνει ένα μήνυμα αναφοράς σε όλους τους συντονιστές. Αυτοί με την σειρά τους συλλέγουν αυτά τα μηνύματα και υπολογίζουν βάση αυτών το καινούργιο σύνολο για τις ολοκληρωμένες εργασίες D και το σύνολο εσφαλμένων επεξεργαστών F.

Γύρος Διάδοσης (Disseminate Round) – Κάθε επεξεργαστής ο οποίος θεωρεί τον εαυτό του συντονιστή ή έλαβε μήνυμα αναφοράς στον γύρο συλλογής στέλνει μηνύματα περίληψης σε όλους του μη εσφαλμένους επεξεργαστές. Τα μηνύματα αυτά λαμβάνονται από τους επεξεργαστές στο βήμα παραλαβής. Στο βήμα υπολογισμού ενημερώνονται τα σύνολα *D* και *F* βάση της πληροφορίας των μηνυμάτων και αφαιρούνται οι εσφαλμένοι συντονιστές από την δομή L . Κάθε επεξεργαστής σε αυτό το σημείο αναλαμβάνει να εκτελέσει μία καινούργια εργασία

με βάση τον κανόνα *Load Balancing*. Βάση του κανόνα *Bal (T – D , P – F , i)* για κάθε επεξεργαστή *p*: κατατάσσει τις εργασίες στο σύνολο *T-D* και τους επεξεργαστές στο σύνολο *P-F* βάση των id's τους. Έστω ότι ο επεξεργαστής έχει την θέση *r* στο σύνολο *P-F* τότε επιλέγει να εκτελέσει την *(r mod |T-D|)*^{οστή} εργασία.

4.4.3 Λεπτομέρειες Υλοποίησης Do-Um

Πιο κάτω μας παρουσιάζεται ο ψευδοκώδικας βάση του οποίου έγινε η υλοποίηση του αλγορίθμου *Do-Um* (Εικόνα 4.5).

```
repeat
  COLLECT ROUND
    Send:
      send report(D, F, i) to all j ∈ L(ℓ)
    Receive:
      R := set of received report messages
    Compute:
      D := D ∪ ⋃_{m∈R} m.D
      F := F ∪ ⋃_{m∈R} m.F
  DISSEMINATE ROUND
    Send:
      If i ∈ L(ℓ) or R ≠ ∅ then
        send summary(D, F, i) to all j ∈ P − F
    Receive:
      S := the set of received summary messages
      C := {m.pid | m ∈ S} /* Acting coordinators */
    Compute:
      D := D ∪ ⋃_{m∈S} m.D
      F := F ∪ ⋃_{m∈S} m.F
      F := F ∪ L(ℓ) \ C
      If (C ≠ ∅) then /* Coordinators attended */
        L := P − F
        ℓ = 0
      Else /* No coordinator attended */
        ℓ := ℓ + 1
      Let τ be Bal(T − D, P − F, i)
      Perform task τ
      D := D ∪ {τ}
until D = T
```

Εικόνα 4.5 Ψευδοκώδικας Αλγόριθμου Do-Um για ένα επεξεργαστή p_i [3]

Βάση του αλγορίθμου, σε πρώτη φάση υλοποιήθηκαν πλήρως μέσα στην εφαρμογή μας οι δύο κλάσεις μηνυμάτων, η κάθε μία εκ των οποίων αναπαριστά το μήνυμα αναφοράς και περίληψης αντίστοιχα. Η λογική της υλοποίησης των δύο αυτών κλάσεων είναι η ίδια με αυτή του αλγορίθμου AN *(βλ. υποκεφάλαιο 4.2)*. Βασική διαφορά είναι ότι και τα δύο είδη μηνυμάτων εδώ είναι τρίκλινα. Συγκεκριμένα, αποτελούνται από το σύνολο D, F και το αναγνωριστικό του επεξεργαστή αποστολέα *pid*.

Ο αριθμός των εργασιών καθώς και το σύνολο των επεξεργαστών που θα λάβουν μέρος στο κάθε σενάριο στην εκτέλεση της εφαρμογής για δική μας ευκολία καθορίζεται μέσο του αρχείου διαμόρφωσης. Κατά τον πρώτο γύρο γίνονται γνωστοί όλοι οι επεξεργαστές του υπολογισμού σε κάθε επεξεργαστή και αρχικοποιείται το σύνολο των εργασιών tasks έτσι ώστε να έχουν μοναδικό αναγνωριστικό. Το σύνολο των ολοκληρωμένων εργασιών αρχικά είναι ίσο με το κενό σύνολο.

```
// Initialiaze all known processors
for (NodeID neighbor : nodes) {
     aliveproc.add(neighbor);
     processors.add(neighbor);

}

// Initialiaze all tasks to be done
for (int i = 1; i <= no_tasks; i++) {
     tasks.add("task" + i);
}

// Donetasks will be empty beacuse of initialize
donetasks.clear();
```

Ακολούθως, μετά τον γύρο της αρχικοποίησης ξεκινάει η υλοποίηση του γύρου Συλλογής. Βάση αυτού ένας ενεργός επεξεργαστής στέλνει ένα μήνυμα αναφοράς σε όλους τους συντονιστές.

```
// Create and send message to all coordinators in L
ReportMessage ms = new ReportMessage(donetasks,failedproc);


for(int i=(int)(Math.pow(2, level)-1); i<(Math.pow(2,
level+1)-1) && i<aliveproc.size(); i++){
     try {
          getCommLayer().sendTCP(ms, aliveproc.get(i));
          total_msgs++;
     } catch (IOException e) {
          e.printStackTrace();
     }
}
```

Αυτοί με την σειρά τους συλλέγουν αυτά τα μηνύματα και υπολογίζουν βάση αυτών το καινούργιο σύνολο για τις ολοκληρωμένες εργασίες D και το σύνολο εσφαλμένων επεξεργαστών F.

```
if (msg instanceof ReportMessage) {

      //Act as a coordinator
      cordinator=true;

      //Get the message to a handler m
      ReportMessage m=(ReportMessage)msg;

      //Compute D
      for(int i=0; i<m.DoneWork.size(); i++){

            if(!donetasks.contains(m.DoneWork.get(i)))
                  donetasks.add(m.DoneWork.get(i));
      }

      //Compute F
      for(int i=0; i<m.Failured.size(); i++){

            if(!failedproc.contains(m.Failured.get(i)))
                  failedproc.add(m.Failured.get(i));
      }
}
```

Κάθε επεξεργαστής ο οποίος θεωρεί τον εαυτό του συντονιστή ή έλαβε μήνυμα αναφοράς στον γύρο συλλογής στέλνει μηνύματα περίληψης σε όλους του μη εσφαλμένους επεξεργαστές. Τα μηνύματα αυτά λαμβάνονται από τους επεξεργαστές στο βήμα παραλαβής. Στο βήμα υπολογισμού ενημερώνονται τα σύνολα *D* και *F* βάση της πληροφορίας των μηνυμάτων και αφαιρούνται οι εσφαλμένοι συντονιστές από την δομή L .

```
if (msg instanceof SummaryMessage){

      Cordinators.add(sender);
      SummaryMessage m =(SummaryMessage)msg;

      //Compute stage
      for(int i=0; i<m.DoneWork.size(); i++){
```

```
            if(!donetasks.contains(m.DoneWork.get(i)))
                donetasks.add(m.DoneWork.get(i));
        }

        for(int i=0; i<m.Failured.size(); i++){

            if(!failedproc.contains(m.Failured.get(i)))
                failedproc.add(m.Failured.get(i));
        }
}
```

Ακολουθεί η υλοποίηση του κανόνα *Load Balancing.* Ο κανόνας αυτός, *Bal (T – D , P – F , i)* για κάθε επεξεργαστή *p*: κατατάσσει τις εργασίες στο σύνολο *T-D* και τους επεξεργαστές στο σύνολο *P-F* βάση των id's τους. Έστω ότι ο επεξεργαστής έχει την θέση *r* στο σύνολο *P-F* τότε επιλέγει να εκτελέσει την $(r \bmod |T\text{-}D|)^{οστή}$ εργασία.

```
        //Must take the job with Balance Rule
        //First we must get the T-D (tasks - done tasks)

        for(int i=0; i<tasks.size(); i++){
            if(!donetasks.contains(tasks.get(i)))
                utasks.add(tasks.get(i));
        }

        //Get our rank from the temporary P-F set.
        for(int i=0; i<tempalive.size(); i++){
            if(tempalive.get(i).equals(getNodeId())){
                rank = i;
                break;
        }
        //Execute the rank mod utasks size task
        job = utasks.get(rank % utasks.size());
        donetasks.add(job);
```

Ο πλήρης με επαρκές σχόλια κώδικας για τον αλγόριθμο *Do-Um* επισυνάπτεται στο τέλος της εργασίας στο Παράρτημα Γ .

Κεφάλαιο 5

Πειραματική Αξιολόγηση Αλγορίθμων σε πραγματικό περιβάλλον

5.1 Μετρικές Αξιολόγησης Πολυπλοκότητας

Μετά τη φάση της υλοποίησης των τριών αλγορίθμων που επιλύουν το πρόβλημα *Do-ALL* ακολουθεί η πειραματική αξιολόγησή τους. Απώτερος στόχος της αξιολόγησης αυτής είναι να διαφανεί σε ποιο βαθμό οι αλγόριθμοι είναι αποτελεσματικοί, αποδοτικοί, ανεκτικοί σε σφάλματα και κατά πόσο η αξιολόγηση τους συνάδει με την θεωρητική ανάλυση τους, εάν αυτή υπάρχει.

Οι μετρικές που θα ληφθούν υπόψη για αυτή την αξιολόγηση είναι η πολυπλοκότητα Εργασίας και η πολυπλοκότητα Μηνυμάτων όπως αυτές ορίζονται στο υποκεφάλαιο 2.3 .

5.2 Μεθοδολογία

Κατά την πειραματική εκτέλεση των αλγορίθμων που υλοποιήσαμε στο PlanetLab, υπήρξαν κάποιοι περιορισμοί. Όπως προαναφέρθηκε, ο αριθμός των μηχανών που μπορούσαμε να αναθέσουμε σε κάθε μέρισμα ήταν περιορισμένος, έτσι, σε κάθε πειραματική εκτέλεση δεσμεύσαμε το πολύ 128 μηχανές. Κάθε πειραματική εκτέλεση έγινε για σενάριο 8, 16, 32, 64 και 128 κόμβων. Η επιλογή αυτή των τιμών αποσκοπεί σε μια καλύτερη και αναλογική σύγκριση των διαφόρων σεναρίων μεταξύ τους. Οι μετρικές που τέθηκαν υπό μελέτη και ανάλυση είναι αυτές της Πολυπλοκότητας Εργασίας και της Πολυπλοκότητας Μηνυμάτων όπως ορίζονται στο υποκεφάλαιο 2.3. Η επεξήγηση των διαφόρων σεναρίων που έλαβαν χώρο επεξηγούνται στο επόμενο υποκεφάλαιο.

5.3 Σενάρια

Στο συγκεκριμένο υποκεφάλαιο θα γίνει μια σύντομη και συγχρόνως περιεκτική περιγραφή των διαφόρων σεναρίων που εφαρμόστηκαν στον κάθε αλγόριθμο. Τα σενάρια αυτά στο σύνολο τους αποσκοπούν στο να γίνει μια ορθολογική και από πολλές σκοπιές αξιολόγηση. Αυτή η αξιολόγηση μπορεί να αφορά τόσο τον κάθε αλγορίθμου ξεχωριστά όσο και συγκρίσεις μεταξύ αλγορίθμων.

5.3.1 Σενάριο Χωρίς Σφάλματα

Όπως εξυπακούεται και από το όνομα του σεναρίου έχουμε απουσία σφαλμάτων/ καταρρεύσεων των επεξεργαστών. Κάθε επεξεργαστής θα λαμβάνει μέρος στον υπολογισμό καθ’ όλες τις επαναλήψεις μέχρι την διεκπαιρέωση όλων των εργασιών που αναθέσαμε στο τρέχων πρόβλημα.

5.3.2 Τυχαία Σφάλματα

Τέτοια σφάλματα αναμένονται σε πραγματικές εκτελέσεις. Βάση αυτών ένας επεξεργαστής καταρρέει με μία πιθανότητα *error_rate* στο βήμα αποστολής κάθε γύρου. Συγκεκριμένα για τους αλγόριθμους AN και AR, οι οποίοι προαπαιτούν δίκτυο με αξιόπιστη πολυεκπομπή, ο κάθε επεξεργαστής αποστέλλει μήνυμα με μία πιθανότητα 0.5 σε όλους η κανένα προτού καταρρεύσει. Σε αντίθεση με αυτό, για τον αλγόριθμο Do-Um ο οποίος δεν προαπαιτεί αξιοπιστία δικτύου, ο κάθε επεξεργαστής προτού καταρρεύσει αποστέλλει μήνυμα με μία πιθανότητα 0.5 στον κάθε επεξεργαστή ξεχωριστά.

5.3.3 Σφάλματα Συντονιστών

Ενώ η κατάρρευση συντονιστών είναι ίσως ένα μη ρεαλιστικό σενάριο, ωστόσο είναι βοηθητικό να κατανοήσουμε την επιρροή που έχουν αυτού του είδους οι καταρρεύσεις στον υπολογισμό. Επιλέγουμε ένα μοτίβο καταρρεύσεων τέτοιο ώστε κάθε συντονιστής θα καταρρέει πριν από το βήμα αποστολής ως ότου απομείνει ενεργό μόνο το τελευταίο στρώμα της δεντρικής δομής των αλγορίθμων.

5.3.4 Σφάλματα αποστολής συντονιστών

Η ιδέα αυτού του σεναρίου είναι η ίδια με το προηγούμενο σενάριο με την διαφορά ότι κάθε συντονιστής καταρρέει κατά το βήμα αποστολής. Όπως και με το σενάριο τυχαίων σφαλμάτων, για τον αλγόριθμο AN, ο οποίος προαπαιτεί αξιοπιστία

δικτύου, κάθε επεξεργαστής θα αποστέλλει μήνυμα με μία πιθανότητα 0.5 σε όλους η κανένα προτού καταρρεύσει. Ομοίως για τις απαιτήσεις του αλγόριθμου Do-Um, ο κάθε επεξεργαστής θα αποστέλλει μήνυμα με μία πιθανότητα 0.5 στον κάθε επεξεργαστή ξεχωριστά προτού καταρρεύσει.

5.3.5 Σφάλματα κατώτατου ορίου

Σε αυτά τα σενάρια κατώτατου ορίου ο αντίπαλος συντρίβει επεξεργαστές μόνο όταν αυτοί αναθέτονται σε εργασίες κατά τα στάδια υπολογισμού. Ο αντίπαλος προσδιορίζει το σύνολο των μη διεκπεραιωμένων εργασιών *U* σε κάθε επανάληψη και επιλέγει $\frac{1}{log\, n}|U|$ εργασίες οι οποίες έχουν τον πιο μικρό αριθμό ανατεθειμένων επεξεργαστών σε αυτές. Ακολούθως συντρίβει αυτούς τους επεξεργαστές εάν υπάρχουν. Αυτό συνεχίζει να εφαρμόζεται ως ότου ο αριθμός των εργασιών U είναι μεγαλύτερος από 1. Το συγκεκριμένο σενάριο αποσκοπεί στην σύγκριση 2 αλγορίθμων όπου ο αριθμός των επεξεργαστών θα είναι αναβαλλόμενος για τις διάφορες εκτελέσεις, εξού και η σημαντικότητα αυτού του σεναρίου συναντάτε στο κεφ.6 που γίνεται σύγκριση των αλγορίθμων AN και Do-Um. Περισσότερες πληροφορίες για την στρατηγική που εφαρμόζεται υπάρχουν στο [1].

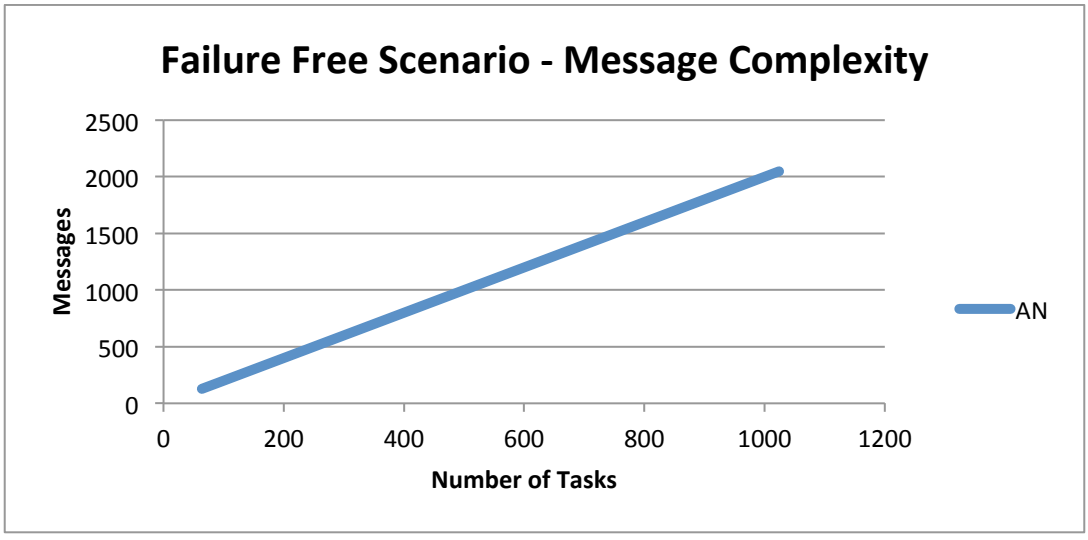
5.4 Πειραματική Αξιολόγηση Αλγορίθμου AN

Στα υποκεφάλαια που ακολουθούν περιγράφεται η πειραματική αξιολόγηση του αλγορίθμου AN βάση των διαφόρων σεναρίων που εφαρμόστηκαν για αυτόν σε πραγματικό διαδικτυακό κατανεμημένο περιβάλλον, το PlanetLab. Συγκεκριμένα για τον αλγόριθμο AN εφαρμόστηκαν τα σενάρια Απουσίας Σφαλμάτων, Τυχαίων Σφαλμάτων, Σφαλμάτων Συντονιστών και Σφαλμάτων Αποστολής Συντονιστών. Το

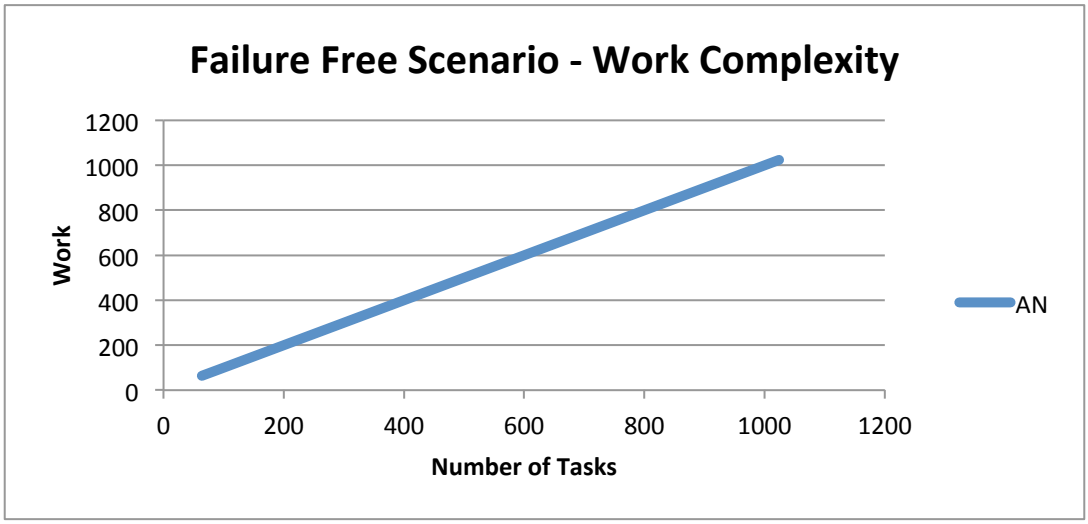
πλήθος *n* των επεξεργαστών θα παραμένει σταθερό και έχει την τιμή 64 ενώ ο αριθμός των εργασιών *t* θα κυμαίνεται από 64, 128, 256, 512 και 1024.

5.4.1 Σενάριο Χωρίς Σφάλματα

Η μορφή των γραφικών παραστάσεων αναμένουμε να είναι γραμμική. Αυτό προκύπτει γιατί σε κάθε επανάληψη κάθε επεξεργαστής εκτελεί μία εργασία, επομένως στο σύνολο εκτελούνται *n* εργασίες όσο και το πλήθος των επεξεργαστών. Για να εκτελεστούν όλες οι t εργασίες χρειάζονται $\lceil t/n \rceil$ επαναλήψεις. Άρα, εκτελούνται $n\lceil t/n \rceil$ εργασίες. Επίσης, ο συνολικός αριθμός των μηνυμάτων θα είναι $2n\lceil t/n \rceil$ εφόσον σε κάθε επανάληψη αποστέλνονται *2n* μηνύματα (*n* μηνύματα τύπου αναφοράς και *n* μηνύματα τύπου περίληψης). Συνεπώς, τόσο η πολυπλοκότητα των μηνυμάτων όσο και η πολυπλοκότητα εργασίας πρέπει να είναι της τάξεως $O(t)$. Ακολουθούν οι γραφικές παραστάσεις για την αξιολόγηση του σεναρίου χωρίς σφάλματα. Η πρώτη (Γράφημα 5.1) συσχετίζεται με την πολυπλοκότητα Μηνυμάτων και η δεύτερη (Γράφημα 5.2) με την πολυπλοκότητα Εργασίας.



Γράφημα 5.1 Πολυπλοκότητα Μηνυμάτων στο σενάριο Απουσίας Σφαλμάτων.



Γράφημα 5.2 Πολυπλοκότητα Εργασίας στο σενάριο Απουσίας Σφαλμάτων.

Και στις δύο γραφικές στον άξονα χ συναντάμε τον αριθμό των εργασιών που εκτελέστηκαν. Για την γραφική παράσταση που αφορά την πολυπλοκότητα μηνυμάτων (Γράφημα 5.1), στον άξονα ψ λαμβάνει χώρο ο συνολικός αριθμός μηνυμάτων που απεστάλησαν κατά την διάρκεια της κάθε εκτέλεσης. Στην περίπτωση της γραφικής παράστασης που αναλύει την πολυπλοκότητα εργασίας (Γράφημα 5.2), στον άξονα ψ συναντάμε τον συνολικό αριθμό βημάτων που έγιναν στο σύνολο όλων των επεξεργαστών μέχρι να ολοκληρωθεί πλήρως η κάθε εκτέλεση.

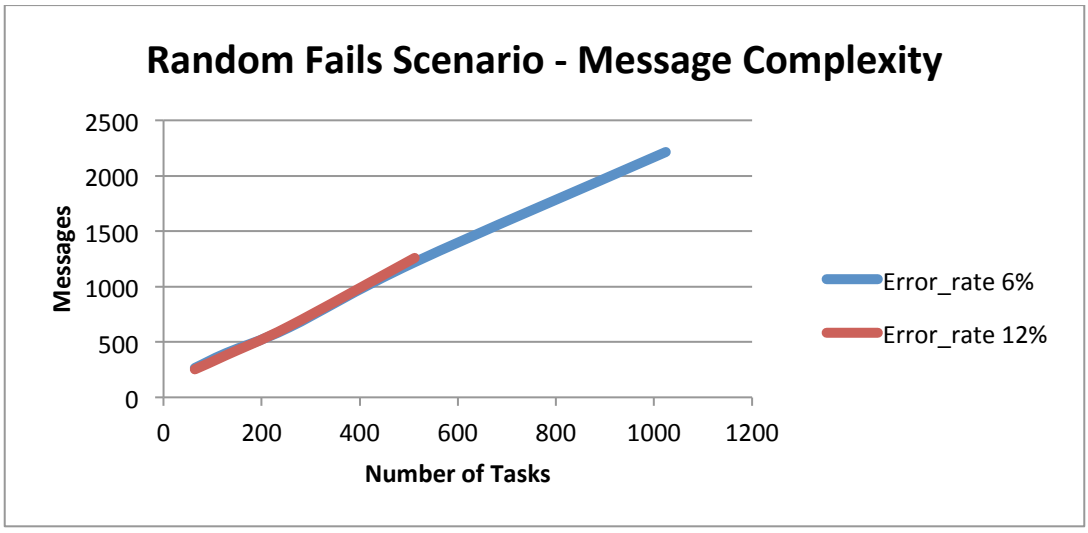
Οι πιο πάνω γραφικές παραστάσεις έχουν γραμμική μορφή. Τόσο η πολυπλοκότητα των Μηνυμάτων όσο και η πολυπλοκότητα Εργασίας είναι της τάξεως $O(t)$, επομένως, συμπίπτουν και επαληθεύουν τα αναμενόμενα αποτελέσματα που προκύπτουν από την θεωρία.

5.4.2 Σενάριο Τυχαίων Σφαλμάτων

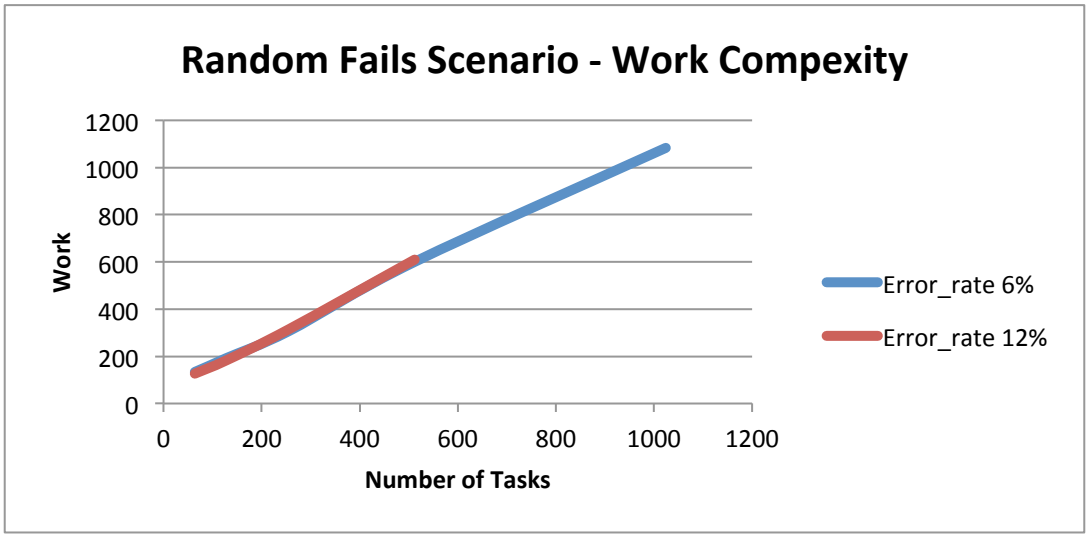
Για σκοπούς αξιολόγησης, κάθε περίπτωση μέχρι που αριθμός των εργασιών να γίνει ίσος με 512, εκτελέστηκε για τιμές *error_rate* ίσο με 6 και 12 %. Ακολούθως για αριθμό εργασιών ίσο με 1024 εκτελέστηκε μόνο το σενάριο με *error_rate* ίσό με 6 %.

Αυτό που αναμένεται είναι και οι δύο περιπτώσεις και στις δύο γραφικές παραστάσεις, πολυπλοκότητας Εργασίας – Μηνυμάτων, να έχουν όμοια συμπεριφορά και συγκεκριμένα η μορφή τους να είναι γραμμική. Αυτό οφείλεται στο γεγονός ότι σε κάθε επανάληψη ο αριθμός των σφαλμάτων θα κυμαίνεται ομοιόμορφα στο επίπεδο της τάξης της τιμής *error_rate* και ένας αναλογικός, σε σχέση με τους ενεργούς επεξεργαστές, αριθμός εργασιών και μηνυμάτων, θα ολοκληρώνεται και θα αποστέλλεται αντίστοιχα, σε κάθε επανάληψη ως ότου εκπληρωθούν όλες οι εργασίες.

