Ατομική Διπλωματική Εργασία

ΥΛΟΠΟΙΗΣΗ ΚΑΙ ΠΕΙΡΑΜΑΤΙΚΗ ΑΞΙΟΛΟΓΗΣΗ ΚΑΤΑΝΕΜΗΜΕΝΩΝ ΑΛΓΟΡΙΘΜΩΝ ΠΛΗΡΟΦΟΡΗΣΗΣ ΜΕ ΤΗ ΧΡΗΣΗ ΤΩΝ ΒΙΒΛΙΟΘΗΚΩΝ GossipLib ΚΑΙ YALPS

Χρυστάλλα Σοφοκλέους

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάιος 2011

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΥΛΟΠΟΙΗΣΗ ΚΑΙ ΠΕΙΡΑΜΑΤΙΚΗ ΑΞΙΟΛΟΓΗΣΗ ΚΑΤΑΝΕΜΗΜΕΝΩΝ ΑΛΓΟΡΙΘΜΩΝ ΠΛΗΡΟΦΟΡΗΣΗΣ ΜΕ ΤΗ ΧΡΗΣΗ ΤΩΝ ΒΙΒΛΙΟΘΗΚΩΝ GossipLib ΚΑΙ YALPS

Χρυστάλλα Σοφοκλέους

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 2011

Ευχαριστίες

Στο σημείο αυτό θα ήθελα να ευχαριστήσω τον επιβλέπων καθηγητή Δρ. Χρύση Γεωργίου, Επίκουρο καθηγητή του τμήματος Πληροφορικής του Πανεπιστημίου Κύπρου, ο οποίος μου έδωσε την ευκαιρία να εργαστώ στην παρούσα 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 στήριξη.

Περίληψη

Οι αλγόριθμοι μετάδοσης πληροφοριών σε κατανεμημένα συστήματα είναι ένα πρόβλημα με μεγάλες διαστάσεις στις μέρες μας λόγω της ανάπτυξης των κατανεμημένων συστημάτων και της ανάγκης για ενημέρωση των κόμβων των συστημάτων αυτών. Στην εργασία αυτή μελετήθηκαν τρεις αλγόριθμοι πληροφόρησης οι οποίοι έχουν αξιολογηθεί θεωρητικά και στόχος της εργασίας είναι να επιβεβαιωθεί και πρακτικά η αξιολόγησή τους. Οι αλγόριθμοι αυτοί είναι ο αλγόριθμος PUSH, ο αλγόριθμος PUSH&PULL και ο αλγόριθμος MEDIAN-COUNTER. Ο τελευταίος αλγόριθμος μελετήθηκε και στην περίπτωση που το περιβάλλον στο οποίο τρέχει υπόκειται σε σφάλματα δικτύου.

Αφού μελετήθηκαν εις βάθος οι αλγόριθμοι έγινε η εκμάθηση δύο βιβλιοθηκών, GossipLib και YALPS, οι οποίες χρησιμοποιήθηκαν για την υλοποίηση των αλγορίθμων στη γλώσσα προγραμματισμού JAVA. Στη συνέχεια έγινε η υλοποίηση των αλγορίθμων και η προσομοίωση τους με σκοπό να καταμετρηθούν οι επιδόσεις τους. Χρησιμοποιώντας τις μετρήσεις αυτές δημιουργήθηκαν οι αντίστοιχες γραφικές παραστάσεις οι οποίες βοήθησαν στην αξιολόγηση του κάθε αλγορίθμου ξεχωριστά. Η εμπειρική αξιολόγηση των αλγορίθμων οδήγησε στο συμπέρασμα ότι η πειραματική τους απόδοση συναύει με τη θεωρητική αξιολόγησή τους. Επιπλέον, από τη μεταξύ τους σύγκριση καταλήξαμε στο συμπέρασμα ότι και οι τρεις έχουν παρόμοια επίδοση με βασική διαφορά την ευρωστία του αλγορίθμου MEDIAN-COUNTER.

Περιεχόμενα

ΚΕΦΑΛΑΙΟ 1

Εισαγωγή

1.1 Κίνητρο διπλωματικής εργασίας

Η μετάδοση πληροφοριών στα σύγχρονα κατανεμημένα συστήματα αποτελεί σημαντική παράμετρο στην αποδοτικότητά τους. Μία κλάση αλγορίθμων πληροφόρησης, οι επιδημικοί αλγόριθμοι [1], χρησιμοποιούνται συχνά για την αποτελεσματική μετάδοση ενημερώσεων ή άλλων πληροφοριών σε κατανεμημένα περιβάλλοντα. Οι αλγόριθμοι αυτοί χρησιμοποιούν ένα απλό τυχαιοποιημένο μηχανισμό επικοινωνίας, κατά τον οποίο σε κάθε γύρο επικοινωνίας, οι κόμβοι του δικτύου επιλέγουν τυχαία και ομοιόμορφα ένα γείτονά τους με τον οποίο θα ανταλλάξουν πληροφορίες. Ο μηχανισμός αυτός, παρά την απλότητά του, καθιστά τον αλγόριθμο επεκτάσιμο και παρέχει υψηλή ανεκτικότητα σε σφάλματα δικτύου. Γι' αυτό το λόγο οι επιδημικοί αλγόριθμοι είναι μέγιστης σημασίας.

Το σημείο που διαφοροποιεί τους επιδημικούς αλγορίθμους μεταξύ τους είναι ο τρόπος με τον οποίο οι κόμβοι επιλέγουν τις πληροφορίες που θα ανταλλάξουν με τον γείτονα που επέλεξαν. Αυτό το σημείο είναι το βασικό πρόβλημα αυτής της κλάσης αλγορίθμων γιατί, λόγω της τυχαιοποιημένης επικοινωνίας, οι κόμβοι του δικτύου συχνά δεν γνωρίζουν ποιές πληροφορίες έχει ήδη παραλάβει ο γείτονάς τους και ποιες όχι και αυτό μπορεί να οδηγήσει σε άσκοπη μετάδοση πληροφοριών.

Έρευνες που έγιναν σε επιδημικούς αλγόριθμους [1], απέδειξαν ότι το ποσοστό των αχρείαστων πληροφοριών που μεταδίδονται μπορεί να μειωθεί σημαντικά χωρίς να επηρεάζεται η ευρωστία των αλγορίθμων. Συγκεκριμένα, σχεδιάστηκαν εύρωστοι αλγόριθμοι οι οποίοι ενημερώνουν πλήρως όλους τους κόμβους του δικτύου σε $O(\ln n)$

γύρους επικοινωνίας και με $O(n \ln \ln n)$ μεταδόσεις, όπου το n είναι ο αριθμός των κόμβων. Εύρωστο αλγόριθμο ονομάζουμε τον αλγόριθμο ο οποίος είναι αποδοτικός και ταυτόχρονα είναι ανεκτικός σε σφάλματα. Από την άλλη όμως πλευρά αποδείχτηκε ότι οι αλγόριθμοι που χρησιμοποιούν μόνο την τοπική τους γνώση για να παίρνουν αποφάσεις χρειάζεται να στείλουν $\Omega(n \ln \ln n)$ μηνύματα για την κάθε πληροφορία, ανεξαρτήτως του αριθμού των γύρων επικοινωνίας που θα χρειαστεί να τρέξει ο αλγόριθμος.

Αφού λοιπόν υπάρχουν επιδημικοί αλγόριθμοι οι οποίοι θεωρητικά είναι εύρωστοι και ολοκληρώνουν την ενημέρωση του δικτύου σε $O(\ln n)$ γύρους επικοινωνίας και με $O(n \ln \ln n)$ μεταδόσεις, μπορούμε να έχουμε αποδοτικά κατανεμημένα συστήματα στα οποία όλοι οι κόμβοι θα είναι ενημερωμένοι και συνεπείς ανά πάσα στιγμή. Οι αλγόριθμοι αυτοί όμως, δεν έχουν δοκιμαστεί σε πρακτικό επίπεδο, και αυτό είναι το κίνητρο που οδήγησε στην εργασία αυτή.

1.2 Στόχος διπλωματικής εργασίας

Η παρούσα διπλωματική εργασία στοχεύει στην πειραματική αξιολόγηση αλγορίθμων πληροφόρησης η οποία θα υποδείξει σε ποιο βαθμό, η θεωρητική πολυπλοκότητα των αλγορίθμων αυτών συναύει με την πρακτική αξιολόγησή τους. Συγκεκριμένα υλοποιούνται και αξιολογούνται τρεις αλγόριθμοι πληροφόρησης κάθε ένας από τους οποίους χρησιμοποιεί διαφορετική μέθοδο για τερματισμό και διαφορετική μέθοδο επιλογής των πληροφοριών που θα μεταδοθούν. Και στους τρεις αλγόριθμους, ο κάθε κόμβος στο δίκτυο έχει στατική πληροφορία η οποία πρέπει με το τέλος του αλγορίθμου να έχει μεταδοθεί σε όλους τους υπόλοιπους κόμβους του δικτύου. Οπότε, αφού υπάρχουν n κόμβοι, υπάρχουν και n πληροφορίες για να μεταδοθούν.

Ο πρώτος αλγόριθμος που υλοποιείται, τον οποίο καλούμε PUSH, δεν χρησιμοποιεί καμία μέθοδο για επιλογή της πληροφορίας που θα σταλεί. Αντί αυτού, σε κάθε γύρο επικοινωνίας ο κάθε κόμβος επιλέγει τυχαία και ομοιόμορφα έναν από τους γείτονές του και στέλλει σε αυτόν όλες τις πληροφορίες που γνωρίζει μέχρι το συγκεκριμένο γύρο επικοινωνίας. Ο μηχανισμός που χρησιμοποιεί ο αλγόριθμος αυτός για να τερματίσει είναι επίσης απλός και βασίζεται στο γεγονός ότι οι πληροφορίες είναι στατικές και οι κόμβοι γνωρίζουν το συνολικό αριθμό πληροφοριών που υπάρχουν στο

δίκτυο. Έτσι, όταν ένας κόμβος μάθει όλες τις *n* πληροφορίες σταματά να στέλλει στους γείτονές του αυτά που γνωρίζει και μόνο στην περίπτωση που θα παραλάβει μήνυμα από κάποιον γείτονά του θα του απαντήσει με όσα γνωρίζει, δηλαδή με όλες τις πληροφορίες του δικτύου. Με αυτό τον τρόπο όταν όλοι οι κόμβοι μάθουν όλες τις πληροφορίες θα σταματήσουν να ανταλλάσουν μηνύματα και ο αλγόριθμος θα τερματίσει.

Ο δεύτερος αλγόριθμος, που καλούμε PUSH&PULL, βασίζεται στο γεγονός ότι κάθε πληροφορία έχει προθεσμία μετάδοσης, επομένως από κάποιο σημείο και μετά δεν θα πρέπει να μεταδίδεται. Σε κάθε γύρο επικοινωνίας ο κάθε κόμβος επιλέγει τυχαία και ομοιόμορφα έναν από τους γείτονές του και στέλλει σε αυτόν τις πληροφορίες που γνωρίζει μέχρι τον συγκεκριμένο γύρο επικοινωνίας και που δεν έχει λήξει η προθεσμία μετάδοσής τους. Επίσης στον ίδιο γύρο επικοινωνίας ο κόμβος αυτός λαμβάνει από τον ίδιο γείτονα τις πληροφορίες του, των οποίων η προθεσμία μετάδοσης δεν έχει ακόμη λήξει. Ο αλγόριθμος αυτός τερματίζει όταν όλες οι πληροφορίες του δικτύου έχουν ξεπεράσει την προθεσμία μετάδοσής τους, οπότε κανένας κόμβος δεν έχει κάποια πρόσφατη πληροφορία για να ανταλλάξει με το γείτονά του.

Ο τρίτος αλγόριθμος, που καλούμε MEDIAN-COUNTER, βασίζεται επίσης στο γεγονός ότι κάθε πληροφορία παύει να είναι χρήσιμη από κάποιο σημείο και μετά αλλά δεν ορίζει ακριβή προθεσμία μετάδοσης για τις πληροφορίες. Θεωρεί ότι μια πληροφορία ενός κόμβου μπορεί να είναι σε μια από τις τέσσερις καταστάσεις A, B, C ή D ανάλογα με το χρονικό διάστημα που βρίσκεται στον κόμβο αυτό, και ένας κόμβος μεταδίδει μόνο τις πληροφορίες του που βρίσκονται σε μία από τις καταστάσεις B ή C. Η μετάδοση πληροφοριών γίνεται όπως και στον αλγόριθμο PUSH&PULL και προς τις δύο κατευθύνσεις και ο αλγόριθμος τερματίζει όταν κανένας κόμβος δεν έχει πληροφορίες σε κατάσταση B ή C για να μεταδώσει.

Ο τελευταίος αλγόριθμος υλοποιείται και αξιολογείται και με σφάλματα δικτύου για να δειχθεί κατά πόσον ο αλγόριθμος είναι εύρωστος. Εφαρμόζονται δύο ειδών σφάλματα δικτύου τα οποία συμβαίνουν με κάποια πιθανότητα. Στο πρώτο έχουμε τους κόμβους του δικτύου να καταρρέουν ενώ στο δεύτερο έχουμε σφάλματα στην εγκατάσταση σύνδεσης μεταξύ δύο κόμβων.

Ως στόχος λοιπόν αυτής της διπλωματικής εργασίας είναι αρχικά η εις βάθος κατανόηση των πιο πάνω αλγορίθμων και στη συνέχεια η υλοποίηση και η πειραματική αξιολόγησή τους. Σημαντικό μέρος της εργασίας είναι η κατανόηση και η εκμάθηση δύο νέων βιβλιοθηκών της γλώσσας προγραμματισμού Java, των βιβλιοθηκών GossipLip [4] και Yalps [5], οι οποίες χρησιμοποιούνται για την υλοποίηση των αλγορίθμων πληροφόρησης με τους οποίους ασχολείται η εργασία.

1.3 Μεθοδολογία

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

Αρχικά μελέτησα διάφορους αλγόριθμους πληροφόρησης κάποιους από τους οποίους μου εισηγήθηκε ο Δρ. Χρύσης Γεωργίου ενώ κάποιους άλλους τους περιέγραφαν επιστημονικά άρθρα [1] που μου δόθηκαν επίσης από τον επιβλέπων καθηγητή. Αφού κατανόησα τη χρησιμότητα των αλγορίθμων αυτών αλλά και την ακριβής λειτουργία τους έπρεπε να προχωρήσω στην υλοποίησή τους για να καταφέρω στο τέλος να τους αξιολογήσω και να διεκπεραιώσω την εργασία μου. Για το σκοπό αυτό έπρεπε να εξοικειωθώ με τις δύο νέες βιβλιοθήκες της Java, GossipLib και Yalps, οι οποίες ήταν απαραίτητες για την υλοποίηση των αλγορίθμων πληροφόρησης. Καθώς η υλοποίηση των βιβλιοθηκών αυτών όπως και της τεκμηρίωσής τους δεν ήταν ολοκληρωμένα τη στιγμή που ξεκίνησα την εργασία, χρειάστηκε καθοδήγηση από ένα μέλος της ερευνητικής ομάδας του ιδρύματος INRIA για το πώς θα μπορούσα να τις χρησιμοποιήσω στην εργασία μου. Αφού ενημερώθηκα επαρκώς για τη χρήση των βιβλιοθηκών ξεκίνησα την περεταίρω εκμάθησή τους υλοποιώντας σε αυτές απλά παραδείγματα αλγορίθμων πληροφόρησης. Μετά την εκμάθηση των δύο βιβλιοθηκών ακολούθησε η υλοποίηση των τριών αλγορίθμων πληροφόρησης PUSH, PUSH&PULL και MEDIAN-COUNTER. Στη συνέχεια έγινε η πειραματική αξιολόγηση προσομοιώνοντας τους αλγορίθμους που υλοποίησα ώστε να διαφανεί κατά πόσον η θεωρητική αξιολόγηση των αλγορίθμων συναύει με την πρακτική τους αξιολόγηση. Τέλος σύγκρινα τους αλγορίθμους μεταξύ τους και κατέληξα σε συμπεράσματα σχετικά με την απόδοσή τους.

1.4 Οργάνωση της Εργασίας

Η εργασία αυτή αποτελείται από οκτό κεφάλαια. Το επόμενο κεφάλαιο, Κεφάλαιο 2, ασχολείται με το υπόβαθρο της εργασίας δηλαδή με τη γενική περιγραφή των αλγορίθμων πληροφόρησης καθώς και την περιγραφή των δύο βιβλιοθηκών που χρησιμοποιήθηκαν στην εργασία αυτή. Η υλοποίηση και η αξιολόγηση των αλγορίθμων αρχίζει με το Κεφάλαιο 3 το οποίο ασχολείται με τον αλγόριθμο PUSH. Ακολουθεί η μελέτη του αλγορίθμου PUSH&PULL στο Κεφάλαιο 4. Ο αλγόριθμος MEDIAN-COUNTER αξιολογείται στο Κεφάλαιο 5 ενώ στο επόμενο κεφάλαιο, Κεφάλαιο 6, αξιολογούνται οι δύο υλοποιήσεις του αλγορίθμου αυτού στις οποίες υπάρχουν σφάλματα. Στο Κεφάλαιο 7 γίνεται η σύγκριση μεταξύ των τριών αλγορίθμων και η εργασία κλείνει με το Κεφάλαιο 8 όπου περιγράφονται τα γενικά συμπεράσματα της εργασίας καθώς και μελλοντικές εργασίες.

ΚΕΦΑΛΑΙΟ 2

Υπόβαθρο

2.1 Πρωτόκολλα Πληροφόρησης

Στις μέρες μας εκτός από το κλασσικό μοντέλο πελάτη-εξυπηρετητή (client-server) έχουν αναπτυχθεί σε μεγάλο βαθμό τα δίκτυα ομοτίμων (Peer-to-Peer Networks). Η αποκεντρωμένη φύση των δικτύων ομοτίμων οδήγησε στη δημιουργία νέων πρωτοκόλλων, όπως είναι τα πρωτόκολλα πληροφόρησης για αποκεντρωμένα δίκτυα. Τα πρωτόκολλα πληροφόρησης περιλαμβάνουν τους αλγόριθμους πληροφόρησης, ή αλλιώς επιδημικούς αλγόριθμους, οι οποίοι είναι υπεύθυνοι για την επικοινωνία στα δίκτυα ομοτίμων. Ονομάζονται επιδημικοί αλγόριθμοι γιατί μεταδίδουν την πληροφορία με τρόπο παρόμοιο με αυτόν που ένας υιός εξαπλώνεται σε ένα βιολογικό πληθυσμό [2].

Υπάρχουν διάφορα πρωτόκολλα πληροφόρησης τα οποία διαφέρουν κυρίως στον τρόπο με τον οποίο μεταδίδουν την πληροφορία ή στον τρόπο με τον οποίο την διαχειρίζονται. Οι τρείς επικρατέστερες μορφές πρωτοκόλλων πληροφόρησης είναι: [2]

Πρωτόκολλα διάχισης (dissemination ή rumor-mongering protocols): Η διαδικασία που χρησιμοποιείται για τη διάδοση μιας πληροφορίας έχει στυλ κουτσομπολιού. Δηλαδή, όταν ένας κόμβος παραλάβει μία νέα πληροφορία την χαρακτηρίζει ως ‘hot’. Όσο κάποιος κόμβος κρατά μία ‘hot’ πληροφορία επιλέγει περιοδικά και με τυχαίο τρόπο έναν άλλο κόμβο στο δίκτυο και του την στέλλει.

Πρωτόκολλα αντι-εντροπίας (anti-entropy protocols): Χρησιμοποιείται στην περίπτωση όπου υπάρχουν επαναλαμβανόμενα δεδομένα. Κάθε κόμβος επιλέγει τακτικά και με τυχαίο τρόπο έναν άλλο κόμβο στο δίκτυο και στέλλει σε αυτόν όλα τα δεδομένα που κρατά. Ο παραλήπτης συγκρίνει τα δικά του δεδομένα με αυτά που παράλαβε και φροντίζει να διορθώσει τυχόν διαφορές που υπάρχουν μεταξύ των δύο δεδομένων.

Πρωτόκολλα Συνδυασμένης Διάδοσης (protocols that compute aggregates): Χρησιμοποιούνται για τον υπολογισμό στοιχείων τα οποία αφορούν ολόκληρο το δίκτυο. Επιτυγχάνεται με τη συλλογή πληροφοριών από όλους τους κόμβους του δικτύου και ακολούθως οι πληροφορίες αυτές συνδυάζονται για να καταλήξουν σε μία πληροφορία που θα αφορά όλο το δίκτυο.

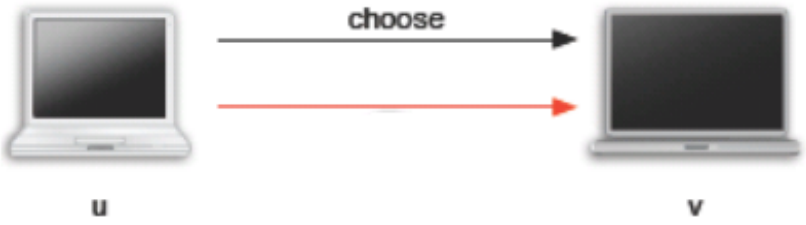
Η εργασία αυτή ασχολείται με τις δύο πρώτες μορφές πρωτοκόλλων πληροφόρησης, τα πρωτόκολλα διάχισης και τα πρωτόκολλα αντι-εντροπίας. Τα πρωτόκολλα αντι-εντροπίας είναι πολύ αξιόπιστα αλλά για να μεταδώσουν οι κόμβοι όλα τους τα δεδομένα χρειάζεται πολύς χρόνος και κόστος. Αντίθετα, τα πρωτόκολλα διάχισης διαδίδουν μόνο τις πρόσφατες πληροφορίες και έτσι χρειάζονται λιγότερο χρόνο και κόστος για να τις μεταδώσουν. Μία πρακτική λύση είναι να συνδυαστούν τα δύο πρωτόκολλα για να αξιοποιηθούν τα πλεονεκτήματα και των δύο.

2.2 Αλγόριθμοι Πληροφόρησης

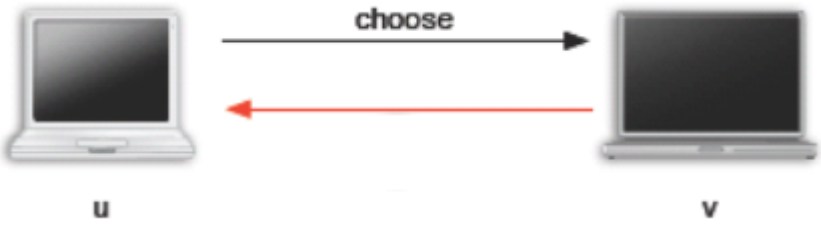
Οι αλγόριθμοι πληροφόρησης έχουν ως στόχο να ενημερωθούν όλοι οι κόμβοι σε ένα δίκτυο με όλες τις πληροφορίες που υπάρχουν στον δίκτυο. Είναι από τη φύση τους πιθανοτικοί αλγόριθμοι γιατί βασίζονται σε τυχαία επιλογή κόμβου. Κάθε κόμβος χρησιμοποιεί την τοπική του γνώση και ένα αλγόριθμο για τυχαία και ομοιόμορφη επιλογή κόμβου για να επιλέξει το γείτονά του με τον οποίο θα ανταλλάξει πληροφορίες. Οι αλγόριθμοι δουλεύουν σε γύρους όπου κάθε γύρος αποτελείται από τη φάση επικοινωνίας και τη φάση επεξεργασίας. Στη φάση επικοινωνίας ο κάθε κόμβος επιλέγει αρχικά τον κόμβο με τον οποίο θα επικοινωνήσει, εγκαθιστά τις απαραίτητες συνδέσεις για να μπορέσει να επικοινωνήσει μαζί του και στη συνέχεια γίνεται η ανταλλαγή των πληροφοριών. Στη φάση επεξεργασίας οι κόμβοι που παρέλαβαν πληροφορίες τις επεξεργάζονται ανάλογα με τις ανάγκες του αλγορίθμου και

καθορίζουν με βάση τα νέα δεδομένα ποιές πληροφορίες πρέπει να μεταδώσουν στον επόμενο γύρο.

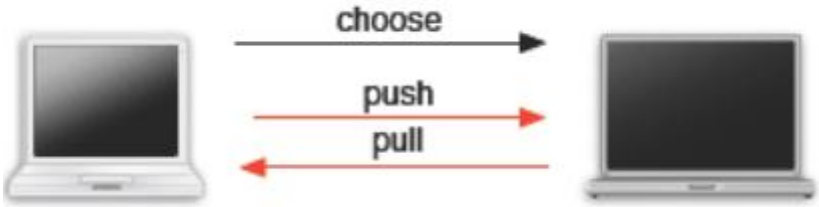
Ένα σημαντικό χαρακτηριστικό ενός αλγορίθμου πληροφόρησης είναι το μοντέλο μετάδοσης πληροφοριών που χρησιμοποιεί. Υπάρχουν δύο βασικά μοντέλα που εντέλει μπορεί να συνδυαστούν για να βελτιωθεί η απόδοση του αλγορίθμου [3]. Το ένα είναι το μοντέλο ώθησης (push) και το άλλο το μοντέλο λήψης (pull). Στο μοντέλο ώθησης ο κόμβος που έχει την πληροφορία την στέλνει στον κόμβο που την χρειάζεται, ενώ στο μοντέλο λήψης ο κόμβος που χρειάζεται την πληροφορία τη λαμβάνει από τον κόμβο που την έχει. Στο μοντέλο που συνδυάζει ώθηση και λήψη μηνυμάτων συμβαίνουν και οι δύο λειτουργίες στον ίδιο γύρο. Στα σχήματα πιο κάτω παρουσιάζονται τα τρία μοντέλα με τη σειρά που αναφέρθηκαν:



Εικόνα 2.1 Μοντέλο ώθησης (Push) [3]



Εικόνα 2.2 Μοντέλο λήψης (Pull) [3]



Εικόνα 2.3 Μοντέλο ώθησης και λήψης (Push And Pull) [3]

Ένα κρίσιμο σημείο των αλγορίθμων πληροφόρησης είναι το σημείο στο οποίο ο αλγόριθμος πρέπει να τερματίσει. Δεν είναι πολύ πρακτικό να θεωρηθεί ότι ένας αλγόριθμος πληροφόρησης θα τερματίσει όταν κάθε κόμβος του δικτύου θα παραλάβει κάποιο μήνυμα. Αυτό γιατί στα δυναμικά δίκτυα κάποιοι κόμβοι μπορεί να καταρρεύσουν ενώ νέοι κόμβοι μπορεί να εισαχθούν στο δίκτυο. Ένας τρόπος για να τερματίσει ο αλγόριθμος είναι να οριστούν κάποια όρια στα οποία θα θεωρείται ότι ο αλγόριθμος έχει εξαπλώσει αρκετά την πληροφορία. Έτσι όταν ο αλγόριθμος φτάσει στα όρια αυτά οι κόμβοι του δικτύου θα έχουν πιθανότατα παραλάβει την πληροφορία και ο αλγόριθμος θα τερματίζει.

Συνοψίζοντας, οι αλγόριθμοι πληροφόρησης είναι ευρέως διαδεδομένοι λόγω της απλότητάς τους και της ανεκτικότητάς τους σε σφάλματα δικτύου καθώς και της ανεκτικότητάς τους στην αλλαγή τοπολογίας του δικτύου.

2.3 Η βιβλιοθήκη YALPS

Η βιβλιοθήκη YALPS [5] είναι μία ανοικτού κώδικα βιβλιοθήκη της Java η οποία σχεδιάστηκε για να διευκολύνει την ανάπτυξη, τον έλεγχο και τη συντήρηση κατανεμημένων εφαρμογών. Μία εφαρμογή που γράφεται χρησιμοποιώντας τη βιβλιοθήκη YALPS μπορεί να τρέξει είτε σε προσομοίωση είτε σε πραγματικά κατανεμημένα συστήματα χωρίς καμία αλλαγή στον κώδικα της εφαρμογής. Αυτό είναι ένα πολύ σημαντικό πλεονέκτημα της βιβλιοθήκης γιατί μας επιτρέπει να δημιουργήσουμε το πρωτότυπο μιας εφαρμογής και να την ελέγξουμε πλήρως σε προσομοίωση και μόνον όταν η εφαρμογή θα είναι έτοιμη θα την βγάλουμε στο πραγματικό σύστημα.

Οι εφαρμογές της βιβλιοθήκης YALPS είναι οργανωμένες σαν μία συλλογή από εργασίες ‘Tasks’ τις οποίες διαχειρίζεται ο διαχειριστής εργασιών ‘TaskManager’. Ο διαχειριστής εργασιών τους επιτρέπει να καθορίσουν το χρόνο στον οποίο θα εκτελεστούν, για παράδειγμα μπορεί να εκτελεστούν άμεσα ή μετά από κάποιο χρονικό διάστημα ή μπορούν να εκτελούνται περιοδικά. Επίσης, οι εργασίες μπορούν να ομαδοποιηθούν για να διαχειρίζονται σαν ομάδα αντί σαν ατομικές εργασίες. Αυτό επιτρέπει στην εφαρμογή να αλλάξει εντελώς την εκτέλεσή της με μία απλή διαφοροποίηση σε μία ομάδα εργασιών. Ακόμη, για να μπορεί η ίδια εφαρμογή να

τρέξει είτε σε προσομοίωση είτε σε πραγματικό κατανεμημένο σύστημα υπάρχουν δύο υλοποιήσεις του διαχειριστή εργασιών, ο SimulatedTaskManager και ο ExecutorTaskManager, αντίστοιχα.

Ένα άλλο στοιχείο της βιβλιοθήκης YALPS είναι ο τρόπος με τον οποίο επιτυγχάνεται η επικοινωνία. Για το σκοπό αυτό υπάρχει το στρώμα επικοινωνίας της YALPS 'YALPS's CommunicationLayer' για το οποίο υπάρχουν επίσης δύο εξειδικεύσεις, μία για να τρέξει η εφαρμογή σε προσομοίωση και μία για να μπορεί να τρέξει σε πραγματικό σύστημα. Για το πραγματικό σύστημα τα μηνύματα της εφαρμογής δρομολογούνται μέσω των πρωτοκόλλων UDP/TCP ενώ για την προσομοίωση της εφαρμογής χρησιμοποιείται μία ειδική υποδομή επικοινωνίας της YALPS. Και στις δύο περιπτώσεις, το στρώμα επικοινωνίας της YALPS μας προσφέρει λειτουργίες για έλεγχο και αξιολόγηση των κατανεμημένων εφαρμογών, όπως είναι ο περιορισμός του εξερχόμενου εύρους ζώνης και η επιλογή για απώλεια μηνυμάτων κατά την εκτέλεση εφαρμογής.

Τέλος, τα μηνύματα που ανταλλάσσονται μέσω των εφαρμογών της YALPS πρέπει να είναι σειριοποιήσιμα, 'serialized'. Σειριοποίηση ενός μηνύματος είναι η διαδικασία μετατροπής του μηνύματος σε μια μορφή η οποία μπορεί να αποθηκευτεί ή να μεταδοθεί μέσω μίας σύνδεσης ενός δικτύου [6]. Σε αυτή την εργασία τα μηνύματα σειριοποιούνται για να μπορέσουν να μεταδοθούν από τον ένα κόμβο στον άλλο μέσω του δικτύου. Όταν το μήνυμα φτάσει στον παραλήπτη τότε μπορεί εύκολα να επανέλθει στην αρχική του μορφή. Υπάρχουν δύο τρόποι για να επιτευχθεί αυτό σε μία εφαρμογή της YALPS. Είτε χρησιμοποιώντας την πρότυπη μέθοδο για σειριοποίηση της Java (κλάση Serializable), είτε χρησιμοποιώντας ένα πιο αποτελεσματικό μηχανισμό ο οποίος μπορεί να υλοποιηθεί μέσω της διεπαφής της βιβλιοθήκης YALPS και με βάση τις απαιτήσεις της εφαρμογής.

Έτσι, έχοντας τη βιβλιοθήκη YALPS στα χέρια μας η οποία μας δίνει έτοιμα όλα τα απαραίτητα στοιχεία μίας κατανεμημένης εφαρμογής, μπορούμε πολύ εύκολα να υλοποιήσουμε κατανεμημένες εφαρμογές, να τις προσομοιώσουμε ή να τις τρέξουμε σε πραγματικά κατανεμημένα συστήματα χωρίς να χρειάζεται να ανησυχούμε για τα πρακτικά θέματα δικτύων ή και για άλλες τεχνικές δυσκολίες.

2.3.1 Συστατικά στοιχεία της βιβλιοθήκης YALPS

Βασικές Διεπαφές της YALPS

Διεπαφή NodeID

Οι κόμβοι μίας εφαρμογής δηλώνονται ως αντικείμενα της διεπαφής NodeID, χωρίς να έχει σημασία αν η εφαρμογή θα τρέξει σε προσομοίωση ή σε πραγματικό σύστημα. Στην πραγματικότητα όμως υπάρχουν δύο κλάσεις που υλοποιούν τη διεπαφή NodeID, η κλάση E_NodeID για την εκτέλεση της εφαρμογής σε πραγματικό σύστημα και η κλάση S_NodeID για την προσομοίωση της εφαρμογής. Το ποιά από τις δύο υλοποιήσεις θα επιλεχθεί καθορίζεται από το αρχείο διαμόρφωσης (configuration file) τη στιγμή που θα τρέξει η εφαρμογή.

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

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

όπου το αντικείμενο '`cl`' αντιστοιχεί στο υπόστρωμα επικοινωνίας το οποίο χρησιμοποιεί η εφαρμογή και η συνάρτηση '`getNodeIdFromString(NodeName)`' επιστρέφει το μοναδικό αναγνωριστικό του κόμβου με το όνομα '`NodeName`'. Το μοναδικό αναγνωριστικό που επιστρέφει η συνάρτηση φυλάγεται στη μεταβλητή '`MyNode`'.

Διεπαφή YalpsNode

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

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

```
public class MyApplication extends AbstractYalpsNode{

   ....

}
```

Διεπαφή Task

Οι εργασίες (tasks) αποτελούν το ενεργό μέρος των εφαρμογών της YALPS. Μία εργασία είναι απλά ένα αντικείμενο που υλοποιεί τη διεπαφή Task. Για να ολοκληρωθεί η υλοποίηση της διεπαφής Task πρέπει να προστεθεί κώδικας στη μέθοδο run() του αντικειμένου ο οποίος θα περιγράφει τη λειτουργία της εργασίας. Το εργαλείο που χρησιμοποιούν οι εφαρμογές YALPS για να προγραμματίσουν τις εργασίες τους είναι το αντικείμενο τύπου TaskManager που υπάρχει σε κάθε εφαρμογή. Επιπλέον, οι εργασίες μπορούν να ομαδοποιηθούν χρησιμοποιώντας το μηχανισμό ομαδοποίησης εργασιών (TaskGroup) για να τις διαχειρίζεται ο TaskManager σαν ομάδα. Η δημιουργία μίας εργασίας γίνεται με τον εξής τρόπο:

```
class MyTask implements Task{

   public void run(){

   ....

   }

}
```

Υπάρχουν τρία είδη εργασιών. Οι εργασίες που εκτελούνται αμέσως μόλις γίνει η καταχώρησή τους, οι εργασίες που εκτελούνται σε συγκεκριμένο χρονικό διάστημα μετά την καταχώρησή τους και οι εργασίες που εκτελούνται περιοδικά. Η διεπαφή TaskManager μας προσφέρει διάφορες μεθόδους για τον προγραμματισμό της εκτέλεσης των εργασιών.

Άμεση εκτέλεση εργασιών:

Η μέθοδος

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

καταχωρεί την εργασία t και την συσχετίζει με την ομάδα εργασιών g. Αυτός ο τρόπος καταχώρησης της εργασίας t, συνεπάγεται ότι η εργασία θα εκτελεστεί άμεσα. Η μέθοδος αυτή μπορεί να κληθεί από τον διαχειριστή εργασιών "tm" της εφαρμογής με τον εξής τρόπο:

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

όπου `tm.getSystemTaskGroup()` θα επιστρέψει την προεπιλεγμένη ομάδα εργασιών του συστήματος.

Αναβεβλημένη εκτέλεση εργασιών:

Αν θέλουμε μία εργασία να μην εκτελεστεί άμεσα αλλά μετά από συγκεκριμένο χρονικό διάστημα τότε υπάρχει η μέθοδος

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

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

Περιοδικές εργασίες:

Τελευταίο είδος εργασίας είναι η εργασία που εκτελείτε περιοδικά, δηλαδή ανά τακτά χρονικά διαστήματα. Για το σκοπό αυτό υπάρχει η μέθοδος

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

η οποία διαφέρει με τις προηγούμενες στο ότι καταχωρεί την 't' ως περιοδική εργασία (PeriodicTask) αντί σαν απλή εργασία. Η PeriodicTask είναι μία διεπαφή που εντάσσεται στη διεπαφή Task και έχει κάποιες επιπλέον μεθόδους τις οποίες χρειάζονται αυτού του είδους οι εργασίες όπως είναι η μέθοδος `getPeriod()`. Η μέθοδος αυτή επιστρέφει το χρονικό διάστημα στο οποίο η εργασία θα πρέπει να επανεκτελεστεί κάτι το οποίο καθορίζεται από την παράμετρο '`long millis`'.

Αποστολή Μηνυμάτων

Η επικοινωνία μεταξύ των κόμβων μίας εφαρμογής YALPS γίνεται μέσω ενός αντικειμένου τύπου CommunicationLayer το οποίο υπάρχει σε όλες τις εφαρμογές της

YALPS. Το αντικείμενο αυτό έχει δύο διαφορετικές μεθόδους για αποστολή μηνυμάτων, η μία χρησιμοποιείται για αποστολή μηνυμάτων χρησιμοποιώντας το πρωτόκολλο TCP και η άλλη για αποστολή μηνυμάτων χρησιμοποιώντας το πρωτόκολλο UDP.

Η μέθοδος

```
public void sendTCP(Serializable msg, NodeID dest)

throws IOException;
```

στέλλει το μήνυμα ‘msg’ στον κόμβο ‘dest’ χρησιμοποιώντας το TCP πρωτόκολλο επικοινωνίας.

Η μέθοδος

```
public abstract void sendUDP(Serializable msg, NodeID dest)

throws IOException;
```

στέλλει το μήνυμα ‘msg’ στον κόμβο ‘dest’ χρησιμοποιώντας το UDP πρωτόκολλο επικοινωνίας.

Παραλαβή Μηνυμάτων

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

Η μέθοδος

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

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

Αρχικοποίηση του Κόμβου

Μία σημαντική μέθοδος της κλάσης AbstractYalpsNode είναι αυτή που επιτρέπει στον κόμβο να εκτελέσει όλες τις αρχικοποιήσεις που είναι απαραίτητες για την περεταίρω λειτουργία της εφαρμογής. Οι αρχικοποιήσεις αυτές αφορούν συνήθως

προγραμματισμό των εργασιών του κόμβου ή αποστολή κάποιων μηνυμάτων στους υπόλοιπους κόμβους της εφαρμογής.

Η μέθοδος

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

κληρονομείται και πάλι από την κλάση AbstractYalpsNode και στην υλοποίησή της μπορούν να συμπεριληφθούν όλες οι αρχικοποιήσεις. Ένα παράδειγμα υλοποίησης της μεθόδου είναι:

```
public void startNode(){

   tm.registerTask(new myTask(), tm.getSystemTaskGroup());

}
```

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

2.3.2 Εκτελεστές μίας Εφαρμογής YALPS

Αφού υλοποιηθεί μία κλάση της διεπαφής YalpsNode δημιουργείται η εφαρμογή YALPS η οποία μπορεί να εκτελεστεί είτε σε προσομοίωση είτε σε πραγματικό κατανεμημένο σύστημα. Υπάρχουν δύο εκτελεστές της YALPS οι οποίοι χρησιμοποιούνται ανάλογα με τον τρόπο που θέλουμε να τρέξει η εφαρμογή.

Ο εκτελεστής

```
yalps.launchers.ExecutorNodeStarter
```

χρησιμοποιείται για να τρέξουμε την εφαρμογή σε πραγματικό σύστημα.

Ο εκτελεστής

```
yalps.launchers.SimulatedNodeStarter
```

χρησιμοποιείται για να τρέξουμε την εφαρμογή σε προσομοίωση.

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

2.3.3 Αρχείο Διαμόρφωσης

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

Δηλώσεις Αρχικοποίησης Αντικειμένων

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

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

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

Δηλώσεις Ρύθμισης Ιδιοτήτων

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

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

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

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

void set+‘property name’();

Όπου με το ‘+’ δηλώνω τη συνένωση των δύο συμβολοσειρών, ‘set’ και ‘property name’. Η παράμετρος της μεθόδου πρέπει να είναι συμβατή με τον τύπο της τιμής

‘value’. Ο εκτελεστής της YALPS περνά αυτόματα την τιμή ‘value’ σαν παράμετρο στην πιο πάνω μέθοδο μετατρέποντάς την στον κατάλληλο τύπο.

Ακολουθεί ένα παράδειγμα αρχείου διαμόρφωσης της YALPS:

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

node.nodeList=10
```

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

```
void setNodeList(Type myParam) {

   ....

}
```

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

Αρχεία Διαμόρφωσης για Προσομοίωση και Πραγματική Εκτέλεση

Μία εφαρμογή γράφεται για να τρέξει σε προσομοίωση ή για να τρέξει σε πραγματικό σύστημα. Και στις δύο περιπτώσεις το αρχείο διαμόρφωσης θα έχει την ίδια μορφή με τη μόνη διαφορά στο σημείο που δηλώνονται οι κόμβοι της εφαρμογής. Ακολουθούν παραδείγματα και για τις δύο περιπτώσεις.

Παράδειγμα δήλωσης των κόμβων της εφαρμογής που θα τρέξει σε προσομοίωση:

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

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

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

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

2.3.4 Εκτέλεση μίας εφαρμογής της βιβλιοθήκης YALPS

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

Παράδειγμα εντολής για προσομοίωση της εφαρμογής:

java -cp /yalps.jar launchers.SimulatedNodeStarter myExample-sim.conf

Παράδειγμα εντολής για τρέξιμο της εφαρμογής σε πραγματικό σύστημα:

java -cp /yalps.jar launchers.ExecutorNodeStarter myExample-exec.conf

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

2.4 Η βιβλιοθήκη GossipLib

Η βιβλιοθήκη GossipLib [4] είναι μία βιβλιοθήκη της Java η οποία υποστηρίζει την ανάπτυξη εφαρμογών πληροφόρησης. Τα πρωτόκολλα πληροφόρησης εμφανίστηκαν αρχικά σαν μία οικογένεια αλγορίθμων πληροφόρησης οι οποίοι χρησιμοποιούνταν για τη συντήρηση κατανεμημένων βάσεων δεδομένων. Οι αλγόριθμοι αυτοί στη συνέχεια εφαρμόστηκαν σε ένα πιο ευρύ φάσμα καταστάσεων, όπως είναι η διάδοση δεδομένων σε ένα δίκτυο ομοτίμων.

Η GossipLib είναι μία βιβλιοθήκη που έχει σαν βάση της τη βιβλιοθήκη YALPS. Σχεδιάστηκε για να διευκολύνει τη διαδικασία ανάπτυξης εφαρμογών πληροφόρησης. Για το σκοπό αυτό προσφέρει ένα σύνολο βοηθητικών κλάσεων οι οποίες μπορούν να αποτελέσουν το σημείο εκκίνησης στη δημιουργία μίας εφαρμογής πληροφόρησης. Βασικό μέλημα της βιβλιοθήκης είναι η διευκόλυνση στον έλεγχο της ορθότητας και της αποδοτικότητας κατανεμημένων εφαρμογών καθώς και η επαναχρησιμοποίηση του κώδικα. Κάθε συστατικό σε μία εφαρμογή πληροφόρησης της GossipLib μπορεί να χρησιμοποιηθεί στην ανάπτυξη μίας νέας και πιο πολύπλοκης εφαρμογής.

Ένας αριθμός πρότυπων πρωτοκόλλων πληροφόρησης είναι υλοποιημένα και έτοιμα για χρήση στη βιβλιοθήκη GossipLib. Τα πρωτόκολλα αυτά μπορούν να χρησιμοποιηθούν όπως είναι, χωρίς καμία αλλαγή, ή μπορούν να επεκταθούν ανάλογα με τις ανάγκες ενός νέου πρωτοκόλλου ή μιας νέας εφαρμογής πληροφόρησης.

Όπως και στη βιβλιοθήκη YALPS, έτσι και στην GossipLib μία εφαρμογή μπορεί να διαμορφωθεί και να αναπτυχθεί ως τελικό προϊόν το οποίο θα τρέξει σε πραγματικό κατανεμημένο σύστημα ή μπορεί να αναπτυχθεί σαν πρωτότυπο το οποίο θα χρησιμοποιηθεί για έρευνα και για προσομοίωση της εφαρμογής. Και πάλι ο κώδικας της εφαρμογής θα είναι ο ίδιος, ανεξάρτητα του πώς θα τρέξει η εφαρμογή, και η μόνη αλλαγή που χρειάζεται να γίνει είναι σε μία δήλωση στο αρχείο διαμόρφωσης.

2.4.1 Συστατικά στοιχεία της βιβλιοθήκης GossipLib

Κλάση Peer

Οι κόμβοι ενός συστήματος στη βιβλιοθήκη GossipLib ονομάζονται ‘peers’. Μία σημαντική κλάση της βιβλιοθήκης είναι η κλάση Peer η οποία αντιπροσωπεύει ένα κόμβο σε μία εφαρμογή ή σε ένα πρωτόκολλο πληροφόρησης. Η κλάση αυτή προσφέρει όλες τις απαραίτητες συναρτήσεις για τη διαχείριση ενός κόμβου όπως είναι η επεξεργασία των δεδομένων που κρατά ο κόμβος και η επεξεργασία των στοιχείων του κόμβου όπως είναι η ηλικία του.

Ακολουθεί ένα παράδειγμα δήλωσης ενός κόμβου και καθορισμός των δεδομένων που έχει:

```
Peer myPeer=new Peer();

myPeer.addPeerData(myData);
```

όπου ‘myData’ πρέπει να είναι σειριοποιήσιμο στοιχείο, όπως συμβαίνει και στην YALPS.

Διεπαφές Peer Selector και Peer Selection

Η βιβλιοθήκη GossipLib προσφέρει τη διεπαφή PeerSelector η οποία επιτρέπει σε έναν κόμβο μίας εφαρμογής να επιλέξει ένα μη προκαθορισμένο σύνολο από τους

υπόλοιπους κόμβους της εφαρμογής με τους οποίους θέλει να επικοινωνήσει. Ένα αντικείμενο τύπου PeerSelector μπορεί να χρησιμοποιήσει τις συναρτήσεις της διεπαφής PeerSelection για να επιλέξει τους κόμβους με τους οποίους θα επικοινωνήσει.

Οι συναρτήσεις της διεπαφής PeerSelection για επιλογή κόμβων είναι:

```
View getView() throws UnavailableViewException;
```

η οποία επιστρέφει ένα σύνολο από κόμβους της εφαρμογής.

