ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΑΤΡΩΝ
Τμήμα Μηχανικών Ηλεκτρονικών Υπολογιστών
και Πληροφορικής

Διπλωματική Εργασία

Μελέτη Κατανεμημένων Αλγορίθμων για Ασύρματα Δίκτυα Αισθητήρων

Δήμητρα Πατρούμπα ΑΜ:2923
patroump@ceid.upatras.gr
Επιβλέπων: Επίκουρος καθηγητής Σωτήρης Νικολετσέας

14 Νοεμβρίου 2007

Πρόλογος

Το όραμα της Περιρρέουσας Νοημοσύνης θέλει τις συσκευές να ενσωματωθούν στο περιβάλλον του ανθρώπου από όπου θα συλλέγουν και θα επεξεργάζονται πληροφορίες. Επιπλέον οι συσκευές θα αλληλεπιδρούν με τον άνθρωπο προσφέροντάς του real-time δεδομένα παντού και πάντα. Η ανάπτυξη της τεχνολογίας των ενσωματωμένων συστημάτων και των ασύρματων επικοινωνιών έχουν οδηγήσει στην πραγματοποίηση ενος μέρους του οράματος μέσω των Ασύρματων Δικτύων Αισθητήρων (Wireless Sensor Networks ή WSNs). Τα WSNs αποτελούνται από ένα μεγάλο αριθμό κόμβων οι οποίοι αλληλεπιδρούν με το περιβάλλον μέσω αισθητήρων, επικοινωνούν μεταξύ τους και συνεργάζονται φέροντας εις πέρας εργασίες που δε θα μπορούσε να ολοκληρώσει ένας μόνο κόμβος. Τα WSNs διαφέρουν από τα παραδοσιακά δίκτυα. Κάθε κόμβος του δικτύου διαθέτει περιορισμένη υπολογιστική δύναμη και ενεργειακούς πόρους, ενώ είναι ευάλωτοι σε καταστροφές από εξωτερικούς παράγοντες, επομένως η δομή του δικτύου αλλάζει πολύ γρήγορα με την πάροδο του χρόνου. Βεβαίως όμως, η δυνατότητα επικοινωνίας και επεξεργασίας των κόμβων καθιστά τα WSNs ένα ισχυρό εργαλείο για την υποστήριξη πολλών εφαρμογών του πραγματικού κόσμου.

Ένα δίκτυο αισθητήρων λοιπόν αποτελεί ένα κατανεμημένο σύστημα. Με βάση την παρατήρηση αυτή στην παρούσα εργασία μελετάται μια κατηγορία αλγορίθμων, οι Κυματικοί αλγόριθμοι, οι οποίοι λύνουν πολλά από τα θεμελιώδη προβλήματα που παρουσιάζονται κατά το σχεδιασμό πρωτοκόλλων για κατανεμημένα συστηματα. Η μελέτη γίνεται στα πλαίσια κάποιων υπαρχόντων πρωτοκόλλων, τα πρωτόκολλα Directed Diffussion και Probabilistic forwarding protocol (PFR), μιας και θα ήταν δυνατόν να χρησιμοποιηθούν οι κυματικοί αλγόριθμοι σε κάποια από τα στάδια των πρωτοκόλλων αυτών για την αύξηση της απόδοσής τους.

Έτσι, υλοποιήθηκαν οι αλγόριθμοι αναζήτησης πρώτα-κατά-βάθος (Depth-first Search), αναζήτησης πρώτα-κατά-εύρος (Breadth-first Search) και ο αλγόριθμος Echo, καθώς επίσης και ένας απλός αλγόριθμος πλημμύρας για την εύρεση των γειτόνων κάθε κόμβου. Η υλοποίηση έγινε στο περιβάλλον του TinyOS, το λειτουργικό σύστημα των δικτύων αισθητήρων και τα πειράματα πραγματοποιήθηκαν με τη βοήθεια του προσομοιωτή του TinyOS, τον TOSSIM.

Ευχαριστίες

Αρχικά θα ήθελα να ευχαριστήσω τον επιβλέπων της διπλωματικής μου, επίκουρο καθηγητή Σωτήρη Νικολετσέα για την ευκαιρία μου έδωσε και την εμπιστοσύνη που μου έδειξε αναθέτοντάς μου τη διπλωματική αυτή. Τον ευχαριστώ για τις πολύτιμες συμβουλές του, καθώς επίσης και για την ελευθερία δράσης που μου έδωσε.

Επίσης, θα ήθελα να ευχαριστήσω τον Δρ. Γιάννη Χατζηγιαννάκη, ο οποίος ήταν ο συνεπιβλέπων στη διπλωματική μου. Ο πολύτιμος χρόνος και η ενέργεια που αφιέρωσε σε όλα τα στάδια της εκπόνησης της διπλωματικής ήταν πολύτιμα για την επιτυχή ολοκλήρωσή της. Τον ευχαριστώ ιδιαιτέρως για την παρότρυνση, τις επισημάνσεις και τις συμβουλές του.

Τέλος, ευχαριστώ την οικογένειά μου και τους φίλους μου για την στήριξη που πάντα μου προσφέρουν.

Περιεχόμενα

Πρόλογος **3**

Ευχαριστίες **5**

1 Εισαγωγή στα Ασύρματα Δίκτυα Αισθητήρων **11**

1.1 Γενικά . . . 11

1.2 Εφαρμογές των Δικτύων Αισθητήρων . . . 13

1.3 Σχεδιασμός ενός δικτύου αισθητήρων . . . 15

1.3.1 Ανοχή σε σφάλματα . . . 15

1.3.2 Κλιμακωσιμότητα . . . 16

1.3.3 Περιορισμοί του υλικού . . . 16

1.3.4 Περιβάλλον . . . 17

1.3.5 Τοπολογία Δικτύου . . . 17

1.3.6 Κατανάλωση ενέργειας . . . 18

1.3.7 Κόστος παραγωγής . . . 19

2 Πλαίσιο Μελέτης Κυματικών Αλγορίθμων **21**

2.1 Directed Diffusion . . . 22

2.1.1 Ονομασία . . . 23

2.1.2 Ενδιαφέροντα και Gradients . . . 24

2.1.3 Διάδοση δεδομένων (Data propagation) . . . 29

2.1.4 Ενίσχυση για σύσταση μονοπατιών και Truncation . . . 30

2.1.5 Αναλυτική εκτίμηση . . . 34

2.2 A Probabilistic Forwarding Protocol . . . 36

2.2.1 Το Μοντέλο . . . 37
2.2.2 Το Πρόβλημα . . . 37
2.2.3 Το Πρωτόκολλο . . . 38

3 Κυματικοί Αλγόριθμοι 41
3.1 Ορισμός και χρήση των Κυματικών Αλγορίθμων . . . 42
3.1.1 Αρχικές υποθέσεις . . . 42
3.1.2 Ορισμοί . . . 42
3.1.3 Στοιχειώδη αποτελέσματα για τους Κυματικούς Αλγορίθμους . . 45
3.1.4 Διάδοση πληροφορίας με ανάδραση . . . 47
3.1.5 Συγχρονισμός . . . 48
3.2 Ο κυματικός αλγόριθμος Echo . . . 49
3.3 Αλγόριθμοι Διάσχισης . . . 52
3.3.1 Διάσχιση συνεκτικών δικτύων . . . 53
3.4 Αναζήτηση πρώτα κατά βάθος . . . 56
3.4.1 Πολυπλοκότητα χρόνου . . . 56
3.4.2 Ο αλγόριθμος . . . 57
3.5 Αναζήτηση πρώτα κατά εύρος . . . 59
3.5.1 Ο αλγόριθμος . . . 59
3.5.2 Χαρακτηριστικά του BFS . . . 60

4 Το Λ.Σ TinyOS και η nesC 61
4.1 Θέματα σχεδιασμού ενός λειτουργικού συστήματος για Ασύρματα Δίκτυα Αισθητήρων . . . 61
4.2 Το λειτουργικό σύστημα TinyOS . . . 63
4.3 Η γλώσσα προγραμματισμού nesC . . . 67
4.4 Άλλα Λειτουργικά Συστήματα . . . 73
4.4.1 Mate . . . 73
4.4.2 MagnetOS . . . 74
4.4.3 MANTIS . . . 75
4.4.4 OSPM . . . 76
4.4.5 EYES OS . . . 77

4.4.6 SenOS 78
4.4.7 EMERALDS 79
4.4.8 PicOS 79

5 Μελέτη Κυματικών Αλγορίθμων 81

5.1 Ο προσομοιωτής του TinyOS, TOSSIM 82
5.2 Διεπαφές 84
5.3 Modules 86
5.3.1 Το BFSM module 87
5.3.2 Το DFS module 94
5.3.3 Το NeighborsDetection module 107
5.3.4 Το Echo module 107
5.4 Configurations 115
5.5 Πειράματα 118

6 Συμπεράσματα και Μελλοντική Εργασία 121

Βιβλιογραφία 123

Κεφάλαιο 1

Εισαγωγή στα Ασύρματα Δίκτυα Αισθητήρων

1.1 Γενικά

Στα τέλη της δεκαετίας του '90 αναπτύχθηκε στον κόσμο των υπολογιστών και της πληροφορικής το όραμα της λεγόμενης 'Περιρρέουσας Νοημοσύνης ' (Ambient Intelligence). Σε έναν κόσμο όπου έχει αναπτυχθεί η Περιρρέουσα Νοημοσύνη, οι συσκευές συλλέγουν και επεξεργάζονται πληροφορίες από το φυσικό περιβάλλον με σκοπό τον έλεγχο των φυσικών διεργασιών καθώς και την αλληλεπίδραση με τον άνθρωπο. Όσο το μέγεθος των συσκευών γίνεται όλο και μικρότερο, η ενσωμάτωσή τους στο περιβάλλον μας θα γίνεται εντονότερη. η τεχνολογία θα βρίσκεται παντού και πάντα (everywhere, everytime).

Για να γίνει πραγματικότητα το όραμα της Περιρρέουσας Νοημοσύνης απαιτούνται κάποιες βασικές τεχνολογίες. Καταρχήν η τεχνολογία των ενσωματωμένων συστημάτων. Τα ενσωματωμένα συστήματα είναι μια ευρέως γνωστή τεχνολογία. Μάλιστα κάποιες εκτιμήσεις δείχνουν ότι το 98% όλων των υπολογιστικών συσκευών είναι ενσωματωμένα συστήματα, συστήματα ειδικού σκοπού. Σπάνια σήμερα μπορεί να βρεθεί νοικοκυριό όπου κάποιος ενσωματωμένος υπολογισμός δεν ελέγχει τις ηλεκτρικές συσκευές.

Ένας ακόμα καίριος παράγοντας για την ανάπτυξη της Περιβαλλοντικής Νοημοσύνης είναι η επικοινωνία. Θα πρέπει οι συσκευές να είναι σε θέση να συνεργάζονται και

η πληροφορίες να μεταφέρονται εκεί που είναι απαραίτητο για να σχηματιστεί μια ολοκληρωμένη εικόνα του περιβάλλοντος. Η ενσύρματη επικοινωνία είναι ακατάλληλη για τέτοιου είδους εφαρμογές. Έτσι η ασύρματη επικοινωνία μεταξύ των ενσωματωμένων συσκευών είναι αναπόφευκτη.

Τα τελευταία χρόνια έχουν κάνει την εμφάνισή τους τα ασύρματα δίκτυα αισθητήρων ή Wireless Sensor Networks(WSN), τα οποία πραγματοποιούν ως ένα σημείο το όραμα της Περιβαλλοντικής Νοημοσύνης. Τα WSNs αποτελούνται από έναν μεγάλο αριθμό κόμβων. Οι κόμβοι αυτοί μπορούν να αλληλεπιδρούν με το περιβάλλον, να επικοινωνούν ασύρματα μεταξύ τους και να συνεργάζονται, φέροντας εις πέρας εργασίες που δε θα μπορούσε να ολοκληρώσει ένας μοναδικός κόμβος. Η δυνατότητα επικοινωνίας και συνεργασίας μεταξύ των κόμβων ανάγει τα WSNs σε ένα κατανεμημένο σύστημα.Οι κόμβοι αποτελούνται τουλάχιστον από αισθητήρες, περιορισμένη υπολογιστική μονάδα, ασύρματη μονάδα επικοινωνίας και λειτουργούν με μπαταρία. Τα WSNs είναι πολύ ισχυρά μιας και μπορούν να υποστηρίξουν πολλές εφαρμογές του πραγματικού κόσμου.Εξαιτίας της ευελιξίας τους αυτής δεν υπάρχει ένα μοναδικό σύνολο απαιτήσεων που να καθορίζει επακριβώς όλα τα WSNs. Για παράδειγμα, σε πολλές εφαρμογές η διατήρηση ενέργειας είναι πολύτιμη. Συχνά, η χωρητικότητα της μπαταρίας είναι άμεσα συνυφασμένη με το μέγεθος του κόμβου, ενώ η τιμή του έχει άμεσο αντίκτυπο στην ποιότητα των αισθητήρων του, επηρεάζοντας την ακρίβεια του αποτελέσματος του. Γενικά, αν οι κόμβοι που είναι κοντά στο υπό παρατήρηση φαινόμενο είναι απλοί, αλλά σε μεγάλο αριθμό, μπορούν να κάνουν την αρχιτεκτονική του συστήματος απλή και αποδοτικότερη ως προς τη διατήρηση της ενέργειας, καθώς διευκολύνουν την κατανεμημένη δειγματοληψία (ο εντοπισμός ενός αντικειμένου, για παράδειγμα, απαιτεί κατανεμημένο σύστημα).

Τα WSNs διαφέρουν από τα παραδοσιακά δίκτυα, μιας και είναι δυναμικά μεταβαλλόμενα. Η εξάντληση των ενεργειακών πόρων έχει ως αποτέλεσμα την παύση της λειτουργίας κάποιων κόμβων ενώ, εξαιτίας του μικρού τους μεγέθους, κάποιοι μπορεί να καταστραφούν από εξωτερικούς παράγοντες. Έτσι, ίσως χρειαστεί η εισαγωγή νέων κόμβων στο δίκτυο.

Για να να γίνει πιο κατανοητή η ευελιξία των WSNs ας δούμε κάποιες από τις εφαρμογές τους.

1.2 Εφαρμογές των Δικτύων Αισθητήρων

Η τεχνολογία των Ασύρματων Δικτύων Αισθητήρων μπορεί να εφαρμοστεί σε πολλές εφαρμογές του πραγματικού κόσμου και να φέρει στην επιφάνεια κάποιες εντελώς καινούριες.

Ένα κρίσιμο και πρωτεύον συστατικό των κόμβων των ασύρματων δικτύων αισθητήρων είναι ο αισθητήρας. Για πολλές παραμέτρους του φυσικού περιβάλλοντος υπάρχει η κατάλληλη τεχνολογία αισθητήρα που μπορεί να ενσωματωθεί σε ένα WSN. Οι πιο ευρέως χρησιμοποιούμενοι είναι οι αισθητήρες θερμοκρασίας, υγρασίας, ήχου, πίεσης και οι χημικοί αισθητήρες.

Μια σύντομη λίστα με τις πιο βασικές εφαρμογές παρουσιάζεται παρακάτω:

- **Πρόληψη Καταστροφών.** Μια από τις πιο συχνά αναφερόμενες εφαρμογές των WSNs είναι στην πρόληψη καταστροφών. Ένα τυπικό σενάριο για εφαρμογές αυτής της κατηγορίας είναι η ανίχνευση πυρκαγιών. Οι κόμβοι αισθητήρων είναι εξοπλισμένοι με θερμόμετρα και μπορούν να υπολογίσουν τη θέση τους τρέχοντας κάποιον αλγόριθμο εντοπισμού θέσης (localization). Τους κόμβους αυτούς μπορούμε να τους απλώσουμε σε ένα δάσος, πετώντας τους από ένα αεροπλάνο. Έτσι σχηματίζεται ένας θερμοκρασιακός χάρτης της περιοχής και σε περίπτωση υψηλών θερμοκρασιών και χαμηλής υγρασίας που υπονοούν πυρκαγιά ενημερώνουν τους πυροσβέστες.

- **Έλεγχος του περιβάλλοντος και της βιοποικιλότητας.** Τα WSNs μπορούν να χρησιμοποιηθούν για να ελέγχουν το περιβάλλον ως προς τους χημικούς ρύπους ή ακόμα και για το σχηματισμό μιας εικόνας ως προς τον αριθμό των διαφορετικών ειδών πανίδας και χλωρίδας μια περιοχής.

- **Ευφυή Κτίρια.** Τα μεγάλα κτίρια συχνά καταναλώνουν μεγάλα ποσά ενέργειας εξαιτίας λανθασμένης χρήσης των συσκευών Air Condtitioning (HVAC). Μια αποδοτικότερη, πραγματικού-χρόνου και ακριβέστερη παρακολούθηση της θερμοκρασίας, της υγρασίας και άλλων παραμέτρων μπορεί να μειώσει την κατανάλωση ενέργειας. Επίσης, μπορούν να χρησιμοποιηθούν για την παρακολούθηση των μηχανικών καταπονήσεων σε κτίρια ή γέφυρες που βρίσκονται σε σεισμικά ενεργές

ζώνες, ενώ άλλου τύπου αισθητήρες μπορούν χρησιμοποιηθούν για τον εντοπισμό εγκλωβισμένων ανθρώπων σε περιπτώσεις σεισμού. Οι αισθητήρες μπορούν να τοποθετηθούν στα κτίρια τη στιγμή της κατασκευής τους ή αφού έχουν κατασκευαστεί. Σε αυτές τις εφαρμογές η εξοικονόμηση ενέργειας για τους αισθητήρες είναι πολύ σημαντική απαίτηση.

- **Διαχείριση Εγκαταστάσεων.** Τα WSNs μπορούν να χρησιμοποιηθούν για εφαρμογές διαχείρισης μεγάλων εγκαταστάσεων, όπως θέματα ασφαλείας. Η είσοδος των ανθρώπων στις εγκαταστάσεις μπορεί να γίνεται χωρίς κλειδιά, αλλά με τη χρήση κάποιου πομπού, ενώ μπορούν να εντοπίζονται πιθανοί εισβολείς. Επίσης σε χημικές εγκαταστάσεις τα WSNs θα μπορούσαν να χρησιμοποιηθούν για τον εντοπισμό διαρροών.

- **Συντήρηση Μηχανών.** Αισθητήρες μπορούν να τοποθετηθούν σε δυσπρόσιτα σημεία μηχανών για να ελέγχουν τους κραδασμούς που υποδεικνύουν ανάγκη για συντήρηση. Παραδείγματα τέτοιων μηχανών είναι αυτόματες μηχανές ή οι άξονες των τροχών των τρένων.

- **Εφαρμογές στη Γεωργία.** Η εφαρμογή WSNs σε καλλιεργήσιμες εκτάσεις με τοποθέτηση αισθητήρων μέτρησης υγρασίας και ανάλυσης της σύστασης του εδάφους επιτρέπει την ακριβέστερη και αποδοτικότερη λίπανση και άρδευση των εκτάσεων. Επίσης, η εκτροφή ζώων μπορεί να ωφεληθεί τοποθετώντας αισθητήρες στα ζώα που ελέγχουν την κατάσταση της υγείας τους.

- **Εφαρμογές στον τομέα της υγείας.** Η χρήση WSNs στον τομέα της υγεία μπορεί να αποδειχτεί πολύ ωφέλιμη. Όμως υπάρχουν αρκεά ηθικά διλήμματα πάνω στο θέμα αυτό.Οι πιθανές εφαρμογές εκτείνονται από την άμεση τοποθέτηση αισθητήρων στον ίδιο τον ασθενή για την παρακολούθηση της υγείας του και ίσως αυτόματη χορήγηση φαρμάκων, μέχρι την παρακολούθηση των ιατρών και των ασθενών στα νοσοκομεία.

- **Ευφυή οδικά συστήματα.** Στα ευφυή οδικά συστήματα αισθητήρες τοποθετούνται στους δρόμους, ακόμα και στα κράσπεδα των δρόμων οι οποίοι συλλέγουν

πληροφορίες για την κίνηση και την κατάσταση του οδικού δικτύου γενικότερα και επικοινωνούν με τους οδηγούς δίνοντάς τους χρήσιμες πληροφορίες.

- **Στρατιωτικές Εφαρμογές.** Τα WSNs μπορούν να είναι ενιαίο και αναπόσπαστο τμήμα των στρατιωτικών συστημάτων. Τα χαρακτηριστικά των WSNs, όπως είναι η γρήγορη τοποθέτηση τους, η αυτοοργάνωση και η ανοχή στα σφάλματα, τα μετατρέπουν σε μια πολλά υποσχόμενη τεχνολογία για τα στρατιωτικά συστήματα. Κάποιες από τις πιθανές στρατιωτικές εφαρμογές τους είναι η παρακολούθηση της κατάστασης των εξοπλισμών και των πολεμοφοδίων, η στενή παρακολούθηση του πεδίου της μάχης, η αναγνώριση των εχθρικών δυνάμεων, η εκτίμηση των καταστροφών μετά από μάχη καθώς και ο εντοπισμός και η αναγνώριση χημικής, ατομικής ή βιολογικής επίθεσης.

1.3 Παράγοντες που επηρεάζουν το σχεδιασμό ενός δικτύου αισθητήρων

Πολλοί είναι οι παράγοντες που επηρεάζουν το σχεδιασμό ενός δικτύου αισθητήρων. Οι παράγοντες αυτοί αποτελούν έναν οδηγό για τη δημιουργία των πρωτοκόλλων και των αλγορίθμων που εφαρμόζονται στα δίκτυα αισθητήρων.

1.3.1 Ανοχή σε σφάλματα

Για ένα δίκτυο αισθητήρων, η πιθανότητα να σταματήσει η λειτουργία κάποιων κόμβων λόγω καταστροφής ή εξάντλησης της ενέργειάς τους είναι πολύ μεγάλη. Το σφάλμα σε ένα κόμβο δε θα πρέπει να επηρεάζει τη συνολική λειτουργία του δικτύου. Επομένως, η αξιοπιστία ή η ανοχή σε σφάλματα ορίζεται ως η ικανότητα διατήρησης των λειτουργιών του δικτύου χωρίς διακοπές εξαιτίας αποτυχιών των κόμβων. Για να είναι ανεκτή η καταστροφή κάποιων κόμβων, η χρησιμοποίηση πλεοναζόντων κόμβων είναι απαραίτητη. Βεβαίως, όμως, τα επίπεδα της επιθυμητής αξιοπιστίας εξαρτώνται από την ίδια την εφαρμογή στην οποία χρησιμοποιείται το δίκτυο αισθητήρων.

1.3.2 Κλιμακωσιμότητα

Ανάλογα με την εφαρμογή, ένα δίκτυο αισθητήρων μπορεί να αποτελείται από ένα αριθμό κόμβων που φτάνει τις εκατοντάδες ή και τις χιλιάδες. Επομένως, οι τα πρωτόκολλα και οι αρχιτεκτονικές που εφαρμόζονται θα πρέπει όχι μόνο να ανταπεξέλθονται σε διαφορετικούς αριθμούς κόμβων, αλλά και να αξιοποιούν την πυκνότητα του δικτύου.

1.3.3 Περιορισμοί του υλικού

Οι κόμβοι ενός δικτύου αισθητήρων απαρτίζεται από τέσσερις βασικές μονάδες: *μονάδα ανίχνευσης, μονάδα επεξεργασίας δεδομένων, πομποδέκτης* και *μονάδα πηγής ενέργειας*. Επιπλέον, μπορεί να υπάρχουν και κάποιες άλλες μονάδες, όπως ένα σύστημα εύρεσης τοποθεσίας, γεννήτρια ενέργειας και κινητήρας. Η μονάδα ανίχνευσης αποτελείται από τον αισθητήρα και από έναν ADC ενώ η μονάδα επεξεργασίας συνδέεται με μια μονάδα μνήμης. Όλες οι παραπάνω μονάδες πρέπει να τοποθετηθούν σε μια συσκευή μεγέθους που μπορεί να είναι μικρότερη και από ένα κυβικό εκατοστό.

Το μικρό μεγέθος βάζει περιορισμούς και στην ενέργεια που μπορεί να αποθηκευτεί σε έναν κόμβο. Για παράδειγμα, σε έναν smart dust κόμβο η συνολική αποθηκευμένη ενέργεια είναι της τάξης του 1J. Ο χρόνος ζωής ενός κόμβου μπορεί να επιμηκυνθεί αντλώντας ενέργεια από το περιβάλλον του. Μια πιθανή τεχνική είναι η χρησιμοποίηση φωτοβολταϊκών στοιχείων για αποθήκευση της ηλιακής ενέργειας.

Οι πομποδέκτες είναι απαραίτητος εξοπλισμός των κόμβων αλλά και η κύρια πηγή κατανάλωσης ενέργειας. Συνήθως χρησιμοποιούνται πομποδέκτες RF τεχνολογίας, αλλά μπορούν να χρησιμοποιηθούν και συσκευές οπτικής επικοινωνίας.

Αν και η τεχνολογία στους μικροεπεξεργαστές έχει αναπτυχθεί τόσο ώστε να έχουν μεγαλύτερη υπολογιστική δύναμη σε μικρότερο μέγεθος, ωστόσο οι υπολογιστικές και οι δυνατότητες μνήμης που χρησιμοποιούνται στους κόμβους αισθητήρων δεν είναι πολύ μεγάλες. Για παράδειγμα, σε έναν κόμβο 'έξυπνης σκόνης ' η επεξεργαστική μονάδα είναι ένας 4MHz Atmel AVR 8535 μικροελεγκτής, με 8 KB flash μνήμη, 512 bytes RAM και 512 bytes EEPROM. Το λειτουργικό σύστημα που χρησιμοποιείται είναι το TinyOS.