Πιο κάτω παρουσιάζονται οι γραφικές παραστάσεις για την αξιολόγηση του σεναρίου με τυχαία σφάλματα που αφορά τον αλγόριθμο AN. Η πρώτη (Γράφημα 5.3) συσχετίζεται με την πολυπλοκότητα Μηνυμάτων και η δεύτερη (Γράφημα 5.4) με την πολυπλοκότητα Εργασίας.



Γράφημα 5.3 Πολυπλοκότητα Μηνυμάτων για το σενάριο Τυχαίων Σφαλμάτων.



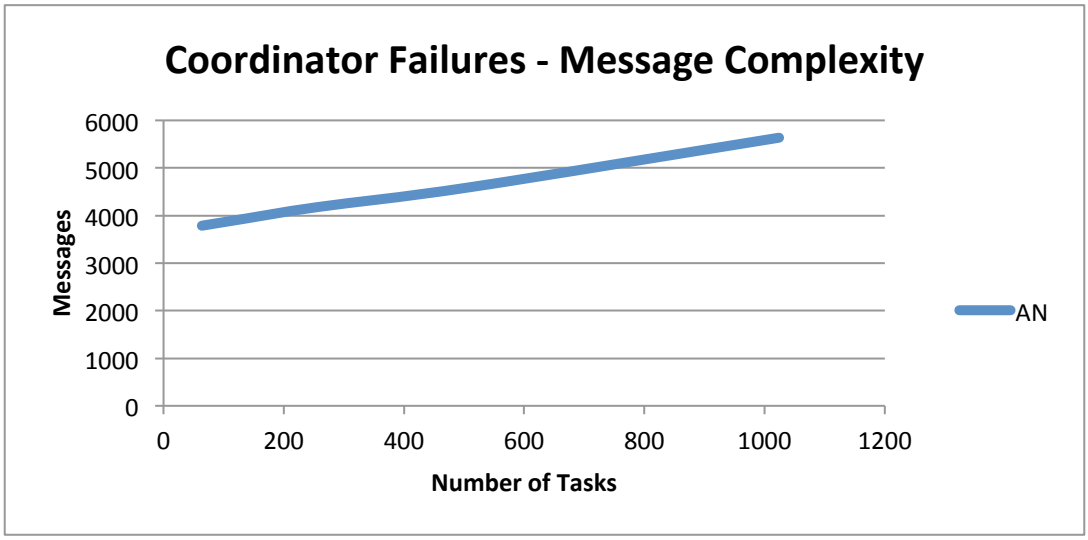
Γράφημα 5.4 Πολυπλοκότητα Εργασίας για το σενάριο Τυχαίων Σφαλμάτων.

Και στις δύο γραφικές στον άξονα χ συναντάμε τον αριθμό των εργασιών που εκτελέστηκαν. Για την γραφική παράσταση που αφορά την πολυπλοκότητα μηνυμάτων (Γράφημα 5.3), στον άξονα ψ λαμβάνει χώρο ο συνολικός αριθμός μηνυμάτων που απεστάλησαν κατά την διάρκεια της κάθε εκτέλεσης. Στην περίπτωση της γραφικής παράστασης που αναλύει την πολυπλοκότητα εργασίας (Γράφημα 5.1), στον άξονα ψ συναντάμε τον συνολικό αριθμό βημάτων που έγιναν στο σύνολο όλων των επεξεργαστών μέχρι να ολοκληρωθεί πλήρως η κάθε εκτέλεση.

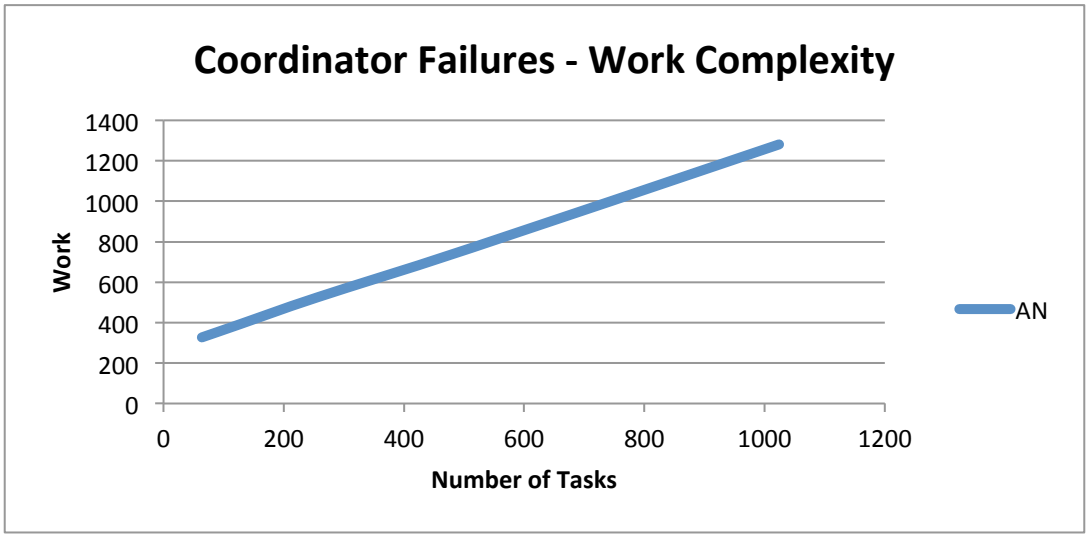
Αναλύοντας τις πιο πάνω γραφικές παραστάσεις παρατηρούμε όντως να έχουν γραμμική μορφή. Μέχρι και τον αριθμό εργασιών 512 τα δύο σενάρια με *error_rate* 6 και 12 % έχουν όμοια και σχεδόν ταυτόσημη συμπεριφορά τόσο στην πολυπλοκότητα Εργασίας όσο και στην πολυπλοκότητα Μηνυμάτων. Αυτό οφείλεται στον τρόπο που διαχειρίζεται τα σφάλματα ο αλγόριθμος AN καθώς επίσης και στο γεγονός ότι η γεννήτρια τυχαίων αριθμών για την λήψη της πιθανότητας *error_rate* είχε ομοιόμορφη κατανομή. Επομένως, οι γραφικές παραστάσεις συμπίπτουν και επαληθεύουν τα αναμενόμενα αποτελέσματα που προκύπτουν από την θεωρία.

5.4.3 Σφάλματα Συντονιστών

Βάση αυτού του σεναρίου, όπου οι συντονιστές θα καταρρέουν ως ότου μείνουν ενεργοί μόνο οι επεξεργαστές στο τελευταίο στρώμα αναμένουμε να έχουμε γραφικές παραστάσεις για την πολυπλοκότητα Μηνυμάτων και Εργασίας σε γραμμική μορφή. Έστω ο αριθμός των επεξεργαστών που λαμβάνουν μέρος στον υπολογισμό είναι *n* και ο αριθμός των εργασιών *t*. Εάν λάβουμε υπόψη την δενδρική δομή που χρησιμοποιεί ο αλγόριθμος AN τότε θα φτάσουμε στο τελευταίο στρώμα μετά από τις πρώτες *log n - 1* επαναλήψεις και θα έχουμε το πολύ $\lfloor n/2 \rfloor$ ζωντανούς επεξεργαστές να εκτελέσουν το πολύ *t* εναπομείναντες εργασίες. Αυτό ισοδυναμεί με την επίλυση του αρχικού προβλήματος χωρίς σφάλματα από $\lfloor n/2 \rfloor$ επεξεργαστές με ένα μικρό επιπλέον κόστος *c* Εργασίας και Μηνυμάτων που προκάλεσαν οι πρώτες *log n - 1* επαναλήψεις με καταρρεύσεις στους συντονιστές. Σε επόμενο βήμα παρουσιάζονται οι γραφικές παραστάσεις για την αξιολόγηση του σεναρίου με Σφάλματα Συντονιστών που εφαρμόστηκε στον αλγόριθμο AN. Η πρώτη (Γράφημα 5.5) συσχετίζεται με την πολυπλοκότητα Μηνυμάτων και η δεύτερη (Γράφημα 5.6) με την πολυπλοκότητα Εργασίας.



Γράφημα 5.5 Πολυπλοκότητα Μηνυμάτων για το σενάριο Σφαλμάτων Συντονιστών.



Γράφημα 5.6 Πολυπλοκότητα Εργασίας για το σενάριο Σφαλμάτων Συντονιστών.

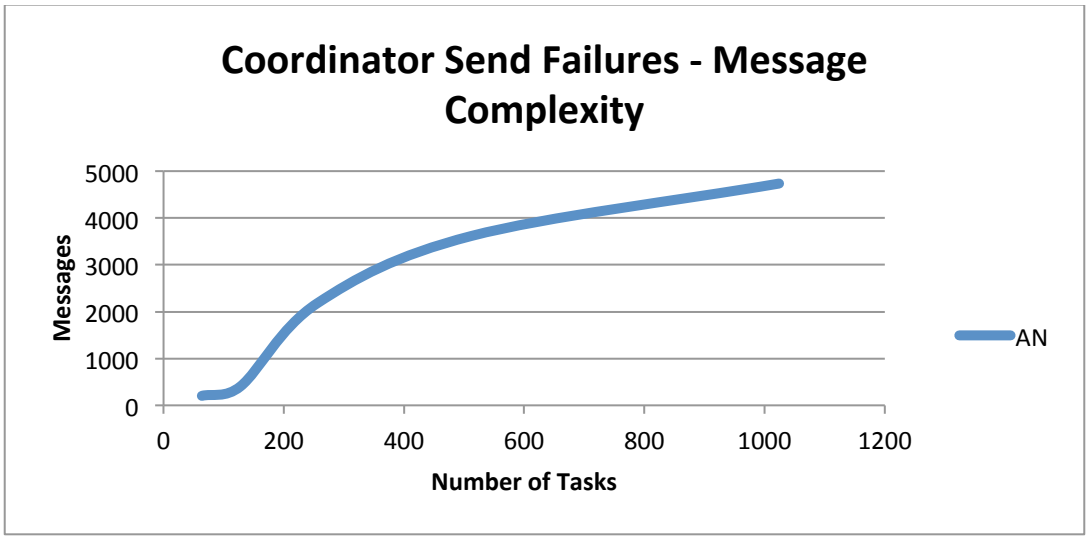
Και στις δύο γραφικές στον άξονα χ συναντάμε τον αριθμό των εργασιών που εκτελέστηκαν. Για την γραφική παράσταση που αφορά την πολυπλοκότητα μηνυμάτων (Γράφημα 5.5), στον άξονα ψ λαμβάνει χώρο ο συνολικός αριθμός μηνυμάτων που απεστάλησαν κατά την διάρκεια της κάθε εκτέλεσης. Στην περίπτωση της γραφικής παράστασης που αναλύει την πολυπλοκότητα εργασίας (Γράφημα 5.6), στον άξονα ψ συναντάμε τον συνολικό αριθμό βημάτων που έγιναν στο σύνολο όλων των επεξεργαστών μέχρι να ολοκληρωθεί πλήρως η κάθε εκτέλεση.

Αναλύοντας τις πιο πάνω γραφικές παραστάσεις παρατηρούμε να έχουν γραμμική μορφή τάξεως $O(t + c)$, επομένως, συμπίπτουν και επαληθεύουν τα αναμενόμενα αποτελέσματα που προκύπτουν από την θεωρία.

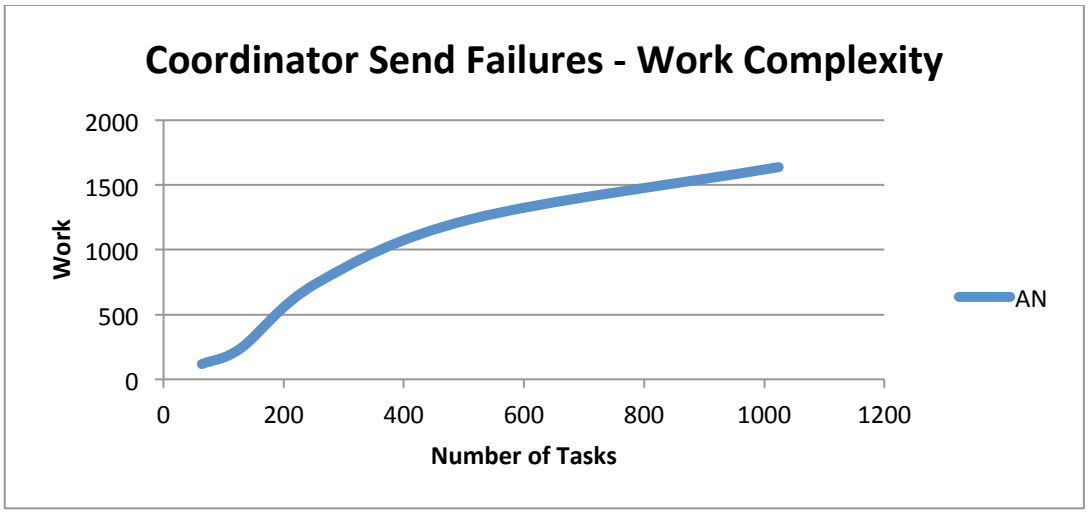
5.4.4 Σφάλματα Αποστολής Συντονιστών

Σε αυτό το σενάριο αναμένουμε και οι δύο γραφικές παραστάσεις, πολυπλοκότητας Εργασίας και πολυπλοκότητας Μηνυμάτων, να είναι λογαριθμικής μορφής. Σε αντίθεση με το προηγούμενο σενάριο, Σφάλματα Συντονιστών, εδώ προτού καταρρεύσει ένας συντονιστής αποστέλλει μηνύματα περίληψης προς όλους η κανένα με μία πιθανότητα ίση με 0,5. Επακόλουθο αυτού είναι να μεταδίδεται κάποια σημαντική γνώση για τις ολοκληρωμένες εργασίες και να αποστέλλεται ένας επίσης σημαντικός αριθμός μηνυμάτων κατά τις πρώτες $\log n - 1$ επαναλήψεις.

Συγκεκριμένα, η γνώση που μεταδίδεται και ο αριθμός μηνυμάτων που αποστέλλονται μέχρι να φτάσουμε στο τελευταίο επίπεδο της δεντρικής δομής του αλγορίθμου θα είναι λογαριθμικά συναρτήσει του αριθμού των ενεργών επεξεργαστών, δηλαδή της τάξεως $O(\log n)$. Ακολουθούν οι γραφικές παραστάσεις για την αξιολόγηση του σεναρίου με Σφάλματα Αποστολής Συντονιστών που εφαρμόστηκε στον αλγόριθμο AN. Η πρώτη (Γράφημα 5.7) συσχετίζεται με την πολυπλοκότητα Μηνυμάτων και η δεύτερη (Γράφημα 5.8) με την πολυπλοκότητα Εργασίας.



Γράφημα 5.7 Πολυπλοκότητα Μηνυμάτων στο σενάριο Σφαλμάτων Αποστολής Συντονιστών.



Γράφημα 5.8 Πολυπλοκότητα Εργασίας για το σενάριο Σφαλμάτων Αποστολής Συντονιστών.

Και στις δύο γραφικές στον άξονα χ συναντάμε τον αριθμό των εργασιών που εκτελέστηκαν. Για την γραφική παράσταση που αφορά την πολυπλοκότητα μηνυμάτων (Γράφημα 5.7), στον άξονα ψ λαμβάνει χώρο ο συνολικός αριθμός μηνυμάτων που απεστάλησαν κατά την διάρκεια της κάθε εκτέλεσης. Στην περίπτωση της γραφικής παράστασης που αναλύει την πολυπλοκότητα εργασίας (Γράφημα 5.8), στον άξονα ψ συναντάμε τον συνολικό αριθμό βημάτων που έγιναν στο σύνολο όλων των επεξεργαστών μέχρι να ολοκληρωθεί πλήρως η κάθε εκτέλεση.

Αναλύοντας τις πιο πάνω γραφικές παραστάσεις παρατηρούμε όντως να έχουν λογαριθμική μορφή και να επικυρώνουν τα όσα αναφέρθηκαν προηγουμένως. Επομένως, αυτές οι γραφικές παραστάσεις επαληθεύουν τα αναμενόμενα αποτελέσματα που προκύπτουν από την θεωρία.

5.4.5 Συμπεράσματα

Σκοπός των σεναρίων που εφαρμόσαμε στον αλγόριθμο ΑΝ ήταν να μας παρουσιαστούν τυχών αδυναμίες του αλγορίθμου, να αναλυθεί το πως αντιμετωπίζονται οι διάφορες περιπτώσεις σφαλμάτων και τι επιπτώσεις επιφέρει η κάθε μία από αυτές στην πρόοδο του υπολογισμό και στις μετρικές αξιολόγησης.

Βάση το σενάριο Απουσίας Σφαλμάτων, ο αλγόριθμος χωρίς καμία καθυστέρηση και χωρίς κανένα επιπλέον κόστος στην πολυπλοκότητα Εργασίας και Μηνυμάτων επιλύει το πρόβλημα Do-All αποδοτικά όπως και ήταν αναμενόμενο από την θεωρία. Στο σενάριο Τυχαίων Σφαλμάτων το οποίο αντικατοπτρίζει κατά κύριο άξονα πραγματικές εκτελέσεις, παρατηρήθηκε ότι για ένα ποσοστό σφαλμάτων 12% ο αλγόριθμος διαχειρίστηκε αυτά τα σφάλματα και επίλυσε το πρόβλημα Do-all δίδοντας μας μάλιστα ικανοποιητικές και ανεκτικές τιμές μετρικών όσο αφορά την πολυπλοκότητα Εργασίας και Μηνυμάτων. Στα σενάρια Κατάρρευσης Συντονιστών και Κατάρρευσης Αποστολής Συντονιστών, τα οποία ίσως και να είναι μη ρεαλιστικά, αυτό που παρατηρήθηκε κατά κύριο άξονα είναι το δεύτερο να επιτυγχάνει

καλύτερες αποδόσεις τόσο στην πολυπλοκότητα εργασίας όσο και στον συνολικό αριθμό μηνυμάτων.

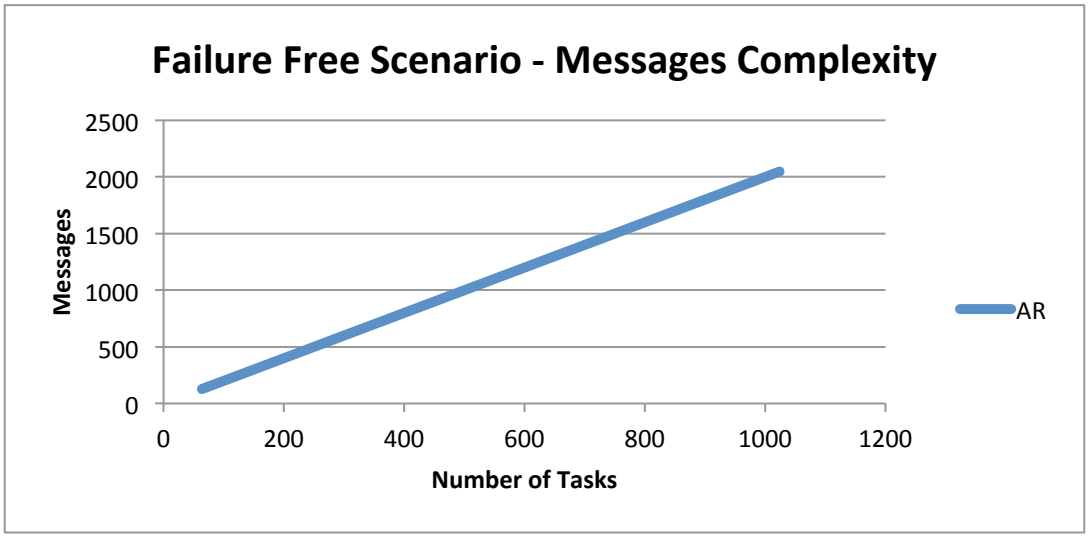
Βάση των σεναρίων αυτών, δεν παρουσιάστηκαν εμφανές αδυναμίες στον αλγόριθμο. Αντιθέτως όλα τα πιο πάνω σενάρια επέφεραν τα αναμενόμενα αποτελέσματα και είχαν πολυπλοκότητα Εργασίας και Μηνυμάτων καλύτερης τάξεως από αυτή της χειρότερης περίπτωσης.

5.5 Πειραματική Αξιολόγηση Αλγορίθμου AR

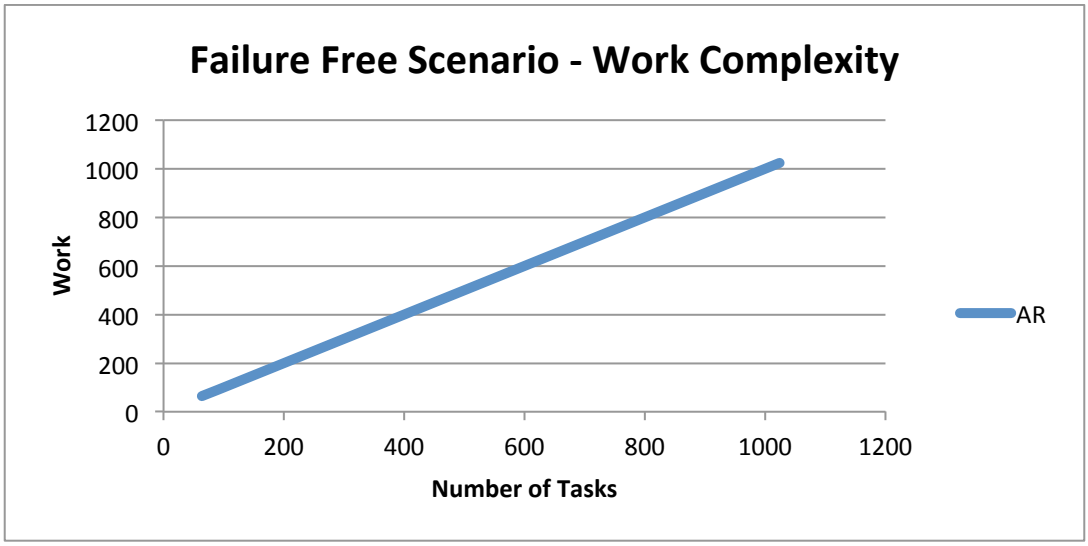
Στα υποκεφάλαια που ακολουθούν περιγράφεται η πειραματική αξιολόγηση του αλγορίθμου AR βάση των διαφόρων σεναρίων που εφαρμόστηκαν για αυτόν σε πραγματικό διαδικτυακό κατανεμημένο περιβάλλον, το PlanetLab. Συγκεκριμένα για τον αλγόριθμο AR εφαρμόστηκαν τα σενάρια Απουσίας Σφαλμάτων και Τυχαίων Σφαλμάτων. Επιπρόσθετα, για σκοπούς αξιολόγησης στα υποκεφάλαια 5.7.3 και 5.7.4 παραβιάζονται κάποιες συνθήκες, όπως θα επεξηγηθούν πλήρως, για να επικυρωθεί η συμπεριφορά του αλγορίθμου αυτού σε τέτοιες παραβιάσεις. Το πλήθος *n* των επεξεργαστών θα παραμένει σταθερό και έχει την τιμή 64 ενώ ο αριθμός των εργασιών *t* θα κυμαίνεται από 64, 128, 256, 512 και 1024.

5.5.1 Σενάριο Χωρίς Σφάλματα

Αυτό που αναμένουμε από τις πιο πάνω γραφικές παραστάσεις είναι να έχουν γραμμική μορφή και συγκεκριμένα τόσο η πολυπλοκότητα των Μηνυμάτων όσο και η πολυπλοκότητα Εργασίας να είναι $O(t)$. Σε πρώτο στάδιο παρουσιάζονται οι γραφικές παραστάσεις για την αξιολόγηση του σεναρίου χωρίς σφάλματα. Η πρώτη (Γράφημα 5.9) συσχετίζεται με την πολυπλοκότητα Μηνυμάτων και η δεύτερη (Γράφημα 5.10) με την πολυπλοκότητα Εργασίας.



Γράφημα 5.9 Πολυπλοκότητα Μηνυμάτων για το σενάριο Απουσίας Σφαλμάτων.



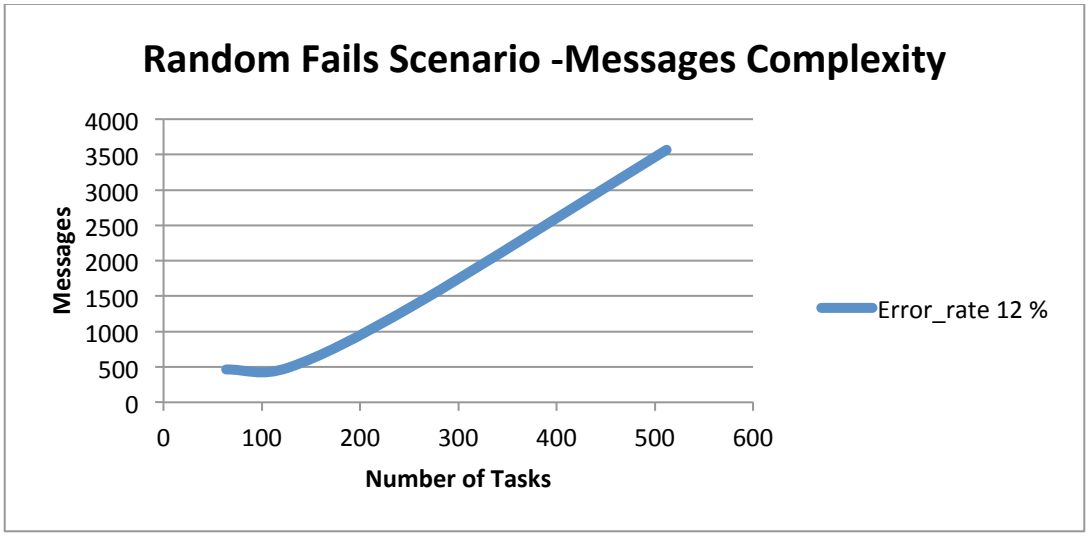
Γράφημα 5.10 Πολυπλοκότητα Εργασίας για το σενάριο Απουσίας Σφαλμάτων.

Και στις δύο γραφικές στον άξονα χ συναντάμε τον αριθμό των εργασιών που εκτελέστηκαν. Για την γραφική παράσταση που αφορά την πολυπλοκότητα μηνυμάτων (Γράφημα 5.9), στον άξονα ψ λαμβάνει χώρο ο συνολικός αριθμός μηνυμάτων που απεστάλησαν κατά την διάρκεια της κάθε εκτέλεσης. Στην περίπτωση της γραφικής παράστασης που αναλύει την πολυπλοκότητα εργασίας (Γράφημα 5.10), στον άξονα ψ συναντάμε τον συνολικό αριθμό βημάτων που έγιναν στο σύνολο όλων των επεξεργαστών μέχρι να ολοκληρωθεί πλήρως η κάθε εκτέλεση.