```
Peer getPeer() throws EmptyViewException;
```

η οποία επιστρέφει έναν μόνο κόμβο της εφαρμογής.

Διεπαφή GossipApplication

Η βασική αφαιρετική κλάση της βιβλιοθήκης GossipLib είναι η AbstractGossipApplication η οποία υλοποιεί τη διεπαφή GossipApplication. Η κλάση AbstractGossipApplication αντιπροσωπεύει μία γενική εφαρμογή πληροφόρησης και διευκολύνει την ανάπτυξη τέτοιων εφαρμογών αφού περιέχει κωδικοποίηση των συστατικών μιας εφαρμογής τα οποία είναι απαραίτητα για όλες τις εφαρμογές της GossipLib.

Συνεπώς, η δημόσια κλάση σε κάθε εφαρμογή της GossipLib μπορεί να επεκτείνει την κλάση AbstractGossipApplication με τον εξής τρόπο:

```
public class MyApplication extends AbstractGossipApplication {

   ....

}
```

Παραδείγματα συστατικών τα οποία περιέχονται στην κλάση AbstractGossipApplication και τα οποία είναι απαραίτητα για τις εφαρμογές πληροφόρησης είναι οι δύο συναρτήσεις

```
void activeCycle();
```

```
void passiveCycle(Serializable msg, Peer sender);
```

Οι εφαρμογές της GossipLib αποτελούνται από δύο βασικά μέρη, το ενεργό και το παθητικό μέρος.

Ενεργό μέρος

Κάθε εφαρμογή της GossipLib έχει ένα ενεργό μέρος στο οποίο ο κάθε κόμβος της εφαρμογής μπορεί να δημιουργήσει ένα μήνυμα και να επιλέξει με κάποιο τρόπο κάποιους από τους γείτονές του για τους το στείλει. Για το σκοπό αυτό υπάρχει στην αφαιρετική κλάση AbstractGossipApplication η συνάρτηση

```
void activeCycle();
```

η οποία πρέπει να υλοποιηθεί από κάθε εφαρμογή της GossipLib ανάλογα με τις ανάγκες της συγκεκριμένης εφαρμογής.

Παθητικό μέρος

Εκτός από το ενεργό μέρος, μία εφαρμογή έχει και το παθητικό της μέρος. Ένας κόμβος βρίσκεται στο παθητικό μέρος της εφαρμογής όταν παραλαμβάνει μήνυμα από κάποιο άλλο κόμβο. Για το σκοπό αυτό υπάρχει στην αφαιρετική κλάση AbstractGossipApplication η συνάρτηση

```
void passiveCycle(Serializable msg, Peer sender);
```

όπου ‘msg’ το μήνυμα που παράλαβε ο κόμβος και ‘sender’ ο αποστολέας του μηνύματος. Η συνάρτηση πρέπει να υλοποιηθεί από κάθε εφαρμογή της GossipLib για να καθορίσει στον κόμβο πώς πρέπει να επεξεργαστεί τα μηνύματα που παραλαμβάνει κατά την εκτέλεση της εφαρμογής.

Διεπαφή GossipSubstrate

Από τα πιο βασικά συστατικά στοιχεία μίας εφαρμογής GossipLib είναι η διεπαφή GossipSubstrate η οποία αντιπροσωπεύει το υπόστρωμα της εφαρμογής πληροφόρησης. Η διεπαφή αυτή περιλαμβάνει συναρτήσεις για τη δημιουργία μιας νέας εφαρμογής και για την εκτέλεση των λειτουργιών επικοινωνίας.

Για την αποστολή μηνυμάτων υπάρχουν οι πιο κάτω συναρτήσεις:

```
int gossip(SendStartedListener l, ApplId applId, Serializable m, Peer p);
```

η οποία στέλλει το σειριοποιημένο μήνυμα ‘m’ στον κόμβο ‘p’ της εφαρμογής ‘applID’. Επιστρέφει τον αριθμό των μηνυμάτων που στάλθηκαν.

```
int gossip(SendStartedListener l, ApplId applId, Serializable m, PeerSelection s);
```

η οποία στέλλει το σειριοποιημένο μήνυμα ‘m’ στους κόμβους του συνόλου ‘s’ της εφαρμογής ‘applID’. Επιστρέφει τον αριθμό των μηνυμάτων που στάλθηκαν.

```
int reliableGossip(SendStartedListener l, ApplId applId, Serializable m, Peer p);
```

η οποία στέλλει το σειριοποιημένο μήνυμα ‘m’ στον κόμβο ‘p’ της εφαρμογής ‘applID’ χρησιμοποιώντας το πρωτόκολλο TCP. Επιστρέφει τον αριθμό των μηνυμάτων που στάλθηκαν.

```
int reliableGossip(SendStartedListener l, ApplId applId, Serializable m, PeerSelection s);
```

η οποία στέλλει το σειριοποιημένο μήνυμα ‘m’ στους κόμβους του συνόλου ‘s’ της εφαρμογής ‘applID’ χρησιμοποιώντας το πρωτόκολλο TCP. Επιστρέφει τον αριθμό των μηνυμάτων που στάλθηκαν.

Gossip Cycle Listener

Μία εφαρμογή της GossipLib αποτελείται από κύκλους πληροφόρησης και κάθε κύκλος αποτελείται από δύο βασικά μέρη, το ενεργό και το παθητικό. Σε κάθε κύκλο της εφαρμογής δημιουργείται ένας καινούριος GossipCycleListener ο οποίος είναι υπεύθυνος να αφουγκράζεται τα γεγονότα που συμβαίνουν στον συγκεκριμένο κύκλο στην εφαρμογή. Για το σκοπό αυτό υπάρχει η διεπαφή GossipCycleListener και η συνάρτησή της

```
void newGossipCycle(GossipApplication appl);
```

η οποία δημιουργεί ένα νέο κύκλο πληροφόρησης στην εφαρμογή ‘appl’.

2.4.2 Εκτέλεση μίας εφαρμογής της βιβλιοθήκης GossipLib

Η βιβλιοθήκη GossipLib βασίζεται στη βιβλιοθήκη YALPS και προσφέρει σε αυτήν κάποιες επιπλέον λειτουργίες. Ο τρόπος με τον οποίο εκτελείται μία εφαρμογή της βιβλιοθήκης GossipLib είναι όμοιος με τον τρόπο που εκτελείται μία εφαρμογή της βιβλιοθήκης YALPS.

2.5 Πλατφόρμα Υλοποίησης και Αξιολόγησης Αλγορίθμων

Για την πειραματική αξιολόγηση των αλγορίθμων χρησιμοποιήθηκε ο προσομοιωτής 'SimulatedNodeStarter' της βιβλιοθήκης YALPS ο οποίος εκτέλεσε τις εφαρμογές σε προσομοίωση.

Η μηχανή στην οποία προσομοιώθηκαν οι εφαρμογές έχει τα πιο κάτω χαρακτηριστικά:

Επεξεργαστής: Intel® Core™ i5 CPU - 460 M με ταχύτητα ρολογιού 2.53GHz

RAM: 4 GB

ΚΕΦΑΛΑΙΟ 3

Αλγόριθμος PUSH

3.1 Περιγραφή Αλγορίθμου

Ο πρώτος αλγόριθμος με τον οποίο ασχολείται η εργασία, τον οποίο καλούμε αλγόριθμο PUSH [1], είναι ένας πολύ απλός αλγόριθμος πληροφόρησης ο οποίος αποδεικνύεται ότι παρά την απλότητά του είναι αρκετά αποδοτικός. Η λειτουργία του αλγορίθμου είναι η εξής: Σε κάθε γύρο επικοινωνίας, ο κάθε κόμβος επιλέγει τυχαία και ομοιόμορφα έναν από τους γείτονές του και στέλλει σε αυτόν όλες τις πληροφορίες που γνωρίζει μέχρι τον τρέχοντα γύρο επικοινωνίας. Οι κόμβοι του συστήματος γνωρίζουν το συνολικό αριθμό πληροφοριών που υπάρχουν στο δίκτυο και έτσι μπορούν να αποφασίσουν σε κάθε γύρο επικοινωνίας αν έχουν παραλάβει όλες τις πληροφορίες του δικτύου ή όχι. Όταν ένας κόμβος μάθει όλες τις πληροφορίες σταματά να στέλλει στους γείτονές του αυτά που γνωρίζει και μόνο στην περίπτωση που θα παραλάβει μήνυμα από κάποιον γείτονά του θα του απαντήσει με όσα γνωρίζει, δηλαδή με όλες τις πληροφορίες του δικτύου. Με αυτό τον τρόπο όταν όλοι οι κόμβοι μάθουν όλες τις πληροφορίες θα σταματήσουν να ανταλλάσουν μηνύματα και ο αλγόριθμος θα τερματίσει.

3.2 Υλοποίηση Αλγορίθμου

Αρχικά δηλώνονται οι κόμβοι του συστήματος και στον καθένα ανατίθεται μία διαφορετική πληροφορία η οποία θα πρέπει με το τέλος του αλγορίθμου να έχει μεταδοθεί σε όλους τους υπόλοιπους κόμβους. Ο αλγόριθμος εκτελείται σε γύρους

επικοινωνίας και κάθε γύρος επικοινωνίας αποτελείται από το ενεργό και το παθητικό μέρος.

3.2.1 Ενεργό μέρος

Το ενεργό μέρος της εφαρμογής αποτελείται από τη συνάρτηση

```
protected void periodicRun() {

}
```

Στην αρχή του κάθε γύρου επικοινωνίας, ο κάθε κόμβος ενημερώνει τις πληροφορίες που γνωρίζει προσθέτοντας σε αυτές τις πληροφορίες που παρέλαβε στον προηγούμενο γύρο. Οι πληροφορίες που παραλαμβάνει ένας κόμβος σε κάποιο γύρο επικοινωνίας δεν μπορούν να μεταδοθούν από τον παραλήπτη μέσα στον ίδιο γύρο, επειδή η επικοινωνία των κόμβων του συστήματος σε ένα γύρο γίνεται παράλληλα. Γι' αυτό το λόγο οι πληροφορίες που παραλαμβάνει ένας κόμβος δεν προστίθενται αμέσως στο χώρο της μνήμης όπου ο κόμβος κρατά τις πληροφορίες που γνωρίζει αλλά σε κάποιο άλλο χώρο της μνήμης του. Έτσι, πριν ξεκινήσει η ουσιαστική διαδικασία του ενεργού μέρους του κάθε γύρου επικοινωνίας, ο κόμβος πρέπει να ενημερώσει τις πληροφορίες που γνωρίζει προσθέτοντας σε αυτές τις πληροφορίες που παρέλαβε στον προηγούμενο γύρο. Για κάθε πληροφορία που παρέλαβε στον προηγούμενο γύρο θα ελέγξει κατά πόσο γνωρίζει ήδη την πληροφορία αυτή, διαφορετικά θα την προσθέσει στη λίστα με τις πληροφορίες του. Πιο κάτω φαίνεται ο τρόπος που οργανώνει τα περιεχόμενά του ο κάθε κόμβος:

```
class NodeContent implements Serializable {
  private static final long serialVersionUID = 1L;
  ArrayList<String> rumors;
  ArrayList<String> tempRumors;
  . . .
}
```

όπου στη λίστα 'rumors' ο κάθε κόμβος φυλάει τις πληροφορίες που γνωρίζει και που μπορεί να μεταδώσει στον τρέχοντα γύρο, ενώ στη λίστα 'tempRumors' φυλάει τις πληροφορίες που παραλαμβάνει στον τρέχοντα γύρο επικοινωνίας και οι οποίες θα μπορούν να μεταδοθούν από τον επόμενο γύρο.

Αφού ολοκληρώθηκε η ενημέρωση των πληροφοριών των κόμβων, ο κάθε κόμβος θα πρέπει να ελέγξει κατά πόσον γνωρίζει όλες τις πληροφορίες του συστήματος. Αυτό μπορεί να γίνει γιατί οι πληροφορίες είναι στατικές και ο συνολικός αριθμός των πληροφοριών του συστήματος είναι γνωστός. Αν κάποιος κόμβος δεν γνωρίζει όλες τις πληροφορίες θα συνεχίσει με την εκτέλεση του ενεργού μέρους του γύρου διαφορετικά θα το αγνοήσει και θα εξέλθει από τη συνάρτηση. Το κομμάτι κώδικα που αφορά αυτό το σημείο είναι

```
if (myContent.rumors.size()==numOfPeers)
   return;
```

Όπου ‘numOfPeers’ ισούται με το συνολικό αριθμό πληροφοριών του συστήματος αφού κάθε κόμβος ‘peer’ έχει αρχικά μία πληροφορία στην προσωπική του μνήμη. Το αντικείμενο ‘myContent’ αντιπροσωπεύει τα περιεχόμενα του κόμβου.

Οι κόμβοι που δεν γνωρίζουν όλες τις πληροφορίες, συνεχίζουν δημιουργώντας το μήνυμα που θα μεταδώσουν σε αυτό το γύρο. Ο κάθε κόμβος βάζει στο μήνυμά του όλες τις πληροφορίες που έμαθε μέχρι και τον προηγούμενο γύρο επικοινωνίας. Το κομμάτι κώδικα για αυτό το σημείο είναι

```
data.rumors=(ArrayList<String>) myContent.rumors.clone();
```

όπου ‘data’ το μήνυμα που θα σταλεί και η συνάρτηση ‘myContent.rumors.clone()’ αντιγράφει στη μεταβλητή ‘rumors’ του μηνύματος όλες τις πληροφορίες που γνωρίζει ο κόμβος.

Αφού έχουν δημιουργηθεί τα μηνύματα που θα σταλούν σε αυτό το γύρο, μένει να επιλέξει ο κάθε κόμβος του συστήματος τον γείτονά του με τον οποίο θα επικοινωνήσει και θα του στείλει το μήνυμα. Η επιλογή γίνεται με τυχαίο τρόπο χρησιμοποιώντας την κλάση Random της βιβλιοθήκης Java. Το κομμάτι κώδικα που αφορά αυτό το σημείο είναι

```
int size=peers.size();

int rand=new Random().nextInt(size);

Peer p=peers.get(rand);
```

Όπου ‘peers’ η λίστα με τους κόμβους του συστήματος, η συνάρτηση ‘peers.size()’ επιστρέφει τον αριθμό των κόμβων του συστήματος και η συνάρτηση ‘peers.get(rand)’ επιστρέφει τον κόμβο που βρίσκεται στην θέση ‘rand’ της λίστας με τους κόμβους.

Τέλος, για να ολοκληρωθεί το ενεργό μέρος του γύρου επικοινωνίας μένει να στείλουν οι κόμβοι το μήνυμά τους στον γείτονα που επέλεξαν. Το κομμάτι κώδικα που αφορά αυτό το σημείο είναι

```
getCommLayer().sendUDP(data, neighbour);
```

Όπου ‘data’ το μήνυμα που θα σταλεί και ‘neighbour’ ο γείτονας στον οποίο θα σταλεί το μήνυμα.

3.2.2 Παθητικό μέρος

Το παθητικό μέρος της εφαρμογής αποτελείται από τη συνάρτηση

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

}
```

Ένας κόμβος εισέρχεται αυτόματα στο παθητικό μέρος του γύρου επικοινωνίας όταν παραλάβει κάποιο μήνυμα. Οι κόμβοι που έχουν όλες τις πληροφορίες του συστήματος δεν χρειάζεται να επεξεργαστούν το μήνυμα που παρέλαβαν. Το μόνο που κάνουν είναι να απαντήσουν στον αποστολέα του μηνύματος με όλες τις πληροφορίες που γνωρίζουν έτσι ώστε να τον βοηθήσουν και αυτόν να μάθει όλες τις πληροφορίες και να σταματήσει τη μετάδοση μηνυμάτων. Το κομμάτι κώδικα που αφορά αυτό το σημείο είναι

```
if (rumorList.size()==numOfPeers){
   RumorsMessageSt data = new RumorsMessageSt();
   data.rumors = (ArrayList<String>) rumorList.clone();
   getCommLayer().sendUDP(data, sender);
   return true;
}
```

Όπου ‘`rumorList`’ η λίστα με τις πληροφορίες του παραλήπτη και η συνάρτηση ‘`rumorList.clone()`’ αντιγράφει τις πληροφορίες του κόμβου παραλήπτη σε μία νέα λίστα την οποία θα στείλει πίσω στον αποστολέα. Στη συνέχεια εξέρχεται από τη συνάρτηση κα τελειώνει το παθητικό μέρος του γύρου.

Αν ο παραλήπτης δεν έχει όλες τις πληροφορίες του συστήματος τότε θα επεξεργαστεί το μήνυμα που παρέλαβε. Για κάθε πληροφορία του μηνύματος θα ελέγξει αν έχει ήδη παραλάβει την πληροφορία αυτή στον τρέχοντα γύρο, διαφορετικά θα την φυλάξει στη λίστα με τις πληροφορίες που παρέλαβε σε αυτό το γύρο, και όχι στη λίστα με όλες τις πληροφορίες που έχει ο κόμβος. Αφού τελειώσει με την επεξεργασία όλων των πληροφοριών του μηνύματος, εξέρχεται από τη συνάρτηση κα τελειώνει το παθητικό μέρος του γύρου. Η σύμπτυξη των πληροφοριών που παρέλαβε ο κόμβος σε αυτό τον γύρο με τις πληροφορίες που έμαθε ο κόμβος στους προηγούμενους γύρους θα γίνει στο ενεργό μέρος του επόμενου γύρου επικοινωνίας.

3.3 Πειραματική Αξιολόγηση Αλγορίθμου

Αφού υλοποιήθηκε ο αλγόριθμος συνεχίζουμε με την προσομοίωση και την πειραματική του αξιολόγηση.

3.3.1 Μεθοδολογία Πειραματικής Αξιολόγησης

Επειδή το περιβάλλον στο οποίο έτρεξε η εφαρμογή δεν είναι πραγματικό, υπήρξαν κάποιοι περιορισμοί στην προσομοίωση της εφαρμογής. Αυτό οφείλεται στο ότι κατά την εκτέλεση της εφαρμογής, όλοι οι κόμβοι του συστήματος τρέχουν τις λειτουργίες τους πάνω στην ίδια μηχανή. Συνεπώς, ο αριθμός των κόμβων της εφαρμογής περιορίζεται λόγω της περιορισμένης μνήμης της μηχανής καθώς και λόγω των δυνατοτήτων του διαθέσιμου επεξεργαστή. Ο μέγιστος αριθμός κόμβων που μπορέσαμε να χρησιμοποιήσουμε στην προσομοίωση της συγκεκριμένης εφαρμογής ήταν 300 κόμβοι. Προσομοιώσαμε την εφαρμογή για 20, 40, 60, …, 300 κόμβους. Αυτό επαναλήφθηκε πέντε φορές για να παρθεί στο τέλος ο μέσος όρος των αντίστοιχων περιπτώσεων προσομοίωσης έτσι ώστε οι τελικές μετρήσεις της προσομοίωσης να είναι πιο ρεαλιστικές.

Λόγω του ότι το σύστημα είναι κατανεμημένο και ο κάθε κόμβος στο δίκτυο έχει μόνο τοπική γνώση, δεν μπορεί ένας κόμβος να γνωρίζει τη στιγμή που ενημερώθηκαν όλοι οι κόμβοι στο δίκτυο. Γι' αυτό το λόγο δεν μπορεί κάποιος κόμβος να σταματήσει να λειτουργεί αφού μπορεί ανά πάσα στιγμή να παραλάβει μήνυμα από κάποιον κόμβο ο οποίος δεν έχει ακόμη ενημερωθεί πλήρως. Συνεπώς ο αλγόριθμος PUSH δεν τερματίζει μόλις ολοκληρώσει τη λειτουργία του. Αντί αυτού καθορίζω εγώ το γύρο επικοινωνίας στον οποίο θέλω να τερματίσει ο αλγόριθμος. Ο γύρος αυτός αποφάσισα να είναι ο γύρος 20 επειδή μετά από αρκετές προσομοιώσεις του αλγορίθμου παρατήρησα ότι ο αλγόριθμος ολοκλήρωνε τη λειτουργία του σε 14 το πολύ γύρους επικοινωνίας. Ο γύρος στον οποίο καταλαβαίνω ότι ο αλγόριθμος ολοκλήρωσε τη λειτουργία του είναι ο γύρος στον οποίο δεν στάλθηκε κανένα μήνυμα. Για να μην έχει σταλθεί κανένα μήνυμα σε κάποιο γύρο έπεται ότι όλοι οι κόμβοι έχουν ενημερωθεί αφού από τη λειτουργία του αλγορίθμου σε κάθε γύρο επικοινωνίας όλοι οι κόμβοι που δεν έχουν ενημερωθεί πλήρως στέλλουν μήνυμα στον κόμβο που θα επιλέξουν τον τρέχοντα γύρο. Επομένως αφού βρω το γύρο επικοινωνίας στον οποίο ο αλγόριθμος ολοκλήρωσε τη λειτουργία του παίρνω τις αντίστοιχες μετρήσεις προσομοίωσης.

3.3.2 Αποτελέσματα Πειραματικής Αξιολόγησης

Θεωρητικά ο αλγόριθμος ενημερώνει όλους τους κόμβους του συστήματος με όλες τις πληροφορίες σε $O(\ln n)$ γύρους επικοινωνίας και χρησιμοποιώντας $O(n \ln n)$ μηνύματα, όπου n ο αριθμός των κόμβων. Τα δύο αυτά άνω φράγματα αποδείχθηκαν αυστηρότυπα στο [1]. Σκοπός της εργασίας μας είναι να συμπεράνουμε κατά πόσον οι θεωρητικές αυτές μετρήσεις επιβεβαιώνονται στην πράξη.

Ακολουθούν οι τελικές μετρήσεις της προσομοίωσης του αλγορίθμου που πάρθηκαν από το μέσο όρο των πέντε προσομοιώσεων της κάθε μιας από τις πιο κάτω περιπτώσεις. Η στήλη 'Nodes' αντιπροσωπεύει τον αριθμό των κόμβων στην προσομοίωση. Η στήλη 'Rounds' αντιπροσωπεύει τον αριθμό των γύρων επικοινωνίας που χρειάστηκε να τρέξει ο αλγόριθμος μέχρι να ολοκληρώσει την εκτέλεσή του, δηλαδή είναι ο γύρος στον οποίο στάλθηκαν τα τελευταία μηνύματα στην προσομοίωση. Η στήλη 'Time (msec)' αντιπροσωπεύει το χρόνο (σε χιλιοστά του δευτερολέπτου) που πήρε για να ολοκληρωθεί ο αλγόριθμος, δηλαδή μετριέται από την αρχή της εκτέλεσης της προσομοίωσης μέχρι το τέλος του γύρου στον οποίο

ολοκληρώνεται ο αλγόριθμος. Η στήλη ‘Messages’ αντιπροσωπεύει το συνολικό αριθμό μηνυμάτων που στάλθηκαν κατά τη διάρκεια της προσομοίωσης, δηλαδή είναι ο αριθμός μηνυμάτων που αποστέλλονται από τον πρώτο γύρο προσομοίωσης μέχρι και τον γύρο στον οποίο ολοκληρώνεται ο αλγόριθμος.

Nodes	Rounds	Time (msec)	Messages
20	9,00	105,80	145,40
40	10,00	153,20	337,00
60	11,20	230,80	550,20
80	11,80	318,20	782,40
100	12,00	452,40	1028,20
120	12,60	577,80	1264,80
140	12,80	790,60	1513,20
160	13,20	974,60	1770,80
180	13,20	1311,40	2023,00
200	13,40	1651,80	2292,60
220	13,60	1961,20	2559,20
240	13,80	2504,80	2825,40
260	14,00	3066,60	3101,80
280	14,00	3600,60	3365,60
300	14,00	4556,80	3639,40

Πίνακας 3.1 Αλγόριθμος PUSH

3.3.3 Ανάλυση αποτελεσμάτων

Οι μετρήσεις που πάρθηκαν από την προσομοίωση του αλγορίθμου οδήγησαν στις πιο κάτω παρατηρήσεις ως προς τις τρεις παραμέτρους αξιολόγησης του αλγορίθμου, τους γύρους επικοινωνίας, το χρόνο εκτέλεσης και τον αριθμό μηνυμάτων:

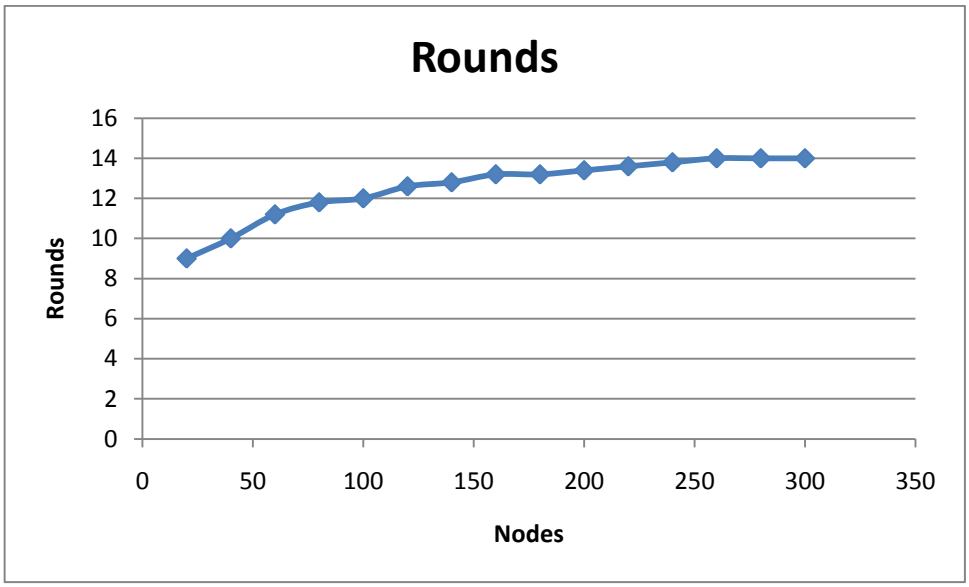
Γύροι Επικοινωνίας

Θεωρητικά ο αριθμός των γύρων επικοινωνίας στην εκτέλεση του αλγορίθμου είναι λογαριθμικός συναρτήσει των κόμβων τους συστήματος και είναι τάξεως $O(\ln n)$, όπου n ο αριθμός των κόμβων.

Από τον πίνακα 3.1 παρατηρούμε ότι ο αριθμός των γύρων επικοινωνίας που τρέχει ο αλγόριθμος μέχρι να ενημερωθούν όλοι οι κόμβοι μεγαλώνει καθώς μεγαλώνει ο αριθμός των κόμβων του συστήματος. Ο ρυθμός όμως με τον οποίο αυξάνονται οι

γύροι επικοινωνίας δεν είναι μεγάλος. Συγκεκριμένα παρατηρούμε ότι όσο ο αριθμός των κόμβων είναι μικρός, ο αριθμός των γύρων αυξάνεται με μεγαλύτερο ρυθμό ενώ όσο ο αριθμός των κόμβων μεγαλώνει, ο ρυθμός αύξησης των γύρων επικοινωνίας μικραίνει. Για παράδειγμα όταν οι κόμβοι αυξήθηκαν από 20 σε 100 έχουμε αύξηση στους γύρους επικοινωνίας ίση με 3 γύρους, ενώ όταν οι κόμβοι αυξάνονται από 220 σε 300 η αύξηση στους γύρους επικοινωνίας είναι πολύ πιο μικρή αφού είναι ίση με 0,4 γύρους. Αυτή η συμπεριφορά μοιάζει να είναι λογαριθμική επομένως αναμένουμε μία λογαριθμική γραφική παράσταση η οποία θα επιβεβαιώνει το θεωρητικό άνω φράγμα για τους γύρους επικοινωνίας που είναι $O(\ln n)$.

Πιο κάτω φαίνεται η γραφική παράσταση του αριθμού των γύρων επικοινωνίας συναρτήσει του αριθμού των κόμβων του συστήματος:



Σχήμα 3.1 Αλγόριθμος PUSH – Γύροι Επικοινωνίας

Όντως, η γραφική παράσταση που πάρθηκε από τις μετρήσεις για τους γύρους επικοινωνίας είναι λογαριθμική σε σχέση με τον αριθμό των κόμβων. Αυτό συναύει με το θεωρητικό άνω φράγμα $O(\ln n)$ για τους γύρους επικοινωνίας που χρειάζεται να τρέξει ο αλγόριθμος για να ενημερωθούν όλοι οι κόμβοι του συστήματος.

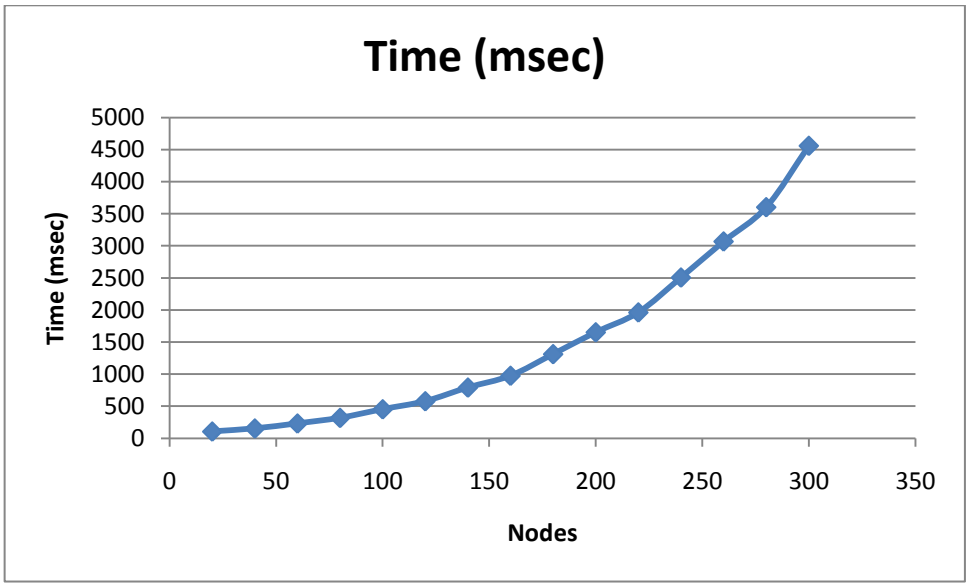
Χρόνος Εκτέλεσης

Ο χρόνος εκτέλεσης του αλγορίθμου αναμέναμε να είναι ανάλογος των γύρων επικοινωνίας. Για να συμβεί όμως αυτό πρέπει οι κόμβοι του συστήματος να

σπαταλούν σταθερό χρόνο σε κάθε γύρο επικοινωνίας. Αυτό συμβαίνει μόνο στην περίπτωση που οι κόμβοι δεν επεξεργάζονται καθόλου τις πληροφορίες που παραλαμβάνουν. Επειδή όμως για τους σκοπούς της εφαρμογής μας οι κόμβοι του συστήματος επεξεργάζονται όλες τις πληροφορίες που παραλαμβάνουν, ο χρόνος που χρειάζεται ο κάθε γύρος επικοινωνίας δεν είναι σταθερός αλλά εξαρτάται από το πλήθος των πληροφοριών που παραλαμβάνει ο κάθε κόμβος στον τρέχοντα γύρο. Συνεπώς ο χρόνος εκτέλεσης του αλγορίθμου δεν αναμένεται να είναι λογαριθμικός ως προς των αριθμό των κόμβων, όπως συμβαίνει με τους γύρους επικοινωνίας, αλλά αναμένεται να είναι πολύ μεγαλύτερος.

Όπως παρατηρούμε στον πίνακα 3.1, ο χρόνος εκτέλεσης μεγαλώνει όσο μεγαλώνει ο αριθμός των κόμβων. Ο ρυθμός όμως με τον οποίο αυξάνεται ο χρόνος εκτέλεσης του αλγορίθμου δεν είναι σταθερός, αλλά μεγαλώνει όσο αυξάνονται οι κόμβοι στο σύστημα. Για παράδειγμα όταν οι κόμβοι αυξήθηκαν από 20 σε 100 έχουμε αύξηση στον χρόνο εκτέλεσης ίση με 346,6 χιλιοστά του δευτερολέπτου, ενώ όταν οι κόμβοι αυξάνονται από 220 σε 300 η αύξηση στον χρόνο εκτέλεσης είναι πολύ πιο μεγάλη αφού είναι ίση με 2595,60 χιλιοστά του δευτερολέπτου. Επομένως, ο χρόνος εκτέλεσης του αλγορίθμου αυξάνεται με μεγάλους ρυθμούς.

Πιο κάτω φαίνεται η γραφική παράσταση του χρόνου εκτέλεσης του αλγορίθμου συναρτήσει του αριθμού των κόμβων του συστήματος:



Σχήμα 3.2 Αλγόριθμος PUSH – Χρόνος Εκτέλεσης

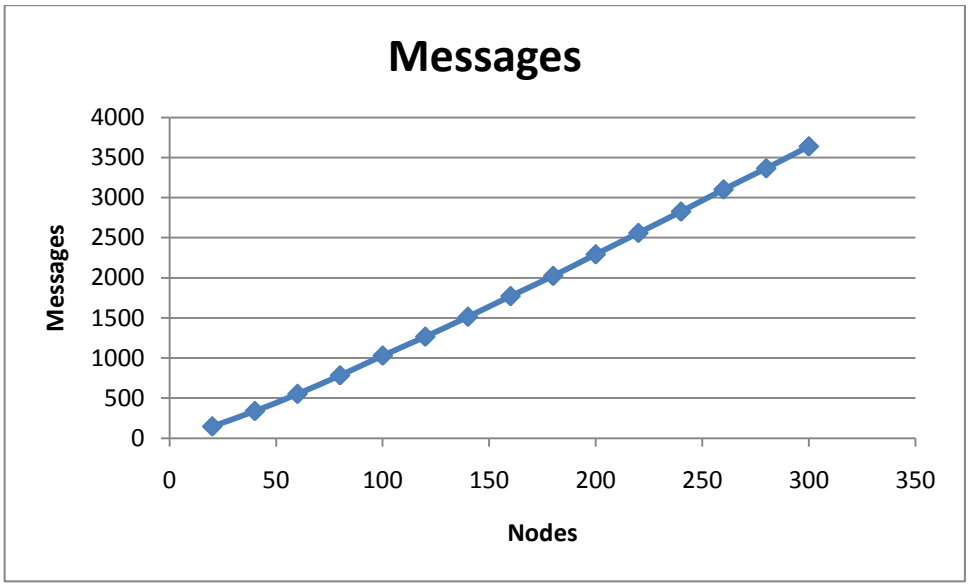
Όντως, όπως φαίνεται και στη γραφική παράσταση, ο χρόνος εκτέλεσης του αλγορίθμου δεν αυξάνεται λογαριθμικά ως προς τον αριθμό των κόμβων του συστήματος, όπως συμβαίνει με τους γύρους επικοινωνίας, και ούτε αυξάνεται γραμμικά, όπως συμβαίνει με τον αριθμό μηνυμάτων. Ο χρόνος εκτέλεσης αυξάνεται με μεγαλύτερους ρυθμούς οι οποίοι ξεπερνούν το άνω φράγμα $O(n \log n)$, όπου n ο αριθμός των κόμβων. Αυτό οφείλεται στο χρόνο που σπαταλούν οι κόμβοι σε κάθε γύρο επικοινωνίας για να επεξεργαστούν τις πληροφορίες που παραλαμβάνουν αλλά και για την εκτέλεση άλλων λειτουργιών.

Μηνύματα

Θεωρητικά ο συνολικός αριθμός μηνυμάτων που αποστέλλονται κατά τη διάρκεια εκτέλεσης του αλγορίθμου είναι ανάλογος του αριθμού των κόμβων του συστήματος και συγκεκριμένα είναι της τάξεως $O(n \ln n)$, όπου n ο αριθμός των κόμβων.

Από τον πίνακα 3.1 παρατηρούμε ότι ο αριθμός των μηνυμάτων μεγαλώνει καθώς μεγαλώνει ο αριθμός των κόμβων του συστήματος. Αυτό είναι λογικό αφού για να ενημερωθούν περισσότεροι κόμβοι θα πρέπει να σταλούν περισσότερα μηνύματα. Για παράδειγμα όταν οι κόμβοι αυξήθηκαν από 20 σε 100 έχουμε αύξηση στον αριθμό των μηνυμάτων ίση με 882,8 μηνύματα, ενώ όταν οι κόμβοι αυξάνονται από 220 σε 300 η αύξηση στον αριθμό των μηνυμάτων είναι ίση με 1080,2 μηνύματα. Επομένως παρατηρούμε ότι ο ρυθμός αύξησης του αριθμού των μηνυμάτων σε συνάρτηση με τους κόμβους είναι περίπου σταθερός και ότι ο αριθμός των μηνυμάτων είναι περίπου ανάλογος του αριθμού των κόμβων. Αυτή η συμπεριφορά μοιάζει να είναι γραμμική επομένως αναμένουμε μία γραφική παράσταση η οποία θα επιβεβαιώνει το θεωρητικό άνω φράγμα για τον αριθμό των μηνυμάτων που είναι $O(n \ln n)$.

Πιο κάτω φαίνεται η γραφική παράσταση του συνολικού αριθμού των μηνυμάτων συναρτήσει του αριθμού των κόμβων του συστήματος:



Σχήμα 3.3 Αλγόριθμος PUSH – Αριθμός Μηνυμάτων

Όντως, η γραφική παράσταση που πάρθηκε από τις μετρήσεις για τον αριθμό μηνυμάτων είναι σχεδόν γραμμική σε σχέση με τον αριθμό των κόμβων. Φαίνεται να υπάρχει μία πολύ μικρή καμπύλη στην ευθεία της γραφικής παράστασης η οποία όμως ήταν αναμενόμενη αφού από τις μετρήσεις μας ο αριθμός των μηνυμάτων δεν ήταν ακριβώς ανάλογος του αριθμού των κόμβων, αλλά είχε μία μικρή διαφορά. Η διαφορά αυτή δημιουργεί την μικρή καμπύλη που υπάρχει στην γραφική και δικαιολογεί το ‘ln n’ στο θεωρητικό άνω φράγμα. Συνεπώς, επιβεβαιώνεται στην πράξη το θεωρητικό άνω φράγμα $O(n \ln n)$ για το συνολικό αριθμό μηνυμάτων που χρειάζεται να σταλούν κατά την εκτέλεση του αλγορίθμου έτσι ώστε να ενημερωθούν όλοι οι κόμβοι του συστήματος.

3.4 Συμπεράσματα

Αφού μελετήσαμε ξεχωριστά τις βασικές παραμέτρους του αλγορίθμου, καταλήγουμε στα πιο κάτω συμπεράσματα:

Καθώς αυξάνεται ο αριθμός των κόμβων στην προσομοίωση, αυξάνονται οι γύροι επικοινωνίας που πρέπει να τρέξει ο αλγόριθμος ώστε να ολοκληρώσει τη λειτουργία του, αυξάνεται ο χρόνος εκτέλεσης του αλγορίθμου και τέλος αυξάνεται και ο συνολικός αριθμός των μηνυμάτων που ανταλλάσσονται. Κάθε μια από τις τρεις αυτές παραμέτρους αυξάνεται με διαφορετικό ρυθμό και μας οδηγεί σε διαφορετικά συμπεράσματα.

Ο αριθμός των γύρων επικοινωνίας αυξάνεται λογαριθμικά ως προς τον αριθμό των κόμβων. Η λογαριθμική αύξηση των γύρων επικοινωνίας είναι πολύ αποδοτική αφού για μεγάλο αριθμό κόμβων στο σύστημα, η περεταίρω αύξηση στον αριθμό των κόμβων δεν θα οδηγήσει σε μεγάλες αλλαγές στον αριθμό των γύρων επικοινωνίας. Ο χρόνος όμως εκτέλεσης του αλγορίθμου δεν είναι αρκετά καλός αφού είναι αυτός που αυξάνεται με το μεγαλύτερο ρυθμό. Παρόλα αυτά ο χρόνος εκτέλεσης δεν μας απασχολεί τόσο γιατί δεν εξαρτάται μόνο από το χρόνο που χρειάζεται μέχρι να ενημερωθούν όλοι οι κόμβοι του συστήματος αλλά εξαρτάται και από τις επιπρόσθετες λειτουργίες που ο κάθε προγραμματιστής προσθέτει στην εφαρμογή του. Γι' αυτό και για την αξιολόγηση του αλγορίθμου χρησιμοποιείται κυρίως ο αριθμός των γύρων επικοινωνίας αντί ο χρόνος εκτέλεσης.

Ο αριθμός των μηνυμάτων που ανταλλάσσονται κατά τη διάρκεια του αλγορίθμου αυξάνεται με αρκετά καλούς ρυθμούς, σχεδόν γραμμικά, και είναι της τάξεως $O(n \ln n)$, όπου n το πλήθος των κόμβων. Αξίζει όμως να σημειωθεί ότι παρόλο που ο αριθμός των μηνυμάτων φαίνεται να είναι ικανοποιητικός, το μέγεθος των μηνυμάτων αυτών αναμένουμε να είναι πολύ μεγάλο. Αυτό γιατί λόγω της φύσης του αλγορίθμου ο κάθε κόμβος βάζει στο μήνυμα που θα στείλει όλες τις πληροφορίες που γνωρίζει μέχρι τον τρέχοντα γύρο χωρίς να λαμβάνει υπόψη του ότι κάποια πληροφορία μπορεί να μην είναι χρήσιμη στον παραλήπτη. Το μέγεθος των μηνυμάτων του αλγορίθμου θα μελετηθεί στο Κεφάλαιο 7, για να μπορέσει να συγκριθεί με το μέγεθος των μηνυμάτων των υπολοίπων αλγορίθμων και έτσι να οδηγήσει σε εξαγωγή καλύτερων συμπερασμάτων ως προς την παράμετρο αυτή.

Τέλος, οι πιο πάνω παρατηρήσεις μας οδηγούν στο γενικότερο συμπέρασμα ότι ο παρόν αλγόριθμος πληροφόρησης παρά την απλότητα με την οποία λειτουργεί είναι αποτελεσματικός και αρκετά αποδοτικός. Ακόμη καταλήγουμε στο συμπέρασμα ότι η πρακτική ανάλυση του αλγορίθμου συναύει με τη θεωρητική αφού τόσο θεωρητικά όσο και πρακτικά ο αλγόριθμος τερματίζει σε $O(\ln n)$ γύρους επικοινωνίας και χρησιμοποιώντας $O(n \ln n)$ μηνύματα.

ΚΕΦΑΛΑΙΟ 4

Αλγόριθμος PUSH&PULL

4.1 Περιγραφή Αλγορίθμου

Ο αλγόριθμος PUSH&PULL [1] προσπαθεί να μειώσει το χρόνο που χρειάζεται για την ενημέρωση όλων των κόμβων του δικτύου, χρησιμοποιώντας σε κάθε γύρο επικοινωνίας εκτός από τη λειτουργία της ώθησης (push) και τη λειτουργία της λήψης (pull). Κατά τις δύο αυτές λειτουργίες τα μηνύματα που ανταλλάσσονται δεν περιέχουν όλες τις πληροφορίες που γνωρίζει ο αποστολέας αλλά οι πληροφορίες φιλτράρονται έτσι ώστε να μην γίνεται άσκοπη διάδοση πληροφοριών. Το φιλτράρισμα βασίζεται στο γεγονός ότι κάθε πληροφορία έχει προθεσμία μετάδοσης, επομένως από κάποιο σημείο και μετά δεν θα πρέπει να μεταδίδεται. Όταν ένας κόμβος δημιουργεί μία πληροφορία, αρχικοποιεί τον μετρητή που ελέγχει την προθεσμία μετάδοσής της να είναι ίσος με μηδέν. Ο μετρητής αυτός μεταδίδεται πάντα μαζί με την πληροφορία και αυξάνεται σε κάθε γύρο επικοινωνίας. Όσο ο μετρητής προθεσμίας μετάδοσης μίας πληροφορίας είναι μικρότερος από την τιμή της προθεσμίας μετάδοσης, η οποία είναι τάξεως $\log_3 n + O(\ln \ln n)$ [1], η πληροφορία θεωρείται ότι είναι ‘hot’. Σε κάθε γύρο επικοινωνίας ο κάθε κόμβος επιλέγει τυχαία και ομοιόμορφα έναν από τους γείτονές του και στέλλει (push) σε αυτόν όλες τις πληροφορίες που γνωρίζει μέχρι τον συγκεκριμένο γύρο επικοινωνίας και οι οποίες είναι ‘hot’. Επίσης στον ίδιο γύρο επικοινωνίας ο κόμβος αυτός λαμβάνει (pull) από τον ίδιο γείτονα όλες τις ‘hot’ πληροφορίες που έχει. Ο αλγόριθμος τερματίζει όταν η προθεσμία μετάδοσης όλων των πληροφοριών του δικτύου έχει λήξει, οπόταν κανένας κόμβος δεν έχει κάποια ‘hot’ πληροφορία για να ανταλλάξει με κάποιο άλλο κόμβο.

4.2 Υλοποίηση Αλγορίθμου

Αρχικά δηλώνονται οι κόμβοι του συστήματος και στον καθένα ανατίθεται μία διαφορετική πληροφορία με αρχικό μετρητή προθεσμίας μετάδοσης ίσο με μηδέν. Για την αρχικοποίηση του μετρητή προθεσμίας μετάδοσης της πληροφορίας είναι υπεύθυνος ο κατασκευαστής της πληροφορίας στον οποίο υπάρχει ο ακόλουθος κώδικας

```
rumor(String c, Integer round){
   ....
   this.birth=round;
}
```

Όπου 'birth' ο γύρος επικοινωνίας στον οποίο δημιουργήθηκε η πληροφορία και αρχικοποιείται με 'round' δηλαδή με τον αριθμό του τρέχοντα γύρου επικοινωνίας ο οποίος είναι προς το παρόν ίσος με μηδέν.

Ακολούθως, πριν ξεκινήσει η εκτέλεση του αλγορίθμου υπολογίζεται η προθεσμία μετάδοσης μίας πληροφορίας η οποία καθορίζει το χρονικό διάστημα κατά το οποίο μία πληροφορία θα θεωρείται 'hot'.

Ο αλγόριθμος εκτελείται σε γύρους επικοινωνίας και κάθε γύρος επικοινωνίας αποτελείται από το ενεργό και το παθητικό μέρος.