Η τυχαία τοποθέτηση των κόμβων αισθητήρων κάνει την χρήση ενός συστήματος

εύρεσης της ακριβής τοποθεσίας του κόμβου απαραίτητη. Τα συστήματα αυτά απαιτούνται και από πολλούς αλγορίθμους δρομολόγησης. Σε πολλά δίκτυα αισθητήρων οι κόμβοι είναι εξοπλισμένοι με ένα global positioning system (GPS).

1.3.4 Περιβάλλον

Οι εφαρμογές των δικτύων αισθητήρων απαιτούν το στήσιμο των δικτύων σε οποιοδήποτε περιβάλλον. Οι συνθήκες που επικρατούν μπορεί να είναι ακραίες, όμως το δίκτυο θα πρέπει, ανεπιτήρητο, να λειτουργεί χωρίς διακοπές και με τη μέγιστη ακρίβεια. Οι αισθητήρες μπορεί να λειτουργούν σε:

- Στο βυθό ενός ωκεανού
- Στο κέντρο ενός κυκλώνα
- Στο πεδίο μιας μάχης
- Σε μεγάλα κτίρια, σε σπίτια ή σε καταστήματα

1.3.5 Τοπολογία Δικτύου

Η διατήρηση της τοπολογίας ενός δικτύου με ένα τόσο μεγάλο αριθμό κόμβων, ειδικά από τη στιγμή που πολλοί κόμβοι παρουσιάζουν βλάβες και αχρηστεύονται, απαιτεί πολύ προσεκτικούς χειρισμούς. Μπορούμε να διακρίνουμε τρεις φάσεις για τη διατήρηση της τοπολογίας σε ένα δίκτυο αισθητήρων:

Φάση προετοιμασίας και ανάπτυξης της τοπολογίας

Οι κόμβοι σε ένα δίκτυο αισθητήρων συνήθως τοποθετούνται με τυχαίο τρόπο. Κοινές πρακτικές είναι οι ρίψεις από αεροπλάνο ή από καταπέλτη, όμως μπορούν να τοποθετηθούν και χειρωνακτικά, ένας ένας. Ο μεγάλος αριθμός κόμβων εμποδίζει την ανάπτυξη μιας προσεκτικά μελετημένης διάταξης. Εν τούτοις η αρχική τοποθέτηση θα πρέπει να ικανοποιεί κάποια κριτήρια, όπως την ελαχιστοποίηση του κόστους εγκατάστασης και τη μεγιστοποίηση της ευελιξίας, της αξιοπιστίας και της δυνατότητας αυτοοργάνωσης του δικτύου.

Φάση μετά την ανάπτυξη της τοπολογίας

Η τοπολογία ενός δικτύου αισθητήρων είναι επιρρεπής σε συχνές αλλαγές, μετά την ανάπτυξή της. Οι αλλαγές αυτές οφείλονται σε βλάβες στους κόμβους ή ενεργειακής εξάντλησής τους, στην εμφάνιση εμποδίων που μετακινούνται και στη μετακίνηση των κόμβων που μπορεί να συμβεί εξατίας φυσικών παραγόντων (μπορεί για παράδειγμα να μετακινηθούν λόγω αέρα ή να παρασυρθούν από τη βροχή). Επομένως, τα πρωτόκολλα που σχεδιάζονται για ένα δίκτυο αισθητήρων θα πρέπει να εξασφαλίζουν την αποκατάσταση - επανοργάνωση του δικτύου ακόμη και μετά από μεγάλες αλλαγές.

Φάση επανατοποθέτησης πρόσθετων κόμβων

Συχνά καινούριοι κόμβοι προστίθενται στο δίκτυο για να αντικαταστήσουν τους κατεστραμένους. Έτσι θα πρέπει να ξεκινήσει μια φάση επανοργάνωσης του δικτύου ώστε να αξιοποιούνται όλοι οι κόμβοι. Η φάση αυτή είναι προαιρετική και συμβαίνει όταν θέλουμε να αυξηθεί η διάρκεια ζωής του δικτύου.

1.3.6 Κατανάλωση ενέργειας

Οι κόμβοι των αισθητήρων είναι εξοπλισμένοι με μια περιορισμένη πηγή ενέργειας, λόγω του μικρού μεγέθους τους. Στις περισσότερες εφαρμογές η αναπλήρωση των ενεργειακών πόρων είναι αδύνατη. Επομένως, ο χρόνος ζωής ενός κόμβου σε ένα δίκτυο αισθητήρων και ως εκ τούτου και ο χρόνος ζωής του ίδιου του δικτύου, εξαρτάται άμεσα από τη διάρκεια ζωής των μπαταριών. Έτσι λοιπόν, η αποδοτική διαχείριση ενέργειας είναι πρωτεύοντος σημασίας κατά το σχεδιασμό ενός αλγορίθμου ή πρωτοκόλλου για ένα δίκτυο αισθητήρων.

Παρόλα αυτά, ο χρόνος ζωής ενός δικτύου έχει κάποια trade-offs σε σχέση με την ποιότητα υπηρεσιών: καταναλώνοντας περισσότερη ενέργεια αυξάνεται η ακρίβεια των αποτελεσμάτων που παράγει ένας κόμβος, όμως μειώνεται ο χρόνος ζωής του. Οι σχεδιαστές θα πρέπει να εξισορροπούν τους περιορισμούς αυτούς και να σχεδιάζουν πρωτόκολλα που τους αντιμετωπίζουν με το βέλτιστο τρόπο, ανάλογα βεβαίως και με την εφαρμογή.

Τρεις είναι οι τομείς όπου καταναλώνεται το μεγαλύτερο μέρος της ενέργειας: η

μονάδα ανίχνευσης, η μονάδα επικοινωνίας και η μονάδα επεξεργασίας των δεδομένων.

Μονάδα ανίχνευσης περιβάλλοντος

Καθοριστικό ρόλο στην κατανάλωση ενέργειας από τη μονάδα ανίχνευσης παίζει η πολυπλοκότητα της ανίχνευσης. Τα επίπεδα θορύβου από το περιβάλλον μπορούν να προκαλέσουν άυξηση της πολυπλοκότητας ανίχνευσης. Επιπλέον, η σποραδική παρακολούθηση του περιβάλλοντος απαιτεί λιγότερη ενέργεια από τη συνεχή επιτήρηση.

Μονάδα επικοινωνίας

Η αποστολή και λήψη δεδομένων είναι οι πιο δαπανηρές ενέργειες που επιτελεί ένας κόμβος. Μάλιστα, κατά την αποστολή των μηνυμάτων το σήμα θα πρέπει να ενισχυθεί ώστε να έχουμε αξιόπιστη μετάδοση και να υπάρχουν όσο το δυνατόν λιγότερα σφάλματα, τα οποία προκαλούνται από τον παρεμβάλλοντα θόρυβο και τις ανακλάσεις. Στον υπολογισμό της ενέργειας για την επικοινωνία πρέπει να συνυπολογιστεί και η ενέργεια που απαιτείται για την εκκίνηση του πομποδέκτη.

Μονάδα επεξεργασίας δεδομένων

Για την ελαχιστοποίηση της ενέργειας που καταναλώνεται στη μονάδα επεξεργασίας δεδομένων, οι επεξεργαστές παρέχουν καταστάσεις λειτουργίας που καταναλώνουν διαφορετικά ποσά ενέργειας. Έτσι, ανάλογα με το φόρτο εργασίας οι επεξεργαστές μεταβαίνουν στις αντίστοιχες καταστάσεις, χωρίς να απαιτείται η συνεχής κανονική τους λειτουργία.

1.3.7 Κόστος παραγωγής

Εφόσον ένα δίκτυο αισθητήρων αποτελείται από ένα μεγάλο αριθμό κόμβων, είναι σημαντικό το κόστος του κόμβου να έχει την ελάχιστη δυνατή τιμή. Έχει τεθεί σαν στόχος η τιμή ενός κόμβου να φτάσει το 1 ευρώ. Έτσι, από τη στιγμή που ένας κόμβος αισθητήρα πρέπει να επιτελέσει ένα αρκετά μεγάλο αριθμό λειτουργιών, είναι ένα ενδιαφέρον (αλλά αρκετά δύσκολο) θέμα η επίτευξη της ελαχιστοποίησης της τιμής του.

Κεφάλαιο 2

Πλαίσιο Μελέτης Κυματικών Αλγορίθμων

Η αποτελεσματικότητα των δικτύων αισθητήρων οφείλεται στην ικανότητά τους να επικοινωνούν και να συνεργάζονται ώστε να φέρουν σε πέρας δύσκολες εργασίες παρακολούθησης μιας περιοχής. Η συνεργασία αυτή έγκειται στην αλληλεπίδραση μεταξύ γειτονικών κόμβων και διάδοση με αυτό τον τρόπο της πληροφορίας μέσα στο δίκτυο.

Στο παρόν κεφάλαιο μελετάται η πρακτική του *directed-diffusion* για την επίτευξη αυτής της συνεργασίας. Σύμφωνα με το directed-diffusion, όταν ένας κόμβος ενδιαφέρεται για κάποια πληροφορία, διαχέει το ενδιαφέρον του μέσα στο δίκτυο. Έτσι, όταν κάποιος άλλος κόμβος έχει κάποια δεδομένα που ταιριάζουν στην πληροφορία που ζητήθηκε, τα στέλνει πίσω στον κόμβο που έστειλε το ενδιαφέρον. Το directed-diffusion λοιπόν, εισάγει ένα δεδομενο-κεντρικό τρόπο λειτουργίας του δικτύου. Η διάδοση του ενδιαφέροντος στο δίκτυο των αισθητήρων θα πρέπει να γίνεται με έναν αποδοτικό τρόπο, έτσι ώστε να ικανοποιούνται οι απαιτήσεις για ευρωστία, κλιμακωσιμότητα και μικρή κατανάλωση ενέργειας.

Επίσης, γίνεται μια επισκόπηση του *PFR* πρωτοκόλλου, το οποίο αποτελείται από δύο φάσεις. Η πρώτη δεν είναι τίποτε άλλο παρά ένας μηχανισμός πλημμύρας για διάδοση πληροφορίας και αρχικοποίηση του δικτύου. Η δεύτερη φάση είναι το πιθανοτικό τμήμα του πρωτοκόλλου, εκεί όπου η μετάδοση από κόμβο σε κόμβο πραγματοποιείται (ή όχι) με μια συγκεκριμένη πιθανότητα. Το PFR επιλέχθηκε να παρουσιαστεί μιας και μπορεί

να αποτελέσει μέρος του directed-diffusion· έτσι η διάχυση της πληροφορίας μπορεί να γίνει με πιθανοτικό τρόπο.

Χρησιμοποιώντας τα παραπάνω πρωτόκολλα μπορούμε να ορίσουμε με ακρίβεια τον τρόπο λειτουργίας ενός κυματικού αλγορίθμου. Όπως περιγράφεται στο επόμενο κεφάλαιο, οι κυματικοί αλγόριθμοι δίνουν αποδοτικές και αξιόπιστες λύσεις σε κάποια θεμελιώδη προβλήματα, που εμφανίζονται συχνά κατά το σχεδιασμό πρωτοκόλλων για κατανεμημένα συστήματα. Συνεπώς, η βέλτιστη μελέτη ενός κυματικού αλγορίθμου προκύπτει όταν γίνεται στα πλαίσια της ακριβέστερης γνώσης για το υπό μελέτη σύστημα.

Ας δούμε τα παραπάνω πρωτόκολλα πιο αναλυτικά.

2.1 Directed Diffusion

Ας δούμε ένα παράδειγμα για να καταλάβουμε τη λογική του directed-diffusion. Έστω σε ένα δίκτυο αισθητήρων, ένας ή περισσότεροι χειριστές θέτουν σε κάποιον κόμβο του δικτύου ερωτήσεις της μορφής: 'Πόσους πεζούς παρατηρείς στη γεωγραφική περιοχή Χ;' ή 'Σε ποια κατεύθυνση κινείται το όχημα στην περιοχή Υ;'. Το ερώτημα αυτό θα μετατραπεί σε ένα ενδιαφέρον που διαχέεται (χρησιμοποιώντας για παράδειγμα broadcast, γεωγραφική δρομολόγηση) προς τους κόμβους στην περιοχή Χ ή στην περιοχή Υ. Όταν ένας κόμβος στην περιοχή αυτή λάβει ένα ενδιαφέρον, ενεργοποιεί τους αισθητήρες του, οι οποίοι ξεκινάνε να συλλέγουν πληροφορίες για κινούμενα αντικείμενα. Όταν οι αισθητήρες αναφέρουν την παρατήρηση της ύπαρξης κάποιου κινούμενου αντικειμένου, η πληροφορία αυτή δρομολογείται στην αντίστροφη (σε σχέση με αυτή του ενδιαφέροντος) κατεύθυνση του μονοπατιού. Οι ενδιάμεσοι κόμβοι ίσως να ενοποιήσουν τα δεδομένα, για παράδειγμα να υποδείξουν με μεγαλύτερη ακρίβεια τη θέση του κινούμενου αντικειμένου, συνδυάζοντας πληροφορίες που φτάνουν από πολλούς κόμβους. Ένα ενδιαφέρον χαρακτηριστικό του directed-diffusion είναι το γεγονός ότι η διάδοση του ενδιαφέροντος και της πληροφορίας και η ενοποίηση των δεδομένων αποφασίζεται μετά από τοπικές αλληλεπιδράσεις, δηλαδή με μηνύματα που ανταλλάσσονται μεταξύ γειτονικών κόμβων.

Τα στοιχεία που συνθέτουν το directed-diffusion είναι τα: ενδιαφέροντα, μηνύματα δεδομένων, gradients, ενισχύσεις. Ένα *ενδιαφέρον* είναι ένα ερώτημα που περιγράφει

αυτό που ζητάει ο χρήστης. *Δεδομένα* είναι οι πληροφορίες που συλλέγονται ή επεξεργάζονται και περιγράφουν ένα φυσικό φαινόμενο. Ένα παράδειγμα δεδομένων είναι το *γεγονός* που είναι μια σύντομη περιγραφή του γεγονότος που παρατηρείται. Τα δεδομένα ονοματίζονται χρησιμοποιώντας ζευγάρια χαρακτηριστικών-τιμών. Μια εργασία παρατήρησης ενός γεγονότος διαδίδεται μέσα στο δίκτυο αισθητήρων σαν ένα ενδιαφέρον για δεδομένα που έχουν ένα συγκεκριμένο όνομα. Από αυτή τη διάδοση προκύπτουν τα *gradients*, που είναι ουσιαστικά μια τιμή κατεύθυνσης που δημιουργείται σε κάθε κόμβο που λαμβάνει ένα ενδιαφέρον. Η κατεύθυνση αυτή δείχνει στον κόμβο από τον οποίο προήλθε το ενδιαφέρον. Τα δεδομένα που συλλέγονται επιστρέφουν στον κόμβο που δημιούργησε το ενδιαφέρον χρησιμοποιώντας τα gradients και κινούμενα σε μονοπάτια προς την κατεύθυνση που δείχνουν αυτά τα gradients. Το δίκτυο αισθητήρων ενισχύει ένα ή περισσότερα από αυτά τα μονοπάτια.

2.1.1 Ονομασία

Δεδομένων των δυνατοτήτων ανίχνευσης που υποστηρίζει το δίκτυο αισθητήρων, η επιλογή ενός σχήματος ονοματολογίας για τα δεδομένα είναι το πρώτο βήμα για το σχεδιασμό του directed diffusion για το δίκτυο. Όπως αναφέρθηκε και παραπάνω, το όνομα ενός ενδιαφέροντος αποτελείται από ένα σύνολο χαρακτηριστικών και απαιτούμενων τιμών για τα χαρακτηριστικά αυτά. Τα δεδομένα που στέλνονται ως απόκριση στο ενδιαφέρον ονομάζονται και αυτά με βάση ένα παρόμοιο σχήμα ονομασίας.

Έστω για παράδειγμα δημιουργείται μια αίτηση, ένα ενδιαφέρον, για ανίχνευση κινούμενων οχημάτων. Τότε το όνομα του ενδιαφέροντος θα έχει ως εξής:

type = wheeled vehicle – εύρεση οχήματος με ρόδες
interval = 20 ms – αποστολή γεγονότος κάθε 20ms
duration = 10 s – για τα επόμενα 10s
rect = [-100, 100, 200, 400] – από αισθητήρες που βρίσκονται σε αυτή την περιοχή

Αν κάποιος κόμβος ανιχνεύσει ένα κινούμενο όχημα που απαντάει στο παραπάνω ερώτημα θα δημιουργήσει ένα μήνυμα δεδομένων με το ακόλουθο όνομα:

type = wheeled vehicle – τύπος οχήματος που ανιχνεύθηκε

interval = truck – στιγμιότυπο αυτού του τύπου
location = [125, 220] – θέση κόμβου
intensity = 0.6 – το πλάτος του σήματος
intensity = 0.85 – βεβαιότητα ότι η πληροφορία ταιριάζει στις απαιτήσεις του ενδιαφέροντος
timestamp = 01 : 20 : 40 – χρονική στιγμή δημιουργίας του γεγονότος

2.1.2 Ενδιαφέροντα και Gradients

Καταρχήν ας ονομάσουμε sink τον κόμβο μέσω του οποίου εισάγεται τυχαία το ενδιαφέρον στο δίκτυο.

Α. Διάδοση Ενδιαφέροντος Χρησιμοποιώντας το παραπάνω παράδειγμα, έστω ότι μια αίτηση με συγκεκριμένο τύπο (type), περιοχή (rect), διάρκεια (duration) 10s και μια περίοδο (interval) των 10ms αρχικοποιείται σε ένα κόμβο. Ο κόμβος sink καταγράφει την αίτηση· μετά από χρόνο (duration) η αίτηση διαγράφεται από τον κόμβο.

Για κάθε ενεργή αίτηση, ο κόμβος sink περιοδικά εκπέμπει ένα μήνυμα ενδιαφέροντος σε όλους τους γείτονές του. Το αρχικό ενδιαφέρον περιέχει τα παραπάνω χαρακτηριστικά περιοχής και διάρκειας, όμως η περίοδος είναι πολύ μεγαλύτερη. Διαισθητικά μπορούμε να αντιληφθούμε το αρχικό αυτό μήνυμα ενδιαφέροντος σαν διερευνητικό,που προσπαθεί να εξακριβώσει αν όντως υπάρχουν κόμβοι που εντοπίζονται το όχημα που ζητείται. Για να γίνει κάτι τέτοιο, το διερευνητικό αυτό μήνυμα περιέχει ένα ρυθμό δεδομένων (το αντίστροφο της περιόδου) που είναι πολύ χαμηλός.
Τότε το μήνυμα παίρνει τη μορφή:

type = wheeled vehicle
interval = 1s
rect = [-100, 200, 200, 400]
timestamp = 01 : 20 : 40 //hh:mm:ss
expiresAt = 01 : 30 : 40

Εδώ πρέπει να σημειωθεί ότι το μήνυμα ενδιαφέροντος είναι ‘soft state’, δηλαδή

περιοδικά ανανεώνεται από τον κόμβο sink, απλά στέλνοντάς το ξανά αυξάνοντας το timestamp. Αυτή είναι κάτι το σημαντικό γιατί η αποστολή των μηνυμάτων ενδιαφέροντος δεν είναι αξιόπιστη. Ο ρυθμός ανανέωσης είναι μια σχεδιαστική παράμετρος που προκαλεί ένα trade-off μεταξύ overhead και αύξησης αξιοπιστίας ακόμα και με μηνύματα ενδιαφέροντος που χάνονται.

Κάθε κόμβος διατηρεί ένα χώρο αποθήκευσης για τα μηνύματα ενδιαφέροντος που λαμβάνει· ο χώρος αυτός ονομάζεται *interest cache*. Για κάθε ενδιαφέρον υπάρχει μια εγγραφή. Οι εγγραφές αυτές διατηρούν πληροφορία μόνο για τον γειτονικό κόμβο από όπου ελήφθη το ενδιαφέρον και όχι για τον κόμβο sink. Κάθε εγγραφή αποτελείται από το πεδίο timestamp, που δείχνει τη χρονική στιγμή που ελήφθη το ενδιαφέρον, και από τα πεδία gradients,διατηρεί ένα για κάθε γειτονικό κόμβο. Κάθε gradient πεδίο περιέχει ένα πεδίο datarate, που αντιστοιχεί στο ρυθμό μετάδοσης δεδομένων που ζήτησε ο αντίστοιχος γείτονας, και ένα πεδίο duration, που δείχνει το χρόνο ζωής του ενδιαφέροντος. Η τιμή του duration θα πρέπει να είναι μικρότερη από την καθυστέρηση του δικτύου.

Ας δούμε τώρα τι συμβαίνει όταν ένας κόμβος λαμβάνει ένα μήνυμα ενδιαφέροντος. Καταρχήν ελέγχει την interest cache για το αν υπάρχει ήδη μια εγγραφή για το συγκεκριμένο μήνυμα. Αν δεν υπάρχει, δημιουργεί μια νέα εγγραφή. Τα πεδία της εγγραφής που περιγράφησαν παραπάνω παίρνουν τις τιμές τους από το περιεχόμενο του μηνύματος ενδιαφέροντος. Γυρνώντας στο παράδειγμά μας, βλέπουμε πως η εγγραφή για το συγκεκριμένο ενδιαφέρον θα έχει ένα gradient για τον κόμβο από όπου ελήφθη το μήνυμα και ένα datarate με την τιμή του ενός γεγονότος ανά δευτερόλεπτο. Για να είναι δυνατό κάτι τέτοιο, θα πρέπει να είναι δυνατός και ο διαχωρισμός των κόμβων. Έτσι, κάθε κόμβος απαιτείται να έχει ένα μοναδικό αναγνωριστικό που να τον χαρακτηρίζει. Για παράδειγμα θα μπορούσαν να χρησιμοποιηθούν οι 802.11 MAC διευθύνσεις, οι Bluetooth cluster διευθύνσεις ή τοπικές εφήμερες μοναδικές διευθύνσεις. Αν ο κόμβος βρει στην interest cache μια εγγραφή για το συγκεκριμένο μήνυμα ενδιαφέροντος, αλλά δεν υπάρχει gradient για τον αποστολέα, προσθέτει ένα με τη συγκεκριμένη τιμή. Επιπλέον, ενημερώνει τις τιμές των timestamp και duration. Τέλος, αν ο κόμβος βρει μια εγγραφή που έχει και τιμή gradient για το συγκεκριμένο αποστολέα, απλά ενημερώνει τα πεδία timestamp και duration.

Σχετικά με τα gradients. Όταν εκπνέει ο χρόνος ζωής κάποιου gradient, τότε

αφαιρείται από την εγγραφή του ενδιαφέροντος. Δεν λήγουν όλα τα gradients στον ίδιο χρόνο· εξαρτάται από την τιμή των timestamp και duration, τα οποία έχουν αρχικοποιηθεί από τον κόμβο sink. Όταν έχουν λήξει όλα τα gradients για μια εγγραφή ενδιαφέροντος, η εγγραφή αυτή αφαιρείται από την cache.

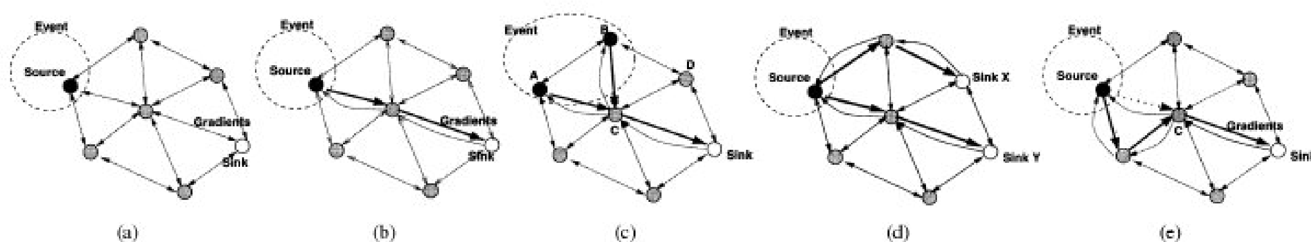
Μετά τους παραπάνω ελέγχους, ο κόμβος ίσως αποφασίσει να στείλει το ενδιαφέρον που έλαβε προς ένα υποσύνολο των γειτόνων του, χωρίς να στείλει και κάποια επιπλέον πληροφορία για τον κόμβο που δημιούργησε αρχικά το ενδιαφέρον. Έτσι φαίνεται σαν να δημιουργήθηκε σε αυτόν τον κόμβο το ενδιαφέρον και όχι σε κάποιο μακρινό κόμβο sink. Έτσι θα το αντιληφθούν και οι γείτονες που θα λάβουν το ενδιαφέρον από αυτόν τον κόμβο. Το παραπάνω είναι ένα παράδειγμα τοπικής αλληλεπίδρασης και με αυτή τη διαδικασία το ενδιαφέρον ταξιδεύει και διαδίδεται μέσα στο δίκτυο αισθητήρων. Δεν επαναποστέλονται όλα τα ενδιαφέροντα που λαμβάνει ένας κόμβος. Κάποια από αυτά ίσως αποφασίσει να μην τα στείλει, λόγω του ότι έστειλε ένα ίδιο ενδιαφέρον πριν ένα μικρό χρονικό διάστημα.