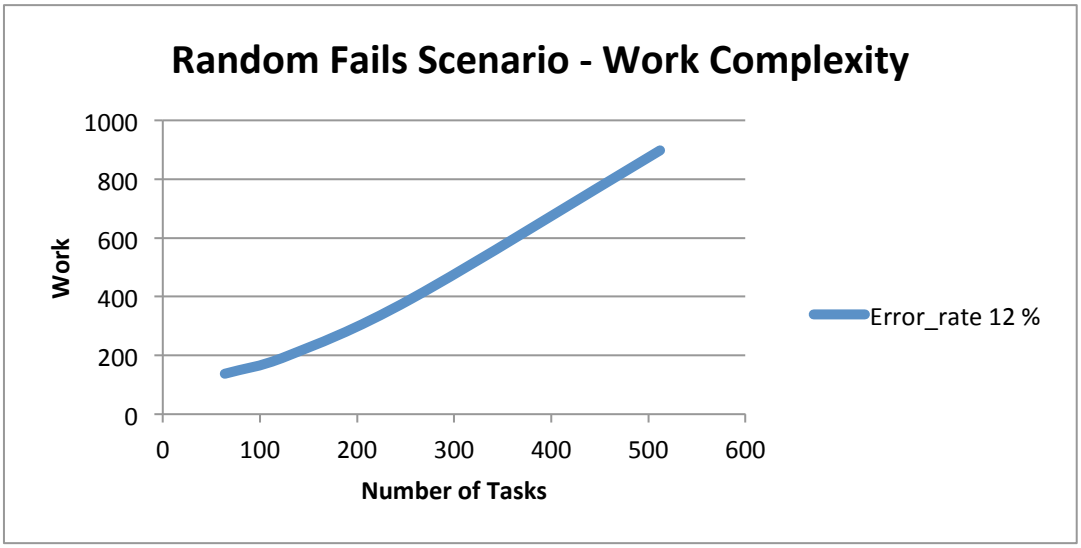
Αυτό που προκύπτει από τις πιο πάνω γραφικές παραστάσεις είναι η επικύρωση του γεγονότος ότι ο αλγόριθμος AR εν απουσία σφαλμάτων έχει πανομοιότυπη συμπεριφορά με αυτή του αλγορίθμου AN όπως αυτό τονίζεται στο [6]. Επομένως, τόσο η πολυπλοκότητα των μηνυμάτων όσο και η πολυπλοκότητα εργασίας είναι της τάξεως $O(t)$, όπως αυτό επικυρώνεται και από τις πιο πάνω γραφικές παραστάσεις.

5.5.2 Σενάριο Τυχαίων Σφαλμάτων

Το σενάριο αυτό έχει ακριβώς ως περιγράφτηκε στο υποκεφάλαιο 5.3.2 με μία επιπλέον τροποποίηση. Η τροποποίηση αυτή προνοεί όπως κατά το πρώτο γύρο κάθε επανάληψης, ένας υποτιθέμενος μη-ενεργός επεξεργαστής επανεκκίνει με μία πιθανότητα *restart_rate* ίση με αυτή του *error_rate*, συγκεκριμένα 12 %. Εάν θεωρήσουμε ότι η γεννήτρια τυχαίων αριθμών για την λήψη της πιθανότητας *error_rate* και *restart_rate* θα έχει ομοιόμορφη κατανομή, τότε αυτό που αναμένουμε είναι σε κάθε επανάληψη να έχουμε ένα αριθμό καταρρεύσεων σχεδόν ίσο με τον αριθμό των επανεκκινήσεων. Επομένως, από τις πιο πάνω γραφικές παραστάσεις αναμένουμε να είναι γραμμικής μορφής και συγκεκριμένα η πολυπλοκότητα των Μηνυμάτων και Εργασίας θα είναι της τάξεως $O(t)$. Η πρώτη γραφική που παρουσιάζεται (Γράφημα 5.11) συσχετίζεται με την πολυπλοκότητα Μηνυμάτων και η δεύτερη (Γράφημα 5.12) με την πολυπλοκότητα Εργασίας.



Γράφημα 5.11 Πολυπλοκότητα Μηνυμάτων για το σενάριο Τυχαίων Σφαλμάτων.



Γράφημα 5.12 Πολυπλοκότητα Εργασίας για το Σενάριο Τυχαίων Σφαλμάτων.

Και στις δύο γραφικές στον άξονα χ συναντάμε τον αριθμό των εργασιών που εκτελέστηκαν. Για την γραφική παράσταση που αφορά την πολυπλοκότητα μηνυμάτων (Γράφημα 5.11), στον άξονα ψ λαμβάνει χώρο ο συνολικός αριθμός μηνυμάτων που απεστάλησαν κατά την διάρκεια της κάθε εκτέλεσης. Στην περίπτωση της γραφικής παράστασης που αναλύει την πολυπλοκότητα εργασίας (Γράφημα 5.12), στον άξονα ψ συναντάμε τον συνολικό αριθμό βημάτων που έγιναν στο σύνολο όλων των επεξεργαστών μέχρι να ολοκληρωθεί πλήρως η κάθε εκτέλεση.

Αυτό που τελικά προκύπτει από τις πιο πάνω γραφικές παραστάσεις είναι η επικύρωση του γεγονότος ότι η γεννήτρια τυχαίων αριθμών για την λήψη των πιθανοτήτων *error_rate* και *restart_rate* είχε ομοιόμορφη κατανομή, δηλαδή ο αριθμός τον καταρρεύσεων σε κάθε επανάληψη ήταν σχεδόν ίσος με τον αριθμό των επανεκκινήσεων. Επίσης προκύπτει και το γεγονός ότι και οι δύο γραφικές παραστάσεις έχουν γραμμική μορφή. Επομένως, τόσο η πολυπλοκότητα των Μηνυμάτων όσο και η πολυπλοκότητα Εργασίας είναι της τάξεως $O(t)$, όπως αυτό επικυρώνεται και από τις πιο πάνω γραφικές παραστάσεις.

5.5.3 Παραβίαση συνθήκης 26-περιοριμένο

Όπως προαναφέρθηκε και σε προηγούμενο κεφάλαιο έχει αποδειχθεί ότι ο αλγόριθμος AR είναι ανεκτικός σε f καταρρεύσεις επεξεργαστών ($f < p$) και ανεκτικός σε πρότυπα καταρρεύσεων/επανεκκινήσεων 26-περιορισμένα. Θυμίζουμε ότι με τον όρο κ-περιορισμένα ορίζουμε το πρότυπο καταρρεύσεων κατά το οποίο υπάρχει τουλάχιστον ένας ζωντανός επεξεργαστής σε κ ακολουθιακά βήματα του αλγορίθμου. Σκοπός μας ήταν να παραβιάσουμε αυτή την προϋπόθεση για να μελετήσουμε την συμπεριφορά του αλγορίθμου.

Εφόσον εκτελέσαμε ένα πεπερασμένο αριθμό επαναλήψεων για την ορθή μελέτη του σεναρίου παραβίασης της συνθήκης 26-περιορισμένο, διακρίθηκαν δύο περιπτώσεις. Συγκεκριμένα στην πρώτη περίπτωση δημιουργούνταν πρότυπα καταρρεύσεων /επανεκκινήσεων 18-περιορισμένα. Βάσει αυτών, ανεξαρτήτως των επανεκκινήσεων που υπήρξαν, επειδή υπήρξε μάρτυρας για ολόκληρες τις δύο φάσεις, 18 βήματα, δεν επηρεάστηκε ο υπολογισμός και η γενική γνώση των επεξεργαστών. Ο υπολογισμός σημείωνε πρόοδο και η εκτέλεση μετά από ένα πεπερασμένο αριθμό βημάτων τερμάτισε χωρίς αυτό να έχει αντίκτυπο στον συνολικό χρόνο ολοκλήρωσης του αλγορίθμου.

Στην δεύτερη περίπτωση δημιουργήθηκαν πρότυπα καταρρεύσεων/ επανεκκινήσεων 18-περιορισμένα και 9-περιορισμένα. Βάσει αυτών, δεν μπορούσε να αποδοθεί καμία εγγύηση για την πρόοδο του υπολογισμού και την ενημέρωση της γενικής γνώσης του κάθε επεξεργαστή. Επίσης, δεν μπορούσε να υπάρξει καμία εγγύηση ή πρόβλεψη για τον αριθμό των επαναλήψεων που θα εκτελούσε ο αλγόριθμος μέχρι τον τερματισμό ή ακόμη και αν θα τερμάτιζε ο αλγόριθμος μετά από ένα πεπερασμένο αριθμό βημάτων.

5.5.4 Ακραία Περίπτωση παραβίασης 26-περιορισμένο

Όπως και στο προηγούμενο υποκεφάλαιο 5.5.3. έτσι και εδώ δημιουργήσαμε ένα ακραίο σενάριο που παραβιάζει το πρότυπο καταρρεύσεων/επανεκκινήσεων 26-περιορισμένο που προαπαιτεί ο αλγόριθμος AR. Δημιουργήσαμε λοιπόν ένα σενάριο το οποίο κάθε επεξεργαστής είναι ενεργός μόνο για μία φάση, δηλαδή για 9 βήματα. Πιο αναλυτικά, έστω *n* ο αριθμός των επεξεργαστών, τότε σε κάθε επανάληψη λάμβαναν μέρος στον υπολογισμό *n/2* επεξεργαστές και οι υπόλοιποι *n/2* θεωρούνταν εσφαλμένοι. Με το τέλος της επανάληψης οι πρώτοι *n/2* επεξεργαστές θεωρούνταν τώρα εσφαλμένοι και οι υπόλοιποι *n/2* επεξεργαστές ξεκινούσαν να λαμβάνουν μέρος στον υπολογισμό. Αυτές οι εναλλαγές πραγματοποιήθηκαν για ένα πεπερασμένο αριθμό βημάτων και ακολούθως ο αλγόριθμος τερματίστηκε από δική μας επιλογή.

Αποτέλεσμα αυτής της εκτέλεσης ήταν σε κάθε επανάληψη παρά του αριθμού των εργασιών που διεκπεραιώθηκαν, η γνώση αυτή να μην μεταδίδεται. Επομένως, η εγγύηση που μπορεί να αποδοθεί για αυτό το ακραίο σενάριο είναι το γεγονός ότι καμία γνώση δεν θα μεταδοθεί, καμία πρόοδος στον υπολογισμό δεν θα καταγραφεί και ο αλγόριθμος σε καμία μελλοντική φάση δεν θα τερματίσει.

5.5.5 Συμπεράσματα

Σκοπός των σεναρίων που εφαρμόσαμε στον αλγόριθμο AR ήταν να μας παρουσιαστούν τυχών αδυναμίες του αλγορίθμου, να αναλυθεί το πως αντιμετωπίζονται οι διάφορες περιπτώσεις σφαλμάτων και τι επιπτώσεις επιφέρει η κάθε μία από αυτές στην πρόοδο του υπολογισμό και στις μετρικές αξιολόγησης.

Βάση το σενάριο Απουσίας Σφαλμάτων, ο αλγόριθμος χωρίς καμία καθυστέρηση και χωρίς κανένα επιπλέον κόστος στην πολυπλοκότητα Εργασίας και Μηνυμάτων επιλύει το πρόβλημα Do-All αποδοτικά και έχει πανομοιότυπη συμπεριφορά με τον

αλγόριθμο AN, όπως αυτό ήταν αναμενόμενο από την θεωρία. Στο σενάριο Τυχαίων Σφαλμάτων το οποίο αντικατοπτρίζει κατά κύριο άξονα πραγματικές εκτελέσεις, παρατηρήθηκε ότι για ένα ποσοστό σφαλμάτων 12% ο αλγόριθμος διαχειρίστηκε αυτά τα σφάλματα και επίλυσε το πρόβλημα Do-all δίδοντας μας μάλιστα ικανοποιητικές και ανεκτικές τιμές μετρικών όσο αφορά την πολυπλοκότητα Εργασίας και Μηνυμάτων. Ακολούθησε η παραβίαση συνθήκης 26-περιορισμένο. Σε αυτή διακρίναμε τον αλγόριθμο να αδυνατεί να δώσει εγγυήσεις για την πρόοδο του υπολογισμού και την επίλυση του προβλήματος, χωρίς όμως κατ' ανάγκη αυτό να υπονοεί τον μη τερματισμό του αλγορίθμου. Σε τελικό στάδιο εφαρμόστηκε το ακραίο σενάριο που παραβιάζει την συνθήκη 26-περιορισμένο. Σε αυτό ήταν εμφανές η αδυναμία του αλγόριθμου να επιλύσει το πρόβλημα ούτε ακόμη μετά από ένα μη-πεπερασμένο αριθμό επαναλήψεων.

Βάση των δύο πρώτων σεναρίων, δεν παρουσιάστηκαν εμφανές αδυναμίες στον αλγόριθμο. Αντιθέτως, με την παραβίαση της συνθήκης 26-περιορισμένο παρατηρήσαμε τον αλγόριθμο να αδυνατεί να δώσει εγγυήσεις για την πρόοδο του υπολογισμού και την επίλυση του προβλήματος. Συγκεκριμένα η πολυπλοκότητα Εργασίας και Μηνυμάτων για αυτές τις παραβιάσεις, σε περίπτωση τερματισμού, δεν μπορεί να θεωρείτε αποδοτική και ανεκτική εφόσον είναι απρόβλεπτη.

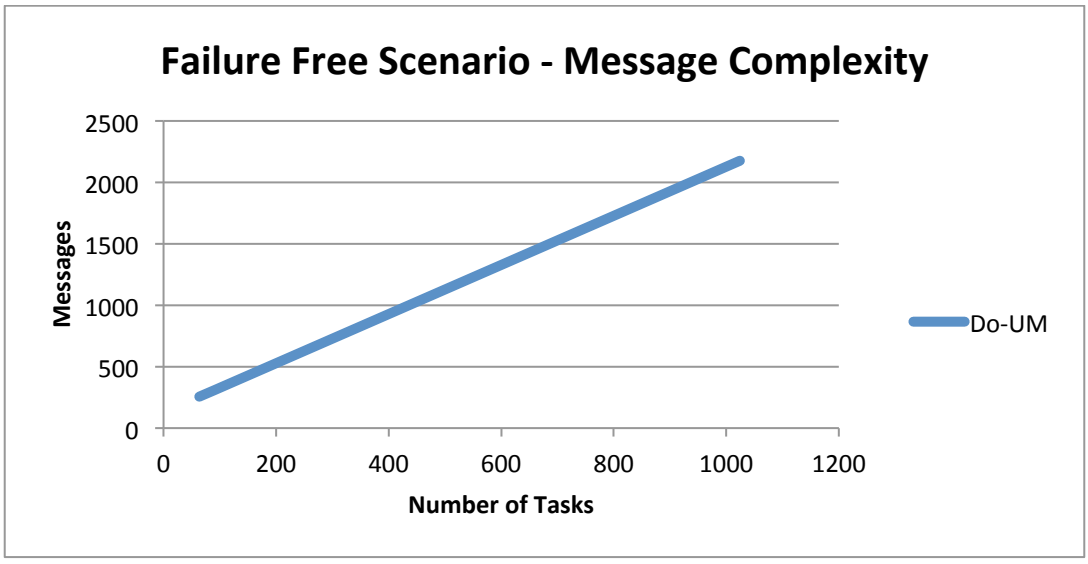
5.6 Πειραματική Αξιολόγηση Αλγορίθμου Do-Um

Στα πιο κάτω υποκεφάλαια περιγράφεται η πειραματική αξιολόγηση του αλγορίθμου Do-Um βάση των διαφόρων σεναρίων που εφαρμόστηκαν για αυτόν σε πραγματικό διαδικτυακό κατανεμημένο περιβάλλον, το PlanetLab. Ομοίως με τον αλγόριθμο AN, έτσι και για τον αλγόριθμο Do-Um εφαρμόστηκαν τα σενάρια Απουσίας Σφαλμάτων, Τυχαίων Σφαλμάτων, Σφαλμάτων Συντονιστών, Σφαλμάτων Αποστολής Συντονιστών και Σφαλμάτων Κατώτατου Ορίου. Το πλήθος n των επεξεργαστών θα παραμένει σταθερό και έχει την τιμή 64 ενώ ο αριθμός των εργασιών t θα κυμαίνεται από 64, 128, 256, 512 και 1024.

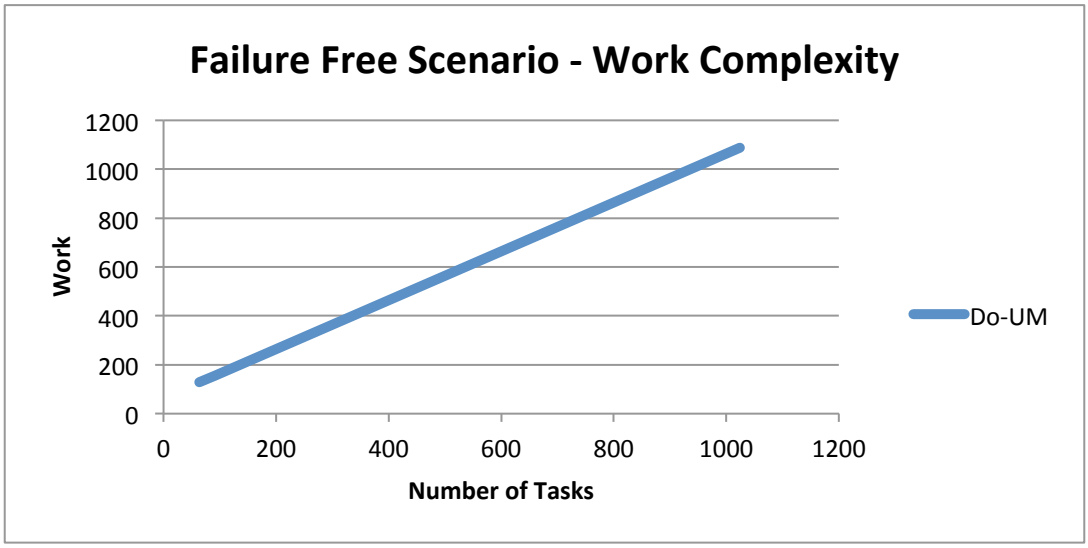
5.6.1 Σενάριο Χωρίς Σφάλματα

Η μορφή των πιο πάνω γραφικών παραστάσεων αναμένουμε να είναι γραμμική. Αυτό προκύπτει γιατί σε κάθε επανάληψη κάθε επεξεργαστής εκτελεί μία εργασία, επομένως στο σύνολο εκτελούνται *n* εργασίες όσο και το πλήθος των επεξεργαστών. Για να εκτελεστούν όλες οι *t* εργασίες χρειάζονται $\lceil t/n \rceil$ επαναλήψεις. Άρα, εκτελούνται $n\lceil t/n \rceil$ εργασίες. Επίσης, ο συνολικός αριθμός των μηνυμάτων θα είναι $2n\ \lceil t/n \rceil$ εφόσον σε κάθε επανάληψη αποστέλνονται *2n* μηνύματα (*n* μηνύματα τύπου αναφοράς και *n* μηνύματα τύπου περίληψης). Συνεπώς, τόσο η πολυπλοκότητα των μηνυμάτων όσο και η πολυπλοκότητα εργασίας θα είναι της τάξεως $O(t)$.

Πιο κάτω παρουσιάζονται οι γραφικές παραστάσεις για την αξιολόγηση του σεναρίου χωρίς σφάλματα. Η πρώτη (Γράφημα 5.13) συσχετίζεται με την πολυπλοκότητα Μηνυμάτων και η δεύτερη (Γράφημα 5.14) με την πολυπλοκότητα Εργασίας.



Γράφημα 5.13 Πολυπλοκότητα Μηνυμάτων για το σενάριο Απουσίας Σφαλμάτων.



Γράφημα 5.14 Πολυπλοκότητα Εργασίας για το σενάριο Απουσίας Σφαλμάτων.

Και στις δύο γραφικές στον άξονα χ συναντάμε τον αριθμό των εργασιών που εκτελέστηκαν. Για την γραφική παράσταση που αφορά την πολυπλοκότητα μηνυμάτων (Γράφημα 5.13), στον άξονα ψ λαμβάνει χώρο ο συνολικός αριθμός μηνυμάτων που απεστάλησαν κατά την διάρκεια της κάθε εκτέλεσης. Στην περίπτωση της γραφικής παράστασης που αναλύει την πολυπλοκότητα εργασίας (Γράφημα 5.14), στον άξονα ψ συναντάμε τον συνολικό αριθμό βημάτων που έγιναν στο σύνολο όλων των επεξεργαστών μέχρι να ολοκληρωθεί πλήρως η κάθε εκτέλεση.

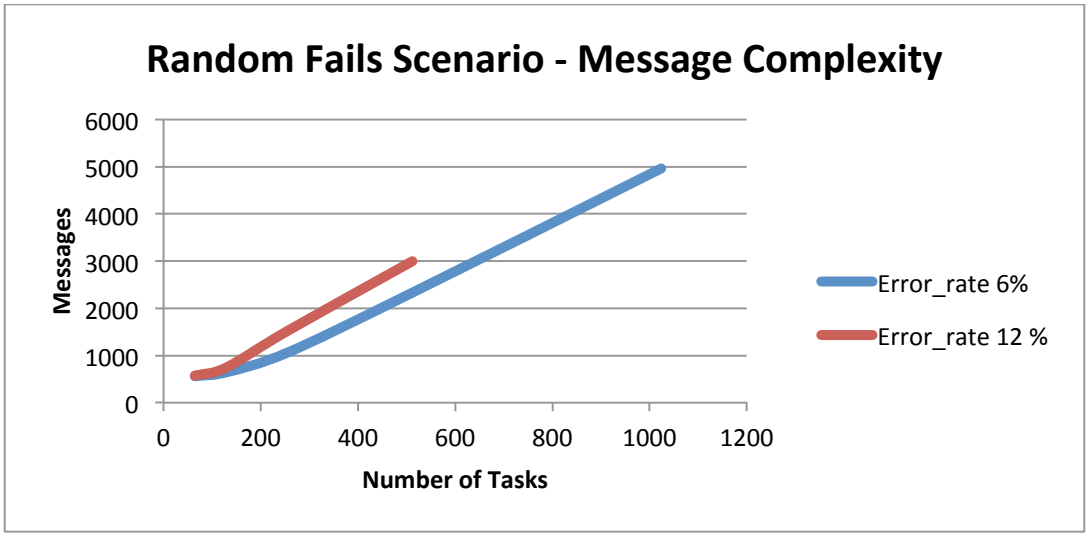
Οι πιο πάνω γραφικές παραστάσεις έχουν γραμμική μορφή. Τόσο η πολυπλοκότητα των Μηνυμάτων όσο και η πολυπλοκότητα Εργασίας είναι της τάξεως $O(t)$, επομένως, συμπίπτουν και επαληθεύουν τα αναμενόμενα αποτελέσματα που προκύπτουν από την θεωρία.

5.6.2 Σενάριο Τυχαίων Σφαλμάτων

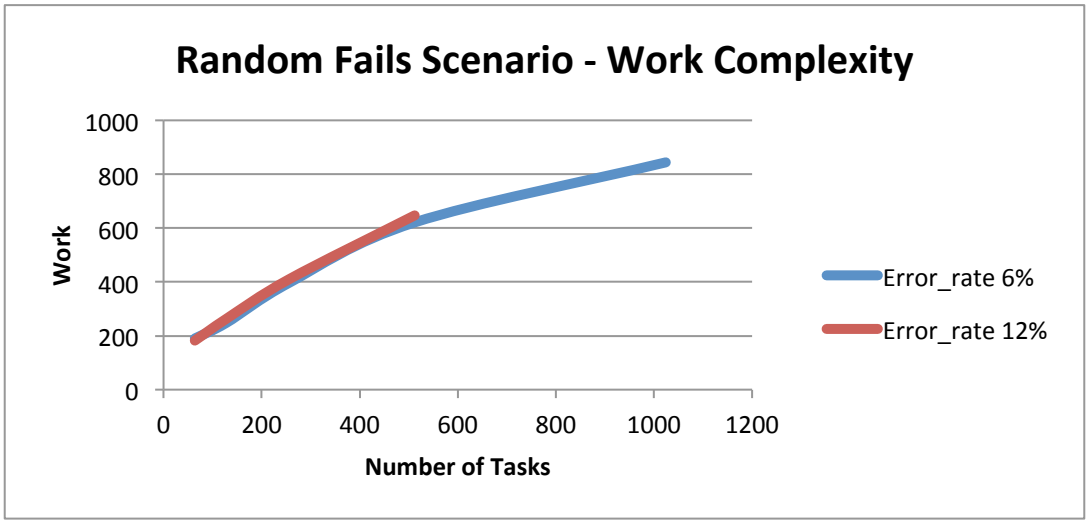
Για σκοπούς αξιολόγησης, κάθε περίπτωση μέχρι που αριθμός των εργασιών να γίνει ίσος με 512, εκτελέστηκε για τιμές *error_rate* ίσο με 6 και 12 %. Ακολούθως για αριθμό εργασιών ίσο με 1024 εκτελέστηκε μόνο το σενάριο για τιμή *error_rate* ίση με

6 %. Αυτό που αναμένεται είναι και οι δύο περιπτώσεις και στις δύο γραφικές παραστάσεις, πολυπλοκότητας Εργασίας – Μηνυμάτων, να έχουν όμοια συμπεριφορά και συγκεκριμένα η μορφή τους να είναι κατά πλείστον γραμμική. Το γεγονός ότι αναμένουμε γραμμική μορφή στις γραφικές παραστάσεις οφείλεται στο ότι σε κάθε επανάληψη ο αριθμός των σφαλμάτων θα κυμαίνεται ομοιόμορφα στο επίπεδο της τάξης της τιμής *error_rate.* Η μορφή είναι κατά πλείστον γραμμική λόγο της μη προϋπόθεσης δικτύου με αξιόπιστη πολυεκπομπή, έτσι, ο αριθμός εργασιών και μηνυμάτων, που θα ολοκληρώνεται και θα αποστέλλεται αντίστοιχα σε κάθε επανάληψη ως ότου εκπληρωθούν όλες οι εργασίες, δεν θα είναι κατ' ανάγκη αναλογικός.

Ακολουθούν οι γραφικές παραστάσεις για την αξιολόγηση του σεναρίου με τυχαία σφάλματα για τον αλγόριθμο Do-Um. Η πρώτη γραφική παράσταση (Γράφημα 5.15) συσχετίζεται με την πολυπλοκότητα Μηνυμάτων και η δεύτερη (Γράφημα 5.16) με την πολυπλοκότητα Εργασίας.



Γράφημα 5.15 Πολυπλοκότητα Μηνυμάτων για το σενάριο Τυχαίων Σφαλμάτων.



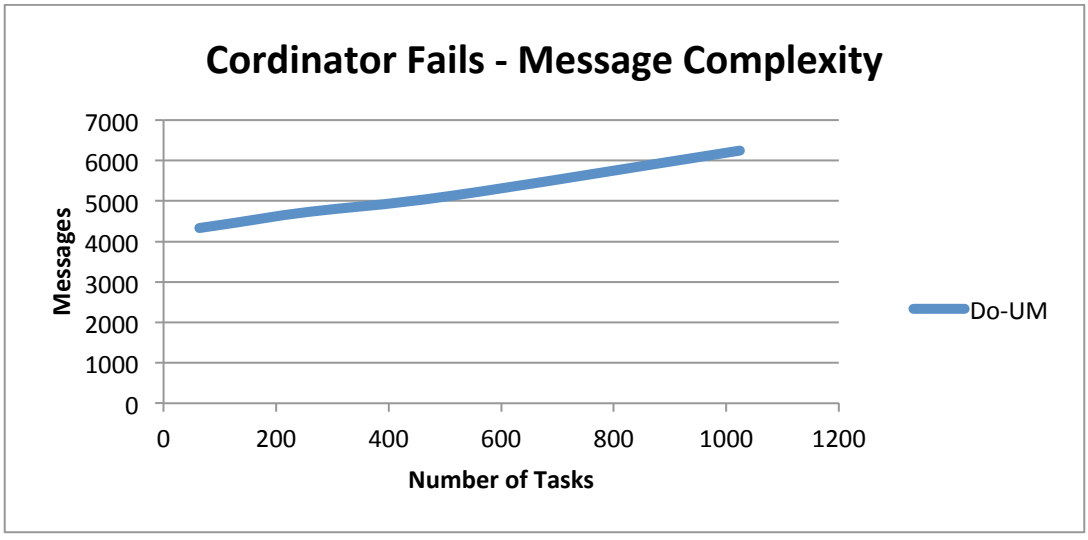
Γράφημα 5.16 Πολυπλοκότητα Εργασίας για το σενάριο Τυχαίων Σφαλμάτων.