4.2.1 Ενεργό μέρος

Το ενεργό μέρος της εφαρμογής αποτελείται από τη συνάρτηση

```
protected void periodicRun() {
}
```

Στην αρχή του κάθε γύρου επικοινωνίας, ο κάθε κόμβος ενημερώνει τις πληροφορίες που γνωρίζει προσθέτοντας σε αυτές τις πληροφορίες που παρέλαβε στον προηγούμενο γύρο. Οι πληροφορίες που παραλαμβάνει ένας κόμβος σε κάποιο γύρο επικοινωνίας δεν μπορούν να μεταδοθούν από τον παραλήπτη μέσα στον ίδιο γύρο επειδή η επικοινωνία των κόμβων του συστήματος σε ένα γύρο γίνεται παράλληλα. Γι' αυτό το λόγο οι πληροφορίες που παραλαμβάνει ένας κόμβος δεν προστίθενται αμέσως στο χώρο της

μνήμης όπου ο κόμβος κρατά τις πληροφορίες που γνωρίζει αλλά σε κάποιο άλλο χώρο της μνήμης του. Έτσι, πριν ξεκινήσει η ουσιαστική διαδικασία του ενεργού μέρους του κάθε γύρου επικοινωνίας, ο κόμβος πρέπει να ενημερώσει τις πληροφορίες που γνωρίζει προσθέτοντας σε αυτές τις πληροφορίες που παρέλαβε στον προηγούμενο γύρο. Για κάθε πληροφορία που παρέλαβε στον προηγούμενο γύρο θα ελέγξει κατά πόσο γνωρίζει ήδη την πληροφορία αυτή, διαφορετικά θα την προσθέσει στη λίστα με τις πληροφορίες του. Πιο κάτω φαίνεται ο τρόπος που οργανώνει τα περιεχόμενά του ο κάθε κόμβος:

```
class NodeContentListDynamic implements Serializable {
   private static final long serialVersionUID = 1L;
   ArrayList<rumor> rumors;
   ArrayList<rumor> tempRumors;
   ArrayList<rumor> rumorsToSend;
   . . .
}
```

όπου στη λίστα ‘rumors’ ο κάθε κόμβος φυλάει τις πληροφορίες που γνωρίζει και που μπορεί να μεταδώσει στον τρέχοντα γύρο, ενώ στη λίστα ‘tempRumors’ φυλάει τις πληροφορίες που παραλαμβάνει στον τρέχοντα γύρο επικοινωνίας και οι οποίες θα μπορούν να μεταδοθούν από τον επόμενο γύρο. Στη λίστα ‘rumorsToSend’ φυλάει τις πληροφορίες που βρίσκονται στη λίστα ‘rumors’ και οι οποίες είναι ‘hot’, δηλαδή τις πληροφορίες που θα μεταδόσει στον τρέχοντα γύρο.

Αφού ολοκληρώθηκε η ενημέρωση των πληροφοριών των κόμβων, θα πρέπει ο κάθε κόμβος να ελέγξει αν έχει ‘hot’ πληροφορίες να διαδώσει. Για να γίνει αυτό ο κάθε κόμβος ελέγχει την προθεσμίας μετάδοσης της κάθε πληροφορίας που γνωρίζει και όποια πληροφορία είναι ‘hot’ την προσθέτει σε μία ειδική λίστα με το όνομα ‘rumorsToSend’. Το κομμάτι κώδικα για τον έλεγχο της λήξης της προθεσμίας μετάδοσης μίας πληροφορίας είναι

```
if ((round-this.birth)<=maxAge)
   return true;
return false;
```

Όπου ‘round’ ο τρέχοντας γύρος επικοινωνίας, ‘birth’ ο γύρος στον οποίο δημιουργήθηκε η πληροφορία και ‘maxAge’ η τιμή της προθεσμίας μετάδοσης.

Οι κόμβοι που δεν έχουν ‘hot’ πληροφορίες για να στείλουν θα βγουν από το ενεργό μέρος της εφαρμογής ενώ οι υπόλοιποι συνεχίζουν την εκτέλεση του ενεργού μέρους δημιουργώντας το μήνυμα που θα στείλουν σε αυτό το γύρο κατά τη διαδικασία push. Ο κάθε κόμβος βάζει στο μήνυμά του όλες τις πληροφορίες που έμαθε μέχρι και τον προηγούμενο γύρο επικοινωνίας και οι οποίες είναι ‘hot’. Το κομμάτι κώδικα για αυτό το σημείο είναι

```
data.rumors=(ArrayList<rumor>) myContent.rumorsToSend.clone();

data.pull=false;
```

όπου ‘data’ το μήνυμα που θα σταλεί και η συνάρτηση ‘myContent.rumorsToSend.clone()’ αντιγράφει στη μεταβλητή ‘rumors’ του μηνύματος όλες τις ‘hot’ πληροφορίες του κόμβου. Επίσης, η μεταβλητή ‘pull’ του μηνύματος παίρνει την τιμή ‘**false**’ η οποία δηλώνει ότι το μήνυμα αποστέλλεται κατά τη διαδικασία push της εφαρμογής.

Αφού έχουν δημιουργηθεί τα μηνύματα που θα σταλούν σε αυτό το γύρο, μένει να επιλέξει ο κάθε κόμβος του συστήματος τον γείτονά του με τον οποίο θα επικοινωνήσει και θα του στείλει το μήνυμα. Η επιλογή γίνεται με τυχαίο τρόπο χρησιμοποιώντας την κλάση Random της βιβλιοθήκης Java. Το κομμάτι κώδικα που αφορά αυτό το σημείο είναι

```
int size=peers.size();

int rand=new Random().nextInt(size);

Peer p=peers.get(rand);
```

Όπου ‘peers’ η λίστα με τους κόμβους του συστήματος, η συνάρτηση ‘peers.size()’ επιστρέφει τον αριθμό των κόμβων του συστήματος και η συνάρτηση ‘peers.get(rand)’ επιστρέφει τον κόμβο που βρίσκεται στην θέση ‘rand’ της λίστας με τους κόμβους.

Τέλος, για να τελειώσει το ενεργό μέρος του γύρου επικοινωνίας μένει να στείλουν οι κόμβοι το μήνυμά τους στον γείτονα που επέλεξαν. Το κομμάτι κώδικα που αφορά αυτό το σημείο είναι

```
getCommLayer().sendUDP(data, neighbour);
```

Όπου 'data' το μήνυμα που θα σταλεί και 'neighbour' ο γείτονας στον οποίο θα σταλεί το μήνυμα.

4.2.2 Παθητικό μέρος

Το παθητικό μέρος της εφαρμογής αποτελείται από τη συνάρτηση

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

}
```

Ένας κόμβος εισέρχεται αυτόματα στο παθητικό μέρος του γύρου επικοινωνίας όταν παραλάβει κάποιο μήνυμα. Η πρώτη ενέργεια του κάθε παραλήπτη είναι να ελέγξει αν ο αποστολέας του μηνύματος θέλει και αυτός να ανακτήσει τις 'hot' πληροφορίες που γνωρίζει ο παραλήπτης, αν απαιτείται δηλαδή η εκτέλεση διαδικασίας pull. Αν ναι, ο παραλήπτης θα ελέγξει αν έχει 'hot' πληροφορίες και θα τις στείλει πίσω στον αποστολέα. Αυτό γίνεται με το ακόλουθο κομμάτι κώδικα

```
if (m.pull ==false){

data.rumors= (ArrayList<rumor>) receiverContent.rumorsToSend.clone();

data.pull =true;

getCommLayer().sendUDP(data, sender);

}
```

Όπου 'm' το μήνυμα που παραλήφθηκε. Αν η μεταβλητή 'pull' του μηνύματος 'm' έχει την τιμή '**false**' τότε το μήνυμα στάλθηκε κατά τη διαδικασία push άρα απομένει να εκτελεστεί η διαδικασία pull. Κατά τη διαδικασία pull ο παραλήπτης δημιουργεί το μήνυμα 'data' με το οποίο θα απαντήσει στον αποστολέα. Στη μεταβλητή 'rumors' του μηνύματος 'data' αντιγράφει όλες τις 'hot' πληροφορίες που γνωρίζει ενώ στη μεταβλητή 'pull' αναθέτει την τιμή '**true**' για να δηλώσει ότι το μήνυμα αποστέλλεται κατά τη διαδικασία pull.

Αφού τελείωσε με τη διαδικασία pull, ο παραλήπτης θα επεξεργαστεί το μήνυμα που παρέλαβε στη διαδικασία push. Για κάθε πληροφορία του μηνύματος θα ελέγξει αν έχει ήδη παραλάβει την πληροφορία αυτή στον τρέχοντα γύρο, διαφορετικά θα την φυλάξει στη λίστα με τις πληροφορίες που παρέλαβε σε αυτό το γύρο, και όχι στη λίστα με όλες τις πληροφορίες που έχει ο κόμβος. Όταν τελειώσει με την επεξεργασία όλων των πληροφοριών του μηνύματος, εξέρχεται από τη συνάρτηση κα τελειώνει το παθητικό μέρος του γύρου. Η σύμπτυξη των πληροφοριών που παρέλαβε ο κόμβος σε αυτό τον γύρο με τις πληροφορίες που έμαθε ο κόμβος στους προηγούμενους γύρους θα γίνει στο ενεργό μέρος του επόμενου γύρου επικοινωνίας.

4.3 Πειραματική Αξιολόγηση Αλγορίθμου

Αφού υλοποιήθηκε ο αλγόριθμος συνεχίζουμε με την προσομοίωση και την πειραματική του αξιολόγηση.

4.3.1 Μεθοδολογία Πειραματικής Αξιολόγησης

Πριν ξεκινήσει η αξιολόγηση του αλγορίθμου έπρεπε να καθοριστεί ποια θα είναι η ακριβής τιμή της προθεσμίας μετάδοσης μίας πληροφορίας. Οι δημιουργοί του αλγορίθμου [1] έχουν θέση την τιμή αυτή να είναι της τάξεως $\log_3 n + O(\ln \ln n)$, όπου n ο αριθμός των κόμβων. Η λογική πίσω από αυτή την τιμή είναι ότι, όσο πιο μεγάλη είναι η προθεσμία μετάδοσης μίας πληροφορίας τόσο πιο πιθανό είναι η πληροφορία αυτή να φτάσει σε όλους τους κόμβους του συστήματος. Αυτό γιατί η πληροφορία θα είναι ‘hot’ για περισσότερους γύρους επικοινωνίας και επομένως θα μεταδίδεται για πολλούς γύρους από κόμβο σε κόμβο. Από την άλλη όμως πλευρά, η προθεσμία μετάδοσης δεν μπορεί να είναι πολύ μεγάλη γιατί τότε η πληροφορία θα είναι ‘hot’ για πολλούς γύρους επικοινωνίας και έτσι θα συνεχίσει να διαδίδεται και μετά που θα έχει μαθευτεί από όλους τους κόμβους. Ως αποτέλεσμα, η επικοινωνία στο δίκτυο θα επιβαρύνεται άσκοπα. Αφού δοκιμάστηκαν διάφορες σταθερές τιμές οι οποίες ασυμπτωτικά δεν άλλαζαν την τάξη της επιτρεπτής προθεσμίας μετάδοσης, η καταλληλότερη τιμή αποφασίστηκε να είναι η $\log_3 n + 4\ (\ln \ln n)$. Η τιμή αυτή είναι κατάλληλη για το πλήθος των κόμβων με τους οποίους μπορέσαμε να προσομοιώσουμε τη συγκεκριμένη εφαρμογή, πιθανότατα όμως για πιο μεγάλα συστήματα να χρειάζεται μία διαφορετική σταθερά.

Η εφαρμογή δεν έτρεξε σε πραγματικό σύστημα αλλά σε προσομοίωση κατά τη διάρκεια της οποίας όλοι οι κόμβοι του συστήματος τρέχουν τις λειτουργίες τους πάνω στην ίδια μηχανή. Συνεπώς, ο αριθμός των κόμβων της εφαρμογής περιορίζεται λόγω της περιορισμένης μνήμης της μηχανής καθώς και λόγω των δυνατοτήτων του διαθέσιμου επεξεργαστή. Ο μέγιστος αριθμός κόμβων που μπορέσαμε να χρησιμοποιήσουμε στην προσομοίωση της συγκεκριμένης εφαρμογής ήταν 260 κόμβοι. Προσομοιώσαμε την εφαρμογή για 20, 40, 60, …, 260 κόμβους. Αυτό επαναλήφθηκε πέντε φορές για να παρθεί στο τέλος ο μέσος όρος των αντίστοιχων περιπτώσεων προσομοίωσης έτσι ώστε οι τελικές μετρήσεις της προσομοίωσης να είναι πιο ρεαλιστικές.

Αφού υπολογίστηκε η προθεσμία μετάδοσης της πληροφορίας μπορεί να υπολογιστεί με ακρίβεια και ο γύρος επικοινωνίας στον οποίο ο αλγόριθμος θα ολοκληρώσει τις λειτουργίες του και θα τερματίσει. Αυτό γιατί οι πληροφορίες στο δίκτυο είναι στατικές και δημιουργούνται όλες στο γύρο 0, δηλαδή τη στιγμή που αρχικοποιούνται οι κόμβοι του δικτύου. Επειδή ο μετρητής που χρησιμοποιείται για τον έλεγχο της προθεσμίας μετάδοσης κάθε πληροφορίες αυξάνεται κατά ένα σε κάθε γύρο, ξέρουμε ακριβώς σε πιο γύρο θα λήξει η προθεσμία μετάδοσης των πληροφοριών και επομένως θα τερματίσει ο αλγόριθμος. Έτσι ο αλγόριθμος ολοκληρώνει την εκτέλεσή του και τερματίζει στον γύρο επικοινωνίας που είναι ίσος με την προθεσμία μετάδοσης των πληροφοριών.

4.3.2 Αποτελέσματα Πειραματικής Αξιολόγησης

Θεωρητικά ο αλγόριθμος ενημερώνει όλους τους κόμβους του συστήματος με όλες τις πληροφορίες σε $\log_3 n + O(\ln \ln n)$ γύρους επικοινωνίας και χρησιμοποιώντας $O(n \ln \ln n)$ μηνύματα, όπου n ο αριθμός των κόμβων. Τα δύο αυτά άνω φράγματα αποδείχθηκαν αυστηρότυπα στο [1]. Σκοπός της εργασίας μας είναι να συμπεράνουμε κατά πόσον οι θεωρητικές αυτές μετρήσεις επιβεβαιώνονται στην πράξη.

Ακολουθούν οι τελικές μετρήσεις της προσομοίωσης του αλγορίθμου που πάρθηκαν από το μέσο όρο των πέντε προσομοιώσεων της κάθε μιας από τις πιο κάτω περιπτώσεις. Η στήλη ‘Nodes’ αντιπροσωπεύει τον αριθμό των κόμβων στην προσομοίωση. Η στήλη ‘Rounds’ αντιπροσωπεύει τον αριθμό των γύρων επικοινωνίας

που έτρεξε ο αλγόριθμος μέχρι να τερματίσει, δηλαδή είναι η τιμή του τελευταίου γύρου επικοινωνίας. Η στήλη 'Time (msec)' αντιπροσωπεύει το χρόνο (σε χιλιοστά του δευτερολέπτου) που πήρε για να ολοκληρωθεί ο αλγόριθμος, δηλαδή μετριέται από την αρχή της εκτέλεσης της προσομοίωσης μέχρι το τέλος του τελευταίου γύρου επικοινωνίας. Η στήλη 'Messages' αντιπροσωπεύει το συνολικό αριθμό μηνυμάτων που στάλθηκαν κατά τη διάρκεια της προσομοίωσης, δηλαδή είναι ο αριθμός μηνυμάτων που αποστέλλονται από τον πρώτο μέχρι και τον τελευταίο γύρο της προσομοίωσης.

Nodes	Rounds	Time (msec)	Messages
20	7,00	174,40	280,00
40	9,00	393,20	720,00
60	9,00	648,80	1080,00
80	10,00	1093,40	1600,00
100	10,00	1576,60	2000,00
120	11,00	2560,40	2640,00
140	11,00	3353,80	3080,00
160	11,00	4445,80	3520,00
180	11,00	5891,80	3960,00
200	11,00	7082,20	4400,00
220	12,00	10137,60	5280,00
240	12,00	12721,20	5760,00
260	12,00	16324,00	6240,00

Πίνακας 4.1 Αλγόριθμος PUSH&PULL

4.3.2 Ανάλυση Αποτελεσμάτων

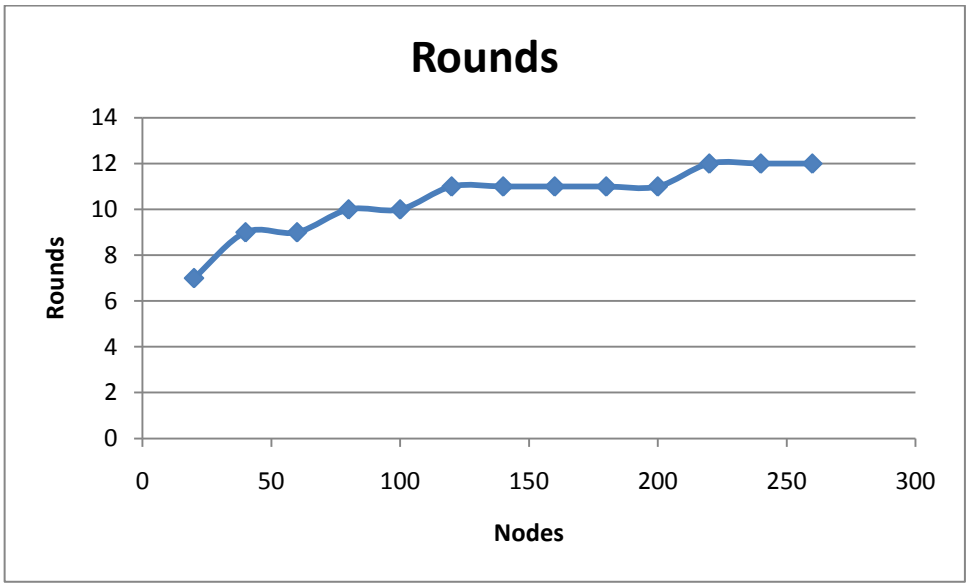
Οι μετρήσεις που πάρθηκαν από την προσομοίωση του αλγορίθμου οδήγησαν στις πιο κάτω παρατηρήσεις ως προς τις τρεις παραμέτρους αξιολόγησης του αλγορίθμου, τους γύρους επικοινωνίας, το χρόνο εκτέλεσης και τον αριθμό μηνυμάτων:

Γύροι Επικοινωνίας

Θεωρητικά ο αριθμός των γύρων επικοινωνίας στην εκτέλεση του αλγορίθμου είναι τάξεως $\log_3 n + O(\ln \ln n)$, όπου n το πλήθος των κόμβων του συστήματος. Ισοδύναμα μπορούμε να πούμε ότι ο αριθμός των γύρων επικοινωνίας είναι λογαριθμικός συναρτήσει των κόμβων τους συστήματος.

Από τον πίνακα 4.1 παρατηρούμε ότι ο αριθμός των γύρων επικοινωνίας που τρέχει ο αλγόριθμος μέχρι να ενημερωθούν όλοι οι κόμβοι μεγαλώνει καθώς μεγαλώνει ο αριθμός των κόμβων του συστήματος. Ο ρυθμός όμως με τον οποίο αυξάνονται οι γύροι επικοινωνίας δεν είναι μεγάλος. Συγκεκριμένα παρατηρούμε ότι όσο ο αριθμός των κόμβων είναι μικρός, ο αριθμός των γύρων αυξάνεται με μεγαλύτερο ρυθμό ενώ όσο ο αριθμός των κόμβων μεγαλώνει, ο ρυθμός αύξησης των γύρων επικοινωνίας μικραίνει. Για παράδειγμα όταν οι κόμβοι αυξήθηκαν από 20 σε 100 έχουμε αύξηση στους γύρους επικοινωνίας ίση με 3 γύρους, ενώ όταν οι κόμβοι αυξάνονται από 180 σε 260 η αύξηση στους γύρους επικοινωνίας είναι πολύ πιο μικρή αφού είναι ίση με 1 γύρο. Αυτή η συμπεριφορά μοιάζει να είναι λογαριθμική επομένως αναμένουμε μία λογαριθμική γραφική παράσταση η οποία θα επιβεβαιώνει το θεωρητικό άνω φράγμα για τους γύρους επικοινωνίας.

Πιο κάτω φαίνεται η γραφική παράσταση του αριθμού των γύρων επικοινωνίας συναρτήσει του αριθμού των κόμβων του συστήματος:

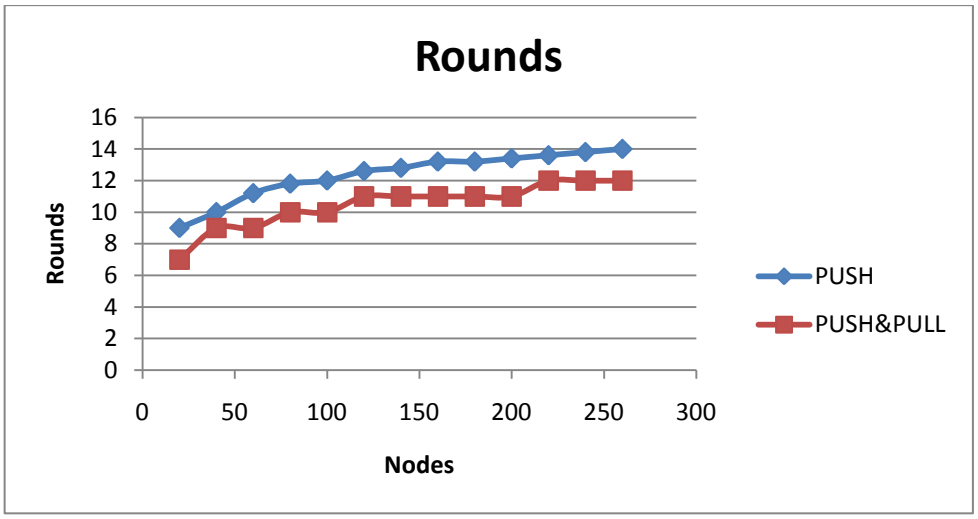


Σχήμα 4.1 Αλγόριθμος PUSH&PULL – Γύροι Επικοινωνίας

Όντως, η γραφική παράσταση που πάρθηκε από τις μετρήσεις για τους γύρους επικοινωνίας είναι λογαριθμική σε σχέση με τον αριθμό των κόμβων. Παρατηρούμε ότι η κλίση της καμπύλης της γραφικής παράστασης δεν αλλάζει με ομαλό ρυθμό και αυτό οφείλεται στο γεγονός ότι ο αριθμός των γύρων επικοινωνίας που θα τρέξει η κάθε προσομοίωση είναι πάντα ο ίδιος για συγκεκριμένο αριθμό κόμβων και είναι ίσος με

την προθεσμία μετάδοσης των πληροφοριών στη συγκεκριμένη προσομοίωση. Οπότε όσες φορές και να προσομοιώναμε την εφαρμογή η προσομοίωση θα χρειάζεται πάντα τον ίδιο αριθμό γύρων και έτσι ο μέσος όρος των προσομοιώσεων δεν εξομαλύνει τις απότομες αλλαγές στην καμπύλη των γύρων επικοινωνίας. Τα σημεία στα οποία αλλάζει η κλίση της γραφικής παράστασης είναι τα σημεία στα οποία αλλάζει η προθεσμία μετάδοσης των πληροφοριών λόγω της αύξησης του αριθμού των κόμβων. Παρόλα αυτά η καμπύλη παραμένει λογαριθμική και επιβεβαιώνεται στην πράξη το θεωρητικό άνω φράγμα $\log_3 n + O(\ln \ln n)$ για τους γύρους επικοινωνίας που χρειάζεται να τρέξει ο αλγόριθμος για να ενημερωθούν όλοι οι κόμβοι του συστήματος.

Στη συνέχεια βλέπουμε τη γραφική παράσταση των γύρων επικοινωνίας του αλγόριθμου PUSH και του αλγόριθμου PUSH&PULL για να δούμε τελικά κατά πόσον η επιπρόσθετη λειτουργία λήψης μηνυμάτων του αλγορίθμου PUSH&PULL διαφοροποίησε την απόδοση του αλγόριθμου ως προς τον αριθμό των γύρων επικοινωνίας που χρειάζεται να τρέξει. Αναμένουμε ότι ο αλγόριθμος PUSH&PULL θα χρειάζεται λιγότερους γύρους μέχρι να ενημερώσει όλους τους κόμβους λόγω του ότι εκτελείται και η λειτουργία λήψης μηνυμάτων εκτός από τη λειτουργία ώθησης, οπότε μεταδίδονται περισσότερα μηνύματα. Επίσης, η λειτουργία λήψης οδηγεί σε γρηγορότερη ενημέρωση των κόμβων λόγω του ότι αναγκάζει κάθε κόμβο στο δίκτυο να παραλάβει τουλάχιστον μία πληροφορία σε κάθε γύρο επικοινωνίας, αφού θα επιλέξει ο ίδιος τον γείτονά του που θα του στείλει την πληροφορία αυτή. Στην περίπτωση του αλγορίθμου PUSH, για να παραλάβει ένας κόμβος πληροφορία σε κάποιο γύρο επικοινωνίας πρέπει ο κόμβος αυτός να επιλεγεί από κάποιο γείτονά του ο οποίος και θα του στείλει την πληροφορία. Επειδή όμως η επιλογή του γείτονα είναι τυχαία δεν μπορούμε να είμαστε σίγουροι ότι όλοι οι κόμβοι θα επιλεγούν και θα παραλάβουν πληροφορία σε κάθε γύρο και έτσι υπάρχει μία καθυστέρηση στην ενημέρωση των κόμβων.



Σχήμα 4.2 Αλγόριθμος PUSH και Αλγόριθμος PUSH&PULL – Γύροι Επικοινωνίας

Όπως παρατηρούμε στο Σχήμα 4.2, σε κάθε περίπτωση προσομοίωσης ο αλγόριθμος PUSH&PULL τερματίζει σε λιγότερους γύρους επικοινωνίας σε σύγκριση με τον αλγόριθμο PUSH. Το γεγονός αυτό οδηγεί στο συμπέρασμα ότι όντως η προσθήκη της λειτουργία λήψης στον αλγόριθμο PUSH&PULL συντίνει στη γρηγορότερη ενημέρωση των κόμβων του δικτύου.

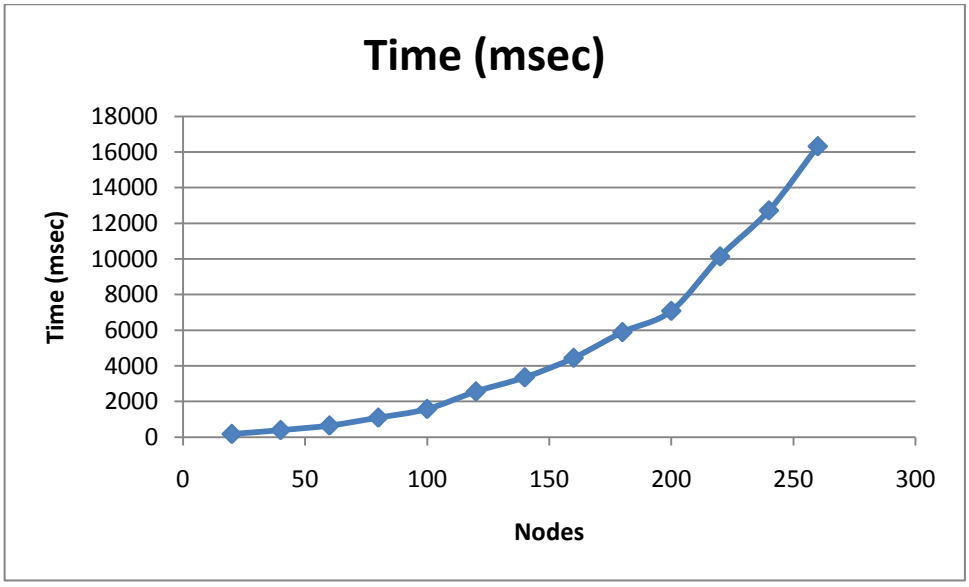
Χρόνος Εκτέλεσης

Ο χρόνος εκτέλεσης του αλγορίθμου αναμέναμε να είναι ανάλογος των γύρων επικοινωνίας. Για να συμβεί όμως αυτό πρέπει οι κόμβοι του συστήματος να σπαταλούν σταθερό χρόνο σε κάθε γύρο επικοινωνίας. Αυτό συμβαίνει μόνο στην περίπτωση που οι κόμβοι δεν επεξεργάζονται καθόλου τις πληροφορίες που παραλαμβάνουν. Επειδή όμως για τους σκοπούς της εφαρμογής μας οι κόμβοι του συστήματος επεξεργάζονται όλες τις πληροφορίες που παραλαμβάνουν, ο χρόνος που χρειάζεται ο κάθε γύρος επικοινωνίας δεν είναι σταθερός αλλά εξαρτάται από το πλήθος των πληροφοριών που παραλαμβάνει ο κάθε κόμβος στον τρέχοντα γύρο. Συνεπώς ο χρόνος εκτέλεσης του αλγορίθμου δεν αναμένεται να είναι λογαριθμικός ως προς των αριθμό των κόμβων, όπως συμβαίνει με τους γύρους επικοινωνίας, αλλά αναμένεται να είναι πολύ μεγαλύτερος.

Όπως παρατηρούμε στον πίνακα 4.1, ο χρόνος εκτέλεσης μεγαλώνει όσο μεγαλώνει ο αριθμός των κόμβων. Ο ρυθμός όμως με τον οποίο αυξάνεται ο χρόνος εκτέλεσης του

αλγορίθμου δεν είναι σταθερός, αλλά μεγαλώνει όσο αυξάνονται οι κόμβοι στο σύστημα. Για παράδειγμα όταν οι κόμβοι αυξήθηκαν από 20 σε 100 έχουμε αύξηση στον χρόνο εκτέλεσης ίση με 1402,2 χιλιοστά του δευτερολέπτου, ενώ όταν οι κόμβοι αυξάνονται από 180 σε 260 η αύξηση στον χρόνο εκτέλεσης είναι πολύ πιο μεγάλη αφού είναι ίση με 10432,2 χιλιοστά του δευτερολέπτου. Επομένως, ο χρόνος εκτέλεσης του αλγορίθμου αυξάνεται με μεγάλους ρυθμούς.

Πιο κάτω φαίνεται η γραφική παράσταση του χρόνου εκτέλεσης του αλγορίθμου συναρτήσει του αριθμού των κόμβων του συστήματος:



Σχήμα 4.3 Αλγόριθμος PUSH&PULL – Χρόνος Εκτέλεσης

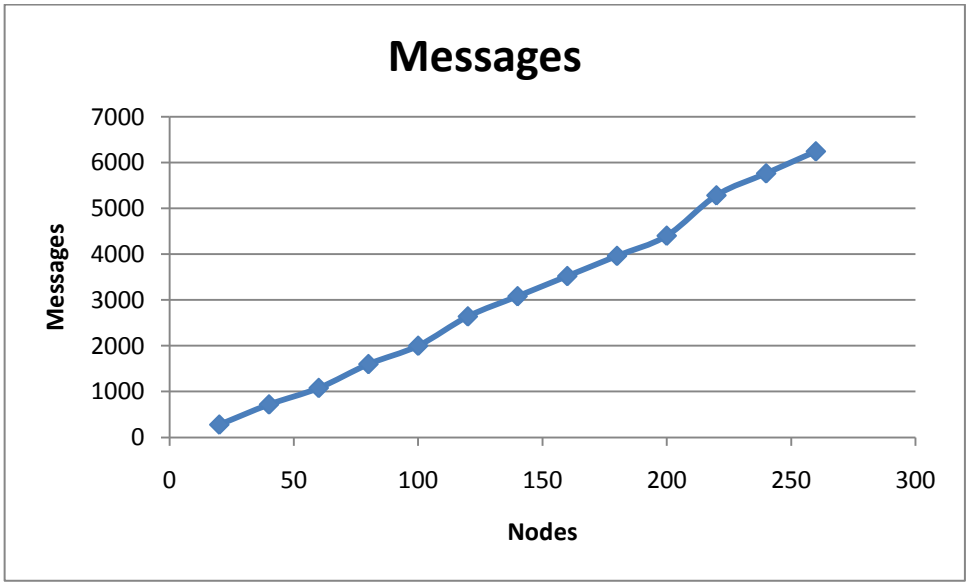
Όντως, όπως φαίνεται και στη γραφική παράσταση, ο χρόνος εκτέλεσης του αλγορίθμου δεν αυξάνεται λογαριθμικά ως προς τον αριθμό των κόμβων του συστήματος, όπως συμβαίνει με τους γύρους επικοινωνίας, και ούτε αυξάνεται γραμμικά, όπως συμβαίνει με τον αριθμό μηνυμάτων. Ο χρόνος εκτέλεσης αυξάνεται με μεγαλύτερους ρυθμούς οι οποίοι ξεπερνούν το άνω φράγμα $O(n \log n)$, όπου n ο αριθμός των κόμβων. Αυτό οφείλεται στο χρόνο που σπαταλούν οι κόμβοι σε κάθε γύρο επικοινωνίας για να επεξεργαστούν τις πληροφορίες που παραλαμβάνουν αλλά και για την εκτέλεση άλλων λειτουργιών.

Μηνύματα

Θεωρητικά ο συνολικός αριθμός μηνυμάτων που μεταδίδονται κατά τη διάρκεια εκτέλεσης του αλγορίθμου είναι ανάλογος του αριθμού των κόμβων του συστήματος και συγκεκριμένα είναι της τάξεως $O(n \ln \ln n)$, όπου n ο αριθμός των κόμβων.

Από τον πίνακα 4.1 παρατηρούμε ότι ο αριθμός των μηνυμάτων μεγαλώνει καθώς μεγαλώνει ο αριθμός των κόμβων του συστήματος. Αυτό είναι λογικό αφού για να ενημερωθούν περισσότεροι κόμβοι θα πρέπει να σταλούν περισσότερα μηνύματα. Για παράδειγμα όταν οι κόμβοι αυξήθηκαν από 20 σε 100 έχουμε αύξηση στον αριθμό των μηνυμάτων ίση με 1720 μηνύματα, ενώ όταν οι κόμβοι αυξάνονται από 180 σε 260 η αύξηση στον αριθμό των μηνυμάτων είναι ίση με 2280 μηνύματα. Επομένως παρατηρούμε ότι ο ρυθμός αύξησης του αριθμού των μηνυμάτων σε συνάρτηση με τους κόμβους είναι περίπου σταθερός και ότι ο αριθμός των μηνυμάτων είναι περίπου ανάλογος του αριθμού των κόμβων. Αυτή η συμπεριφορά μοιάζει να είναι γραμμική επομένως αναμένουμε μία γραφική παράσταση η οποία θα επιβεβαιώνει το θεωρητικό άνω φράγμα για τον αριθμό των μηνυμάτων που είναι $O(n \ln \ln n)$.

Πιο κάτω φαίνεται η γραφική παράσταση του συνολικού αριθμού των μηνυμάτων συναρτήσει του αριθμού των κόμβων του συστήματος:



Σχήμα 4.4 Αλγόριθμος PUSH&PULL – Αριθμός Μηνυμάτων

Όντως, η γραφική παράσταση που πάρθηκε από τις μετρήσεις για τον αριθμό μηνυμάτων είναι σχεδόν γραμμική σε σχέση με τον αριθμό των κόμβων. Φαίνεται να

υπάρχει μία πολύ μικρή καμπύλη στην ευθεία της γραφικής παράστασης η οποία όμως ήταν αναμενόμενη αφού από τις μετρήσεις μας ο αριθμός των μηνυμάτων δεν ήταν ακριβώς ανάλογος του αριθμού των κόμβων, αλλά είχε μία μικρή διαφορά. Η διαφορά αυτή δημιουργεί την μικρή καμπύλη που υπάρχει στην γραφική και δικαιολογεί το ‘ln ln *n*’ στο θεωρητικό άνω φράγμα. Συνεπώς, επιβεβαιώνεται στην πράξη το θεωρητικό άνω φράγμα $O(n \ln \ln n)$ για το συνολικό αριθμό μηνυμάτων που χρειάζεται να σταλούν κατά την εκτέλεση του αλγορίθμου έτσι ώστε να ενημερωθούν όλοι οι κόμβοι του συστήματος.

4.4 Συμπεράσματα

Αφού μελετήσαμε ξεχωριστά τις βασικές παραμέτρους του αλγορίθμου, καταλήγουμε στα πιο κάτω συμπεράσματα:

Καθώς αυξάνεται ο αριθμός των κόμβων στην προσομοίωση, αυξάνονται οι γύροι επικοινωνίας που πρέπει να τρέξει ο αλγόριθμος ώστε να ολοκληρώσει τη λειτουργία του, αυξάνεται ο χρόνος εκτέλεσης του αλγορίθμου και τέλος αυξάνεται και ο συνολικός αριθμός των μηνυμάτων που ανταλλάσσονται. Κάθε μια από τις τρεις αυτές παραμέτρους αυξάνεται με διαφορετικό ρυθμό και μας οδηγεί σε διαφορετικά συμπεράσματα.

Ο αριθμός των γύρων επικοινωνίας αυξάνεται λογαριθμικά ως προς τον αριθμό των κόμβων. Η λογαριθμική αύξηση των γύρων επικοινωνίας είναι πολύ αποδοτική αφού για μεγάλο αριθμό κόμβων στο σύστημα, η περεταίρω αύξηση στον αριθμό των κόμβων δεν θα οδηγήσει σε μεγάλες αλλαγές στον αριθμό των γύρων επικοινωνίας. Ο χρόνος όμως εκτέλεσης του αλγορίθμου δεν είναι αρκετά καλός αφού είναι αυτός που αυξάνεται με το μεγαλύτερο ρυθμό. Παρόλα αυτά ο χρόνος εκτέλεσης δεν μας απασχολεί τόσο γιατί δεν εξαρτάται μόνο από το χρόνο που χρειάζεται μέχρι να ενημερωθούν όλοι οι κόμβοι του συστήματος αλλά εξαρτάται και από τις επιπρόσθετες λειτουργίες που ο κάθε προγραμματιστής προσθέτει στην εφαρμογή του. Γι’ αυτό και για την αξιολόγηση του αλγορίθμου χρησιμοποιείται κυρίως ο αριθμός των γύρων επικοινωνίας αντί ο χρόνος εκτέλεσης.

Η διάδοση των μηνυμάτων και προς τις δύο κατευθύνσεις (push και pull) συντείνει στη μείωση του συνολικού αριθμού των μηνυμάτων που μεταδίδονται κατά τη διάρκεια του

αλγορίθμου ο οποίος είναι τάξεως $O(n \ln \ln n)$. Ο αριθμός των μηνυμάτων είναι πολύ ικανοποιητικός γιατί αυξάνεται σχεδόν γραμμικά συναρτήσει των κόμβων του συστήματος αφού η τιμή '$\ln \ln n$' είναι αρκετά μικρή και σχεδόν καλύπτεται από την τιμή n.

Επίσης, ένα σημαντικό πλεονέκτημα του αλγορίθμου είναι ότι φιλτράρει τα μηνύματα που αποστέλλονται και έτσι το μέγεθος των μηνυμάτων δεν είναι πολύ μεγάλο και δεν επιβαρύνει την επικοινωνία του δικτύου με επιπλέον φόρτο. Αυτό όμως εξαρτάται και από τον δημιουργό της εφαρμογής αφού εναπόκειται σε αυτόν να καθορίσει επ' ακριβώς την προθεσμία μετάδοσης μίας πληροφορίας η οποία παίζει σημαντικό ρόλο στο μέγεθος του μηνύματος αφού είναι αυτή που καθορίζει ποιες πληροφορίες πρέπει να μεταδοθούν σε κάθε γύρο επικοινωνίας. Στην εφαρμογή μας όμως το φιλτράρισμα των μηνυμάτων δεν μειώνει το μέγεθος των μηνυμάτων λόγω του ότι οι πληροφορίες είναι στατικές και η προθεσμία όλων των πληροφοριών λήγει στον ίδιο γύρο επικοινωνίας. Οπότε μέχρι να τερματίσει ο αλγόριθμος όλες οι πληροφορίες είναι 'hot' και μεταδίδονται. Το φιλτάρισμα είναι χρήσιμο στην περίπτωση όπου υπάρχουν δυναμικές πληροφορίες στο δίκτυο. Σε αυτή την περίπτωση η προθεσμία της κάθε πληροφορίας θα έληγε σε διαφορετικό γύρο επικοινωνίας και κάποιες πληροφορίες θα σταματούσαν να είναι 'hot' και να μεταδίδονται πριν τον τερματισμό του αλγορίθμου. Έτσι το μέγεθος το μηνυμάτων θα μειωνόταν. Το μέγεθος των μηνυμάτων του αλγορίθμου θα μελετηθεί στο Κεφάλαιο 7, για να μπορέσει να συγκριθεί με το μέγεθος των μηνυμάτων των υπολοίπων αλγορίθμων και έτσι να οδηγήσει σε εξαγωγή καλύτερων συμπερασμάτων ως προς την παράμετρο αυτή.

Από την άλλη όμως πλευρά ο ακριβής υπολογισμός της προθεσμίας μετάδοσης μίας πληροφορίας καθορίζει την ακριβή στιγμή που θα τερματίσει ο αλγόριθμος. Αυτό το σημείο μπορεί να θεωρηθεί ως μειονέκτημα για τον αλγόριθμο γιατί τον κάνει να μην είναι ανεκτικός σε σφάλματα τα οποία μπορεί να καθυστερήσουν την ενημέρωση κάποιων κόμβων του συστήματος.

Τέλος, οι πιο πάνω παρατηρήσεις μας οδηγούν στο γενικότερο συμπέρασμα ότι ο παρόν αλγόριθμος πληροφόρησης είναι αποτελεσματικός και αρκετά αποδοτικός. Ακόμη καταλήγουμε στο συμπέρασμα ότι η πρακτική ανάλυση του αλγορίθμου συναύει με τη θεωρητική αφού τόσο θεωρητικά όσο και πρακτικά ο αλγόριθμος ολοκληρώνεται σε $\log_3 n + O(\ln \ln n)$ γύρους επικοινωνίας και χρησιμοποιώντας $O(n \ln \ln n)$ μηνύματα.

ΚΕΦΑΛΑΙΟ 5

Αλγόριθμος MEDIAN-COUNTER

5.1 Περιγραφή Αλγορίθμου

Ο τελευταίος αλγόριθμος με τον οποίο θα ασχοληθεί η εργασία αυτή, αλγόριθμος MEDIAN-COUNTER [1], προσπαθεί να διατηρήσει τα πλεονεκτήματα των προηγούμενων αλγορίθμων και να προσθέσει σε αυτά ακόμη ένα σημαντικό χαρακτηριστικό για τους αλγόριθμους πληροφόρησης. Το χαρακτηριστικό αυτό είναι η ανεκτικότητα σε σφάλματα δικτύου και είναι ίσως το πιο σημαντικό χαρακτηριστικό ενός τέτοιου αλγορίθμου γιατί σε πραγματικά περιβάλλοντα δεν συναντούμε δίκτυα τα οποία να μην υπόκεινται σε σφάλματα. Για να επιτευχθεί η ανεκτικότητα σε σφάλματα δικτύου, ο αλγόριθμος αποφεύγει τον ακριβής υπολογισμό της στιγμής που θα τερματίσει ο αλγόριθμος. Αντί αυτού χρησιμοποιεί έναν πιο κατανεμημένο μηχανισμό για τερματισμό του αλγορίθμου ο οποίος αναγνωρίζει τη στιγμή που έχουν ενημερωθεί όλοι οι κόμβοι του συστήματος. Ο μηχανισμός αυτός, όπως και στον αλγόριθμο PUSH&PULL, βασίζεται επίσης στο γεγονός ότι κάθε πληροφορία παύει να είναι χρήσιμη από κάποιο σημείο και μετά. Για να αποφευχθεί όμως ο αυστηρός τερματισμός του αλγορίθμου, η προθεσμία μετάδοσης της κάθε πληροφορίας δεν μειώνεται σε κάθε γύρο επικοινωνίας. Ο μηχανισμός τερματισμού που χρησιμοποιείται βασίζεται στον κανόνα Median rule, ο οποίος αναγνωρίζει τη στιγμή που κάποια πληροφορία έχει εξαπλωθεί αρκετά στο δίκτυο. Ο αλγόριθμος MEDIAN-COUNTER θεωρεί ότι μια πληροφορία ενός κόμβου μπορεί να είναι σε μια από τις τέσσερις καταστάσεις A, B, C ή D, ανάλογα με το χρονικό διάστημα που η πληροφορία βρίσκεται στον κόμβο αυτό, και ένας κόμβος μεταδίδει μόνο τις πληροφορίες του που βρίσκονται σε μία από τις καταστάσεις B ή C. Μία πληροφορία που βρίσκεται σε κατάσταση B ή C έχει και ένα μετρητή, έστω m, ο οποίος χρησιμοποιείται για να αναγνωριστεί η στιγμή που η

πληροφορία πρέπει να μεταβεί στην επόμενη κατάσταση. Θα συμβολίζουμε με B(m) την κατάσταση ενός κόμβου που βρίσκεται σε κατάσταση B ως προς μια πληροφορία και έχει μετρητή ίσο με m και αντίστοιχα θα συμβολίζουμε με C(m) την κατάσταση ενός κόμβου που βρίσκεται σε κατάσταση C ως προς μια πληροφορία και έχει μετρητή ίσο με m. Η μετάδοση πληροφοριών γίνεται όπως και στον αλγόριθμο PUSH&PULL και προς τις δύο κατευθύνσεις και ο αλγόριθμος τερματίζει όταν κανένας κόμβος δεν έχει πληροφορίες σε κατάσταση B ή C για να μεταδώσει.

Πιο συγκεκριμένα, ένας κόμβος *v*, για την πληροφορία *r*, βρίσκεται σε:

Κατάσταση **A**: Ο κόμβος *v* δεν γνωρίζει ακόμη την πληροφορία *r*. Καθώς ένας κόμβος βρίσκεται στην κατάσταση A, αν παραλάβει την πληροφορία *r* μόνο από κόμβους σε κατάσταση B τότε η κατάσταση του κόμβου *v* ως προς τη *r* θα αλλάξει σε κατάσταση B(1). Αν ο κόμβος *v* παραλάβει την πληροφορία *r* από κάποιο κόμβο σε κατάσταση C τότε η κατάσταση του κόμβο *v* ως προς την πληροφορία *r* μετατρέπεται σε C(0).

Κατάσταση **B**: Ο κόμβος *v* γνωρίζει την πληροφορία *r*. Κάθε κόμβος σε αυτή την κατάσταση έχει και ένα μετρητή που δηλώνει το χρονικό διάστημα που η συγκεκριμένη πληροφορία βρίσκεται σε αυτόν τον κόμβο. Όταν ένας κόμβος δημιουργεί μία πληροφορία η κατάστασή του είναι B(1).. Στην κατάσταση B εφαρμόζεται ο Κανόνας Μεσαίου (Median Rule): με βάση αυτόν τον κανόνα αν κατά τη διάρκεια ενός γύρου επικοινωνίας ο κόμβος *v* ο οποίος βρίσκεται σε κατάσταση B(m) ως προς την πληροφορία *r* παραλάβει την *r* από περισσότερους κόμβους σε κατάσταση B(m΄) με m΄>=m παρά από κόμβους σε κατάσταση B(m΄΄) με m΄΄<m τότε η κατάσταση του *v* ως προς την πληροφορία *r* μετατρέπεται σε B(m+1), δηλαδή αυξάνεται ο μετρητής του *v* ως προς την *r* κατά ένα. Σε αυτό τον κανόνα υπάρχει μία εξαίρεση: αν ο μετρητής του *v* ως προς την *r* αυξηθεί στον μέγιστο μετρητή ο οποίος είναι τάξεως $O(\ln \ln n)$, τότε η κατάσταση του *v* μετατρέπεται σε C(0). Επίσης αν ένας κόμβος *v* σε κατάσταση B ως προς μία πληροφορία *r* παραλάβει την *r* από κόμβο σε κατάσταση C τότε και η κατάσταση του *v* ως προς την *r* μετατρέπεται σε C(0).

Κατάσταση **C**: Ένας κόμβος παραμένει σε αυτή την κατάσταση για τάξεως $O(\ln \ln n)$ γύρους επικοινωνίας. Μόλις φτάσει τον μέγιστο αριθμό γύρων η κατάστασή του v ως προς την *r* μετατρέπεται σε D.

Κατάσταση **D**: Ο κόμβος *v* σταματά να διαδίδει την πληροφορία *r*.

Τέλος, για να αποφευχθεί ο μη τερματισμός του αλγορίθμου στην απίθανη περίπτωση που αποτύχει ο μηχανισμός τερματισμού, καθορίζουμε ότι κάθε κόμβος σταματά να διαδίδει μία πληροφορία μετά από ένα σταθερό αριθμό γύρων επικοινωνίας τάξεως $O(\ln n)$, ασχέτως της κατάστασης στην οποία βρίσκεται ο κόμβος. Για να αποτύχει ο μηχανισμός τερματισμού πρέπει ένας κόμβος να μην καταφέρει να μεταβεί στην κατάσταση C για κάποια πληροφορία. Από τη στιγμή που ένας κόμβος βρίσκεται στην κατάσταση C ως προς κάποια πληροφορία τότε ο μετρητής του ως προς αυτή την κατάσταση θα αυξάνεται σε κάθε γύρο επικοινωνίας κατά ένα, άρα είναι σίγουρο ότι μετά από $O(\ln \ln n)$ γύρους ο μετρητής θα αυξηθεί στη μέγιστη τιμή και ο κόμβος θα μεταβεί στην κατάσταση D όπου η πληροφορία σταματά να μεταδίδεται. Καθώς όμως ένας κόμβος βρίσκεται στην κατάσταση B ως προς κάποια πληροφορία, η αύξηση του μετρητή για αυτή την κατάσταση δεν συμβαίνει σε κάθε γύρο επικοινωνίας αλλά συμβαίνει μόνο υπό τις περιστάσεις που περιγράφονται στον κανόνα Median rule. Έτσι ένας κόμβος σε κατάσταση B ως προς κάποια πληροφορία μπορεί να μην παραλάβει την πληροφορία αυτή με μεγαλύτερο μετρητή από τον δικό του όσες φορές χρειάζεται για να αυξήσει τον μετρητή του και συνεπώς να μην καταφέρει να μεταβεί σε κατάσταση C η οποία θα τον οδηγήσει στον τερματισμό.

5.2 Υλοποίηση Αλγορίθμου

Αρχικά δηλώνονται οι κόμβοι του συστήματος και στον καθένα ανατίθεται μία διαφορετική πληροφορία. Για την αρχικοποίηση της κατάστασης του κόμβου είναι υπεύθυνος ο κατασκευαστής της πληροφορίας στον οποίο υπάρχει ο ακόλουθος κώδικας