Γενικά, υπάρχουν πολλές εναλλακτικές αποφάσεις που μπορεί να ληφθούν σχετικά με τη διάδοση του ενδιαφέροντος, όπως φαίνεται και στον πίνακα 2.1. Η πιο απλή είναι η επαναμετάδοση του ενδιαφέροντος σε όλους τους γειτονικούς κόμβους. Αυτό ισοδυναμεί με τη διαδικασία της πλημμύρας. Αν δεν είναι γνωστή η πληροφορία για το ποιοι κόμβοι μπορούν να εξυπηρετήσουν το ενδιαφέρον, αυτή ίσως να είναι η μοναδική λύση. Υπάρχουν και άλλοι τρόποι επαναμετάδοσης, όπως είναι η γεωγραφική δρομολόγηση, με την οποία περιορίζεται η έκταση του χώρου για τη διάδοση. Κάτι τέτοιο είναι επιθυμητό από την άποψη της διατήρησης ενέργειας. Επίσης, για δίκτυο αισθηρων με ακίνητους κόμβους, ένας κόμβος μπορεί να χρησιμοποιήσει μια cache δεδομένων για την επαναμετάδοση των ενδιαφερόντων. Για παράδειγμα, αν κάποιος κόμβος 'ακούσει' δεδομένα να στέλνονται σε έναν γειτονικό του κόμβο A από ένα κόμβο που βρίσκεται στην περιοχή που ορίζει το χαρακτηριστικό rect,ως απόκριση σε ενδιαφέρον, μπορεί να κατευθύνει το ενδιαφέρον σε αυτόν τον κόμβο.

B. Εγκαθίδρυση των Gradients Στην περίπτωση της διάδοσης των μηνυμάτων στο δίκτυο με τη μέθοδο της πλημμύρας, τα gradients σχηματίζονται με τον τρόπο

Diffusion Element	Design Choices
Interest Propagation	• Echo algorithm • Constrained or directional flooding based on location • Directional propagation based on previously cached data
Data Propagation	• Breadth-first Search algorithm • Multipath delivery with selective quality along different paths • PFR protocol using the BFS and Echo Algorithms
Data caching and aggregation	• For robust data delivery in the face of node failure • For coordinated sensing and data reduction • For directing interests
Reinforcement	• Rules for deciding when to reinforce • Rules for how many neighbors to reinforce • Negative reinforcement mechanisms and rules

Πίνακας 2.1: Σχεδιαστικές αποφάσεις για το directed diffusion



Σχήμα 2.1: a.Εγκαθίδρυση των gradients b.Ενίσχυση c.Πολλαπλοί sources d.Πολλαπλοί sinks e.Επιδιόρθωση

που φαίνεται στο σχήμα 2.1(a). Αυτό που μπορεί να παρατηρήσει κανείς είναι η άμεση συνέπεια της τοπικής αλληλεπίδρασης μεταξύ των κόμβων, το γεγονός δηλαδή ότι σχηματίζεται ένα gradient για κάθε ζευγάρι γειτονικών κόμβων. Όταν ένας κόμβος λάβει ένα μήνυμα ενδιαφέροντος από τους γειτονικούς του κόμβους δεν είναι σε θέση να γνωρίζει αν το ενδιαφέρον αυτό έρχεται σε απόκριση ενός ενδιαφέροντος που έστειλε νωρίτερα ή αν προέρχεται από έναν άλλο μακρινό κόμβο sink που βρίσκεται στην άλλη πλευρά του γείτονά του. Κάτι τέτοιο οδηγεί στην πιθανή λήψη από ένα κόμβο του ίδιου αντιγράφου χαμηλού ρυθμού δεδομένων από κάθε γειτονικό του κόμβο, κάτι που δεν είναι ιδιαιτέως επιθυμητό· από την άλλη πλευρά βέβαια, κάτι τέτοιο συνεισφέρει στην ανάκαμψη από κατεστραμμένα μονοπάτια ή στην ενίσχυση των μονοπατιών που αποδεικνύεται ότι είναι πιο αποδοτικά και επιπλέον δεν παρασύρει τον αλγόριθμο σε κύκλους ώστε να εγκλωβιστεί σε επαναληπτόμενους βρόχους.

Αυτό που κάνουν ουσιαστικά τα gradients σε ένα κόμβο είναι να ορίζουν για κάθε γειτονικό του κόμβο μια κατεύθυνση και ένα ρυθμό μετάδοσης δεδομένων για την αποστολή των γεγονότων. Έτσι ο σχεδιαστής μπορεί να χρησιμοποιήσει το χαρακτηριστικό αυτό με τον τρόπο που τον διευκολύνει περισσότερο. Στο σχήμα 2.1(c) παρουσιάζεται ένα gradient που παίρνει δυαδικές τιμές. Στο παράδειγμά μας, το gradient ορίζει δύο τιμές που καθορίζουν το ρυθμό που θα στέλνονται τα γεγονότα. Σε άλλα δίκτυα αισθητήρων μπορεί να χρησιμοποιηθεί για τη μετάδοση των μηνυμάτων στους επόμενους κόμβους, στέλνοντάς τα με πιθανοτικό τρόπο προς διάφορα μονοπάτια, επιτυγχάνοντας έτσι εξισορρόπηση του φορτίου του δικτύου.

Γ. Γενικά Η διαδικασία του Interest propagation, της διάδοσης δηλαδή του ενδιαφέροντος, εκτελείται με σκοπό να στήσει έτσι το δίκτυο των αισθητήρων ώστε να δημιουργήσει δρόμους που θα οδηγούν τις πληροφορίες που συλλέγουν και επεξεργάζονται οι κόμβοι προς τον κόμβο sink. Όπως περιγράφηκε παραπάνω, η διάδοση γίνεται κάθε φορά σε τοπικό επίπεδο, δηλαδή μεταξύ γειτονικών κόμβων.

Ο τρόπος διάδοσης που περιγράφηκε παραπάνω βασίστηκε σε ένα συγκεκριμένο παράδειγμα, σε ένα συγκεκριμένο ερώτημα που θα πρέπει να απαντήσουν οι κόμβοι. Έτσι, για διαφορετικά ερωτήματα κάποιοι κανόνες της διάδοσης μπορεί να διαφέρουν, όπως για παράδειγμα ο ρυθμός μετάδοσης δεδομένων. Παρόλα αυτά, οι βασικοί κανόνες, όπως η διατήρηση της interest cache σε κάθε κόμβο, οι κανόνες διάδοσης του ενδιαφέροντος,κ.ά, δεν αλλάζουν.

2.1.3 Διάδοση δεδομένων (Data propagation)

Όταν ένας κόμβος λάβει ένα μήνυμα ενδιαφέροντος και η θέση του βρίσκεται στην περιοχή που ορίζεται από το χαρακτηριστικό rect που απαιτεί το ενδιαφέρον, βάζει σε λειτουργία τους αισθητήρες του και αρχίζει να συλλέγει πληροφορίες για το περιβάλλον του. Όταν εντοπίσει κάποιο στόχο, ελέγχει την cache των ενδιαφερόντων και ψάχνει να βρει αν υπάρχει κάποια εγγραφή που να ταιριάζει στα χαρακτηριστικά του στόχου που εντόπισε. Μόλις βρει μια τέτοια εγγραφή, υπολογίζει το gradient με το μεγαλύτερο ρυθμό αποστολής δεδομένων που απαιτήθηκε και ρυθμίζει τους αισθητήρες του ώστε να παράγουν γεγονότα με αυτό το ρυθμό. Στο παράδειγμά μας ο ρυθμός αυτός είναι ένα γεγονός ανά δευτερόλεπτο. Ο κόμβος που εντόπισε το στόχο ονομάζεται source. Ο κόμβος source, λοιπόν, ξεκινάει να στέλνει σε όλους τους κόμβους για τους οποίους έχει gradient για το συγκεκριμένο ενδιαφέρον, μια περιγραφή του γεγονότος κάθε δευτερόλεπτο, ένα μήνυμα δεδομένων δηλαδή ανά δευτερόλεπτο. Η περιγραφή του γεγονότος έχει την παρακάτω μορφή:

type = wheeled vehicle //τύπος του οχήματος που εντοπίστηκε
instance = truck //στιγμιότυπο αυτού του τύπου
location = [125, 220] // θέση του κόμβου
intensity = 0.6 // μέτρο του πλάτους του σήματος

confidence = 0.85 // βεβαιότητα ότι η πληροφορία ταιριάζει στις απαιτήσεις του ενδιαφέροντος
timestamp = 01 : 20 : 40 // χρόνος παραγωγής του γεγονότος

Το μήνυμα στέλνεται ξεχωριστά σε κάθε σχετικό κόμβο.

Ας δούμε τώρα τι γίνεται όταν ένας κόμβος λαμβάνει ένα μήνυμα δεδομένων. Αρχικά ελέγχει την interest cache για εύρεση εγγραφής που να ταιριάζει με τα χαρακτηριστικά που περιέχει το μήνυμα δεδομένων που έλαβε. Αν δεν υπάρχει, δεν προχωράει σε καμιά ενέργεια. Αν υπάρχει, τότε ελέγχει την *data cache* που σχετίζεται με τη συγκεκριμένη εγγραφή ενδιαφέροντος. Η data cache διατηρεί όλα τα δεδομένα που έχουν φτάσει στον κόμβο και που απαντούν στα ενδιαφέροντα που έχει αποθηκευμένα. Αν βρεθεί μια εγγραφή σε αυτή την cache, ο κόμβος πάλι δεν προχωράει σε καμιά ενέργεια. Αν όμως δεν υπάρχει προσθέτει την εγγραφή και στέλνει το μήνυμα των δεδομένων σε όλους τους γειτονικούς του κόμβους.

Στο σημείο αυτό ο κόμβος θα πρέπει να αποφασίσει το ρυθμό με τον οποίο θα στέλνει τα δεδομένα. Έτσι υπολογίζει το ρυθμό των εισερχόμενων δεδομένων, ελέγχοντας τη data cache. Κατόπι, ελέγχει τα gradients που υπάρχουν για το συγκεκριμένο ενδιαφέρον. Αν όλες ορίζουν ρυθμό μετάδοσης μεγαλύτερο ή ίσο από τον εισερχόμενο, ο κόμβος απλά επαναμεταδίδει τα μηνύματα στους κατάλληλους γειτονικούς κόμβους. Αν όμως κάποια gradients ορίζουν μικρότερο ρυθμό μετάδοσης, θα πρέπει πρώτα να ρίξει το ρυθμό των εισερχόμενων μηνυμάτων στην κατάλληλη τιμή.

2.1.4 Ενίσχυση για σύσταση μονοπατιών και Truncation

Όπως περιγράφηκε στις προηγούμενες παραγράφους, αρχικά ο κόμβος sink διαδίδει ένα ενδιαφέρον για ανίχνευση ενός γεγονότος που έχει σαν χαρακτηριστικό ένα μικρό ρυθμό μετάδοσης δεδομένων. Τα γεγονός αυτό καλείται *διερευνητικό γεγονός* και στόχο έχει τη δημιουργία μονοπατιών και την αποκατάσταση σε περίπτωση καταστροφής τους. Όταν ένας κόμβος source ανιχνεύσει ένα στόχο που ταιριάζει στο ενδιαφέρον, στέλνει διερευνητικά γεγονότα προς το sink, μέσω πολλαπλών μονοπατιών. Όταν ο sink λάβει αυτά τα διερευνητικά γεγονότα, ενισχύει κάποιο συγκεκριμένο γειτονικό κόμβο, για να λαμβάνει από εκεί τα πραγματικά δεδομένα. Τα gradients που δημιουργούνται με τον

τρόπο αυτό ονομάζονται *data gradients.*

Α. Σύσταση μονοπατιών χρησιμοποιώντας θετική ενίσχυση Γενικά, οι κανόνες που χρησιμοποιεί το directed diffusion για τη σύσταση των μονοπατιών είναι κανόνες οδηγούμενοι από τα δεδομένα. Ένα παράδειγμα τέτοιου κανόνα παρουσιάζεται στα παρακάτω: Ένας κόμβος ενισχύει ένα γειτονικό του κόμβο όταν λαμβάνει ένα μήνυμα δεδομένων από αυτόν το οποίο δεν το έχει ξαναλάβει. Τότε ο κόμβος στέλνει πίσω το αρχικό μήνυμα ενδιαφέροντος απαιτώντας αυτή τη φορά μικρότερο τιμή για το χαρακτηριστικό interval. Το παράδειγμά μας θα έχει ως εξής:

type = wheeled vehicle
interval = 10ms
rect = [-100, 200, 200, 400]
timestamp = 01 : 22 : 35
expiresAt = 01 : 30 : 40

Όταν ο γειτονικός κόμβος λάβει το μήνυμα αυτό, διαπιστώνει ότι υπάρχει ήδη στην interest cache ένα gradient προς τον κόμβο από τον οποίο έλαβε το μήνυμα και επιπλέον ότι ο ρυθμός δεδομένων που ορίζεται σε αυτό το μήνυμα είναι μεγαλύτερος από τον αποθηκευμένο. Αν αφού ελέγξει όλα τα υπόλοιπα gradients διαπιστώσει ότι ο ρυθμός δεδομένων του νέου μηνύματος είναι μεγαλύτερος από οποιονδήποτε αποθηκευμένο, θα πρέπει να ενισχύσει τουλάχιστον έναν γειτονικό του κόμβο. Η data cache χρησιμοποιείται για το σκοπό αυτό. Και πάλι εφαρμόζονται τοπικοί κανόνες για την επίτευξη της ενίσχυσης, όπως είναι το παράδειγμα του κόμβου που επιλέγει να ενισχύσει τον γειτονικό κόμβο από όπου έλαβε το τελευταίο γεγονός που ταιριάζει στο ενδιαφέρον ή να επιλέξει να στείλει σε όλους τους κόμβους από όπου έλαβε πρόσφατα δεδομένα. Τα παραπάνω δείχνουν ότι ενισχύονται μόνο όσοι κόμβοι στέλνουν διερευνητικά δεδομένα, ενώ δεν είναι απαραίτητο να ενισχυθούν κόμβοι που ήδη στέλνουν με υψηλούς ρυθμούς δεδομένα. Ακολουθώντας λοιπόν την παραπάνω διαδικασία σχηματίζεται ένα μονοπάτι από τον κόμβο source προς τον κόμβο sink, μέσα από το οποίο στέλνονται τα δεδομένα (βλ. σχήμα 2.1(b)).

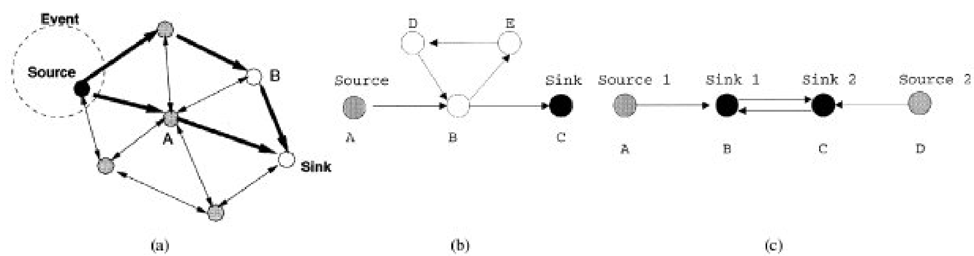
Η παραπάνω διαδικασία καταλήγει στο σχηματισμό του μονοπατιού που εμπειρικά

αποδεικνύεται να έχει τη μικρότερη καθυστέρηση. Έτσι φαίνεται να είναι επιρρεπής σε πιθανές αλλαγές στην ποιότητα του μονοπατιού. Για παράδειγμα, αν από κάποιο μονοπάτι φτάσουν τα δεδομένα πιο γρήγορα από τα υπόλοιπα, ο κόμβος sink θα προσπαθήσει να χρησιμοποιήσει αυτό για να λαμβάνει τα δεδομένα. Για να γίνει όμως κάτι τέτοιο θα πρέπει να σταλεί ένα καινούριο γεγονός. Η πρακτική αυτή μπορεί να απαιτεί τη σπατάλη αρκετών από τους πόρους του συστήματος. Βέβαια υπάρχουν και πιο πολύπλοκες πρακτικές, όπως είναι η επιλογή του κόμβου που στέλνει τα περισσότερα γεγονότα ή αυτού που πάντα στέλνει γεγονότα πιο γρήγορα από τους υπόλοιπους. Οι αποφάσεις αυτές δημιουργούν trade-offs μεταξύ αντιδραστικότητας και σταθερότητας.

Β. Δημιουργία μονοπατιού για πολλαπλούς sink και source κόμβους
Οι παραπάνω κανόνες είναι εύκολο να εφαρμοστούν και σε δίκτυα με πολλαπλούς sink και source κόμβους. Αν για παράδειγμα υπάρχουν πολλοί κόμβοι source, έστω δύο, όπως φαίνεται στο σχήμα 3γ, τα δεδομένα θα φτάνουν στον κόμβο sink μέσω των κόμβων C και D. Έστω ο κόμβος C έχει μόνιμα μικρότερη καθυστέρηση. Τότε η διαδικασία που περιγράφηκε παραπάνω θα επιλέξει να ενισχύσει το μονοπάτι μέσω του C. Αν όμως ο sink λαμβάνει τα δεδομένα του B πιο γρήγορα μέσω του κόμβου D, αλλά τα δεδομένα του A πιο γρήγορα μέσω του C, θα προσπαθήσει να εμισχύσει και τους δύο κόμβους ώστε να λαμβάνει και από τους δύο. Το πρόβλημα σε αυτή την περίπτωση είναι ότι θα λαμβάνει τα δεδομένα δύο sources και από τους δύο κόμβους, κάτι που δεν είναι αποδοτικό ως προς την κατανάλωση ενέργειας. Με λίγη όμως παραπάνω πολυπλοκότητα μπορεί να βρεθεί κάποια καλύτερη λύση.

Η περίπτωση που υπάρχουν παραπάνω από ένας κόμβοι sink, τότε η διαδικασία θα δουλέψει και πάλι σωστά. Έστω για παράδειγμα αρχικά υπάρχει η sink Υ, όπως φαίνεται στο σχήμα 2.1(d) και έχει ενισχύσει ένα υψηλής ποιότητας μονοπάτι προς τον κόμβο source. Αν επιλεχθεί και ο κόμβος X για την παραγωγή ενός πανομοιότυπου ενδιαφέροντος, τότε ο κόμβος αυτός μπορεί να ακολουθήσει τη διαδικασία της ενίσχυσης για να δημιουργήσει το μονοπάτι που φαίνεται και στο σχήμα, αλλά χωρίς να απαιτείται να στείλει μηνύματα διερεύνυσης· μπορεί να το επιτύχει χρησιμοποιώντας απλά τη data cache του.

Γ. Τοπική επιδιόρθωση για κατεστραμμένα μονοπάτια Οι κανόνες της ενίσχυσης μπορούν χρησιμοποιηθούν και από ενδιάμεσους κόμβους με σκοπό την επιδιόρθωση μονοπατιών που έχουν καταστραφεί. Αν κάποιος ενδιάμεσος κόμβος διαπιστώσει ότι κάποιος γειτονικός κόμβος διαπιστώνει ότι η μετάδοση των δεδομένων από τον γείτονά του στο μονοπάτι δεν είναι πια αξιόπιστη μπορεί εφαρμόσει ενίσχυση για τη δημιουργία ενός άλλου μονοπατιού. Όπως περιγράφηκε στις προηγούμενες παραγράφους η διαδικασία της ενίσχυσης σε ένα κόμβο προκαλεί την εκκίνηση της ίδιας διαδικασίας και για τους άλλους κόμβους του μονοπατιού. Με αυτό τον τρόπο θα δημιουργηθεί ξανά το εμπειρικά καλύτερο μονοπάτι, αλλά ίσως να υπάρχει σπατάλη των διαθέσιμων πόρων. Στο σχήμα 2.1(e) φαίνεται μια τέτοια περίπτωση, με τον κόμβο C να προκαλεί τη δημιουργία ενός νέου μονοπατιού.



Σχήμα 2.2: Αρνητική ενίσχυση για καταστροφή μονοπατιών και αφαίρεση βρόχων. a.Πολλαπλά μονοπάτια b.Βρόχος που μπορεί να αφαιρεθεί c.Βρόχος που δε μπορεί να αφαιρεθεί.

Δ. Καταστροφή(Truncation) μονοπατιών χρησιμοποιώντας αρνητική ενίσχυση Σε ένα δίκτυο υπάρχει πιθανότητα να ενισχυθούν παραπάνω από ένα μονοπάτια. Ένα τέτοιο παράδειγμα φαίνεται στο σχήμα 2.2(α), όπου υπάρχουν δύο μονοπάτια, ένα που διέρχεται από τον κόμβο A και το δεύτερο από τον κόμβο B. Αν το μονοπάτι του B παρουσιάζει μόνιμα καλύτερη συμπεριφορά από αυτό του A, θα πρέπει να υπάρξει ένας τρόπος ώστε το δεύτερο να ενισχυθεί αρνητικά.

Ένας soft state μηχανισμός είναι η περιοδική ενίσχυση του B μονοπατιού και παύση των υπόλοιπων gradients. Τελικά, όλα τα gradients κατά μήκος του μονοπατιού A θα υποβιβαστούν σε διερευνητικά gradients. Μια άλλη τεχνική είναι η σαφής υποβάθμιση του μονοπατιού A, στέλνοντας ένα μήνυμα αρνητικής ενίσχυσης στον κόμβο A, όπου

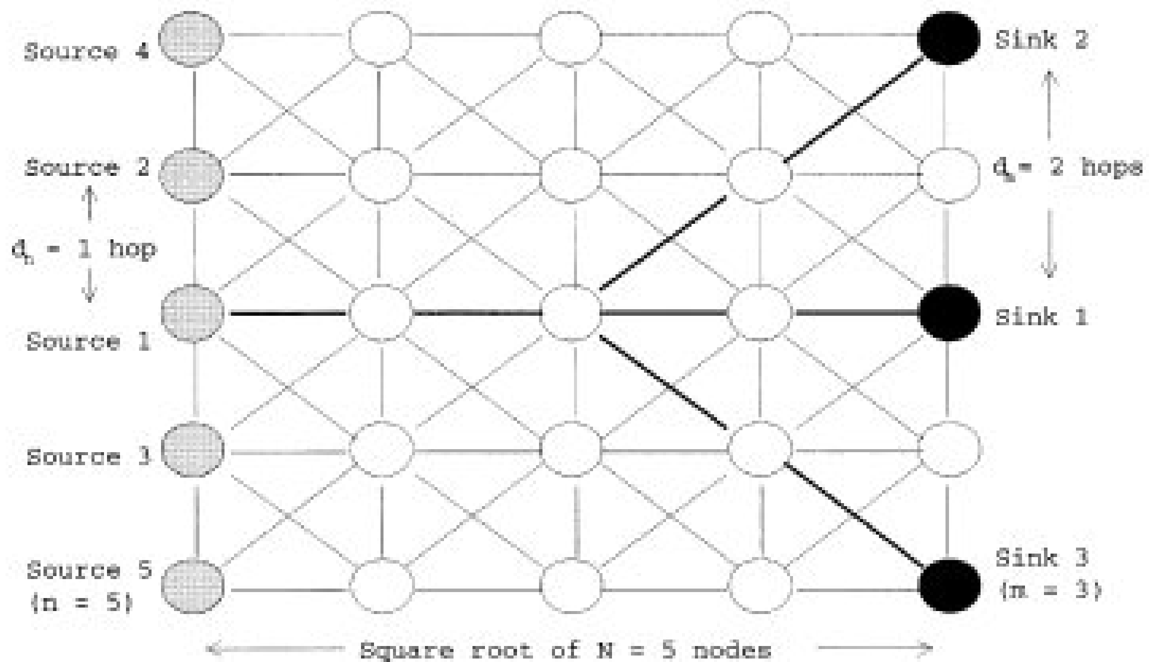
αρνητική ενίσχυση σημαίνει μήνυμα ενδιαφέροντος που ορίζει το μικρότερο ρυθμό μετάδοσης δεδομένων. Όταν ο κόμβος A λάβει αυτό το μήνυμα θα υποβαθμίσει το gradient προς τον sink. Επιπλέον, αν όλα του τα gradients καταλήξουν να είναι διερευνητικά, ο A θα στείλει αρνητικές ενισχύσεις σε όλους τους κόμβους από τους οποίους λαμβάνει δεδομένα. Οι διαδοχικές αντιδράσεις ενίσχυσησ(αρνητικής αυτή τη φορά) εγγυώνται πως το μονοπάτι θα υποβαθμιστεί σε μικρό χρονικό διάστημα, σε βάρος βέβαια της αυξημένης χρήσης των πόρων του συστήματος.

Η απόφαση ενός κόμβου για το ποιον γειτονικό κόμβο θα πρέπει να εμισχύσει αρνητικά βασίζεται στον παρακάτω τοπικό κανόνα: θα ενισχυθεί αρνητικά ο κόμβος από τον οποίο δεν έχει λάβει καθόλου γεγονότα για ένα συγκεκριμένο χρονικό διάστημα. Μια παραλλαγή είναι η αρνητική ενίσχυση του κόμβου από τον οποίο έχει λάβει τα λιγότερα γεγονότα σε ένα συγκεκριμένο χρονικό διάστημα.

Δ. Αφαίρεση βρόχων χρησιμοποιώντας αρνητική ενίσχυση Η τεχνική της αρνητικής ενίσχυσης μπορεί να χρησιμοποιηθεί και για την αφαίρεση των βρόχων, αφού τα μηνύματα που διαδίδονται από μονοπάτια που περιέχουν κύκλους ποτέ δεν θα είναι αυτά που θα φτάνουν πρώτα. Παρόλο που ένα μήνυμα που περνάει από ένα κύκλο θα καταπνιγεί χρησιμοποιώντας την cache των μηνυμάτων, η καταστροφή των μονοπατιών χρησιμοποιώντας αρνητική ενίσχυση προσφέρει κάποιο πλεονέκτημα στη διατήρηση της ενέργειας. Κάποιοι κύκλοι δε θα πρέπει να αφαιρώνται, όπως συμβαίνει με τον κύκλο του σχήματος 2.2(c).