Και στις δύο γραφικές στον άξονα χ συναντάμε τον αριθμό των εργασιών που εκτελέστηκαν. Για την γραφική παράσταση που αφορά την πολυπλοκότητα μηνυμάτων (Γράφημα 5.15), στον άξονα ψ λαμβάνει χώρο ο συνολικός αριθμός μηνυμάτων που απεστάλησαν κατά την διάρκεια της κάθε εκτέλεσης. Στην περίπτωση της γραφικής παράστασης που αναλύει την πολυπλοκότητα εργασίας (Γράφημα 5.16), στον άξονα ψ συναντάμε τον συνολικό αριθμό βημάτων που έγιναν στο σύνολο όλων των επεξεργαστών μέχρι να ολοκληρωθεί πλήρως η κάθε εκτέλεση.

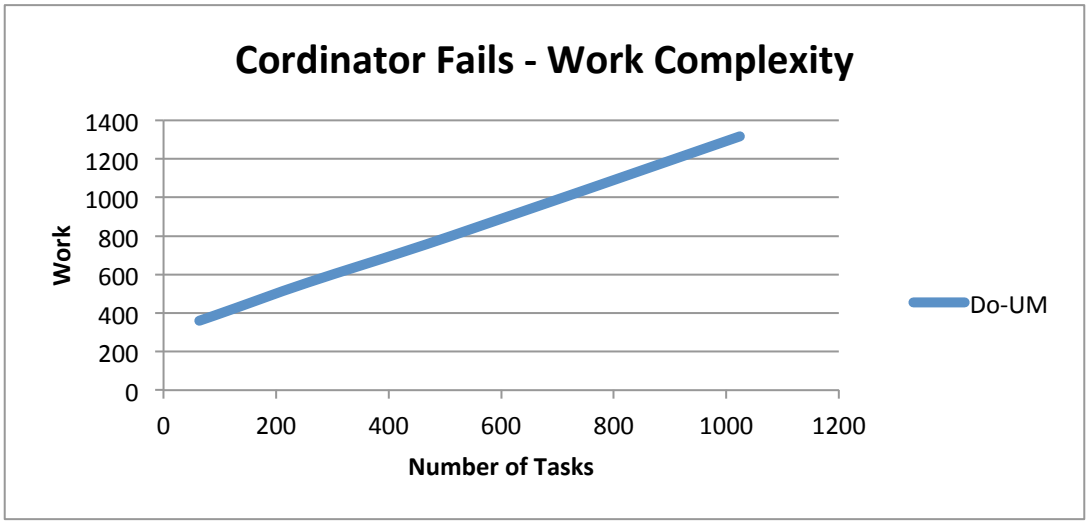
Αναλύοντας τις πιο πάνω γραφικές παραστάσεις παρατηρούμε επί το πλείστο να έχουν γραμμική μορφή. Μέχρι και τον αριθμό εργασιών 512 τα δύο σενάρια με *error_rate* 6 και 12 % έχουν όμοια συμπεριφορά τόσο στην πολυπλοκότητα Εργασίας όσο και στην πολυπλοκότητα Μηνυμάτων. Αυτό οφείλεται στον τρόπο που διαχειρίζεται τα σφάλματα ο αλγόριθμος Do-Um καθώς επίσης και στο γεγονός ότι η γεννήτρια τυχαίων αριθμών για την λήψη της πιθανότητας *error_rate* είχε ομοιόμορφη κατανομή. Όπως αναφέρθηκε και πιο πάνω, οι γραφικές παραστάσεις είναι επί το πλείστο γραμμικές λόγο της μη προϋπόθεσης δικτύου με αξιόπιστη πολυεκπομπή. Επομένως, οι γραφικές παραστάσεις συμπίπτουν και επαληθεύουν τα αναμενόμενα αποτελέσματα.

5.6.3 Σφάλματα Συντονιστών

Βάση αυτού του σεναρίου, όπου οι συντονιστές θα καταρρέουν ως ότου μείνουν ενεργοί μόνο οι επεξεργαστές στο τελευταίο στρώμα αναμένουμε να έχουμε γραφικές παραστάσεις για την πολυπλοκότητα Μηνυμάτων και Εργασίας σε γραμμική μορφή. Έστω ο αριθμός των επεξεργαστών που λαμβάνουν μέρος στον υπολογισμό είναι *n* και ο αριθμός των εργασιών *t*. Εάν λάβουμε υπόψη την δενδρική δομή που χρησιμοποιεί ο αλγόριθμος Do-Um τότε θα φτάσουμε στο τελευταίο στρώμα μετά από τις πρώτες *log n - 1* επαναλήψεις και θα έχουμε το πολύ $\lfloor n/2 \rfloor$ ζωντανούς επεξεργαστές να εκτελέσουν το πολύ *t* εναπομείναντες εργασίες. Αυτό ισοδυναμεί με την επίλυση του αρχικού προβλήματος χωρίς σφάλματα από $\lfloor n/2 \rfloor$ επεξεργαστές με ένα μικρό επιπλέον κόστος *c* Εργασίας και Μηνυμάτων που προκάλεσαν οι πρώτες *log n - 1* επαναλήψεις με καταρρεύσεις στους συντονιστές. Σε αυτό το σημείο θα παρουσιάσουμε τις γραφικές παραστάσεις για την αξιολόγηση του σεναρίου με Σφάλματα Συντονιστών που εφαρμόστηκε στον αλγόριθμο Do-Um. Η πρώτη (Γράφημα 5.17) συσχετίζεται με την πολυπλοκότητα Μηνυμάτων και η δεύτερη (Γράφημα 5.18) με την πολυπλοκότητα Εργασίας.



Γράφημα 5.17 Πολυπλοκότητα Μηνυμάτων για το σενάριο Σφαλμάτων Συντονιστών.



Γράφημα 5.18 Πολυπλοκότητα Εργασίας για το σενάριο Σφαλμάτων Συντονιστών.

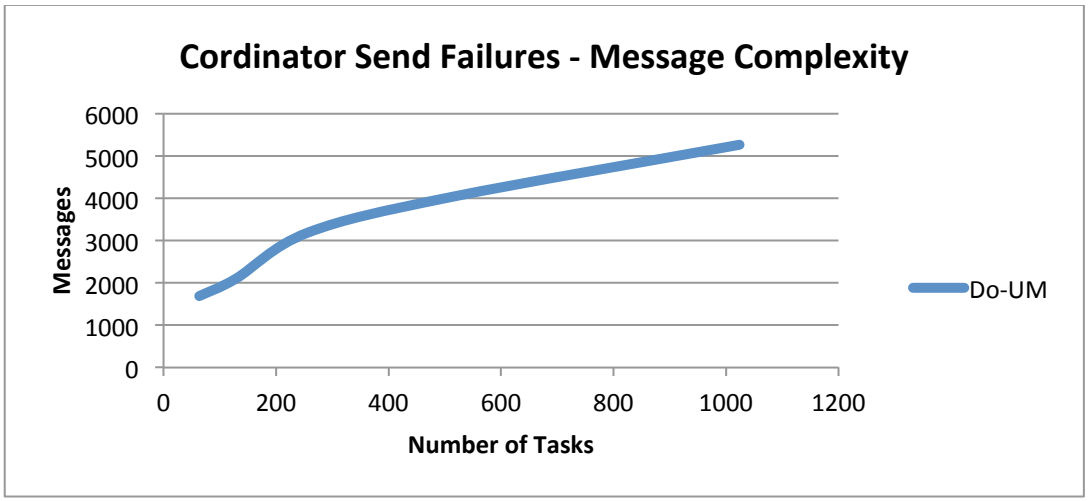
Και στις δύο γραφικές στον άξονα χ συναντάμε τον αριθμό των εργασιών που εκτελέστηκαν. Για την γραφική παράσταση που αφορά την πολυπλοκότητα μηνυμάτων (Γράφημα 5.17), στον άξονα ψ λαμβάνει χώρο ο συνολικός αριθμός μηνυμάτων που απεστάλησαν κατά την διάρκεια της κάθε εκτέλεσης. Στην περίπτωση της γραφικής παράστασης που αναλύει την πολυπλοκότητα εργασίας (Γράφημα 5.18), στον άξονα ψ συναντάμε τον συνολικό αριθμό βημάτων που έγιναν στο σύνολο όλων των επεξεργαστών μέχρι να ολοκληρωθεί πλήρως η κάθε εκτέλεση.

Αναλύοντας τις πιο πάνω γραφικές παραστάσεις παρατηρούμε όντως να έχουν γραμμική μορφή της τάξεως $O(t + c)$, επομένως, αυτές οι γραφικές επαληθεύουν τα αναμενόμενα αποτελέσματα που προκύπτουν από την ανάλυση μας.

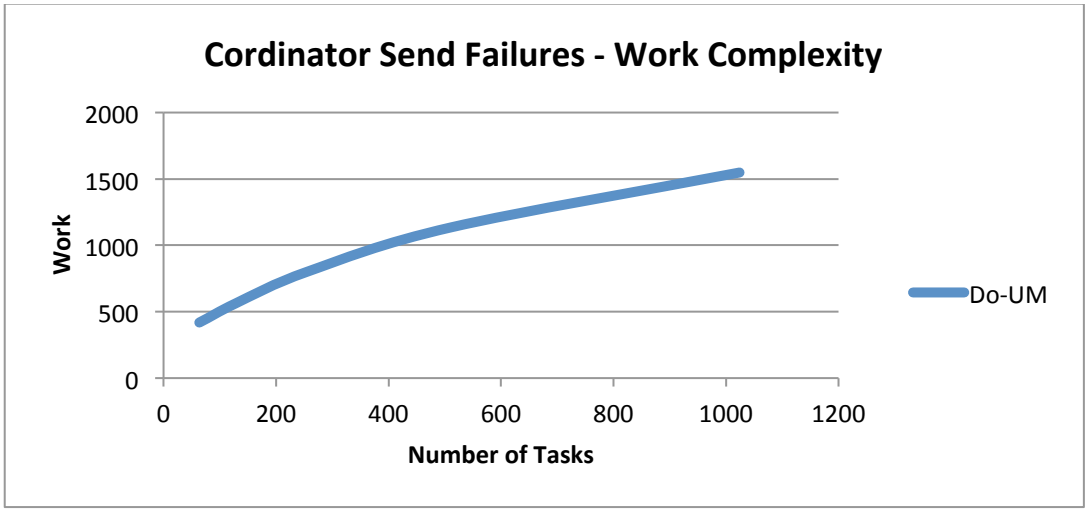
5.6.4 Σφάλματα Αποστολής Συντονιστών

Σε αυτό το σενάριο αναμένουμε και οι δύο γραφικές παραστάσεις, πολυπλοκότητας Εργασίας και πολυπλοκότητας Μηνυμάτων, να είναι λογαριθμικής μορφής. Σε αντίθεση με το προηγούμενο σενάριο, Σφάλματα Συντονιστών, εδώ προτού καταρρεύσει ένας συντονιστής αποστέλλει μηνύματα περίληψης προς όλους η κανένα με μία πιθανότητα ίση με 0,5. Επακόλουθο αυτού είναι να μεταδίδεται

κάποια σημαντική γνώση για τις ολοκληρωμένες εργασίες και να αποστέλλεται ένας επίσης σημαντικός αριθμός μηνυμάτων κατά τις πρώτες $\log n - 1$ επαναλήψεις. Συγκεκριμένα, η γνώση που μεταδίδεται και ο αριθμός μηνυμάτων που αποστέλλονται μέχρι να φτάσουμε στο τελευταίο επίπεδο της δεντρικής δομής του αλγορίθμου θα είναι λογαριθμικά συναρτήση του αριθμού των ενεργών επεξεργαστών, δηλαδή της τάξεως $O(\log n)$. Σε επόμενο βήμα παρουσιάζονται οι γραφικές παραστάσεις για την αξιολόγηση του σεναρίου με Σφάλματα Αποστολής Συντονιστών που εφαρμόστηκε στον αλγόριθμο Do-Um. Η πρώτη (Γράφημα 5.19) συσχετίζεται με την πολυπλοκότητα Μηνυμάτων και η δεύτερη (Γράφημα 5.20) με την πολυπλοκότητα Εργασίας.



Γράφημα 5.19 Πολυπλοκότητα Μηνυμάτων για το σενάριο Σφαλμάτων Αποστολής Συντονιστών.



Γράφημα 5.20 Πολυπλοκότητα Εργασίας για το σενάριο Σφαλμάτων Αποστολής Συντονιστών.

Και στις δύο γραφικές στον άξονα χ συναντάμε τον αριθμό των εργασιών που εκτελέστηκαν. Για την γραφική παράσταση που αφορά την πολυπλοκότητα μηνυμάτων (Γράφημα 5.19), στον άξονα ψ λαμβάνει χώρο ο συνολικός αριθμός μηνυμάτων που απεστάλησαν κατά την διάρκεια της κάθε εκτέλεσης. Στην περίπτωση της γραφικής παράστασης που αναλύει την πολυπλοκότητα εργασίας (Γράφημα 5.20), στον άξονα ψ συναντάμε τον συνολικό αριθμό βημάτων που έγιναν στο σύνολο όλων των επεξεργαστών μέχρι να ολοκληρωθεί πλήρως η κάθε εκτέλεση.

Αναλύοντας τις πιο πάνω γραφικές παραστάσεις παρατηρούμε όντως να έχουν λογαριθμική μορφή και να επικυρώνουν τα όσα αναφέρθηκαν προηγουμένως. Επομένως, αυτές οι γραφικές παραστάσεις επαληθεύουν τα αναμενόμενα αποτελέσματα που προέκυψαν από την εμπειρική ανάλυση μας.

5.6.5 Συμπεράσματα

Σκοπός των σεναρίων που εφαρμόσαμε στον αλγόριθμο Do-Um ήταν να μας παρουσιαστούν τυχών αδυναμίες του αλγορίθμου, να αναλυθεί το πως αντιμετωπίζονται οι διάφορες περιπτώσεις σφαλμάτων και τι επιπτώσεις επιφέρει η κάθε μία από αυτές στην πρόοδο του υπολογισμό και στις μετρικές αξιολόγησης.

Αρχικά εκτελέσαμε το σενάριο Απουσίας Σφαλμάτων, ο αλγόριθμος με μόνη καθυστέρηση την πρώτη επανάληψη, κατά την οποία καμία εργασία δεν εκτελείτε, με ένα επιπλέον κόστος της τάξεως $O(n)$ Μηνυμάτων και Εργασίας επιλύει το πρόβλημα Do-All αποδοτικά όπως και ήταν αναμενόμενο από την θεωρία. Στο σενάριο Τυχαίων Σφαλμάτων το οποίο αντικατοπτρίζει κατά κύριο άξονα πραγματικές εκτελέσεις, παρατηρήθηκε ότι για ένα ποσοστό σφαλμάτων 12% ο αλγόριθμος διαχειρίστηκε αυτά τα σφάλματα και επίλυσε το πρόβλημα Do-all . Οι τιμές μετρικών όσο αφορά την πολυπλοκότητα Εργασίας και Μηνυμάτων ήταν ικανοποιητικές και ανεκτικές εάν συμμεριστούμε την απουσία δικτύου με αξιόπιστη πολυεκπομπή. Στα σενάρια Κατάρρευσης Συντονιστών και Κατάρρευσης Αποστολής Συντονιστών, τα οποία ίσως και να είναι μη ρεαλιστικά, αυτό που παρατηρήθηκε κατά κύριο άξονα είναι το

δεύτερο να επιτυγχάνει καλύτερες αποδόσεις τόσο στην πολυπλοκότητα εργασίας όσο και στον συνολικό αριθμό μηνυμάτων.

Βάση των σεναρίων αυτών, η μόνη εμφανές αδυναμία στον αλγόριθμο είναι η πολυπλοκότητα Μηνυμάτων που δημιουργείται λόγο της απουσίας αξιόπιστης πολλαπλής διανομής *(reliable multicast)*. Όλα τα πιο πάνω σενάρια στο σύνολο τους επέφεραν τα αναμενόμενα αποτελέσματα βάση εμπειρικής ανάλυσης και είχαν πολυπλοκότητα Εργασίας και Μηνυμάτων καλύτερης τάξεως από αυτή της χειρότερης περίπτωσης.

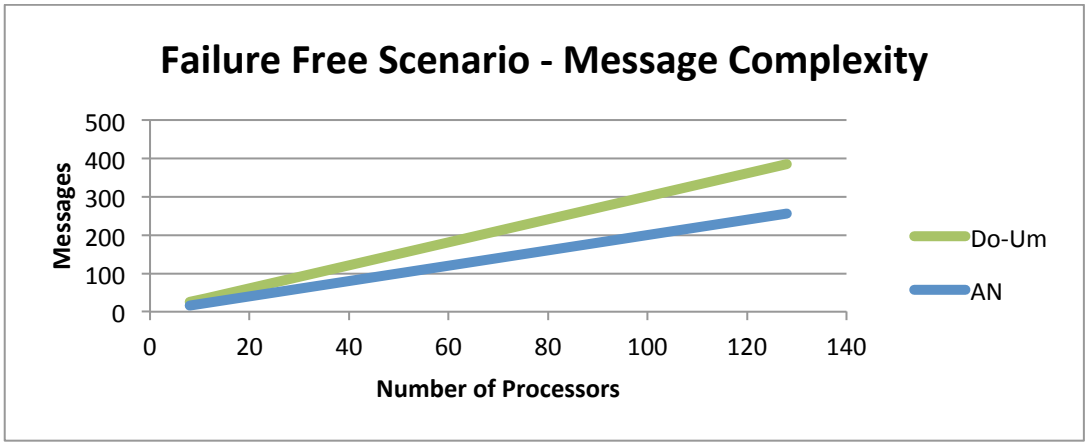
Κεφάλαιο 6

Σύγκριση Αλγορίθμων AN και Do-Um

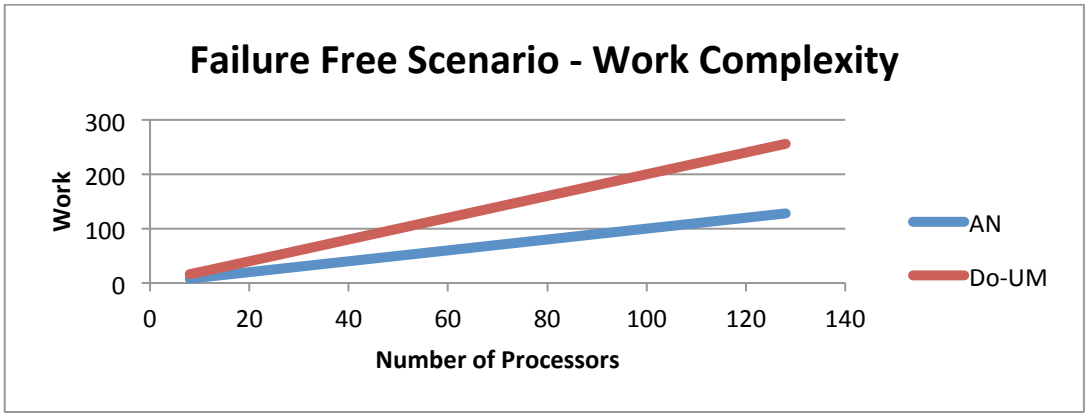
Αφού μελετήθηκαν και αξιολογήθηκαν ξεχωριστά οι δύο αλγόριθμοι AN και Do-UM στα διάφορα σενάρια που αναφέρονται στο υποκεφάλαιο 5.3 ως προς τις δύο παραμέτρους, πολυπλοκότητας Μηνυμάτων και πολυπλοκότητας Εργασίας, ακολουθεί η μεταξύ τους σύγκριση. Ο κύριος άξονας της σύγκρισης αυτής θα είναι ως προς τις ίδιες παραμέτρους έτσι ώστε να διαπιστωθούν τυχών πλεονεκτήματα και μειονεκτήματα του καθενός στα διάφορα σενάρια που εφαρμόστηκαν. Σε αντίθεση με τον τρόπο αξιολόγησης του κάθε αλγορίθμου ξεχωριστά (κεφ. 5), εδώ, τόσο το πλήθος *n* των επεξεργαστών όσο και ο αριθμός των εργασιών *t* θα είναι μεταβαλλόμενα. Συγκεκριμένα το πλήθος των επεξεργαστών *n* θα έχει τιμές 8, 16, 32, 64 και 128 και ο αριθμός των εργασιών *t* θα κυμαίνεται από 64, 128, 256, 512 και 1024.

6.1 Σύγκριση ως Προς τον Σενάριο Απουσίας Σφαλμάτων

Σε αυτό το σενάριο αναμένουμε και οι δύο γραφικές παραστάσεις, πολυπλοκότητας Εργασίας και πολυπλοκότητας Μηνυμάτων, να είναι γραμμικές. Συγκεκριμένα, για το σενάριο χωρίς σφάλματα αναμένουμε και τους δύο αλγόριθμους να έχουν όμοια συμπεριφορά. Λόγο του γεγονότος ότι στον αλγόριθμο Do-Um στην πρώτη επανάληψη δεν εκτελείτε καμία εργασία, αναμένουμε τον αλγόριθμο AN να έχει πιο καλύτερα αποτελέσματα και στις δύο γραφικές παραστάσεις. Πιο κάτω παρουσιάζονται οι γραφικές παραστάσεις για την αξιολόγηση του σεναρίου χωρίς σφάλματα. Η πρώτη (Γράφημα 6.1) συσχετίζεται με την πολυπλοκότητα Μηνυμάτων και η δεύτερη (Γράφημα 6.2) με την πολυπλοκότητα Εργασίας.



Γράφημα 6.1 Πολυπλοκότητα Μηνυμάτων στο σενάριο Απουσίας Σφαλμάτων



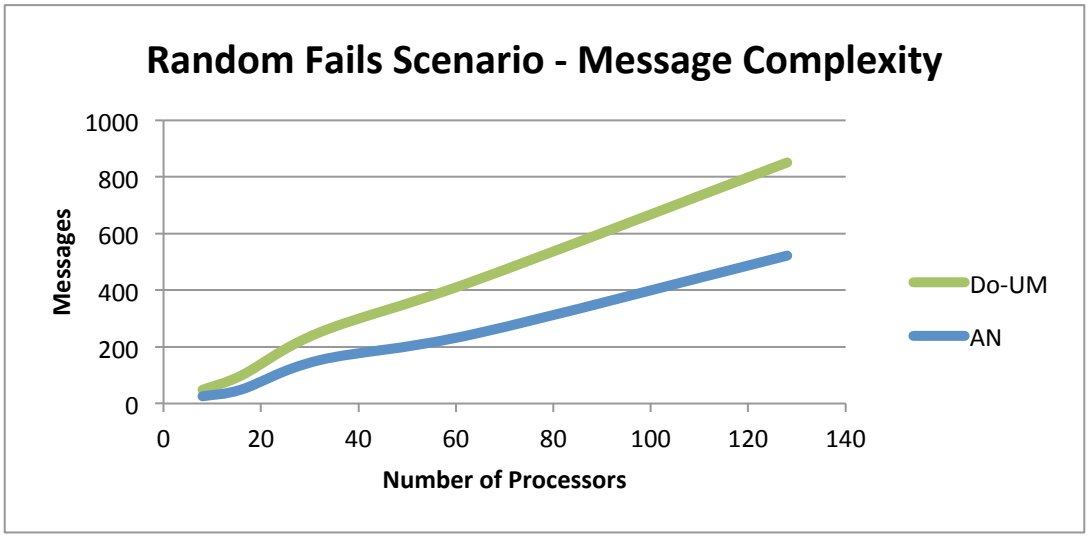
Γράφημα 6.2 Πολυπλοκότητα Εργασίας για το Σενάριο Απουσίας Σφαλμάτων.

Και στις δύο γραφικές στον άξονα χ συναντάμε τον αριθμό *n* των επεξεργαστών που λαμβάνουν μέρος στον υπολογισμό. Για την γραφική παράσταση που αφορά την πολυπλοκότητα μηνυμάτων (Γράφημα 6.1), στον άξονα ψ λαμβάνει χώρο ο συνολικός αριθμός μηνυμάτων που απεστάλησαν κατά την διάρκεια της κάθε εκτέλεσης. Στην περίπτωση της γραφικής παράστασης που αναλύει την πολυπλοκότητα εργασίας (Γράφημα 6.2), στον άξονα ψ συναντάμε τον συνολικό αριθμό βημάτων που έγιναν στο σύνολο όλων των επεξεργαστών μέχρι να ολοκληρωθεί πλήρως η κάθε εκτέλεση.

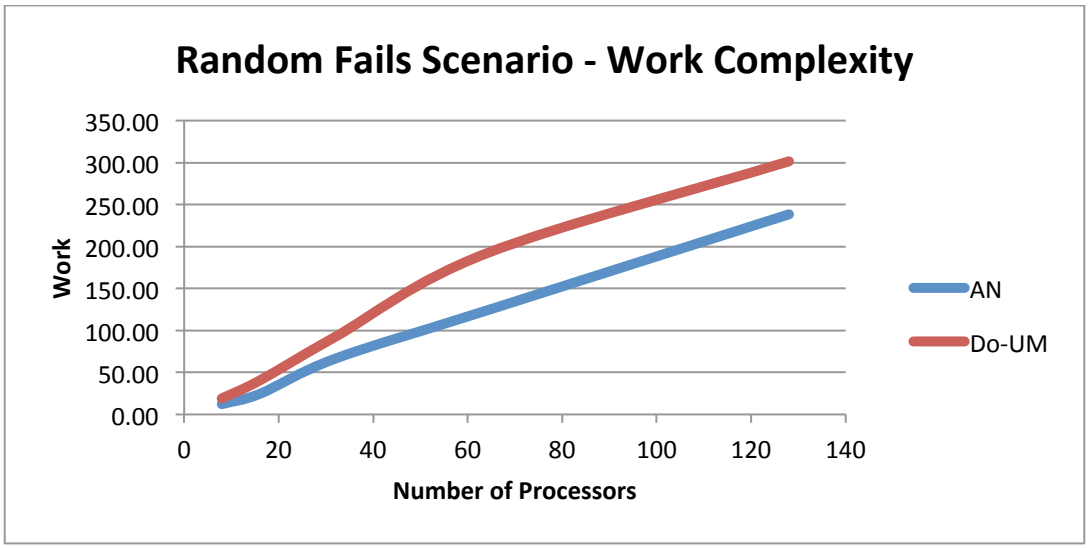
Αναλύοντας τις πιο πάνω γραφικές παραστάσεις παρατηρούμε όντως να έχουν γραμμική μορφή και να επικυρώνουν τα όσα αναφέρθηκαν προηγουμένως. Επομένως, αυτές οι γραφικές παραστάσεις επαληθεύουν τα αναμενόμενα αποτελέσματα που προκύπτουν από την θεωρία και για αυτό το σενάριο απουσίας σφαλμάτων καθιστούν πιο αποδοτικό τον αλγόριθμο AN έναντι του Do-Um.

6.2 Σύγκριση ως Προς τον Σενάριο Τυχαίων Σφαλμάτων

Σε αυτό το σενάριο όλες οι εκτελέσεις που έλαβαν χώρο είχαν error_rate ίσο με 12% . Αναμένουμε λοιπόν και τις δύο γραφικές παραστάσεις, πολυπλοκότητας Εργασίας και πολυπλοκότητας Μηνυμάτων, να είναι γραμμικές. Συγκεκριμένα, για την πολυπλοκότητα Εργασίας και οι δύο αλγόριθμοι πρακτικά θα πρέπει να κυμαίνονται στα ίδια επίπεδα. Όσο αφορά την πολυπλοκότητα Μηνυμάτων τότε οι δύο αλγόριθμοι περιμένουμε να έχουν όμοια συμπεριφορά, με τον Do-Um όμως να αποστέλλει περίπου τα διπλάσια μηνύματα από τον AN. Αυτό προκύπτει από το γεγονός ότι ο αλγόριθμος AN έχει την προϋπόθεση ύπαρξης δικτύου με αξιόπιστη πολυεκπομπή ενώ ο Do-Um όχι, αναγκάζοντας έτσι τον αλγόριθμο Do-Um να διπλασιάζει τον αριθμό των συντονιστών πιο γρήγορα έναντι του AN. Πιο κάτω παρουσιάζονται οι γραφικές παραστάσεις για την αξιολόγηση του σεναρίου με τυχαία σφάλματα. Η πρώτη (Γράφημα 6.3) συσχετίζεται με την πολυπλοκότητα Μηνυμάτων και η δεύτερη (Γράφημα 6.4) με την πολυπλοκότητα Εργασίας.