```
rumorM(String c){

   ....

   this.state='B';

   this.ctr=1;

}
```

Όπου ‘`state`’ η αρχική κατάσταση του κόμβου ως προς την αρχική του πληροφορία και ‘`ctr`’ ο μετρητής που αντιστοιχεί στην αρχική κατάσταση του κόμβου. Μόλις ένας

κόμβος δημιουργεί μία πληροφορία η κατάστασή του ως προς την πληροφορία αυτή είναι B(1).

Ακολούθως, πριν ξεκινήσει η εκτέλεση του αλγορίθμου υπολογίζεται ο μέγιστος αριθμός (ctrMax) τον οποίο μπορεί να πάρει ο μετρητής ενός κόμβου που βρίσκεται σε κατάσταση B ή C. Επίσης για να μην υπάρχει περίπτωση να μην τερματίσει ο αλγόριθμος, ορίζεται και ο μέγιστος αριθμός των γύρων επικοινωνίας (maxNumOfRounds) κατά τους οποίους μπορεί μία πληροφορία να διαδίδεται.

Ο αλγόριθμος εκτελείται σε γύρους επικοινωνίας και κάθε γύρος επικοινωνίας αποτελείται από το ενεργό και το παθητικό μέρος.

5.2.1 Ενεργό μέρος

Το ενεργό μέρος της εφαρμογής αποτελείται από τη συνάρτηση

```
protected void periodicRun() {

}
```

Στην αρχή του κάθε γύρου επικοινωνίας, ο κάθε κόμβος ενημερώνει τις πληροφορίες που γνωρίζει προσθέτοντας σε αυτές τις πληροφορίες που παρέλαβε στον προηγούμενο γύρο. Οι πληροφορίες που παραλαμβάνει ένας κόμβος σε κάποιο γύρο επικοινωνίας δεν μπορούν να μεταδοθούν από τον παραλήπτη μέσα στον ίδιο γύρο επειδή η επικοινωνία των κόμβων του συστήματος σε ένα γύρο γίνεται παράλληλα. Γι' αυτό το λόγο οι πληροφορίες που παραλαμβάνει ένας κόμβος δεν προστίθενται αμέσως στο χώρο της μνήμης όπου ο κόμβος κρατά τις πληροφορίες που γνωρίζει αλλά σε κάποιο άλλο χώρο της μνήμης του. Έτσι, πριν ξεκινήσει η ουσιαστική διαδικασία του ενεργού μέρους του κάθε γύρου επικοινωνίας, ο κόμβος πρέπει να ενημερώσει τις πληροφορίες που γνωρίζει προσθέτοντας σε αυτές τις πληροφορίες που παρέλαβε στον προηγούμενο γύρο. Πιο κάτω φαίνεται ο τρόπος που οργανώνει τα περιεχόμενά του ο κάθε κόμβος:

```
class NodeContentListDynamicM implements Serializable {
  private static final long serialVersionUID = 1L;
  ArrayList<rumorM> rumors;
  ArrayList<rumorM> tempRumors;
  ArrayList<rumorM> rumorsToSend;
```

```
    . . .
}
```

όπου στη λίστα ‘rumors’ ο κάθε κόμβος φυλάει τις πληροφορίες που γνωρίζει και που μπορεί να μεταδώσει στον τρέχοντα γύρο, ενώ στη λίστα ‘tempRumors’ φυλάει τις πληροφορίες που παραλαμβάνει στον τρέχοντα γύρο επικοινωνίας και οι οποίες θα μπορούν να μεταδοθούν από τον επόμενο γύρο. Στη λίστα ‘rumorsToSend’ φυλάει τις πληροφορίες που βρίσκονται στη λίστα ‘rumors’ και οι οποίες είναι σε κατάσταση B ή C, δηλαδή τις πληροφορίες που θα μεταδόσει στον τρέχοντα γύρο.

Για την ενημέρωση των πληροφοριών ενός κόμβου ακολουθείται η πιο κάτω διαδικασία:

Κάθε πληροφορία που παραλαμβάνει ένας κόμβος βρίσκεται σε κατάσταση B ή C. Αν ο κόμβος έχει ήδη στην μνήμη του την πληροφορία που παρέλαβε τότε υπάρχουν τρεις περιπτώσεις:

Στην περίπτωση που παραλήφθηκε μία πληροφορία σε κατάσταση C και ο κόμβος παραλήπτης βρίσκεται σε κατάσταση B ως προς την πληροφορία αυτή, τότε ο παραλήπτης θα αλλάξει την κατάστασή του σε C(-1). Ο μετρητής εδώ αρχικοποιείται με -1 γιατί στη συνέχεια, όπως θα δούμε, θα αυξηθεί κατά ένα και έτσι η κατάσταση θα γίνει C(0) όπως επιθυμούμε. Πιο κάτω φαίνεται ο αντίστοιχος κώδικας

```
if (tempRumor.state=='C' && tempMyContent.state=='B'){
   tempMyContent.state='C';
   tempMyContent.numOfRounds=-1;
}
```

Όπου ‘tempRumor’ η πληροφορία που παραλήφθηκε και ‘tempMyContent’ η αντίστοιχη πληροφορία την οποία γνωρίζει ο παραλήπτης. Όταν η κατάσταση του κόμβου μετατρέπεται σε C αρχικοποιείται και ο μετρητής ‘numOfRounds’ ο οποίος θα καθορίσει τη στιγμή που η κατάσταση του κόμβου θα μετατραπεί σε D.

Στην περίπτωση που παραλήφθηκε μία πληροφορία σε κατάσταση B και ο κόμβος παραλήπτης βρίσκεται και αυτός σε κατάσταση B ως προς την πληροφορία αυτή, τότε ο παραλήπτης θα ελέγξει κατά πόσον ο μετρητής της κατάστασης της πληροφορίας που

παράλαβε είναι μεγαλύτερος ή μικρότερος από τον δικό του και θα αυξήσει τον αντίστοιχο μετρητή. Πιο κάτω φαίνεται ο αντίστοιχος κώδικας

```
if (tempRumor.state=='B' && tempMyContent.state=='B'){
   if (tempRumor.ctr>=tempMyContent.ctr)
      tempMyContent.biggerB++;
   else
      tempMyContent.smallerB++;

}
```

Όπου 'tempRumor' η πληροφορία που παραλήφθηκε, 'tempMyContent' η αντίστοιχη πληροφορία την οποία γνωρίζει ο παραλήπτης, 'biggerB' το πλήθος των αντίστοιχων πληροφοριών που παρέλαβε με μετρητή κατάστασης μεγαλύτερο ή ίσο με το δικό του και 'smallerB' το πλήθος των αντίστοιχων πληροφοριών που παρέλαβε με μετρητή μικρότερο από το δικό του.

Στην περίπτωση που παραλήφθηκε μία πληροφορία σε κατάσταση C και ο κόμβος παραλήπτης βρίσκεται σε κατάσταση A ως προς την πληροφορία αυτή, τότε ο παραλήπτης θα αλλάξει την κατάστασή του σε C(-1). Ο μετρητής εδώ αρχικοποιείται με -1 γιατί στη συνέχεια, όπως θα δούμε, θα αυξηθεί κατά ένα και έτσι η κατάσταση θα γίνει C(0) όπως επιθυμούμε. Πιο κάτω φαίνεται ο αντίστοιχος κώδικας

```
if (tempRumor.state=='C' && tempMyContent.state=='A'){
   tempMyContent.state='C';
   tempMyContent.numOfRounds=-1;
}
```

Όπου 'tempRumor' η πληροφορία που παραλήφθηκε και 'tempMyContent' η αντίστοιχη πληροφορία την οποία γνωρίζει ο παραλήπτης. Όταν η κατάσταση του κόμβου μετατρέπεται σε C αρχικοποιείται και ο μετρητής 'numOfRounds' ο οποίος θα καθορίσει τη στιγμή που η κατάσταση του κόμβου θα μετατραπεί σε D.

Αν ο κόμβος δεν γνωρίζει την πληροφορία που παρέλαβε τότε υπάρχουν δύο περιπτώσεις:

Στην περίπτωση που παρέλαβε πληροφορία σε κατάσταση C τότε την φυλάει και διατηρεί την κατάσταση C αρχικοποιώντας το μετρητή κατάστασης. Αν παρέλαβε πληροφορία σε κατάσταση B τη φυλάει με κατάσταση A. Ο λόγος που φυλάει την

πληροφορία σε κατάσταση A και όχι σε κατάσταση B είναι γιατί ακόμη γίνεται ο έλεγχος για το ποιά θα είναι η τελική κατάσταση με την οποία θα φυλάξει την πληροφορία, οπότε δε θεωρείται ακόμα ότι ο κόμβος γνωρίζει την πληροφορία και επομένως δε θέλουμε να εφαρμοστεί σε αυτήν ο κανόνας Median rule. Πιο κάτω φαίνεται ο αντίστοιχος κώδικας

```
if (tempRumor.state=='C')
   tempRumor.numOfRounds=-1;

if (tempRumor.state=='B')
   tempRumor.state='A';

myContent.rumors.add(tempRumor);
```

Όπου 'tempRumor' η πληροφορία που παραλήφθηκε και η συνάρτηση 'myContent.rumors.add(tempRumor)' προσθέτει την πληροφορία στις πληροφορίες που γνωρίζει ο κόμβος. Για τις πληροφορίες που φύλαχτηκαν σε κατάσταση A, αφού έχουν ελεγχτεί όλες οι πληροφορίες που παραλήφθηκαν στον προηγούμενο γύρο και δεν έχει παραληφθεί η ίδια πληροφορία σε κατάσταση C, τότε η κατάσταση της πληροφορίας πρέπει να μετατραπεί από A σε B(1). Πιο κάτω φαίνεται ο αντίστοιχος κώδικας

```
if (tempR.state=='A'){
   tempR.state='B';
   tempR.ctr=1;
   tempR.biggerB=0;
   tempR.smallerB=0;
}
```

Αφού ελέγχτηκε μία μία η κάθε πληροφορία που παραλήφθηκε στον προηγούμενο γύρο και προστέθηκε στις πληροφορίες του κόμβου με την κατάλληλη κατάσταση, μένει να ενημερωθούν οι μετρητές για την κατάστασης της κάθε πληροφορίας που γνωρίζει πλέον ο κόμβος.

Αν ο κόμβος βρίσκεται σε κατάσταση B ως προς την πληροφορία τότε θα ελέγξει αν έχει παραλάβει περισσότερες πληροφορίες με μεγαλύτερο μετρητή κατάστασης από τον δικό του παρά με μικρότερο μετρητή και αν ναι θα αυξήσει τον μετρητή του. Στη συνέχεια θα ελέγξει αν ο μετρητής του έφτασε το μέγιστο όριο οπότε θα πρέπει η κατάσταση να μετατραπεί σε C(0). Πιο κάτω φαίνεται ο αντίστοιχος κώδικας

```
if (tempR.state=='B'){
   if (tempR.biggerB>tempR.smallerB)
      tempR.ctr++;
   tempR.smallerThanCtrMax(ctrMax);
   ....
}
```

Όπου η συνάρτηση ‘`smallerThanCtrMax(ctrMax)`’ ελέγχει αν ο μετρητής για την πληροφορία ‘`tempR`’ έφτασε στο μέγιστο όριο και αν ναι αλλάζει την κατάσταση του κόμβου σε C(0).

Αν ο κόμβος βρίσκεται σε κατάσταση C ως προς την πληροφορία τότε απλά θα αυξήσει το μετρητή του και θα ελέγξει αν ο μετρητής έφτασε στο μέγιστο όριο οπότε θα πρέπει η κατάσταση να μετατραπεί σε D. Πιο κάτω φαίνεται ο αντίστοιχος κώδικας

```
if (tempR.state=='C'){
   tempR.numOfRounds++;
tempR.checkForStateD(ctrMax);
}
```

Όπου η συνάρτηση ‘`checkForStateD(ctrMax)`’ ελέγχει αν ο μετρητής για την πληροφορία ‘`tempR`’ έφτασε στο μέγιστο όριο και αν ναι αλλάζει την κατάσταση του κόμβου σε D.

Αφού ολοκληρώθηκε η ενημέρωση των πληροφοριών των κόμβων, θα πρέπει ο κάθε κόμβος να ελέγξει αν βρίσκεται σε κατάσταση B ή C ως προς κάποια πληροφορία, οπότε θα πρέπει να τη διαδώσει. Για να γίνει αυτός ο έλεγχος, ο κάθε κόμβος ελέγχει την κατάστασή του ως προς κάθε πληροφορία που γνωρίζει και για κάθε κατάσταση B ή C προσθέτει την αντίστοιχη πληροφορία σε μία ειδική λίστα με το όνομα ‘`rumorsToSend`’. Το κομμάτι κώδικα για τον έλεγχο της κατάστασης του κόμβου ως προς κάθε πληροφορία που κρατά είναι

```
if (temp.state=='B' || temp.state=='C'){
   if (roundNum-temp.birth<=maxNumOfRounds)
   myContent.rumorsToSend.add(temp);
}
```

Όπου ‘temp’ η πληροφορία την οποία ελέγχει και ‘temp.state’ η κατάσταση του κόμβου ως προς την πληροφορία αυτή. Σημαντικός είναι επίσης και ο έλεγχος που γίνεται ως προς τον μέγιστο αριθμό γύρων που τρέχει ο αλγόριθμος ο οποίος θα αποτρέψει τη διάδοση της πληροφορίας σε περίπτωση που απέτυχε ο μηχανισμός τερματισμού του αλγόριθμου. Για τον έλεγχο αυτό έχουμε ‘roundNum’ να είναι ο αριθμός του τρέχοντα γύρου επικοινωνίας και ‘temp.birth’ να είναι ο γύρος στον οποίο δημιουργήθηκε η πληροφορία. Αν η διαφορά των δύο αυτών τιμών, roundNum-temp.birth, δεν είναι μικρότερη από το επιτρεπτό όριο, maxNumOfRounds, η πληροφορία δε θα διαδοθεί.

Οι κόμβοι που δεν βρίσκονται σε κατάσταση B ή C ως προς κάποια πληροφορία και επομένως δεν έχουν πληροφορίες για να στείλουν θα βγουν από το ενεργό μέρος της εφαρμογής. Οι υπόλοιποι κόμβοι συνεχίζουν την εκτέλεση του ενεργού μέρους δημιουργώντας το μήνυμα που θα στείλουν σε αυτό το γύρο κατά τη διαδικασία push. Ο κάθε κόμβος βάζει στο μήνυμά του όλες τις πληροφορίες που έμαθε μέχρι και τον προηγούμενο γύρο επικοινωνίας και οι οποίες τον καθιστούν σε κατάσταση B ή C. Το κομμάτι κώδικα για αυτό το σημείο είναι

```
data.rumors=(ArrayList<rumorM>) myContent.rumorsToSend.clone();

data.pull=false;
```

όπου ‘data’ το μήνυμα που θα σταλεί και η συνάρτηση ‘myContent.rumorsToSend.clone()’ αντιγράφει στη μεταβλητή ‘rumors’ του μηνύματος όλες τις πληροφορίες του κόμβου που είναι σε κατάσταση B ή C. Επίσης, η μεταβλητή ‘pull’ του μηνύματος παίρνει την τιμή ‘**false**’ η οποία δηλώνει ότι το μήνυμα αποστέλλεται κατά τη διαδικασία push της εφαρμογής.

Αφού έχουν δημιουργηθεί τα μηνύματα που θα σταλούν σε αυτό το γύρο, μένει να επιλέξει ο κάθε κόμβος του συστήματος τον γείτονά του με τον οποίο θα επικοινωνήσει και θα του στείλει το μήνυμα. Η επιλογή γίνεται με τυχαίο τρόπο χρησιμοποιώντας την κλάση Random της βιβλιοθήκης Java. Το κομμάτι κώδικα που αφορά αυτό το σημείο είναι

```
int size=peers.size();

int rand=new Random().nextInt(size);
```

```
Peer p=peers.get(rand);
```

Όπου ‘peers’ η λίστα με τους κόμβους του συστήματος, η συνάρτηση ‘peers.size()’ επιστρέφει τον αριθμό των κόμβων του συστήματος και η συνάρτηση ‘peers.get(rand)’ επιστρέφει τον κόμβο που βρίσκεται στην θέση ‘rand’ της λίστας με τους κόμβους.

Τέλος, για να ολοκληρωθεί το ενεργό μέρος του γύρου επικοινωνίας μένει να στείλουν οι κόμβοι το μήνυμά τους στον γείτονα που επέλεξαν. Το κομμάτι κώδικα που αφορά αυτό το σημείο είναι

```
getCommLayer().sendUDP(data, neighbour);
```

Όπου ‘data’ το μήνυμα που θα σταλεί και ‘neighbour’ ο γείτονας στον οποίο θα σταλεί το μήνυμα.

5.2.2 Παθητικό μέρος

Το παθητικό μέρος της εφαρμογής αποτελείται από τη συνάρτηση

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

}
```

Ένας κόμβος εισέρχεται αυτόματα στο παθητικό μέρος του γύρου επικοινωνίας όταν παραλάβει κάποιο μήνυμα. Η πρώτη ενέργεια του κάθε παραλήπτη είναι να ελέγξει αν ο αποστολέας του μηνύματος θέλει και αυτός να ανακτήσει τις πληροφορίες που γνωρίζει ο παραλήπτης, αν απαιτείται δηλαδή η εκτέλεση διαδικασίας pull. Αν ναι, ο παραλήπτης θα ελέγξει αν έχει πληροφορίες σε κατάσταση B ή C και θα τις στείλει πίσω στον αποστολέα. Αυτό γίνεται με το ακόλουθο κομμάτι κώδικα

```
if (m.pull==false){

data.rumors=(ArrayList<rumorM>)receiverContent.rumorsToSend.clone();

data.pull=true;

getCommLayer().sendUDP(data, sender);

}
```

Όπου ‘m’ το μήνυμα που παραλήφθηκε. Αν η μεταβλητή ‘pull’ του μηνύματος ‘m’ έχει την τιμή ‘false’ τότε το μήνυμα στάλθηκε κατά τη διαδικασία push άρα απομένει

να εκτελεστεί η διαδικασία pull. Κατά τη διαδικασία pull ο παραλήπτης δημιουργεί το μήνυμα ‘data’ με το οποίο θα απαντήσει στον αποστολέα. Στη μεταβλητή ‘rumors’ του μηνύματος ‘data’ αντιγράφει όλες τις πληροφορίες που γνωρίζει και που βρίσκεται σε κατάσταση B ή C ως προς αυτές ενώ στη μεταβλητή ‘pull’ αναθέτει την τιμή ‘**true**’ για να δηλώσει ότι το μήνυμα αποστέλλεται κατά τη διαδικασία pull.

Αφού τελείωσε με τη διαδικασία pull, ο παραλήπτης θα επεξεργαστεί το μήνυμα που παρέλαβε στη διαδικασία push. Για κάθε πληροφορία του μηνύματος θα ελέγξει αν έχει ήδη παραλάβει την πληροφορία αυτή στον τρέχοντα γύρο, διαφορετικά θα την φυλάξει στη λίστα με τις πληροφορίες που παρέλαβε σε αυτό το γύρο, και όχι στη λίστα με όλες τις πληροφορίες που έχει ο κόμβος. Όταν τελειώσει με την επεξεργασία όλων των πληροφοριών του μηνύματος, εξέρχεται από τη συνάρτηση κα τελειώνει το παθητικό μέρος του γύρου. Η σύμπτυξη των πληροφοριών που παρέλαβε ο κόμβος σε αυτό τον γύρο με τις πληροφορίες που έμαθε ο κόμβος στους προηγούμενους γύρους θα γίνει στο ενεργό μέρος του επόμενου γύρου επικοινωνίας.

5.3 Πειραματική Αξιολόγηση Αλγορίθμου

Αφού υλοποιήθηκε ο αλγόριθμος συνεχίζουμε με την προσομοίωση και την πειραματική του αξιολόγηση.

5.3.1 Μεθοδολογία Πειραματικής Αξιολόγησης

Πριν ξεκινήσει η αξιολόγηση του αλγορίθμου έπρεπε να καθοριστεί ποια θα είναι η ακριβής μέγιστη τιμή που μπορεί να πάρει ο μετρητής ενός κόμβου που βρίσκεται σε κατάσταση B ή C. Οι δημιουργοί του αλγορίθμου [1] έχουν θέσει την τιμή αυτή να είναι της τάξεως $O(\ln \ln n)$, όπου n το πλήθος των κόμβων της εφαρμογής. Αρχικά ορίστηκε ο μέγιστος μετρητής να είναι ίσος με $2*(\ln \ln n)$ και έτσι προλάβαιναν όλοι οι κόμβοι του συστήματος να ενημερωθούν. Το αποτέλεσμα της πιο πάνω εξίσωσης έπαιρνε τιμές από 2 μέχρι 4, λόγω του ότι το n ήταν περιορισμένο στην προσομοίωση της εφαρμογής. Επειδή όμως για την αξιολόγηση της εφαρμογής θέλαμε ο μέγιστος μετρητής να είναι κοινός για όλες τις περιπτώσεις προσομοίωσης, επιλέξαμε τη σταθερά 4 ως την τιμή του μέγιστου μετρητή. Επιλέχτηκε η τιμή 4 γιατί ήταν αυτή που οδηγούσε όλες τις περιπτώσεις προσομοίωσης στο να ενημερώσουν όλους τους

κόμβους του συστήματος στον επιθυμητό χρόνο. Επίσης για να μην υπάρχει περίπτωση να μην τερματίσει ο αλγόριθμος, έπρεπε να οριστεί ο μέγιστος αριθμός των γύρων επικοινωνίας (maxNumOfRounds) κατά τους οποίους μπορεί μία πληροφορία να διαδίδεται. Ο αριθμός αυτός τέθηκε από τους δημιουργούς του αλγορίθμου [1] να είναι τάξεως $O(\ln n)$ και επιλέχτηκε να είναι ίσος με $10*(\ln n)$ γιατί παρατηρήσαμε ότι η τιμή αυτή ήταν σε όλες τις περιπτώσεις προσομοίωσης αρκετά πιο μεγάλη από τον μέγιστο μετρητή. Με αυτό τον τρόπο δεν εμπόδιζει το μηχανισμό τερματισμού του αλγορίθμου αφού αν ο αλγόριθμος τερματίσει σωστά θα τερματίσει πολύ πιο γρήγορα από τους $10*(\ln n)$ γύρους επικοινωνίας. Οι δύο αυτές τιμές που επιλέχτηκαν είναι κατάλληλες για το πλήθος των κόμβων με τους οποίους μπορέσαμε να προσομοιώσουμε τη συγκεκριμένη εφαρμογή, πιθανότατα όμως για πιο μεγάλα συστήματα να χρειάζονται διαφορετικές τιμές.

Όπως αναφέρθηκε, ο αριθμός των κόμβων της εφαρμογής είναι περιορισμένος. Αυτό γιατί η εφαρμογή δεν έτρεξε σε πραγματικό σύστημα αλλά σε προσομοίωση κατά τη διάρκεια της οποίας όλοι οι κόμβοι του συστήματος τρέχουν τις λειτουργίες τους πάνω στην ίδια μηχανή. Συνεπώς, ο αριθμός των κόμβων της εφαρμογής περιορίζεται λόγω της περιορισμένης μνήμης της μηχανής καθώς και λόγω των δυνατοτήτων του διαθέσιμου επεξεργαστή. Ο μέγιστος αριθμός κόμβων που μπορέσαμε να χρησιμοποιήσουμε στην προσομοίωση της συγκεκριμένης εφαρμογής ήταν 240 κόμβοι. Προσομοιώσαμε την εφαρμογή για 20, 40, 60, …, 240 κόμβους. Αυτό επαναλήφθηκε πέντε φορές για να παρθεί στο τέλος ο μέσος όρος των αντίστοιχων περιπτώσεων της προσομοίωσης έτσι ώστε οι τελικές μετρήσεις της προσομοίωσης να είναι πιο ρεαλιστικές.

Ο αλγόριθμος ολοκληρώνει την εκτέλεσή του όταν κανένας κόμβος στο δίκτυο δεν έχει πληροφορίες σε κατάσταση B ή C για να στείλει. Σε αυτό το σημείο είτε θα έχουν ενημερωθεί όλοι οι κόμβοι με όλες τις πληροφορίες και όλες οι πληροφορίες τους είναι σε κατάσταση D, είτε κάποιοι κόμβοι έχουν ενημερωθεί και οι πληροφορίες βρίσκονται σε κατάσταση D ενώ κάποιοι άλλοι δεν έχουν ακόμα ενημερωθεί και βρίσκονται σε κατάσταση A. Ακόμη και αν ισχύει η δεύτερη περίπτωση, ο αλγόριθμος πάλι θα ολοκληρώσει την εκτέλεσή του αφού δεν υπάρχουν κόμβοι οι οποίοι να έχουν πληροφορίες σε κατάσταση B ή C για να ενημερώσουν τους κόμβους που βρίσκονται σε κατάσταση A. Λόγω όμως του ότι το σύστημα είναι κατανεμημένο και ο κάθε

κόμβος στο δίκτυο έχει μόνο τοπική γνώση, δεν μπορεί ένας κόμβος να γνωρίζει τη στιγμή που κανένας κόμβος στο δίκτυο δεν θα έχει πληροφορίες σε κατάσταση B ή C. Γι' αυτό το λόγο δεν μπορεί κάποιος κόμβος να σταματήσει να λειτουργεί αφού μπορεί ανά πάσα στιγμή να παραλάβει μήνυμα από κάποιον κόμβο ο οποίος έχει ακόμη πληροφορίες σε κατάσταση B ή C. Συνεπώς ο αλγόριθμος MEDIAN-COUNTER δεν τερματίζει μόλις ολοκληρώσει τη λειτουργία του. Αντί αυτού καθορίζω εγώ το γύρο επικοινωνίας στον οποίο θέλω να τερματίσει ο αλγόριθμος. Ο γύρος αυτός αποφάσισα να είναι ο γύρος 20 επειδή μετά από αρκετές προσομοιώσεις του αλγορίθμου παρατήρησα ότι ο αλγόριθμος ολοκλήρωνε τη λειτουργία του σε 16 το πολύ γύρους επικοινωνίας. Ο γύρος στον οποίο καταλαβαίνω ότι ο αλγόριθμος ολοκλήρωσε τη λειτουργία του είναι ο γύρος στον οποίο δεν στάλθηκε κανένα μήνυμα. Για να μην έχει σταλθεί κανένα μήνυμα σε κάποιο γύρο έπεται ότι κανένας κόμβος δεν έχει πληροφορίες σε κατάσταση B ή C αφού από τη λειτουργία του αλγορίθμου σε κάθε γύρο επικοινωνίας όλοι οι κόμβοι που έχουν τέτοιες πληροφορίες θα τις στείλουν στον κόμβο που θα επιλέξουν στον τρέχοντα γύρο. Επομένως αφού βρω το γύρο επικοινωνίας στον οποίο ο αλγόριθμος ολοκλήρωσε τη λειτουργία του παίρνω τις αντίστοιχες μετρήσεις προσομοίωσης.

5.3.2 Αποτελέσματα Πειραματικής Αξιολόγησης

Θεωρητικά ο αλγόριθμος ενημερώνει όλους τους κόμβους του συστήματος με όλες τις πληροφορίες σε $O(\ln n)$ γύρους επικοινωνίας και χρησιμοποιώντας $O(n \ln \ln n)$ μηνύματα, όπου n ο αριθμός των κόμβων. Τα δύο αυτά άνω φράγματα αποδείχθηκαν αυστηρότυπα στο [1]. Σκοπός της εργασίας μας είναι να συμπεράνουμε κατά πόσον οι θεωρητικές αυτές μετρήσεις επιβεβαιώνονται στην πράξη.

Ακολουθούν οι τελικές μετρήσεις της προσομοίωσης του αλγορίθμου που πάρθηκαν από το μέσο όρο των πέντε προσομοιώσεων της κάθε μιας από τις πιο κάτω περιπτώσεις. Η στήλη 'Nodes' αντιπροσωπεύει τον αριθμό των κόμβων στην προσομοίωση. Η στήλη 'Rounds' αντιπροσωπεύει τον αριθμό των γύρων επικοινωνίας που χρειάστηκε να τρέξει ο αλγόριθμος μέχρι να ολοκληρώσει την εκτέλεσή του, δηλαδή είναι ο γύρος στον οποίο στάλθηκαν τα τελευταία μηνύματα στην προσομοίωση. Η στήλη 'Time (msec)' αντιπροσωπεύει το χρόνο (σε χιλιοστά του δευτερολέπτου) που πήρε για να ολοκληρωθεί ο αλγόριθμος, δηλαδή μετριέται από την

αρχή της εκτέλεσης της προσομοίωσης μέχρι το τέλος του γύρου στον οποίο ολοκληρώνεται ο αλγόριθμος. Η στήλη 'Messages' αντιπροσωπεύει το συνολικό αριθμό μηνυμάτων που στάλθηκαν κατά τη διάρκεια της προσομοίωσης, δηλαδή είναι ο αριθμός μηνυμάτων που αποστέλλονται από τον πρώτο γύρο προσομοίωσης μέχρι και τον γύρο στον οποίο ολοκληρώνεται ο αλγόριθμος.

Nodes	Rounds	Time (msec)	Messages
20	12,60	296,40	481,40
40	13,40	602,40	1027,00
60	14,00	1117,80	1579,60
80	14,40	1792,00	2170,20
100	14,60	2610,40	2775,20
120	14,80	3666,00	3360,80
140	15,00	4909,80	3952,80
160	15,00	6486,00	4549,80
180	15,00	8248,20	5150,00
200	15,00	10458,00	5835,60
220	15,40	13300,00	6475,00
240	15,40	22964,00	7113,20

Πίνακας 5.1 Αλγόριθμος MEDIAN-COUNTER

5.3.3 Ανάλυση Αποτελεσμάτων

Οι μετρήσεις που πάρθηκαν από την προσομοίωση του αλγορίθμου οδήγησαν στις πιο κάτω παρατηρήσεις ως προς τις τρεις παραμέτρους αξιολόγησης του αλγορίθμου, τους γύρους επικοινωνίας, το χρόνο εκτέλεσης και τον αριθμό μηνυμάτων:

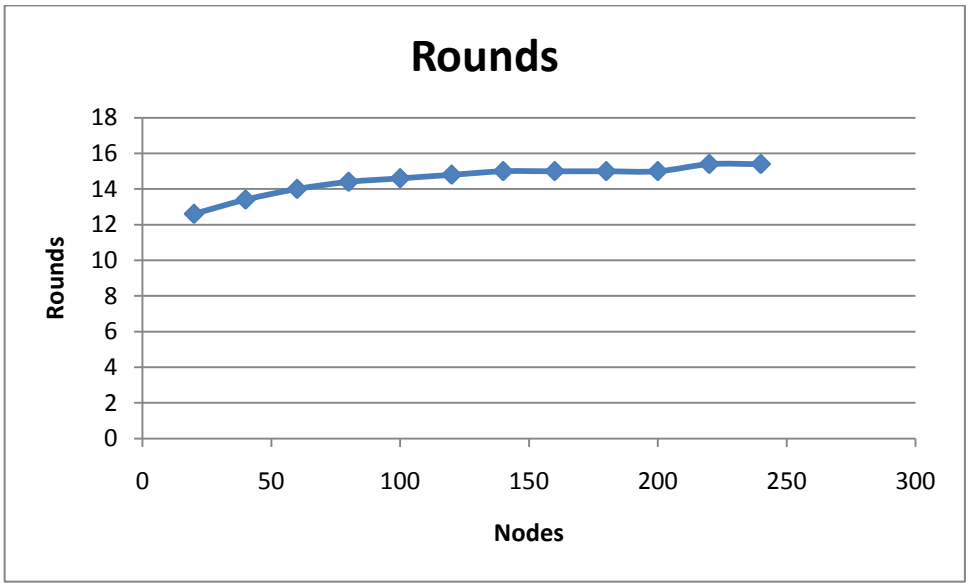
Γύροι Επικοινωνίας

Θεωρητικά ο αριθμός των γύρων επικοινωνίας στην εκτέλεση του αλγορίθμου είναι λογαριθμικός συναρτήσει των κόμβων τους συστήματος και είναι ίσος με $O(\ln n)$, όπου n ο αριθμός των κόμβων.

Από τον πίνακα 5.1 παρατηρούμε ότι ο αριθμός των γύρων επικοινωνίας που τρέχει ο αλγόριθμος μέχρι να ενημερωθούν όλοι οι κόμβοι μεγαλώνει καθώς μεγαλώνει ο αριθμός των κόμβων του συστήματος. Ο ρυθμός όμως με τον οποίο αυξάνονται οι γύροι επικοινωνίας δεν είναι μεγάλος. Συγκεκριμένα παρατηρούμε ότι όσο ο αριθμός των κόμβων είναι μικρός, ο αριθμός των γύρων αυξάνεται με μεγαλύτερο ρυθμό ενώ

όσο ο αριθμός των κόμβων μεγαλώνει, ο ρυθμός αύξησης των γύρων επικοινωνίας μικραίνει. Για παράδειγμα όταν οι κόμβοι αυξήθηκαν από 20 σε 80 έχουμε αύξηση στους γύρους επικοινωνίας ίση με 1,8 γύρους, ενώ όταν οι κόμβοι αυξάνονται από 180 σε 240 η αύξηση στους γύρους επικοινωνίας είναι πολύ πιο μικρή αφού είναι ίση με 0,4 γύρους. Αυτή η συμπεριφορά μοιάζει να είναι λογαριθμική επομένως αναμένουμε μία λογαριθμική γραφική παράσταση η οποία θα επιβεβαιώνει το θεωρητικό άνω φράγμα για τους γύρους επικοινωνίας που είναι $O(\ln n)$.

Πιο κάτω φαίνεται η γραφική παράσταση του αριθμού των γύρων επικοινωνίας συναρτήσει του αριθμού των κόμβων του συστήματος:



Σχήμα 5.1 Αλγόριθμος MEDIAN-COUNTER – Γύροι Επικοινωνίας

Όντως, η γραφική παράσταση που πάρθηκε από τις μετρήσεις για τους γύρους επικοινωνίας είναι λογαριθμική σε σχέση με τον αριθμό των κόμβων. Συνεπώς, επιβεβαιώνεται στην πράξη το θεωρητικό άνω φράγμα $O(\ln n)$ για τους γύρους επικοινωνίας που χρειάζεται να τρέξει ο αλγόριθμος για να ενημερωθούν όλοι οι κόμβοι του συστήματος.

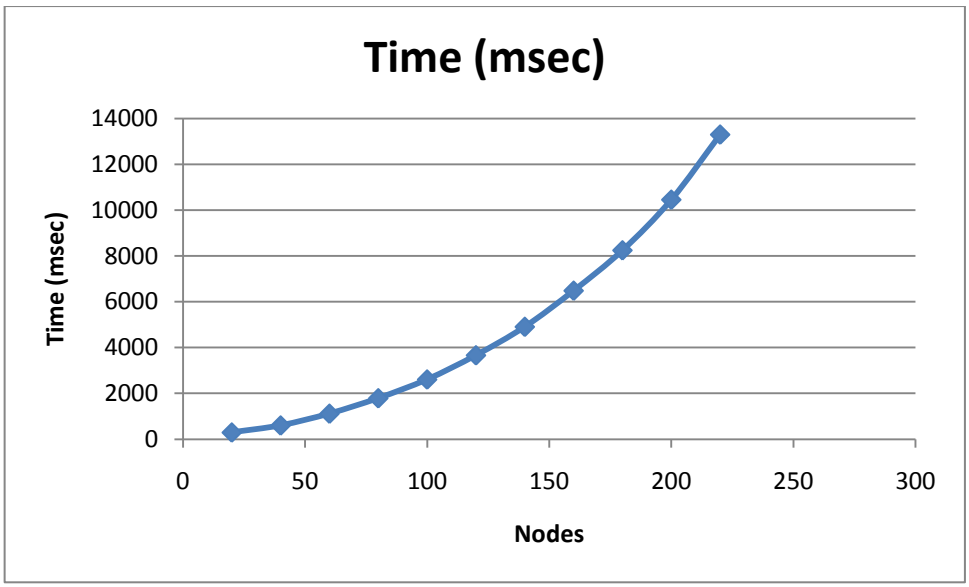
Χρόνος Εκτέλεσης

Ο χρόνος εκτέλεσης του αλγορίθμου αναμέναμε να είναι ανάλογος των γύρων επικοινωνίας. Για να συμβεί όμως αυτό πρέπει οι κόμβοι του συστήματος να σπαταλούν σταθερό χρόνο σε κάθε γύρο επικοινωνίας. Αυτό συμβαίνει μόνο στην

περίπτωση που οι κόμβοι δεν επεξεργάζονται καθόλου τις πληροφορίες που παραλαμβάνουν. Επειδή όμως για τους σκοπούς της εφαρμογής μας οι κόμβοι του συστήματος επεξεργάζονται όλες τις πληροφορίες που παραλαμβάνουν, ο χρόνος που χρειάζεται ο κάθε γύρος επικοινωνίας δεν είναι σταθερός αλλά εξαρτάται από το πλήθος των πληροφοριών που παραλαμβάνει ο κάθε κόμβος στον τρέχοντα γύρο. Συνεπώς ο χρόνος εκτέλεσης του αλγορίθμου δεν αναμένεται να είναι λογαριθμικός ως προς των αριθμό των κόμβων, όπως συμβαίνει με τους γύρους επικοινωνίας, αλλά αναμένεται να είναι πολύ μεγαλύτερος.

Όπως παρατηρούμε στον πίνακα 5.1, ο χρόνος εκτέλεσης μεγαλώνει όσο μεγαλώνει ο αριθμός των κόμβων. Ο ρυθμός όμως με τον οποίο αυξάνεται ο χρόνος εκτέλεσης του αλγορίθμου δεν είναι σταθερός, αλλά μεγαλώνει όσο αυξάνονται οι κόμβοι στο σύστημα. Για παράδειγμα όταν οι κόμβοι αυξήθηκαν από 20 σε 80 έχουμε αύξηση στον χρόνο εκτέλεσης ίση με 1495,6 χιλιοστά του δευτερολέπτου, ενώ όταν οι κόμβοι αυξάνονται από 160 σε 220 η αύξηση στον χρόνο εκτέλεσης είναι πολύ πιο μεγάλη αφού είναι ίση με 6814 χιλιοστά του δευτερολέπτου. Επομένως, ο χρόνος εκτέλεσης του αλγορίθμου αυξάνεται με μεγάλους ρυθμούς.

Πιο κάτω φαίνεται η γραφική παράσταση του χρόνου εκτέλεσης του αλγορίθμου συναρτήσει του αριθμού των κόμβων του συστήματος:



Σχήμα 5.2 Αλγόριθμος MEDIAN-COUNTER – Χρόνος Εκτέλεσης

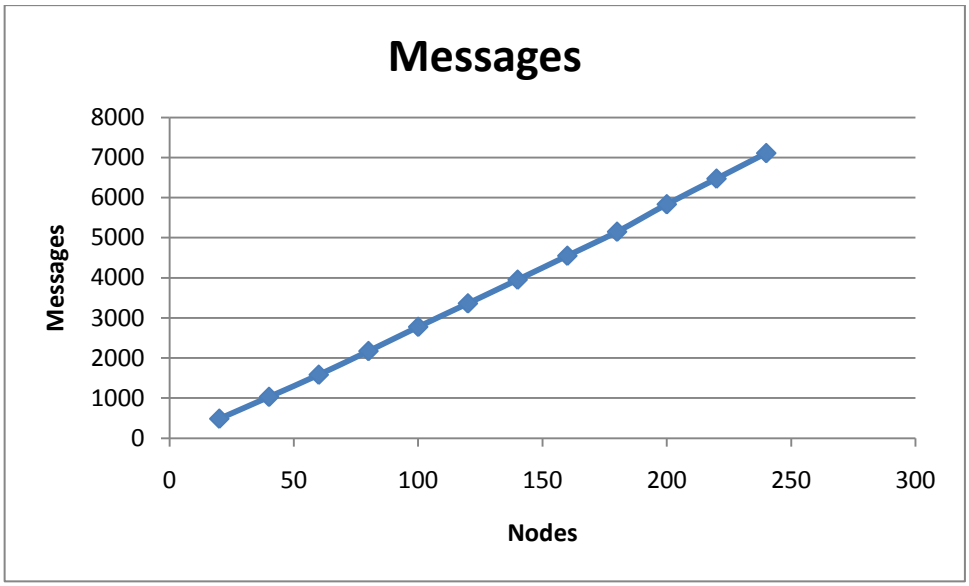
Όντως, όπως φαίνεται και στη γραφική παράσταση, ο χρόνος εκτέλεσης του αλγορίθμου δεν αυξάνεται λογαριθμικά ως προς τον αριθμό των κόμβων του συστήματος, όπως συμβαίνει με τους γύρους επικοινωνίας, και ούτε αυξάνεται γραμμικά, όπως συμβαίνει με τον αριθμό μηνυμάτων. Ο χρόνος εκτέλεσης αυξάνεται με μεγαλύτερους ρυθμούς οι οποίοι ξεπερνούν το άνω φράγμα $O(n \log n)$, όπου n ο αριθμός των κόμβων. Αυτό οφείλεται στο χρόνο που σπαταλούν οι κόμβοι σε κάθε γύρο επικοινωνίας για να επεξεργαστούν τις πληροφορίες που παραλαμβάνουν αλλά και για την εκτέλεση άλλων λειτουργιών.

Μηνύματα

Θεωρητικά ο συνολικός αριθμός μηνυμάτων που αποστέλλονται κατά τη διάρκεια εκτέλεσης του αλγορίθμου είναι ανάλογος του αριθμού των κόμβων του συστήματος και συγκεκριμένα είναι της τάξεως $O(n \ln \ln n)$, όπου n ο αριθμός των κόμβων.

Από τον πίνακα 5.1 παρατηρούμε ότι ο αριθμός των μηνυμάτων μεγαλώνει καθώς μεγαλώνει ο αριθμός των κόμβων του συστήματος. Αυτό είναι λογικό αφού για να ενημερωθούν περισσότεροι κόμβοι θα πρέπει να σταλούν περισσότερα μηνύματα. Για παράδειγμα όταν οι κόμβοι αυξήθηκαν από 20 σε 100 έχουμε αύξηση στον αριθμό των μηνυμάτων ίση με 2293,8 μηνύματα, ενώ όταν οι κόμβοι αυξάνονται από 160 σε 240 η αύξηση στον αριθμό των μηνυμάτων είναι ίση με 2563,4 μηνύματα. Επομένως παρατηρούμε ότι ο ρυθμός αύξησης του αριθμού των μηνυμάτων σε συνάρτηση με τους κόμβους είναι περίπου σταθερός και ότι ο αριθμός των μηνυμάτων είναι περίπου ανάλογος του αριθμού των κόμβων. Αυτή η συμπεριφορά μοιάζει να είναι γραμμική επομένως αναμένουμε μία γραφική παράσταση η οποία θα επιβεβαιώνει το θεωρητικό άνω φράγμα για τον αριθμό των μηνυμάτων που είναι $O(n \ln \ln n)$.

Πιο κάτω φαίνεται η γραφική παράσταση του συνολικού αριθμού των μηνυμάτων συναρτήσει του αριθμού των κόμβων του συστήματος:



Σχήμα 5.3 Αλγόριθμος MEDIAN-COUNTER – Αριθμός Μηνυμάτων

Όντως, η γραφική παράσταση που πάρθηκε από τις μετρήσεις για αριθμό των μηνυμάτων είναι σχεδόν γραμμική σε σχέση με τον αριθμό των κόμβων. Φαίνεται να υπάρχει μία πολύ μικρή καμπύλη στην ευθεία της γραφικής παράστασης η οποία όμως ήταν αναμενόμενη αφού από τις μετρήσεις μας ο αριθμός των μηνυμάτων δεν ήταν ακριβώς ανάλογος του αριθμού των κόμβων, αλλά είχε μία μικρή διαφορά. Η διαφορά αυτή δημιουργεί την μικρή καμπύλη που υπάρχει στην γραφική και δικαιολογεί το ‘ln ln n’ στο θεωρητικό άνω φράγμα. Συνεπώς, επιβεβαιώνεται στην πράξη το θεωρητικό άνω φράγμα $O(n \ln \ln n)$ για το συνολικό αριθμό μηνυμάτων που χρειάζεται να σταλούν κατά την εκτέλεση του αλγορίθμου έτσι ώστε να ενημερωθούν όλοι οι κόμβοι του συστήματος.

5.4 Συμπεράσματα

Αφού μελετήσαμε ξεχωριστά τις βασικές παραμέτρους του αλγορίθμου, καταλήγουμε στα πιο κάτω συμπεράσματα:

Καθώς αυξάνεται ο αριθμός των κόμβων στην προσομοίωση, αυξάνονται οι γύροι επικοινωνίας που πρέπει να τρέξει ο αλγόριθμος ώστε να ολοκληρώσει τη λειτουργία του, αυξάνεται ο χρόνος εκτέλεσης του αλγορίθμου και τέλος αυξάνεται και ο συνολικός αριθμός των μηνυμάτων που ανταλλάσσονται. Κάθε μια από τις τρεις αυτές

παραμέτρους αυξάνεται με διαφορετικό ρυθμό και μας οδηγεί σε διαφορετικά συμπεράσματα.