2.1.5 Αναλυτική εκτίμηση

Έστω ότι η μορφή του δικτύου ακολουθεί το μοντέλο ενός grid των N κόμβων (βλ. σχήμα 2.3). Στο σχήμα αυτό φαίνονται οι συνδέσεις μεταξύ των γειτονικών κόμβων. Κάθε ενδιάμεσος κόμβος έχει ακριβώς οχτώ γειτονικούς κόμβους. Οι n sources βρίσκονται στην αριστερή πλευρά του grid ενώ όλοι οι m sinks κόμβοι στη δεξιά πλευρά. Ο πρώτος κόμβος source βρίσκεται στο κέντρο της αριστερής πλευράς. Καθένας από τους i υπόλοιπους sources βρίσκονται $d_n\lfloor i/2 \rfloor$ βήματα πάνω (αν το i είναι άρτιος) ή κάτω (αν το i είναι περιττός) από το πρώτο source. Το ίδιο ισχύει και για τους sinks



Σχήμα 2.3: Παράδειγμα δικτύου μορφής τετραγωνικού grid.

μόνο που η αποστάσεις είναι d_m βήματα. Δεδομένου ότι οι soutces και οι sinks είναι τοποθετημένοι μόνο κάθετα, η $\sqrt{N}$ δε θα έχει τιμή μικρότερη από το $\max(\text{nd}_n,\text{md}_m)$.

Για να απλοποιηθεί η ανάλυση, υποθέτουμε ότι το δένδρο όπου εφαρμόζονται οι τοπικοί αλγόριθμοι του directed diffusion είναι η ένωση των συντομότερων μονοπατιών με ρίζες κάθε ένα από τα sources. Αυτή η υπόθεση είναι προσεγγιστικά έγκυρη όταν το δίκτυο λειτουργεί με φορτία χαμηλών μεγεθών. Επιπλέον, η διάδοση επιλέγει το συντομότερο μονοπάτι σύμφωνα με την παρακάτω κανόνα: Από ένα sink προς ένα source, μια διαγώνια ακμή είναι πάντα η επόμενη μετάβαση αν οδηγεί στο συντομότερο μονοπάτι· διαφορετικά επιλέγεται μια οριζόντια ακμή.

Αν όλα τα sources στέλνουν πανομοιότυπες εκτιμήσεις για τη θέση του στόχου, τότε, δεδομένου του ότι το diffusion μπορεί να αποσιωποιήσει τα διπλά μηνύματα σε επίπεδο εφαρμογής, το κόστος της διάδοσης των δεδομένων είναι το διπλάσιο του αριθμού των συνδέσεων της ένωσης όλων των δένδρων συντομότερων διαδρομών με ρίζες τα sources. Επομένως:

$C_d = C(UT_1 \rightarrow n) = C(T_1) +$

$$\sum_{j=2}^{n}\{H(T_j - UT_{1\rightarrow(j-1)}) + D(T_j - UT_{1\rightarrow(j-1)})\}$$

όπου

$H(T_j - UT_{1\rightarrow(j-1)}) = H(T_j)$
$D(T_j - UT_{1\rightarrow(j-1)}) =$

$$2\{\lceil\frac{m+(jmod2)}{2}\rceil d_n + \sum_{l=1}^{min(\lfloor\lfloor j/2\rfloor d_n/d_m\rfloor,\lfloor m-(jmod2)/2)\rfloor)} min(d_n, d_n\lfloor\frac{j}{2}\rfloor - ld_m)\}$$

C_d είναι $O(n\sqrt{N})$ για $m \ll \sqrt{N}$.

2.2 A Probabilistic Forwarding Protocol for Efficient Data Propagation in Sensor Networks

Ένα σημαντικό πρόβλημα για τα WSNs είναι το πρόβλημα της 'τοπικής ανίχνευσης και προώθησης ', η ανίχνευση δηλαδή τοπικά ενός γεγονότος και η αποδοτική ως προς την ενέργεια και τον χρόνο προώθηση των δεδομένων που γνωστοποιούν το γεγονός σε κάποιο κέντρο ελέγχου, το sink.

Το PFR (Probabilistic Forwarding Protocol), το οποίο προτείνεται στο [7] από τους Ι. Χατζηγιαννάκη, Τ. Δημητρίου, Σ. Νικολετσέα και Π. Σπυράκη, προσπαθεί να ελαχιστοποιήσει την κατανάλωση ενέργειας για το πρόβλημα αυτό σε ένα δύο διαστάσεων lattice δίκτυο αισθητήρων, χρησιμοποιώντας πιθανοτική επιλογή συγκεκριμένων μονοπατιών για την προώθηση των δεδομένων προς το sink.

Η βασική ιδέα πίσω από τη δημιουργία του PFR είναι η αποφυγή της πλημμύρας για τη μετάδοση των δεδομένων, επιλέγοντας με ένα πιθανοτικό τρόπο τη διάδοση από κόμβο σε κόμβο έτσι ώστε να επιτευχθεί ένα μονοπάτι που πλησιάζει το βέλτιστο μονοπάτι μετάδοσης μεταξύ του κόμβου που ανιχνεύει το γεγονός και του sink. Θα πρέπει να

επισημανθεί ότι το πρωτόκολλο χρησιμοποιεί μόνο τοπικές πληροφορίες και δεν προϋποθέτει γνώση για όλο το δίκτυο, ενώ δε χρησιμοποιεί μηνύματα ελέγχου για να αποκτήσει αυτή τη γνώση. Επιπλέον, εξαιτίας της πιθανοτικής του φύσης, το πρωτόκολλο χρησιμοποιεί ένα μηχανισμό πλημμύρας για τη δημιουργία ενός αρκετά μεγάλου μετώπου αισθητήρων, οι οποίοι επεξεργάζονται τα δεδομένα προς μετάδοση, έτσι ώστε να αποφευχθεί η πρόωρη αποτυχία του αλγορίθμου. Μόλις δημιουργηθεί το μέτωπο αυτό, ξεκινάει η πιθανοτική προώθηση.

2.2.1 Το Μοντέλο

Όπως αναφέρθηκε και παραπάνω το μοντέλο του δικτύου που υιοθετείται είναι το διδιάστατο lattice δίκτυο, με n x n κόμβους. Κάθε μια από τις συκευές έχει μέγιστη ακτίνα μετάδοσης R. Οι συσκευές τοποθετούνται οριζόντια και κάθετα με απόσταση δ μεταξύ τους. Το δ εξαρτάται από το R και επιπλέον δίνει την πυκνότητα του δικτύου. Στη συγκεκριμένη περίπτωση επιλέγεται $\mathrm{R} = \delta\sqrt{2}$. Έτσι κάθε εσωτερική συσκευή έχει οχτώ γειτονικούς κόμβους.

Ένα και μοναδικό σημείο του δικτύου αποτελεί τον κόμβο sink S, που αντιπροσωπεύει το κέντρο ελέγχου. Στη βασική υπόθεση ο κόμβος αυτός είναι ακίνητος, σε κάποια παραλλαγή όμως θα μπορούσε να κινείται ώστε να συλλέγει πληροφορίες που δεν κατάφεραν να φτάσουν σε αυτόν.

Η μόνη γνώση που υπάρχει σε κάθε κόμβο είναι αυτή της θέσης του μέσα στο δίκτυο καθώς επίσης και η θέση του sink, δεν υπάρχει δηλαδή γνώση για όλη την κατάσταση του δικτύου. Ας σημειωθεί ότι ακόμα και αυτή η υπόθεση μπορεί να χαλαρώσει, για παράδειγμα ίσως να ήταν αρκετή απλώς η πληροφορία για τη γενική κατεύθυνση του sink και ο κόμβος να μπορεί να την αναγνωρίσει από τα μηνύματα που λαμβάνει.

2.2.2 Το Πρόβλημα

Ας υποθέσουμε ότι κάποιος κόμβος p ανιχνεύει ένα γεγονός E. Τότε το πρόβλημα της διάδοσης είναι το εξής:

'Πώς μπορεί ο κόμβος p, μέσω συνεργασίας με τους υπόλοιπους κόμβους, να διαδόσει αποδοτικά την πληροφορία info(E), γνωστοποιώντας την πραγματοποίηση του γεγονότος

E, στον κόμβο sink S;'

Η ελαχιστοποίηση της κατανάλωσης ενέγειας γίνεται αποφεύγοντας την πλημμύρα και ελαχιστοποιώντας των αριθμό των μεταβάσεων κατά τη διαδικασία της διάδοσης. Μια άλλη πλευρά του προβλήματος είναι το πόσο κοντά στον κόμβο sink φτάνει η διάδοση των δεδομένων στην περίπτωση που δε φτάνουν όλα τα δεδομένα στον sink.

2.2.3 Το Πρωτόκολλο

Το πρωτόκολλο εξελίσσεται σε δύο φάσεις:

Η φάση της δημιουργίας του 'μετώπου': Όπως αναφέρθηκε και παραπάνω, εξαιτίας της πιθανοτικής φύσης του πρωτοκόλλου και με στόχο την επιβίωση της διαδικασίας διάδοσης των μηνυμάτων, αρχικά κατασκευάζεται ένα μέτωπο που αποτελείται από έναν αρκετά ικανοποιητικό αριθμό κόμων αισθητήρων, με τη χρήση ενός μηχανισμού πλημμύρας. Κατά τη φάση αυτή λοιπόν, κάθε κόμβος που λαμβάνει δεδομένα τα προωθεί ντετερμινιστικά προς τον κόμβο sink, στέλνοντας σε κάθε βήμα την πληροφορία σε όλους τους γειτονικούς του κόμβους. Επιπλέον (για την περίπτωση του παραπάνω μοντέλου), αν επιλεχθεί η κατάλληλη ακτίνα μετάδοσης, τότε ο κάθε κόμβος θα στέλνει σε τρεις μόνο από τους γειτονικούς του κόμβους, αφού μόνο αυτοί θα είναι πιο κοντά στον κόμβο sink. Βέβαια, αν δεν είναι γνωστή η κατεύθυνση προς την οποία βρίσκεται ο sink, τότε οι κόμβοι αναγκαστικά προωθούν σε όλους τους γείτονές τους τα δεδομένα.

Η φάση της πιθανοτικής προώθησης: Σε αυτή τη φάση κάθε κόμβος επιλέγει πιθανοτικά τη μετάδοση των δεδομένων, έτσι ώστε να επιτευχθεί ένα μονοπάτι από τον κόμβο που ανιχνεύει το γεγονός προς την πηγή που να πλησιάζει το βέλτιστο. Ο κάθε κόμβος, δηλαδή, στέλνει με μια κατάλληλα επιλεγμένη πιθανότητα p ή δεν στέλνει με πιθανότητα 1-p.

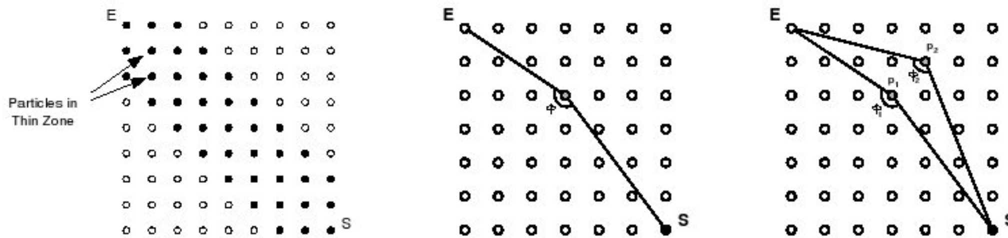
Η p υπολογίζεται ως εξής: $\mathrm{P}_{fw} = \frac{\phi}{\pi}$,

όπου φ είναι η γωνία που σχηματίζεται από τη γραμμή που ενώνει τον κόμβο που ανιχνεύει το γεγονός και τον κόμβο που ετοιμάζεται να στείλει τα δεδομένα, και τη γραμμή που ενώνει το δεύτερο αυτό κόμβο με τον κόμβο sink. Επομένως, όσο μεγαλύτερη είναι

η γωνία φ, τόσο πιο κοντά στο βέλτιστο είναι το μονοπάτι που σχηματίζεται, με μέγιστη βέβαια τιμή την $\varphi = \pi$, κάτι που σημαίνει ότι ο κόμβος βρίσκεται πάνω στο βέλτιστο μονοπάτι.

Επιπλέον, αν για δύο κόμβους p_1, p_2 ισχύει $\varphi_1 > \varphi_2$, τότε η p_1 βρίσκεται πιο κοντά στο βέλτιστο μονοπάτι και επομένως $P_{fw}(p_1) > P_{fw}(p_2)$.

Το παρακάτω σχήμα παρουσιάζει γραφικά όλα τα παραπάνω:



Σχήμα 2.4: **a.**Η λεπτή ζώνη κόμβων γύρω από τη γραμμή που ενώνει τον κόμβο που ανιχνεύει το γεγονός E και τον κόμβο sink. **b.**Η γωνία φ **c.**Η γωνία φ και η εγγύτητα στη βέλτιστη γραμμή.

Κεφάλαιο 3

Κυματικοί Αλγόριθμοι

Κατά το σχεδιασμό κατανεμημένων αλγορίθμων για ποικίλες εφαρμογές εμφανίζονται συχνά κάποια στοιχειώδη προβλήματα ως υπολειτουργίες. Σε αυτά τα προβλήματα περιλαμβάνονται η διάδοση πληροφορίας, η επίτευξη συγχρονισμού των διεργασιών, ο έλεγχος της εκτέλεσης ορισμένων γεγονότων σε κάθε διεργασία ή, τέλος, ο υπολογισμός της τιμής μιας συνάρτησης, από την οποία η διεργασία διατηρεί μέρος της εισόδου. Οι εργασίες αυτές υλοποιούνται με το πέρασμα μηνυμάτων σύμφωνα με κάποιο προδιαγεγραμμένο σχήμα, που δεν εξαρτάται από την τοπολογία και που εξασφαλίζει τη συμμετοχή όλων των διεργασιών. Πραγματικά, τα παραπάνω προβλήματα είναι τόσο θεμελιώδη, ώστε πιο πολύπλοκα προβλήματα, όπως η εκλογή αρχηγού, η ανίχνευση τερματισμού, ο αμοιβαίος αποκλεισμός, μπορούν να επιλυθούν μόνο όταν η επικοινωνία μεταξύ των διεργασιών συμβαίνει μέσω αυτού του σχήματος περάσματος μηνυμάτων.

Στη βιβλιογραφία οι αλγόριθμοι διάδοσης της πληροφορίας με πέρασμα μηνυμάτων συναντώνται και σαν κυματικοί(wave) αλγόριθμοι. Η σπουδαιότητά τους δικαιολογεί την ξεχωριστή μεταχείρισή τους, απομονώνοντάς τους από την εφαρμογή όπου θα ενσωματωθούν, για δύο κυρίως λόγους. Καταρχάς διευκολύνει την αργότερη διαχείριση πιο εξελιγμένων αλγορίθμων, αφού οι υπορουτίνες τους θα έχουν ήδη μελετηθεί. Δεύτερον, κάποια συγκεκριμένα προβλήματα κατανεμημένου υπολογισμού μπορούν να επιλυθούν με generic υλοποιήσεις που παράγουν ένα συγκεκριμένο αλγόριθμο όταν παραμετροποιηθούν με έναν συγκεκριμένο κυματικό αλγόριθμο. Η ίδια κατασκευή μπορεί

να χρησιμοποιηθεί για να δώσει αλγορίθμους σε δίκτυα διαφορετικών τοπολογιών ή σε διαφορετικές υποθέσεις για την αρχικοποίηση των διεργασιών.

3.1 Ορισμός και χρήση των Κυματικών Αλγορίθμων

3.1.1 Αρχικές υποθέσεις

Στο κεφάλαιο αυτό υποθέτουμε ότι δε συμβάνουν αλλαγές στην τοπολογία του δικτύου, ότι δηλαδή είναι σταθερή, ότι το δίκτυο είναι μη διευθυνόμενο, οπότε σε κάθε κανάλι μπορούν να μεταφέρονται μηνύματα και προς τις δύο κατευθύνσεις, και τέλος, το δίκτυο είναι συνεκτικό, δηλαδή υπάρχει μονοπάτι μεταξύ δύο οποιωνδήποτε κόμβων. Το σύνολο των κόμβων-διεργασιών του δικτύου ονομάζεται **P** και το σύνολο των καναλιών **E**. Επίσης, υποθέτουμε ότι το σύστημα είναι ασύγχρονο και οι έννοιες του καθολικού χρόνου και του ρολογιού πραγματικού χρόνου δεν υφίστανται, αν και οι αλγόριθμοι που θα παρουσιαστούν παρακάτω μπορούν να χρησιμοποιηθούν και με σύγχρονα συστήμα, πιθανώς με κάποιες τροποποιήσεις για την αποφυγή αδιεξόδων.

3.1.2 Ορισμοί

Ένα κατανεμημένο σύστημα αποτελείται από ένα σύνολο διεργασιών και ένα σύστημα επικοινωνίας. Οι διεργασίες αλληλεπιδρούν με το επικοινωνιακό σύστημα με τα εσωτερικά γεγονότα (*internal events*), καθώς επίσης στέλνοντας (*send*) και λαμβάνοντασ(*receive*) γεγονότα. Ένας κατανεμημένος αλγόριθμος είναι ένα σύνολο τοπικών αλγορίθμων, έναν για κάθε διεργασία. Εξαιτίας του μη-ντετερμινισμού μέσα στις διεργασίες και στο σύστημα επικοινωνίας, οι κατανεμημένοι αλγόριθμοι επιτρέπουν ένα μεγάλο σύνολο πιθανών υπολογισμών. Ένας υπολογισμός είναι ένα σύνολο γεγονότων, που είναι μερικώς ταξινομημένα σύμφωνα με τη σχέση της αιτιατής προτεραιότητας $\preceq$. Ο αριθμός των γεγονότων του υπολογισμού C συμβολίζεται με $|C|$ και το υποσύνολο των γεγονότων που συμβαίνουν στη διαδικασία p ονομάζεται C_p. Επίσης υποτίθεται ότι

υπάρχει ένας ειδικός τύπος εωτερικόυ γεγονότος που καλείται *decide* γεγονός. Στους αλγορίθμους που θα παρουσιαστούν παρακάτω, το γεγονός αυτό υποδηλώνεται με την έκφραση *decide*. Ένας κυματικός αλγόριθμος ανταλλάσσει έναν πεπερασμένο αριθμό μηνυμάτων και έπειτα λαμβάνει μια απόφαση που εξαρτάται από κάποιο γεγονός σε κάθε διεργασία.

Ορισμός 1. *Ένας κυματικός αλγόριθμος είναι ένας κατανεμημένος αλγόριθμος που ικανοποιεί τις εξής τρεις απαιτήσεις:*

1. **Τερματισμός** – *Κάθε υπολογισμός είναι πεπερασμένος:*

$$\forall C : |C| < \infty$$

2. **Απόφαση** – *Κάθε υπολογισμός περιέχει τουλάχιστον ένα decide γεγονός:*

$$\forall C : \exists e \in C: \text{το e είναι ένα decide γεγονός}$$

3. **Εξάρτηση** – *Σε κάθε υπολογισμό κάθε decide γεγονός προηγείται αιτιολογικά από ένα γεγονός σε κάθε διεργασία:*

   $\forall C : \exists e \in C:$ (το e είναι ένα decide γεγονός $\Rightarrow \forall q \in P\ \exists f \in C_q : f \preceq e$)

Ένας υπολογισμός ενός αλγορίθμου ονομάζεται *κύμα* (wave). Επίσης, γίνεται ένας επιπλέον διαχωρισμός κατά τον υπολογισμό, σε *αρχικοποιητές* (initiators) και *μη αρχικοποιητές*(non-initiators). Ένας αρχικοποιητής είναι αυτός που ξεκινάει την εκτέλεση ενός τοπικού αλγορίθμου τυχαία. Ένας μη αρχικοποιητής συμμετέχει στον αλγόριθμο μόνο όταν φτάνει ένα μήνυμα και προκαλεί την έναρξη της εκτέλεσης του αλγορίθμου. Το πρώτο γεγονός ενός αρχικοποιητή είναι ένα εσωτερικό γεγονός ή ένα γεγονός αποστολής, ενώ το πρώτο γεγονός ενός αρχικοποιητή είναι ένα γεγονός λήψης.
Μια λίστα χαρακτηριστικών που εμφανίζονται στους κυματικούς αλγορίθμους και που τους κάνει να ξεχωρίζουν μεταξύ τους φαίνεται παρακάτω:

1. **Συγκεντρωτισμός (Centralization)** – Ένας αλγόριθμος καλείται συγκεντρωτικός όταν πρέπει να υπάρχει ακριβώς ένας αρχικοποιητής σε κάθε υπολογισμό και αποκεντρωτικός αν μπορεί να ξεκινήσει τυχαία από ένα αυθαίρετο υποσύνολο των διεργασιών. Οι συγκεντρωτικοί αλγόριθμοι καλούνται και αλγόριθμοι μοναδικής πηγής, ενώ οι αποκεντρωτικοί καλούνται αλγόριθμοι πολλαπλών πηγών. Ο συγκντρωτισμός επηρεάζει σημαντικά την πολυπλοκότητα των κυματικών αλγορίθμων.

2. **Τοπολογία** – Ένας αλγόριθμος μπορεί μα σχεδιαστεί για μια συγκεκριμένη τοπολογία, για παράδειγμα ένα δακτύλιο, ένα δέντρο, μια κλίκα κτλ.

3. **Αρχική γνώση** – Ένας αλγόριθμος μπορεί να υποθέσει τη διαθεσιμότητα διαφόρων τύπων αρχικής γνώσης στις διεργασίες, όπως για παράδειγμα:

   (α′) *Την ταυτότητα της διεργασίας.* Κάθε διεργασία γνωρίζει τη μοναδική ταυτότητα που την χαρακτηρίζει.

   (β′) *Τις ταυτότητες των γειτονικών κόμβων*

   (γ′) *Αίσθηση προσανατολισμού.*

   (δ′) *Αριθμό αποφάσεων.* Σε κάθε διεργασία λαμβάνεται τουλάχιστον μία απόφαση. Ο αριθμός των διεργασιών που αποφασίζουν μπορεί να ποικίλει· από μία μέχρι και όλες οι διεργασίες μπορούν να προκαλέσουν ένα decide γεγονός.

   (ε′) *Πολυπλοκότητα.* Αφορά τον αριθμό μηνυμάτων ή των bits που ανταλλάσσονται και τον χρόνο που απαιτεί κάθε υπολογισμός για να ολοκληρωθεί.

Τα μηνύματα που χρησιμοποιούνται από τους αλγορίθμους συνήθως είναι κενά μηνύματα, μιας και ο ρόλος τους δεν είναι η μεταφορά πληροφορίας αλλά η αιτιατότητα. Ίσως, όμως, να χρειάζεται η προσθήκη κάποιων bits για το διαχωρισμό των μηνυμάτων, όταν οι αλγόριθμοι χρησιμοποιούν διαφορετικού τύπου μηνύματα. Βέβαια, όταν ένας κυματικός αλγόριθμος είναι εφαρμοσμένος υπάρχουν περισσότερες πληροφορίες που περιλαμβάνονται στα μηνύματα, όταν υπάρχουν πολλά κύματα που διαδίδονται και πρέπει να είναι γνωστό σε ποιο κύμα ανήκει κάθε μήνυμα.

Ένα υποσύνολο των κυματικών αλγορίθμων που σχηματίζονται από συγκεντρωτικούς κυματικούς αλγορίθμους έχουν τις εξής δύ ο ιδιότητες: Πρώτον, αρχικοποιητής είναι η μόνη διεργασία που αποφασίζει και δεύτερον, όλα τα γογονότα είναι ταξινομημένα με βάση τη σειρά *αιτιατότητας*. Το υποσύνολο αυτό των αλγορίθμων καλούνται *αλγόριθμοι διάσχισης* (traversal algorithms).

3.1.3 Στοιχειώδη αποτελέσματα για τους Κυματικούς Αλγορίθμους

Στην παράγραφο αυτή παρουσιάζονται κάποια στοιχειώδη λήμματα που περιγράφουν τη δομή των κυματικών αλγορίθμων, καθώς επίσης και κάποια κατώτατα όρια για την πολυπλοκότητα μηνυμάτων.

Καταρχάς, σε έναν υπολογισμό ένα γεγονός σε έναν αρχικοποιητή προηγείται οποιουδήποτε γεγονότος.

Λήμμα 1. *Για κάθε γεγονός $e \in C$ υπάρχει ένας αρχικοποιητής p και ένα γεγονός f στο C_p έτσι ώστε $f \preceq e$.*

Απόδειξη. Επιλέγουμε το γεγονός f έτσι ώστε να είναι το ελάχιστο (το παλιότερο) γεγονός στην ιστορία του e, δηλαδή να ισχύει ότι $f \preceq e$ και δεν υπάρχει άλλο γεγονός $f' \prec f$. Ένα τέτοιο γεγονός υπάρχει, αφού η ιστορία του e είναι πεπερασμένη. Αρκεί, λοιπόν, να δείξουμε ότι η διεργασία p όπου συμβαίνει το f, είναι αρχικοποιητής. Εδώ πρέπει να σημειώσουμε ότι το f είναι το πρώτο γεγονός που συμβαίνει στην p· αν δεν ίσχυε κάτι τέτοιο θα υπήρχε κάποιο άλλο ελάχιστο γεγονός, το οποίο βεβαίως θα προηγούνταν του f. Αν υποθέσουμε ότι η p είναι μη αρχικοποιητής, τότε το πρώτο γεγονός που θα συνέβαινε θα ήταν η λήψη ενός μηνύματος, της οποίας όμως θα προηγούνταν το γεγονός της αποστολής. Κάτι τέτοιο δεν μπορεί να ισχύει, αφού αντιβαίνει στο ότι το γεγονός f είναι ελάχιστο. Επομένως, η p είναι αρχικοποιητής.