Γράφημα 6.3 Πολυπλοκότητα Μηνυμάτων για το σενάριο Τυχαίων Σφαλμάτων.



Γράφημα 6.4 Πολυπλοκότητα Εργασίας για το Σενάριο Τυχαίων Σφαλμάτων.

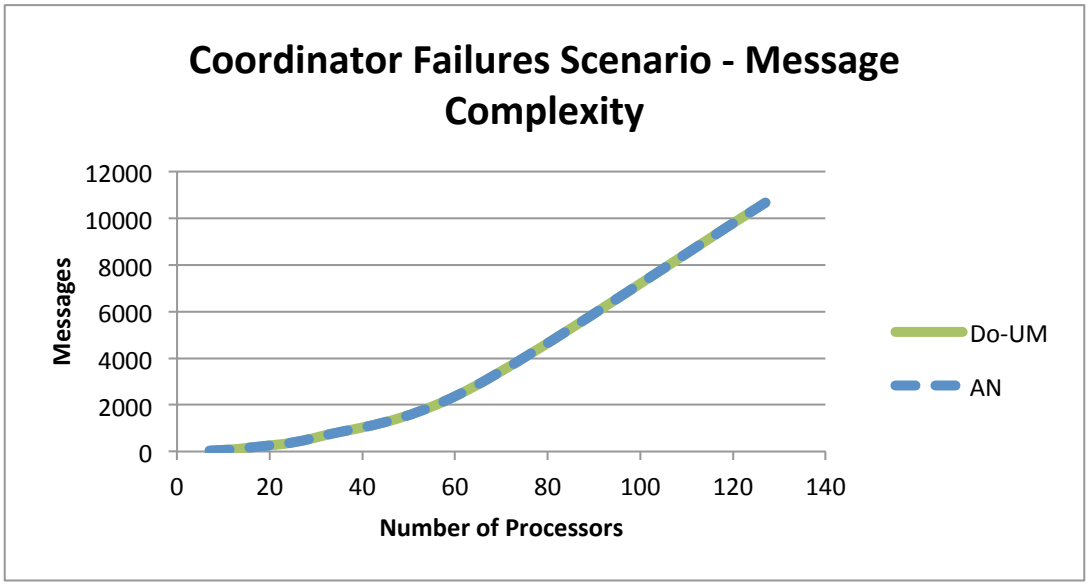
Και στις δύο γραφικές στον άξονα χ συναντάμε τον αριθμό *n* των επεξεργαστών που λαμβάνουν μέρος στον υπολογισμό. Για την γραφική παράσταση που αφορά την πολυπλοκότητα μηνυμάτων (Γράφημα 6.3), στον άξονα ψ λαμβάνει χώρο ο συνολικός αριθμός μηνυμάτων που απεστάλησαν κατά την διάρκεια της κάθε εκτέλεσης. Στην περίπτωση της γραφικής παράστασης που αναλύει την πολυπλοκότητα εργασίας (Γράφημα 6.4), στον άξονα ψ συναντάμε τον συνολικό αριθμό βημάτων που έγιναν στο σύνολο όλων των επεξεργαστών μέχρι να ολοκληρωθεί πλήρως η κάθε εκτέλεση.

Παρατηρώντας τις πιο πάνω γραφικές παραστάσεις βλέπουμε όντως να έχουν την αναμενόμενη γραμμική μορφή και να επικυρώνουν τα όσα αναφέρθηκαν προηγουμένως. Επομένως, αυτές οι γραφικές παραστάσεις επαληθεύουν τα αναμενόμενα αποτελέσματα που προκύπτουν από την θεωρία και για αυτό το σενάριο, με τυχαία σφάλματα, καθιστούν πιο αποδοτικό τον αλγόριθμο AN έναντι του Do-Um.

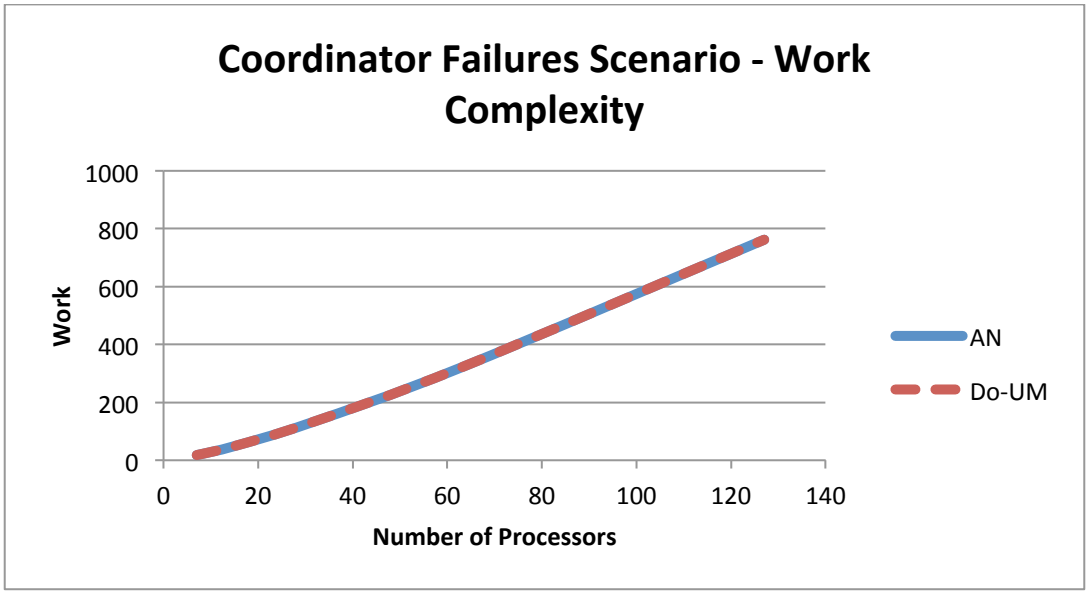
6.3 Σύγκριση ως Προς τον Σενάριο Σφαλμάτων Συντονιστών

Σε αυτό το σενάριο αναμένουμε την πολυπλοκότητα εργασίας να είναι χειρότερη από αυτή του σεναρίου με τυχαία σφάλματα. Αυτό θα οφείλεται στο γεγονός ότι τέτοιου είδους σενάρια, που δημιουργούν μοτίβα καταρρεύσεων στους συντονιστές, είναι ιδιαίτερα καταστροφικά σε αλγορίθμους που η ιδέα τους βασίζεται σε χρήση συντονιστών. Όσο αφορά την πολυπλοκότητα μηνυμάτων, και αυτή με την σειρά της αναμένουμε να είναι χειρότερη από αυτή του σεναρίου με τυχαία σφάλματα. Το μοτίβο καταρρεύσεων των επεξεργαστών αναγκάζει τον αλγόριθμο να φτάσει στο τελευταίο επίπεδο, στο οποίο, όλοι οι ενεργοί επεξεργαστές θα συμπεριφερθούν σαν συντονιστές, αποστέλλοντας έτσι ένα τετραγωνικό αριθμό μηνυμάτων. Συνεπώς θα πρέπει να αναμένουμε και για τους δύο αλγόριθμους παρόμοια συμπεριφορά και η γραφική παράσταση της πολυπλοκότητας μηνυμάτων να είναι παραβολή.

Σε πρώτο στάδιο παρουσιάζεται η γραφική παράσταση που αφορά την πολυπλοκότητα Μηνυμάτων (Γράφημα 6.5) και έπειτα η γραφική για την πολυπλοκότητα Εργασίας (Γράφημα 6.6).



Γράφημα 6.5 Πολυπλοκότητα Μηνυμάτων για το σενάριο Σφαλμάτων Συντονιστών.



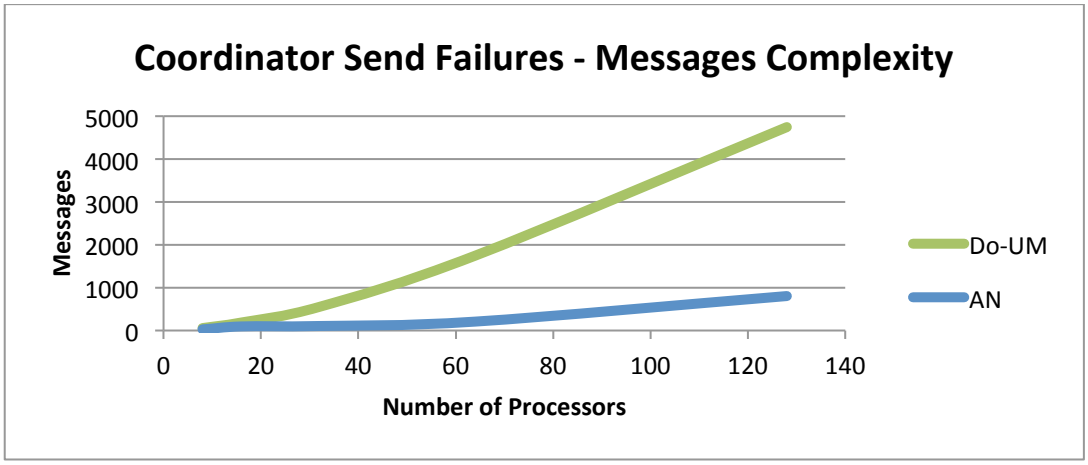
Γράφημα 6.6 Πολυπλοκότητα Εργασίας για το σενάριο Σφαλμάτων Συντονιστών.

Και στις δύο γραφικές στον άξονα χ συναντάμε τον αριθμό *n* των επεξεργαστών που λαμβάνουν μέρος στον υπολογισμό. Για την γραφική παράσταση που αφορά την πολυπλοκότητα μηνυμάτων (Γράφημα 6.5), στον άξονα ψ λαμβάνει χώρο ο συνολικός αριθμός μηνυμάτων που απεστάλησαν κατά την διάρκεια της κάθε εκτέλεσης. Στην περίπτωση της γραφικής παράστασης που αναλύει την πολυπλοκότητα εργασίας (Γράφημα 6.6), στον άξονα ψ συναντάμε τον συνολικό αριθμό βημάτων που έγιναν στο σύνολο όλων των επεξεργαστών μέχρι να ολοκληρωθεί πλήρως η κάθε εκτέλεση.

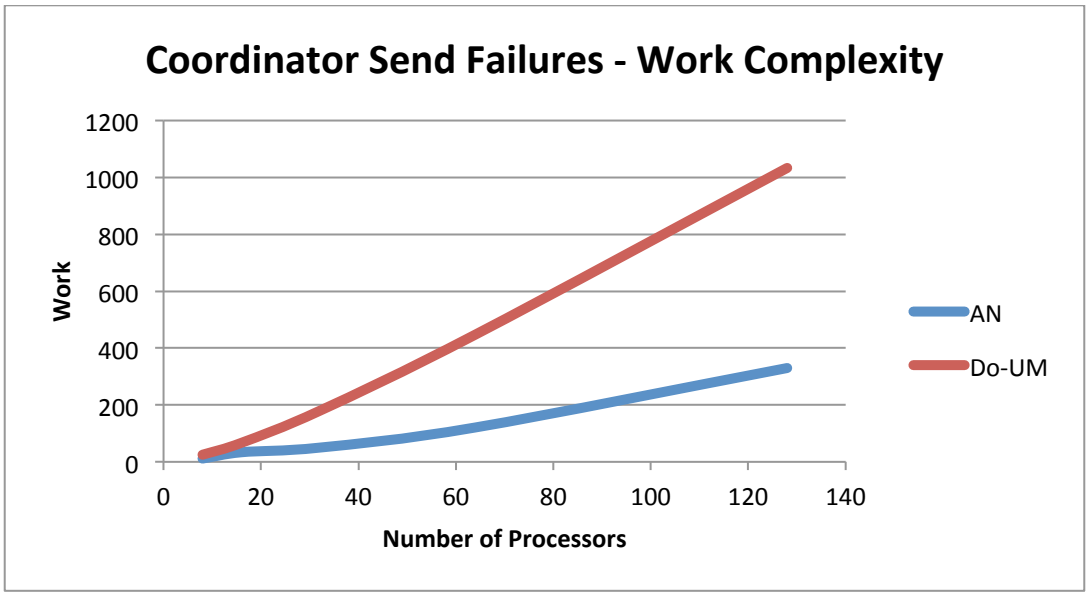
Παρατηρώντας τις πιο πάνω γραφικές παραστάσεις βλέπουμε όντως να έχουν την αναμενόμενη με αυτή που προαναφέραμε μορφή και να επικυρώνουν τα όσα αναφέρθηκαν προηγουμένως. Επομένως, σε αυτό το σενάριο η συμπεριφορά των δύο αλγορίθμων AN και Do-Um είναι πανομοιότυπη και επαληθεύονται τα αναμενόμενα αποτελέσματα που προκύπτουν από την πιο πάνω θεωρητική ανάλυση.

6.4 Σύγκριση ως Προς τον Σενάριο Σφαλμάτων Αποστολής Συντονιστών

Σε αυτό το σενάριο αναμένουμε την πολυπλοκότητα μηνυμάτων να είναι σίγουρα καλύτερη από αυτή του προηγούμενου σεναρίου με τα σφάλματα συντονιστών. Αυτό θα οφείλεται στο γεγονός ότι στο τρέχων σενάριο οι συντονιστές πριν καταρρεύσουν με μία πιθανότητα 0,5 στέλνουν μηνύματα περίληψης. Ο αριθμός μηνυμάτων του αλγορίθμου AN αναμένεται να είναι σε πιο χαμηλά επίπεδα έναντι του Do-Um λόγο της ύπαρξης δικτύου με αξιόπιστη πολυεκπομπή σε αυτόν. Η πολυπλοκότητα εργασίας και αυτή με την σειρά της αναμένουμε να είναι καλύτερη από το προηγούμενο σενάριο και συγκεκριμένα η πολυπλοκότητα εργασίας του αλγόριθμου AN θα βρίσκεται σε πιο χαμηλά επίπεδα παρά του Do-Um. Όπως και για την πολυπλοκότητα Μηνυμάτων, έτσι και εδώ, καθοριστικό ρόλο σε αυτά τα αποτελέσματα διαδραματίζει η ύπαρξη αξιόπιστης πολλαπλής διανομής στον αλγόριθμο AN. Η πρώτη γραφική που παρουσιάζεται (Γράφημα 6.7) συσχετίζεται με την πολυπλοκότητα Μηνυμάτων και η δεύτερη (Γράφημα 6.8) με την πολυπλοκότητα Εργασίας.



Γράφημα 6.7 Πολυπλοκότητα Μηνυμάτων για το σενάριο Σφαλμάτων Αποστολής Συντονιστών.



Γράφημα 6.8 Πολυπλοκότητα Εργασίας για το σενάριο Σφαλμάτων Αποστολής Συντονιστών.

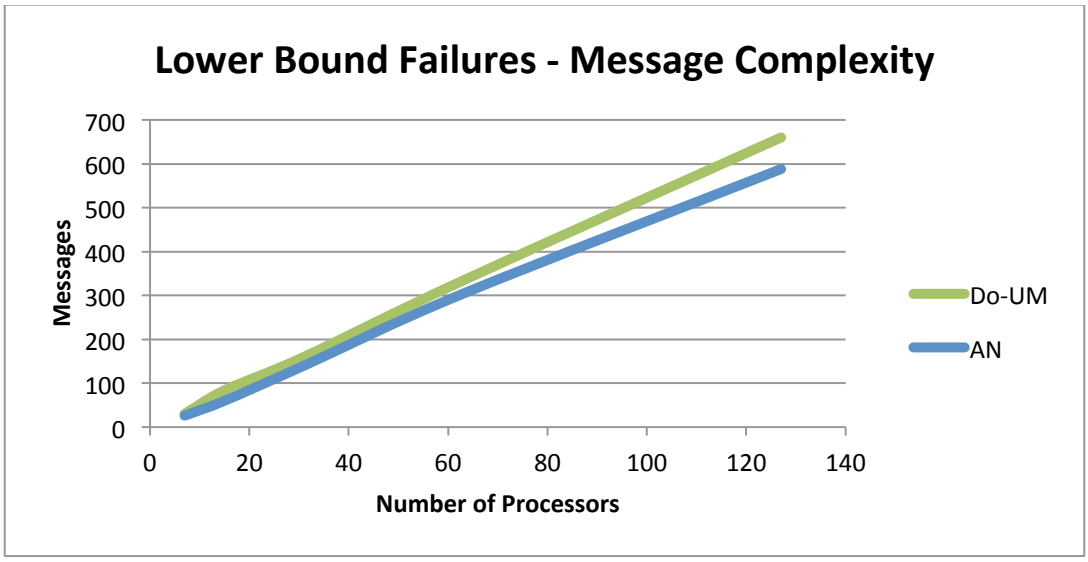
Και στις δύο γραφικές στον άξονα χ συναντάμε τον αριθμό *n* των επεξεργαστών που λαμβάνουν μέρος στον υπολογισμό. Για την γραφική παράσταση που αφορά την πολυπλοκότητα μηνυμάτων (Γράφημα 6.7), στον άξονα ψ λαμβάνει χώρο ο συνολικός αριθμός μηνυμάτων που απεστάλησαν κατά την διάρκεια της κάθε εκτέλεσης. Στην περίπτωση της γραφικής παράστασης που αναλύει την πολυπλοκότητα εργασίας (Γράφημα 6.8), στον άξονα ψ συναντάμε τον συνολικό αριθμό βημάτων που έγιναν στο σύνολο όλων των επεξεργαστών μέχρι να ολοκληρωθεί πλήρως η κάθε εκτέλεση.

Αναλύοντας τις πιο πάνω γραφικές παραστάσεις παρατηρούμε όντως τον αλγόριθμο AN να υπερτερεί του Do-Um και να επικυρώνουν τα όσα αναφέρθηκαν προηγουμένως. Επομένως, επαληθεύονται τα αναμενόμενα αποτελέσματα που προκύπτουν από την πιο πάνω θεωρητική ανάλυση.

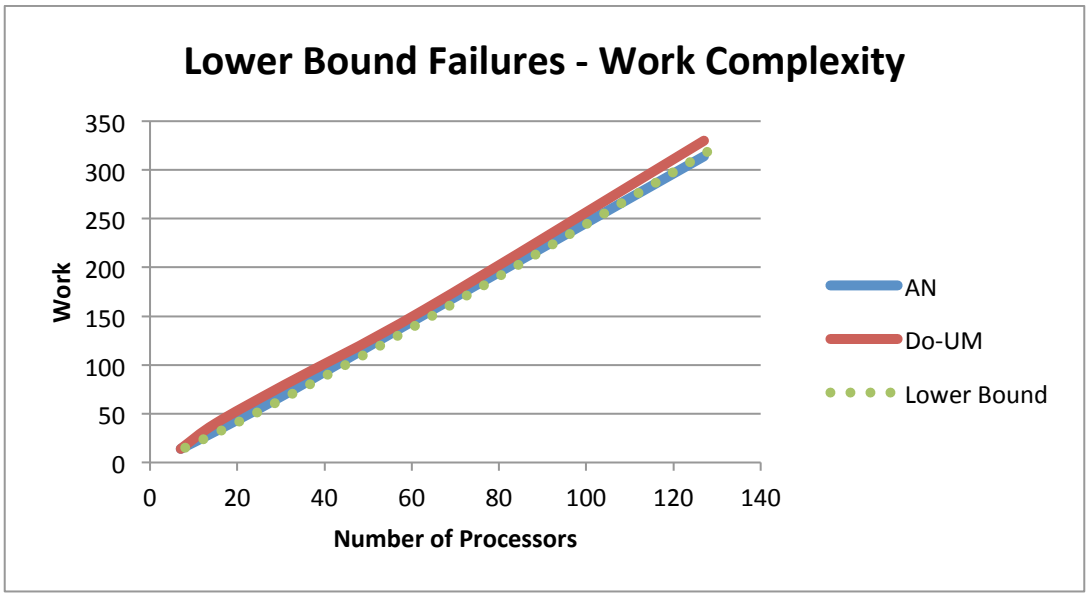
6.5 Σύγκριση ως Προς τον Σενάριο Κατώτατου Ορίου

Σε αυτό το σενάριο αναμένουμε την πολυπλοκότητα εργασίας και για τους δύο αλγόριθμους, AN και Do-Um, να κυμαίνεται στα επίπεδα του κατώτατου ορίου. Συγκεκριμένα, γνωρίζουμε από το [11] ότι αυτή η στρατηγική προκαλεί πολυπλοκότητα εργασίας της τάξεως του $\Omega(t + n\frac{\log n}{\log\log n})$. Για την πολυπλοκότητα

μηνυμάτων αναμένουμε την ίδια συμπεριφορά και από τους δύο αλγορίθμους. Πιο αναλυτικά, αν αναλογιστούμε ότι στον αλγόριθμο Do-Um πρώτα λαμβάνει χώρο η επικοινωνία και μετά η εκτέλεση εργασίας, αυτό θα έχει σαν αντίκτυπο να εκτελείται μια επιπλέον επανάληψη σε σχέση με τον αλγόριθμο AN. Επομένως, ο αλγόριθμος AN θα πρέπει να έχει πιο αποδοτική πολυπλοκότητα μηνυμάτων έναντι του αλγόριθμου Do-Um. Η πρώτη γραφική που παρουσιάζεται (Γράφημα 6.9) συσχετίζεται με την πολυπλοκότητα Μηνυμάτων και η δεύτερη (Γράφημα 6.10) με την πολυπλοκότητα Εργασίας.



Γράφημα 6.9 Πολυπλοκότητα Μηνυμάτων για το σενάριο Κατώτατου Ορίου.



Γράφημα 6.10 Πολυπλοκότητα Εργασίας για το σενάριο Κατώτατου Ορίου.

Και στις δύο γραφικές στον άξονα χ συναντάμε τον αριθμό *n* των επεξεργαστών που λαμβάνουν μέρος στον υπολογισμό. Για την γραφική παράσταση που αφορά την πολυπλοκότητα μηνυμάτων (Γράφημα 6.9), στον άξονα ψ λαμβάνει χώρο ο συνολικός αριθμός μηνυμάτων που απεστάλησαν κατά την διάρκεια της κάθε εκτέλεσης. Στην περίπτωση της γραφικής παράστασης που αναλύει την πολυπλοκότητα εργασίας (Γράφημα 6.10), στον άξονα ψ συναντάμε τον συνολικό αριθμό βημάτων που έγιναν στο σύνολο όλων των επεξεργαστών μέχρι να ολοκληρωθεί πλήρως η κάθε εκτέλεση.

Για να μπορέσουμε να αναλύσουμε ορθά την πολυπλοκότητα εργασίας και να εξάγουμε έγκυρα αποτελέσματα, στο γράφημα που συσχετίζεται με αυτή έχουμε χαράξει και την συνάρτηση που αντιπροσωπεύει το κατώτατο όριο εργασίας. Συγκεκριμένα, μετά από μελέτη που έγινε στο [11] η συνάρτηση αυτή είναι η $(n \log n) \,/\, \log \log n$. Όπως παρατηρούμε και από το γράφημα αυτό, όντως το σενάριο αυτό αναγκάζει και τους δύο αλγόριθμους να εκτελούν εργασία που κυμαίνεται στα επίπεδα του κατώτατου ορίου. Οι δύο αλγόριθμοι έχουν όμοια συμπεριφορά διότι το σενάριο αυτό του κατώτατου ορίου δεν έχει να συμμεριστεί με την επικοινωνία, η οποία ουσιαστικά είναι και η κύρια διαφορά των δύο αλγόριθμων. Στην γραφική παράσταση που αντιπροσωπεύει την πολυπλοκότητα μηνυμάτων τα αποτελέσματα είναι όντως τα αναμενόμενα με τον αλγόριθμο AN να υπερτερεί έναντι του Do-Um. Επομένως, επικυρώνονται τα όσα προαναφέρθηκαν και επαληθεύονται τα αναμενόμενα αποτελέσματα που προκύπτουν από την πιο πάνω θεωρητική ανάλυση.

6.6 Συμπεράσματα

Μέσα από αυτή τη σύγκριση των δύο αλγορίθμων ως προς τον αριθμό των μηνυμάτων και την πολυπλοκότητα εργασίας, παρατηρήσαμε όλα τα πειραματικά αποτελέσματα από την εκτέλεση των αλγορίθμων στο PlanetLab βάση των διαφόρων σεναρίων, να επικυρώνουν και να επιβεβαιώνουν τη θεωρητική ανάλυση των αλγορίθμων. Παρατηρήσαμε επίσης τον αλγόριθμο Do-Um ο οποίος δεν βασίζεται σε

ύπαρξη δικτύου με αξιόπιστη πολυεκπομπή να επιλύει το πρόβλημα Do-All με επιδόσεις οι οποίες είναι συγκρίσιμες ως προς τον αλγόριθμο AN, ο οποίος βασίζεται στην ύπαρξη δικτύου με αξιόπιστη πολυεκπομπή.

Συμπερασματικά, η επιλογή ενός εκ των δύο αλγόριθμων για την επίλυση του προβλήματος Do-All στο μοντέλο ανταλλαγής μηνυμάτων καθορίζεται αποκλειστικά από τον παράγοντα της αξιοπιστίας του δικτύου. Αυτό οφείλεται στο γεγονός ότι ο αλγόριθμος AN αδυνατεί να επιλύσει το πρόβλημα αυτό χωρίς την ύπαρξη αξιοπιστίας δικτύου ενώ ο αλγόριθμος Do-Um είναι ικανός να το επιλύσει, ανεξαρτήτου ύπαρξης η όχι αξιοπιστίας στο δίκτυο. Θα ήταν όμως ανορθόδοξη η επιλογή του αλγορίθμου Do-Um έναντι του AN για την επίλυση του προβλήματος Do-All σε αξιόπιστο δίκτυο, εφόσον ο δεύτερος είναι σαφώς πιο αποδοτικός από τον πρώτο τόσο στην πολυπλοκότητα εργασίας τόσο και στην πολυπλοκότητα μηνυμάτων.

Κεφάλαιο 7

Επίλογος

7.1 Γενικά Συμπεράσματα

Το πρόβλημα της συνεταιριστικής εκτέλεσης μεγάλων υπολογιστικών συνόλων εργασιών από κατανεμημένες υπολογιστικές οντότητες, γνωστό ως πρόβλημα Do-All, έχει απασχολήσει πολλούς ερευνητές. Είναι ένα πρόβλημα το οποίο βρίσκει εφαρμογή σε διάφορα μοντέλα υπολογισμού κάτω από διάφορες προϋποθέσεις σχετικά με τον συγχρονισμό, σφάλματα, αποτυχίες/αστοχίες και ύπαρξης η όχι αξιοπιστίας δικτύου. Οι ερευνητές αυτοί προσπάθησαν να αναπτύξουν διάφορους αλγόριθμους που να είναι ικανοί να επιλύουν το πρόβλημα αυτό και ταυτόχρονα να είναι αποδοτικοί ως προς τις διάφορες μετρικές που αξιολογούνται. Η επιλογή ενός τέτοιου εύρωστου αλγορίθμου, δηλαδή αλγορίθμου αποδοτικού και συνάμα ανεκτικού σε σφάλματα, είναι καθοριστική για τη γενικότερη απόδοση στην επίλυση του προβλήματος.