Ο αριθμός των γύρων επικοινωνίας αυξάνεται λογαριθμικά ως προς τον αριθμό των κόμβων. Η λογαριθμική αύξηση των γύρων επικοινωνίας είναι πολύ αποδοτική αφού για μεγάλο αριθμό κόμβων στο σύστημα, η περεταίρω αύξηση στον αριθμό των κόμβων δεν θα οδηγήσει σε μεγάλες αλλαγές στον αριθμό των γύρων επικοινωνίας. Ο χρόνος όμως εκτέλεσης του αλγορίθμου δεν είναι αρκετά καλός αφού είναι αυτός που αυξάνεται με το μεγαλύτερο ρυθμό. Παρόλα αυτά ο χρόνος εκτέλεσης δεν μας απασχολεί τόσο γιατί δεν εξαρτάται μόνο από το χρόνο που χρειάζεται μέχρι να ενημερωθούν όλοι οι κόμβοι του συστήματος αλλά εξαρτάται και από τις επιπρόσθετες λειτουργίες που ο κάθε προγραμματιστής προσθέτει στην εφαρμογή του. Γι' αυτό και για την αξιολόγηση του αλγορίθμου χρησιμοποιείται κυρίως ο αριθμός των γύρων επικοινωνίας αντί ο χρόνος εκτέλεσης.

Η διάδοση των μηνυμάτων και προς τις δύο κατευθύνσεις συντείνει στη μείωση του συνολικού αριθμού των μηνυμάτων που ανταλλάσσονται κατά τη διάρκεια του αλγορίθμου ο οποίος είναι τάξεως $O(n \ln \ln n)$. Ο αριθμός των μηνυμάτων είναι πολύ ικανοποιητικός γιατί αυξάνεται σχεδόν γραμμικά συναρτήσει των κόμβων του συστήματος αφού η τιμή '$\ln \ln n$' είναι αρκετά μικρή και σχεδόν καλύπτεται από την τιμή n. Επίσης, ένα σημαντικό πλεονέκτημα του αλγορίθμου είναι ότι οι κόμβοι δεν διαδίδουν όλες τις πληροφορίες που γνωρίζουν αλλά μόνο αυτές που βρίσκονται σε κατάσταση B ή C και έτσι το μέγεθος των μηνυμάτων δεν είναι πολύ μεγάλο και δεν επιβαρύνει την επικοινωνία του δικτύου με επιπλέον φόρτο. Το μέγεθος των μηνυμάτων του αλγορίθμου θα μελετηθεί στο Κεφάλαιο 7, για να μπορέσει να συγκριθεί με το μέγεθος των μηνυμάτων των υπολοίπων αλγορίθμων και έτσι να οδηγήσει σε εξαγωγή καλύτερων συμπερασμάτων ως προς την παράμετρο αυτή.

Ένα καθοριστικό στοιχείο του αλγόριθμου είναι ότι οι κόμβοι που βρίσκονται στην κατάσταση B δεν αυξάνουν τον μετρητή τους σε κάθε γύρο επικοινωνίας αλλά τον αυξάνουν μόνο στην περίπτωση που παραλάβουν σε ένα γύρο περισσότερα αντίγραφα της ίδιας πληροφορίας με μεγαλύτερο μετρητή παρά με μικρότερο. Αυτός ο κανόνας βοηθά τον αλγόριθμο να τερματίζει στην κατάλληλη στιγμή χωρίς όμως να προκαθορίζει την ακριβή στιγμή που θα τερματίσει. Έτσι αναμένουμε ο αλγόριθμος να είναι ανεκτικός σε σφάλματα δικτύου, κάτι που θα μελετήσουμε στη συνέχεια.

Τέλος, οι πιο πάνω παρατηρήσεις μας οδηγούν στο γενικότερο συμπέρασμα ότι ο παρόν αλγόριθμος πληροφόρησης είναι αποτελεσματικός και αρκετά αποδοτικός. Ακόμη καταλήγουμε στο συμπέρασμα ότι η πρακτική ανάλυση του αλγορίθμου συναύει με τη θεωρητική αφού τόσο θεωρητικά όσο και πρακτικά ο αλγόριθμος ολοκληρώνεται σε $O(\ln n)$ γύρους επικοινωνίας και χρησιμοποιώντας $O(n \ln \ln n)$ μηνύματα.

ΚΕΦΑΛΑΙΟ 6

Αλγόριθμος MEDIAN-COUNTER Με Σφάλματα

6.1 Ευρωστία Αλγορίθμου MEDIAN-COUNTER

Ο αλγόριθμος MEDIAN-COUNTER, όπως αξιολογήθηκε στο Κεφάλαιο 5, είναι ένας αρκετά αποδοτικός αλγόριθμος πληροφόρησης στην περίπτωση όπου δεν υπάρχουν καθόλου σφάλματα στο δίκτυο. Επιτυγχάνει να ενημερώσει όλους τους κόμβους του συστήματος σε $O(\ln n)$ γύρους επικοινωνίας και χρησιμοποιώντας $O(n \ln \ln n)$ μηνύματα. Τι συμβαίνει όμως στην πιο ρεαλιστική περίπτωση κατά την οποία υπάρχουν σφάλματα στο δίκτυο; Εξακολουθεί να ενημερώνει όλους τους κόμβους του συστήματος με όλες τις πληροφορίες; Αν ναι, τότε με τι χρόνο και κόστος το επιτυγχάνει; Αυτό είναι το θέμα που θα μας απασχολήσει στη συνέχεια του κεφαλαίου αυτού.

Θεωρητικά, ο αλγόριθμος MEDIAN-COUNTER είναι ανεκτικός σε διαφόρων ειδών σφάλματα δικτύου. Υποθέτουμε μία τυχαία κατανομή με βάση την οποία θα

συμβαίνουν τα σφάλματα και η οποία θα οδηγήσει τελικά σε f το πολύ σφάλματα δικτύου. Τότε, όπως αποδείκτηκε αυστηρότυπα στο [1], ο αλγόριθμος θα ενημερώσει όλους τους κόμβους του συστήματος εκτός από $O(f)$ κόμβους σε $O(\ln n)$ γύρους επικοινωνίας και χρησιμοποιώντας $O(n \ln \ln n)$ μηνύματα, όπου n ο αριθμός των κόμβων.

Αναμένουμε ότι όντως ο αλγόριθμος θα είναι ανεκτικός στα σφάλματα λόγω του ευέλικτου μηχανισμού που χρησιμοποιεί για να τερματίσει. Η κρίσιμη κατάσταση μίας πληροφορίας, η κατάσταση δηλαδή κατά την οποία η πληροφορία θεωρείται πρόσφατη είναι η κατάσταση Β. Κατά τη διάρκεια που ένας κόμβος βρίσκεται σε αυτήν την κατάσταση ως προς μια πληροφορία, ο μετρητής του αυξάνεται με βάση τον κανόνα μεσαίου (Median rule) ο οποίος είναι αυτός που προσφέρει την ευρωστία στον αλγόριθμο. Ο κανόνας μεσαίου οδηγεί ένα κόμβο που βρίσκεται σε κατάσταση Β ως προς κάποια πληροφορία να αυξήσει τον μετρητή του μόνο στην ακόλουθη περίπτωση: Ο κόμβος έχει παραλάβει την πληροφορία αυτή στον τρέχοντα γύρο επικοινωνίας από περισσότερους κόμβους που βρίσκονται σε κατάσταση Β με μετρητή μεγαλύτερο ή ίσο με τον δικό του παρά από κόμβους σε κατάσταση Β αλλά με μετρητή μικρότερο από τον δικό του.

Αυτός ο κανόνας συνεπάγεται ότι ο μετρητής του κόμβου σε κατάσταση Β δεν αυξάνεται σε κάθε γύρο επικοινωνίας. Επομένως η στιγμή που μια πληροφορία σταματά να βρίσκεται στην κρίσιμη κατάσταση Β, δηλαδή η στιγμή που ο μετρητής του κόμβου αυξήθηκε στο μέγιστο όριο, δεν είναι προκαθορισμένη αλλά εξαρτάται από την πορεία της εκτέλεσης του αλγορίθμου. Γι’ αυτό το λόγο αναμένουμε ότι τα σφάλματα κατά την εκτέλεση του αλγορίθμου θα οδηγήσουν τους κόμβους στο να παραλάβουν λιγότερες πληροφορίες σε κάποιο γύρο και επομένως οι κόμβοι που βρίσκονται σε κατάσταση Β πιθανόν να καθυστερήσουν την αύξηση των μετρητών τους. Αυτό όμως δε θα οδηγήσει στο να τερματίσει ο αλγόριθμος πριν να ενημερωθούν οι κόμβοι του συστήματος γιατί όσο υπάρχουν κόμβοι στην κατάσταση Β ή C ο αλγόριθμος δεν τερματίζει αλλά συνεχίζει να μεταδίδει τις πληροφορίες.

Στη συνέχεια θα υλοποιηθεί και θα αξιολογηθεί ο αλγόριθμος MEDIAN-COUNTER σε δύο περιβάλλοντα προσομοίωσης τα οποία θα υπόκεινται σε σφάλματα για να αποφασιστεί κατά πόσο ο αλγόριθμος είναι και πρακτικά εκτός από θεωρητικά εύρωστος. Τα δύο είδη σφαλμάτων που θα υλοποιηθούν είναι τα σφάλματα στην εγκατάσταση σύνδεσης

μεταξύ δύο κόμβων του συστήματος και η κατάρρευση κόμβων στο σύστημα. Και στις δύο περιπτώσεις τα σφάλματα θα συμβαίνουν σε τυχαίες στιγμές και σε τυχαίους κόμβους με βάση κάποια προκαθορισμένη πιθανότητα.

6.2 Αλγόριθμος MEDIAN-COUNTER Με Σφάλματα Συνδέσεων

Η πρώτη περίπτωση σφαλμάτων που θα μελετηθεί είναι τα σφάλματα συνδέσεων. Οι κόμβοι ενός δικτύου που υπόκειται σε τέτοιου είδους σφάλματα δεν επιτυγχάνουν πάντα να εγκαταστήσουν σύνδεση με τον γείτονα που επιθυμούν. Δηλαδή υπάρχει κάποια πιθανότητα να αποτύχει η προσπάθεια για εγκατάσταση σύνδεσης μεταξύ δύο κόμβων και το γεγονός αυτό θα εμποδίσει τους κόμβους από το να επικοινωνήσουν και να ανταλλάξουν πληροφορίες.

6.2.1 Περιγραφή Αλγορίθμου

Ο αλγόριθμος λειτουργεί με τον ίδιο τρόπο όπως και ο αλγόριθμος MEDIAN-COUNTER χωρίς σφάλματα με τη μόνη διαφορά ότι οι συνδέσεις που επιλέγουν οι κόμβοι να εγκαταστήσουν σε κάθε γύρο επικοινωνίας δεν εγκαθίστανται πάντα. Αντί αυτού οι συνδέσεις εγκαθίστανται χρησιμοποιώντας μία αυθαίρετη κατανομή πιθανοτήτων. Η κατανομή πιθανοτήτων επιστρέφει αποτέλεσμα 1 ή 0 και έτσι καθορίζεται αν θα εγκατασταθεί ή όχι μία σύνδεση. Για παράδειγμα έστω ότι ένας κόμβος u επιλέγει σε κάποιο γύρο επικοινωνίας έναν άλλο κόμβο v για να του στείλει πληροφορίες και επίσης για να λάβει πληροφορίες από αυτόν. Τότε υπάρχει πιθανότητα να μην διεκπεραιωθεί η επικοινωνία μεταξύ των κόμβων u και v λόγω του ότι η μεταξύ τους σύνδεση δεν εγκαταστάθηκε επιτυχώς.

6.2.2 Υλοποίηση Αλγορίθμου

Η υλοποίηση του αλγορίθμου είναι όμοια με αυτήν του αλγορίθμου MEDIAN-COUNTER χωρίς σφάλματα. Η μόνη διαφορά των δύο υλοποιήσεων βρίσκεται στα σημεία της εφαρμογής όπου εγκαθίσταται σύνδεση μεταξύ δύο κόμβων του συστήματος για να ανταλλάξουν πληροφορίες μέσω των λειτουργιών ώθησης (push) και λήψης (pull). Η εγκατάσταση των συνδέσεων αυτών επιλέξαμε να αποτυγχάνει με πιθανότητα 15%. Επιλέχτηκε αυτή η πιθανότητα παρόλο που είναι αρκετά μεγαλύτερη από τη μέση

πιθανότητα σφαλμάτων όπως αναφέρεται στο [7] για να ελεγχθεί αν ο αλγόριθμος παραμένει εύρωστος ακόμη και αν ο αριθμός των σφαλμάτων μεγαλώσει τόσο πολύ.

Ενεργό μέρος

Στο ενεργό μέρος της εφαρμογής υπάρχει η εξής διαφοροποίηση κατά τη διαδικασία ώθησης (push)

```
protected void periodicRun() {

   . . .

   double fail=Math.random();

   . . .

   if (fail>0.15)
     getCommLayer().sendUDP(data, neighbour);
}
```

Όπου η συνάρτηση ‘`Math.random()`’ χρησιμοποιείται για την αυθαίρετη κατανομή πιθανοτήτων. Το αποτέλεσμα της συνάρτησης είναι ένας τυχαίος πραγματικός αριθμός μεταξύ 0 και 1 και φυλάγεται στη μεταβλητή ‘`fail`’. Επειδή θέλουμε τα σφάλματα στην εγκατάσταση σύνδεσης να εμφανίζονται με πιθανότητα 15%, η ώθηση των δεδομένων ‘`data`’ στο γείτονα ‘`neighbour`’ δεν θα εκτελεστεί στην περίπτωση που η τιμή της μεταβλητής ‘`fail`’ είναι μικρότερη ή ίση με 0.15.

Παθητικό μέρος

Στο παθητικό μέρος της εφαρμογής υπάρχει η εξής διαφοροποίηση κατά τη διαδικασία ώθησης (pull)

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

   . . .

   double fail=Math.random();

   . . .

   if (fail>0.15)
     getCommLayer().sendUDP(data, neighbour);
}
```

Όπου η συνάρτηση ‘`Math.random()`’ χρησιμοποιείται για την αυθαίρετη κατανομή πιθανοτήτων. Το αποτέλεσμα της συνάρτησης είναι ένας τυχαίος πραγματικός αριθμός μεταξύ 0 και 1 και φυλάγεται στη μεταβλητή ‘`fail`’. Επειδή θέλουμε τα σφάλματα στην εγκατάσταση σύνδεσης να εμφανίζονται με πιθανότητα 15%, η λήψη των δεδομένων ‘`data`’ από τον γείτονα ‘`neighbour`’ δεν θα εκτελεστεί στην περίπτωση που η τιμή της μεταβλητής ‘`fail`’ είναι μικρότερη ή ίση με 0.15.

6.2.3 Πειραματική Αξιολόγηση Αλγορίθμου

Αφού υλοποιήθηκε ο αλγόριθμος συνεχίζουμε με την προσομοίωση και την πειραματική του αξιολόγηση.

Μεθοδολογία Πειραματικής Αξιολόγησης

Η μεθοδολογία της πειραματικής αξιολόγησης είναι ίδια με αυτήν της αξιολόγησης του αλγόριθμου MEDIAN-COUNTER χωρίς σφάλματα με τη διαφορά ότι έπρεπε να αποφασιστεί η πιθανότητα με την οποία θα συμβαίνουν τα σφάλματα στην εγκατάσταση των συνδέσεων της εφαρμογής. Η πιθανότητα αυτή αποφασίστηκε να είναι ίση με 15% γιατί τα σφάλματα στα μεγάλα δίκτυα δεν ξεπερνούν αυτή την πιθανότητα [7]. Έτσι θεωρήσαμε ότι το 15% είναι μία ικανοποιητικά μεγάλη πιθανότητα για την αξιολόγηση της εφαρμογής.

Αποτελέσματα Πειραματικής Αξιολόγησης

Θεωρητικά ο αλγόριθμος είναι ανεκτικός στα σφάλματα συνδέσεων και εξακολουθεί να ενημερώνει όλους εκτός από το 15% των κόμβων του συστήματος με όλες τις πληροφορίες σε $O(\ln n)$ γύρους επικοινωνίας και χρησιμοποιώντας $O(n \ln \ln n)$ μηνύματα, όπου n ο αριθμός των κόμβων. Τα δύο αυτά άνω φράγματα αποδείχθηκαν αυστηρότυπα στο [1]. Σκοπός της εργασίας μας είναι να συμπεράνουμε κατά πόσον οι θεωρητικές αυτές ιδιότητες του αλγορίθμου ως προς τα σφάλματα συνδέσεων επιβεβαιώνονται στην πράξη.

Ακολουθούν οι τελικές μετρήσεις της προσομοίωσης του αλγορίθμου που πάρθηκαν από το μέσο όρο των πέντε προσομοιώσεων της κάθε μιας από τις πιο κάτω περιπτώσεις. Η στήλη ‘Nodes’ αντιπροσωπεύει τον αριθμό των κόμβων στην

προσομοίωση. Η στήλη ‘Rounds’ αντιπροσωπεύει τον αριθμό των γύρων επικοινωνίας που χρειάστηκε να τρέξει ο αλγόριθμος μέχρι να ολοκληρώσει την εκτέλεσή του, δηλαδή είναι ο γύρος στον οποίο στάλθηκαν τα τελευταία μηνύματα στην προσομοίωση. Η στήλη ‘Time (msec)’ αντιπροσωπεύει το χρόνο (σε χιλιοστά του δευτερολέπτου) που πήρε για να ολοκληρωθεί ο αλγόριθμος, δηλαδή μετριέται από την αρχή της εκτέλεσης της προσομοίωσης μέχρι το τέλος του γύρου στον οποίο ολοκληρώνεται ο αλγόριθμος. Η στήλη ‘Messages’ αντιπροσωπεύει το συνολικό αριθμό μηνυμάτων που στάλθηκαν κατά τη διάρκεια της προσομοίωσης, δηλαδή είναι ο αριθμός μηνυμάτων που αποστέλλονται από τον πρώτο γύρο προσομοίωσης μέχρι και τον γύρο στον οποίο ολοκληρώνεται ο αλγόριθμος.

Nodes	Rounds	Time (msec)	Messages
20	14,40	259,00	423,00
40	16,00	555,60	903,40
60	17,20	1047,80	1411,40
80	17,80	1604,20	1921,20
100	18,20	2414,60	2460,00
120	18,40	3222,60	3024,80
140	18,40	4253,60	3530,20
160	18,80	5365,00	4041,80
180	18,80	7132,40	4701,80
200	19,00	8973,20	5243,00
220	19,40	11351,20	5831,00
240	19,40	15546,00	6493,00

Πίνακας 6.1 Αλγόριθμος MEDIAN-COUNTER με Σφάλματα Συνδέσεων

Ανάλυση Αποτελεσμάτων

Οι μετρήσεις που πάρθηκαν από την προσομοίωση του αλγορίθμου οδήγησαν στις πιο κάτω παρατηρήσεις ως προς τις τρεις παραμέτρους αξιολόγησης του αλγορίθμου, τους γύρους επικοινωνίας, το χρόνο εκτέλεσης και τον αριθμό μηνυμάτων:

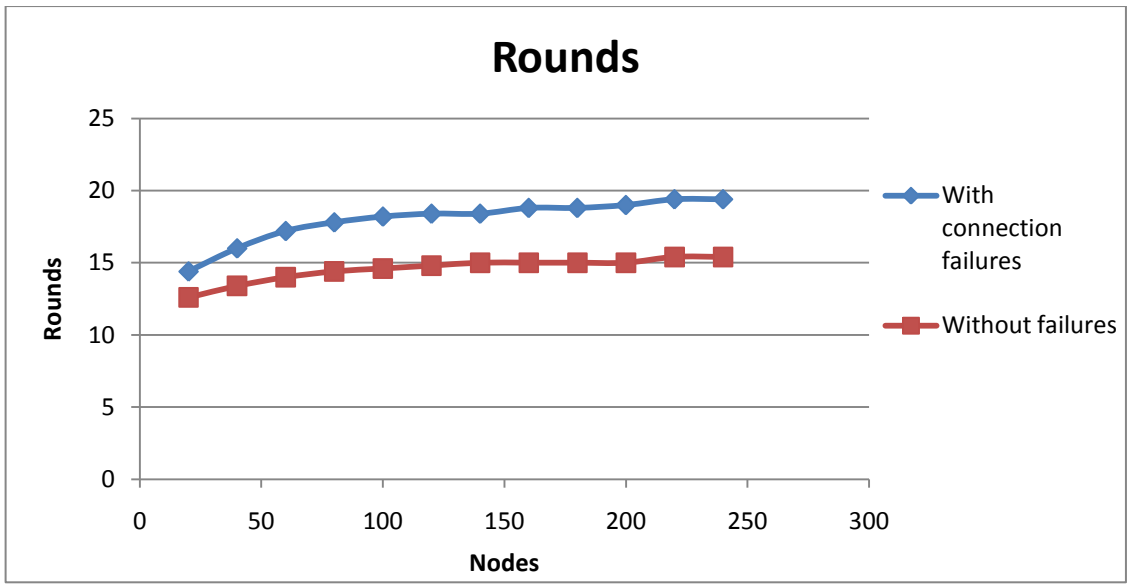
Γύροι Επικοινωνίας

Θεωρητικά ο αριθμός των γύρων επικοινωνίας στην εκτέλεση του αλγορίθμου με σφάλματα συνδέσεων διατηρεί τον λογαριθμικό ρυθμό αύξησης ως προς τον αριθμό των κόμβων του συστήματος. Αναμένουμε όμως ότι ο αριθμός των γύρων επικοινωνίας

στην περίπτωση των σφαλμάτων συνδέσεων θα είναι μεγαλύτερος από την περίπτωση όπου δεν υπάρχουν σφάλματα. Αυτό θα ήταν λογικό αφού η αποτυχία στην εγκατάσταση των συνδέσεων οδηγεί σε μείωση του αριθμού των μηνυμάτων που ανταλλάσσονται μεταξύ των κόμβων και ως αποτέλεσμα αναμένουμε κάποια καθυστέρηση στην ενημέρωσή τους.

Όπως παρατηρούμε στον πίνακα 6.1 η αύξηση στον αριθμό των γύρων επικοινωνίας φαίνεται να είναι λογαριθμική αφού όσο μεγαλώνει το πλήθος των κόμβων, μειώνεται ο ρυθμός αύξησης στον αριθμό των γύρων. Επίσης παρατηρούμε ότι στην περίπτωση της προσομοίωσης της εφαρμογής χωρίς σφάλματα με 20 κόμβους χρειάστηκαν κατά μέσο όρο 12,6 γύροι μέχρι να τερματίσει ο αλγόριθμος ενώ στην περίπτωση με τα σφάλματα χρειάστηκαν κατά μέσο όρο 14,40 γύροι. Κάτι ανάλογο ισχύει και για τις υπόλοιπες περιπτώσεις προσομοίωσης, όταν το πλήθος των κόμβων αυξάνεται. Οπότε είναι προφανές ότι στην περίπτωση των σφαλμάτων ο αριθμός των γύρων επικοινωνίας είναι μεγαλύτερος.

Πιο κάτω φαίνεται η γραφική παράσταση του αριθμού των γύρων επικοινωνίας συναρτήσει του αριθμού των κόμβων του συστήματος. Η γραφική παράσταση αναπαριστά την περίπτωση του αλγορίθμου MEDIAN-COUNTER με σφάλματα συνδέσεων καθώς και την περίπτωση του ίδιου αλγορίθμου χωρίς σφάλματα:



Σχήμα 6.1 Αλγόριθμος MEDIAN-COUNTER με Σφάλματα Συνδέσεων – Γύροι Επικοινωνίας

Όντως, η γραφική παράσταση των γύρων επικοινωνίας στην περίπτωση όπου υπάρχουν σφάλματα συνδέσεων εξακολουθεί να είναι λογαριθμική σε σχέση με τον αριθμό των κόμβων. Επίσης παρατηρούμε ότι, όπως ήταν αναμενόμενο, στην περίπτωση των σφαλμάτων ο αριθμός των γύρων επικοινωνίας είναι μεγαλύτερος από αυτόν στην περίπτωση όπου δεν υπάρχουν σφάλματα. Αρχικά, όταν δηλαδή ο αριθμός των κόμβων της εφαρμογής είναι ίσος με 20, ο αριθμός των γύρων επικοινωνίας στην περίπτωση των σφαλμάτων είναι κατά 15% μεγαλύτερος από τον αριθμό των γύρων επικοινωνίας του αλγορίθμου χωρίς σφάλματα, αφού αυξήθηκε από 12,6 σε 14,4 γύρους. Καθώς όμως αυξάνεται ο αριθμός των κόμβων, η διαφορά μεταξύ των γύρων επικοινωνίας στις δύο παραλλαγές του αλγορίθμου αυξάνεται. Συνεπώς παρόλο που τα σφάλματα συνδέσεων συμβαίνουν με πιθανότητα 15%, η αύξηση στους γύρους επικοινωνίας δεν είναι πάντα ίση με 15% αλλά, όπως παρατηρήθηκε, αρχικά είναι ίση με 15% και στη συνέχεια αυξάνεται καθώς αυξάνεται ο αριθμός των κόμβων. Αυτό πιθανότητα να συμβαίνει γιατί αυξάνεται ο αριθμός των κόμβων που πρέπει να ενημερωθούν και ταυτόχρονα αυξάνεται και ο συνολικός αριθμός σφαλμάτων αφού αυξάνονται οι γύροι επικοινωνίας.

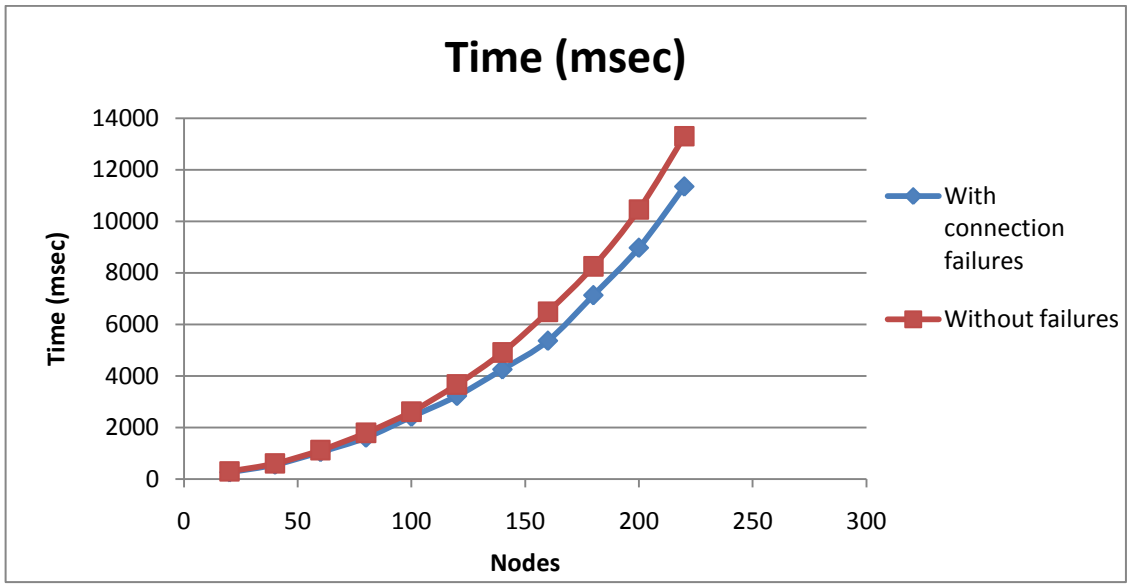
Παρόλα αυτά, και στις δύο παραλλαγές του αλγορίθμου, με σφάλματα συνδέσεων και χωρίς σφάλματα, ο αριθμός των γύρων επικοινωνίας αυξάνεται λογαριθμικά ως προς αριθμό των κόμβων. Αυτό συναύει με το θεωρητικό άνω φράγμα $O(\ln n)$ για τους γύρους επικοινωνίας που χρειάζεται να τρέξει ο αλγόριθμος στην περίπτωση των σφαλμάτων.

Χρόνος Εκτέλεσης

Ο χρόνος εκτέλεσης του αλγορίθμου αναμέναμε να είναι ανάλογος των γύρων επικοινωνίας. Όπως όμως παρατηρήσαμε και στην προσομοίωση του αλγορίθμου χωρίς σφάλματα, ο χρόνος εκτέλεσης δεν είναι ανάλογος των γύρων επικοινωνίας αλλά είναι πολύ μεγαλύτερος λόγω του ότι οι κόμβοι σπαταλούν πολύ χρόνο στον κάθε γύρο για να επεξεργαστούν όλες τις πληροφορίες παραλαμβάνουν από τους γείτονές τους. Κάτι παρόμοιο αναμένουμε να συμβεί και σε αυτή την περίπτωση προσομοίωσης όπου υπάρχουν τα σφάλματα συνδέσεων αφού οι κόμβοι εξακολουθούν να επεξεργάζονται όλες τις πληροφορίες που παραλαμβάνουν.

Όπως παρατηρούμε στον πίνακα 6.1, ο χρόνος εκτέλεσης μεγαλώνει όσο μεγαλώνει ο αριθμός των κόμβων. Ο ρυθμός όμως με τον οποίο αυξάνεται ο χρόνος εκτέλεσης του αλγορίθμου δεν είναι σταθερός, αλλά μεγαλώνει όσο αυξάνονται οι κόμβοι στο σύστημα. Επομένως, ο χρόνος εκτέλεσης του αλγορίθμου δεν είναι λογαριθμικός ως προς τον αριθμό των κόμβων αλλά φαίνεται να είναι πολύ μεγαλύτερης τάξεως. Ακόμη παρατηρούμε ότι σε κάθε περίπτωση προσομοίωσης, ο αλγόριθμος χωρίς σφάλματα έχει μεγαλύτερο χρόνο εκτέλεσης από τον αλγόριθμο με σφάλματα συνδέσεων.

Πιο κάτω φαίνεται η γραφική παράσταση του χρόνου εκτέλεσης του αλγορίθμου συναρτήσει του αριθμού των κόμβων του συστήματος. Η γραφική παράσταση αναπαριστά την περίπτωση του αλγορίθμου MEDIAN-COUNTER με σφάλματα συνδέσεων καθώς και την περίπτωση του ίδιου αλγορίθμου χωρίς σφάλματα:



Σχήμα 6.2 Αλγόριθμος MEDIAN-COUNTER με Σφάλματα Συνδέσεων – Χρόνος Εκτέλεσης

Όντως, ο ρυθμός με τον οποίο αυξάνεται ο χρόνος εκτέλεσης του αλγορίθμου με τα σφάλματα συνδέσεων είναι της ίδιας τάξης με το ρυθμό με τον οποίο αυξάνεται ο χρόνος εκτέλεσης στον αλγόριθμο χωρίς τα σφάλματα. Αυτό ήταν αναμενόμενο αφού οι λειτουργίες που εκτελούν οι δύο αλγόριθμοι σε κάθε γύρο επικοινωνίας είναι περίπου οι ίδιες.

Το γεγονός ότι στην κάθε περίπτωση προσομοίωσης των δύο αλγορίθμων ο αλγόριθμος χωρίς σφάλματα έχει μεγαλύτερο χρόνο εκτέλεσης από ότι ο αλγόριθμος με τα σφάλματα συνδέσεων δεν ήταν αναμενόμενο. Αναμέναμε ότι αφού ο αλγόριθμος με τα

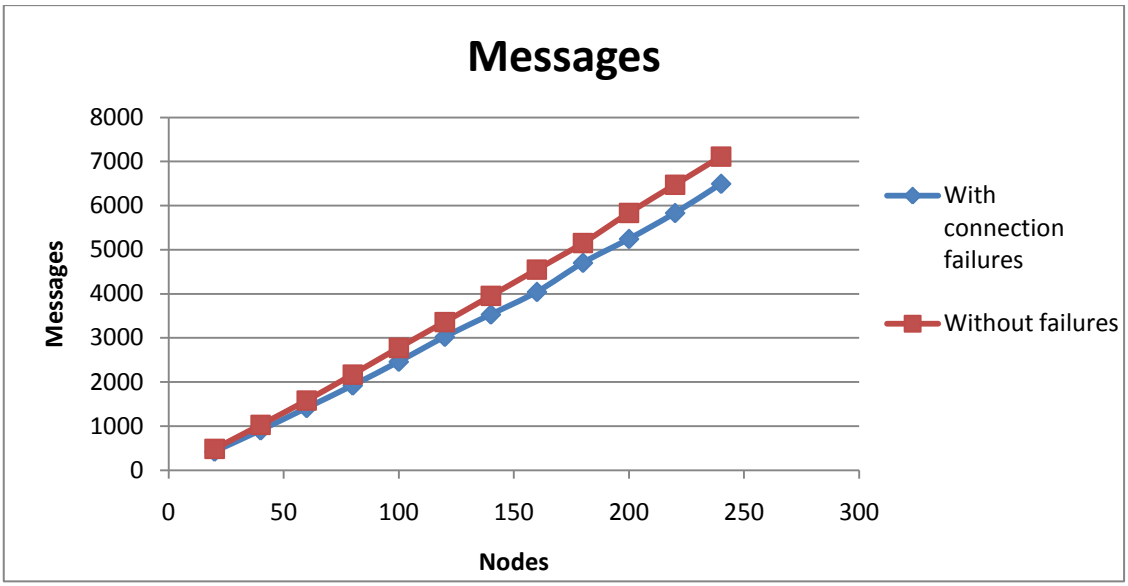
σφάλματα είχε μεγαλύτερο αριθμό γύρων επικοινωνίας σε κάθε περίπτωση προσομοίωσης θα είχε και μεγαλύτερο χρόνο εκτέλεσης. Απ' ότι φαίνεται όμως ο αριθμός των επιπλέον μηνυμάτων που αποστέλλονται στον αλγόριθμο με τα σφάλματα λόγω των περισσότερων γύρων επικοινωνίας είναι μικρότερος από τον αριθμό των επιπλέον μηνυμάτων που αποστέλλονται στον αλγόριθμο χωρίς τα σφάλματα λόγω της ανεμπόδιστης εγκατάστασης των συνδέσεων. Το ότι στον αλγόριθμο με τα σφάλματα έχουμε 15% των συνδέσεων να μην εγκατασταίνονται, μειώνει κατά πολύ τον αριθμό των μηνυμάτων που ανταλλάσσονται σε κάθε γύρο επικοινωνίας σε σύγκριση με τον αριθμό των μηνυμάτων που θα ανταλλάσσονταν αν δεν υπήρχαν τα σφάλματα. Επομένως στην περίπτωση του αλγορίθμου χωρίς τα σφάλματα ανταλλάσσονται συνολικά περισσότερα μηνύματα και επειδή τα μηνύματα αυτά πρέπει να επεξεργαστούν από τους παραλήπτες τους αυξάνεται ο χρόνος εκτέλεσης του αλγορίθμου.

Μηνύματα

Θεωρητικά ο συνολικός αριθμός μηνυμάτων που ανταλλάσσονται κατά τη διάρκεια εκτέλεσης του αλγορίθμου με σφάλματα συνδέσεων εξακολουθεί να είναι ανάλογος του αριθμού των κόμβων του συστήματος και συγκεκριμένα είναι της τάξεως $O(n \ln \ln n)$, όπου n ο αριθμός των κόμβων.

Όπως και στην περίπτωση του αλγορίθμου χωρίς σφάλματα, έτσι και στην περίπτωση του αλγορίθμου με σφάλματα συνδέσεων αναμένουμε ότι ο αριθμός των μηνυμάτων θα μεγαλώνει καθώς μεγαλώνει ο αριθμός των κόμβων του συστήματος. Αυτό θα είναι λογικό αφού για να ενημερωθούν περισσότεροι κόμβοι θα πρέπει να σταλούν περισσότερα μηνύματα. Από τον πίνακα φαίνεται ότι καθώς μεγαλώνει το πλήθος των κόμβων μεγαλώνει και ο αριθμός των μηνυμάτων και αυτό πιθανόν να οδηγήσει και πάλι στην επιβεβαίωση του άνω φράγματος $O(n \ln \ln n)$ για τα μηνύματα.

Πιο κάτω φαίνεται η γραφική παράσταση του αριθμού των μηνυμάτων συναρτήσει του αριθμού των κόμβων του συστήματος. Η γραφική παράσταση αναπαριστά την περίπτωση του αλγορίθμου MEDIAN-COUNTER με σφάλματα συνδέσεων καθώς και την περίπτωση του ίδιου αλγορίθμου χωρίς σφάλματα:



Σχήμα 6.3 Αλγόριθμος MEDIAN-COUNTER με Σφάλματα Συνδέσεων – Αριθμός Μηνυμάτων

Όντως, η γραφική παράσταση του αριθμού των μηνυμάτων στην περίπτωση όπου υπάρχουν σφάλματα συνδέσεων εξακολουθεί να είναι σχεδόν γραμμική σε σχέση με τον αριθμό των κόμβων. Παρόλο όμως που τα μηνύματα αυξάνονται σχεδόν γραμμικά και στις δύο περιπτώσεις, παρατηρούμε ότι στην περίπτωση των σφαλμάτων ο αριθμός των μηνυμάτων είναι λίγο μικρότερος από αυτόν στην περίπτωση όπου δεν υπάρχουν σφάλματα. Αφού ο ίδιος αριθμός κόμβων ενημερώνονται με λιγότερα μηνύματα, στην περίπτωση των σφαλμάτων, συμπαιρένουμε ότι τα μηνύματα που μεταδίδονται στον αλγόριθμο MEDIAN-COUNTER χωρίς σφάλματα είναι περισσότερα από όσα χρειάζονται για την ενημέρωση των κόμβων. Ο λόγος για τον οποίο μεταδίδονται περισσότερα μηνύματα από όσα χρειάζονται είναι για να καθιστούν τον αλγόριθμο ανεκτικό σε τέτοιου είδους σφάλματα. Η μείωση του αριθμού των μηνυμάτων που μεταδίδονται στην περίπτωση των σφαλμάτων επικοινωνίας οφείλεται στο γεγονός ότι 15% των συνδέσεων που κανονικά θα εγκαθίστονταν τελικά δεν εγκαθίστανται επομένως μεταδίδονται λιγότερα μηνύματα.

Παρόλα αυτά, και στις δύο παραλλαγές του αλγορίθμου, με σφάλματα συνδέσεων και χωρίς σφάλματα, ο αριθμός των μηνυμάτων εξακολουθεί να είναι ανάλογος του αριθμού των κόμβων στην προσομοίωση. Αυτό συναύει με το θεωρητικό άνω φράγμα $O(n \ln \ln n)$ για τον αριθμό των μηνυμάτων που αποστέλλονται στον αλγόριθμο στην περίπτωση των σφαλμάτων συνδέσεων.

6.2.4 Συμπεράσματα

Αφού μελετήσαμε τον αλγόριθμο MEDIAN-COUNTER και στην περίπτωση όπου υπάρχουν σφάλματα στην εγκατάσταση των συνδέσεων μεταξύ των κόμβων του συστήματος καταλήξαμε στο συμπέρασμα ότι όντως η θεωρητική ανάλυση του αλγορίθμου ως προς τα σφάλματα συνδέσεων συναύει με την πρακτική του αξιολόγηση. Αυτό γιατί θεωρητικά ο αλγόριθμος ενημερώνει όλους εκτός από το 15% των κόμβων του συστήματος σε $O(\ln n)$ γύρους επικοινωνίας και χρησιμοποιώντας $O(n \ln \ln n)$ μηνύματα. Στην πράξη παρατηρήσαμε ότι όντως ο αλγόριθμος τερματίζει σε $O(\ln n)$ γύρους επικοινωνίας και χρησιμοποιώντας $O(n \ln \ln n)$ μηνύματα. Αξίζει όμως να σημειωθεί ότι στην προσομοίωση της συγκεκριμένης εφαρμογής ενημερώνονταν όλοι οι κόμβοι της εφαρμογής με όλες τις πληροφορίες και όχι όλοι οι κόμβοι εκτός από το 15% όπως αναμέναμε. Επειδή όμως το μην ενημερωθούν το 15% των κόμβων είναι ένα θεωρητικό άνω φράγμα, τα αποτελέσματα που πάρθηκαν εξακολουθούν να συναύουν με τα θεωρητικά.

6.3 Αλγόριθμος Median-Counter Με Σφάλματα Κατάρρευσης Κόμβων

Η δεύτερη περίπτωση σφαλμάτων που θα μελετηθεί είναι η κατάρρευση κόμβων. Οι κόμβοι ενός δικτύου που υπόκειται σε τέτοιου είδους σφάλματα έχουν κάποια πιθανότητα να καταρρεύσουν κατά την εκτέλεση της εφαρμογής και να μην επανέλθουν ξανά σε κανονική λειτουργία.

6.3.1 Περιγραφή Αλγορίθμου

Ο αλγόριθμος λειτουργεί με τον ίδιο τρόπο όπως και ο αλγόριθμος MEDIAN-COUNTER χωρίς σφάλματα με τη μόνη διαφορά ότι σε κάθε γύρο επικοινωνίας υπάρχει μία πιθανότητα f να καταρρεύσουν κάποιοι κόμβοι του δικτύου. Όταν ένας κόμβος καταρρέει δεν υπάρχει περίπτωση να επανέλθει κανονικά σε λειτουργία σε κάποια στιγμή στο μέλλον. Επίσης, επειδή οι κόμβοι έχουν μόνο τοπική γνώση δεν μπορούν να γνωρίζουν αν κάποιος κόμβος στο σύστημα έχει καταρρεύσει, οπότε συνεχίζουν να λειτουργούν σαν να μην συνέβη ποτέ το σφάλμα. Οι κόμβοι στην εφαρμογή καταρρέουν με βάση κάποια κατανομή πιθανοτήτων η οποία επιστρέφει αποτέλεσμα 1 ή 0 καθορίζοντας με αυτό τον τρόπο αν θα καταρρεύσει ή όχι ένας κόμβος.

6.3.2 Υλοποίηση Αλγορίθμου

Η υλοποίηση του αλγορίθμου είναι όμοια με αυτήν του αλγορίθμου MEDIAN-COUNTER χωρίς σφάλματα. Η μόνη διαφορά των δύο υλοποιήσεων βρίσκεται στο τέλος του ενεργού μέρους ενός γύρου επικοινωνίας όπου με πιθανότητα 15-16% ο κάθε κόμβος μπορεί να καταρρεύσει και στο παθητικό μέρος όπου πρέπει να καθοριστεί ότι οι κόμβοι που έχουν καταρρεύσει δεν εκτελούν καμία λειτουργία όταν κάποιος γείτονάς τους επιλέξει να τους στείλει κάποιο μήνυμα. Επιλέχτηκε η πιθανότητα 15-16% παρόλο που είναι αρκετά μεγαλύτερη από τη μέση πιθανότητα σφαλμάτων όπως αναφέρεται στο [7] για να ελεγχθεί αν ο αλγόριθμος παραμένει εύρωστος ακόμη και αν ο αριθμός των σφαλμάτων μεγαλώσει τόσο πολύ.

Ενεργό μέρος

Στο ενεργό μέρος της εφαρμογής υπάρχει η εξής διαφοροποίηση

```
protected void periodicRun() {

   if (myContent.fail)
     return;


   . . .

   double fail=Math.random();

   if (fail<0.01)
        myContent.fail=true;

}
```

Μόλις ξεκινήσει το ενεργό μέρος του γύρου θα ελεγχθεί κατά πόσον ο κόμβος δεν έχει καταρρεύσει, διαφορετικά θα επιστρέψει από τη συνάρτηση χωρίς να εκτελέσει καμία λειτουργία. Ο έλεγχος γίνεται χρησιμοποιώντας το στοιχείο '`myContent.fail`' του κόμβου το οποίο αρχικοποιείται με την τιμή 'false' και παίρνει την τιμή '**`true`**' όταν ο κόμβος καταρρέει.

Στο τέλος του γύρου θα γίνει η πιθανή κατάρρευση του κόμβου. Η συνάρτηση '`Math.random()`' χρησιμοποιείται για την αυθαίρετη κατανομή πιθανοτήτων. Το αποτέλεσμα της συνάρτησης είναι ένας τυχαίος πραγματικός αριθμός μεταξύ 0 και 1 και φυλάγεται στη μεταβλητή '`fail`'. Η κατάρρευση ενός κόμβου θα συμβεί στην

περίπτωση που η τιμή της μεταβλητής ‘fail’ είναι μικρότερη από 0.01. Με αυτό τον τρόπο ο κάθε κόμβος έχει 1% πιθανότητα να καταρρεύσει σε κάθε γύρο επικοινωνίας. Έτσι, επειδή ο μέσος όρος των γύρων επικοινωνίας που τρέχει ο αλγόριθμος είναι 15-16 γύρους, υπάρχει πιθανότητα 15-16% να καταρρεύσει ο κάθε κόμβος κατά τη διάρκεια της προσομοίωσης. Αν ένας κόμβος καταρρεύσει τότε η τιμή του στοιχείου ‘myContent.fail’ του κόμβου αυτού γίνεται ‘**true**’. Είναι σημαντικό το ότι ο κώδικας αυτός εκτελείται στο τέλος του γύρου επικοινωνίας, αφού δηλαδή ο κόμβος θα έχει ωθήσει (push) τις πληροφορίες του σε κάποιο γείτονά του. Έτσι ακόμη και αν ένας κόμβος καταρρεύσει από τον πρώτο κιόλας γύρο, η γνώση του δε θα χαθεί γιατί θα την έχει ήδη στείλει σε κάποιον άλλο κόμβο στο σύστημα. Αν καταρρεύσει και αυτός ο κόμβος πριν προλάβει να μεταδόσει τη γνώση σε κάποιον άλλο τότε η γνώση θα χαθεί, αλλά η πιθανότητα να συβμεί κάτι τέτοιο είναι πολύ μικρή.

Παθητικό μέρος

Στο παθητικό μέρος της εφαρμογής υπάρχει η εξής διαφοροποίηση

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

  if (receiverContent.fail)
    return true;

  . . .

}
```

Μόλις ξεκινήσει το παθητικό μέρος του γύρου θα ελεγχθεί κατά πόσον ο κόμβος δεν έχει καταρρεύσει, διαφορετικά θα επιστρέψει από τη συνάρτηση χωρίς να εκτελέσει καμία λειτουργία. Ο έλεγχος γίνεται χρησιμοποιώντας το στοιχείο ‘receiverContent.fail’ του κόμβου παραλήπτη το οποίο δηλώνει κατά πόσον ο κόμβος έχει καταρρεύσει.

6.3.3 Πειραματική Αξιολόγηση Αλγορίθμου

Αφού υλοποιήθηκε ο αλγόριθμος συνεχίζουμε με την προσομοίωση και την πειραματική του αξιολόγηση.