Λήμμα 2. *Έστω C ένα κύμα με έναν αρχικοποιητή p και για κάθε μη αρχικοποιητή q ας είναι father(q) ο γείτονας του q από τον οποίο ο q έλαβε ένα μήνυμα κατά το πρώτο του γεγονός. Τότε ο γράφος $T = (P, E_T)$, με $E_T = \{qr\colon q \neq p \wedge r = father_q\}$, είναι*

ένα επικαλυπτικό δέντρο με κατεύθυνση προς το p.

Απόδειξη. Εφόσον ο αριθμός των κόμβων του T υπερβαίνει κατά ένα τον αριθμό των ακμών, αρκεί να δείξουμε ότι το T δεν περιέχει κύκλο. Το παραπάνω ισχύει αφού το e_q είναι το πρώτο γεγονός του q, το qr $\in E_T$ δείχνει ότι $e_r \preceq e_q$, ενώ ο τελεστής $\preceq$ επιτελεί μερική ταξινόμηση.

Λήμμα 3. *Έστω C ένα κύμα και $d_p \in C$ ένα decide γεγονός στη διεργασία p. Τότε*

$$\forall q \neq p : \exists f \in C_q : (f \preceq d_p \wedge f \text{ είναι ένα γεγονός αποστολής})$$

Αυτό πρακτικά σημαίνει ότι στην ιδιότητα της εξάρτησης στον ορισμό των κυματικών αλγορίθμων, όπου καθενός decide γεγονότος προηγείται ένα γεγονός f που προκαλεί την απόφαση, μπορεί να επιλεγεί το f να είναι ένα γεγονός αποστολής για όλες τις διεργασίες q εκτός από τη διεργασία όπου το decide γεγονός συμβαίνει.

Απόδειξη. Εφόσον το C είναι κύμα υπάρχει $f \in C_q$, που προηγείται του d_p αιτιολογικά. Επιλέγουμε το f να είναι το τελευταίο γεγονός του C_q που προηγείται του d_p. Για να δείξουμε ότι το f είναι ένα γεγονός αποστολής, πρέπει να προσέξουμε ότι ο ορισμός της αιτιολογικότητας υποδηλώνει το γεγονός ότι υπάρχει μια σειρά (αλυσίδα αιτιοτήτων) f $= e_0, e_1, ..., e_k = d_p$, έτσι ώστε για κάθε $i < k$, e_i και e_{i+1} είναι είτε αλλεπάλληλα γεγονότα της ίδιας διεργασίας ή ένα send - receive ζευγάρι. Εφόσον το f ειναι το τελευταίο γεγονός στην q που προηγείται του d_p, το e_1 συμβαίνει σε μια διεργασία διαφορετική από την q και επομένως το f είναι ένα γεγονός αποστολής.

Το λήμμα 3 δίνει αμέσως και το κατώτατο όριο για τον αριθμό των μηνυμάτων που ανταλλάσσονται σε ένα κύμα. Το όριο αυτό είναι τα N-1 μηνύματα. Για την περίπτωση όπου υπάρχει ένας μοναδικός αρχικοποιητής και το decide γεγονός συμβαίνει σε αυτόν (όπως για παράδειγμα στους αλγόριθμους διάσχισης), τότε το κατώτατο όριο είναι τα N μηνύματα. Για αλγορίθμους που εφαρμόζονται σε τυχαία δίκτυα, το κατώτατο όριο των μηνυμάτων είναι τουλάχιστον $|E|$. Τα παραπάνω ορίζονται και αποδεικνύονται με τα επόμενα δύο θεωρήματα.

Θεώρημα 1. *Αν C είναι ένα κύμα με έναν αρχικοποιητή p, τέτοιον ώστε ένα decide γεγονός d_p συμβαίνει στον p, τότε τουλάχιστον N μηνύματα ανταλλάσσονται στο C.*

Απόδειξη. Από το λήμμα 1 έχουμε ότι ένα γεγονός στη διεργασία p προηγείται κάθε γεγονότος του C. Επιπλέον, αν χρησιμοποιήσουμε μία αλυσίδα διάταξη αιτιοτήτων όπως αυτή του λήμματος 3, εύκολα μπορούμε να δούμε ότι τουλάχιστον ένα γεγονός αποστολής συμβαίνει στην p. Επίσης, από το λήμμα 3 προκύπτει ότι και σε όλες τις άλλες διεργασίες συμβαίνει ένα γεγονός αποστολής. Επομένως, καταλήγουμε στον αριθμό των N μηνυμάτων αποστολής.

Θεώρημα 2. *Έστω A ένας κυματικός αλγόριθμος για τυχαία δίκτυα χωρίς κάποια αρχική γνώση για τις ταυτότητες των γειτονικών κόμβων. Τότε ο A ανταλλάσσει τουλάχιστον $|E|$ μηνύματα σε κάθε υπολογισμό.*

Απόδειξη. Ας υποθέσουμε ότι ο A έχει έναν υπολογισμό C, ο οποίος ανταλλάσσει λιγότερα από $|E|$ μηνύματα και έστω ότι υπάρχει ένα κανάλι xy, όπου δε μεταφέρονται καθόλου μηνύματα. Δημιουργούμε ένα νέο δίκτυο G' προσθέτοντας έναν ακόμη κόμβο, τον z, ανάμεσα στους x και y. Οι κόμβοι του δικτύου δεν έχουν αρχική γνώση για τις ταυτότητες των γειτονικών τους κόμβων· έτσι η αρχική κατάσταση των κόμβων στα δύο δίκτυα είναι ίδια. Επομένως, όλα τα γεγονότα του C μπορούν να εφαρμοστούν στο G' με την ίδια σειρά. Τώρα, όμως, το decide γεγονός στον κόμβο z δεν έπεται κάποιου άλλου γεγονότος.

3.1.4 Διάδοση πληροφορίας με ανάδραση

Σε κάποιες εφαρμογές απαιτείται κάποια συγκεκριμένη πληροφορία να μεταδοθεί σε όλες τις διεργασίες και κάποιες από αυτές πρέπει να λάβουν μια ειδοποίηση για την ολοκλήρωση της παραπάνω μετάδοσης. Οι κυματικοί αλγόριθμοι είναι οι πιο κατάλληλοι για τις περιπτώσεις αυτές. Στην παράγραφο αυτή μελετώνται οι αλγόριθμοι διάδοσης πληροφορίας με ανάδραση (propagation of information with feedback ή PIF). Καταρχάς, σχηματίζεται ένα υποσύνολο των διεργασιών που έχουν το ίδιο μήνυμα M και το οποίο πρέπει να αποσταλεί και να γίνει δεκτό από όλες τις διεργασίες. Κάποιες διεργασίες πρέπει να ειδοποιηθούν για τον τερματισμό της διαδικασίας εκπομπής, όταν και μόνο όταν όλες οι διεργασίες έχουν δεχτεί το μήνυμα M. Θα πρέπει, δηλαδή, να προκληθεί ένα ειδικό γεγονός ‘ειδοποίησης ’, το οποίο θεωρείται ένα decide γεγονός. Ο αλγόριθμος θα πρέπει να χρησιμοποιεί πεπερασμένο αριθμό μηνυμάτων.

Θεώρημα 3. *Κάθε PIF αλγόριθμος είναι κυματικός αλγόριθμος.*

Απόδειξη. Έστω P ένας PIF αλγόριθμος. Κάθε υπολογισμός του P θα πρέπει να είναι πεπερασμένου αριθμού βημάτων και θα πρέπει να συμβαίνει ένα γεγονός ειδοποίησης (decide). Αν σε κάποιον υπολογισμό του P συμβεί ένα γεγονός ειδοποίησης d_p το οποίο δεν έπεται κάποιου άλλου γεγονότος σε μια διεργασία q, τότε υπάρχει μια εκτέλεση του P όπου η ειδοποίηση συμβαίνει πριν η q λάβει κάποιο μήνυμα, κάτι που αντιβαίνει στις απαιτήσεις.

Θεώρημα 4. *Κάθε κυματικός αλγόριθμος μπορεί να χρησιμοποιηθεί σαν PIF αλγόριθμος.*

Απόδειξη. Έστω A είναι ένας κυματικός αλγόριθμος. Για να χρησιμοποιηθεί ο A σαν PIF αλγόριθμος θα πρέπει οι διεργασίες που αρχικά έχουν την πληροφορία M να είναι αυτές που θα ξεκινήσουν την εκτέλεση του A. Η πληροφορία M επισυνάπτεται σε κάθε μήνυμα του A. Αυτό είναι δυνατό, αφού από κατασκευής οι αρχικοποιητές του A γνωρίζουν το M και οι μη αρχικοποιητές δεν στέλνουν κανένα μήνυμα πριν λάβουν κάποιο, έτσι ώστε να έχουν την πληροφορία M. Ένα decide γεγονός σε ένα κύμα έπεται ενός άλλου γεγονότος σε κάθε διεργασία. Έτσι, όταν το decide γεγονός συμβεί, κάθε διεργασία θα γνωρίζει το M και αυτό θεωρείται η απαιτούμενη, από τον PIF αλγόριθμο, ειδοποίηση.

Ο PIF αλγόριθμος που δημιουργήθηκε έχει την ίδια πολυπλοκότητα μηνυμάτων και τις ίδιες ιδιότητες με τον κυματικό αλγόριθμο A. Η μόνη διαφορά είνα στην πολυπλοκότητα bit, η οποία μπορεί να μειωθεί επισυνάπτοντας το M μόνο στο πρώτο μήνυμα που στέλνεται σε κάθε κανάλι. Αν είναι w ο αριθμός των bits του M, τότε η πολυπλοκότητα μηνυμάτων του PIF ξεπερνάει αυτή του κυματικού κατά w$|E|$.

3.1.5 Συγχρονισμός

Όταν πρέπει να επιτευχθεί καθολικός συγχρονισμός μεταξύ των διεργασιών, οι κυματικοί αλγόριθμοι είναι και πάλι οι πιο κατάλληλοι. Το πρόβλημα του συγχρονισμού (Synchronization ή SYN) ορίζεται ως εξής: Σε κάθε διεργασία q πρέπει να εκτελεστεί ένα γεγονός a_q και σε κάποιες διεργασίες πρέπει να εκτελεστεί το γεγονός b_p έτσι ώστε

η εκτέλεση όλων των a_q να έχει ολοκληρωθεί πριν οποιοδήποτε από τα b_p εκτελεστεί. Ο αλγόριθμος πρέπει να χρησιμοποιεί πεπερασμένο αριθμό μηνυμάτων. Σε έναν αλγόριθμο SYN τα γεγονότα b_p θεωρούνται decide γεγονότα.

Θεώρημα 5. *Κάθε SYN αλγόριθμος είναι ένας κυματικός αλγόριθμος.*

Απόδειξη. Έστω S ένας SYN αλγόριθμος. Κάθε υπολογισμός του S θα πρέπει να είναι πεπερασμένου αριθμού βημάτων και θα πρέπει να συμβαίνει ένα b_p γεγονός (decide). Αν σε κάποιον υπολογισμό του S συμβεί ένα γεγονός b_p το οποίο δεν έπεται κάποιου γεγονότος a_q σε μια διεργασία q, τότε υπάρχει μια εκτέλεση του S όπου το b_p συμβαίνει πριν το a_q.

Θεώρημα 6. *Κάθε κυματικός αλγόριθμος μπορεί να χρησιμοποιηθεί σαν SYN αλγόριθμος.*

Απόδειξη. Έστω A είναι ένας κυματικός αλγόριθμος. Για να χρησιμοποιηθεί σαν SYN αλγόριθμος, θα πρέπει κάθε διεργασία q να εκτελεί ένα a_q πριν η q στείλει κάποιο μήνυμα του A ή προκαλέσει κάποιο decide γεγονός στον A. Το γεγονός b_p συμβαίνει μετά από ένα decide γεγονός στην p. Από το λήμμα 3, κάθε decide γεγονός έπεται αιτιολογικά από ένα a_q για κάθε q.

Ο SYN αλγόριθμος που δημιουργήθηκε έχει την ίδια πολυπλοκότητα μηνυμάτων και τις ίδιες ιδιότητες με τον κυματικό αλγόριθμο A.

3.2 Ο κυματικός αλγόριθμος Echo

Ο αλγόριθμος Echo είναι ένας συγκεντρωτικός αλγόριθμος για δίκτυα τυχαίας τοπολογίας. Προτάθηκε πρώτη φορά από τον Chang και έτσι μερικές φορές καλείται και ‘ο αλγόριθμος echo του Chang’. Μία κάπως πιο αποδοτική έκδοση προτάθηκε από τον Segall. Στην παράγραφο αυτή θα παρουσιαστεί η δεύτερη αυτή έκδοση.

Ο αλγόριθμος μεταδίδει tok μηνύματα σε όλες τις διεργασίες με τη μέθοδο της πλημμύρας, δημιουργώντας έτσι ένα επικαλυπτικό δέντρο, όπως αυτό που ορίζεται στο λήμμα 2. Τα toks επιστρέφονται πίσω διαμέσω των ακμών του δέντρου. Ακολουθεί μια σύντομη περιγραφή του αλγορίθμου.

Ο αρχικοποητής στέλνει μηνύματα σε όλους τους γειτονικούς του κόμβους. Κάθε μη αρχικοποιητής, όταν λάβει το πρώτο μήνυμα προωθεί μηνύματα σε όλους τους γειτονικούς του κόμβους, εκτός από αυτόν από τον οποίο έλαβε το πρώτο αυτό μήνυμα. Όταν ο μη αρχικοποιητής λάβει μηνύματα από όλους τους γείτονές του, στέλνει ένα μήνυμα echo στον πατέρα του. Όταν ο αρχικοποιητής λάβει ένα μήνυμα από όλους τους γειτονικούς του κόμβους αποφασίζει.

Θεώρημα 7. *Ο αλγόριθμος echo είναι ένας κυματικός αλγόριθμος.*

Απόδειξη. Καταρχάς, εφόσον κάθε διεργασία στέλνει το πολύ ένα μήνυμα σε κάθε κανάλι της, ο αριθμός των μηνυμάτων που ανταλλάσσονται σε κάθε υπολογισμό είναι πεπερασμένος. Έστω γ η τελική κατάσταση σε έναν υπολογισμό C με αρχικοποιητή τον p_0.

Για αυτή την κατάσταση ορίζουμε ένα γράφο T = (P,E_T) έτσι ώστε pq $\in E_T \iff father_p = q$. Για να δείξουμε ότι ο γράφος αυτός αποτελεί ένα δέντρο πρέπει να αποδειχτεί ότι ο αριθμός των ακμών του είναι κατά ένα μικρότερος από τον αριθμό των κόμβων του. Εδώ πρέπει να παρατηρήσουμε ότι κάθε διεργασία που συμμετέχει στον C στέλνει μηνύματα σε όλους τους γείτονές της, εκτός αυτού από τον οποίο έλαβε το πρώτο μήνυμα (αν η διεργασία είναι μη αρχικοποιητής). Επομένως, κάθε γείτονάς της λαμβάνει τουλάχιστον ένα μήνυμα στον C και συμμετέχει επίσης σε αυτόν. Συνεπώς, ισχύει ότι $father_p \neq udef$ για κάθε $p \neq p_0$. Για να είναι ο T δέντρο θα πρέπει επιπλέον να μην περιέχει κύκλους. Η απόδειξη αυτού είναι όμοια με αυτή του λήμματος 2.

Η ρίζα του δέντρου είναι ο κόμβος - διεργασία p_0. Συμβολίζουμε με T_p το σύνολο των κόμβων του δικτύου που ανήκουν στο δέντρο του p. Οι ακμές του δικτύου που δεν ανήκουν στο T ονομάζονται *frond* ακμές. Στην κατάσταση γ κάθε διεργασία έχει στείλει μηνύματα σε όλες τις γειτονικές διεργασίας, εκτός από τον πατέρα της. Έτσι κάθε frond ακμή έχει μεταφέρει ένα μήνυμα σε κάθε διεύθυνση στον C. Έστω ότι είναι f_p το γεγονός όπου η διεργασία p στέλνει ένα μήνυμα στον πατέρα της και g_p το γεγονός όπου ο πατέρας λαμβάνει αυτό το μήνυμα. Με επαγωγή στους κόμβους του δικτύου αποδεικνύονται τα εξής:

1. Ο υπολογισμός C περιέχει το γεγονός f_p για κάθε $p \neq p_0$.

2. Για όλα τα s $\in T_p$ υπάρχει ένα γεγονός e $\in C_s$, τέτοιο ώστε να προηγείται του g_p, δηλαδή $e \preceq g_p$.

Υποθέτουμε τις εξής δύο περιπτώσεις:

1. Η p είναι φύλλο.
   Στην περίπτωση αυτή η p έχει λάβει ένα μήνυμα από τον πατέρα της και από όλους τους γείτονές της· όλα τα υπόλοιπα κανάλια είναι fronds. Έτσι ήταν δυνατή η αποστολή του tok μηνύματος στον πατέρα της, και αφού η γ είναι η τελική κατάσταση, η αποστολή εξετελέσθη. Το T_p περιέχει μόνο την p και είναι προφανές ότι $f_p \preceq g_p$.

2. Η p δεν είναι φύλλο.
   Εδώ, η p έχει λάβει κατά τον υπολογισμό C ένα μήνυμα από τον πατέρα της. Με επαγωγή, ο C περιλαμβάνει το $f_{p'}$, που είναι το γεγονός αποστολής των μηνυμάτων της p προς όλα τα παιδιά της p'. Εφόσος η κατάσταση γ είναι η τελική, ο C περιλαμβάνει επίσης και το $g_{p'}$. Συνεπώς, είναι εφαρμόσιμη η αποστολή tok στον πατέρα της p, και αφού η γ είναι τελική, η αποστολή εξετελέσθη. Το T_p προκύπτει από την ένωση των $T_{p'}$ των παιδιών της p και της ίδιας της p. Η επαγωγική υπόθεση μπορεί να χρησιμοποιηθεί για να αποδειχτεί ότι σε κάθε διεργασία υπάρχει ένα γεγονός που προηγείται του g_p.

Από τα παραπάνω προκύπτει ότι η p_0 έχει λάβει ένα μήνυμα από όλους τους γειτονικούς της κόμβους και έχει εκτελέσει ένα decide γεγονός, το οποίο έπεται κάποιου άλλου γεγονότος σε κάθε διεργασία. Έτσι αποδείχτηκε ότι ο αλγόριθμος echo είναι ένας κυματικός αλγόριθμος.

Ακολουθεί ο ψευδοκώδικας για τον αλγόριθμο echo, ο οποίος δημιουργεί ένα επικαλυπτικό δέντρο το οποίο μπορεί να χρησιμοποιηθεί από κάποιον αλγόριθμο που θα εκτελεστεί στη συνέχεια· για παράδειγμα ο αλγόριθμος Merlin-Segall ο οποίος υπολογίζει πίνακες δρομολόγησης συντομότερων διαδρομών, υποθέτει την ύπαρξη κάποιου επικαλυπτικού δέντρου.

---

var rec_p : integer **init** 0; – **Counts number of received messages **
$father_p$: P **init** udef;

For the initiator:
begin forall q $\in Neigh_p$ **do** send $\langle tok \rangle$ to q;
while $rec_p < \#Neigh_p$ **do**
begin receive $\langle tok \rangle$; $rec_p := rec_p + 1$ **end**;
decide
end

For non-initiators:
begin receive $\langle tok \rangle$ from neighbor q; $father_p$:=q; $rec_p := rec_p + 1$;
forall q $\in Neigh_p$, q $\neq father_p$ **do** send $\langle tok \rangle$ to q;
while $rec_p < \#Neigh_p$ **do**
begin receive $\langle tok \rangle$; $rec_p := rec_p + 1$ **end**;
send $\langle tok \rangle$ to $father_p$
end

Στο τέλος του αλγορίθμου, κάθε κόμβος γνωρίζει τον πατέρα του, όμως δε γνωρίζει τους κόμβους - παιδιά του, ενώ όλα τα μηνύματα που στέλνονται είναι πανομοιότυπα. Αν απαιτείται η γνώση των παιδιών, ο αλγόριθμος μπορεί να τροποποιηθεί ελαφρώς ούτως ώστε να αποστέλεται διαφορετικό μήνυμα στον πατέρα. Έτσι, ο κάθε κόμβος θα είναι σε θέση να αναγνωρίσει ποιοι κόμβοι είναι παιδιά του λόγω του ότι θα λαμβάνει διαφορετικά μηνύματα από αυτούς.

3.3 Αλγόριθμοι Διάσχισης

Στην παράγραφο αυτή μελετάται μια ειδική κατηγορία κυματικών αλγορίθμων, οι αλγόριθμοι διάσχισης. Σε αυτούς όλα τα γεγονότα σε ένα κύμα διατάσσονται σύμφωνα με τη σχέση αιτιότητας, ενώ το τελευταίο γεγονός συμβαίνει στην ίδια διεργασία όπου συνέβη και το πρώτο γεγονός.

Ορισμός 2. *Ένας αλγόριθμος διάσχισης είναι ένας αλγόριθμος με τα εξής χαρακτηριστικά:*

1. Σε κάθε υπολογισμό υπάρχει ένας αρχικοποιητής, ο οποίος ξεκινάει τον αλγόριθμο στέλνοντας ακριβώς ένα μήνυμα.

2. Μια διεργασία, μέχρι να λάβει ένα μήνυμα, είτε στέλνει κάποιο ή αποφασίζει.

3. Ο αλγόριθμος τερματίζει στον αρχικοποιητή και όταν αυτό συμβεί, κάθε διεργασία έχει στείλει τουλάχιστον ένα μήνυμα.

Από τις δύο πρώτες ιδιότητες καταλαβαίνει κανείς ότι σε κάθε πεπερασμένο υπολογισμό ακριβώς μία διεργασία αποφασίζει. Σε κάθε αποδεκτή κατάσταση του αλγορίθμου είτε υπάρχει ακριβώς ένα μήνυμα υπό μετάδοση ή ακριβώς μια διεργασία μόλις έλαβε ένα μήνυμα και δεν έχει στείλει ακόμα απάντηση. Κοιτώντας πιο αφαιρετικά την εκτέλεση του αλγορίθμου, θα λέγαμε πως το σύνολο των μηνυμάτων μιας εκτέλεσης μπορεί να θεωρηθεί ένα αντικείμεο (token) το οποίο, περνώντας από διεργασία σε διεργασία, τις επισκέπτεται τελικά όλες. Τότε η διεργασία που κατέχει το token κατέχει ταυτόχρονα και κάποια 'προνόμια' σε σχέση με τις υπόλοιπες. Αλγόριθμοι διάσχισης έχουν χρησιμοποιηθεί για το πρόβλημα εκλογής αρχηγού και το πρόβλημα αμοιβαίου αποκλεισμού.

Ορισμός 3. *Ένας αλγόριθμος είναι αλγόριθμος f-διάσχισης για κάποια κλάση δικτύων*

1. *αν είναι αλγόριθμος διάσχισης, για την κλάση αυτή και*

2. *αν σε κάθε κάθε υπολογισμό ο αλγόριθμος έχει επισκεφτεί τουλάχιστον $min(N, x+1)$ διεργασίες μετά από $f(x)$ περάσματα των tokens.*

3.3.1 Διάσχιση συνεκτικών δικτύων

Ο Tarry έδωσε έναν αλγόριθμο διάσχισης για τυχαία συνεκτικά δίκτυα το 1895. Ο αλγόριθμος αυτός βασίζεται σε δύο κανόνες:

R1. Μια διεργασία δεν προωθεί δύο φορές το ίδιο token στο ίδιο κανάλι.

R2. Ένας μη-αρχικοποιητής προωθεί το token στον πατέρα του, μόνο αν δεν υπάρχει άλλη επιλογή, σύμφωνα με τον κανόνα R1.

Ακολουθεί ο αλγόριθμος του Tarry σε μορφή ψευδοκώδικα.

---

var $used_p[q]$: boolean **init** false for each q $\in Neigh_p$; **Indicates whether p has already sent to q **
$father_p$: process **init** udef;

For the initiator only, execute once:
begin $father_p$:= p ; choose q $\in Neigh_p$;
$used_p[q]$:= true ; send $\langle tok \rangle$ to q;
end;

For each process, upon receipt of $\langle tok \rangle$ from q_0:
begin if $father_p$:=udef; **then** $father_p := q_0$;
if $\forall q \in Neigh_p : used_p[q]$
then decide
else if $\exists q \in Neigh_p : (q \neq father_p \wedge \neg used_p[q])$
then begin choose $q \in Neigh_p \setminus \{father_p\}$
with $\neg used_p[q]$;
$used_p[q]$:=true ; send $\langle tok \rangle$ to q;
end
else begin $used_p[father_p]$:=true ;
send $\langle tok \rangle$ to $father_q$;
end
end

Θεώρημα 8. *Ο αλγόριθμος του Tarry είναι ένας αλγόριθμος διάσχισης.*

Απόδειξη. Επειδή το token στέλνεται το πολύ μια φορά σε κάθε κατεύθυνση σε κάθε κανάλι, χρειάζεται το πολύ $2|E|$ χρόνος για να τερματίσει ο αλγόριθμος. Επίσης, επειδή

κάθε κάθε διεργασία στέλνει το πολύ μια φορά το token σε κάθε κανάλι, λαμβάνει το πολύ μια φορά το token από κάθε κανάλι, ενώ κάθε φορά που ένας μη αρχικοποιητής p κρατάει το token, έχει πραγματοποιήσει μία λήψη παραπάνω από τις αποστολές του token. Αυτό δείχνει ότι ο αριθμός των καναλιών του p είναι κατά ένα τουλάχιστον μεγαλύτερος των χρησιμοποιημένων καναλιών του, οπότε ο p δεν αποφασίζει αλλά προωθεί το token. Από τα παραπάνω εύκολα μπορεί να δει κανείς πως μόνο ο αρχικοποιητής αποφασίζει.