Στην εργασία αυτή υλοποιήθηκαν και αξιολογήθηκαν τρεις αλγόριθμοι που επιλύουν το πρόβλημα Do-All βασισμένοι στο μοντέλο ανταλλαγής μηνυμάτων. Αυτοί οι αλγόριθμοι αν και ήταν βασισμένοι ίδια λογική, δηλαδή στην χρήση πολλαπλών συντονιστών, και είχαν αρκετά κοινά στοιχεία ως προς την δομή και την κατασκευή

τους, εντούτοις κάθε ένας από αυτούς εξυπηρετούσε διαφορετικές απαιτήσεις. Αξίζει να σημειώσουμε ότι σε αρκετά σενάρια που εξετάσαμε και οι τρεις αλγόριθμοι είχαν παρόμοια συμπεριφορά και αποτελέσματα ως προς την πολυπλοκότητα εργασίας και μηνυμάτων.

Η πειραματική αξιολόγηση του κάθε αλγορίθμου ξεχωριστά, τις πλείστες φορές οδήγησε σε κοινά συμπεράσματα με τους υπόλοιπους αλγορίθμους. Κύριο συμπέρασμα είναι το γεγονός ότι με την ορθή χρήση και επιλογή του καθενός από αυτούς τους αλγόριθμους, μπορούμε αποδοτικά να επιλύσουμε το πρόβλημα Do-All ακόμη και αν κατά την εκτέλεση υπάρχουν καταρρεύσεις/επανεκκινήσεις κόμβων ή δεν μας παρέχεται κατ' ανάγκη αξιοπιστία δικτύου.

Εάν το κατανεμημένο περιβάλλον το οποίο θα γίνει εφαρμογή κάποιου αλγορίθμου για την επίλυση του προβλήματος Do-All παρέχει στον χρήστη αξιόπιστη πολυεκπομπή τότε η πιο αποδοτικές επιλογές που έχει ο χρήστης είναι μεταξύ των αλγορίθμων AN και AR. Συγκεκριμένα εάν το κατανεμημένο σύστημα επιτρέπει επανεκκινήσεις επεξεργαστών τότε αποκλειστική επιλογή είναι αυτή του αλγορίθμου AR. Σε αντίθετη περίπτωση όποια και να είναι η επιλογή, AN ή AR, τότε η απόδοση θα είναι ακριβώς στα ίδια επίπεδα εφόσον ο αλγόριθμος AR χωρίς επανεκκινήσεις συμπεριφέρεται πανομοιότυπα με τον AN. Η χρησιμότητα του αλγορίθμου Do-Um βρίσκεται στο γεγονός ότι δεν προαπαιτεί την ύπαρξη δικτύου με αξιόπιστη πολυεκπομπή. Επομένως, σε ένα τέτοιο κατανεμημένο περιβάλλον που απουσιάζει η αξιοπιστία δικτύου, η επιλογή του αλγορίθμου Do-Um είναι αναγκαστική εφόσον κανείς από τους AN και AR δεν θα είναι ικανός να αντεπεξέλθει σε τέτοιες συνθήκες και να επιλύσει αποδοτικά το πρόβλημα Do-All.

Συνοψίζοντας, αντιλαμβανόμαστε ότι είναι δύσκολο ένας αλγόριθμος να προσφέρει όλα τα πλεονεκτήματα που μπορεί να έχει ένας όμοιος του αλγόριθμος που να επιλύει το πρόβλημα Do-All, χωρίς να αυξάνει το κόστος του αλγορίθμου είτε σε πολυπλοκότητα εργασίας είτε σε πολυπλοκότητα μηνυμάτων. Ανάλογα, λοιπόν, με τις ανάγκες και απαιτήσεις του κάθε κατανεμημένου συστήματος θα πρέπει κάθε φορά να επιλέγεται εκείνος ο αλγόριθμος, το κόστος του οποίου δεν θα ξεπερνά το ελάχιστο κόστος που απαιτεί ο χρήστης.

7.2 Δυσκολίες που Παρουσιάστηκαν και Ξεπεράστηκαν

Στα πλαίσια αυτής της Διπλωματικής Εργασίας παρουσιάστηκαν διάφορες δυσκολίες, τόσο στην υλοποίηση με την χρήση της βιβλιοθήκης YALPS όσο και στην εφαρμογή των αλγορίθμων στο PlanetLab, οι οποίες σε μελλοντικά στάδια ξεπεράστηκαν. Οι δυσκολίες αυτές μπορεί να ήταν χρονοβόρες, όμως οι μελλοντικές ευκολίες που μας παρείχαν τόσο στην υλοποίηση όσο και στην εκτέλεση των αλγορίθμων ήταν τεράστιας αξίας.

Πιο αναλυτικά, σε πρώτη φάση πρόβλημα αποτέλεσε η αναγκαία αυτοδίδακτη εκμάθηση της βιβλιοθήκης YALPS για την υλοποίηση των αλγορίθμων. Η δυσκολία αυτή οφείλεται στο γεγονός ότι η βιβλιοθήκη YALPS έχει δημιουργηθεί πρόσφατα. Επομένως, η μόνη βοήθεια που μας παρείχε η επίσημη ιστοσελίδα του YALPS [4] ήταν ένα πρόχειρο εγχειρίδιο χρήσης που είχε αναρτηθεί σε αυτήν. Αφιερώθηκαν αρκετές ώρες σε αυτό το εγχειρίδιο σε συνδυασμό με προηγούμενες διπλωματικές εργασίες, που ασχολήθηκαν επίσης με την υλοποίηση αλγορίθμων μέσω YALPS, έτσι ώστε να γίνει κατανοητή η βιβλιοθήκη και συνεπώς να γίνει εφικτή η υλοποίηση και η επιτυχές εκτέλεση των αλγορίθμων σε αυτή.

Το επόμενο πρόβλημα που παρουσιάστηκε και έπρεπε να τύχει επίλυσης ήταν βασισμένο στην πειραματική αξιολόγηση των αλγορίθμων που υλοποιήθηκαν. Αρχικά, κατά την εφαρμογή των αλγορίθμων στο PlanetLab ο αριθμός των κόμβων που μπορούσαν να ανατεθούν στο μέρισμα μας ήταν φραγμένος μέχρι τους 150, επομένως, αξιοποιήθηκαν μόνο 128 μηχανές.

Σε δεύτερο στάδιο παρουσιάστηκε το πρόβλημα του συγχρονισμού. Συγκεκριμένα, και οι τρεις αλγόριθμοι AN, AR και Do-Um είναι βασισμένοι σε συγχρονισμένο μοντέλο υπολογισμού. Το PlanetLab ως ένα πραγματικό κατανεμημένο διαδικτυακό περιβάλλον δεν προσφέρει καμία εγγύηση ως προς τον συγχρονισμό των μηχανών. Για να γίνει εφικτός ο συγχρονισμός αυτός των υπολογιστικών οντοτήτων του PlanetLab έπρεπε αρχικά να επιλέξουμε για το μέρισμα

μας μηχανές τέτοιες ώστε ο χρόνος ρολογιού τους στο σύνολο να κυμαίνεται στα ίδια επίπεδα. Ακολούθως θα έπρεπε η κάθε μηχανή εφόσον δεχόταν την εντολή εκκίνησης να ξεκινήσει την εκτέλεση του αλγορίθμου ταυτόχρονα με τις υπόλοιπες μηχανές. Αυτό τελικά έγινε κατορθωτό μέσα από αυτοδίδακτη εκμάθηση δημιουργίας εκτελέσιμων αρχείων τύπου shell. Τα αρχεία αυτού του τύπου είναι μία ακολουθία εντολών που δίνουν στον χρήστη πρόσβαση στο λειτουργικό σύστημα, έτσι ώστε παρά την λήψη της εντολής έναρξης σε κάθε κόμβο επιτρέψαμε στην εκκίνηση εκτέλεσης να λάβει χώρο μετά από ένα συγκεκριμένο χρονικό διάστημα.

Πρόβλημα εμφανίστηκε επίσης κατά την εκτέλεση των αλγορίθμων στο PlanetLab στην παθητική μέθοδο receive() που χρησιμοποιούσε η εφαρμογή YALPS η οποία ήταν υπεύθυνη για την παραλαβή μηνυμάτων. Ενώ ο κάθε κόμβος έστελνε χωρίς πρόβλημα τα μηνύματα του σε γειτονικούς κόμβους, αυτοί με την σειρά τους αδυνατούσαν να τα παραλάβουν. Αποταθήκαμε σε μέλος της ερευνητικής ομάδας του ιδρύματος INRIA για συμβουλές περί του προβλήματος και τελικά έγινε εφικτή η επίλυση του όταν το συγκεκριμένο μέλος έστειλε μια νέα αναβαθμισμένη εκδοχή της βιβλιοθήκης YALPS.

Παρόλα τα προβλήματα που συναντήθηκαν, όλα αυτά ξεπεράστηκαν επιτυχώς προσφέροντας μας μάλιστα μια ύψιστης σημασίας τριβή με προβλήματα που απασχολούν καθημερινά τον κλάδο του κατανεμημένου προγραμματισμού στον τομέα της πληροφορικής.

7.3 Μελλοντική Εργασία

Οι στόχοι της εργασίας σε γενικά επίπεδα έχουν επιτευχθεί πλήρως. Βάση της εργασίας αυτής, προέκυψαν όμως, απορρέοντα θέματα τα οποία θα μπορούσαν να μελετηθούν σε μελλοντικές εργασίες έτσι ώστε να δημιουργηθεί μια πιο ολοκληρωμένη εικόνα στην αξιολόγηση τόσο των πιο πάνω αλγορίθμων όσο και

άλλων κατανεμημένων αλγορίθμων που βασίζονται στο μοντέλο ανταλλαγής μηνυμάτων.

Υλοποιήσαμε τρεις κατανεμημένους αλγορίθμους δυναμικού προγραμματισμού που επιλύουν το πρόβλημα Do-All βασισμένοι πλήρως στην ιδεολογία της βιβλιοθήκης YALPS. Αν και συναντήσαμε αρκετές δυσκολίες αρχικά, μόλις αυτές ξεπεράστηκαν, η υλοποίηση απλοποιήθηκε σε μεγάλο βαθμό. Επομένως, πρόκληση για μελλοντική εργασία θα αποτελούσε το γεγονός υλοποίησης των πιο πάνω αλγόριθμων χωρίς την χρήση της βιβλιοθήκης YALPS. Συγκεκριμένα, η υλοποίηση εξ' ολοκλήρου θα είναι βασισμένη στην γλώσσα αντικειμενοστρεφής προγραμματισμού Java, κτίζοντας έτσι από το μηδέν όλες τις απαραίτητες διεπαφές και μεθόδους που στο σύνολο τους θα αντιπροσώπευαν ένα κατανεμημένο αλγόριθμο δυναμικού προγραμματισμού. Σημαντικό προνόμιο που θα αποκτούσαμε από μια τέτοια υλοποίηση θα ήταν η σύγκριση αλγορίθμων υλοποιημένων τόσο με την βιβλιοθήκη YALPS όσο και με τις δικές μας ανεπτυγμένες βιβλιοθήκες.

Από τους αλγορίθμους που υλοποιήσαμε και αξιολογήσαμε, ο αλγόριθμος Do-Um είναι ο μόνος εκ των τριών για τον οποίο δεν έχει ακόμη λάβει χώρο η θεωρητική του ανάλυση και αξιολόγηση. Επομένως, σημαντική μελλοντική εργασία θα μπορούσε να αποτελέσει η θεωρητική ανάλυση του αλγορίθμου Do-Um.

Μελετήσαμε αλγόριθμους συνεταιριστικού κατανεμημένου υπολογισμού οι οποίοι στο σύνολο τους επιλύουν το πρόβλημα Do-All για το μοντέλο ανταλλαγής μηνυμάτων. Βάσει των αλγορίθμων που μελετήσαμε, θα μπορούσαμε να τους κατατάξουμε, με γνώμονα τις απαιτήσεις τους, σε αυτούς που προϋποθέτουν ή όχι την ύπαρξη δικτύου που υποστηρίζει αξιόπιστη πολυεκπομπή. Εφαρμόζοντας αυτή την νοητή κατάταξη αντιλαμβανόμαστε ότι θα ήταν ορθολογικό, όπως ο αλγόριθμος AR επιτρέπει επανεκκινήσεις επεξεργαστών κατά τον υπολογισμό και είναι βασισμένος στον AN, για μελλοντική εργασία, να μελετούσαμε και να αναπτύσσαμε ένα αλγόριθμο βασισμένο στην ιδεολογία του Do-Um. Ένας αλγόριθμος τέτοιος ώστε χωρίς να προαπαιτεί αξιοπιστία δικτύου πολλαπλής διανομής, θα μπορούσε να επιλύσει το πρόβλημα Do-All και θα ήταν ικανός να διαχειριστεί επανεκκινήσεις επεξεργαστών.

Αναφορές

1. Georgiou, C. and A.A. Shvartsman, *Do-All Computing in Distributed Systems: Cooperation in the Presence of Adversity*. 2008: Springer.
2. Georgiou, C. and A.A. Shvartsman, *Cooperative Task-oriented Computing: Algorithms and Complexity*. 2011: Morgan & Claypool.
3. Davtyan, S., et al., *Coordinated Cooperative Work Using Undependable Processors with Unreliable Broadcast.* 2013.
4. Yalps. *Open-source Java library*. 2010; Available from: http://yalps.gforge.inria.fr.
5. Chun, B., et al., *PlanetLab: an overlay testbed for broad-coverage services.* SIGCOMM Comput. Commun. Rev., 2003. **33**(3): p. 3-12.
6. Chlebus, B.S., R. De Prisco, and A.A. Shvartsman, *Performing tasks on restartable message-passing processors*, in *Distributed Algorithms*. 1997, Springer. p. 96-110.
7. Chlebus, B.S., D.R. Kowalski, and A. Lingas. *The do-all problem in broadcast networks*. in *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*. 2001. ACM.
8. Georgiou, C., D.R. Kowalski, and A.A. Shvartsman, *Efficient gossip and robust distributed computation.* Theoretical Computer Science, 2005. **347**(1): p. 130-166.
9. INRIA. Available from: http://www.inria.fr/.
10. Dwork, C., J.Y. Halpern, and O. Waarts, *Performing work efficiently in the presence of faults.* SIAM Journal on Computing, 1998. **27**(5): p. 1457-1491.
11. Kanellakis, P.C. and A.A. Shvartsman, *Fault-tolerant parallel computation*. 1997: Kluwer Academic Publishers.
12. Anderson, R.J. and H. Woll, *Algorithms for the certified write-all problem.* SIAM Journal on Computing, 1997. **26**(5): p. 1277-1283.
13. Malewicz, G., *A work-optimal deterministic algorithm for the certified write-all problem with a nontrivial number of asynchronous processors.* SIAM Journal on Computing, 2005. **34**(4): p. 993-1024.
14. Georgiou, C., A. Russell, and A.A. Shvartsman, *Work-competitive scheduling for cooperative computing with dynamic groups.* SIAM Journal on Computing, 2005. **34**(4): p. 848-862.
15. Malewicz, G., A. Russell, and A.A. Shvartsman, *Distributed scheduling for disconnected cooperation.* Distributed Computing, 2006. **18**(6): p. 409-420.
16. De Prisco, R., A. Mayer, and M. Yung. *Time-optimal message-efficient work performance in the presence of faults*. in *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*. 1994. ACM.
17. Galil, Z., A. Mayer, and M. Yung. *Resolving message complexity of byzantine agreement and beyond*. in *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on*. 1995. IEEE.
18. *Intel Corporation*. 1968; Available from: http://www.intel.com/.
19. *Hewlett-Packard Company*. 1957; Available from: http://www.hp.com/.
20. *PlanetLab*. 2003; Available from: https://http://www.planet-lab.org/doc.
21. *Eclipse Juno*. 2001; Available from: http://www.eclipse.org/juno/.

Παράρτημα Α

Κώδικας Αλγορίθμου ΑΝ σε γλώσσα προγραμματισμού JAVA

```
import java.net.UnknownHostException;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.Set;
import java.util.concurrent.TimeUnit;
import yalps.BasicPeriodicTask;
import yalps.NodeID;
import yalps.TaskManager;
import yalps.executor.E_NodeID;
import yalps.launchers.AbstractYalpsNode;
import yalps.taskgroup.SystemTaskGroup;


public class BroadcastApplicationWithSerializable extends AbstractYalpsNode{


/* Variable Declarations
 * String job          -> Keeps the name of the job we done
 * int init_port       -> Keeps the initial number of the first port
 * int msg_count       -> Keeps the received number of messages
 * int total_msgs      -> Keeps the total number of sent messages
 * int id              -> Keeps the id of the processor
 * int round           -> Keeps the current round id
 * int no_tasks        -> Keeps the total number of tasks to be done
 * int level           -> Keeps the level of the phase
 * int l_pos           -> Keeps our position in Local view
 * int times_s         -> Keeps the number of S_Messages we get each round
 * boolean alive       -> Declares if as a processor i am alive.
 * boolean attended    -> Declares if a round is attended or not.
 * boolean cordinator  -> Declares if a processor it's set as cordinator
 * bolean[] mark_alive -> Declares if a specific processor is alive or not.
 * ArrayList tasks     -> ArrayList which keeps the set of tasks to be done
 * ArrayList aliveproc -> ArrayList which keeps the set of alive processors
 * ArrayList donetasks -> ArrayList which keeps the set of completed tasks
 * ArrayList utasks    -> ArrayList which keeps the set of utasks
 * ArrayList p         -> ArrayList which keeps the set of alive proc's
 * ArrayList p_tasks   -> ArrayList which keeps the pending tasks i have done
 * ArrayList alive_ids -> ArrayList which keeps the numeric id of each proc
 *
 */

public String job=null;
public int temp_all_nodes=0;
public int l_kill =0;
```

```
public int times_kill=0;
public int init_port=0;
public int msg_count=0;
public int total_msgs=0;
public int id;
public int round=0;
public int no_tasks=0;
public int level=0;
public int l_pos=0;
public int times_s=0;
public boolean alive = true;
public boolean c_send = false;
public boolean attended = false;
public boolean cordinator = false;
boolean [] mark_alive;
ArrayList<E_NodeID> aliveproc = new ArrayList<E_NodeID>();
ArrayList<String> tasks = new ArrayList<String>();
ArrayList<String> donetasks = new ArrayList<String>();
ArrayList<String> utasks = new ArrayList<String>();
ArrayList<E_NodeID> p = new ArrayList<E_NodeID>();
ArrayList<String> p_tasks = new ArrayList<String>();
ArrayList<Integer> alive_ids = new ArrayList<Integer>();  ///
protected ArrayList<E_NodeID> DestinationList = new ArrayList<E_NodeID>();


class MessageGenerationTask extends BasicPeriodicTask {

	@Override
	protected TaskManager getTaskManager() {
		return
BroadcastApplicationWithSerializable.this.getTaskManager();
	}

@Override
protected void periodicRun() {

	/*
	 * If it's the first round then:
	 * Initialize the size of boolean table
	 * Initialize node1 as cordinator
	 * Initialize tasks, unaccound tasks , donetasks
	 * Initialize all neighboors
	 * Set the id of each Node
	 */


if (round == 0) {


	//Set the id of each node from port
	System.out.println("I am "+ getNodeId()+" .My node id: " + id );
	getOut().println("I am "+ getNodeId()+" .My node id: " + id );

	// Initialize the size of our boolean table
	mark_alive = new boolean [DestinationList.size()];

	// Set Node1 as the cordinator
```

```
        if (id==1) {
                cordinator = true;
        }

        // Initialiaze all known processors
        for (E_NodeID neighbor : DestinationList) {
                aliveproc.add(neighbor);
        }

        System.out.println("Initial AliveProccessors are: " +
        aliveproc.toString());
        getOut().println("Initial AliveProccessors are: " +
        aliveproc.toString());

        // Initialiaze all tasks to be done
        // Initialiaze all Unaccound tasks to be done
        // Donetasks will be empty beacuse of initialize
        // Unaccound tasks will be same as tasks at initialize
        for (int i = 1; i <= no_tasks; i++) {
                tasks.add("task" + i);
                utasks.add("task" + i);
        }
        donetasks.clear();

} else {

        /* It's not the initialize round. Start the
         * implementation of algorithm AN.
         * We suppose that 3 rounds of a yalps task will
         * represent 1 phase of our algorithm.
         * This is made up to help us with sync.
         */

        /*
         * This if block is the 1st stage of our algorithm where
         * An alive node will get a task, Create a Report message
         * and sent this message to all coordinators
         * corresponding to it's local view.
         */

if((round%3 ==1)&&alive){


        //Get our position from local view.
        for(int i=0; i<aliveproc.size(); i++){

        if(aliveproc.get(i).toString().equals(getNodeId().toString())){
                l_pos=i;
                break;
                }
        }

                //Take the job with Load Balance rule.
                //We must remove it local from Utasks
                //and add it in pending tasks list.
                job = utasks.get(l_pos % utasks.size());
                utasks.remove(l_pos % utasks.size()); //
```

```
                p_tasks.add(job);

                System.out.println("Node"+id+" "+getNodeId() + "get the " + job);
                getOut().println("Node"+id+" "+getNodeId() + "get the " + job);

                // Create and send a report message to all coordinators
                // corresponding to local view
                ReportMessage ms = new ReportMessage(p_tasks,id);

                for(int i=0; i<Math.pow(2, level) && i<aliveproc.size(); i++){

                        try {
                                TimeUnit.MILLISECONDS.sleep(80);
                        } catch (InterruptedException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                        }
                        System.out.println("Node"+id+" "+getNodeId() + " sending
                                report to cordinator: " + aliveproc.get(i));
                        getOut().println("Node"+id+" "+getNodeId() + " sending
                                report to cordinator: " + aliveproc.get(i));
                        MyTCPBasedMsgSenderTask myTask1 = new
                                MyTCPBasedMsgSenderTask(ms, aliveproc.get(i), cl);
                        tm.registerTask(myTask1,  tm.getSystemTaskGroup(), 0);
                        total_msgs++;
                }


        }else if((round%3 == 2)&&alive){

                /*
                 * This if block is the 2nd stage of our algorithm where
                 * a cordinator will Create Summary message with done work
                 * and alive processors and then send it to all alive nodes.
                 */

                if(cordinator){

                        //create Summary messages with alive processors and done work
                        SummaryMessage m3 =new SummaryMessage(donetasks,p,alive_ids,id);

                        //Send the message to all alive nodes
                        for(int i=0; i<p.size(); i++){
                                try {
                                        TimeUnit.MILLISECONDS.sleep(70);
                                } catch (InterruptedException e) {
                                        // TODO Auto-generated catch block
                                        e.printStackTrace();
                                }

                                MyTCPBasedMsgSenderTask myTask2 = new
                                        MyTCPBasedMsgSenderTask(m3, p.get(i), cl);
                                tm.registerTask(myTask2,  tm.getSystemTaskGroup(), 0);
                                total_msgs++;
                        }
```

```
                //empty the alive nodes list for this cordinator
                //so he will fill it up again in next round
                alive_ids.clear();
                p.clear();
                cordinator=false;
        }

}else if ((round%3==0)&&alive){

        /*
         * This if block is the 3rd stage of our algorithm where
         * we are going to check if the phase was attended or not.
         * If it was unattended means we did not get any messages
         * from any cordinator so we have to delete them from our local view.
         * If it was attended we do nothing cz all work was done while we
         * received the messages.
         */

        if(!attended){

                // If phase is unattended then we have to erase
                // all coordinators from our local view

                for(int i=(int) Math.pow(2, level) -1; (i>=0 &&
                        i<aliveproc.size());  i--){
                        System.out.println(getNodeId()+ " Removed cordinator " +
                        aliveproc.get(i) + " Beacuse is dead.");
                        getOut().println(getNodeId()+ " Removed cordinator " +
                        aliveproc.get(i) + " Beacuse is dead.");
                        aliveproc.remove(i);
                }
                level++;
                System.out.println("Unattended Phase. Level updated to" + level);
                getOut().println("Unattended Phase. Level updated to " + level);
                getOut().println("Node"+id+" "+getNodeId()+" My Local view after
                deletion : " + aliveproc.toString());

        }else{
                //phase is Attended. We must empty our pending tasks list.
                System.out.println(getNodeId()+"Phase "+ round/3 + " Attended.");
                getOut().println(getNodeId()+"Phase "+ round/3 + " Attended.");
                p_tasks.clear();
        }

        // We consider that at the beggining of every round
        // the phase will be unattended. This value will be
        // set true in the middle of the round if its really attended.
                attended=false;
    }
   }

        // We increase the round counter
        // and set the variables for the next round.
        round++;
        times_s = 0;
        cordinator=false;
}
```

```
@Override
public long getPeriod() {
	// each round is 1000 milis
	return 1000;
}

@Override
public boolean isActive() {

	/*
	 * Simulation for a Node runs if and only if
	 * he is alive and there are tasks to be done.
	 */

	if((utasks.isEmpty()&&(round>0)&&(round%3==0))||alive==false){
		System.out.println("Node" +id+" " +getNodeId()+" is Finished.");
		System.out.println("Total Messages Sent: " + total_msgs);
		System.out.println("Total Phases are: "+round/3+".");
		System.out.println("Total time: "+ (round/3)*getPeriod()+" ms.");
		getOut().println("Node" +id+" " +getNodeId()+" is Finished.");
		getOut().println("Total Messages Sent: " + total_msgs);
		getOut().println("Total Phases are: "+round/3+".");
		getOut().println("Total time: "+ (round/3)*getPeriod()+" ms.");
		System.exit(0);
			return false;
		}else
			return true;
	}

}

/**
 * The set of nodes to which to send messages
 */
private Set<NodeID> nodes=new HashSet<NodeID>();
/**
 * The current sequence number for messages generated by this node.
 */

@Override
public void startNode() {

	tm.registerPeriodicTask(new MessageGenerationTask(),
SystemTaskGroup.getInstance(), 0);

}

@Override
public boolean receive(short header, Object msg, NodeID sender) {

	/*
	 * Implementation of the receive method.
	 * Here we divide the method to 2 cases
	 * since we have 2 kinds of messages.
	 * For each case we act accordingly
	 */
```

```
E_NodeID snd = (E_NodeID) sender;
int snd_id = 0;


if (msg instanceof ReportMessage) {
	/*
	 * Since we get this kind of a message
	 * we know that we must act as cordinator.
	 * We must fill the data with done work.
	 * After this we must save the sender as alive.
	 */

	if(alive){
		cordinator=true;

		ReportMessage m=(ReportMessage)msg;
		snd_id = m.id;

		for(int i=0; i<m.pending.size(); i++){

			if(!donetasks.contains(m.pending.get(i)))
				donetasks.add(m.pending.get(i));

		}

		alive_ids.add(snd_id);
		p.add(snd);
		msg_count++;

		System.out.println("Cordinator Node"+id+" "+getNodeId() +"
		Just read a Report Message from Node"+snd_id+" "+snd);
		getOut().println("Cordinator Node"+id+" "+getNodeId() +"
		Just read a Report Message from Node"+snd_id+" "+snd);
	}

}else if (msg instanceof SummaryMessage){

	/*
	 * Since we get this kind of a message
	 * we know that it's sender is a cordinator,
	 * hence the phase will be marked as attended.
	 * Also our local view level will set up to 0.
	 */

	attended=true;
	level=0;

	//Get the message with done work and alive processors
	SummaryMessage m =(SummaryMessage)msg;
	snd_id = m.id;

	// Update our knowledge according to message info.
	// Updated knowledge will be for the Done Work and the Utasks.
	// Unaccound = Tasks - Done
	for(int i=0; i<m.Dw.size(); i++){
		if(!donetasks.contains(m.Dw.get(i)))
			donetasks.add(m.Dw.get(i));
```

```
                        if(utasks.contains(m.Dw.get(i)))
                                utasks.remove(m.Dw.get(i));

                }

                //Check printings
                System.out.println();
                System.out.println("Done tasks " + donetasks.toString());
                getOut().println();
                getOut().println("Done tasks " + donetasks.toString());

                //Initialize all processors as dead.
                //At next mark alive ones one by one.
                for(int k=0; k<mark_alive.length; k++)
                        mark_alive[k]=false;

                times_s++;
                if(times_s==1){
                        aliveproc.clear();
                }

                //mark all alive processors at boolean mark_alive table
                for(int i=0; i<m.Pids.size(); i++){

                        int node_pos = m.Pids.get(i) - 1 ;
                        mark_alive [node_pos] = true;
                }

                //after marking all alive we have to add them sorted
                //to the aliveproc list with linear reading of all coordinators
                int pos=0;
                for (E_NodeID neighbor : DestinationList) {
                        //if it's alive we add him to the list
                        if(mark_alive[pos]==true){
                                if(!aliveproc.contains(neighbor))
                                        aliveproc.add(neighbor); }
                        pos++;
                }

                //check message
                System.out.println("Alive processors: " + aliveproc.toString());
                getOut().println("Alive processors: " + aliveproc.toString());

                //delete the m msg
                m.clear();

                System.out.println("Node"+id+" "+getNodeId() +" Just read a
                Summary Message from Cordinator Node"+snd_id+" "+snd);
                System.out.println();
                getOut().println("Node"+id+" "+getNodeId() +" Just read a Summary
                Message from Cordinator Node"+snd_id+" "+snd);
                getOut().println();
        }

        return true;
}
```

```
/**
 * @param the name of the file from which to read the list of nodes
 * Get their id's sepperated by semicollum
 */
public void setNodeList(String nodeList) {
	String[] nodeStrings=nodeList.split(";");
	for (String node : nodeStrings) {
		try {
			nodes.add(cl.getNodeIdFromString(node));
		} catch (UnknownHostException e) {
			e.printStackTrace();
		}
	}
}

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();
	}

}

public void setInitialport(int port){
	System.out.println("Initial Port is: " + port);
	getOut().println("Initial Port is: " + port);
	init_port=port;
}

public void setTasks(int ta){
	System.out.println("Total num of tasks are: " + ta);
	getOut().println("Total num of tasks are: " + ta);
	no_tasks=ta;
}

public void setId(int n_id){
	System.out.println("My id is: " + n_id);
	getOut().println("My id is: " + n_id);
		id=n_id;
	}

}
```

```java
/*
 * This is the class which implements the kind of
 * the message which will be sent from a Node to the
 * cordinator reporting to him that he finished a specific tasks.
 */

import java.io.Serializable;
import java.util.ArrayList;

class ReportMessage implements Serializable{

	private static final long serialVersionUID = 1L;

	// ArrayList with Pending Done Work and Sender id
	ArrayList<String> pending = new ArrayList<String>();
	public int id;

	@SuppressWarnings("unchecked")
	ReportMessage (ArrayList<String> value, int id){
		this.pending = (ArrayList<String>) value.clone();
		this.id = id;
	}
}

/*
 * This is the class which implements the kind of
 * the message which will be sent from a Cordinator as Summary
 * to a node reporting to him all done work until now and
 * all alive processors
 */
import java.io.Serializable;
import java.util.ArrayList;
import yalps.executor.E_NodeID;

class SummaryMessage implements Serializable{

	private static final long serialVersionUID = 1L;

	// ArrayList with Done Work
	ArrayList<String> Dw = new ArrayList<String>();
	// ArrayList with Alive Processors Ids
	ArrayList<E_NodeID> Pw = new ArrayList<E_NodeID>();
	// ArrayList with Alive Processors Ids
	ArrayList<Integer> Pids = new ArrayList<Integer>();
	// Sender id
	public int id;

	@SuppressWarnings("unchecked")
	SummaryMessage (ArrayList<String> value , ArrayList<E_NodeID> val,
	ArrayList<Integer> v ,int id){
		this.Dw = (ArrayList<String>) value.clone();
		this.Pw = (ArrayList<E_NodeID>) val.clone();
		this.Pids =(ArrayList<Integer>) v.clone();
		this.id = id;
	}

}
```

```
/*
 * This is the class which implements the send part of a TcpMessage
 */

import java.io.IOException;
import yalps.NodeID;
import yalps.Task;
import yalps.communication.CommunicationLayer;
import yalps.executor.E_NodeID;
import yalps.executor.RealWorldCommunicationLayer;



public class MyTCPBasedMsgSenderTask implements Task {

	E_NodeID destination;
	RealWorldCommunicationLayer commLayer;
	ReportMessage ReportMsg = null; //
	SummaryMessage SummaryMsg = null; //
	MyHeadedCheapSerializable headedCheapMsg = null;
	long taskPeriod = 10000; // Runs each taskPeriod miliseconds 10000


	public MyTCPBasedMsgSenderTask(ReportMessage msg1, NodeID dest,
CommunicationLayer comm){
		this.ReportMsg = msg1;
		this.destination = (E_NodeID) dest;
		this.commLayer = (RealWorldCommunicationLayer) comm;
	}
	public MyTCPBasedMsgSenderTask(SummaryMessage msg2, NodeID dest,
CommunicationLayer comm){
		this.SummaryMsg = msg2;
		this.destination = (E_NodeID) dest;
		this.commLayer = (RealWorldCommunicationLayer) comm;
	}

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

	@Override
	public void run() {

		if(ReportMsg!=null){   // msg to send is Serializable

			try {
				System.out.println("[Application] - is sending a
				Periodic Report Mesage SERIALIZABLE msg with Regular
				TCP  to [" + destination +"]");
				commLayer.sendTCP(ReportMsg, destination);
			} catch (IOException e) {
				e.printStackTrace();
			}
		}
```

```
			if(SummaryMsg!=null){   // msg to send is Serializable

				try {
					System.out.println("[Application] - is sending a
					Periodic Summary Mesage SERIALIZABLE msg with
					Regular TCP  to [" + destination +"]");
					commLayer.sendTCP(SummaryMsg, destination);
				} catch (IOException e) {
					e.printStackTrace();
				}
			}
		}
		public void setPeriod(int period){
			this.taskPeriod = period;
		}

}

import java.io.IOException;
import yalps.communication.GenericDeserializationMap;
import yalps.executor.cheapserializer.HeadedCheapSerializable;



public class MyDeserializationMap extends GenericDeserializationMap {


		public final static short MYHEADEDCHEAPSERIALIZABLE =
		LAST_RESERVED_HEADER +1;

		@Override
		public Class<? extends HeadedCheapSerializable> getClassByHeader (
				short header) throws IOException{

			switch (header) {

			case MYHEADEDCHEAPSERIALIZABLE:
				return MyHeadedCheapSerializable.class;
			default:
				return super.getClassByHeader(header);

			}
		}
}
```