Μεθοδολογία Πειραματικής Αξιολόγησης

Η μεθοδολογία της πειραματικής αξιολόγησης είναι ίδια με αυτήν της αξιολόγησης του αλγόριθμου MEDIAN-COUNTER χωρίς σφάλματα με τη διαφορά ότι έπρεπε να αποφασιστεί η πιθανότητα με την οποία θα συμβαίνει η κατάρρευση των κόμβων στην εφαρμογή. Η πιθανότητα αυτή αποφασίστηκε να είναι γύρω στο 15% γιατί τα σφάλματα στα πλείστα μεγάλα δίκτυα δεν ξεπερνούν αυτή την πιθανότητα [7]. Δεν ήταν δυνατό η πιθανότητα να είναι ακριβώς ίση με 15% λόγω του ότι ο αριθμός των γύρων επικοινωνίας που παίρνει για να ολοκληρωθεί μία προσομοίωση δεν είναι εκ των προτέρων γνωστός. Έτσι παρόλο που μπορεί να καθοριστεί ποιό είναι ακριβώς το 15% των κόμβων που θα καταρρεύσουν κατά τη διάρκεια της εφαρμογής, δεν μπορεί να προκαθοριστεί ο γύρος στον οποίο θα καταρρεύσει ο κάθε κόμβος. Αντί αυτού επιλέχτηκε ο κάθε κόμβος να καταρρέει με πιθανότητα 1% σε κάθε γύρο επικοινωνίας και επειδή ο μέσος όρος των γύρων που τρέχει μία προσομοίωση είναι ίσος με 15,85 γύρους η συνολική πιθανότητα κατάρρευσης ενός κόμβου είναι τελικά γύρω στο 15-16%.

Αποτελέσματα Πειραματικής Αξιολόγησης

Θεωρητικά ο αλγόριθμος είναι ανεκτικός στα σφάλματα κατάρρευσης κόμβων και εξακολουθεί να ενημερώνει όλους τους κόμβους, εκτός από αυτούς που έχουν καταρρεύσει, με όλες τις πληροφορίες σε $O(\ln n)$ γύρους επικοινωνίας και χρησιμοποιώντας $O(n \ln \ln n)$ μηνύματα, όπου n ο αριθμός των κόμβων. Τα δύο αυτά άνω φράγματα αποδείχθηκαν αυστηρότυπα στο [1]. Σκοπός της εργασίας μας είναι να συμπεράνουμε κατά πόσον οι θεωρητικές αυτές ιδιότητες του αλγορίθμου ως προς τα σφάλματα κατάρρευσης κόμβων επιβεβαιώνονται στην πράξη.

Ακολουθούν οι τελικές μετρήσεις της προσομοίωσης του αλγορίθμου που πάρθηκαν από το μέσο όρο των πέντε προσομοιώσεων της κάθε μιας από τις πιο κάτω περιπτώσεις. Η στήλη ‘Nodes’ αντιπροσωπεύει τον αριθμό των κόμβων στην προσομοίωση. Η στήλη ‘Rounds’ αντιπροσωπεύει τον αριθμό των γύρων επικοινωνίας που χρειάστηκε να τρέξει ο αλγόριθμος μέχρι να ολοκληρώσει την εκτέλεσή του, δηλαδή είναι ο γύρος στον οποίο στάλθηκαν τα τελευταία μηνύματα στην προσομοίωση. Η στήλη ‘Time (msec)’ αντιπροσωπεύει το χρόνο (σε χιλιοστά του

δευτερολέπτου) που πήρε για να ολοκληρωθεί ο αλγόριθμος, δηλαδή μετριέται από την αρχή της εκτέλεσης της προσομοίωσης μέχρι το τέλος του γύρου στον οποίο ολοκληρώνεται ο αλγόριθμος. Η στήλη 'Messages' αντιπροσωπεύει το συνολικό αριθμό μηνυμάτων που στάλθηκαν κατά τη διάρκεια της προσομοίωσης, δηλαδή είναι ο αριθμός μηνυμάτων που αποστέλλονται από τον πρώτο γύρο προσομοίωσης μέχρι και τον γύρο στον οποίο ολοκληρώνεται ο αλγόριθμος.

Nodes	Rounds	Time (msec)	Messages
20	12,80	275,40	450,00
40	14,20	564,40	933,40
60	14,80	1045,00	1455,40
80	15,40	1706,80	2049,00
100	15,80	2464,60	2612,40
120	16,20	3316,20	3111,40
140	16,20	4513,60	3753,80
160	16,60	5815,40	4277,40
180	16,80	7296,20	4820,80
200	17,00	9488,20	5442,40
220	17,20	11677,00	5928,40
240	17,20	15597,60	6547,20

Πίνακας 6.2 Αλγόριθμος MEDIAN-COUNTER Με Σφάλματα Κατάρρευσης Κόμβων

Ανάλυση Αποτελεσμάτων

Οι μετρήσεις που πάρθηκαν από την προσομοίωση του αλγορίθμου οδήγησαν στις πιο κάτω παρατηρήσεις ως προς τις τρεις παραμέτρους αξιολόγησης του αλγορίθμου, τους γύρους επικοινωνίας, το χρόνο εκτέλεσης και τον αριθμό μηνυμάτων:

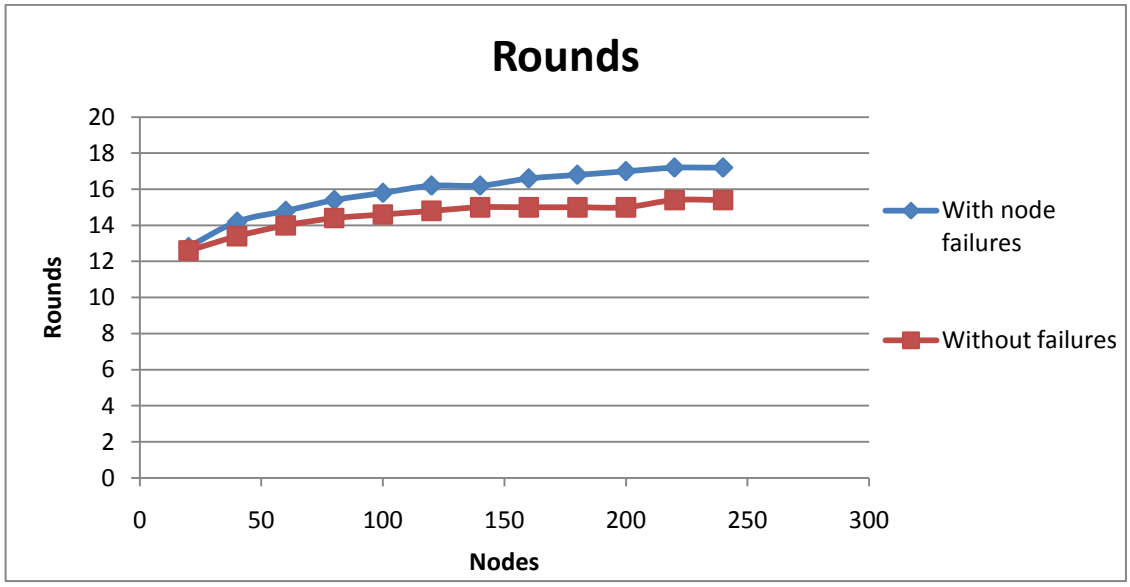
Γύροι Επικοινωνίας

Θεωρητικά ο αριθμός των γύρων επικοινωνίας στην εκτέλεση του αλγορίθμου με σφάλματα κατάρρευσης κόμβων διατηρεί τον λογαριθμικό ρυθμό αύξησης ως προς τον αριθμό των κόμβων του συστήματος. Αναμένουμε όμως ότι ο αριθμός των γύρων επικοινωνίας στην περίπτωση των σφαλμάτων θα είναι μεγαλύτερος από την περίπτωση όπου δεν υπάρχουν σφάλματα. Αυτό θα ήταν λογικό αφού κάποιοι κόμβοι καταρρέουν και επομένως έχουμε λιγότερους κόμβους να αποστέλλουν μηνύματα

στους κόμβους του συστήματος που δεν έχουν καταρρεύσει. Επίσης έχουμε και μηνύματα να αποστέλλονται σε κόμβους που έχουν καταρρεύσει αντί στους υπόλοιπους κόμβους λόγω του ότι οι κόμβοι δεν γνωρίζουν ότι γείτονές τους έχουν καταρρεύσει. Ως αποτέλεσμα τα μηνύματα που φτάνουν στους κόμβους που δεν έχουν καταρρεύσει είναι λιγότερο και έτσι αναμένουμε κάποια καθυστέρηση στην ενημέρωσή των κόμβων του συστήματος.

Όπως παρατηρούμε στον πίνακα 6.2 η αύξηση στον αριθμό των γύρων επικοινωνίας φαίνεται να είναι λογαριθμική αφού όσο μεγαλώνει το πλήθος των κόμβων, μειώνεται ο ρυθμός αύξησης στον αριθμό των γύρων επικοινωνίας. Επίσης παρατηρούμε ότι σε κάθε περίπτωση προσομοίωσης της εφαρμογής χωρίς σφάλματα χρειάζονται λιγότεροι γύροι μέχρι να τερματίσει ο αλγόριθμος σε σύγκριση με την περίπτωση όπου υπάρχουν σφάλματα κατάρρευσης. Οπότε είναι προφανές ότι στην περίπτωση των σφαλμάτων ο αριθμός των γύρων επικοινωνίας είναι μεγαλύτερος.

Πιο κάτω φαίνεται η γραφική παράσταση του αριθμού των γύρων επικοινωνίας συναρτήσει του αριθμού των κόμβων του συστήματος. Η γραφική παράσταση αναπαριστά την περίπτωση του αλγορίθμου MEDIAN-COUNTER με σφάλματα κατάρρευσης κόμβων καθώς και την περίπτωση του ίδιου αλγορίθμου χωρίς σφάλματα:



Σχήμα 6.4 Αλγόριθμος MEDIAN-COUNTER με Σφάλματα Κατάρρευσης Κόμβων – Γύροι Επικοινωνίας

Όντως, η γραφική παράσταση των γύρων επικοινωνίας στην περίπτωση όπου υπάρχουν σφάλματα κατάρρευσης κόμβων εξακολουθεί να είναι λογαριθμική σε σχέση με τον

αριθμό των κόμβων. Επίσης παρατηρούμε ότι, όπως ήταν αναμενόμενο, στην περίπτωση των σφαλμάτων ο αριθμός των γύρων επικοινωνίας είναι μεγαλύτερος από αυτόν στην περίπτωση όπου δεν υπάρχουν σφάλματα. Αρχικά, όταν δηλαδή ο αριθμός των κόμβων της εφαρμογής είναι ίσος με 20, η αύξηση στον αριθμό των γύρων επικοινωνίας στην περίπτωση των σφαλμάτων είναι πολύ μικρή σε σύγκριση με τον αριθμό των γύρων επικοινωνίας του αλγορίθμου χωρίς σφάλματα και είναι μόλις 0,20 γύρους. Συγκεκριμένα, αυξήθηκε από 12,6 σε 12,8 γύρους επικοινωνίας. Καθώς όμως αυξάνεται ο αριθμός των κόμβων, η διαφορά μεταξύ των γύρων επικοινωνίας στις δύο παραλλαγές του αλγορίθμου αυξάνεται. Όπως όμως παρατηρούμε από τους Πίνακες 5.1 και 6.2, η διαφορά αυξάνεται με πολύ μικρούς ρυθμούς και σε καμία περίπτωση δεν ξεπερνά το 15%. Λόγω όμως του ότι η διαφορά αυξάνεται καθώς αυξάνεται ο αριθμός των κόμβων, πιθανόν στο μέλλον η διαφορά μεταξύ των γύρων προσομοίωσης στην περίπτωση του αλγορίθμου με τα σφάλματα σε σχέση με την περίπτωση όπου δεν υπάρχουν σφάλματα να φτάσει ή και να ξεπεράσει το 15%.

Παρόλο που τα σφάλματα κατάρρευσης κόμβων συμβαίνουν με πιθανότητα 15-16%, η αύξηση στους γύρους επικοινωνίας δεν είναι ίση με 15-16%. Όπως παρατηρήθηκε, η αύξηση στους γύρους επικοινωνίας είναι πολύ μικρότερη από 15% αλλα αυξάνεται καθώς αυξάνεται ο αριθμός των κόμβων. Αυτό πιθανότητα να συμβαίνει γιατί καθώς αυξάνεται ο αριθμός των κόμβων, αυξάνονται και οι γύροι επικοινωνίας στην προσομοίωση και ταυτόχρονα αυξάνεται και ο συνολικός αριθμός των κόμβων που καταρρέουν. Ετσι, αφού αυξάνονται τα σφάλματα κόμβων, αυξάνεται και η καθυστέρηση στην ενημέρωση των κόμβων.

Παρόλα αυτά, και στις δύο παραλλαγές του αλγορίθμου, με σφάλματα κατάρρευσης κόμβων και χωρίς σφάλματα, ο αριθμός των γύρων επικοινωνίας αυξάνεται λογαριθμικά ως προς αριθμό των κόμβων. Αυτό συναύει με το θεωρητικό άνω φράγμα $O(\ln n)$ για τους γύρους επικοινωνίας που χρειάζεται να τρέξει ο αλγόριθμος στην περίπτωση των σφαλμάτων.

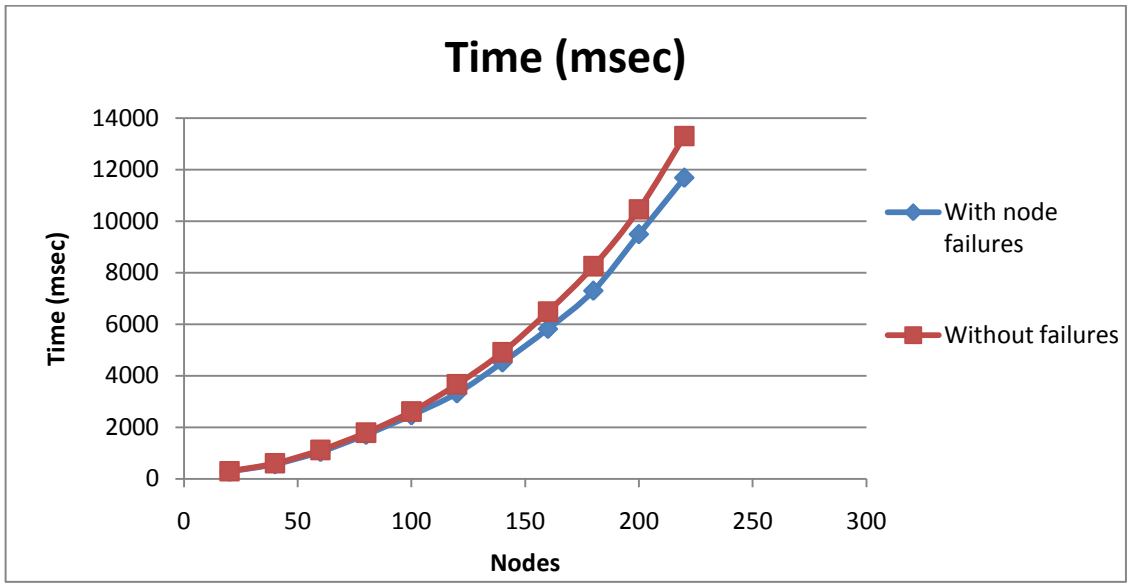
Χρόνος Εκτέλεσης

Ο χρόνος εκτέλεσης του αλγορίθμου αναμέναμε να είναι ανάλογος των γύρων επικοινωνίας. Όπως όμως παρατηρήσαμε και στην προσομοίωση του αλγορίθμου χωρίς σφάλματα, ο χρόνος εκτέλεσης δεν είναι ανάλογος των γύρων επικοινωνίας αλλά

είναι πολύ μεγαλύτερος λόγω του ότι οι κόμβοι σπαταλούν πολύ χρόνο στον κάθε γύρο για να επεξεργαστούν όλες τις πληροφορίες παραλαμβάνουν από τους γείτονές τους. Κάτι παρόμοιο αναμένουμε να συμβεί και σε αυτή την περίπτωση προσομοίωσης όπου υπάρχουν κόμβοι που καταρρέουν αφού οι κόμβοι που δεν έχουν καταρρεύσει εξακολουθούν να επεξεργάζονται όλες τις πληροφορίες που παραλαμβάνουν.

Όπως παρατηρούμε στον πίνακα 6.2, ο χρόνος εκτέλεσης μεγαλώνει όσο μεγαλώνει ο αριθμός των κόμβων. Ο ρυθμός όμως με τον οποίο αυξάνεται ο χρόνος εκτέλεσης του αλγορίθμου δεν είναι σταθερός, αλλά μεγαλώνει όσο αυξάνονται οι κόμβοι στο σύστημα. Επομένως, ο χρόνος εκτέλεσης του αλγορίθμου δεν είναι λογαριθμικός ως προς τον αριθμό των κόμβων αλλά φαίνεται να είναι πολύ μεγαλύτερης τάξεως. Ακόμη παρατηρούμε ότι σε κάθε περίπτωση προσομοίωσης, ο αλγόριθμος χωρίς σφάλματα έχει μεγαλύτερο χρόνο εκτέλεσης από τον αλγόριθμο με σφάλματα.

Πιο κάτω φαίνεται η γραφική παράσταση του χρόνου εκτέλεσης του αλγορίθμου συναρτήσει του αριθμού των κόμβων του συστήματος. Η γραφική παράσταση αναπαριστά την περίπτωση του αλγορίθμου MEDIAN-COUNTER με σφάλματα κατάρρευσης κόμβων καθώς και την περίπτωση του ίδιου αλγορίθμου χωρίς σφάλματα:



Σχήμα 6.5 Αλγόριθμος MEDIAN-COUNTER με Σφάλματα Κατάρρευσης Κόμβων –Χρόνος Εκτέλεσης

Όντως, ο ρυθμός με τον οποίο αυξάνεται ο χρόνος εκτέλεσης του αλγορίθμου με τα σφάλματα κατάρρευσης κόμβων είναι της ίδιας τάξης με το ρυθμό με τον οποίο

αυξάνεται ο χρόνος εκτέλεσης στον αλγόριθμο χωρίς τα σφάλματα. Αυτό ήταν αναμενόμενο αφού οι λειτουργίες που εκτελούν οι δύο αλγόριθμοι σε κάθε γύρο επικοινωνίας είναι περίπου οι ίδιες.

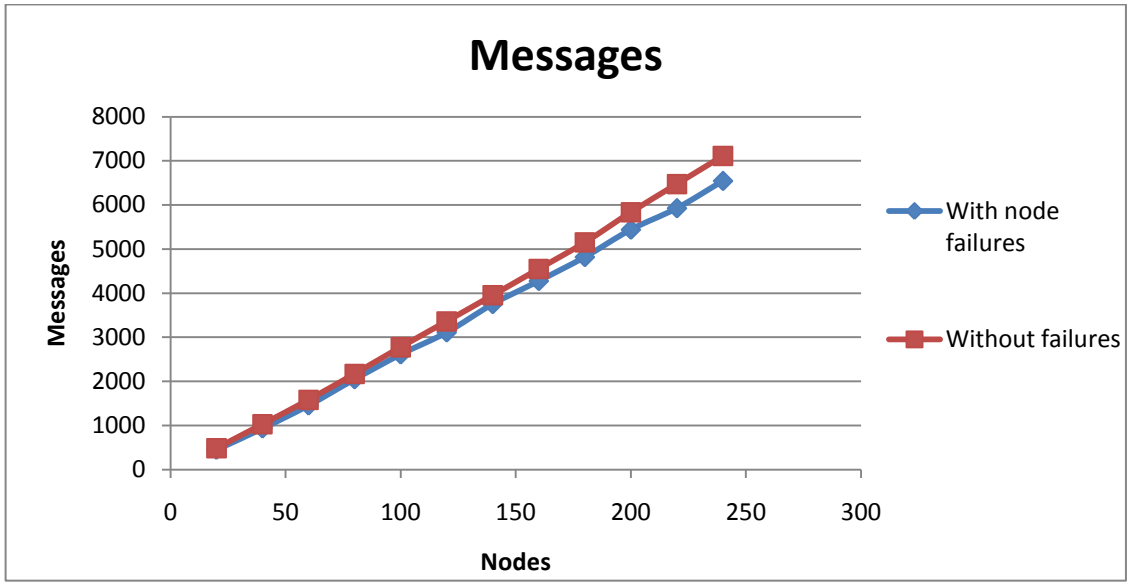
Το γεγονός ότι στην κάθε περίπτωση προσομοίωσης των δύο αλγορίθμων ο αλγόριθμος χωρίς σφάλματα έχει μεγαλύτερο χρόνο εκτέλεσης από ότι ο αλγόριθμος στον οποίο καταρρέουν οι κόμβοι δεν ήταν αναμενόμενο. Αναμέναμε ότι αφού ο αλγόριθμος με τα σφάλματα είχε μεγαλύτερο αριθμό γύρων επικοινωνίας σε κάθε περίπτωση προσομοίωσης θα είχε και μεγαλύτερο χρόνο εκτέλεσης. Αυτό όμως δεν ισχύει και ο λόγος είναι πιθανότατα ο εξής: Κάποιοι από τους κόμβους του συστήματος καταρρέουν και συνεπώς δεν εκτελούν τις λειτουργίες που θα εκτελούσαν υπό κανονικές συνθήκες σε ένα γύρο επικοινωνίας. Αφού δεν εκτελούν τις λειτουργίες αυτές, ο χρόνος εκτέλεσης μειώνεται. Επίσης, επειδή ο αριθμός των κόμβων μειώνεται, μειώνεται και ο αριθμός των μηνυμάτων που μεταδίδονται στο δίκτυο και ως αποτέλεσμα μειώνεται ο χρόνος που χρειάζονται οι παραλήπτες των μηνυμάτων για να τα επεξεργαστούν. Επομένως, για να παίρνει λιγότερο χρόνο εκτέλεσης ο αλγόριθμος με τα σφάλματα κατάρρευσης έπεται ότι ο επιπλέον χρόνος που χρειάζονται οι επιπλέον γύροι επικοινωνίας είναι λιγότερος από τον χρόνο που γλιτώνει η προσομοίωση λόγω της κατάρρευσης των κόμβων και του μειωμένου αριθμού μηνυμάτων που ανταλλάσσονται στο σύστημα.

Μηνύματα

Θεωρητικά ο συνολικός αριθμός μηνυμάτων που ανταλλάσσονται κατά τη διάρκεια εκτέλεσης του αλγορίθμου με σφάλματα κατάρρευσης κόμβων εξακολουθεί να είναι ανάλογος του αριθμού των κόμβων του συστήματος και συγκεκριμένα είναι της τάξεως $O(n \ln \ln n)$, όπου n ο αριθμός των κόμβων.

Όπως και στην περίπτωση του αλγορίθμου χωρίς σφάλματα, έτσι και στην περίπτωση του αλγορίθμου με σφάλματα κατάρρευσης κόμβων αναμένουμε ότι ο αριθμός των μηνυμάτων θα μεγαλώνει καθώς μεγαλώνει ο αριθμός των κόμβων του συστήματος. Αυτό θα είναι λογικό αφού για να ενημερωθούν περισσότεροι κόμβοι θα πρέπει να σταλούν περισσότερα μηνύματα. Από τον πίνακα φαίνεται ότι καθώς μεγαλώνει το πλήθος των κόμβων μεγαλώνει και ο αριθμός των μηνυμάτων και αυτό πιθανόν να οδηγήσει και πάλι στην επιβεβαίωση του άνω φράγματος $O(n \ln \ln n)$ για τα μηνύματα.

Πιο κάτω φαίνεται η γραφική παράσταση του αριθμού των μηνυμάτων συναρτήσει του αριθμού των κόμβων του συστήματος. Η γραφική παράσταση αναπαριστά την περίπτωση του αλγορίθμου MEDIAN-COUNTER με σφάλματα κατάρρευσης κόμβων καθώς και την περίπτωση του ίδιου αλγορίθμου χωρίς σφάλματα:



Σχήμα 6.6 Αλγόριθμος MEDIAN-COUNTER με Σφάλματα Κατάρρευσης Κόμβων – Αριθμός Μηνυμάτων

Όντως, η γραφική παράσταση του αριθμού των μηνυμάτων στην περίπτωση όπου υπάρχουν σφάλματα κατάρρευσης κόμβων εξακολουθεί να είναι σχεδόν γραμμική σε σχέση με τον αριθμό των κόμβων. Παρόλο όμως που τα μηνύματα αυξάνονται σχεδόν γραμμικά και στις δύο περιπτώσεις, παρατηρούμε ότι στην περίπτωση των σφαλμάτων κατάρρευσης ο αριθμός των μηνυμάτων είναι λίγο μικρότερος από αυτόν στην περίπτωση όπου δεν υπάρχουν σφάλματα. Αφού ο ίδιος αριθμός κόμβων ενημερώνονται με λιγότερα μηνύματα, στην περίπτωση των σφαλμάτων, συμπαιρένουμε ότι τα μηνύματα που μεταδίδονται στον αλγόριθμο MEDIAN-COUNTER χωρίς σφάλματα είναι περισσότερα από όσα χρειάζονται για την ενημέρωση των κόμβων. Ο λόγος για τον οποίο μεταδίδονται περισσότερα μηνύματα από όσα χρειάζονται είναι για να καθιστούν τον αλγόριθμο ανεκτικό σε τέτοιου είδους σφάλματα. Η μείωση του αριθμού των μηνυμάτων που μεταδίδονται στην περίπτωση των σφαλμάτων κατάρρευσης κόμβων οφείλεται στα πιο κάτω: Από τη μία αφού καταρρέουν κόμβοι στο σύστημα και σταματούν να στέλλουν μηνύματα είναι λιγότερα

τα μηνύματα που αποστέλλονται στους κόμβους του συστήματος που δεν έχουν καταρρεύσει. Από την άλλη πλευρά οι κόμβοι που χρειάζεται να ενημερωθούν μειώθηκαν, αφού κάποιοι από αυτούς έχουν καταρρεύσει, και έτσι χρειάζεται λιγότερος αριθμός μηνυμάτων μέχρι να ενημερωθούν όλοι οι κόμβοι. Έτσι παρόλο που ο αριθμός των γύρων επικοινωνίας αυξάνεται στην περίπτωση των σφαλμάτων, η κατάρρευση των κόμβων οδηγεί σε μείωση του συνολικού αριθμού μηνυμάτων που χρειάζεται να μεταδοθούν μεταξύ των κόμβων του συστήματος.

Παρόλα αυτά, και στις δύο παραλλαγές του αλγορίθμου, με σφάλματα κατάρρευσης κόμβων και χωρίς σφάλματα, ο αριθμός των μηνυμάτων εξακολουθεί να είναι ανάλογος του αριθμού των κόμβων στην προσομοίωση. Αυτό συναύει με το θεωρητικό άνω φράγμα $O(n \ln \ln n)$ για τον αριθμό των μηνυμάτων που αποστέλλονται στον αλγόριθμο στην περίπτωση των σφαλμάτων κατάρρευσης κόμβων.

6.3.4 Συμπεράσματα

Αφού μελετήσαμε τον αλγόριθμο MEDIAN-COUNTER και στην περίπτωση όπου υπάρχουν σφάλματα κατάρρευσης κόμβων, καταλήξαμε στο συμπέρασμα ότι όντως η θεωρητική ανάλυση του αλγορίθμου ως προς τα σφάλματα κατάρρευσης κόμβων συναύει με την πρακτική του αξιολόγηση. Αυτό γιατί θεωρητικά ο αλγόριθμος ενημερώνει όλους εκτός από το 15% των κόμβων του συστήματος σε $O(\ln n)$ γύρους επικοινωνίας και χρησιμοποιώντας $O(n \ln \ln n)$ μηνύματα. Στην πράξη παρατηρήσαμε ότι όντως ο αλγόριθμος τερματίζει σε $O(\ln n)$ γύρους επικοινωνίας και χρησιμοποιώντας $O(n \ln \ln n)$ μηνύματα και καταφέρνει να ενημερώσει όλους τους κόμβους του συστήματος οι οποίοι δεν έχουν καταρρεύσει. Κατά την προσομοίωση του αλγορίθμου παρατήρησα επίσης ότι όντως το ποσοστό των κόμβων που καταρρέουν σε κάθε προσομοίωση είναι γύρω στο 15% και με το τέλος της προσομοίωσης το υπόλοιπο 85% των κόμβων έχει μάθει όλες τις πληροφορίες.

6.4 Σύγκριση Πειραματικής Αξιολόγησης του αλγορίθμου MEDIAN-COUNTER με Σφαλμάτα Συνδέσεων και Σφαλμάτα Κατάρρευσης Κόμβων

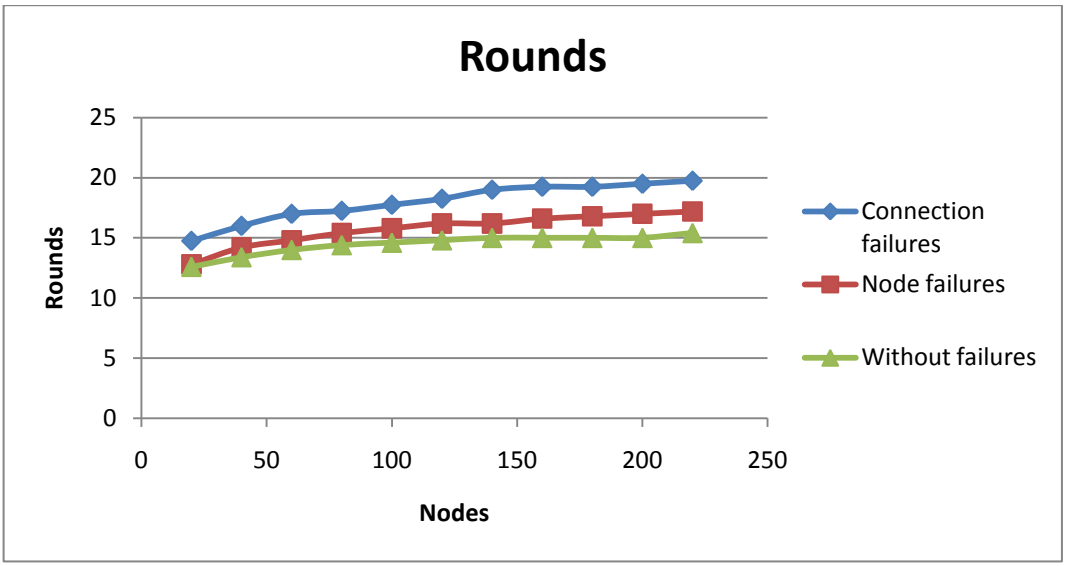
Σε αυτό το σημείο θα συγκρίνουμε τις δύο περιπτώσεις σφαλμάτων του αλγορίθμου MEDIAN-COUNTER. Όπως παρατηρήσαμε στα υποκεφάλαια **6.2** και **6.3** η απόδοση του

αλγορίθμου MEDIAN-COUNTER επηρεάζεται από τυχόν σφάλματα στο δίκτυο αλλά παραμένει ικανοποιητική. Αξίζει όμως να δούμε κατά πόσον η απόδοση του αλγορίθμου επηρεάζεται όχι μόνο από την εμφάνιση σφαλμάτων αλλά και από το είδος των σφαλμάτων που συμβαίνουν.

Θα συγκρίνουμε την απόδοση του αλγορίθμου MEDIAN-COUNTER στις δύο περιπτώσεις σφαλμάτων ως προς τις τρεις παραμέτρους αξιολόγησης του αλγορίθμου, τους γύρους επικοινωνίας, το χρόνο εκτέλεσης και τον αριθμό μηνυμάτων:

Γύροι Επικοινωνίας

Πιο κάτω φαίνεται η γραφική παράσταση του αριθμού των γύρων επικοινωνίας συναρτήσει του αριθμού των κόμβων του συστήματος. Η γραφική παράσταση αναπαριστά τον αλγορίθμο MEDIAN-COUNTER στην περίπτωση σφαλμάτων συνδέσεων, στην περίπτωση σφαλμάτων κατάρρευσης κόμβων καθώς και στην περίπτωση στην οποία δεν υπάρχουν σφάλματα:



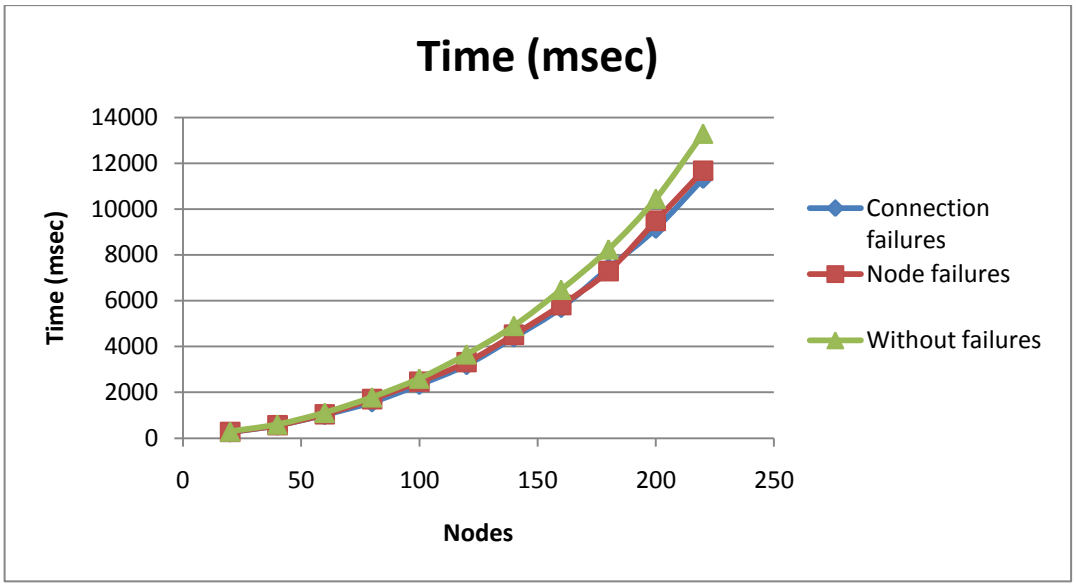
Σχήμα 6.7 Αλγόριθμος MEDIAN-COUNTER με Σφάλματα Συνδέσεων και Σφάλματα Κατάρρευσης Κόμβων – Γύροι Επικοινωνίας

Όπως παρατηρούμε στο Σχήμα 6.7, η απόδοση του αλγορίθμου MEDIAN-COUNTER ως προς τους γύρους επικοινωνίας διαφέρει ανάλογα με το είδος των σφαλμάτων που συμβαίνουν. Στην περίπτωση των σφαλμάτων συνδέσεων η προσομοίωση του αλγορίθμου χρειάζεται αρκετά μεγαλύτερο αριθμό γύρων επικοινωνίας για να

τερματίσει σε σύγκριση με την περίπτωση των σφαλμάτων κατάρρευσης κόμβων. Η διαφορά στους γύρους επικοινωνίας μεταξύ των δύο περιπτώσεων σφαλμάτων, όπως φαίνεται στη γραφική παράσταση, αυξάνεται με μικρούς ρυθμούς καθώς αυξάνεται το πλήθος των κόμβων της εφαρμογής. Συγκεκριμένα, από τους Πίνακες 6.1 και 6.2 παρατηρούμε ότι όταν ο αριθμός των κόμβων στην εφαρμογή είναι ίσος με 20 η διαφορά είναι ίση με 1,95 γύρους επικοινωνίας ενώ όταν ο αριθμός των κόμβων αυξηθεί σε 240, η διαφορά αυξάνεται στους 2,55 γύρους. Το γεγονός ότι χρειάζονται λιγότεροι γύροι επικοινωνίας στην περίπτωση των σφαλμάτων κατάρρευσης κόμβων οφείλεται στο ότι λόγω των σφαλμάτων μειώνονται τα μηνύματα που μεταδίδονται, όπως συμβαίνει και με τα σφάλματα συνδέσεων, αλλά μειώνεται και ο αριθμός των κόμβων που χρειάζεται να ενημερωθούν αφού 15% από αυτούς έχουν καταρρεύσει. Στην περίπτωση όμως των σφαλμάτων συνδέσεων, μειώνεται ο αριθμός των μηνυμάτων που αποστέλλονται αλλά ο αριθμός των κόμβων που πρέπει να ενημερωθούν παραμένει ο ίδιος. Επομένως, χρειάζονται περισσότεροι γύροι επικοινωνίας μέχρι να ενημερωθούν όλοι οι κόμβοι.

Χρόνος Εκτέλεσης

Πιο κάτω φαίνεται η γραφική παράσταση του χρόνου εκτέλεσης συναρτήσει του αριθμού των κόμβων του συστήματος. Η γραφική παράσταση αναπαριστά τον αλγορίθμο MEDIAN-COUNTER στην περίπτωση σφαλμάτων συνδέσεων, στην περίπτωση σφαλμάτων κατάρρευσης κόμβων καθώς και στην περίπτωση στην οποία δεν υπάρχουν σφάλματα:

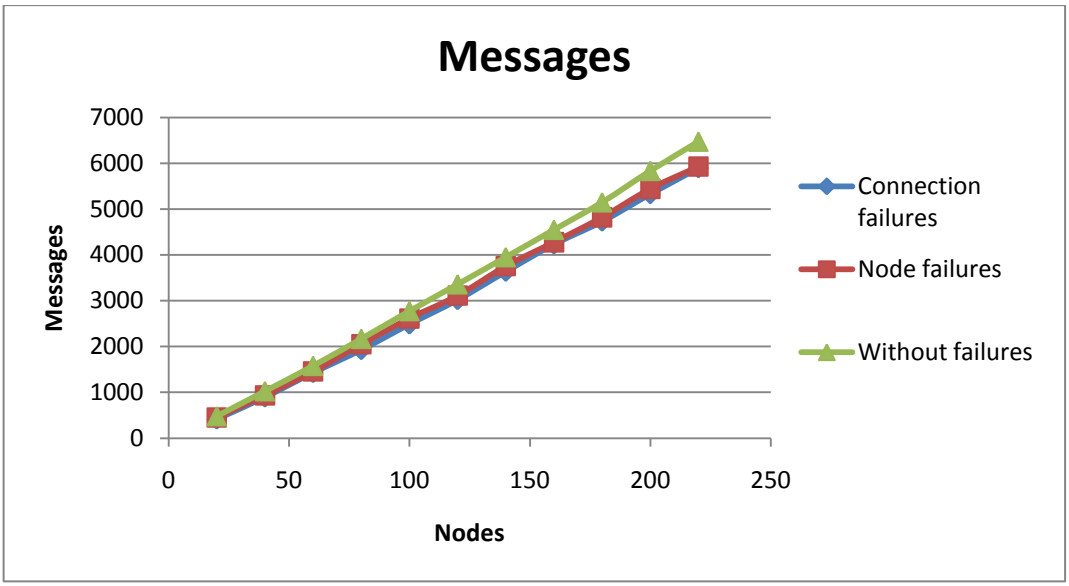


Σχήμα 6.8 Αλγόριθμος MEDIAN-COUNTER με Σφάλματα Συνδέσεων και Σφάλματα Κατάρρευσης Κόμβων – Χρόνος Εκτέλεσης

Όπως φαίνεται στο Σχήμα 6.8 ο χρόνος που χρειάζεται για να τερματίσει η προσομοίωση του αλγορίθμου και στις δύο περιπτώσεις σφαλμάτων, σφάλματα συνδέσεων και σφάλματα κατάρρευσης κόμβων, είναι ο ίδιος. Παρόλο που ο αριθμός των γύρων επικοινωνίας στην περίπτωση των σφαλμάτων συνδέσεων είναι μεγαλύτερος, παρατηρούμε ότι δεν χρειάζεται μεγαλύτερος χρόνος εκτέλεσης. Αυτό οφείλεται στο ότι και στις δύο περιπτώσεις τα σφάλματα συμβαίνουν με πιθανότητα περίπου 15% και έτσι μειώνεται με παρόμοιο τρόπο ο αριθμός μηνυμάτων που μεταδίδονται. Συνεπώς μειώνεται με παρόμοιο τρόπο και ο χρόνος που χρειάζεται για να επεξεργαστούν οι κόμβοι παραλήπτες τα μηνύματα αυτά. Επομένως, η απόδοση του αλγορίθμου MEDIAN-COUNTER ως προς τον χρόνο εκτέλεσης διαφέρει αναλόγως με το αν θα συμβούν σφάλματα ή όχι, αλλά δεν εξαρτάται από το είδος τως σφαλμάτων που συμβαίνουν.

Μηνύματα

Πιο κάτω φαίνεται η γραφική παράσταση του αριθμού των μηνυμάτων συναρτήσει του αριθμού των κόμβων του συστήματος. Η γραφική παράσταση αναπαριστά τον αλγορίθμο MEDIAN-COUNTER στην περίπτωση σφαλμάτων συνδέσεων, στην περίπτωση σφαλμάτων κατάρρευσης κόμβων καθώς και στην περίπτωση στην οποία δεν υπάρχουν σφάλματα:



Σχήμα 6.9 Αλγόριθμος MEDIAN-COUNTER με Σφάλματα Συνδέσεων και Σφάλματα Κατάρρευσης Κόμβων – Αριθμός Μηνυμάτων

Όπως φαίνεται στο Σχήμα 6.9 ο αριθμός των μηνυμάτων που μεταδίδονται κατά τη διάρκεια της προσομοίωσης του αλγορίθμου και στις δύο περιπτώσεις σφαλμάτων, σφάλματα συνδέσεων και σφάλματα κατάρρευσης κόμβων, είναι ο ίδιος. Αυτό οφείλεται στο ότι και στις δύο περιπτώσεις τα σφάλματα συμβαίνουν με πιθανότητα περίπου 15%, συνεπώς ο αριθμός των μηνυμάτων που μεταδίδονται μειώνεται και στα δύο είδη σφαλμάτων με παρόμοιους ρυθμούς. Επομένως, η απόδοση του αλγορίθμου MEDIAN-COUNTER ως προς τον αριθμό μηνυμάτων που μεταδίδονται διαφέρει αναλόγως με το αν θα συμβούν σφάλματα ή όχι, αλλά δεν εξαρτάται από το είδος τως σφαλμάτων που συμβαίνουν.

6.5 Συμπεράσματα

Στο Κεφάλαιο 5 αξιολογήσαμε τον αλγόριθμο MEDIAN-COUNTER σε περιβάλλον το οποίο δεν υπόκειται σε σφάλματα και καταλήξαμε στο συμπέρασμα ότι ο αλγόριθμος είναι αποδοτικός αφού ενημερώνει όλους του κομβους του συστήματος σε $O(\ln n)$ γύρους επικοινωνίας και χρησιμοποιώντας $O(n \ln \ln n)$ μηνύματα. Στο παρόν κεφάλαιο, αξιολογήσαμε τον αλγόριθμο MEDIAN-COUNTER σε περιβάλλον το οποίο υπόκειται σε σφάλματα, σφάλματα συνδέσεων και σφάλματα κατάρρευσης κομβων, και παρατηρήσαμε ότι ο αλγόριθμος συνεχίζει να ενημερώνει τους κόμβους σε $O(\ln n)$ γύρους επικοινωνίας και χρησιμοποιώντας $O(n \ln \ln n)$ μηνύματα. Έτσι, συμπεραίνουμε

ότι ο αλγόριθμος MEDIAN-COUNTER εκτός από αποδοτικός είναι και ανεκτικός σε σφάλματα.

Παρατηρήσαμε ότι στην περίπτωση που συνέβαιναν σφάλματα κατά τη διάρκεια της προσομοίωσης υπήρχε αύξηση στον αριθμό των γύρων επικοινωνίας που χρειαζόταν η προσομοίωση για να ολοκληρωθεί. Αυτή όμως η αύξηση δεν οδήγησε σε αύξηση στο χρόνο εκτέλεσης του αλγορίθμου ούτε σε αύξηση στον αριθμό των μηνυμάτων που μεταδίδονταν. Αντιθέτως, οι δύο αυτές παραμέτροι μειώθηκαν όταν είχαμε σφάλματα χωρίς να εμποδίζουν τους κόμβους του δικτύου από το να ενημερωθούν. Οπόταν, συμπεράναμε ότι ο αλγόριθμος MEDIAN-COUNTER για να καταφέρνει να είναι εύρωστος μεταδίδει περισσότερα μηνύματα από όσα είναι απαραίτητα για να ενημερωθούν οι κόμβοι και ως αποτέλεσμα σπαταλά περισσότερο χρόνο εκτέλεσης από όσο χρειάζεται πραγματικά.

Κλείνοντας, καταλήγουμε στο συμπέρασμα ότι ο αλγόριθμος MEDIAN-COUNTER είναι εύρωστος και η πειραματική μας αξιολόγηση στην περίπτωση των σφαλμάτων συναύει με τη θεωρητική αφού στην περίπτωση που συνέβαιναν σφάλματα με πιθανότητα f εξακολουθούσαν να ενημερώνονταν όλοι οι κόμβοι του δικτύου εκτός από $O(f)$ κόμβους σε $O(\ln n)$ γύρους επικοινωνίας και χρησιμοποιώντας $O(n \ln \ln n)$ μηνύματα.

ΚΕΦΑΛΑΙΟ 7

Σύγκριση Αλγορίθμων

7.1 Σύγκριση Πειραματικής Ανάλυσης Αλγορίθμων

Αφού μελετήθηκαν και αξιολογήθηκαν ξεχωριστά οι τρεις αλγόριθμοι PUSH, PUSH&PULL και MEDIAN-COUNTER μένει να γίνει η μεταξύ τους σύγκριση για να αποφασιστεί τελικά αν κάποιος από αυτούς υπερτερεί έναντι των υπολοίπων. Ο αλγόριθμος MEDIAN-COUNTER θα αξιολογηθεί για την εκδοχή του στην οποία δεν υπάρχουν σφάλματα αφού όπως είπαμε και στις υπόλοιπες εκδοχές τα αποτελέσματα είναι παρόμοια. Η σύγκριση θα γίνει ξεχωριστά για την κάθε παράμετρο των τριών αλγορίθμων, γύροι επικοινωνίας, χρόνος εκτέλεσης, αριθμός μηνυμάτων και μέγεθος μηνυμάτων και στο τέλος θα οδηγήσει στα γενικά συμπεράσματα.