Όταν ο αλγόριθμος τερματίζει κάθε διεργασία έχει λάβει και έχει προωθήσει το token. Αυτό προκύπτει από τα εξής τρία σημεία:

- *Όλα τα γειτονικά κανάλια του αρχικοποιητή έχουν χρησιμοποιηθεί τουλάχιστον μία φορά για κάθε κατεύθυνση.*
  Κάθε κανάλι έχει χρησιμοποιηθεί από τον αρχικοποιητή για να στείλει το token, διαφορετικά ο αλγόριθμος δε θα τερμάτιζε. Επιπλέον, ο αρχικοποιητής λαμβάνει το token τόσες φορές όσες το έστειλε και από διαφορετικό κανάλι κάθε φορά· αυτό σημαίνει ότι το λαμβάνει μια φορά από κάθε κανάλι.

- *Για κάθε διεργασία p, όλα τα γειτονικά της κανάλια έχουν χρησιμοποιηθεί μια φορά για κάθε κατεύθυνση.*
  Υποθέτοντας ότι δεν ισχύει η πρόταση, επιλέγουμε την πιο παλιά από τις διεργασίες που επισκέφτηκε ο αλγόριθμος, έστω την p. Σύμφωνα με την προηγούμενη πρόταση, η p δεν είναι ο αρχικοποιητής. Όλα τα γειτονικά κανάλια του $father_p$ έχουν χρησιμοποιηθεί μια φορά για κάθε κατεύθυνση, που σημαίνει ότι η p έχει στείλει το token στον πατέρα της, κάτι που με τη σειρά του δείχνει ότι η p έχει χρησιμοποιήσει όλα τα γειτονικά της κανάλια για να στείλει το token. Εφόσον, όμως, το token καταλήγει στον αρχικοποιητή, η p έχει λάβει το token τόσες φορές όσες το έστειλε· οπότε το έλαβε μια φορά από κάθε γειτονικό κανάλι, κάτι που δημιουργεί αντίφαση.

- *Όλες οι διεργασίες έχουν λάβει το token και κάθε κανάλι έχει χρησιμοποιηθεί μια φορά και προς τις δύο κατευθύνσεις.*
  έστω υπάρχουν διεργασίες που δεν έχουν λάβει το token. Αν είναι p και q γειτονικές διεργασίες, έτσι ώστε η p να έχει λάβει το token ενώ η q όχι, τότε η

κατάσταση αυτή αντιτίθεται στην πρόταση ότι κάθε κανάλι της p έχει χρησιμοποιηθεί και προς τις δύο κατευθύνσεις. Έτσι, όλες οι διεργασίες έχουν λάβει το token και όλα τα κανάλια έχουν χρησιμοποιηθεί και προς τις δύο κατευθύνσεις, σύμφωνα με την προηγούμενη πρόταση.

Ο αλγόριθμος του Tarry ορίζει ένα γεννητικό δέντρο στο δίκτυο, όπως δείχνει και το λήμμα 2, με ρίζα του δέντρο τον αρχικοποιητή. Κάθε μη αρχικοποιητής έχει αποθηκεύσει τον πατέρα του στο τέλος κάθε υπολογισμού. Αν επιπλέον απαιτείται η γνώση των παιδιών των κόμβων, αυτό μπορεί να επιτευχθεί με την ανταλλαγή κάποιων παραπάνω ειδικών μηνυμάτων.

3.4 Αναζήτηση πρώτα κατά βάθος

3.4.1 Πολυπλοκότητα χρόνου

Στην παράγραφο αυτή θα δούμε τον αλγόριθμο αναζήτησης πρώτα κατά βάθος ή Depth-first Search (DFS). Ο αλγόριθμος αυτός κατασκευάζει ένα γεννητικό δέντρο με την πρόσθετη ιδιότητα ότι κάθε frond ακμή συνδέει δύο κόμβους, ο ένας από τους οποίους είναι ο πρόγονος του άλλου. Όπως αναφέρθηκε και νωρίτερα, μια frond ακμή είναι αυτή που δεν ανήκει στο γεννητικό δέντρο. Δοσμένου του γεννητικού δέντρου T του δικτύου G, για κάθε p, το T[p] αντιστοιχεί στο σύνολο των διεργασιών του υποδέντρου p και A[p] αντιστοιχεί στους προγόνους του p, τους κόμβους δηλαδή που βρίσκονται στο μονοπάτι μεταξύ της ρίζας του δέντρου και του κόμβου p. Μπορεί κανείς να παρατηρήσει πως αν $q \in T[p] \iff p \in A[q]$.

Ορισμός 4. *Ένα γεννητικό δέντρο T του G είναι ένα δέντρο αναζήτησης πρώτα-κατά-βάθος αν για κάθε frond ακμή pq,* $q \in T[p] \vee q \in A[p]$.

Τα δέντρα αναζήτησης πρώτα-κατά-βάθος χρησιμοποιούνται σε πολλούς αλγορίθμους γράφων.

Ορισμός 5. *Η πολυπλοκότητα χρόνου ενός κατανεμημένου αλγορίθμου είναι ο μέγιστος χρόνος που απαιτείται από έναν υπολογισμό του αλγορίθμου, υπό τις παρακάτω προϋποθέσεις:*

- T1. *Μια διεργασία μπορεί να εκτελέσει οποιοδήποτε πεπερασμένο αριθμό γεγονότων σε μηδενικό χρόνο.*
- T2. *Ο χρόνος που μεσολαβεί μεταξύ μιας αποστολής και μιας λήψης ενός μηνύματος είναι το πολύ μια μονάδα χρόνου.*

Η μονάδα χρόνου που χρησιμοποιείται για τη μέτρηση της διάρκειας μιας κατανεμημένης εκτέλεσης ισούται με τη μέγιστη καθυστέρηση μηνύματος αυτής της εκτέλεσης.

Λήμμα 4. *Για τους αλγορίθμους διάσχισης η πολυπλοκότητα χρόνου ισούται με την πολυπλοκότητα μηνυμάτων.*

Απόδειξη. Τα μηνύματα ανταλλάσσονται σειριακά και κάθε ανταλλαγή μπορεί να διαρκέσει μία μονάδα χρόνου.

3.4.2 Ο αλγόριθμος

Ο αλγόριθμος αναζήτησης πρώτα-κατά-βάθος (Depth-first Search ή DFS) προκύπτει αν στον αλγόριθμο του Tarry προσθέσουμε έναν ακόμα κανόνα, ο οποίος περιορίζει την δυνατότητα ελεύθερης επιλογής γείτονα για την προώθηση του token.

R3. Όταν μια διεργασία λάβει το token το στέλνει πίσω χρησιμοποιώντας το ίδιο κανάλι, αν αυτό είναι επιτρεπτό από τους κανόνες R1 και R2.

Ακολουθεί ο DFS με τη μορφή ψευδοκώδικα.

Θεώρημα 9. *Ο κλασσικός DFS αλγόριθμος υπολογίζει ένα γεννητικό δέντρο αναζήτησης πρώτα-κατά-βάθος χρησιμοποιώντας $2|E|$ μηνύματα και $2|E|$ μονάδες χρόνου.*

Απόδειξη. Ο DFS αλγόριθμος υλοποιεί τον αλγόριθμο του Tarry, άρα είναι αλγόριθμος διάσχισης που υπολογίζει ένα γεννητικό δέντρο. Έχει ήδη αποδειχτεί ότι κάθε κανάλι μεταφέρει δύο μηνύματα (ένα σε κάθε κατεύθυνση), το οποίο αποδεικνύει ότι η πολυπλοκότητα μηνυμάτων είναι $2|E|$ και αφού τα μηνύματα ανταλλάσσονται σειριακά, με κάθε ανταλλαγή να διαρκεί μία μονάδα χρόνου, προκύπτει ότι η πολυπλοκότητα χρόνου είναι επίσης $2|E|$. Απομένει να αποδειχτεί ότι ο τρίτος κανόνας που προστέθηκε συνεπάγεται τη δημιουργία ενός δέντρου αναζήτησης πρώτα-κατά-βάθος.

---

var $used_p[q]$: boolean **init** false for each q $\in Neigh_p$; **Indicates whether p has already sent to q **

$father_p$: process **init** udef;

For the initiator only, execute once:

begin $father_p$:= p ; choose q $\in Neigh_p$;

$used_p[q]$:= true ; send $\langle tok \rangle$ to q;

end

For each process, upon receipt of $\langle tok \rangle$ from q_0:

begin if $father_p$:=udef; **then** $father_p := q_0$;

if $\forall q \in Neigh_p : used_p[q]$

then decide

else if $\exists q \in Neigh_p : (q \neq father_p \wedge \neg used_p[q])$

then begin if $father_p \neq q_0 \wedge \neg used_p[q_0]$

then q := q_0

else choose $q \in Neigh_p \setminus \{father_p\}$

with $\neg used_p[q]$;

$used_p[q]$:=true ; send $\langle tok \rangle$ to q;

end

else begin $used_p[father_p]$:=true ;

send $\langle tok \rangle$ to $father_q$;

end

end

Καταρχάς, ο κανόνας R3 δείχνει ότι το πρώτο πέρασμα μιας frond ακμής ακολουθείται άμεσα από ένα δεύτερο πέρασμα, προς την ανάποδη κατεύθυνση. Αν υποθέσουμε ότι η pq είναι μια frond ακμή και η p είναι η πρώτη διεργασία που τη χρησιμοποιεί, τότε, η q έχει ήδη λάβει το token μια φορά πριν το λάβει από την p, αλλιώς η p θα ήταν ο πατέρας της q, και η μεταβλητή $used_q[p]$ θα είναι ψευδής, αφού η p είναι η πρώτη που χρησιμοποιεί την ακμή pq. Επομένως, εφαρμόζοντας τον κανόνα R3, η q στέλνει άμεσα

το token πίσω στην p.

Τώρα θα αποδειχτεί ότι αν η pq είναι frond ακμή, που χρησιμοποιείται πρώτα από την p, τότε $q \in A[p]$: Έστω το μονοπάτι που ακολούθησε το token πριν σταλεί μέσω της pq. Αφού η pq είναι frond ακμή, τότε η q έχει ήδη λάβει κάποια στιγμή το token, πριν φτάσει σε αυτή μέσω της ακμής ...,q,...,p, q

Αν στο παραπάνω μονοπάτι αντικαταστήσουμε τα σχήματα της μορφής 'r_1, r_2, r_1', όπου $r_1 r_2$ είναι frond ακμή, με το r_1, τότε θα αφαιρεθούν όλες οι frond ακμές από το μονοπάτι. Επομένως, το μονοπάτι αυτό θα ανήκει στο T και θα αποτελείται μόνο από τις ακμές που χρησιμοποιήθηκαν πριν την πρώτη χρησιμοποίηση της ακμής pq. Αν η q δεν είναι πρόγονος της p, τότε η ακμή από την q προς την $father_q$ θα έχει χρησιμοποιηθεί πριν χρησιμοποιηθεί η ακμή qp, το οποίο αντιτίθεται στον κανόνα R2 του αλγορίθμου.

3.5 Αναζήτηση πρώτα κατά εύρος

Σε αυτή την παράγραφο θα παρουσιαστεί ένας ακόμη αλγόριθμος, ο οποίος κατασκευάζει ένα γεννητικό δέντρο αναζήτησης πρώτα-κατά-εύρος (Breadth-first Search 'h BFS).

Ορισμός 5. *Ένα γεννητικό δέντρο T ενός δικτύου G αποτελεί δέντρο αναζήτησης πρώτα-κατά-εύρος, αν για κάθε κόμβο, το μονοπάτι για κάθε δέντρο που κατευθύνεται είναι ένα μονοπάτι ελαχίστων βημάτων στο G.*

3.5.1 Ο αλγόριθμος

Ο υπολογισμός ενός BFS δέντρου γίνεται με τη βοήθεια ενός συγκεντρωτικού αλγορίθμου που αρχικοποιείται από τη ρίζα του δέντρου, με έναν, δηλαδή, αλγόριθμο διάσχισης. Η κατασκευή του BFS δέντρου προχωράει ανά επίπεδα. Μόλις ολοκληρωθεί το επίπεδο i, οι γειτονικοί κόμβοι του επιπέδου αυτού προστίθενται στο δέντρο σαν επίπεδο i+1.

Ο αλγόριθμος λειτουργεί ως εξής: Όλες οι διεργασίες διατηρούν μια μεταβλητή d_u, οπου αποθηκεύουν τις αποστάσεις τους από τη ρίζα του δέντρου u_0. Η μεταβλητή αυτή έχει αρχικά την τιμή 0 για τη ρίζα του δέντρου και ∞ για όλες τις άλλες διεργασίες. Για να ξεκινήσει ο αλγόριθμος, ο αρχικοποιητής, δηλαδή η διεργασία u_0, στέλνει την

τιμή της d_{u_0}, σε όλους τους γειτονικούς κόμβους. Σε κάθε γύρο, αν μια διεργασία λάβει ένα μήνυνα m, από την p και ισχύει m + 1< d_u, θέτει d_u = m + 1 και τη μεταβλητή $father_u$ = p.

3.5.2 Χαρακτηριστικά του BFS

Ο αλγόριθμος BFS παρουσιάζει τα εξής χαρακτηριστικά:

- Αν d(u) είναι η απόσταση της u_0 από την u στο δίκτυο G, τότε κατά την εκτέλεση του αλγορίθμου, για οποιεσδήποτε γειτονικές διεργασίες u,p είτε ισχύει $d_p < d_u + 1$, είτε το d_i στέλνεται από την u στην p.
- Σε κάθε χρονική στιγμή της εκτέλεσης, αν η d_u δεν είναι άπειρη, τότε αντιστοιχεί στο μήκος ενός μονοπατιού από τη ρίζα του δέντρου προς την u.
- Ισχύει ότι d(u)$\leq d_u < n$.
- Η τιμή d_u αλλάζει το πολύ n φορές.
- Η πολυπλοκότητα μηνυμάτων είναι O(nm).

Λήμμα 5. *Για κάθε u θα ισχύει $d_u = d(u)$ εντός χρόνου d(u)n((l) + (m)).*

Απόδειξη. Για d(u) = 0 είναι προφανές. Έστω ότι ισχύει για κάθε p. όπου $d_p \leq k$. Ας πάρουμε μια διεργασία u με d_u = k + 1 και μια γειτονική της p με d_p = k. Σε χρόνο kn((l) + (d)) η p έθεσε d(p) = k και αποφάσισε να στείλει την τιμή k στη u. Σε n(l) επιπλέον χρόνο η p θα στείλει το k στο κανάλι pu. Σε p(d) επιπλέον χρόνο η u θα λάβει το μήνυμα, θα θέσει d_u = k + 1 και θα επιλέξει ως γονέα της την p.

Θεώρημα 10. *Με την εκτέλεση του BFS αλγορίθμου, το σύστημα συγκλίνει σε μια κατάσταση όπου έχει κατασκευαστεί επικαλυπτικό δέντρο T(G): Η απόσταση μιας κορυφής απότ η u_0 είναι ίδια και στο G και στο T(G) και αυτό ολοκληρώνεται σε χρόνο O(δn(l+d)).*

Κεφάλαιο 4

Το λειτουργικό σύστημα TinyOS και η γλώσσα προγραμματισμού nesC

4.1 Θέματα σχεδιασμού ενός λειτουργικού συστήματος για Ασύρματα Δίκτυα Αισθητήρων

Τα παραδοσιακά λειτουργικά συστήματα είναι το λογισμικό του συστήματος και περιλαμβάνουν προγράμματα που διαχειρίζονται τους πόρους του υπολογιστή, ελέγχουν τις περιφερειακές συσκευές και γενικά προσφέρουν μια αφαιρετικότητα στο λογισμικό των εφαρμογών. Επομένως, οι λειτουργίες ενός παραδοσιακού λειτουργικού συστήματος υπάρχουν για να διαχειρίζονται διεργασίες, μνήμη, CPU, σύστημα αρχείων και συσκευές. Συνήθως η υλοποίηση που ακολουθείται είναι ένα κατά τμήματα και κατά επίπεδα πρότυπο, συμπεριλαμβάνοντας ένα κατώτερο επίπεδο, αυτό του πυρήνα (kernel) και ένα ανώτερο επίπεδο που περιέχει τις βιβλιοθήκες του συστήματος. Τα παραδοσιακά λειτουργικά συστήματα δεν είναι κατάλληλα για τα WSNs εξαιτίας των περιορισμένων πόρων, των ποικίλων δεδομενοκεντρικών (data-centric) εφαρμογών και της ευμετάβλητης τοπολογίας τους. Τα WSNs χρειάζονται ένα νέο τύπο λειτουργικού συστήματος,

που να λαμβάνει υπόψιν τα ιδιαίτερα χαρακτηριστικά τους.

Υπάρχουν αρκετά θέματα που θα πρέπει να λαμβάνονται υπόψιν κατά το σχεδιασμό ενός λειτουργικού συστήματος για WSNs.

- *Διαχείριση των διεργασιών του συστήματος και χρονοπρογραμματισμός* Τα παραδοσιακά λειτουργικά συστήματα παρέχουν προστασία στις διεργασίες δεσμεύοντας ένα ξεχωριστό χώρο στη μνήμη για κάθε μία. Έτσι οι διεργασίες διατηρούν τα δεδομένα και τις πληροφορίες τους στο δικό τους χώρο. Αυτό έχει σαν αποτέλεσμα την πολλαπλή αντιγραφή δεδομένων, κάτι που προφανώς δεν είναι αποδοτικό ως προς την κατανάλωση ενέργειας για τα WSNs. Όσο αναφορά τον χρνοπρογραμματισμό, ένας πραγματικού-χρόνου scheduler, όπως ένας earliest deadline first (EDF), για τα WSNs είναι μια καλή επιλογή, για εφαρμογές πραγματικού χρόνου, όμως θα πρέπει να τεθούν κάποια όρια στον αριθμό των διεργασιών, μιας και αυτό καθορίζει την πολυπλοκότητα του EDF scheduler.

- *Διαχείριση μνήμης* Όπως αναφέρθηκε παραπάνω, τα παραδοσιακά λειτουργικά συστήματα δεσμεύουν ένα ξεχωριστό χώρο μνήμης για κάθε διεργασία. Αυτό μπορεί να είναι σημαντικό για θέματα ασφάλειας και προστασίας των διεργασιών, όμως στα WSNs η μνήμη των κόμβων αισθητήρων είναι περιορισμένη. Επομένως, ο διαμοιρασμός του ίδιου χώρου μνήμης μπορεί να μειώσει τις απαιτήσεις.

- *Το μοντέλο του πυρήνα* Τα μοντέλα των 'οδηγούμενου-από-γεγονότα' πυρήνα και μηχανής πεπερασμένων καταστάσεων (finite state machine ή FSM) έχουν χρησιμοποιηθεί για το σχεδιασμό μικροπυρήνων για WSNs. Το 'οδηγούμενα-από-γεγονότα' μοντέλο ίσως να εξυπηρετήσει αρκετά καλά τα WSNs, τα οποία μοιάζουν με 'οδηγούμενα-από-γεγονότα' συστήματα. Ένα γεγονός μπορεί να είναι η αποστολή ή η λήψη ενός πακέτου, η ανίχνευση ενός γεγονότος, συναγερμός για εξάντληση μπαταρίας, κ.ο.κ. Το FSM μοντέλο είναι βολικό για την αναπαράσταση του σύγχρονου χαρακτήρα του συστήματος, της δυνατότητας αντίδρασής του και του συγχρονισμού που απαιτείται.

- *Διεπαφή με το χρήστη (API)* Τα APIs που θα προσφέρουν οι κόμβοι αισθητήρων θα πρέπει να δίνουν στις εφαρμογές τη δυνατότητα πρόσβασης στο hardware.

- *Επαναπρογραμματισμός* Η συμπεριφορά των κόμβων αισθητήρων και οι αλγόριθμοί τους ίσως χρειαστεί να τροποποιηθούν, είτε για βελτίωση της λειτουργικότητάς τους είτε για τη διατήρηση της ενέργειας. Έτσι, το λειτουργικό σύστημα θα πρέπει να είναι σε θέση να επαναπρογραμματιστεί.

- *Σύστημα αρχείων* Οι κόμβοι των αισθητήρων δε διαθέτουν εξωτερικό δίσκο, συνεπώς δε μπορούν να έχουν σύστημα αρχείων.

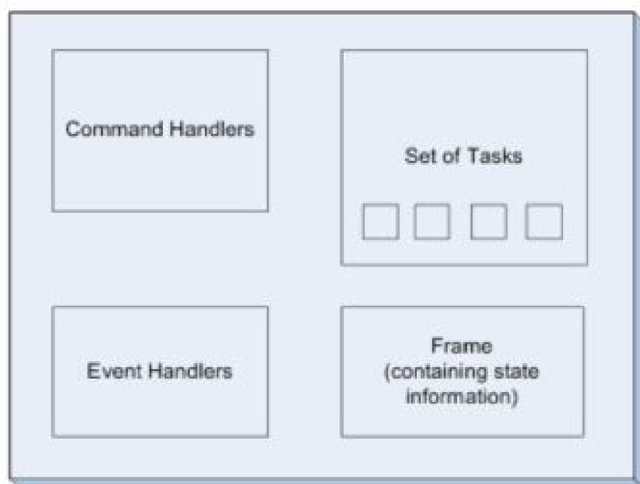
4.2 Το λειτουργικό σύστημα TinyOS

Χιλιάδες ερευνητές σε όλο τον κόσμο έχουν εμπλακεί στην υλοποίηση του λειτουργικού συστήματος TinyOS για εφαρμογές των WSNs. Θα μπορούσε να πει κανείς ότι η επιτυχία των WSNs οφείλεται κατά ένα μεγάλο βαθμό στην ανάπτυξη του λειτουργικού αυτού συστήματος.

Το TinyOS σχεδιάστηκε ειδικά για τα WSNs. Εισάγει ένα δομημένο, 'οδηγούμενου-από-τα-γεγονότα' (event-driven) μοντέλο και ένα σχεδιασμό του λογισμικού βασισμένο σε components. Το λογισμικό αυτό υποστηρίζει σε ένα μεγάλο βαθμό την έννοια του συγχρονισμού, ενισχύει την ευρωστία και ελαχιστοποιεί την κατανάλωση ενέργειας, ενώ διευκολύνει την ανάπτυξη πολύπλοκων αλγορίθμων και πρωτοκόλλων. Το TinyOS αποτελείται από components που συνδέονται με καλά ορισμένα interfaces. Θα μπορούσαμε να το φανταστούμε σαν κομμάτια υλικού που συνδέονται μεταξύ τους. Η μεγάλη ποικιλία σε πλατφόρμες υλικού και σε εφαρμογές αντιμετωπίζεται με την συνένωση των κατάλληλων components.

Το TinyOS λοιπόν διαφέρει από τα παραδοσιακά λειτουργικά συστήματα. Οι έννοιες των Kernel, διαχείριση διεργασιών (process management), εικονική μνήμη (virtual memory), δυναμική δέσμευση μνήμης (dynamic memory allocation), software σήματα (signals) δεν υπάρχουν εδώ. Χωρίς Kernel η διαχείριση του hardware γίνεται απευθείας, υπάρχει μόνο μία διαδικασία στο σύστημα, υπάρχει γραμμικός χώρος διευθύνσεων και η μνήμη ανατίθεται στατικά κατά τη διάρκεια του compile της εφαρμογής. Ως αποτέλεσμα, το TinyOS δεν ξεπερνάει τα 3500 bytes.

Components Το TinyOS προσφέρει ένα Event-driven μοντέλο λογισμικού που επιπλέον υποστηρίζει την τμηματικότητα (modularity). Όλα αυτά επιτυγχάνονται με τη βοήθεια των components. Τα components είναι τα συστατικά στοιχεία του TinyOS και παρέχουν αντίστοιχες λειτουργίες. Ένα component, για παράδειγμα, θα μπορούσε να χειρίζεται τον υπολογισμό των διαδρομών για τη δρομολόγηση των πακέτων.



Σχήμα 4.1: Component

Στο παραπάνω σχήμα (Σχ. 4.1) παρουσιάζεται η γενική δομή ενός component. Το *frame* εμπεριέχει όλη την πληροφορία για την κατάσταση των δεδομένων του component.Στις *εργασίες (tasks)* γίνεται ουσιαστικά όλο το υπολογιστικό μέρος των λειτουργιών των components, ενώ τα *γεγονότα (events)* και οι *εντολές (commands)* που αποτελούν τη διεπαφή (interface) του component χρησιμοποιούνται για τη συνεργασία με άλλα components. Tasks, events και commands εκτελούνται στο χώρο που ορίζεται από τα frames.