Παράρτημα Β

Κώδικας Αλγορίθμου AR σε γλώσσα προγραμματισμού JAVA

```
import java.net.UnknownHostException;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.Set;
import java.util.concurrent.TimeUnit;
import yalps.BasicPeriodicTask;
import yalps.NodeID;
import yalps.TaskManager;
import yalps.executor.E_NodeID;
import yalps.launchers.AbstractYalpsNode;
import yalps.taskgroup.SystemTaskGroup;


public class BroadcastApplicationWithSerializable extends AbstractYalpsNode{


/* Variable Declarations
 * String job          -> Keeps the name of the job we done
 * int init_port       -> Keeps the inital number of the first port
 * int msg_count       -> Keeps the number of received messages
 * int total_msgs      -> Keeps the total number of sent messages
 * int id              -> Keeps the id of the processor
 * int round           -> Keeps the current round id
 * int no_tasks        -> Keeps the total number of tasks to be done
 * int level           -> Keeps the level of the phase
 * int l_pos           -> Keeps our position in Local view
 * int times_i         -> Keeps the number of info messages we got
 * int times_s         -> Keeps the number of summary messages we got
 * boolean alive       -> Declares if as a processor i am alive.
 * boolean finish      -> Declares if the simulation has finished
 * boolean attended    -> Declares if a round is attended or not.
 * boolean cordinator  -> Declares if a processor it's set as cordinator
 * boolean restart     -> Declares if a processor restarted
 * boolean already     -> Declares if a processor already read a summary msg
 * bolean[] mark_alive -> Declares if a specific processor is alive or not.
 * bolean[] mark_resta -> Declares if a processor was restarted or not.
 * ArrayList tasks     -> ArrayList which keeps the set of tasks to be done
 * ArrayList aliveproc -> ArrayList which keeps the local view
 * ArrayList donetasks -> ArrayList which keeps the set of completed tasks
 * ArrayList utasks    -> ArrayList which keeps the set of Utasks
 * ArrayList p         -> ArrayList which keeps the set of alive processors
 * ArrayList restartd  -> ArrayList which keeps the set of restarted proc's
 * ArrayList p_tasks   -> ArrayList which keeps the pending tasks i have done
 * ArrayList DestinationList -> ArrayList which keeps all the execution proc's
```

```
 * ArrayList alive_ids -> ArrayList which keeps the numeric id of an alive pr
 * ArrayList rest_ids  -> ArrayList which keeps the numeric id of res_proc
 */

public String job=null;
public int init_port=0;
public int msg_count=0;
public int total_msgs=0;
public int id;
public int round=0;
public int no_tasks=0;
public int level=0;
public int l_pos=0;
public int times_i=0;
public int times_s=0;
public boolean alive = true;
public boolean finish = false;
public boolean attended = false;
public boolean cordinator = false;
public boolean restart = false;
public boolean already = false;
boolean [] mark_alive ;
boolean [] mark_restarted;
ArrayList<E_NodeID> aliveproc = new ArrayList<E_NodeID>();
ArrayList<String> tasks = new ArrayList<String>();
ArrayList<String> donetasks = new ArrayList<String>();
ArrayList<String> utasks = new ArrayList<String>();
ArrayList<E_NodeID> p = new ArrayList<E_NodeID>();
ArrayList<E_NodeID> restarted = new ArrayList<E_NodeID>();
ArrayList<String> p_tasks = new ArrayList<String>();
ArrayList<Integer> alive_ids = new ArrayList<Integer>();
ArrayList<Integer> rest_ids = new ArrayList<Integer>();
protected ArrayList<E_NodeID> DestinationList = new ArrayList<E_NodeID>();


class MessageGenerationTask extends BasicPeriodicTask {

@Override
protected TaskManager getTaskManager() {
      return BroadcastApplicationWithSerializable.this.getTaskManager();

}


@Override
protected void periodicRun() {


/*
 * If it's the first round then:
 * Initialize the size of boolean tables
 * Initialize node1 as cordinator
 * Initialize tasks, utasks , donetasks
 * Initialize all neighbors
 * Set the id of each Node
 */
```

```
if (round == 0) {

	System.out.println("I am "+ getNodeId()+" and my node id: " + id );
	getOut().println("I am "+ getNodeId()+" and my node id: " + id );

	// Initialize the size of our two boolean tables
	mark_alive = new boolean [DestinationList.size()];
	mark_restarted = new boolean [DestinationList.size()];

	// Set Node1 as the cordinator
	if (id==1) {
		cordinator = true;
		System.out.println("*** " + getNodeId() + " am Cordinator ***");
		System.out.println();
		getOut().println("*** " + getNodeId()   + " am Cordinator ***");
		getOut().println();

	}

	// Initialiaze all known processors
	for (E_NodeID neighbor : DestinationList) {
		aliveproc.add(neighbor);

	}

	System.out.println("Initial AliveProccessors are: " +
	aliveproc.toString());
	getOut().println("Initial AliveProccessors are: " +
	aliveproc.toString());

	/*
	 * Initialiaze all tasks to be done
	 * Initialiaze all Unaccound tasks to be done
	 * Donetasks will be empty
	 * Unaccound tasks will be same as tasks
	 */

	for (int i = 1; i <= no_tasks; i++) {
		tasks.add("task" + i);
		utasks.add("task" + i);
	}
	donetasks.clear();


} else {

	/* It's not the initialize round. Start the implementation of AR.
	 * We suppose that 3 rounds of a yalps task will represent 1 phase
	 * of our algorithm. This is made up to help us with sync.
	 */


	/*
	 * This if block is the 1st stage of our algorithm where
	 * an alive node will get a task, Create a report message
```

```
 * and sent this message to all cordinators corresponding to it's Lw
 */

if((round%3 ==1)&&alive){

	//Get our position from local view.
	for(int i=0; i<aliveproc.size(); i++){

	if(aliveproc.get(i).toString().equals(getNodeId().toString())){
		l_pos=i;
		break;
	  }
	}

	//Take the job with Load Balance rule.
	//We must remove it local from utasks
	//and add it in pending tasks list.
	job = utasks.get(l_pos % utasks.size());
	utasks.remove(l_pos % utasks.size()); //
	p_tasks.add(job);

	System.out.println(getNodeId() + " get the " + job + " .");
	getOut().println(getNodeId() + " get the " + job + " .");

	// Create and send Report message to all coordinators
	ReportMessage ms = new ReportMessage(p_tasks,id);

	for(int i=0; i<Math.pow(2, level) && i<aliveproc.size(); i++){

		getOut().println("Node"+id+" "+getNodeId() + " sending
		report to cordinator: " + aliveproc.get(i));
		MyTCPBasedMsgSenderTask myTask1 = new
		MyTCPBasedMsgSenderTask(ms, aliveproc.get(i), cl);
		tm.registerTask(myTask1,  tm.getSystemTaskGroup(), 0);
		total_msgs++;
	}

}else if((round%3 == 1)&&restart){

	/*
	 * Now we are in the 1st stage of algorithm
	 * and we want only restarted processors to execute
	 * this procedure of sending restart messages to
	 * all processors.
	 * Restarted proc will also Initialize their variables.
	 */

	restarted.clear();
	rest_ids.clear();
	aliveproc.clear();
	alive_ids.clear();

	//Create and send Restart message to all nodes
	RestartMessage m = new RestartMessage(id);

	for (E_NodeID neighbor : DestinationList) {
```

```
                        try {
                                TimeUnit.MILLISECONDS.sleep(70);
                        } catch (InterruptedException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                        }

                        System.out.println("Node"+id+" "+getNodeId() + " sending
                        Restart Message to: " +neighbor);
                        getOut().println("Node"+id+" "+getNodeId() + " sending
                        Restart Message to: " +neighbor);
                        MyTCPBasedMsgSenderTask myTask2 = new
                        MyTCPBasedMsgSenderTask(m, neighbor, cl);
                        tm.registerTask(myTask2,  tm.getSystemTaskGroup(), 0);
                        total_msgs++;
                }
        }

        else if((round%3 == 2)&&alive){

                /*
                 * This if block is the 2nd stage of our algorithm where
                 * a cordinator will Create Summary messages with done work
                 * and alive processors and then send it to all alive nodes.
                 * Also the rest of processors (and the cordinators) will create
                 * the info message and broadcast it to all restarted nodes
                 */

                if(cordinator){


                //create message with alive processors and done work
                SummaryMessage m3 = new
SummaryMessage(donetasks,p,alive_ids,id);

                        //Send the Summary message messages to all alive nodes
                        for(int i=0; i<p.size(); i++){

                                System.out.println("Node"+id+" "+getNodeId() + "
                                sending Summary Message to: " +p.get(i));
                                getOut().println("Node"+id+" "+getNodeId() + "
                                sending Summary Message to: " +p.get(i));
                                MyTCPBasedMsgSenderTask myTask1 = new
                                MyTCPBasedMsgSenderTask(m3, p.get(i), cl);
                                tm.registerTask(myTask1,tm.getSystemTaskGroup(), 0);
                                total_msgs++;
                        }

                        //Send the Summary message messages to all restarted nodes
                        for(int i=0; i<restarted.size(); i++){

                                System.out.println("Node"+id+" "+getNodeId() + "
                                sending Summary Message to: " +restarted.get(i));
                                getOut().println("Node"+id+" "+getNodeId() + "
                                sending Summary Message to: " +restarted.get(i));
                                MyTCPBasedMsgSenderTask myTask2 = new
```

```
                        MyTCPBasedMsgSenderTask(m3, restarted.get(i), cl);
                        tm.registerTask(myTask2,tm.getSystemTaskGroup(), 0);
                        total_msgs++;
                }

                //empty the alive nodes list for this cordinator
                //so he will fill it up again in next round
                alive_ids.clear();
                p.clear();
                cordinator=false;
        }

//create message with info (Unaccound tasks and local view)
InfoMessage ms = new InfoMessage(utasks,aliveproc,id);

//Send the Info message messages to all restarted nodes
for(int i=0; i<restarted.size(); i++){

        System.out.println("Node"+id+" "+getNodeId() + " sending
        Information Message to: " +restarted.get(i));
        getOut().println("Node"+id+" "+getNodeId() + " sending
        Information Message to: " +restarted.get(i));
        MyTCPBasedMsgSenderTask myTask2 = new
        MyTCPBasedMsgSenderTask(ms,restarted.get(i), cl);
        tm.registerTask(myTask2,  tm.getSystemTaskGroup(), 0);
        total_msgs++;
        }

}else if ((round%3==0)&&alive){

        /*
         * This if block is the 3rd stage of our algorithm where
         * we are going to check if the phase was attended or not.
         * If it was unattended means we did not get any messages
         * from any cordinator so we have to delete them from our Lview.
         * Also we have to add Restarted processors sorted by their ID's
         * at the end of our local view list and remove them from the
         * Restarted list.
         *
         * If the phase was attended we have to update our local view
         * with the restarted nodes by add them sorted to the list.
         * This is beacuse they get all the messages from the coords
         * so they can be now used as the rest of the alive processors.
         */

        if(!attended){

                for(int i=(int) Math.pow(2, level) -1; (i>=0 &&
                i<aliveproc.size());  i--){

                        aliveproc.remove(i);
                }

                // Add Restarted processors sorted by their ID's
                //at the end of local view
```

```
                        //Initialize all processors as not restarted.
                        //At next mark restarted ones one by one.
                        for(int k=0; k<mark_restarted.length; k++)
                              mark_restarted[k]=false;

                        //mark all restarted processors at boolean mark_restarted
                        for(int i=0; i<rest_ids.size(); i++){
                              int node_pos = rest_ids.get(i)-1;
                              mark_restarted [node_pos] = true;
                        }

                        //after marking all restarted we have to add them
                        //sorted at the end of our local view
                        int pos=0;
                        for (E_NodeID neighbor : DestinationList) {

                              //if it's restarted or alive we add him to the alive
                              if(mark_restarted[pos]==true){
                                    if(!aliveproc.contains(neighbor))
                                          aliveproc.add(neighbor);
                              }
                              pos++;
                        }

                        //empty restarted list.
                        restarted.clear();
                        rest_ids.clear();

                        //Increase level and check printings
                        level++;
                        System.out.println("Phase "+ round/3+ " is unattended.");
                        System.out.println("Update my level to: " + level);


                  }else{
                        //phase is Attended - all work with updates have been done
                        //in the receive SummaryMessage phase.
                        //We must also empty our pending tasks list.

                        System.out.println("Phase "+ round/3+ " is attended.");
                        p_tasks.clear();
                  }

                  // We consider that at the beggining of every round
                  // the phase will be unattended. This value will be
                  // set true in the middle of the round if its really attended.
                  attended=false;
            finish = ((utasks.isEmpty()&&(round>0)&&(round%3==0))||alive==false);
            }
      }
            //Update Variables
            restarted.clear();
            rest_ids.clear();
            round++;
            times_i=0;
            times_s=0;
```

```
		restart = false;
		cordinator = false;
		already=false;
		p.clear();

	}


	@Override
	public long getPeriod() {
		// we send messages every second
		return 20000;
	}


	@Override
	public boolean isActive() {
		/*
	 * Simulation for a Node runs if and only if
	 * there are tasks to be done.
	 */

	if(finish)   {
		System.out.println("Node" +id+" " +getNodeId()+" is Finished.");
		System.out.println("Total Messages Sent: " + total_msgs);
		System.out.println("Total Phases are: "+round/3+".");
		System.out.println("Total time: "+ (round/3)*getPeriod()+" ms.");
		getOut().println("Node" +id+" " +getNodeId()+" is Finished.");
		getOut().println("Total Messages Sent: " + total_msgs);
		getOut().println("Total Phases are: "+round/3+".");
		getOut().println("Total time: "+ (round/3)*getPeriod()+" ms.");
		System.exit(0);
			return false;
		}else
			return true;
		}

	}



	/**
	 * The set of nodes to which to send messages
	 */
	private Set<NodeID> nodes=new HashSet<NodeID>();

	/**
	 * The current sequence number for messages generated by this node.
	 */

	@Override
	public void startNode() {
		tm.registerPeriodicTask(new MessageGenerationTask(),
	SystemTaskGroup.getInstance(), 0);
	}
```

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

/*
 * Implementation of the receive method.
 * Here we divide the method to 4 cases
 * since we have 4 kinds of messages.
 * For each case we act accordingly
 */

E_NodeID snd = (E_NodeID) sender;
int snd_id = 0;

if (msg instanceof ReportMessage) {
       /*
 * Since we get this kind of a message
 * we know that we must act as cordinator.
 * We must fill the data with done work by adding it to the list d.
 * After this we must save the sender as alive.
 */
if(alive){

       cordinator=true;
       ReportMessage m=(ReportMessage)msg;
       snd_id = m.id;

       for(int i=0; i<m.pending.size(); i++){

              if(!donetasks.contains(m.pending.get(i)))
                     donetasks.add(m.pending.get(i));
       }

       //Variable Updates
       alive_ids.add(snd_id);
       p.add(snd);
       msg_count++;

       System.out.println("Cordinator Node"+id+" "+getNodeId() +" Just read a
       Report Message from Node"+snd_id+" "+snd);
       getOut().println("Cordinator Node"+id+" "+getNodeId() +" Just read a
       Report Message from Node"+snd_id+" "+snd);
       }

}else if (msg instanceof SummaryMessage){

/*
 * Since we get this kind of a message
 * we know that it's sender is a cordinator,
 * hence the phase will be marked as attended.
 * Also our local view level will set up to 0.
 */

already=true;
attended=true;
level=0;
```

```
//Get the message with done work and alive processors
SummaryMessage m =(SummaryMessage)msg;
snd_id = m.id;

// Update our knowledge according to message info.
// Updated knowledge will be for the Done Work and the Unaccound Tasks.
// Unaccound = Tasks - Done
for(int i=0; i<m.Dw.size(); i++){

	if(!donetasks.contains(m.Dw.get(i)))
		donetasks.add(m.Dw.get(i));

	if(utasks.contains(m.Dw.get(i)))
		utasks.remove(m.Dw.get(i));
}

//Check printings
System.out.println();
System.out.println("Done tasks " + donetasks.toString());
getOut().println();
getOut().println("Done tasks " + donetasks.toString());

//Initialize all processors as dead.
//At next mark alive ones and restarted ones
//true one by one.
for(int k=0; k<mark_alive.length; k++)
	mark_alive[k]=false;

times_s++;
if(times_s==1){
	// Empty our Local view if it's the 1st time
	// we get summary message so we copy the new ones
	aliveproc.clear();
}


//mark all alive and restarted processors at boolean mark_alive
for(int i=0; i<m.Pids.size(); i++){
	int node_pos = m.Pids.get(i) - 1 ;
	mark_alive [node_pos] = true;
}

for(int i=0; i<rest_ids.size(); i++){
	int node_pos = rest_ids.get(i)-1;
	mark_alive[node_pos]=true;
}

//after marking all alive and restarted we have to add them sorted
//to the aliveproc list with linear reading of all coordinators
int pos=0;
for (E_NodeID neighbor : DestinationList) {

	if(mark_alive[pos]==true){
		if(!aliveproc.contains(neighbor))
			aliveproc.add(neighbor);
```

```
        }
        pos++;
}

//check message
System.out.println("Processors in Local View: " + aliveproc.toString());
getOut().println("Processors in Local View: " + aliveproc.toString());


//delete the m msg
m.Pw.clear();
m.Pids.clear();

System.out.println("Node"+id+" "+getNodeId() +" Just read a Summary Message
from Cordinator Node"+snd_id+" "+snd);
getOut().println("Node"+id+" "+getNodeId() +" Just read a Summary Message from
Cordinator Node"+snd_id+" "+snd);

}else if (msg instanceof RestartMessage){

/*
 * Since we get this kind of a message
 * we know that it's sender is a restarted processor.
 * Hence we have to add him in the restarted list.
 * If and only if we are an alive processor
 */

//Get the sender id
RestartMessage m =(RestartMessage)msg;
snd_id = m.id;

if(sender.equals(getNodeId())){
        restarted.add(snd);
        rest_ids.add(snd_id);

}

if(alive){
        restarted.add(snd);
        rest_ids.add(snd_id);

        System.out.println("Node"+id+" "+getNodeId() +" Just read a Restart
        Message from Node"+snd_id+" "+snd);
        getOut().println("Node"+id+" "+getNodeId() +" Just read a Restart
Message
        from Node"+snd_id+" "+snd);
  }

}else if (msg instanceof InfoMessage){

/*
 * Since we get this kind of a message
 * we know that we are a restarted processor.
 * Hence we have to update our Local view and Unaccound tasks
 */
```

```
if(!already){

	times_i++;
	if(times_i==1){
		// Empty our Local view and Utasks
		// so we copy the new ones
		aliveproc.clear();
		utasks.clear();
	}

	// Get the msg data to handler m
	InfoMessage m = (InfoMessage) msg;
	snd_id= m.id;

	//Update Processors View and Unaccound Tasks
	for(int i=0; i<m.Lw.size(); i++){
		if(!aliveproc.contains(m.Lw.get(i)))
			aliveproc.add(m.Lw.get(i));
	}

	for(int i=0; i<m.Uw.size(); i++){
		if(!utasks.contains(m.Uw.get(i)))
			utasks.add(m.Uw.get(i));
	}

	times_i++;
}


alive=true;
System.out.println();
System.out.println("My Local view: " + aliveproc.toString());
System.out.println("Node"+id+" "+getNodeId() +" Just read an Information
Message from  Node"+snd_id+" "+snd);
getOut().println("Node"+id+" "+getNodeId() +" Just read an Information Message
from  Node"+snd_id+" "+snd);

}


/**
 * @param the name of the file from which to read the list of nodes
 * Get their id's sepperated by semicollum
 */
public void setNodeList(String nodeList) {
	String[] nodeStrings=nodeList.split(";");
	for (String node : nodeStrings) {
		try {
			nodes.add(cl.getNodeIdFromString(node));
		} catch (UnknownHostException e) {
			e.printStackTrace();
		}
	}
}

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();
		}
	}

	public void setInitialport(int port){
		System.out.println("Initial Port is: " + port);
		getOut().println("Initial Port is: " + port);
		init_port=port;
	}

	public void setTasks(int ta){
		System.out.println("Total num of tasks are: " + ta);
		getOut().println("Total num of tasks are: " + ta);
		no_tasks=ta; }

	public void setId(int n_id){
		System.out.println("My id is: " + n_id);
		getOut().println("My id is: " + n_id);
			id=n_id;}

	}
```

```java
/*
 * This is the class which implements the kind of
 * the message which will be sent from a Node to the
 * cordinator reporting to him that he finished a specific tasks.
 */

import java.io.Serializable;
import java.util.ArrayList;

class ReportMessage implements Serializable{

	private static final long serialVersionUID = 1L;

	// ArrayList with Pending Done Work
	ArrayList<String> pending = new ArrayList<String>();
	// Sender id
	public int id;

	@SuppressWarnings("unchecked")
	ReportMessage (ArrayList<String> value, int id){
		this.pending = (ArrayList<String>) value.clone();
		this.id = id;
	}
}

/*
 * This is the class which implements the kind of
 * the message which will be sent from all processors
 * to a restarted Node to inform him for the Unaccound Tasks
 * and the Local view.
 */

import java.io.Serializable;
import java.util.ArrayList;
import yalps.executor.E_NodeID;

class InfoMessage implements Serializable{

	private static final long serialVersionUID = 1L;