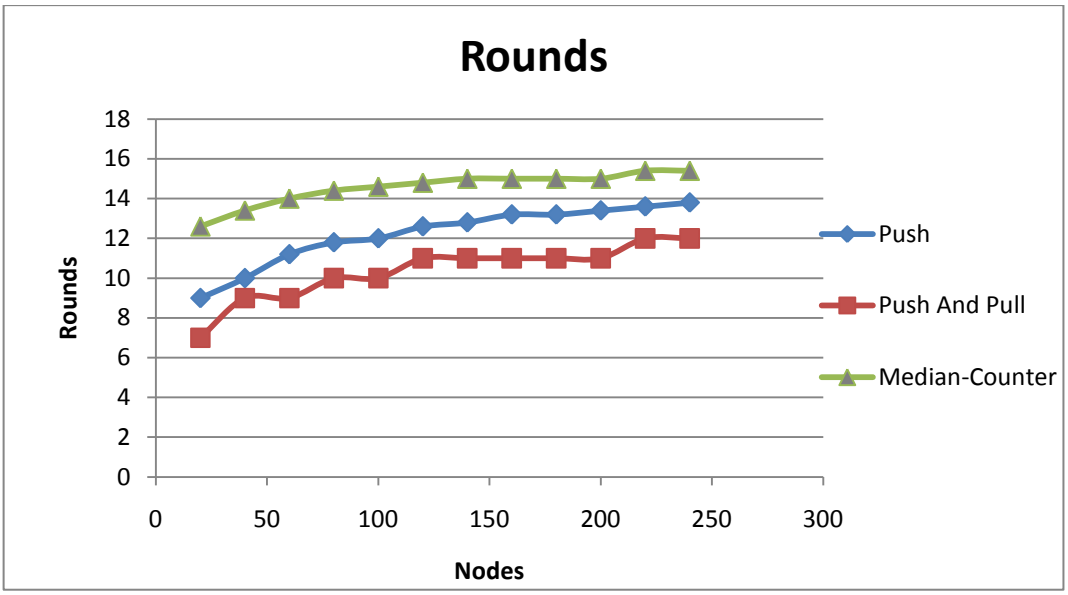
7.1.1 Γύροι Επικοινωνίας

Για να μπορέσουμε να βγάλουμε κάποια συμπεράσματα από τη σύγκριση των τριών αλγορίθμων ως προς τους γύρους επικοινωνίας χρειάζεται να υπενθυμίσουμε τη χρονική στιγμή στην οποία ο κάθε ένας από τους τρεις αλγορίθμους τερματίζει. Στον αλγόριθμο PUSH ένας κόμβος σταματά να στέλλει πληροφορίες όταν ο ίδιος έχει μάθει όλες τις πληροφορίες του δικτύου. Συνεπώς ο αλγόριθμος τερματίζει όταν όλοι οι κόμβοι έχουν ενημερωθεί με όλες τις πληροφορίες του δικτύου. Ο αλγόριθμος PUSH&PULL είναι πιο αυστηρός ως προς τη στιγμή που τερματίζει αφού για συγκεκριμένο αριθμό κόμβων καθορίζει ακριβώς ποιά είναι η προθεσμία μετάδοσης

μίας πληροφορίας. Κάθε πληροφορία έχει ένα μετρητή τον οποίο αυξάνει σε κάθε γύρο επικοινωνίας μέχρι να γίνει ίσος με την προθεσμία μετάδοσης. Σε αυτό το σημείο η πληροφορία θα σταματήσει να μεταδίδεται. Επομένως σε τόσους γύρους επικοινωνίας όση είναι η προθεσμία μετάδοσης της πληροφορίας όλες οι πληροφορίες σταματούν να μεταδίδονται και ο αλγόριθμος τερματίζει. Ο αλγόριθμος MEDIAN-COUNTER κρίνει την κάθε πληροφορία από το χρονικό διάστημα που η πληροφορία βρίσκεται σε συγκεκριμένο κόμβο. Όταν ένας κόμβος γνωρίζει την πληροφορία για μεγάλο χρονικό διάστημα σταματά να τη μεταδίδει. Για το σκοπό αυτό και πάλι χρειάζεται ένας μετρητής για κάθε πληροφορία ο οποίος θα αυξάνεται μέχρι να φτάσει στο μέγιστο όριο. Ο μετρητής όμως αυτός ξεκινά να αυξάνεται από τη στιγμή που ένας κόμβος παραλάβει την πληροφορία και δεν αυξάνεται σε κάθε γύρο επικοινωνίας, αλλά χρησιμοποιείται ένας πιο ευέλικτος κανόνας με βάση τον οποίο καθορίζεται η στιγμή που θα αυξηθεί ο μετρητής. Ετσι δεν μπορεί να αποφασιστεί ακριβώς η στιγμή που θα τερματίσει ο αλγόριθμος MEDIAN-COUNTER.

Επομένως, αναμένουμε ότι ο αλγόριθμος PUSH&PULL ο οποίος καθορίζει αυστηρά τη στιγμή που θα τερματίσει, ανάλογα με το πλήθος των κόμβων, θα χρειάζεται τους λιγότερους γύρους επικοινωνίας. Από την άλλη ο αλγόριθμος MEDIAN-COUNTER ο οποίος περιμένει την κάθε πληροφορία να υπάρξει για συγκεκριμένο χρονικό διάστημα στον κάθε κόμβο είναι ο πιο ευέλικτος και αναμένουμε ότι θα χρειάζεται περισσότερους γύρους επικοινωνίας για να τερματίσει. Τέλος ο αλγόριθμος PUSH ο οποίος δεν είναι αυστηρός ως προς το πότε θα τερματίσει και περιμένει τη στιγμή που όλοι οι κόμβοι μάθουν όλες τις πληροφορίες αναμένουμε να βρίσκεται κάπου στο ενδιάμεσο των άλλων δύο αλγορίθμων ως προς τους γύρους επικοινωνίας.

Πιο κάτω φαίνεται η γραφική παράσταση του αριθμού των γύρων επικοινωνίας συναρτήσει του αριθμού των κόμβων του συστήματος για τους τρεις αλγορίθμους:



Σχήμα 7.1 Σύγκριση Αλγορίθμων - Γύροι Επικοινωνίας

Από τη γραφική παράσταση παρατηρούμε ότι όντως τα αποτελέσματα είναι αυτά που αναμέναμε. Ο αλγόριθμος PUSH&PULL χρειάζεται σε κάθε περίπτωση προσομοίωσης τον μικρότερο αριθμό γύρων επικοινωνίας. Ακόμη λόγω του αυστηρού καθορισμού της στιγμής που θα τερματίσει ο αλγόριθμος PUSH&PULL, παρατηρούμε μία καμπύλη η οποία αλλάζει την κλίση της αρκετά απότομα. Σε αντίθεση με αυτό, οι άλλοι δύο αλγόριθμοι έχουν πιο ομαλές αλλαγές στην κλίση της καμπύλης τους λόγω του ότι είναι πιο ευέλικτοι ως προς τη στιγμή που θα τερματίσουν. Ο αλγόριθμος MEDIAN-COUNTER όπως αναμέναμε χρειάζεται τους περισσότερους γύρους επικοινωνίας μέχρι να τερματίσει αφού περιμένει την κάθε πληροφορία όχι μόνο να μαθευτεί από τον κόμβο αλλά και να ζήσει κάποιο χρονικό διάστημα σε αυτόν. Τέλος ο αλγόριθμος PUSH που τερματίζει μόλις οι κόμβοι ενημερωθούν πλήρως βρίσκεται κάπου στο ενδιάμεσο των άλλων δύο. Παρά τις διαφορές τους, και οι τρεις αλγόριθμοι έχουν εντέλει τον ίδιο ρυθμό αύξησης στους γύρους επικοινωνίας ως προς το πλήθος των κόμβων ο οποίος είναι λογαριθμικός. Συνεπώς και οι τρεις αλγόριθμοι επιβεβαιώνουν το θεωρητικό τους άνω φράγμα ως προς τους γύρους επικοινωνίας.

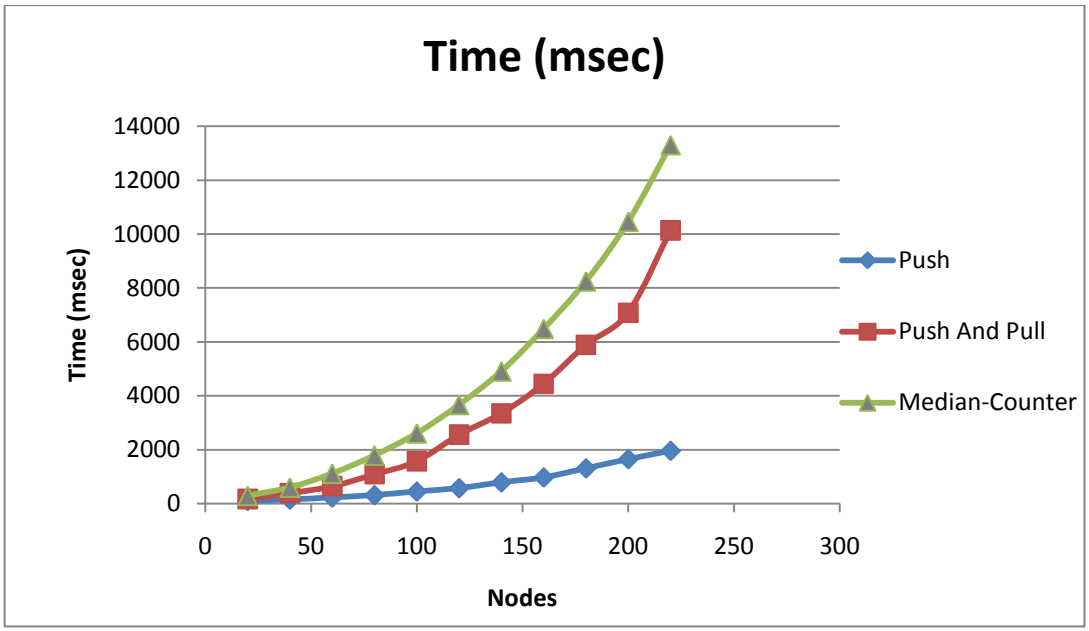
7.1.2 Χρόνος Εκτέλεσης

Ο χρόνος εκτέλεσης των τριών αλγορίθμων, όπως παρατηρήσαμε και κατά την αξιολόγηση του κάθε ενός ξεχωριστά, δεν εξαρτάται μόνο από τον αριθμό των γύρων

επικοινωνίας που χρειάζεται να τρέξει ο κάθε αλγόριθμος. Ο χρόνος εκτέλεσης επηρεάζεται σημαντικά από τις λειτουργίες που έχει να εκτελέσει ο κάθε κόμβος κατά τη διάρκεια ενός γύρου επικοινωνίας ενός συγκεκριμένου αλγόριθμου. Όπως είδαμε, ο αλγόριθμος PUSH είναι ο πιο απλός από τους τρεις αλγόριθμους αφού δε σπαταλά καθόλου χρόνο σε φιλτράρισμα των πληροφοριών ή σε υπολογισμό της χρονικής στιγμής που θα τερματίσει. Αντί αυτού ο κάθε κόμβος στέλλει στο γείτονά του όλες τις πληροφορίες που γνωρίζει μέχρι τον τρέχοντα γύρο και σταματά να στέλλει πληροφορίες όταν ενημερωθεί με όλες τις πληροφορίες που υπάρχουν στο δίκτυο. Επίσης ο χρόνος που χρειάζεται για την επεξεργασία των μηνυμάτων είναι και αυτός λιγότερος λόγω του ότι χρησιμοποιείται μόνο η λειτουργία ώθησης και επομένως είναι λιγότεροι οι κόμβοι που παραλαμβάνουν μηνύματα και που πρέπει να τα επεξεργαστούν. Τα πράγματα όμως δεν είναι τόσο απλά για τους άλλους δύο αλγορίθμους. Εκτός του ότι χρησιμοποιούν και τις δύο λειτουργίες ώθησης και λήψης οι οποίες οδηγούν σε περισσότερα μηνύματα για επεξεργασία, και οι υπόλοιπες λειτουργίες που εκτελούν είναι αρκετά χρονοβόρες. Στον αλγόριθμο PUSH&PULL ο κάθε κόμβος σπαταλά αρκετό χρόνο σε κάθε γύρο επικοινωνίας προσπαθώντας να κρατά και να ενημερώνει τον μετρητή για την προθεσμία μετάδοσης της κάθε πληροφορίας και να φιλτράρει τις πληροφορίες του διαχωρίζοντάς τις σε ‘hot’ και μη. Στον αλγόριθμο MEDIAN-COUNTER οι λειτουργίες που εκτελούνται για την κάθε πληροφορία ενός κόμβου είναι ακόμη πιο χρονοβόρες αφού η κάθε πληροφορία δεν έχει απλά μία ηλικία αλλά μπορεί να μεταβαίνει από μία κατάσταση σε κάποια άλλη ανάλογα με το χρονικό διάστημα που βρίσκεται σε κάποιο κόμβο και με το πλήθος των ίδιων πληροφοριών που θα παραλάβει ο κόμβος αυτός σε κάποιο γύρο.

Επομένως, αναμένουμε ότι ο αλγόριθμος PUSH θα χρειάζεται λιγότερο χρόνο εκτέλεσης από τους άλλους δύο, ενώ από τους αλγόριθμους PUSH&PULL και MEDIAN-COUNTER ο πιο χρονοβόρος αναμένουμε να είναι ο MEDIAN-COUNTER αφού είναι αυτός που εκτελεί τις πιο πολλές και χρονοβόρες διαδικασίες.

Πιο κάτω φαίνεται η γραφική παράσταση του χρόνου εκτέλεσης συναρτήσει του αριθμού των κόμβων του συστήματος για τους τρεις αλγορίθμους:



Σχήμα 7.2 Σύγκριση Αλγορίθμων - Χρόνος Εκτέλεσης

Από τη γραφική παράσταση παρατηρούμε ότι όντως τα αποτελέσματα είναι αυτά που αναμέναμε. Ο αλγόριθμος PUSH χρειάζεται σε κάθε περίπτωση προσομοίωσης τον μικρότερο χρόνο εκτέλεσης και μάλιστα με μεγάλη διαφορά από τους άλλους δύο αλγορίθμους. Ο αλγόριθμος MEDIAN-COUNTER είναι όπως αναμέναμε ο πιο χρονοβόρος ενώ ο αλγόριθμος PUSH&PULL είναι και αυτός αρκετά χρονοβόρος όχι όμως τόσο όσο ο αλγόριθμος MEDIAN-COUNTER.

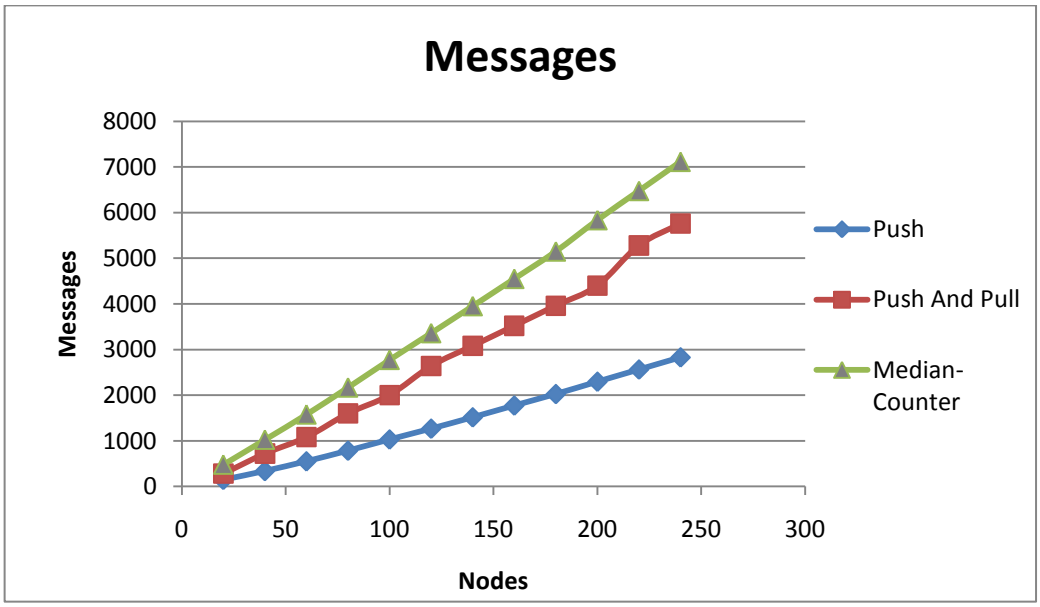
7.1.3 Αριθμός Μηνυμάτων

Ο συνολικός αριθμός των μηνυμάτων που ανταλλάσσονται μεταξύ των κόμβων του συστήματος εξαρτάται από την εκτέλεση των δύο λειτουργιών ώθησης και λήψης μηνυμάτων. Όπως δηλώνει και το όνομά του, ο αλγόριθμος PUSH εκτελεί μόνο τη λειτουργία της ώθησης σε κάθε γύρο επικοινωνίας ενώ οι αλγόριθμοι PUSH&PULL και MEDIAN-COUNTER εκτελούν σε κάθε γύρο και τη λειτουργία της ώθησης και τη λειτουργία της λήψης.

Επομένως, αναμένουμε ότι σε κάθε περίπτωση προσομοίωσης ο αριθμός των μηνυμάτων που ανταλλάσσονται στον αλγόριθμο PUSH θα είναι αρκετά μικρότερος από τον αριθμό των μηνυμάτων που ανταλλάσσονται στους άλλους δύο αλγόριθμους. Ο αριθμός όμως των μηνυμάτων εξαρτάται και από τον αριθμό των γύρων επικοινωνίας που χρειάζεται να τρέξει ο κάθε αλγόριθμος. Επειδή η διαφορά των τριών αλγορίθμων

ως προς τους γύρους επικοινωνίας δεν είναι αρκετά μεγάλη αναμένουμε ότι δεν θα επηρεαστεί πολύ ο αριθμός των μηνυμάτων από τον αριθμό των γύρων επικοινωνίας.

Πιο κάτω φαίνεται η γραφική παράσταση του αριθμού των μηνυμάτων συναρτήσει του αριθμού των κόμβων του συστήματος για τους τρεις αλγορίθμους:



Σχήμα 7.3 Σύγκριση Αλγορίθμων - Αριθμός Μηνυμάτων

Από τη γραφική παράσταση παρατηρούμε ότι όντως τα αποτελέσματα είναι αυτά που αναμέναμε. Ο αριθμός των μηνυμάτων που ανταλλάσσονται κατά την προσομοίωση του αλγόριθμου PUSH είναι πολύ μικρότερος από τον αριθμό των μηνυμάτων που ανταλλάσσονται κατά την προσομοίωση των άλλων δύο αλγορίθμων. Μεταξύ των αλγορίθμων PUSH&PULL και MEDIAN-COUNTER παρατηρούμε ότι μεγαλύτερος αριθμός μηνυμάτων ανταλλάσσονται στον αλγόριθμο MEDIAN-COUNTER και αυτό εξηγείται από τον μεγαλύτερο αριθμό γύρων επικοινωνίας που χρειάζεται ο αλγόριθμος αυτός για να τερματίσει. Παρά τις διαφορές τους, και οι τρεις αλγόριθμοι έχουν εντέλει παρόμοιο ρυθμό αύξησης στον αριθμό των μηνυμάτων ως προς το πλήθος των κόμβων, ο οποίος είναι περίπου γραμμικός. Συνεπώς και οι τρεις αλγόριθμοι επιβεβαιώνουν το θεωρητικό τους άνω φράγμα ως προς τον αριθμό μηνυμάτων.

7.1.4 Μέγεθος Μηνυμάτων

Αφού μελετήθηκε ο αριθμός των μηνυμάτων που μεταδίδονται κατά την εκτέλεση του κάθενός από τους τρεις αλγορίθμους PUSH, PUSH&PULL και MEDIAN-COUNTER, μένει να

αξιολογήσουμε τους αλγορίθμους αυτούς ως προς το μέγεθος των μηνυμάτων που μεταδίδουν. Με τον όρο ‘μέγεθος’ μηνυμάτων δεν εννοώ το μέγεθός τους σε bits, αλλά το μέγεθός τους ως προς τον αριθμό των πληροφοριών (rumors) που εμπεριέχουν. Προτίμησα να μετρήσω τον αριθμό πληροφοριών αντί τον αριθμό των bits λόγω του ότι ο αριθμός των bits θα περιέγραφε την απόδοση των αλγορίθμων ως προς τα συγκεκριμένα μηνύματα που εγώ ανέθεσα στους κόμβους, ενώ ο αριθμός των μηνυμάτων θα οδηγήσει σε πιο γενικά συμπεράσματα ως προς το φιλτράρισμα που εκτελούν οι τρεις αλγορίθμοι. Όπως αναφέρθηκε κατά την περιγραφή του κάθε αλγορίθμου αναμένουμε ότι ο αλγόριθμος PUSH, ο οποίος δεν εκτελεί φιλτράρισμα στις πληροφορίες του, θα έχει πολύ μεγάλο μέγεθος μηνύματος ενώ ο αλγόριθμος MEDIAN-COUNTER που φιλτράρει τις πληροφορίες του και μεταδίδει μόνο αυτές που βρίσκονται σε κατάσταση B ή C θα πετυχαίνει να μειώνει το μέγεθος των μηνυμάτων. Στον αλγόριθμο PUSH&PULL, παρόλο που εκτελείται και πάλι φιλτράρισμα στις πληροφορίες και μεταδίδονται μόνο αυτές που είναι ‘hot’, δεν αναμένουμε να μειώνεται το μέγεθος των μηνυμάτων λόγω του ότι οι πληροφορίες είναι στατικές και δημιουργούνται όλες τις ίδια στιγμή οπότε παύουν όλες να είναι ‘hot’ στον ίδιο γύρο επικοινωνίας.

Στον Πίνακα 7.1 που ακολουθεί, φαίνονται οι σχετικές μετρήσεις για το μέγεθος των μηνυμάτων του κάθε αλγορίθμου για συγκεκριμένο πλήθος κόμβων. Για να υπολογιστεί ο μέσος όρος του μεγέθους των μηνυμάτων, χρειάστηκε ο κάθε κόμβος πριν να αποστείλει ένα μήνυμα να υπολογίζει τον αριθμό των πληροφοριών που εμπεριέχονται σε αυτό και να τον προσθέτει σε μια ειδική μεταβλητή. Με την ολοκλήρωση του αλγορίθμου είχαμε το συνολικό αριθμό πληροφοριών (rumors) που μετάδοσε ο κάθε κόμβος. Λόγω του ότι η εφαρμογή είναι κατανεμημένη δεν μπορούσαν όλοι οι κόμβοι να έχουν μία κοινή μεταβλητή στην οποία να προσθέτουν τον αριθμό των πληροφοριών που μεταδίδουν. Έτσι αφού ολοκληρωνόταν η εφαρμογή, πρόσθετα τον αριθμό πληροφοριών που έστειλε ο κάθε κόμβος και υπολόγιζα το συνολικό αριθμό πληροφοριών που μεταδόθηκαν κατά την εκτέλεση της εφαρμογής. Αυτό επαναλήφθηκε τρεις φορές για κάθε πλήθος κόμβων στον κάθε αλγόριθμο και στο τέλος πάρθηκε ο μέσος όρος των τριών μετρήσεων. Ακολούθως έπρεπε ο συνολικός αριθμός πληροφοριών που μεταδόθηκαν κατά την εκτέλεση της εφαρμογής να διαιρεθεί διά τον αριθμό μηνυμάτων για να παρθεί τελικά ο μέσος όρος των πληροφοριών που εμπεριέχονταν στο κάθε μήνυμα. Πιο κάτω φαίνεται ο πίνακας με τις τελικές μετρήσεις.

Η στήλη ‘Nodes’ αναφέρεται στον αριθμό των κόμβων στην προσομοίωση και η στήλη ‘MessageSize’ στο μέσο όρο του αριθμού πληροφοριών που εμπεριέχονται στο κάθε μήνυμα του αντίστοιχου αλγορίθμου για τον συγκεκριμένο αριθμό κόμβων.

	MessageSize		
Nodes	Push	Push&Pull	Median-Counter
20	7,51	12,38	13,67
40	14,09	24,72	25,51
60	22,62	34,63	37,28
80	29,29	47,51	48,50
100	33,44	57,23	59,33
120	41,93	71,35	70,61
140	48,01	81,72	81,74
160	53,20	91,18	92,66
180	60,05	100,80	103,74
200	64,92	110,43	113,10
220	71,99	128,03	123,25
240	75,77	138,28	133,52

Πίνακας 7.1 Μέγεθος Μηνυμάτων

Ανάλυση Αποτελεσμάτων

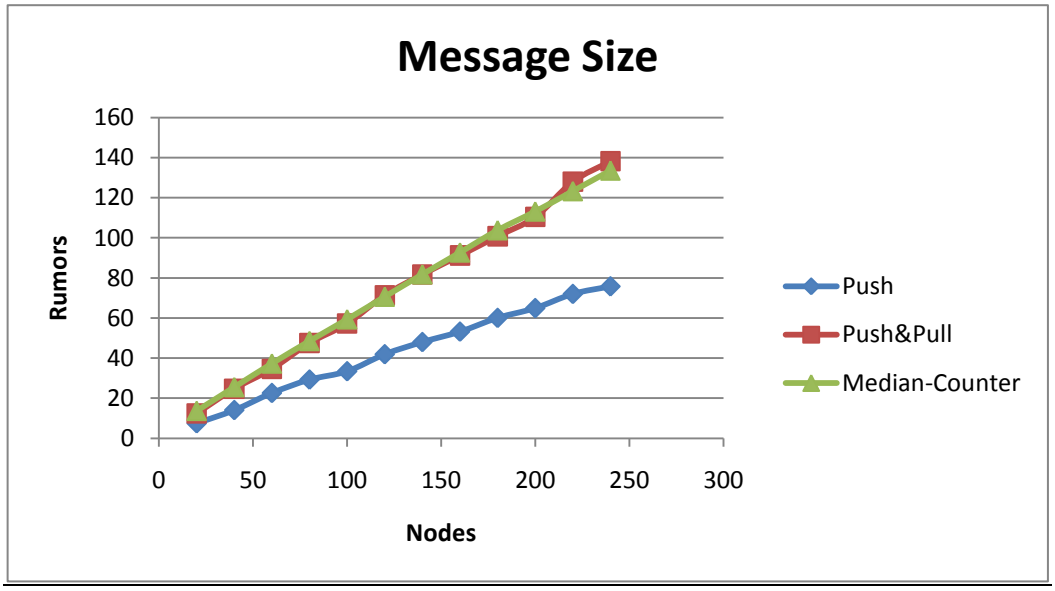
Όπως φαίνεται στον Πίνακα 7.1 όσο μεγαλώνει ο αριθμός των κόμβων στην προσομοίωση μεγαλώνει και το μέγεθος των μηνυμάτων. Αυτό είναι λογικό αφού όπως υλοποιήθηκαν οι τρεις εφαρμογές, μεγαλύτερο πλήθος κόμβων συνεπάγεται περισσότερες πληροφορίες στο δίκτυο οι οποίες πρέπει να μεταδοθούν. Τα αποτελέσματα όμως που αναμέναμε δεν επιβεβαιώθηκαν. Αντιθέτως, ο αλγόριθμος PUSH είναι αυτός με το μικρότερο μέγεθος μηνύματος, ενώ οι αλγόριθμοι PUSH&PULL και MEDIAN-COUNTER έχουν σχεδόν το ίδιο μέγεθος μηνύματος.

Παρόλο που στον αλγόριθμο PUSH&PULL δεν γίνεται κανένα ουσιαστικό φιλτράρισμα, με τον τρόπο που έχει υλοποιηθεί η εφαρμογή, το μέγεθος των μηνυμάτων του δεν είναι μεγαλύτερο από αυτό των μηνυμάτων του αλγορίθμου MEDIAN-COUNTER στον οποίο έχουμε φιλτράρισμα. Για να μειωθεί το μέγεθος το μηνυμάτων του αλγορίθμου MEDIAN-COUNTER πρέπει κάποιες πληροφορίες να μεταβούν σε κατάσταση D ενώ κάποιες άλλες να βρίσκονται ακόμη σε κατάσταση B ή C. Τότε ο αλγόριθμος θα

συνεχίσει να εκτελείται αλλά οι πληροφορίες σε κατάσταση D δε θα μεταδίδονται και το μέγεθος των μηνυμάτων θα μειώνεται. Λόγω του ότι οι πληροφορίες δημιουργούνται όλες τις ίδια στιγμή και δεν υπάρχουν σφάλματα, φαίνεται ότι οι πληροφορίες μεταβαίνουν από τη μια κατάσταση στην άλλη σχεδόν στον ίδιο γύρο επικοινωνίας και έτσι φτάνουν όλες μαζί σε κατάσταση D, οπότε το φιλτάρισμα δεν βοηθά στη μείωση του μεγέθους των μηνυμάτων, όπως συμβαίνει και με τον αλγόριθμο PUSH&PULL.

Επίσης, παρόλο που αναμέναμε ότι ο αλγόριθμος PUSH θα ήταν αυτός με το μεγαλύτερο μέγεθος μηνύματος, παρατηρούμε ότι τελικά έχει το μικρότερο μέγεθος μηνύματος και με αρκετά μεγάλη διαφορά από τους άλλους δύο αλγορίθμους. Αυτό οφείλεται στο γεγονός ότι εκτελείται μόνο η διαδικασία ώθησης μηνυμάτων και έτσι οι κόμβοι καθυστερούν να μάθουν τις πληροφορίες και συνεπώς το μέγεθος των μηνυμάτων τους αργεί να μεγαλώσει. Οι κόμβοι στους άλλους δύο αλγορίθμους μαθαίνουν τις πληροφορίες πιο γρήγορα λόγω της εκτέλεσης της ώθησης αλλά και της έλξης μηνυμάτων. Επομένως τα μηνύματά τους αποκτούν γρήγορα μεγάλο μέγεθος αφού οι κόμβοι μαθαίνουν γρήγορα πολλές πληροφορίες τις οποίες θα μεταδόσουν.

Πιο κάτω φαίνεται η γραφική παράσταση του μεγέθους των μηνυμάτων συναρτήσει του αριθμού των κόμβων του συστήματος για τους τρεις αλγορίθμους:



Σχήμα 7.4 Σύγκριση Αλγορίθμων - Μέγεθος Μηνυμάτων

Όπως βλέπουμε και στη γραφική παράσταση, οι αλγόριθμοι PUSH&PULL και MEDIAN-COUNTER έχουν σχεδόν το ίδιο μέγεθος μηνύματος. Αυτό οφείλεται στο ότι οι

πληροφορίες των κόμβων είναι στατικές και δημιουργούνται όλες την ίδια στιγμή. Αν οι πληροφορίες δεν ήταν στατικές θα φαινόταν καλύτερα το αποτέλεσμα του φιλτραρίσματος των πληροφοριών. Ο αλγόριθμος PUSH φαίνεται να είναι τελικά ο πιο αποδοτικός ως προς το μέγεθος μηνυμάτων, αλλά αυτό συμβαίνει μόνο στην περίπτωση που υπάρχουν μόνο στατικές πληροφορίες στο δίκτυο.

7.2 Συμπεράσματα

Από τη σύγκριση των τριών αλγορίθμων συμπεραίνουμε ότι και οι τρεις αλγόριθμοι είναι αποδοτικοί για την ενημέρωση των κόμβων ενός συστήματος και ο καθένας μπορεί να χρησιμοποιηθεί ανάλογα με τις ανάγκες μιας εφαρμογής. Ο αλγόριθμος PUSH ενημερώνει γρήγορα όλους τους κόμβους του συστήματος και με μικρό αριθμό μηνυμάτων. Επίσης έχει και μικρό μέγεθος μηνυμάτων αλλά μόνο στην περίπτωση στατικών πληροφοριών. Ο αλγόριθμος PUSH&PULL αυξάνει τον αριθμό των μηνυμάτων που ανταλλάσσονται και τον χρόνο εκτέλεσης του αλγορίθμου προσθέτοντας τη λειτουργία της λήψης αλλά μειώνει το μέγεθος των μηνυμάτων φιλτράροντας τις πληροφορίες που μεταδίδονται. Στην περίπτωση όμως της δικής μας υλοποίησης το μέγεθος των μηνυμάτων δεν μειώνεται από το φιλτράρισμα λόγω της στατικής πληροφορίας. Τέλος, ο αλγόριθμος MEDIAN-COUNTER, όπως και ο αλγόριθμος PUSH&PULL, χρησιμοποιεί και τις δύο λειτουργίες, ώθηση και λήψη μηνυμάτων, αυξάνοντας τον αριθμό των μηνυμάτων και το χρόνο εκτέλεσης. Επίσης χρησιμοποιεί φιλτράρισμα πληροφοριών το οποίο όμως και πάλι δεν μείωσε το μέγεθος του μηνύματος λόγω των στατικών πληροφοριών. Το πιο σημαντικό όμως πλεονέκτημα του αλγορίθμου MEDIAN-COUNTER έναντι των άλλων δύο αλγορίθμων είναι η ευρωστία του. Ακόμη και με διαφόρων ειδών σφάλματα ο αλγόριθμος αυτός εξακολουθεί να ενημερώνει τους κόμβους του συστήματος χωρίς να αλλάξει σε μεγάλο βαθμό τον αριθμό των γύρων επικοινωνίας, τον αριθμό των μηνυμάτων καθώς και τον χρόνο εκτέλεσης.

ΚΕΦΑΛΑΙΟ 8

Επίλογος

8.1 Γενικά Συμπεράσματα

Η μετάδοση των πληροφοριών μεταξύ των κόμβων ενός κατανεμημένου συστήματος είναι ένα πρόβλημα που μπορεί να επιλυθεί με πολλούς διαφορετικούς αλγορίθμους. Γι' αυτό και η επιλογή ενός αποδοτικού και εύρωστου αλγόριθμου πληροφόρησης θα είναι καθοριστική για την γενικότερη απόδοση του συστήματος. Στην εργασία αυτή υλοποιήθηκαν και αξιολογήθηκαν τρεις αλγόριθμοι πληροφόρησης οι οποίοι έχουν κάποια κοινά στοιχεία τα οποία χαρακτηρίζουν αυτό το είδος αλγορίθμων, στο τέλος όμως κάθε ένας από τους αλγορίθμους αυτούς είχε διαφορετικά αποτελέσματα στην πειραματική αξιολόγηση και μας οδήγησε σε διαφορετικά συμπεράσματα.

Η απλότητα ενός αλγορίθμου πληροφόρησης όπως είναι ο αλγόριθμος PUSH δεν επιβαρύνει το σύστημα με το κόστος εκτέλεσης επιπλέον λειτουργιών αλλά εκτελεί μόνο τις απαραίτητες λειτουργίες που χρειάζεται ένας τέτοιος αλγόριθμος. Ως αποτέλεσμα, ο χρόνος που χρειάζεται ο αλγόριθμος για να ενημερώσει τους κόμβους είναι αρκετά χαμηλός. Οπότε, αν ο στόχος του συστήματος είναι απλά να ενημερωθούν όλοι οι κόμβοι τότε ο αλγόριθμος PUSH είναι ικανοποιητικός. Τα μηνύματα που μεταδίδονται δεν είναι πολλά, αντιθέτως είναι πολύ λίγα και αυτό μπορεί να θεωρηθεί ως πλεονέκτημα. Επίσης αν η πληροφορία στο δίκτυο είναι στατική, τότε και το μέγεθος των μηνυμάτων δεν είναι πολύ μεγάλο. Ένα σημαντικό στοιχείο του αλγορίθμου PUSH είναι ότι δεν χρησιμοποιεί κάποιο μηχανισμό τερματισμού και ο αλγόριθμος τερματίζει μόνο εάν όλοι οι κόμβοι μάθουν όλες τις πληροφορίες. Αυτό, λόγω του τυχαιοποιημένου τρόπου που γίνεται η επικοινωνία μεταξύ των κόμβων, μπορεί να οδηγήσει σε μη τερματισμό του αλγορίθμου, κάτι που δεν είναι επιθυμητό σε

κανένα σύστημα. Επίσης συνεπάγεται ότι για να λειτουργήσει ο αλγόριθμος πρέπει οι κόμβοι να ξέρουν τον συνολικό αριθμό πληροφοριών που υπάρχουν στο σύστημα, κάτι που δεν μπορεί εύκολα να γίνει σε πραγματική εφαρμογή. Στην ανάλυση του αλγορίθμου ο συνολικός αριθμός των πληροφοριών ήταν ίσος με το συνολικό αριθμό των κόμβων γιατί κάθε κόμβος είχε μια πληροφορία. Σε διαφορετική περίπτωση θα πρέπει να υπολογιστεί ο συνολικός αριθμός πληροφοριών και να μαθευτεί από όλους τους κόμβους του δικτύου.

Σε αντίθεση με τον αλγόριθμο PUSH, ο αλγόριθμος PUSH&PULL χρησιμοποιεί έναν αυστηρότατο μηχανισμό τερματισμού ο οποίος καθορίζει την ακριβής στιγμή που θα τερματίσει ο αλγόριθμος. Αυτό είναι θετικό από την πλευρά του ότι ο αλγόριθμος σίγουρα θα τερματίσει αλλά καθιστά τον αλγόριθμο μη εύρωστο αφού σε περίπτωση που παρουσιαστούν σφάλματα στο δίκτυο ο αλγόριθμος δε θα τα λάβει υπόψη και θα τερματίσει κανονικά στον καθορισμένο χρόνο και πολύ πιθανόν οι κόμβοι του συστήματος να μην έχουν ενημερωθεί. Προσπαθώντας να αυξήσει την πιθανότητα να ενημερωθούν όλοι οι κόμβοι στο προκαθορισμένο χρονικό διάστημα, ο αλγόριθμος PUSH&PULL χρησιμοποιεί εκτός από τη λειτουργία ώθησης μηνυμάτων και τη λειτουργία λήψης μηνυμάτων. Έτσι ο κάθε κόμβος σίγουρα θα παραλάβει μήνυμα σε κάθε γύρο επικοινωνίας ανεξαρτήτως του αν κάποιος γείτονάς του τον επέλεξε για να του στείλει μήνυμα. Το γεγονός όμως ότι σε κάθε γύρο εκτελείται και η ώθηση και η λήψη μηνυμάτων από τον κάθε κόμβο αυξάνει τον αριθμό των μηνυμάτων που μεταδίδονται στο δίκτυο και αυξάνει και τον χρόνο εκτέλεσης της εφαρμογής. Ο χρόνος εκτέλεσης αυξάνεται και λόγω του μηχανισμού τερματισμού του αλγορίθμου ο οποίος απαιτεί επιπλέον λειτουργίες από κάθε κόμβο σε κάθε γύρο επικοινωνίας. Ταυτόχρονα όμως ο αλγόριθμος μειώνει το μέγεθος των μηνυμάτων που αποστέλλει, σε περίπτωση δυναμικής πληροφορίας, μειώνοντας έτσι το φόρτο στο δίκτυο. Ακόμη, ο αλγόριθμος PUSH&PULL όπως και ο επόμενος αλγόριθμος, MEDIAN-COUNTER, δεν χρειάζονται γνώση του συνολικού αριθμού πληροφοριών που υπάρχουν στο δίκτυο όπως συμβαίνει με τον αλγόριθμο PUSH. Αντί αυτού ο κάθε κόμβος χρησιμοποιεί μόνο την τοπική του γνώση όπως συμβαίνει και με τους πλείστους αλγόριθμους πληροφόρησης.

Ο τρίτος και ο πιο πολύπλοκος αλγόριθμος που μελετήθηκε, αλγόριθμος MEDIAN-COUNTER, χρησιμοποιεί και αυτός μηχανισμό τερματισμού χωρίς όμως αυτό να καθιστά τον αλγόριθμο μη εύρωστο. Ο μηχανισμός που χρησιμοποιεί για να τερματίσει δεν

προκαθορίζει την ακριβή στιγμή που θα τερματίσει ο αλγόριθμος αλλά λαμβάνει υπόψη του ότι μία πληροφορία μπορεί να μη μαθευτεί σε κάποιο γύρο και γι' αυτό είναι πιο ευέλικτος ως προς το πια στιγμή η πληροφορία θα σταματήσει να μεταδίδεται. Αυτή η λειτουργία παρόλο που κάνει τον αλγόριθμο ανεκτικό σε σφάλματα του προσθέτει επιπλέον χρόνο εκτέλεσης και οδηγεί σε μετάδοση περισσότερων μηνυμάτων στο δίκτυο. Το επιπλέον όμως κόστος μπορεί να θεωρηθεί ασήμαντο σε σχέση με την ευρωστία που προσθέτει στον αλγόριθμο. Όπως είδαμε ο αλγόριθμος αυτός είτε συμβαίνουν σφάλματα στην εγκατάσταση σύνδεσης, είτε καταρρέουν κόμβοι στο σύστημα θα εξακολουθεί να ενημερώνει τους κόμβους του συστήματος στον επιθυμητό χρόνο και με λογικό επιπλέον κόστος. Επομένως ο αλγόριθμος MEDIAN-COUNTER είναι αυτός που μπορεί πιο εύκολα να υλοποιηθεί και να εφαρμοστεί σε πραγματικά περιβάλλοντα γιατί στα πραγματικά δίκτυα τα σφάλματα είναι αναπόφευκτα.

Αφού έχουν ειπωθεί τα πιο πάνω καταλήγουμε στο συμπέρασμα ότι είναι δύσκολο ένας αλγόριθμος να προσφέρει όλα τα πλεονεκτήματα που μπορεί να έχει ένας αλγόριθμος πληροφόρησης χωρίς να αυξάνει το κόστος του αλγορίθμου είτε σε χρόνο είτε σε μηνύματα. Ανάλογα λοιπόν με τις απαιτήσεις του κάθε κατανεμημένου συστήματος θα πρέπει να επιλεγεί ο κατάλληλος αλγόριθμος έτσι ώστε το κόστος του αλγορίθμου να μην είναι περισσότερο από το ελάχιστο που απαιτεί η εφαρμογή.

Εκτός από τη γενική αντίληψη που αποκτήθηκε γύρω από τους αλγορίθμους πληροφόρησης γενικότερα και συγκεκριμένα για τους αλγορίθμους που μελετήθηκαν στην εργασία αυτή, εκπληρώθηκε και ένας σημαντικός στόχος της εργασίας που ήταν να καταλήξουμε σε συμπεράσματα ως προς το κατά πόσον η θεωρητική και η πρακτική αξιολόγηση των αλγορίθμων που υλοποιήθηκαν συγκλίνουν. Όντως, η πρακτική αξιολόγηση των τριών αλγορίθμων επιβεβαίωσε τα θεωρητικά αποτελέσματα που είχαμε. Συγκεκριμένα, ο αλγόριθμος PUSH ενημερώνει όλους τους κόμβους στο σύστημα σε $O(\ln n)$ γύρους επικοινωνίας και χρησιμοποιώντας $O(n \ln n)$ μηνύματα. Ο αλγόριθμος PUSH&PULL ενημερώνει τους κόμβους σε $\log_3 n + O(\ln \ln n)$ γύρους επικοινωνίας χρησιμοποιώντας $O(n \ln \ln n)$ μηνύματα ενώ ο αλγόριθμος MEDIAN-COUNTER ενημερώνει τους κόμβους σε $O(\ln n)$ γύρους επικοινωνίας και χρησιμοποιώντας $O(n \ln \ln n)$ μηνύματα. Ακόμη, ο αλγόριθμος MEDIAN-COUNTER στην περίπτωση σφαλμάτων επιβεβαιώνει και πάλι τη θεωρία αφού για f το πολύ σφάλματα

ενημερώνει όλους εκτός από $O(f)$ κόμβους τους συστήματος σε $O(\ln n)$ γύρους επικοινωνίας και χρησιμοποιώντας $O(n \ln \ln n)$ μηνύματα.

8.2 Προβλήματα Υλοποίησης Και Πώς Αντιμετωπίστηκαν

Μέσα στα πλαίσια της εργασίας αυτής αντιμετωπιστήκαν κάποια προβλήματα, τόσο κατά την υλοποίηση όσο και κατά την προσομοίωση των αλγορίθμων πληροφόρησης με τους οποίους ασχολήθηκα. Οι πρώτες δυσκολίες που αντιμετώπισα ήταν στην αυτοδίδακτη εκμάθηση των δύο βιβλιοθηκών, GossipLib και YALPS. Λόγω του ότι οι βιβλιοθήκες αυτές έχουν πρόσφατα δημιουργηθεί, η τεκμηρίωσή τους είναι ελάχιστη κάτι που δε βοήθησε καθόλου στην κατανόηση των συναρτήσεων και των λειτουργιών των βιβλιοθηκών. Γι' αυτό το λόγο ζητήθηκε βοήθεια από ένα μέλος της ερευνητικής ομάδας του ιδρύματος INRIA ο οποίος δημιούργησε ένα πρόχειρο εγχειρίδιο χρήσης της βιβλιοθήκης YALPS. Το εγχειρίδιο αυτό αρχικά στάληκε σε μένα και στον επιβλέπων καθηγητή και στη συνέχεια ανέβηκε στη σελίδα της βιβλιοθήκης [5] στον σύνδεσμο 'Tutorial & API'. Ήταν αρκετά βοηθητικό για να κατανοήσω τη βιβλιοθήκη και να μπορέσω να υλοποιήσω και να τρέξω αλγορίθμους σε αυτή, αφού περιλάμβανε επεξήγηση των βασικών συναρτήσεων της βιβλιοθήκης καθώς και περιγραφή του τρόπου με τον οποίο μπορεί να εκτελεστεί μία εφαρμογή της YALPS.

Το επόμενο πρόβλημα που αντιμετώπισα ήταν κατά την πειραματική αξιολόγηση των αλγορίθμων που υλοποιήθηκαν. Η αξιολόγηση των αλγορίθμων πληροφόρησης με τους οποίους ασχολείται η εργασία αυτή έγινε σε προσομοίωση χρησιμοποιώντας τον προσομοιωτή 'SimulatedNodeStarter' της βιβλιοθήκης YALPS. Λόγω της χρήσης εικονικών επεξεργαστών, ο αριθμός των κόμβων που μπορούσαν να χρησιμοποιηθούν στην προσομοίωση ήταν περιορισμένος. Ο μέγιστος αριθμός κόμβων που μπόρεσα να χρησιμοποιήσω ήταν 300 κόμβοι για τον αλγόριθμο PUSH, 260 κόμβοι για τον αλγόριθμο PUSH&PULL και 240 κόμβοι για τον αλγόριθμο MEDIAN-COUNTER. Όσο πιο πολύπλοκος είναι ο αλγόριθμος χρειάζεται περισσότερη χρήση του επεξεργαστή και περισσότερη μνήμη γι' αυτό και ο μέγιστος αριθμός κόμβων μειώνεται. Έτσι η πειραματική αξιολόγηση του αλγορίθμου έγινε κάτω από αυτούς τους περιορισμούς, χωρίς όμως να εμποδίσει την εξαγωγή συμπερασμάτων ως προς την απόδοσή τους. Θα ήταν όμως επιθυμητό να μπορούσαν να χρησιμοποιηθούν περισσότεροι κόμβοι για να είναι πιο γενικά τα αποτελέσματα της αξιολόγησης.

8.3 Μελλοντική Εργασία

Λόγω του ότι στην προσομοίωση χρησιμοποιούνται εικονικοί επεξεργαστές υπήρξαν περιορισμοί ως προς το πλήθος των κόμβων που μπορούσαν να χρησιμοποιηθούν κατά την προσομοίωση της εφαρμογής. Ο μέγιστος αριθμός κόμβων που μπορέσαμε να χρησιμοποιήσουμε σε προσομοίωση ήταν 300 κόμβοι. Δυστυχώς όμως αυτός ο αριθμός ήταν εφικτός μόνο στην προσομοίωση του αλγορίθμου PUSH. Στους υπόλοιπους δύο αλγορίθμους, αλγόριθμος PUSH&PULL και αλγόριθμος MEDIAN-COUNTER, ο μέγιστος αριθμός κόμβων που ήταν δυνατό να χρησιμοποιηθούν ήταν 260 και 240 αντίστοιχα. Αυτό οφείλεται στο ότι οι δύο αυτοί αλγόριθμοι εκτελούν πιο πολύπλοκες λειτουργίες και έτσι οι κόμβοι έχουν περισσότερες απαιτήσεις σε υπολογιστική δύναμη κατά τη διάρκεια της προσομοίωσης.