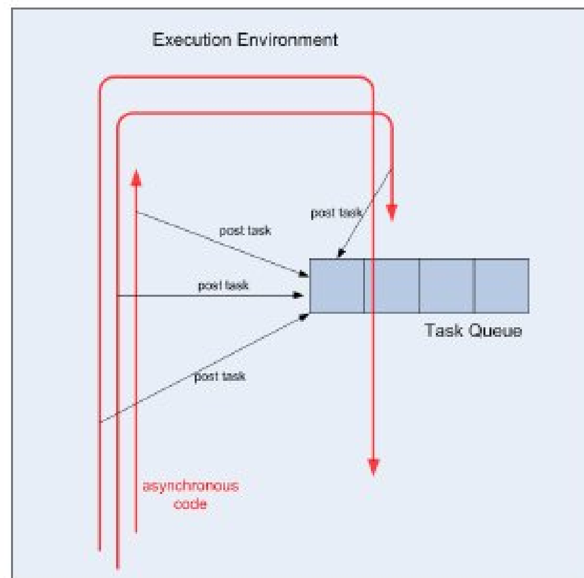
Τα components ιεραρχούνται σε κατηγορίες, ξεκινώντας από αυτά που αποτελούν μια αφαιρετική αναπαράσταση του hardware (hardware abstractions) έως υψηλού επιπέδου components. Αυτά που μοντελοποιούν το hardware παρέχουν interfaces που αντικατοπτρίζουν τις λειτουργίες του και τις διακοπές. Ένα τυπικό hardware abstraction είναι το RFM radio component, το οποίο παρέχει εντολές για έλεγχο των I/O εξόδων του RFM πομποδέκτη. Επίσης παρέχει events για την ενημέρωση άλλων components για

την κατάσταση της αποστολής/λήψης σε επίπεδο bit. Το Radio Byte component είναι ένα παράδειγμα component που βρίσκεται ένα επίπεδο πιο πάνω από τα primitive. Η δουλειά του είναι να επικοινωνεί με το υποκείμενο RFM component, στέλνοντας και λαμβάνοντας δεδομένα προς και από αυτό και να προκαλεί ένα event όταν η μεταφορά ενός byte έχει ολοκληρωθεί. Τέλος τα πιο υψηλού επιπέδου components είναι υπεύθυνα για τον έλεγχο των υποκείμενων components, καθώς επίσης για τη δρομολόγηση, την επεξεργασία των δεδομένων και την εφαρμογή λειτουργιών συνάθροισης.

Tasks Όπως αναφέρθηκε παραπάνω, τα tasks είναι κομμάτια κώδικα όπου γίνεται σχεδόν όλη η υπολογιστική δουλειά και θα πρέπει να εκτελούνται μέχρι να ολοκληρωθούν, αλλά η εκτέλεσή τους μπορεί να διακοπεί από commands και events handlers. Κάθε task καλείται, γίνεται *posted*, από commands και events και εκτελείται ανεξάρτητα από άλλα tasks, είναι δηλαδή atomic εργασία. Επειδή πολλά tasks μπορεί να δημιουργηθούν από διάφορους handlers, ο *Scheduler* του TinyOS ακολουθεί FIFO πολιτική για την εκτέλεσή τους. Όταν κανένα δεν περιμένει να εκτελεστεί ή δεν εκτελείται ο scheduler απενεργοποιεί τον κόμβο.

Στο σχήμα 4.2 φαίνεται το περιβάλλον εκτέλεσης μιας εφαρμογής του TinyOS. Το τμήμα του κώδικα που προσπελαύνεται από handlers διακοπών ονομάζεται 'ασύγχρονος κώδικας '. Αν αυτό το κομμάτι κώδικα προσπελαύνεται ταυτόχρονα από πολλούς handlers μπορεί να δημιουργηθούν συνθήκες συναγωνισμού. Για να αποφευχθεί κάτι τέτοιο, ο προγραμματιστής θα μπορούσε να τοποθετήσει τον ασύχρονο κώδικα σε task.

Εντολές (Commands) και Γεγονότα (Events) Οι εντολές είναι αιτήσεις υψηλού επιπέδου προς χαμηλότερου επιπέδου components, για εκτέλεση μιας συγκεκριμένης εργασίας. Τα γεγονότα είναι σήματα που στέλνουν χαμηλού επιπέδου components προς υψηλότερου επιπέδου components. Οι αντίστοιχοι handlers που υπάρχουν μέσα στα components αυτό που κάνουν είναι να μεταβάλλουν την εσωτερική του κατάσταση, τοποθετώντας κάποιες πληροφορίες στο frame του. Επίσης, μπορούν να καλέσουν σε λειτουργία κάποιο task, κάποια εντολή ή ακόμα και να προκαλέσουν κάποιο άλλο γεγονός. Είναι σημαντικό να προσέξουμε ότι για να διατηρεί τον 'οδηγούμενο-από-τα-γεγονότα' χαρακτήρα του το TinyOS, θα πρέπει και οι εντολές και τα γεγονότα να εκτελούνται



Σχήμα 4.2: Περιβάλλον εκτέλεσης του TinyOS

έως ότου ολοκληρωθούν. Η εκτέλεση των εντολών δε θα πρέπει να περιμένει πολύ να ξεκινήσει, ούτε όμως και να μπλοκάρεται. Τα γεγονότα μπορούν να αναστείλουν τη λειτουργία κάποιου task, αφού είναι πιο σημαντικά. Το αντίστροφο δε μπορεί να συμβεί.

Η λειτουργικότητα των components στο TinyOS περιγράφεται από διεπαφές (interfaces), οι οποίες σχηματίζονται από τα commands και τα events. Ένα components προσφέρει interfaces και χρησιμοποιεί κάποιες από άλλα components. Ο handler του command υλοποιείται στο component-παροχέα, ενώ η υλοποίηση του handler των events στο component-χρήστη.

Split-phase programming Όπως αναφέρθηκε στα παραπάνω, οι εντολές και τα γεγονότα είναι ο μόνος τρόπος επικοινωνίας της μιας συνιστώσας λογισμικού με μια άλλη. Επιπλέον, είδαμε ότι και οι εντολές αλλά και τα γεγονότα θα πρέπει να συνεχίζουν τη λειτουργία τους έως ότου ολοκληρωθούν. Αυτό που δεν είναι σαφές είναι το πώς κάποια συνιστώσα λογισμικού που έχει στείλει μια εντολή σε μια άλλη, θα πάρει κάποιο feedback από αυτό, πώς δηλαδή θα ενημερωθεί για την κατάσταση της αίτησης που έστειλε.

Για παράδειγμα, πώς ένα πρωτόκολλο 'Αυτόματης Επανάληψης Αίτησης ' (Automatic Repeat Request) θα ενημερωθεί από το MAC πρωτόκολλο για το αν ένα πακέτο έχει σταλεί επιτυχώς· Από τα παραπάνω προκύπτει η ιδέα του split-phase προγραμματισμού, ο οποίος χρησιμοποιεί την έννοια των αμφίδρομων διεπαφών: σε κάθε εντολή αντιστοιχίζεται ένα γεγονός. Μια υψηλού επιπέδου συνιστώσα λογισμικού στέλνει μια εντολή έναρξη κάποιας δραστηριότητας σε μια άλλη συνιστώσα λογισμικού. Η εντολή επιστρέφει αμέσως, δείχνοντας την κατάσταση της αίτησης, ακόμα κι αν η λειτουργία δεν έχει ολοκληρωθεί. Όταν ολοκληρωθεί η λειτουργία, η αντίστοιχη συνιστώσα προκαλεί ένα γεγονός προς τις συνιστώσες που θα συνεχίσουν με άλλες δραστηριότητες. Στο ενδιάμεσο, ο επεξεργαστής μπορεί να εξυπηρετήσει άλλα γεγονότα ή tasks ή να έρθει σε κατάσταση 'ύπνου ', αν δεν περιμένει κάποια εργασία στην ουρά για εκτέλεση. Έτσι επιτυγχάνεται και καλή διαχείριση ενέργειας από το σύνολο των συνιστωσών λογισμικού του TinyOS.

4.3 Η γλώσσα προγραμματισμού nesC

Η nesC είναι μια γλώσσα προγραμματισμού ενσωματωμένα διαδικτυωμένα συστήματα. Αποτελεί μια επέκταση τη C και έχει χρησιμοποιηθεί για τη δημιουργία του TinyOS. Έτσι υποστηρίζει το 'οδηγούμενο-από-τα-γεγονότα' μοντέλο προγραμματισμού, τον ταυτοχρονισμό στις λειτουργίες, καθώς επίσης και το αποτελούμενο από ξεχωριστά δομικά στοιχεία (components) μοντέλο σχεδιασμού. Κύριος στόχος είναι ο εύκολος συνδυασμός των δομικών αυτών στοιχείων για τη δημιουργία μιας ολοκληρωμένης εφαρμογής.

Ο σχεδιασμός της nesC βασίζεται στα παρακάτω θέματα:

Η nesC είναι προέκταση της C. Η C παράγει πολύ αποδοτικό κώδικα για όλους τους μικροεπεξεργαστες που χρησιμοποιούνται στα δίκτυα αισθητήρων και επίσης παρέχει λειτουργίες για έυκολη προσπέλαση στο υλικό. Επιπλέον, οι περισσότεροι προγραμματιστές είναι εξοικειωμένοι με τον κώδικα της C. Παρ'ολα αυτά η C έχει και κάποια μειονεκτήματα, όπως η μη υποστήριξη της δημιουργίας ασφαλούς κώδικα ή εφαρμογών που αποτελουνται από ξεχωριστά δομικά μέρη. Αντιθέτως η nesC υποστηρίζει όλα τα παραπάνω.

Ολική ανάλυση του προγράμματος κατά τη διάρκεια της μεταγλώττισης. Στη nesC δεν είναι δυνατή η μεταγλώττιση ξεχωριστών μελών του προγράμματος. Αυτό προσφέρει καλύτερη ανάλυση του κώδικα και καλύτερη απόδοση. Ένα παράδειγμα όπου η πρακτική αυτή αποδεικνύεται χρήσιμη είναι στο χειρισμό καταστάσεων συναγωνισμού.

Στατικός κώδικας. Η πρακτική της δυναμικής δέσμευσης μνήμης δεν υποστηρίζεται από τη nesC, ενώ το γράφημα των διασυνδέσεων είναι γνωστό κατά τη μεταγλώττιση. Το ίδιο το μοντέλο των components και οι διεπαφές (interfaces) που επιδέχονται παραμετροποίησης ελαχιστοποιούν τις ανάγκες της δυναμικής δέσμευσης μνήμης. Οι περιορισμοί αυτοί αυξάνουν την αποδοτικότητα και κάνουν την ανάλυση του κώδικα πολύ πιο απλή και επακριβή.

Η nesC υποστηρίζει και αντικατοπτρίζει το σχεδιασμό του TinyOS. Όπως αναφέρθηκε και παραπάνω, η nesC βασίζεται στο μοντέλο των components και υποστηρίζει το 'οδηγούμενο-από-τα-γεγονότα' μοντέλο λειτουργίας του TinyOS.

Διεπαφές Οι διεπαφές στη nesC είναι αμφίδρομες: ορίζουν μια αλληλεπίδραση μεταξύ δύο συνιστωσών λογισμικού, του παροχέα και του χρήστη, χρησιμοποιώντας πολλαπλές συναρτήσεις. Η διεπαφή ορίζει ένα σύνολο από commands που υλοποιούνται στον παροχέα και ένα σύνολο από events που υλοποιούνται στον χρήστη.

Ο ορισμός μια διεπαφής γίνεται ως εξής:

interface:
    interface identifier { *declaration-list* }
storage-class-specifier: also one of
    command event async

Ο identifier είναι το όνομα της διεπαφής και έχει global scope. Όλες οι διεπαφές έχουν διαφορετικά μεταξύ τους ονόματα. Στο πεδίο declaration list ορίζονται τα commands και τα events. Η λέξη async είναι προαιρετική και χρησιμοποιείται για να δείξει ότι το αντίστοιχο command ή event μπορεί να εκτελεστεί σε έναν χειριστή διακοπών.

Ένα παράδειγμα διεπαφής φαίνεται παρακάτω:

```
interface SendMsg{
  command result_t send(uint16_t address, uint8_t length,
  TOS_MsgPtr msg);
  event result_t sendDone(TOS_MsgPtr msg, result_t success);
}
```

Τα components που παρέχουν τη διεπαφή SendMsg πρέπει να υλοποιούν την εντολή send και τα components που τη χρησιμοποιούν πρέπει να υλοποιούν το γεγονός sendDone. Ο ορισμός της διεπαφής έξω από το component όπου χρησιμοποιείται οδηγεί στη δημιουργία κάποιων προκαθορισμένων διεπαφών, κάνοντας τα components πιο εύχρηστα και πιο ευέλικτα.

Οι λειτουργίες που μοντελοποιούν τον split-phase προγραμματισμό υλοποιούνται τοποθετώντας τις εντολές και τις αντίστοιχες αντιδράσεις (γεγονότα) στην ίδια διεπαφή. Για παράδειγμα, στην παραπάνω διεπαφή SendMsg η εντολή send και το γεγονός sendDone αποτελούν μέλη του split-phased πακέτου Send.

Components Τα components στη nesC είναι δύο τύπων: τα *modules* και τα *configurations*. Τα modules αποτελούν ουσιαστικά την υλοποίηση της εφαρμογής σε κώδικα C, ενώ στα configurations γίνεται το wiring των components, περιγράφεται, δηλαδή, ο τρόπος σύνδεσής τους.

Ο κώδικας ενός component ξεκινάει με τον ορισμό του:

nesC-file:
  {*includes-list module*}
  {*includes-list configuration*}
  ...
module:
  {*module identifier specification module-implementation*}
configuration:

{*configuration identifier specification configuration-implementation*}

Ο ορισμός ενός component αποτελείται από μια προαιρετική λίστα με C αρχεία και τον ορισμό του module ή του configuration, τα οποία ορίζονται με το όνομά τους (*identifier*), τη λίστα των uses interface *i*dentifier και provides interface *i*dentifier (*specification*) και τέλος παραθέτοντας την υλοποίησή τους (*module/configuration implementation*).

- **Modules**

  Ένα module αποτελείται από ένα σύνολο δηλώσεων και ορισμών σε γλώσσα C, ορισμούς των tasks και υλοποιήσεις των εντολών και των γεγονότων. Η υλοποίηση μιας παραμετροποιημένης εντολής ή ενός γεγονότος έχει τη σύνταξη μιας συνάρτησης της C. Για παράδειγμα, ένα component που παρέχει τη διεπαφή SendMsg θα πρέπει το αντίστοιχο module να υλοποιεί την εντολή Send:

```
command result_t Send.send(uint16_t address, uint8_t length,
TOS_MsgPtr msg){
   ...
   return SUCCESS;
}
```

  Μια εντολή καλείται γράφοντας ***call*** *command_identifier(list_of_parameters)*. Ένα γεγονός προκαλείται γράφοντας ***signal*** *event_identifier(list_of_parameters)*. Η εκτέλεση των εντολών και των γεγονότων είναι άμεση, ακριβώς όπως η εκτέλεση μιας συνάρτησης.

  Εκτός από εντολές και γεγονότα, ένα module περιλαμβάνει τον ορισμό των *tasks* (βλ. 4.2). Στα tasks είναι που γίνονται ουσιαστικά οι υπολογιστικές εργασίες της εφαρμογής. Ορίζονται ως εξής:

  task void task_identifier()

  Όπως μπορεί να δει κανείς, τα tasks δεν δέχονται παραμέτρους και δεν επιστρέφουν καμία τιμή, ενώ καλούνται γράφοντας ***post*** *task_identifier*.

Στο σημείο αυτό πρέπει να αναφέρουμε τις *atomic* εκφράσεις. Χρησιμοποιούνται για αμοιβαίο αποκλεισμό, για ενημέρωση σύγχρονων δεδομένων και άλλες παρόμοιες λειτουργίες, μιας και εγγυώνται ότι η εκτέλεση τους γίνεται σαν να μην εκτελείται κανένας άλλος υπολογισμός ταυτόχρονα.

Ένα παράδειγμα χρήσης των atomic εκφράσεων φαίνεται εδώ:

```
bool busy; //global
void f() {
 bool available;
 atomic {
  available =! busy;
  busy=TRUE;
 }
 if (available) do_something;
  atomic busy = FALSE;
}
```

- **Configurations**

  Η υλοποίηση ενός configuration αποτελείται από τη σύνδεση, το wiring όπως ονομάζεται, των components που χρησιμοποιεί η εφαρμογή:

  configuration-implementation:
  implementation { *component-list connection-list* }

  Component-list είναι η λίστα των components που χτίζουν την εφαρμογή και connection-list είναι ο τρόπος που αυτά συνδέονται.

  Η σύνταξη των συνδέσεων φαίνεται εδώ:

  connection:
  endpoint = *endpoint*
  endpoint → *endpoint*
  endpoint ← *endpoint*

Τα endpoints αντιστοιχούν σε κάποιο *στοιχείο ορισμού*, που μπορεί να είναι μια διεπαφή, μια εντολή ή ένα γεγονός. Αν ονομάσουμε *εξωτερικά* τα στοιχεία ορισμού που ανήκουν στον ορισμό του configuration και εσωτερικά τα στοιχεία ορισμού που ανήκουν σε κάποιο από τα components του, τότε η παραπάνω σύνταξη έχει νόημα αν ισχύουν οι παρακάτω περιπτώσεις:

- endpoint = endpoint: Με τη σύνταξη αυτή τα δύο στοιχεία γίνοντα ισότιμα. Για τα στοιχεία αυτά η πρώτη περίπτωση είναι το ένα να είναι εξωτερικό και το άλλο εσωτερικό και να παρέχονται ή να χρησιμοποιούνται και τα δύο, και η δεύτερη περίπτωση είναι και τα δύο στοιχεία να είναι εξωτερικά και το ένα να παρέχεται ενώ το άλλο να χρησιμοποιείται.
- endpoint $\rightarrow$ endpoint: Αυτή η σύνδεση αφορά δύο εσωτερικά στοιχεία. Το στοιχείο στην αρχή του βέλους πάντα χρησιμοποιείται και συνδέεται με το στοιχείο στη δεξιά πλευρά του βέλους, το οποίο πάντα παρέχεται.
- $\text{endpoint}_1 \leftarrow \text{endpoint}_2$: Η σύνδεση αυτή είναι ισότιμη με τη σύνδεση $\text{endpoint}_2 \rightarrow \text{endpoint}_1$.

Και στις τρεις παραπάνω περιπτώσεις συνδέσεων θα πρέπει τα στοιχεία ορισμού να είναι συμβατά μεταξύ τους. Επίσης, αν τα στοιχεία είναι εντολές ή γεγονότα θα πρέπει να έχουν το ίδιο πρωτότυπο, ανώ αν ένα από τα στοιχεία είναι παραμετροποιημένο θα πρέπει και το άλλο να δέχεται τις ίδιες παραμέτρους.

Ένα παράδειγμα της υλοποίησης ενός configuration φαίνεται παρακάτω:

```
configuration C{
 provides interface X;
}
implementation{
  components C1, C2;
  X = C1.X;
  X = C2.X;
}
```

Concurrency Η εκτέλεση μιας εφαρμογής σε nesC αποτελείται από την εκτέλεση των tasks, που πρέπει να εκτελούνται έως ότου ολοκληρωθούν, και από χειριστές διακοπών που καλούνται ασύγχρονα από το hardware. Η μη διακοπή της εκτέλεσης των tasks έως ότου ολοκληρωθεί, τα καθιστά atomic ως προς τα άλλα tasks, όμως δεν είναι atomic ως προς τους χειριστές των διακοπών. Το ασύγχρονο αυτό μονελο λειτουργίας κάνει τη nesC ευάλωτη σε συνθήκες συναγωνισμού, σε συγκεκριμένα σημεία του κώδικα που είναι διαμοιραζόμενα, όπως είναι οι global μεταβλητές. Οι συνθήκες συναγωνισμού μπορούν να αντιμετωπιστούν εισάγοντας τις ενέργειες πρόσβασης σε διαμοιραζόμενα σημεία είτε σε tasks είτε σε atomic εκφράσεις. Κατά τη διάρκεια της μεταγλώττισης η nesC ενημερώνει το χρήστη για πιθανή εμφάνιση συνθηκών συναγωνισμού.

Τυπικά ο κώδικας μια εφαρμογής χωρίζεται σε δύο κατηγορίες:

1. Σύγχρονος Κώδικας (Synchronous Code ή SC): ο κώδικας (συναρτήσεις, εντολές, γεγονότα, tasks) που είναι προσπελάσιμος μόνο από tasks.

2. Ασύγχρονος Κώδικας (Asynchronous Code ή AC): ο κώδικας που είναι προσπελάσιμος από έναν τουλάχιστον χειριστή διακοπών.

Σύμφωνα με τα παρπάνω, προκύπτει το εξής συμπέρασμα:

Μη ύπαρξη συνθηκών συναγωνισμού: Κάθε ενημέρωση σε διαμοιραζόμενη κατάσταση δεδομένων είτε είναι μόνο Σ` είτε συμβαίνει μέσα σε atomic έκφραση. Το σώμα μιας συνάρτησης f που καλείται μέσα από μια atomic έκφραση, θεωρείται ότι βρίσκεται μέσα στην έκφραση όσο οι κλήσεις της συνάρτησης f.

4.4 Άλλα Λειτουργικά Συστήματα

4.4.1 Mate

Το Mate έχει σχεδιαστεί έτσι ώστε να δουλεύει σαν ένα component του TinyOS που βρίσκεται στο υψηλότερο επίπεδο. Είναι ένας διερμηνέας (byte-code interpreter) και σκοπό έχει να κάνει το TinyOS εύκολα προσβάσιμο σε μη ειδικούς προγραμματιστές,

καθώς επίσης και να καταστήσει τον προγραμματισμό ενός ολόκληρου δικτύου αισθητήρων μια εύκολη και γρήγορη διαδικασία. Επιπλέον προσφέρει ένα περιβάλλον εκτέλεσης, που είναι πολύ εξυπηρετικό για το UC-Berkley mote, αφού στο σύστημα αυτό δεν υπάρχει μηχανισμός προστασίας του υλικού. Στο Mate, ο κώδικας του προγράμματος αποτελείται από κάψουλες (capsules). Κάθε κάψουλα έχει 24 εντολές και το μήκος κάθε εντολής είναι 1 byte. Οι κάψουλες περιέχουν πληροφορίες για τον τύπο και την έκδοσή τους και μπορούν να αναπτυχθούν μέσα στο δίκτυο. Το Mate υλοποιεί ένα beaconless (BLESS) ad-hoc πρωτόκολλο δρομολόγησης, ενώ έχει τη δυνατότητα ανάπτυξης νέων πρωτοκόλλων δρομολόγησης. Ένας κόμβος αισθητήρα που λαμβάνει μια καινούρια έκδοση μιας κάψουλας, την εγκαθιστά. Οι κάψουλες ταξινομούνται σε τέσσερις κατηγορίες: κάψουλες αποστολής μηνύματος, λήψης μηνύματος, χρονιστή και υπορουτίνας. Ένα γεγονός προκαλεί την έναρξη της εκτέλεσης του Mate, το οποίο μπορεί να χρησιμοποιηθεί τόσο σαν μια εικονική μηχανή για ανάπτυξη εφαρμογών, όσο και σαν ένα εργαλείο για τη διαχείριση ολόκληρου του δικτύου αισθητήρων.

4.4.2 MagnetOS

Το MagnetOS είναι ένα κατανεμημένο προσαρμοστικό λειτουργικό σύστημα σχεδιασμένο ειδικά για προσαρμογή των εφαρμογών και διατήρηση ενέργειας. Άλλα λειτουργικά συστήματα δεν παρέχουν μηχανισμούς για διασκευή των εφαρμογών σε εύρος δικτύου ή τεχνικές για διατήρηση της ενέργειας με αποδοτική χρήση των διαθέσιμων πόρων των κόμβων από τις εφαρμογές. Οι μηχανισμοί αυτοί συνήθως υλοποιούνται από την ίδια την εφαρμογή, κάτι που δεν είναι ιδιαίτερα αποδοτικό ως προς την κατανάλωση ενέργειας. Οι στόχοι του MagnetOS είναι:

1. η προσαρμογή στους υπάρχοντες πόρους και στις αλλαγές τους, σε σταθερή βάση,

2. η αποδοτικότητα ως προς την κατανάλωση ενέργειας,

3. η παροχή μιας αφαιρετικότητας για τις εφαρμογές,

4. η κλιμακωσιμότητα για εφαρμογή σε μεγάλα δίκτυα.

Το MagnetOS είναι ένα single system image (SSI) ή μια απλή ενιαία εικονική μηχανική Java που περιλαμβάνει στατικά και δυναμικά μέρη. Τα στατικά μέρη ξαναγράφουν την εφαρμογή σε επίπεδο byte και προσθέτουν τις απαραίτητες εντολές στις αρχικές εφαρμογές. Τα δυναμικά μέρη χρησιμοποιούνται για τον έλεγχο των εφαρμογών, τη δημιουργία αντικειμένων και άλλες λειτουργίες. Η SSI αφαιρετικότητα παρέχει περισσότερη ελευθερία στην δημιουργία αντικειμένων και απλοποιεί την ανάπτυξη των εφαρμογών. Το MagnetOS παρέχει μια διεπαφή για τους προγραμματιστές έτσι ώστε να τοποθετούν ρητά τα αντικείμενα, παρακάμπτοντας τις αυτόματες αποφάσεις τοποθέτησης. Το λειτουργικό αυτό σύστημα επίσης, προσφέρει δύο online αλγορίθμους που λαμβάνουν υπόψιν την κατανάλωση ενέργειας (NetNull και NetCenter) για χρήση από εφαρμογές που κινούνται μέσα στο δίκτυο, έτσι ώστε να μειωθεί η κατανάλωση ενέργειας και να αυξηθεί ο χρόνος ζωής του δικτύου. Ο Netpull λειτουργεί με διαδοχικά άλματα στο φυσικό επίπεδο, και ο NetCenter με πολλαπλά άλματα σε επίπεδο δικτύου. Η διαφορά μεταξύ της παραδοσιακής ad-hoc δρομολόγησης και της δρομολόγησης του NetPull (NetCenter) είναι ότι τα άκρα της επικοινωνίας στην ad-hoc δρομολόγηση είναι καθορισμένα, ενώ ο NetPull προσπαθεί να μετακινεί τα άκρα για να πετύχει τη μείωση της κατανάλωσης.