	// ArrayList with Unaccound tasks
	ArrayList<String> Uw = new ArrayList<String>();
	// ArrayList with all alive processors - Local View
	ArrayList<E_NodeID> Lw = new ArrayList<E_NodeID>();
	// Sender id
	public int id;

	@SuppressWarnings("unchecked")
	InfoMessage (ArrayList<String> unaccound, ArrayList<E_NodeID> local
	,int id ){
		this.Uw =(ArrayList<String>) unaccound.clone();
		this.Lw = (ArrayList<E_NodeID>) local.clone();
		this.id=id;
	}
}
```

```java
/*
 * This is the class which implements the kind of
 * the message which will be sent from a restarted Node to
 * all nodes so they be informed that he is alive again.
 */

import java.io.Serializable;


class RestartMessage implements Serializable{

	private static final long serialVersionUID = 1L;

	// Sender id
	public int id;

	RestartMessage (int id){
		this.id = id;
	}
}

/*

 * This is the class which implements the kind of
 * the message which will be sent from a Cordinator as Summary
 * to a node reporting to him all done work until now and
 * all alive processors
 */

import java.io.Serializable;
import java.util.ArrayList;
import yalps.executor.E_NodeID;

class SummaryMessage implements Serializable{

	private static final long serialVersionUID = 1L;

	// ArrayList with Done Work
	ArrayList<String> Dw = new ArrayList<String>();
	// ArrayList with Alive Processors Ids
	ArrayList<E_NodeID> Pw = new ArrayList<E_NodeID>();
	// ArrayList with Alive Processors Ids
	ArrayList<Integer> Pids = new ArrayList<Integer>();
	// Sender id
	public int id;


	@SuppressWarnings("unchecked")
	SummaryMessage (ArrayList<String> value , ArrayList<E_NodeID> val,
ArrayList<Integer> v ,int id){
		this.Dw = (ArrayList<String>) value.clone();
		this.Pw = (ArrayList<E_NodeID>) val.clone();
		this.Pids =(ArrayList<Integer>) v.clone();
		this.id = id;
	}
```

```
}

// Class for the communication
import java.io.IOException;
import yalps.NodeID;
import yalps.Task;
import yalps.communication.CommunicationLayer;
import yalps.executor.E_NodeID;
import yalps.executor.RealWorldCommunicationLayer;

public class MyTCPBasedMsgSenderTask implements Task {

	E_NodeID destination;
	RealWorldCommunicationLayer commLayer;
	ReportMessage ReportMsg = null;
	SummaryMessage SummaryMsg = null;
	InfoMessage InfoMsg = null;
	RestartMessage RestartMsg= null;
	MyHeadedCheapSerializable headedCheapMsg = null;
	long taskPeriod = 10000; // Runs each taskPeriod miliseconds 10000

	public MyTCPBasedMsgSenderTask(ReportMessage msg1, NodeID dest,
CommunicationLayer comm){
		this.ReportMsg = msg1;
		this.destination = (E_NodeID) dest;
		this.commLayer = (RealWorldCommunicationLayer) comm;
	}
	public MyTCPBasedMsgSenderTask(SummaryMessage msg1, NodeID dest,
CommunicationLayer comm){
		this.SummaryMsg = msg1;
		this.destination = (E_NodeID) dest;
		this.commLayer = (RealWorldCommunicationLayer) comm;
	}
	public MyTCPBasedMsgSenderTask(InfoMessage msg1, NodeID dest,
CommunicationLayer comm){
		this.InfoMsg = msg1;
		this.destination = (E_NodeID) dest;
		this.commLayer = (RealWorldCommunicationLayer) comm;
	}
	public MyTCPBasedMsgSenderTask(RestartMessage msg1, NodeID dest,
CommunicationLayer comm){
		this.RestartMsg = msg1;
		this.destination = (E_NodeID) dest;
		this.commLayer = (RealWorldCommunicationLayer) comm; }


	@Override
	public void run() {

		if(ReportMsg!=null){   // msg to send is Serializable

			try {
				System.out.println("[Application] - is sending a
Report Mesage SERIALIZABLE msg with Regular TCP  to [" + destination +"]");
				commLayer.sendTCP(ReportMsg, destination);
```

```
                        } catch (IOException e) {
                                e.printStackTrace();
                        }
                }

                if(SummaryMsg!=null){   // msg to send is Serializable

                        try {
                                System.out.println("[Application] - is sending a
Summary Mesage SERIALIZABLE msg with Regular TCP  to [" + destination +"]");
                                commLayer.sendTCP(SummaryMsg, destination);
                        } catch (IOException e) {
                                e.printStackTrace();
                        }
                }

                if(InfoMsg!=null){   // msg to send is Serializable

                        try {
                                System.out.println("[Application] - is sending an
Information Mesage SERIALIZABLE msg with Regular TCP  to [" + destination
+"]");
                                commLayer.sendTCP(InfoMsg, destination);
                        } catch (IOException e) {
                                e.printStackTrace();
                        }
                }

                if(RestartMsg!=null){   // msg to send is Serializable

                        try {
                                System.out.println("[Application] - is sending a
Restart Mesage SERIALIZABLE msg with Regular TCP  to [" + destination +"]");
                                commLayer.sendTCP(RestartMsg, destination);
                        } catch (IOException e) {
                                e.printStackTrace();
                        }
                }

        public void setPeriod(int period){
                this.taskPeriod = period;
        }
}
```

Παράρτημα Γ

Κώδικας Αλγορίθμου Do-All σε γλώσσα προγραμματισμού JAVA

```
import java.net.UnknownHostException;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.Set;
import java.util.concurrent.TimeUnit;
import yalps.BasicPeriodicTask;
import yalps.NodeID;
import yalps.TaskManager;
import yalps.executor.E_NodeID;
import yalps.launchers.AbstractYalpsNode;
import yalps.taskgroup.SystemTaskGroup;


public class BroadcastApplicationWithSerializable extends AbstractYalpsNode{


/* Variable Declarations
 * String job          -> Keeps the name of the job we done
 * int total_msgs      -> Keeps the total number of sent messages
 * int init_port       -> Keeps the initial number of the first port
 * int id              -> Keeps the id of the processor
 * int rank            -> Keeps the rank for the Balance rule
 * int round           -> Keeps the current round id
 * int no_tasks        -> Keeps the total number of tasks to be done
 * int level           -> Keeps the level of the phase
 * boolean alive       -> Declares if as a processor i am alive.
 * boolean cordinator  -> Declares if a processor it's set as cordinator
 * boolean c_cord      -> Declares if a processor consider it self as coord
 * boolean attended    -> Declares if a phase was attended or not
 * bolean[] mark_alive -> Declares if a specific processor is alive or not.
 * ArrayList tasks     -> ArrayList which keeps the set of tasks to be done
 * ArrayList aliveproc -> ArrayList which keeps the set of alive processors
 * ArrayList tempalive -> ArrayList which keeps tmp the set of alive proc's
 * ArrayList failedproc-> ArrayList which keeps the set of crushed processors
 * ArrayList Cordinators -> ArrayList keeps the set of last active coords
 * ArrayList processors -> ArrayList which keeps the set of all processors
 * ArrayList donetasks  -> ArrayList which keeps the set of completed tasks
 * ArrayList utasks     -> ArrayList which keeps the set of unaccound tasks
 * ArrayList DestinationList -> ArrayList which keeps all the execution proc's
 */

public String job=null;
public int all_nodes=0;
public int l_kill =0;
public int temp_all_nodes=0;
```

```
public int times_kill=0;
public int total_msgs=0;
public int init_port=0;
public int id;
public int rank=0;
public int round=0;
public int no_tasks=0;
public int level=0;
public boolean alive = true;
public boolean cordinator = false;
public boolean c_cord = false;
public boolean attended = false;
ArrayList<E_NodeID> aliveproc = new ArrayList<E_NodeID>();
ArrayList<E_NodeID> tempalive = new ArrayList<E_NodeID>();
ArrayList<E_NodeID> failedproc = new ArrayList<E_NodeID>();
ArrayList<E_NodeID> Cordinators = new ArrayList<E_NodeID>();
ArrayList<E_NodeID> processors = new ArrayList<E_NodeID>();
ArrayList<String> tasks = new ArrayList<String>();
ArrayList<String> donetasks = new ArrayList<String>();
ArrayList<String> utasks = new ArrayList<String>();
protected ArrayList<E_NodeID> DestinationList = new ArrayList<E_NodeID>();


class MessageGenerationTask extends BasicPeriodicTask {


@Override
protected TaskManager getTaskManager() {
	return BroadcastApplicationWithSerializable.this.getTaskManager();
}


@Override
protected void periodicRun() {

/*
 * If it's the first round then:
 * Initialize the Done tasks,
 * tasks, Local view
 */


if (round == 0) {

	// Initialiaze all known processors
	for (E_NodeID neighbor : DestinationList) {
		aliveproc.add(neighbor);
		processors.add(neighbor);
	}


	// Initialiaze all tasks and donetasks
	for (int i = 1; i <= no_tasks; i++) {
		tasks.add("task" + i);
	}
	donetasks.clear();
```

```
} else {

	/* Start the implementation of algorithm Do - UM.
	 * We suppose that 3 rounds of a yalps task will represent 1 phase
	 * of our algorithm. The 1st round of yalps represents the Collect Round
	 * and the 2nd - 3rd round of yalps together they
	 * represent the Disseminate Round.
	 * This is made up to help us with sync.
	 */


	/*
	 * This if block is the 1st round of our algorithm (Collect Round) where
	 * An alive node will create a report message and send this message to
	 * all coordinators corresponding to it's local view.
	 * Receive report message and Compute the new Done work
	 * and Crashed processors.
	 */



	if((round%3 ==1)&&alive){

		// Create and send message to all coordinators
		ReportMessage ms = new ReportMessage(donetasks,failedproc,id);

		for(int i=(int)(Math.pow(2, level)-1); i<(Math.pow(2, level+1)-
		1) && i<aliveproc.size(); i++){

			System.out.println("Node"+id+" "+getNodeId() + " sending
			report to cordinator: " + aliveproc.get(i));
			getOut().println("Node"+id+" "+getNodeId() + " sending
			report to cordinator: " + aliveproc.get(i));
			MyTCPBasedMsgSenderTask myTask1 = new
			MyTCPBasedMsgSenderTask(ms, aliveproc.get(i), cl);
			tm.registerTask(myTask1,  tm.getSystemTaskGroup(), 0);
			total_msgs++;
		}

	}else if((round%3 == 2)&&alive){

		/*
		 * This if block is the 2nd stage of our algorithm
		 * and the 1st phase of Disseminate Round.
		 * A cordinator will Create Summary message with done work
		 * and crushed processors and then send it to all alive nodes.
		 */

		//cordinator is set as true if he received any report message or
		//his id is in the current level of coordinators.
		for(int i=0; i<Math.pow(2, level) && i<aliveproc.size(); i++){
			if(aliveproc.get(i).equals(getNodeId())){
				c_cord=true;
```

```
                        break;
                }
        }

        //Coordinator by myself or because we receive Report Messages
        if(cordinator||c_cord){

        //create messages with Done work and crushed processors
        SummaryMessage m3 = new SummaryMessage(donetasks,failedproc,id);

                //Send the Summary messages to all alive nodes P-F
                //Compute P-F on tempalive array.
                //clear before fill it
                tempalive.clear();

                for(int i=0; i<processors.size(); i++){
                        if(!failedproc.contains(processors.get(i)))
                                tempalive.add(processors.get(i));
                }

                //Now we have computed the P-F set we have
                //to broadcast the summary msg to them.

                for(int i=0; i<tempalive.size(); i++){

                        getOut().println("Cordinator Node"+id+"
                        "+getNodeId() + " sending Summary msg to: " +
                        tempalive.get(i));
                        MyTCPBasedMsgSenderTask myTask1 = new
                        MyTCPBasedMsgSenderTask(m3, tempalive.get(i), cl);
                        tm.registerTask(myTask1,tm.getSystemTaskGroup(), 0);
                        total_msgs++;
                }

                //Clear the P-F set and set the cordinator
                //boolean values equal to false.
                tempalive.clear();
                c_cord=false;
                cordinator=false;
        }

}else if ((round%3==0)&&alive){

        /*
         * This if block is the 2nd phase of our Second round
         * Compute stage. We are going to fill the Failed procs,
         * check if the phase was attended or not.
         * If it was unattended means we did not get any messages
         * from any cordinator so we have to update our level
         * If it was attended we set the level again as 0
         * and compute the new Local view.
         */

        // Failed processors are all that are in failed
        // plus the coordinators of this round which are not in
        // the last active Coordinators list.
```

```
for(int i=0; i<Math.pow(2, level) && i<aliveproc.size(); i++){
	if(!Cordinators.contains(aliveproc.get(i))){
		if(!failedproc.contains(aliveproc.get(i)))
			failedproc.add(aliveproc.get(i));
	}
}

//Check Prints
System.out.println();
System.out.println("Cordinators: "+ Cordinators.toString());
System.out.println("Failed processors: "+ failedproc.toString());
getOut().println();
getOut().println("Cordinators: "+ Cordinators.toString());
getOut().println("Failed processors: "+ failedproc.toString());


if(!Cordinators.isEmpty()){

	//Coordinators attended, mark as attended phase
	attended = true;

	//clear before we compute the alive
	aliveproc.clear();

	for(int i=0; i<processors.size(); i++){
		if(!failedproc.contains(processors.get(i)))
			aliveproc.add(processors.get(i));
	}

	//set the level back to zero
	level=0;

}else{

	//No cordinatoor attended
	//Only compute the P - F (Processors - Crushed processors)
	//but not set it to alive processors

	//clear before fill it
	tempalive.clear();

	for(int i=0; i<processors.size(); i++){
		if(!failedproc.contains(processors.get(i)))
			tempalive.add(processors.get(i));
	}

	//Increase the level
	level++;
}


//Must take the job with Balance Rule
//First we must get the T-D (tasks - done tasks)
for(int i=0; i<tasks.size(); i++){
	if(!donetasks.contains(tasks.get(i)))
```

```
                        utasks.add(tasks.get(i));
                }


                //If it was unattended phase we are going
                //to get the P-F rank from the temporary processors
                //else use it from the alive processors
                if(!attended){

                        //Get our rank from the temp set.
                        for(int i=0; i<tempalive.size(); i++){
                                if(tempalive.get(i).equals(getNodeId())){
                                        rank = i;
                                        break;
                                }
                        }

                }else{

                        //Attended Get our rank from the alive set.
                        for(int i=0; i<aliveproc.size(); i++){
                                if(aliveproc.get(i).equals(getNodeId())){
                                        rank = i;
                                        break;
                                }
                        }
                }

                //Execute the rank mod utasks size task
                job = utasks.get(rank % utasks.size());
                donetasks.add(job);

                //Check Prints
                System.out.println(getNodeId() + " get the " + job + " .");
                System.out.println("Utasks "+ utasks.toString());
                System.out.println("Total Messages Sent "+ total_msgs);
                System.out.println();
                getOut().println("getNodeId() + " get the " + job + " .");
                getOut().println("Utasks "+ utasks.toString());
                getOut().println("Total Messages Sent "+ total_msgs);
                getOut().println();

                //Set new values to variables
                Cordinators.clear();
                utasks.clear();
                attended=false;
        }

}


// We increase the round counter
        round++;
}
```

```
@Override
public long getPeriod() {
	// we send messages every second
	return 20000;
}

@Override
public boolean isActive() {

/*
 * Simulation for a Node runs if and only if
 * he is alive and there are tasks to be done.
 */

if((((donetasks.size()==tasks.size())&&(round>0))&&(round%3==0))||alive==fals
e){
	System.out.println("Node" +id+" " +getNodeId()+" is Finished.");
	System.out.println("Total Messages Sent: " + total_msgs);
	System.out.println("Total Phases are: "+round/3+".");
	System.out.println("Total time: "+ (round/3)*getPeriod()+" ms.");
	getOut().println("Node" +id+" " +getNodeId()+" is Finished.");
	getOut().println("Total Messages Sent: " + total_msgs);
	getOut().println("Total Phases are: "+round/3+".");
	getOut().println("Total time: "+ (round/3)*getPeriod()+" ms.");
	System.exit(0);

		return false;
	}else
		return true;
   }

}


/**
 * The set of nodes to which to send messages
 */
private Set<NodeID> nodes=new HashSet<NodeID>();
/**
 * The current sequence number for messages generated by this node.
 */

@Override
public void startNode() {
	tm.registerPeriodicTask(new MessageGenerationTask(),
SystemTaskGroup.getInstance(), 0);
}


@Override
public boolean receive(short header, Object msg, NodeID sender) {

/*
 * Implementation of the receive method.
 * Here we divide the method to 2 cases
 * since we have 2 kinds of messages.
```

```
 * For each case we act accordingly
 */

E_NodeID snd = (E_NodeID) sender;
int snd_id = 0;


if (msg instanceof ReportMessage) {
/*
 * Since we get this kind of a message
 * we know that we must act as cordinator.
 * We must fill the data with done work by adding it to the list d.
 * and update the crushed processors
 * After this we must save the sender as alive.
 */

if(alive){

	cordinator=true;
	ReportMessage m=(ReportMessage)msg;
	snd_id = m.id;

	for(int i=0; i<m.DoneWork.size(); i++){

		if(!donetasks.contains(m.DoneWork.get(i)))
			donetasks.add(m.DoneWork.get(i));
	}

	for(int i=0; i<m.Failured.size(); i++){

		if(!failedproc.contains(m.Failured.get(i)))
			failedproc.add(m.Failured.get(i));
	}

	System.out.println("Cordinator Node"+id+" "+getNodeId() +" Just read a
	Report Message from Node"+snd_id+" "+snd);
	getOut().println("Cordinator Node"+id+" "+getNodeId() +" Just read a
	Report Message from Node"+snd_id+" "+snd);
  }

}else if (msg instanceof SummaryMessage){


/*
 * Since we get this kind of a message
 * we know that it's sender is a cordinator,
 * put him in the Last active coordinators list
 * and get the Done work and Failed processors
 */

if(alive){

	Cordinators.add(snd);

	//Get the message with done work and alive processors
	SummaryMessage m =(SummaryMessage)msg;
```

```
		snd_id = m.id;

		// Compute stage
		// Update our knowledge according to message info.
		// Updated knowledge will be for the Done Work, Unaccound Tasks.
		// Failed processors

		for(int i=0; i<m.DoneWork.size(); i++){

			if(!donetasks.contains(m.DoneWork.get(i)))
				donetasks.add(m.DoneWork.get(i));
		}

		for(int i=0; i<m.Failured.size(); i++){

			if(!failedproc.contains(m.Failured.get(i)))
				failedproc.add(m.Failured.get(i));
		}

		//Check printings
		getOut().println();
		getOut().println("getNodeId() +"Done tasks " + donetasks.toString());
		getOut().println("Node"+id+" "+getNodeId() +" Just read a Summary
Message
		from Cordinator Node"+snd_id+" "+snd);
		}
	}
		return true;
}


/**
 * @param the name of the file from which to read the list of nodes
 * Get their id's sepperated by semicollum
 */
public void setNodeList(String nodeList) {
		String[] nodeStrings=nodeList.split(";");
		for (String node : nodeStrings) {
			try {
				nodes.add(cl.getNodeIdFromString(node));
			} catch (UnknownHostException e) {
				e.printStackTrace();
			}
		}
}
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();
		}
```

```
}

public void setInitialport(int port){
	System.out.println("Initial Port is: " + port);
getOut().println("Initial Port is: " + port);
	init_port=port;
}

public void setTasks(int ta){
	System.out.println("Total num of tasks are: " + ta);
getOut().println("Total num of tasks are: " + ta);
	no_tasks=ta;
}

public void setId(int n_id){
	System.out.println("My id is: " + n_id);
getOut().println("My id is: " + n_id);
		id=n_id;
	}
}
```

```java
/*
 * This is the class which implements the kind of
 * the message which will be sent from a Cordinator as Summary
 * to a node reporting to him all done work until now and
 * all known failed processors
 */

import java.io.Serializable;
import java.util.ArrayList;
import yalps.executor.E_NodeID;

class SummaryMessage implements Serializable{

	private static final long serialVersionUID = 1L;

	// ArrayList with Done Work
	ArrayList<String> DoneWork = new ArrayList<String>();
	// ArrayList with Crushed Processors Ids
	ArrayList<E_NodeID> Failured = new ArrayList<E_NodeID>();
	// Sender id
	public int id;


	@SuppressWarnings("unchecked")
	SummaryMessage (ArrayList<String> value , ArrayList<E_NodeID> val, int
id){
		this.DoneWork = (ArrayList<String>) value.clone();
		this.Failured = (ArrayList<E_NodeID>) val.clone();
		this.id=id;
	}

}
```

```java
/*
 * This is the class which implements the kind of
 * the message which will be sent from a Node to the
 * cordinator reporting to him that he finished a specific tasks.
 * Report Message contains the Done Work and all the crushed
 * processors that the Node knows.
 */

import java.io.Serializable;
import java.util.ArrayList;
import yalps.executor.E_NodeID;

class ReportMessage implements Serializable{

	private static final long serialVersionUID = 1L;

	// ArrayList with Done Work and Crushed processors
	ArrayList<String> DoneWork = new ArrayList<String>();
	ArrayList<E_NodeID> Failured = new ArrayList<E_NodeID>();
	// Sender id
	public int id;

	@SuppressWarnings("unchecked")
	ReportMessage (ArrayList<String> value , ArrayList<E_NodeID> val , int
	id){
		this.DoneWork = (ArrayList<String>) value.clone();
		this.Failured = (ArrayList<E_NodeID>) val.clone();
		this.id=id;
	}
}
```

```java
// This class implements the communication of a sent TcpMessage
import java.io.IOException;
import yalps.NodeID;
import yalps.Task;
import yalps.communication.CommunicationLayer;
import yalps.executor.E_NodeID;
import yalps.executor.RealWorldCommunicationLayer;


public class MyTCPBasedMsgSenderTask implements Task {

	E_NodeID destination;
	RealWorldCommunicationLayer commLayer;
	ReportMessage ReportMsg = null;
	SummaryMessage SummaryMsg = null;
	MyHeadedCheapSerializable headedCheapMsg = null;
	long taskPeriod = 10000; // Runs each taskPeriod miliseconds 10000

	public MyTCPBasedMsgSenderTask(ReportMessage msg1, NodeID dest,
CommunicationLayer comm){
		this.ReportMsg = msg1;
		this.destination = (E_NodeID) dest;
		this.commLayer = (RealWorldCommunicationLayer) comm;
	}
	public MyTCPBasedMsgSenderTask(SummaryMessage msg1, NodeID dest,
CommunicationLayer comm){
		this.SummaryMsg = msg1;
		this.destination = (E_NodeID) dest;
		this.commLayer = (RealWorldCommunicationLayer) comm;
	}

	@Override
	public void run() {

		if(ReportMsg!=null){   // msg to send is Serializable
			try {
				System.out.println("[Application] - is sending a
				Periodic Report Mesage SERIALIZABLE msg with Regular
				TCP  to [" + destination +"]");
				commLayer.sendTCP(ReportMsg, destination);
			} catch (IOException e) {
				e.printStackTrace();
			}
		}
		if(SummaryMsg!=null){   // msg to send is Serializable
			try {
				System.out.println("[Application] - is sending a
				Periodic Summary Mesage SERIALIZABLE msg with
				Regular TCP  to [" + destination +"]");
				commLayer.sendTCP(SummaryMsg, destination);
			} catch (IOException e) {
				e.printStackTrace();
			}
		}
	}
```

Παράρτημα Δ

Παραδείγματα Εκτέλεσης Αλγορίθμων AN, AR και Do-UM

Παράδειγμα Αρχείου Διαμόρφωσης για 8 κόμβους.

```
node.CLASS=BroadcastApplicationWithSerializable
node.outFileName=outfile_1.log
node.rnd="node"
node.nylon=false
node.initialport=50000
node.tasks=8
node.id=1

node.destination=193.144.21.130:50000;193.144.21.131:50000;130.37.193.143:50000;193.55.112.40:5
0000;193.55.112.41:50000;131.188.44.101:50000;131.188.44.102:50000;138.251.214.77:50000;

node.nodeList=193.144.21.130:50000;193.144.21.131:50000;130.37.193.143:50000;193.55.112.40:500
00;193.55.112.41:50000;131.188.44.101:50000;131.188.44.102:50000;138.251.214.77:50000;

executor.cheapDeserializationMap=MyDeserializationMap

executor.virtualAddress=193.144.21.130
executor.port=50000

executor.debug=true
executor.tokenbucket=true
executor.burstSize=1000000
executor.rnd='comm'
executor.lossRate=0
executor.maxQueueLength=10000000
executor.maxSendDelay=0
executor.uploadBandwidth=0
```

Παράδειγμα Εκτέλεσης Αλγόριθμου .

Initial Port is: 50000
Total num of tasks are: 8
My id is: 1
I am [193.144.21.130,50000] and my node id: 1
*** [193.144.21.130,50000] am Cordinator ***

Initial AliveProccessors are: [[193.144.21.130,50000], [193.144.21.131,50000], [130.37.193.143,50000], [193.55.112.40,50000], [193.55.112.41,50000], [131.188.44.101,50000], [131.188.44.102,50000], [138.251.214.77,50000]]

Node1 My position is: 0
Node1 [193.144.21.130,50000] get the task1 .
Node1 [193.144.21.130,50000] sending report to cordinator: [193.144.21.130,50000]
[RECEIVED] 1395363188640 [193.144.21.130,50000] class ReportMessage [130.37.193.143,50000]
[RECEIVED] 1395363188640 [193.144.21.130,50000] class ReportMessage [193.144.21.130,50000]
Cordinator Node1 [193.144.21.130,50000] Just read a Report Message from Node3 [130.37.193.143,50000]
Cordinator Node1 [193.144.21.130,50000] Just read a Report Message from Node1 [193.144.21.130,50000]
[RECEIVED] 1395363188754 [193.144.21.130,50000] class ReportMessage [193.55.112.40,50000]
Cordinator Node1 [193.144.21.130,50000] Just read a Report Message from Node4 [193.55.112.40,50000]
[RECEIVED] 1395363188854 [193.144.21.130,50000] class ReportMessage [193.55.112.41,50000]
Cordinator Node1 [193.144.21.130,50000] Just read a Report Message from Node5 [193.55.112.41,50000]
[RECEIVED] 1395363188954 [193.144.21.130,50000] class ReportMessage [193.144.21.131,50000]
Cordinator Node1 [193.144.21.130,50000] Just read a Report Message from Node2 [193.144.21.131,50000]
[RECEIVED] 1395363189055 [193.144.21.130,50000] class ReportMessage [131.188.44.102,50000]
Cordinator Node1 [193.144.21.130,50000] Just read a Report Message from Node7 [131.188.44.102,50000]
[RECEIVED] 1395363189657 [193.144.21.130,50000] class ReportMessage [138.251.214.77,50000]
Cordinator Node1 [193.144.21.130,50000] Just read a Report Message from Node8 [138.251.214.77,50000]
[RECEIVED] 1395363191060 [193.144.21.130,50000] class ReportMessage [131.188.44.101,50000]
Cordinator Node1 [193.144.21.130,50000] Just read a Report Message from Node6 [131.188.44.101,50000]
[RECEIVED] 1395363199075 [193.144.21.130,50000] class SummaryMessage [193.144.21.130,50000]

Done tasks [task3, task1, task4, task5, task2, task7, task8, task6]
Alive processors: [[193.144.21.130,50000], [193.144.21.131,50000], [130.37.193.143,50000], [193.55.112.40,50000], [193.55.112.41,50000], [131.188.44.101,50000], [131.188.44.102,50000], [138.251.214.77,50000]]
Node1 [193.144.21.130,50000] Just read a Summary Message from Cordinator Node1 [193.144.21.130,50000]

Node1 [193.144.21.130,50000] is Finished. Total Messages Sent: 9
Node1 [193.144.21.130,50000] Total Phases are: 1. Total time is: 10000 ms.