Αρχικά είχαμε ως στόχο της εργασίας να τρέξουμε τους τρεις αλγόριθμους και σε πραγματικό σύστημα, στο PlanetLab [8], για να μπορέσουν να χρησιμοποιηθούν περισσότεροι κόμβοι και για να δούμε αν τα αποτελέσματα που θα παρθούν είναι όμοια με αυτά της προσομοίωσης. Λόγω όμως τεχνικών δυσκολιών που αντιμετόπιζε η πλατφόρμα PlanetLab και οι οποίες δεν επιδιορθώθηκαν έγκαιρα, ο στόχος αυτός δεν ήταν δυνατό να επιτευχθεί στα πλαίσια αυτής της εργασίας. Θα μπορούσε όμως ο στόχος αυτός να πραγματοποιηθεί σε κάποια μελλοντική εργασία η οποία θα τρέξει τους αλγορίθμους που υλοποιήθηκαν στην εργασία αυτή στο PlanetLab για να παρθούν πιο ρεαλιστικά αποτελέσματα.

8.4 Οφέλη από τη Διπλωματική Εργασία

Μέσα από τη διεκπεραίωση των στόχων της διπλωματικής αυτής εργασίας απέκτησα αρκετά οφέλη. Αρχικά μελέτησα και κατανόησα το σκοπό των αλγορίθμων πληροφόρησης και τη μέγιστη σημασία τους στα σημερινά κατανεμημένα συστήματα. Στη συνέχεια μελέτησα συγκεκριμένους αλγορίθμους πληροφόρησης ξεκινώντας με πιο απλούς αλγόριθμους και καταλήγοντας σε πιο πολύπλοκους και πιο ρεαλιστικούς οι οποίοι με προβλημάτισαν ως προς το αν θα μπορούσαν να γίνουν ακόμη πιο εύρωστοι. Πριν ξεκινήσει το πρακτικό κομμάτι της εργασίας η αυτοδίδακτη εκμάθηση των δύο βιβλιοθηκών GossipLib και YALPS ήταν μία πρωτόγνωρη εμπειρία για μένα αφού δεν είχα ξανά την ευκαιρία να μελετήσω μόνη μου κάποια βιβλιοθήκη χωρίς την βοήθεια

κάποιου διδάσκοντος. Λόγω του ότι οι δύο αυτές βιβλιοθήκες έχουν πρόσφατα δημιουργηθεί δεν είχαν καλή τεκμηρίωση ούτε μπόρεσα να βρω κάποια παραδείγματα υλοποιήσεων που να χρησιμοποιούσαν τις βιβλιοθήκες αυτές και έτσι ήταν δύσκολο αρχικά να τις κατανοήσω. Αφού εξοικειώθηκα με τις δύο αυτές βιβλιοθήκες ξεκίνησα την υλοποίηση των αλγορίθμων η οποία συνέτεινε στην περεταίρω ανάπτυξη των προγραμματιστικών μου ικανοτήτων στη γλώσσα προγραμματισμού JAVA. Τέλος, το σημαντικότατο όφελος που απέκτησα καθ' όλη τη διάρκεια της εργασίας ήταν η εκμάθηση και η κατανόηση της μεθοδολογίας που ακολουθείται για την εκπόνηση μίας τέτοιας εργασίας ξεκινώντας από το στάδιο της έρευνας σχετικά με το θέμα της εργασίας και καταλήγοντας στο στάδιο της αξιολόγησης της εργασίας και των συμπερασμάτων.

ΒΙΒΛΙΟΓΡΑΦΙΑ

[1] R. Karp, C.Schindelhauer, S.Shenker, B.Vocking, "Randomized Rumor Spreading", 2000, 41st Annual Symposium on Foundations of Computer Science

[2] Πρωτόκολλα Πληροφόρησης, [Online]. Available: http://en.wikipedia.org/wiki/Gossip_protocol

[3] Ralitsa Kostadinov, Constantin Adam, "Performance Analysis of the Epidemic Algorithms" [Online]. Available: http://www.s3.kth.se/lcn/courses/2E1624/project-epidemic_algorithms.pdf

[4] Βιβλιοθήκη GossipLib, [Online]. Available: http://gossiplib.gforge.inria.fr/

[5] Βιβλιοθήκη YALPS, [Online]. Available: http://yalps.gforge.inria.fr/

[6] Σειριοποίηση, [Online]. Available: http://en.wikipedia.org/wiki/Serialization

[7] Gianluca Iannaccone, Chen-nee Chuah, Richard Mortier, Supratik Bhattacharyya, Christophe Diot, "Analysis of link failures in an IP backbone" [Online]. Available: http://berkeley.intel-research.net/gianluca/papers/iannaccone.imw02.pdf

[8] Πλατφόρμα PlanetLab, [Online]. Available: http://www.planet-lab.org/

ΠΑΡΑΡΤΗΜΑ Α

Κώδικας Αλγορίθμων που Υλοποιήθηκαν

Α.1 Αλγόριθμος PUSH

```
package yalps.example.application;
import java.io.IOException;
import java.io.Serializable;
import java.net.UnknownHostException;
import java.util.ArrayList;
import java.util.Random;
import yalps.BasicPeriodicTask;
import yalps.NodeID;
import yalps.TaskManager;
import yalps.launchers.AbstractYalpsNode;
import yalps.taskgroup.SystemTaskGroup;
import gossiplib.core.Peer;

//oi rumors tou kathe peer filagontai se mia lista i opoia apotelei
//ta data tou peer aftou (gia na stellonte oloi oi rumors se ena msg)
class NodeContentList implements Serializable {
    private static final long serialVersionUID = 1L;
    ArrayList<String>  rumors; //oi rumors tou peer
    ArrayList<String> tempRumors; //oi rumors pou parelave to peer se ena gyro
    NodeContentList(String name) {
        rumors = new ArrayList<String>();
        rumors.add(name);
        tempRumors = new ArrayList<String>();
```

```
    }
}

class RumorsMessageSt implements Serializable {
    private static final long serialVersionUID = 1L;
    ArrayList<String>  rumors;
    RumorsMessageSt(){
        rumors = new ArrayList<String>();
    }
}
public class exPush extends AbstractYalpsNode{
    Integer numOfPeers;
    class MessageGenerationTask extends BasicPeriodicTask {
        Integer i=0; //metrw posus gyrous tha treksw tin efarmogi
        @Override
        protected TaskManager getTaskManager() {
            return exPush.this.getTaskManager();
        }
        @Override
        protected void periodicRun() {
            i++;
            //vriskw to peer sto opoio anoikei to node afto kai ton
            //arithmo twn dedomenon pou exei afto to peer
            Integer nodeName=Integer.parseInt(getNodeId().getName());
            Peer me=peers.get(nodeName-1);
            NodeContentList myContent=(NodeContentList) me.getPeerData("1");

            //stin arxi tou kathe gyrou epikoinwnias to peer filaei tus rumors pou paralave
ston proigumeno gyro epikoinwnias
```

```
        if (i>1){
            String tempRumor;
            for (Integer r=0; r<myContent.tempRumors.size(); r++){
                tempRumor=myContent.tempRumors.get(r);
                Integer k;
                for (k=0; k<myContent.rumors.size(); k++){
                    if (tempRumor.compareTo(myContent.rumors.get(k))==0)
                        break;
                }
                if (k==myContent.rumors.size()) {//den vrethike to rumor sti lista tou receiving peer
                    //prosthetw to rumor sti lista me ta rumors tou receivng peer
                    myContent.rumors.add(tempRumor);                    }
            }
            //adeiazw ti lista gia na mpun mesa oi rumors pu tha paralavei se afto to gyro epikoinwnias
            myContent.tempRumors.clear();
        }
        //an to peer exei olous tous rumors den tha steilei tipota
        if (myContent.rumors.size()==numOfPeers)
            return;

        //Dimiourgia tou msg pou tha stalei
        RumorsMessageSt data = new RumorsMessageSt();
        data.rumors= (ArrayList<String>) myContent.rumors.clone();

        int size=peers.size();
        int rand;
        NodeID neighbour;
        while(true){
```

```
            //epilego tixaia to peer sto opoio tha steilw to msg me tus rumors
            rand=new Random().nextInt(size);
            Peer p=peers.get(rand);
            neighbour=p.getNodeId();
            try {
                if (neighbour.compareTo(getNodeId())!=0){ //gia na min epileksw ton
eafto mu san neighbour
                    getCommLayer().sendUDP(data, neighbour
                    break;
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
    @Override
    public long getPeriod() {
        // we send messages every second
        return 1000;
    }
    @Override
    public boolean isActive() {
        if (i==20) //trexei 20 gyrous
            return false;
            return true;

    }
}
int end=0;
```

```
    private ArrayList<Peer> peers=new ArrayList<Peer>(); //filaw ta peers
    public void setNodeList(String numNodes) {
        Integer num=Integer.parseInt(numNodes);
        numOfPeers=num;
        for (Integer i=1; i<=num; i++) {
            try {
                NodeID node=cl.getNodeIdFromString(i.toString());
                Peer p=new Peer(node);
                NodeContentList data= new NodeContentList("Msg from
"+node.toString()+"!");
                p.addPeerData("1", data); //to node1 ine sto peer(0)
                peers.add(p);
            } catch (UnknownHostException e) {
                e.printStackTrace();
            }
        }
    }

    //implementation of receive() inherited abstract method
    public boolean receive(short header, Object msg, NodeID sender){
        RumorsMessageSt m=(RumorsMessageSt) msg;
        ArrayList<String> receivedRumors=m.rumors; //oi rumors pou parelave
        //vriskw to peer sto opoio anoikei afto to node (paraliptis)
        Integer nodeName=Integer.parseInt(getNodeId().getName());
        Peer p=peers.get(nodeName-1);
            NodeContentList receiverContent = (NodeContentList) p.getPeerData("1
        ArrayList<String> rumorList = receiverContent.rumors;
        ArrayList<String> tempRumorList = receiverContent.tempRumors; //filaei
prosorina tous rumors pou parelave to peer
        //an to receiving node exei olus tus rumors den tha filaksei tipota kai
```

```
        //tha apantisei ston apostolea me afta pu kserei, diladi me olous tous rumors
        if (rumorList.size()==numOfPeers){
            //an kai o apostoleas exei olous tus rumors, den tha tu apantisw
            if (receivedRumors.size()==numOfPeers)
                return true;
            //Dimiourgia tou msg pou tha stalei
            RumorsMessageSt data = new RumorsMessageSt();
            data.rumors= (ArrayList<String>) receiverContent.rumors.clone();
            try {
                getCommLayer().sendUDP(data, sender);
            } catch (IOException e) {
                e.printStackTrace();
            }
            return true;
        }
        //kathe rumor pu parelave elegxei an iparxei idi o rumor aftos sti lista me tus tempRumors
        //diaforetika ton filaei sti lista tempRumors
        String tempRumor;
        for (Integer r=0; r<receivedRumors.size(); r++){
            tempRumor=receivedRumors.get(r);
            Integer k;
            for (k=0; k<tempRumorList.size(); k++){
                //elegxei an exei paralavei ksana afto to rumor se afto to gyro
                //an oxi ton filaei sti lista tempRumors
                if (tempRumor.compareTo(tempRumorList.get(k))==0)
                    break;
            }
            if (k==tempRumorList.size()) {//den vrethike to rumor sti lista me tus tempRumors tou receiving peer
```

```
                tempRumorList.add(tempRumor); //prosthetw to rumor sti lista me ta
tempRumors tou receivng peer
                }
        }
        return true;
    }
    //implementation of startNode() inherited abstract method
    public void startNode(){
        tm.registerPeriodicTask(new MessageGenerationTask(),
SystemTaskGroup.getInstance(), 0);
    }
}
```

A.2 Αλγόριθμος PUSH&PULL

```
package yalps.example.application;
import java.io.IOException;
import java.io.Serializable;
import java.net.UnknownHostException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Random;
import yalps.BasicPeriodicTask;
import yalps.NodeID;
import yalps.TaskManager;
import yalps.launchers.AbstractYalpsNode;
import yalps.taskgroup.SystemTaskGroup;
import gossiplib.core.Peer;
class rumor implements Serializable{
    private static final long serialVersionUID = 1L;
```

```
    String content;
    Integer birth;
    rumor(String c, Integer round){
        this.content=c;
        this.birth=round;
    }

    boolean isHot(Integer maxAge, Integer round){
        //rumor's age is currentRound-this.birth
        //a rumor is hot if its age is <=log3n +  O(ln(ln(n)))
        if ((round-this.birth)<=maxAge)
            return true;
        return false;
    }
}
//oi rumors tou kathe peer filagontai se mia lista i opoia apotelei
//ta data tou peer aftou (stellonte oloi oi rumors se ena msg)
class NodeContentListDynamic implements Serializable {
    private static final long serialVersionUID = 1L;
    ArrayList<rumor>  rumors; //oi rumors tou peer
    ArrayList<rumor> tempRumors; //oi rumors pou parelave to peer se ena gyro
    ArrayList<rumor> rumorsToSend; //oi hot rumors
    NodeContentListDynamic(rumor r) {
        rumors = new ArrayList<rumor>();
        tempRumors = new ArrayList<rumor>();
        rumorsToSend = new ArrayList<rumor>();
        rumors.add(r);
    }
}
```

```
class RumorsMessage implements Serializable {
    private static final long serialVersionUID = 1L;
    ArrayList<rumor>  rumors;
    Boolean pull; //kathorizei kata poso to minima apostellete kata ti diadiakasia push i pull, pull=true stin pull
    RumorsMessage(){
        rumors = new ArrayList<rumor>();
    }
}
public class exPushAndPull extends AbstractYalpsNode {
Integer numOfPeers;
Integer roundNum=0; //keeps the round number
Integer maxAge;
    class MessageGenerationTask extends BasicPeriodicTask {
        @Override
        protected TaskManager getTaskManager() {
            return exPushAndPull.this.getTaskManager();
        }
        @Override
        protected void periodicRun() {
            roundNum++;
            //vriskw to peer sto opoio anoikei to node afto kai ton
            //arithmo twn dedomenon pou exei afto to peer
            Integer nodeName=Integer.parseInt(getNodeId().getName());
            Peer me=peers.get(nodeName-1);
            NodeContentListDynamic myContent=(NodeContentListDynamic) me.getPeerData("1");
            //stin arxi tou kathe gyrou epikoinwnias to peer filaei tus rumors pou paralave ston proigumeno gyro epikoinwnias
```

```
        if (roundNum>1){
            rumor tempRumor;
            for (Integer r=0; r<myContent.tempRumors.size(); r++){
                tempRumor=myContent.tempRumors.get(r);
                Integer k;
                for (k=0; k<myContent.rumors.size(); k++){
                    if
(tempRumor.content.compareTo(myContent.rumors.get(k).content)==0)
                        break;
                }
                if (k==myContent.rumors.size()) {//den vrethike to rumor sti lista tou
receiving peer
                myContent.rumors.add(tempRumor); //prosthetw to rumor sti lista me ta
rumors tou receivng peer
                }
            }
            //adeiazw ti lista gia na mpun mesa oi rumors pu tha paralavei se afto to
gyro epikoinwnias
            myContent.tempRumors.clear();
        }
        //enimerono ti lista rumorsToSend me tus rumors pu tha stellei to node afto
        rumor temp;
        myContent.rumorsToSend.clear(); //adeiazo ti lista gia na tin enimeroso me ta
nea dedomena
        for (int j=0; j<myContent.rumors.size(); j++){
            temp=myContent.rumors.get(j);
            if (temp.isHot(maxAge, roundNum))
                myContent.rumorsToSend.add(temp);
        }
        //an den exei hot rumors den stellei tipota
```

```
            if (myContent.rumorsToSend.size()==0)
                return;
            //Dimiourgia tou msg pou tha stalei
            RumorsMessage data = new RumorsMessage();
            data.rumors= (ArrayList<rumor>) myContent.rumorsToSend.clone();
            data.pull=false;
            int size=peers.size();
            int rand;
            NodeID neighbour;
            while(true){
                //epilego tixaia to peer sto opoio tha steilw to msg me tus rumors
                rand=new Random().nextInt(size);
                Peer p=peers.get(rand);
                neighbour=p.getNodeId();
                try {
                    if (neighbour.compareTo(getNodeId())!=0){ //gia na min epileksw ton
eafto mu san neighbour
                        getCommLayer().sendUDP(data, neighbour);
                        break;
                    }
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }

        @Override
        public long getPeriod() {
            // we send messages every second
```

```
        return 1000;
    }

    @Override
    public boolean isActive() {
        if (roundNum==maxAge)
            return false;
        return true;
    }
}

private ArrayList<Peer> peers=new ArrayList<Peer>(); //filaw ta peers

public void setNodeList(String numNodes) {
    Integer num=Integer.parseInt(numNodes);
    numOfPeers=num;
    maxAge=(int) Math.round(Math.log(numOfPeers)/Math.log(3)+4*
(Math.log(Math.log(numOfPeers))) );
    for (Integer i=1; i<=num; i++) {
        try {
            NodeID node=cl.getNodeIdFromString(i.toString());
            Peer p=new Peer(node);
            rumor r=new rumor("Msg from "+node.toString()+"!", roundNum);
            NodeContentListDynamic data= new NodeContentListDynamic(r);
            p.addPeerData("1", data); //to node1 ine sto peer(0)
            peers.add(p);
        } catch (UnknownHostException e) {
            e.printStackTrace();
        }
```

```
        }
    }
    //implementation of receive() inherited abstract method
    public boolean receive(short header, Object msg, NodeID sender){
        RumorsMessage m=(RumorsMessage) msg;
        ArrayList<rumor> receivedRumors=m.rumors; //oi rumors pou parelave
        //vriskw to peer sto opoio anoikei afto to node (paraliptis)
        Integer nodeName=Integer.parseInt(getNodeId().getName());
        Peer p=peers.get(nodeName-1);
        NodeContentListDynamic receiverContent = (NodeContentListDynamic)
p.getPeerData("1");
        ArrayList<rumor> tempRumorList = receiverContent.tempRumors; //filaei
prosorina tous rumors pou parelave to peer


        //pull (receiver sends his rumors tu sender)
        //elegxo an o receiver exei hot rumors na steilei
        //an exei esto kai ena hot rumor tha steilei olus tus rumors pu exei ston sender
        if (m.pull==false){
            //an exei hot rumors tha tus steilei
            if (receiverContent.rumorsToSend.size()>0)
            {
                //dimiourgia tou msg pou tha steilei os diadikasia pull
                RumorsMessage data = new RumorsMessage();
                data.rumors= (ArrayList<rumor>) receiverContent.rumorsToSend.clone();
                data.pull=true;
                try {
                    getCommLayer().sendUDP(data, sender);
                } catch (IOException e) {
                    e.printStackTrace();
                }
```

```
            }
        }
        //gia kathe rumor pu parelave elegxei an paralave idi ton rumor afto se afto to
gyro, diaforetika
        //ton filaei sti lista me tus temp rumors tu gia na ton prosthesei ston epomeno gyro
        //sti lista me tus rumors tou
        rumor tempRumor;
        for (Integer r=0; r<receivedRumors.size(); r++){
            tempRumor=receivedRumors.get(r);
            Integer k;
            for (k=0; k<tempRumorList.size(); k++){
                if (tempRumor.content.compareTo(tempRumorList.get(k).content)==0)
                    break;
            }
            if (k==tempRumorList.size()) {//den vrethike to rumor stin temp lista tou
receiving peer
                tempRumorList.add(tempRumor); //prosthetw to rumor sti temp lista me ta
rumors tou receivng peer
            }
        }
        return true;
    }
    //implementation of startNode() inherited abstract method
public void startNode(){
        tm.registerPeriodicTask(new MessageGenerationTask(),
SystemTaskGroup.getInstance(), 0);
    }
}
```

A.3 Αλγόριθμος MEDIAN-COUNTER

```
package yalps.example.application;
```

```
import java.io.IOException;
import java.io.Serializable;
import java.net.UnknownHostException;
import java.util.ArrayList;
import java.util.Random;
import yalps.BasicPeriodicTask;
import yalps.NodeID;
import yalps.TaskManager;
import yalps.launchers.AbstractYalpsNode;
import yalps.taskgroup.SystemTaskGroup;
import gossiplib.core.Peer;

class rumorM implements Serializable{
    private static final long serialVersionUID = 1L;
    String content;   //periexomeno tou rumor
    Character state;  //katastasi tou peer se sxesi me to rumor afto
    Integer ctr;    //an state==B tote ctr=ctr(v,r)
    Integer biggerB;  //num of rumors received with m'>=m
    Integer smallerB; //num of rumors received with m'<m
    Integer numOfRounds; //krata ton arithmo ton rounds pu vriskete o rumor aftos se state C
    Integer birth;     //se pio round dimiourgithike o rumor
    rumorM(String c){
        this.content=c;
        this.state='B';
        this.ctr=1;
        this.numOfRounds=0;
        this.biggerB=0;
        this.smallerB=0;
```

```
        this.birth=0;
    }
    void smallerThanCtrMax(Integer ctrMax){
        if (this.ctr>=ctrMax){
            this.state='C';
            this.numOfRounds=0;
        }
    }
    void checkForStateD(Integer maxNumOfRounds){
        if (this.numOfRounds>=maxNumOfRounds)
            this.state='D';
    }
}
//oi rumors tou kathe peer filagontai se mia lista i opoia apotelei
//ta data tou peer aftou (gia na stellonte oloi oi rumors se ena msg)
class NodeContentListDynamicM implements Serializable {

    private static final long serialVersionUID = 1L;
    ArrayList<rumorM>  rumors;
    ArrayList<rumorM> tempRumors;
    ArrayList<rumorM> rumorsToSend; //oi rumors se state B i C
    NodeContentListDynamicM(rumorM r) {
        rumors = new ArrayList<rumorM>();
        tempRumors = new ArrayList<rumorM>();
        rumorsToSend = new ArrayList<rumorM>();
        rumors.add(r);
    }
}
class RumorsMessageM implements Serializable {
```

```
    private static final long serialVersionUID = 1L;
    ArrayList<rumorM>  rumors;
    Boolean pull; //kathorizei kata poso to minima apostellete kata ti diadiakasia push i pull
    //pull=true stin pull
    RumorsMessageM(){
        rumors = new ArrayList<rumorM>();
    }
}
public class exMedianCounter extends AbstractYalpsNode {
    Integer numOfPeers;
    Integer roundNum=0; //keeps the round number
    Integer ctrMax;
    Integer maxNumOfRounds;
        class MessageGenerationTask extends BasicPeriodicTask {
            @Override
            protected TaskManager getTaskManager() {
                return exMedianCounter.this.getTaskManager();
            }
            @Override
            protected void periodicRun() {
                roundNum++;
                //vriskw to peer sto opoio anoikei to node afto kai ton
                //arithmo twn dedomenon pou exei afto to peer
                Integer nodeName=Integer.parseInt(getNodeId().getName());
                Peer me=peers.get(nodeName-1);
                NodeContentListDynamicM myContent=(NodeContentListDynamicM) me.getPeerData("1");
                if (roundNum>1){
```

```
                //stin arxi tou kathe gyrou epikoinwnias to peer filaei tus rumors pou paralave ston proigumeno gyro epikoinwnias
                rumorM tempRumor;
                for (Integer r=0; r<myContent.tempRumors.size(); r++){
                    tempRumor=myContent.tempRumors.get(r);
                    if (tempRumor.state=='D') //oi rumors stin katastasi D den lamvanonte ipopsi
                        continue;
                    Integer k;
                    for (k=0; k<myContent.rumors.size(); k++){

                        rumorM tempMyContent=myContent.rumors.get(k);
                        if (tempRumor.content.compareTo(tempMyContent.content)==0){
                            //o rumor iparxei idi sti lista me tus rumors tou peer
                            if (tempRumor.state=='C' && tempMyContent.state=='B'){
                tempMyContent.state='C';
    tempMyContent.numOfRounds=-1;}
                            else if (tempRumor.state=='B' && tempMyContent.state=='B')
                            {
                                if (tempRumor.ctr>=tempMyContent.ctr)
    tempMyContent.biggerB++;          else          tempMyContent.smallerB++;
                            }
                            else if (tempRumor.state=='C' && tempMyContent.state=='A'){
    tempMyContent.state='C';                    tempMyContent.numOfRounds=-1;
                }
                            break;
                        }
                    }
                    if (k==myContent.rumors.size()) {//den vrethike to rumor sti lista tou receiving peer
                        if (tempRumor.state=='B')
```

```
                    tempRumor.state='A'; //sto telos tis enimerosis twn rumors oi
rumors me katastasi 'A' metatreponte se rumors me katastasi 'B-1'
                    if (tempRumor.state=='C')
                        tempRumor.numOfRounds=-1;


                    myContent.rumors.add(tempRumor); //prosthetw to rumor sti lista
me ta rumors tou receivng peer                    }
                }
                //adeiazw ti lista gia na mpun mesa oi rumors pu tha paralavei se afto to
gyro epikoinwnias
                myContent.tempRumors.clear();
                for (int j=0; j<myContent.rumors.size(); j++){
                    rumorM tempR=myContent.rumors.get(j);
                    if (tempR.state=='A'){
                        tempR.state='B';
                        tempR.ctr=1;
                        tempR.biggerB=0;
                        tempR.smallerB=0;
                    }
                    else if (tempR.state=='B'){
                        if (tempR.biggerB>tempR.smallerB)
                            tempR.ctr++;
                        tempR.smallerThanCtrMax(ctrMax);
                        tempR.biggerB=0;
                        tempR.smallerB=0;
                    }
                    else if (tempR.state=='C'){
                        tempR.numOfRounds++;
                        tempR.checkForStateD(ctrMax);
                    }
```

```
                }


            }


            //enimerono ti lista rumorsToSend me tus rumors pu tha stellei to node afto
            rumorM temp;
            myContent.rumorsToSend.clear(); //adeiazo ti lista gia na tin enimeroso me ta nea dedomena
            for (int j=0; j<myContent.rumors.size(); j++){
                temp=myContent.rumors.get(j);
                if (temp.state=='B' || temp.state=='C'){ //tha steilei mono tus rumors se katastasi B i C
                    if (roundNum-temp.birth<=maxNumOfRounds) //peer stops propagating the rumor after O(ln(n)) rounds if the counter mechanism fails
                        myContent.rumorsToSend.add(temp);
                }
            }
            //an den exei rumors me state!='D' den stellei tipota
            if (myContent.rumorsToSend.size()==0)
                return;


            //Dimiourgia tou msg pou tha stalei
            RumorsMessageM data = new RumorsMessageM();
            data.rumors= (ArrayList<rumorM>) myContent.rumorsToSend.clone();
            data.pull=false;
            int size=peers.size();
            int rand;
            NodeID neighbour;
            while(true){
                //epilego tixaia to peer sto opoio tha steilw to msg me tus rumors
```

```
                rand=new Random().nextInt(size);
                Peer p=peers.get(rand);
                neighbour=p.getNodeId();
                try {
                    if (neighbour.compareTo(getNodeId())!=0){ //gia na min epileksw ton eafto mu san neighbour
                        getCommLayer().sendUDP(data, neighbour);
                        break;
                    }
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
        @Override
        public long getPeriod() {
            // we send messages every second
            return 1000;
        }
        @Override
        public boolean isActive() {
            if (roundNum==20)
                return false;
            return true;
        }
    }

    private ArrayList<Peer> peers=new ArrayList<Peer>(); //filaw ta peers
```

```
    public void setNodeList(String numNodes) {
        Integer num=Integer.parseInt(numNodes);
        numOfPeers=num;
        ctrMax=4;
        maxNumOfRounds=10*((int) Math.round(Math.log(numOfPeers)));
        for (Integer i=1; i<=num; i++) {
            try {
                NodeID node=cl.getNodeIdFromString(i.toString());
                Peer p=new Peer(node);
                rumorM r=new rumorM("Msg from "+node.toString()+"!");
                NodeContentListDynamicM data= new NodeContentListDynamicM(r);

                p.addPeerData("1", data); //to node1 ine sto peer(0)
                peers.add(p);
                                        } catch (UnknownHostException e) {
                e.printStackTrace();
            }
        }
    }

    //implementation of receive() inherited abstract method
    public boolean receive(short header, Object msg, NodeID sender){
        RumorsMessageM m=(RumorsMessageM) msg;
        ArrayList<rumorM> receivedRumors=m.rumors; //oi rumors pou parelave

        //vriskw to peer sto opoio anoikei afto to node (paraliptis)
        Integer nodeName=Integer.parseInt(getNodeId().getName());
        Peer p=peers.get(nodeName-1);
        NodeContentListDynamicM receiverContent = (NodeContentListDynamicM)
p.getPeerData("1");
```

```
            ArrayList<rumorM> tempRumorList = receiverContent.tempRumors;
            //pull (receiver sends his rumors tu sender)
            //elegxo an o receiver exei rumors sti lista rumorsToSend
            //an exei esto kai ena tetoio rumor tha steilei olus tus rumors aftus ston sender
            if (m.pull==false){
                //an exei rumors me state B i C tus stellei
                if (receiverContent.rumorsToSend.size()>0)
                {
                    //dimiourgia tou msg pou tha steilei os diadikasia pull
                    RumorsMessageM data = new RumorsMessageM();
                    data.rumors= (ArrayList<rumorM>)
receiverContent.rumorsToSend.clone();
                    data.pull=true;
                    try {
                        getCommLayer().sendUDP(data, sender);
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
            }

            //kathe rumor pu parelave se afto to gyro
            //ton filaei sti lista me tus temp rumors tu gia na ton prosthesei ston epomeno
gyro
            //sti lista me tus rumors tou
            rumorM tempRumor;
            for (Integer r=0; r<receivedRumors.size(); r++){
                tempRumor=receivedRumors.get(r);
                tempRumorList.add(tempRumor); //prosthetw to rumor sti lista me tus
tempRumors
```

```
            }
            return true;
        }
        //implementation of startNode() inherited abstract method
public void startNode(){
            tm.registerPeriodicTask(new MessageGenerationTask(),
SystemTaskGroup.getInstance(), 0);
        }
}
```

A.4 Αλγόριθμος MEDIAN-COUNTER με Σφάλματα Συνδέσεων

```
package yalps.example.application;
import java.io.IOException;
import java.io.Serializable;
import java.net.UnknownHostException;
import java.util.ArrayList;
import java.util.Random;
import yalps.BasicPeriodicTask;
import yalps.NodeID;
import yalps.TaskManager;
import yalps.launchers.AbstractYalpsNode;
import yalps.taskgroup.SystemTaskGroup;
import gossiplib.core.Peer;
public class exConnectionFailures extends AbstractYalpsNode {
    Integer numOfPeers;
    Integer roundNum=0; //keeps the round number
    Integer ctrMax;
    Integer maxNumOfRounds;
        class MessageGenerationTask extends BasicPeriodicTask {
```

```
        @Override
        protected TaskManager getTaskManager() {
            return exConnectionFailures.this.getTaskManager();
        }
        @Override
        protected void periodicRun() {
            roundNum++;
            //vriskw to peer sto opoio anoikei to node afto kai ton
            //arithmo twn dedomenon pou exei afto to peer
            Integer nodeName=Integer.parseInt(getNodeId().getName());
            Peer me=peers.get(nodeName-1);
            NodeContentListDynamicM myContent=(NodeContentListDynamicM)
me.getPeerData("1");
            if (roundNum>1){
                //stin arxi tou kathe gyrou epikoinwnias to peer filaei tus rumors pou
paralave ston proigumeno gyro
                //epikoinwnias
                rumorM tempRumor;
                for (Integer r=0; r<myContent.tempRumors.size(); r++){
                    tempRumor=myContent.tempRumors.get(r);
                    if (tempRumor.state=='D') //oi rumors stin katastasi D den lamvanonte
ipopsi
                        continue;
                    Integer k;
                    for (k=0; k<myContent.rumors.size(); k++){
                        rumorM tempMyContent=myContent.rumors.get(k);

                        if (tempRumor.content.compareTo(tempMyContent.content)==0){
                            //o rumor iparxei idi sti lista me tus rumors tou peer
```

```
                        if (tempRumor.state=='C' && tempMyContent.state=='B'){
    tempMyContent.state='C';
    tempMyContent.numOfRounds=-1;
                        }
                        else if (tempRumor.state=='B' && tempMyContent.state=='B')
                        {
                            if (tempRumor.ctr>=tempMyContent.ctr)
                                tempMyContent.biggerB++;
                            else
                            tempMyContent.smallerB++;
                        }
                        else if (tempRumor.state=='C' && tempMyContent.state=='A'){
                tempMyContent.state='C';
    tempMyContent.numOfRounds=-1;
                        }
                        break;
                    }
                }
                if (k==myContent.rumors.size()) {//den vrethike to rumor sti lista tou receiving peer
                    if (tempRumor.state=='B')
                        tempRumor.state='A'; //sto telos tis enimerosis twn rumors oi rumors me katastasi 'A' metatreponte se rumors me katastasi 'B-1'
                    if (tempRumor.state=='C')
                    tempRumor.numOfRounds=-1;
                    myContent.rumors.add(tempRumor); //prosthetw to rumor sti lista me ta rumors tou receivng peer
                }
            }
            //adeiazw ti lista gia na mpun mesa oi rumors pu tha paralavei se afto to gyro epikoinwnias
```

```
                myContent.tempRumors.clear();
                for (int j=0; j<myContent.rumors.size(); j++){
                    rumorM tempR=myContent.rumors.get(j);
                    if (tempR.state=='A'){
                        tempR.state='B';
                        tempR.ctr=1;
                        tempR.biggerB=0;
                        tempR.smallerB=0;
                    }
                    else if (tempR.state=='B'){
                        if (tempR.biggerB>tempR.smallerB)
                            tempR.ctr++;
                        tempR.smallerThanCtrMax(ctrMax);
                        tempR.biggerB=0;
                        tempR.smallerB=0;
                    }
                    else if (tempR.state=='C'){
                        tempR.numOfRounds++;
                        tempR.checkForStateD(ctrMax);
                    }
                }

            }
            //enimerono ti lista rumorsToSend me tus rumors pu tha stellei to node afto
            rumorM temp;
            myContent.rumorsToSend.clear(); //adeiazo ti lista gia na tin enimeroso me ta nea dedomena
            for (int j=0; j<myContent.rumors.size(); j++){
                temp=myContent.rumors.get(j);
```

```
                if (temp.state=='B' || temp.state=='C'){ //tha steilei mono tus rumors se katastasi B i C
                    if (roundNum-temp.birth<=maxNumOfRounds) //peer stops propagating the rumor after O(ln(n)) rounds if the counter mechanism fails
                        myContent.rumorsToSend.add(temp);
                }
            }
            //an den exei rumors me state!='D' den stellei tipota
            if (myContent.rumorsToSend.size()==0)
                return;
            //Dimiourgia tou msg pou tha stalei
            RumorsMessageM data = new RumorsMessageM();
            data.rumors= (ArrayList<rumorM>) myContent.rumorsToSend.clone();
            data.pull=false;
            int size=peers.size();
            int rand;
            NodeID neighbour;
            //random connection failure
            double fail=Math.random();
            while(true){
                //epilego tixaia to peer sto opoio tha steilw to msg me tus rumors
                rand=new Random().nextInt(size);
                Peer p=peers.get(rand);
                neighbour=p.getNodeId();
                try {
                    if (neighbour.compareTo(getNodeId())!=0){ //gia na min epileksw ton eafto mu san neighbour
                        if (fail>0.15) //15% pithanotita g failure. Den exume failures ston prwto gyro gia na min xasume rumors
                            getCommLayer().sendUDP(data, neighbour);
```

```
                    break;
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
    @Override
    public long getPeriod() {
        // we send messages every second
        return 1000;
    }
    @Override
    public boolean isActive() {
        if (roundNum==25)
            return false;
        return true;
    }
}
private ArrayList<Peer> peers=new ArrayList<Peer>(); //filaw ta peers

public void setNodeList(String numNodes) {
    Integer num=Integer.parseInt(numNodes);
    numOfPeers=num;
    ctrMax=4;
    maxNumOfRounds=10*((int) Math.round(Math.log(numOfPeers)));
    for (Integer i=1; i<=num; i++) {
        try {
```

```
                NodeID node=cl.getNodeIdFromString(i.toString());
                Peer p=new Peer(node);
                rumorM r=new rumorM("Msg from "+node.toString()+"!");
                NodeContentListDynamicM data= new NodeContentListDynamicM(r);
                p.addPeerData("1", data); //to node1 ine sto peer(0)
                peers.add(p);
            } catch (UnknownHostException e) {
                e.printStackTrace();
            }
        }
    }

    //implementation of receive() inherited abstract method
    public boolean receive(short header, Object msg, NodeID sender){
        RumorsMessageM m=(RumorsMessageM) msg;
        ArrayList<rumorM> receivedRumors=m.rumors; //oi rumors pou parelave
        //vriskw to peer sto opoio anoikei afto to node (paraliptis)
        Integer nodeName=Integer.parseInt(getNodeId().getName());
        Peer p=peers.get(nodeName-1);
        NodeContentListDynamicM receiverContent = (NodeContentListDynamicM)
p.getPeerData("1");
        ArrayList<rumorM> tempRumorList = receiverContent.tempRumors;
        //pull (receiver sends his rumors tu sender)
        //elegxo an o receiver exei rumors sti lista rumorsToSend
        //an exei esto kai ena tetoio rumor tha steilei olus tus rumors aftus ston sender
        if (m.pull==false){
            //random connection failure
            double fail=Math.random();
            //an exei rumors me state B i C tus stellei
```

```
                if (receiverContent.rumorsToSend.size()>0)
                {
                    //dimiourgia tou msg pou tha steilei os diadikasia pull
                    RumorsMessageM data = new RumorsMessageM();
                    data.rumors= (ArrayList<rumorM>)
receiverContent.rumorsToSend.clone();
                    data.pull=true;
                    try {
                        if (fail>0.15) //15% pithanotita g failure.
                            getCommLayer().sendUDP(data, sender);
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
            }
            //kathe rumor pu parelave se afto to gyro
            //ton filaei sti lista me tus temp rumors tu gia na ton prosthesei ston epomeno
gyro
            //sti lista me tus rumors tou
            rumorM tempRumor;
            for (Integer r=0; r<receivedRumors.size(); r++){
                tempRumor=receivedRumors.get(r);
                tempRumorList.add(tempRumor); //prosthetw to rumor sti lista me tus
tempRumors
            }
            return true;
        }
        //implementation of startNode() inherited abstract method
public void startNode(){
```

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

A.5 Αλγόριθμος MEDIAN-COUNTER με Σφάλματα Κατάρρευσης Κόμβων

```
package yalps.example.application;
import java.io.IOException;
import java.io.Serializable;
import java.net.UnknownHostException;
import java.util.ArrayList;
import java.util.Random;
import yalps.BasicPeriodicTask;
import yalps.NodeID;
import yalps.TaskManager;
import yalps.launchers.AbstractYalpsNode;
import yalps.taskgroup.SystemTaskGroup;
import gossiplib.core.Peer;

//oi rumors tou kathe peer filagontai se mia lista i opoia apotelei
//ta data tou peer aftou (gia na stellonte oloi oi rumors se ena msg)

class NodeContentListDynamicF implements Serializable {

    private static final long serialVersionUID = 1L;
    Boolean fail;
    ArrayList<rumorM>  rumors;
    ArrayList<rumorM> tempRumors;
```

```
    ArrayList<rumorM> rumorsToSend; //oi rumors se state B i C


    NodeContentListDynamicF(rumorM r) {
        rumors = new ArrayList<rumorM>();
        tempRumors = new ArrayList<rumorM>();
        rumorsToSend = new ArrayList<rumorM>();
        rumors.add(r);
        fail=false;
    }
}
public class exNodeFail extends AbstractYalpsNode {
    Integer numOfPeers;
    Integer roundNum=0; //keeps the round number
    Integer ctrMax;
    Integer maxNumOfRounds;
        class MessageGenerationTask extends BasicPeriodicTask {


            @Override
            protected TaskManager getTaskManager() {
                return exNodeFail.this.getTaskManager();
            }
            @Override
            protected void periodicRun() {
                roundNum++;
                //vriskw to peer sto opoio anoikei to node afto kai ton
                //arithmo twn dedomenon pou exei afto to peer
                Integer nodeName=Integer.parseInt(getNodeId().getName());
                Peer me=peers.get(nodeName-1);
```

```
NodeContentListDynamicF myContent=(NodeContentListDynamicF) me.getPeerData("1"
//randomly kapoio peer tha stamata na leiturgei
if (myContent.fail){
    return;
}
if (roundNum>1){
    //stin arxi tou kathe gyrou epikoinwnias to peer filaei tus rumors pou paralave ston proigumeno gyro epikoinwnias
    rumorM tempRumor;
    for (Integer r=0; r<myContent.tempRumors.size(); r++){

        tempRumor=myContent.tempRumors.get(r);
        if (tempRumor.state=='D') //oi rumors stin katastasi D den lamvanonte ipopsi
            continue;

        Integer k;
        for (k=0; k<myContent.rumors.size(); k++){

            rumorM tempMyContent=myContent.rumors.get(k);

            if (tempRumor.content.compareTo(tempMyContent.content)==0){
                //o rumor iparxei idi sti lista me tus rumors tou peer

                if (tempRumor.state=='C' && tempMyContent.state=='B'){
                    tempMyContent.state='C';
                    tempMyContent.numOfRounds=-1;
                }
```

```
                    else if (tempRumor.state=='B' && tempMyContent.state=='B')
                    {
                        if (tempRumor.ctr>=tempMyContent.ctr)
                            tempMyContent.biggerB++;
                        else
                            tempMyContent.smallerB++;
                    }

                    else if (tempRumor.state=='C' && tempMyContent.state=='A'){
                        tempMyContent.state='C';
   tempMyContent.numOfRounds=-1;
                    }
                    break;
                }
            }
            if (k==myContent.rumors.size()) {//den vrethike to rumor sti lista tou receiving peer
                if (tempRumor.state=='B')
                    tempRumor.state='A'; //sto telos tis enimerosis twn rumors oi rumors me katastasi 'A' metatreponte se rumors me katastasi 'B-1'
                if (tempRumor.state=='C')
                    tempRumor.numOfRounds=-1;
   myContent.rumors.add(tempRumor); //prosthetw to rumor sti lista me ta rumors tou receivng peer
            }
        }
        //adeiazw ti lista gia na mpun mesa oi rumors pu tha paralavei se afto to gyro epikoinwnias
        myContent.tempRumors.clear();
        for (int j=0; j<myContent.rumors.size(); j++){
            rumorM tempR=myContent.rumors.get(j);
```

```
                if (tempR.state=='A'){
                    tempR.state='B';
                    tempR.ctr=1;
                    tempR.biggerB=0;
                    tempR.smallerB=0;
                }
                else if (tempR.state=='B'){
                    if (tempR.biggerB>tempR.smallerB)
                        tempR.ctr++;
                    tempR.smallerThanCtrMax(ctrMax);
                    tempR.biggerB=0;
                    tempR.smallerB=0;
                }
                else if (tempR.state=='C'){
                    tempR.numOfRounds++;
                    tempR.checkForStateD(ctrMax);
                }
            }
        }

        //enimerono ti lista rumorsToSend me tus rumors pu tha stellei to node afto
        rumorM temp;
        myContent.rumorsToSend.clear(); //adeiazo ti lista gia na tin enimeroso me ta nea dedomena
        for (int j=0; j<myContent.rumors.size(); j++){
            temp=myContent.rumors.get(j);
            if (temp.state=='B' || temp.state=='C'){ //tha steilei mono tus rumors se katastasi B i C
                if (roundNum-temp.birth<=maxNumOfRounds) //peer stops propagating the rumor after O(ln(n)) rounds if the counter mechanism fails
```

```
                myContent.rumorsToSend.add(temp);
            }
        }
        //an den exei rumors me state!='D' den stellei tipota
        if (myContent.rumorsToSend.size()==0)
            return;
        //Dimiourgia tou msg pou tha stalei
        RumorsMessageM data = new RumorsMessageM();
        data.rumors= (ArrayList<rumorM>) myContent.rumorsToSend.clone();
        data.pull=false;

        int size=peers.size();
        int rand;
        NodeID neighbour;

        while(true){
            //epilego tixaia to peer sto opoio tha steilw to msg me tus rumors
            rand=new Random().nextInt(size);
            Peer p=peers.get(rand);
            neighbour=p.getNodeId();
            try {
                if (neighbour.compareTo(getNodeId())!=0){ //gia na min epileksw ton
eafto mu san neighbour
                    getCommLayer().sendUDP(data, neighbour);
                    break;
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
```

```
        }
        double fail=Math.random();
        if (fail<0.01)
            myContent.fail=true;
    }
    @Override
    public long getPeriod() {
        // we send messages every second
        return 1000;
    }
    @Override
    public boolean isActive() {
        if (roundNum==20)
            return false;

        return true;
    }
}
private ArrayList<Peer> peers=new ArrayList<Peer>(); //filaw ta peers
public void setNodeList(String numNodes) {
    Integer num=Integer.parseInt(numNodes);
    numOfPeers=num;
    ctrMax=4;
    maxNumOfRounds=10*((int) Math.round(Math.log(numOfPeers)));
    for (Integer i=1; i<=num; i++) {
        try {
            NodeID node=cl.getNodeIdFromString(i.toString());
            Peer p=new Peer(node);
            rumorM r=new rumorM("Msg from "+node.toString()+"!");
```

```
                NodeContentListDynamicF data= new NodeContentListDynamicF(r);
                p.addPeerData("1", data); //to node1 ine sto peer(0)
                peers.add(p);

            } catch (UnknownHostException e) {
                e.printStackTrace();
            }
        }
    }
    //implementation of receive() inherited abstract method
    public boolean receive(short header, Object msg, NodeID sender){
        RumorsMessageM m=(RumorsMessageM) msg;
        ArrayList<rumorM> receivedRumors=m.rumors; //oi rumors pou parelave
        //vriskw to peer sto opoio anoikei afto to node (paraliptis)
        Integer nodeName=Integer.parseInt(getNodeId().getName());
        Peer p=peers.get(nodeName-1);

        NodeContentListDynamicF receiverContent = (NodeContentListDynamicF) p.getPeerData("1");
        //an to node pu parelave to minima exei katarrefsei, tha agnohsei apla to minima
        if (receiverContent.fail)
            return true;
        ArrayList<rumorM> tempRumorList = receiverContent.tempRumors;
        //pull (receiver sends his rumors tu sender)
        //elegxo an o receiver exei rumors sti lista rumorsToSend
        //an exei esto kai ena tetoio rumor tha steilei olus tus rumors aftus ston sender
        if (m.pull==false){
            //an exei rumors me state B i C tus stellei
            if (receiverContent.rumorsToSend.size()>0)
```

```
            {
                //dimiourgia tou msg pou tha steilei os diadikasia pull
                RumorsMessageM data = new RumorsMessageM();
                data.rumors=                    (ArrayList<rumorM>)
receiverContent.rumorsToSend.clone();
                data.pull=true;
                try {
                    getCommLayer().sendUDP(data, sender);
                } catch (IOException e) {
                    e.printStackTrace();
                }

            }
        }
        //kathe rumor pu parelave se afto to gyro
        //ton filaei sti lista me tus temp rumors tu gia na ton prosthesei ston epomeno
gyro
        //sti lista me tus rumors tou
        rumorM tempRumor;
        for (Integer r=0; r<receivedRumors.size(); r++){
            tempRumor=receivedRumors.get(r);
            tempRumorList.add(tempRumor); //prosthetw to rumor sti lista me tus
tempRumors
        }
        return true;
    }
    //implementation of startNode() inherited abstract method
    public void startNode(){
        tm.registerPeriodicTask(new                    MessageGenerationTask(),
SystemTaskGroup.getInstance(), 0);
```

```
        }
    }
```