4.4.3 MANTIS

Το MANTIS είναι ένα πολυνηματικό κατανεμημένο λειτουργικό σύστημα, το οποίο με γενικευμένο single-board υλικό, διευκολύνει τη γρήγορη και ευέλικτη ανάπτυξη εφαρμογών. Ο κεντρικός στόχος είναι η διευκόλυνση των προγραμματιστών. Έτσι χρησιμοποιεί την κλασική πολυεπίπεδη πολυνηματική δομή και τυπική γλώσσα προγραμματισμού. Η πολυεπίπεδη δομή του περιέχει πολυνηματισμό, preemptive χρονοπρογραμματισμό με κύλιση χρόνου, I/O συγχρονισμό μέσω αμοιβαίου αποκλεισμού, μια στοίβα πρωτοκόλλων δικτύου και οδηδούς για συσκευές. Ο τρέχων πυρήνας του Magnet υλοποιεί τις παραπάνω συναρτήσεις σε λιγότερο από 500 bytes RAM. Ο ίδιος καθώς και το API υλοποιούνται στη γλώσσα προγραματισμού standard C.

Στην τρέχουσα υλοποίηση του MANTIS, το μέγεθος της RAM που δεσμεύεται για

κάθε νήμα είναι σταθερό. Ο πίνακας των νημάτων, που αποθηκεύεται σε μια καθολική δομή δεδομένων και έχει χωρητικότητα τεσσάρων αντικειμένων, το καθένα από τα οποία είναι 10 bytes και χρησιμοποιείται για να αποθηκεύσει πληροφορίες σχετικά με τα νήματα. Ο χρονοπρογραμματιστής των νημάτων του MANTIS είναι βασισμένος σε προτεραιότητες και round robin στο κάθε επίπεδο προτεραιοτήτων. Επιπλέον, ο χρονοπρογραμματιστής πυροδοτείται μόνο από διακοπές χρονισμού από το υλικό για να εκτελέσει context switching. Οι υπόλοιπες διακοπές εξυπηρετούνται από τους οδηγούς των συσκευών.

Η στοίβα των πρωτοκόλλων δικτύου αποτελείται από τέσσερα επίπεδα: εφαρμογής, δικτύου, MAC, και φυσικό επίπεδο. το MANTIS υλοποιεί τα παραπάνω σαν ένα ή περισσότερα νήματα επιπέδου-χρήστη, κάτι που επιτρέπει ένα trade-off μεταξύ ευελιξίας και απόδοσης. Η στοίβα δικτύου υλοποιείται με ένα βασικό API μεταξύ των επιπέδων. Το MANTIS υλοποιεί την πλημμύρα ως ένα πρωτόκολλο δρομολόγησης και ένα απλό πρωτόκολλο παύσησ-αναμονής για έλεγχο ροής και συμφόρησης. Το συνολικό μέγεθος του κώδικα του πυρήνα, του του χρονοπρογραμματιστή και της στοίβας πρωτοκόλλου δικτύου είναι λιγότερο από 500 bytes και 14kB flash. Το MANTIS υποστηρίζει κάποια πιο εξελιγμένα χαρακτηριστικά, όπως είναι το πολυ-μοντελικό περιβάλλον πρωτοτύπων για δοκιμές στις εφαρμογές των δικτύων αισθητήρων, δυναμικό, δυαδικό, βασισμένο σε ενημερώσεις προγραμματισμό και ένα απομακρυσμένο εξυπηρετητή που διευκολύνει το χρήστη στη σύνδεσή του με το σύστημα και στην επιτήρηση της μνήμης και της κατάστασης των κόμβων αισθητήρων.

4.4.4 OSPM

Το OSPM(ή dynamic power management, DPM - δυναμική διαχείριση ενέργειας) είναι προσανατολισμένο σε τεχνικές διαχείρισης ενέργειας. Η γενικευμένη διαχείριση ενέργειας βασίζεται σε ένα άπληστο αλγόριθμο που θέτει το σύστημα σε κατάσταση ύπνου όσο είναι ανενεργό. Λαμβάνει υπόψιν τους παρακάτω παράγοντες:

- Η μεταφορά σε κατάσταση ύπνου έχει το overhead της διαδικασίας αποθήκευσης της κατάστασης του επεξεργαστή και της αποσύνδεσης από τον παροχέα ενέργειας.

- Η επαναφορά από την κατάσταση ύπνου απαιτεί κάποιον πεπερασμένο χρόνο για να ολοκληρωθεί.
- Όσο πιο βαθειά είναι η κατάσταση ύπνου, τόσο λιγότερη είναι η κατανάλωση ενέργειας, ενώ ο χρόνος αφύπνισης θα είναι μεγαλύτερος.

Επομένως, βασιζόμενο σε ένα συγκεκριμένο μοντέλο άφιξης μηνυμάτων, χρόνου αλλαγής κατάστασης και ρυθμού κατανάλωσης ενέργειας, μειώνει τα αποθέματα ενέργειας. Αν οι τιμές των αποθεμάτων είναι σε θετικά επίπεδα, τότε προκαλεί μια αλλαγή κατάστασης· διαφορετικά διατηρείται η τρέχουσα κατάσταση. Αυτός ο προσαρμοστικός αλγόριθμος τερματισμού αποτελεί ένα trade-off μεταξύ διατήρησης ενέργειας και το κόστος των καθυστερήσεων και πιθανώς των χαμένων μηνυμάτων.

4.4.5 EYES OS

Όπως επισημάνθηκε στην αρχή του κεφαλαίου, ένα λειτουργικό σύστημα για WSNs θα πρέπει να είναι πολύ μικρό ως προς τον κώδικα και τις απαιτήσεις για μνήμη, θα πρέπει να λαμβάνει υπόψιν θέματα ενέργειας ενώ θα πρέπει να διευκολύνει τον κατανεμημένο υπολογισμό και την επαναδιαμόρφωση. Το EYES OS χρησιμοποιεί ένα οδηγούμενο-από-τα-γεγονότα μοντέλο και έναν μηχανισμό με εργασίες (tasks) για να πραγματοποιήσει τις παραπάνω απαιτήσεις. Λειτουργεί με την εξής απλή ακολουθία: πραγματοποιεί ένα υπολογισμό, επιστρέφει μια τιμή και εισέρχεται σε κατάσταση ύπνου. Ένα task μπορεί να χρονοπρογραμματιστεί χρησιμοποιώντας είτε μια πολιτική FIFO, είτε βάση προτεραιοτήτων ή με μια προσέγγιση που βασίζεται σε προθεσμίες, και προκαλείται από γεγονότα με ένα non-blocking τρόπο. Το EYES OS ορίζει ένα API τοπικά και για τα δικτυακά μέρη. Το μέρος της τοπικής πληροφορίας παρέχει συναρτήσεις όπως είναι η πρόσβασηστα δεδομένα του αισθητήρα, διαθεσιμότητα πόρων και την κατάστασή τους, καθώς επίσης και ορισμός των παραμέτρων ή των μεταβλητών μέσα στους κόμβους αισθητήρων. Το δικυακό μέρος παρέχει συναρτήσεις για την αποστολή και λήψη δεδομένων και την συλλογή πληροφοριών από το δίκτυο. Συνοψίζοντας, το EYES OS υλοποιεί δύο σύνολα συναρτήσεων: αυτές που μπορούν να εκτελεστούν κατά την εκκίνηση για να φορτώσει τα τμήματα του λογισμικού και αυτά που μπορούν να παρέχουν πληροφορίες εντοπισμού της θέσης του κόμβου.

Το EYES OS προσφέρει επίσης έναν αποδοτικό μηχανισμό κατανομής του κώδικα με τα παρακάτω χαρακτηριστικά:

1. ενημέρωση του κώδικα στον κόμβο αισθητήρα, συμπεριλαμβανομένου του λειτουργικού συστήματος

2. την αξιοπιστία σε περίπτωση απώλειας πακέτων κατά την ενημέρωση

3. τη χρησιμοποίηση όσο δυνατόν λιγότερους πόρους και

4. την προσωρινή διακοπή της εφαρμογής για ένα μικρό διάστημα καα την ενημέρωση.

Η διαδικασία για την κατανομή του κώδικα ακολουθεί τέσσερα βήματα: αρχικοποίση, κατασκευή μιας απεικόνισης του κώδικα, πιστοποίηση και φόρτωση. Υπάρχουν τρεις επιλογές για την ενημέρωση του κώδικα που τρέχει: διαιρεμένη μνήμη, μια προσέγγιση των δύο φάσεων και ενσωματωμένη EEPROM, που χρησιμοποιείται από το EYES OS.

4.4.6 SenOS

Το SenOS είναι ένα λειτουργικό σύστημα που βασίζεται σε μηχανή πεπερασμένων καταστάσεων. Αποτελείται από τρία μέρη:

1. Από τον πυρήνα, που περιέχει την ακολουθία των καταστάσεων και μια ουρά γεγονότων. Η ακολουθία καταστάσεων περιμένει για είσοδο από την ουρά γεγονότων που είναι FIFO.

2. Από ένα πίνακα μεταβάσεων μεταξύ καταστάσεων όπου φυλάσσονται οι πληροφορίες για τις μεταβάσεις αυτές και τις αντίστοιχες συναρτήσεις χειρισμού τους. Κάθε τέτοιος πίναξας προσδιορίζει μια εφαρμογή. Χρησιμοποιώντας πολλαπλούς πίνακες μετάβασης και εναλλάσσοντάς τους, το SenOS υποστηρίζει πολλαπλές εφαρμογές κατά σύγχρονο τρόπο.

3. Μια βιβλιοθήκη των συναρτήσεων χειρισμού των μεταβάσεων. Ένα εισερχόμενο γεγονός θα τοποθετηθεί στην ουρά γεγονότων.Το πρώτο γεγονός στη ουρά χρονοπρογραμματίζεται και προκαλείται μια μετάβαση, με την οποία καλούνται οι σχετικές συναρτήσεις.

Ο πυρήνας και η βιβλιοθήκη των συναρτήσεων βρίσκονται στη flash ROM ενός κόμβου αισθητήρα, ενώ ο πίνακας μεταβάσεων μπορεί να φορτωθεί ή να τροποποιηθεί κατά τη διάρκεια της εκτέλεσης αφού είναι ανεξάρτητος εφαρμογών. Εφόσον το SenOS βασίζεται σε μηχανή πεπερασμένων καταστάσεων, μπορεί πολύ εύκολα να πραγματοποιήσει συγχρονισμό και επαναδιαμόρφωση. Επιπλέον, μπορεί να επεκταθεί για διαχείριση δικτύου.

4.4.7 EMERALDS

Το EMERALDS είναι ένας επεκτάσιμος μικροπυρήνας γραμμένος σε C++ για ενσωματωμένα, πραγματικού χρόνου συστήματα με ενσωματωμένες εφαρμογές που τρέχουν σε αργές διεργασίες (15 με 25 MHz) και με περιορισμένη μνήμη (32 με 128 kB). Υποστηρίζει πολυνηματικές διεργασίες και πλήρη προστασία μνήμης, που χρονοπρογραμματίζονται χρησιμοποιώντας έναν συνδυασμό νωρίτερης - προθεσμίας - πρώτα και μονοτονικού - ρυθμού χρονοπρογραμματιστή. Οι οδηγοί των συσκευών υλοποιούνται σε επίπεδο χρήστη, ενώ ο χειρισμός των διακοπών γίνεται στο επίπεδο του πυρήνα. Το EMERALDS χρησιμοποιεί σημαφόρους και μεταβλητές κατάστασης για το συγχρονισμό και ταυτόχρονα κληρονομικότητα προτεραιοτήτων. Η επικοινωνία εντός του επεξεργαστή βασίζεται στην ανταλλαγή μηνυμάτων, γραμματοκιβώτια και διαμοιρασμό μνήμης, βελτιστοποιημένα για τις ανάγκες των κόμβων αισθητήρων. Το EMERALDS δε χρησιμοποιεί γραμματοκιβώιο αλλά καθολικές μεταβλητές για την ανταλλαγή πληροφοριών μεταξύ των εργασιών, για τηνα αποφυγή της αποστολής μηνυμάτων. Επίσης, δε λαμβάνει υπόψιν θέματα δικτύου.

4.4.8 PicOS

Μια ιδιότητα των λειτουργικών συστημάτων για μικροελεγκτές με περιορισμένη RAM είναι η προσπάθει για δέσμευση όσο το δυνατόν μικρότερη ποσότητα μνήμης για την επεξεργασία ενός νήματος. Το PicOS είναι γραμμένο σε C για μικροελεγκτές με περιορισμένη on-chip RAM. Στο PicOS όλες οι εργασίες μοιράζονται την ίδια καθολική στοίβα και λειτουργούν ως συν-ρουτίνες. Επίσης, κάθε εργασία είναι σαν μια μηχανή πεπερασμένων καταστάσεων όπου η μετάβαση μεταξύ καταστάσεων προκαλείται από

γεγονότα. Η προσέγγιση αυτή είναι αποδοτική για αντιδραστικές εφαρμογές των οποίων ο ρόλος είναι η απόκριση σε γεγονότα κυρίως, παρά η επεξεργασία δεδομένων. Ο κύκλος μηχανής πολυπλέκεται με μεταξύ πολλαπλών εργασιών. Έχει ελάχιστες απαιτήσεις και υποστηρίζει multitasking, μια επίπεδη δομή διεργασιών, ίσως όμως να μην είναι κατάλληλο για εφαρμογές πραγματικού χρόνου.

Κεφάλαιο 5

Μελέτη - Υλοποίηση Κυματικών Αλγορίθμων στο **TinyOS**

Στο παρόν κεφάλαιο θα μελετηθεί η υλοποίηση κάποιων κυματικών αλγορίθμων στο περιβάλλον του TinyOS και με τη βοήθεια της γλώσσας προγραμματισμού nesC. Καθένας από τους αλγορίθμους αποτελεί ένα Module. Τα Modules διασυνδέονται μεταξύ τους με σκοπό την ανάπτυξη μιας ολοκληρωμένης εφαρμογής και έτσι σχηματίζονται τα Configurations. Υπάρχει ένα Configuration για κάθε εφαρμογή. Η υλοποίηση των παραπάνω συνιστωσών λογισμικού (Components) συμπεριλαμβάνει όλα όσα αναφέρθηκαν στο προηγούμενο κεφάλαιο (διεπαφές, εντολές, γεγονότα, tasks) και τα οποία συνιστούν το TinyOS (δες παρ. 4.2).

Καταρχάς, υλοποιείται ο αλγόριθμος Echo(δες παρ. 3.2). Για να λειτουργήσει ο αλγόριθμος απαιτείται η γνώση κάθε κόμβου για τους γειτονικούς του κόμβους. Με βάση την απαίτηση αυτή και με σκοπό την εύρεση των γειτόνων κάθε κόμβου υλοποιήθηκαν οι αλγόριθμοι αναζήτησης πρώτα-κατά-βάθος (DFS) και αναζήτησης πρώτα-κατά-εύρος (BFS), καθώς επίσης και ένας απλός αλγόριθμος πλημμύρας. Ο χρήστης μπορεί να επιλέξει ένα από τους τρεις αυτούς αλγορίθμους για να τον χρησιμοποιήσει μαζί με τον αλγόριθμο Echo. Έτσι, αν επιλεγεί κάποιος από τους αλγορίθμους αναζήτησης, οι οποίοι δημιουργούν επικαλυπτικά δέντρα πάνω στο δίκτυο, τότε ο αλγόριθμος Echo θα στέλνει τα μηνύματά του πάνω στη δενρική δομή που δημιουργήθηκε.

Για την ανάλυση και δοκιμή των αλγορίθμων, χρησιμοποιήθηκε ο TOSSIM, ο ενσωματωμένος προσομοιωτής του TinyOS.

5.1 Ο προσομοιωτής του TinyOS, TOSSIM

Ο TOSSIM είναι ένας προσομοιωτής για δίκτυα αισθητήρων που βασίζονται στο λειτουργικό σύστημα TinyOS. Με τη βοήθεια αυτού, οι χρήστες μπορούν να αναλύουν και να δοκιμάζουν αλγορίθμους πριν τους χρησιμοποιήσουν σε πραγματικό περιβάλλον. Ο TOSSIM, βέβαια, δεν προσπαθεί να προσομοιώσει το πραγματικό περιβάλλον αλλά να δώσει όσο το δυνατόν πιο αξιόπιστα αποτελέσματα για εφαρμογές του TinyOS και για το λόγο αυτό, επικεντρώνεται στην προσομοίωση της λειτουργίας του TinyOS. Κάποια από τα χαρακτηριστικά του παρουσιάζονται παρακάτω ενώ στο σχήμα 5.1 φαίνεται ένα παράδειγμα εκτέλεσής του.

- **Αξιοπιστία** Εκ των πραγμάτων, ο TOSSIM αιχμαλωτίζει τη συμπεριφορά του TinyOS από το πιο χαμηλό ακόμα επίπεδο. Προσομοιώνει κάθε διακοπή, κάθε μεταφορά δεδομένων στο δίκτυο, κάθε ADC λειτουργία.

- **Χρόνος** Αν και ο TOSSIM μπορεί να καταγράψει με ακρίβεια το χρόνο πρόκλησης μιας διακοπής, εντούτοις δε μοντελοποιεί το χρόνο εκτέλεσης του κώδικα, αλλά θεωρεί ότι εκτελείται στιγμιαία.

- **Μοντέλα** Όπως αναφέρθηκε και παραπάνω, ο TOSSIM δεν προσπαθεί να προσομοιώσει το πραγματικό περιβάλλον. Παρόλα αυτά, παρέχει μια αφαιρετικότητα για συγκεκριμένα φαινόμενα του πραγματικού κόσμου, όπως είναι τα σφάλματα κατά την μετάδοση. Χρησιμοποιώντας τα εργαλεία αυτά οι χρήστες μπορούν να μοντελοποιήσουν τις καταστάσεις που επιθυμούν. Έτσι και ο TOSSIM παραμένει ευέλικτος, χωρίς να απαιτείται να δημιουργήσει τις 'σωστέσ' συνθήκες και η διαδικασία προσομοίωσης παραμένει απλή και αποδοτική.

  - *Ραδιομετάδοση* Ο TOSSIM δεν παρέχει μοντέλο για τη ραδιομετάδοση. Αντί αυτού παρέχει ένα αφαιρετικό μοντέλο μετάδοσης αναξάρτητων λαθών μεταξύ

δύο κόμβων. Ένα εξωτερικό πρόγραμμα μπορεί να παράσχει ένα μοντέλο της ραδιομετάδοσης το οποίο να αντιστοιχηθεί σε αυτά τα λάθη

– *Ενέργεια* Ο TOSSIM δεν παρέχει μοντέλο για την κατανάλωση ενέργειας. Ωστόσο είναι απλό να συλλέξει κανείς σχετικές πληροφορίες από τα components που καταναλώνουν ενέργεια.

- **Ανάπτυξη κώδικα** Ο TOSSIM αναπτύσσεται απέυθείας από τον κώδικα του TinyOS. Για να προσομοιώσει κανείς ένα πρωτόκολλο θα πρέπει να το υλοποιήσει στο TinyOS. Από τη μια πλευρά αυτό είναι πι δύσκολο από μια αφαιρετική υλοποίηση, από την άλλη όμως, ο κώδικας αυτός μπορεί να εφαρμοστεί άμεσα (ίσως με κάποιες τροποποιήσεις) σε πραγματικές συσκευές.

- **Ελαττώματα** Παρά το γεγονός ότι ο TOSSIM συλλαμβάνει τη συμπεριφορά του TinyOS σε πολύ χαμηλό επίπεδο, ωστόσο κάνει κάποιες υποθέσεις απλούστευσης. Έτσι είναι πολύ πιθανό μια εφαρμογή που λειτουργεί σωστά κατά την προσομοίωση, να μην δουλεύει αν εφαρμοστεί σε πραγματικές συσκευές. Ένα τέτοιο παράδειγμα είναι ότι οι διακοπές στον TOSSIM δεν είναι preemptive. Συνεπώς, αν το γεγονός αυτό μπορεί να προκαλέσει την παύση της λειτουργίας σε μια συσκευή, κατά την προσομοίωση μπορεί να μη συμβεί κάτι τέτοιο. Επίσης, αν οι εξυπηρετητής των διακοπών χρειάζονται πολύ χρόνο για να ολοκληρωθούν, τότε μια συσκευή ίσως καταρρεύσει· στην προσομοίωση όμως δε θα προκληθεί κάποιο πρόβλημα, αφού ο κώδικας στον TOSSIM εκτελείται στιγμιαία.

- **Δικτύωση** Προς το παρόν, ο TOSSIM προσομοιώνει την 40Kbit RFM mica στοίβα δικτύου, συμπεριλαμβανομένου τις MAC, τις ενημερώσεις κωδικοποίησης και συγχρονισμού.

- **Εγκυρότητα** Οι εφαρμογές του TinyOS στον πραγματικό κόσμο έχουν δείξει ότι τα δίκτυα του TinyOS είναι περίπλοκα και η συμπεριφορά τους συνεχώς μεταβαλλόμενη. Επομένως, ενώ ο TOSSIM είναι αρκετά καλός για μια πρώτη αίσθηση της λειτουργίας των αλγορίθμων και της μεταξύ τους σύγκρισης, ωστόσο δε θα πρέπει να εκλαμβάνεται ως έγκυρη πηγή πληροφοριών.

Time (4MHz ticks)	Action
3987340	Simulator event is dequeued and handled at time 3987340.
	The clock interrupt handler is called, signaling the application Timer event.
	The application's Timer handler requests a reading from the ADC.
	The ADC component puts a simulator ADC event on the queue with timesta mp 3987540.
	The interrupt handler completes; the clock event re-enqueues itself fo r the next tick.
3987540	Simulator ADC event is dequeued and handled at time 3987540.
	The ADC interrupt handler is called, signaling an ADC ready event with a sensor value.
	The application event handler takes the top three b its and calls LEDs commands.
	The ADC interrupt handler completes.
7987340	Simulator event is dequeued and handled at time 7987340.
	The clock interrupt handler is called, signaling the application Timer event.
	. . . execution continues as above

Σχήμα 5.1: Παράδειγμα εκτέλεσης στον TOSSIM

Στα επόμενα θα δούμε αναλυτικά πώς υλοποιούνται οι αλγόριθμοι στο περιβάλλον του TinyOS.

5.2 Διεπαφές

Οι διεπαφές που χρησιμοποιούνται είναι οι εξής:

- interface StdControl
- interface Timer
- interface SendMsg
- interface ReceiveMsg
- interface Neighborhood

StdControl Είναι η καθιερωμένη διεπαφή ελέγχου του TinyOS. Όλες οι συνιστώσες λογισμικού που απαιτούν αρχικοποίηση ή που μπορούν να σταματησουν να λειτουργούν παρέχουν τη διεπαφή αυτή. Η StdControl παρέχει τρεις εντολές: αρχικοποίηση, εκκίνηση, παύση.

```
interface StdControl{
  command result_t init();
  command result_t start();
  command result_t stop();
}
```

Timer Η διεπαφή αυτή παρέχει έναν γενικευμένο χρονιστή που χρησιμοποιείται για να παράγει γεγονότα σε συγκεκριμένες χρονικές στιγμές. Ορίζει δύο εντολές για έναρξη και παύση του χρονιστή και ένα γεγονός που συμβαίνει όταν ο χρονιστής 'χτυπά', όταν συμπηρώνεται ο χρόνος που έχει οριστεί.

```
interface Timer{
  command result_t start(char type, uint32_t interval);
  command result_t stop();
   event resut_t fired();
}
```

SendMsg Οι αλγόριθμοι που υλοποιήθηκαν λειτουργούν με τη βοήθεια των Active Messages. Στο μοντέλο των Active Messages ορίζεται για κάθε πακέτο που κινείται στο δίκτυο έναν ακέραιο αριθμό που αποτελεί το ID του μηνύματος και το χαρακτηρίζει. Όταν το πακέτο φτάνει στον δέκτη προκαλείται ένα γεγονός που εξυπηρετεί αυτού του τύπου τα πακέτα.

Η SendMsg διεπαφή χρησιμοποιείται για την αποστολή AM μηνυμάτων. Ορίζει μια εντολή για την αποστολή του μηνύματος και ένα γεγονός που προκαλείται όταν ολοκληρώνεται η αποστολή.

```
interface SendMsg{
  command result_t send(uint16_t address, uint8_t length,
  TOS_MsgPtr msg);
  event resut_t sendDone(TOS_MsgPtr msg, result_t success);
}
```

Τα μηνύματα αποτελούνται από δύο μέρη, την επικεφαλίδα και το κυρίως μέρος. Η δομή ενός μηνύματος αποτελείται από τα παρακάτω στοιχεία:

- Διεύθυνση παραλήπτη
- Τύπος
- Ομάδα
- Μέγεθος
- Κυρίως μέρος
- crc

ReceiveMsg Η διεπαφή ReceiveMsg είναι αρμόδια για τη λήψη των μηνυμάτων και έτσι παρέχει το αντίστοιχο γεγονός.

```
interface ReceiveMsg{
 event TOS_MsgPtr  receive(TOS_MsgPtr msg);
}
```

Neighborhood Η διεπαφή αυτή παρέχει πληροφορίες για τη ‘γειτονιά’ ενός κόμβου. Ορίζει δύο εντολές οι οποίες δίνουν τον αριθμό των γειτονικών κόμβων και τους ίδιους τους γείτονες.

```
interface Neighborhood{
 async command uint16_t getSize();
 async command uint16_t * getNeighbors();
}
```

5.3 Modules

Όπως αναφέρθηκε στην αρχή του κεφαλαίου, έχει οριστεί ένα module για κάθε αλγόριθμο που υλοποιήθηκε. Έτσι έχουμε τα εξής modules:

- module DFSM
- module BFSM

- module NeighborsDetectionM
- module EchoM

5.3.1 Το BFSM module

Το module αυτό υλοποιεί τον αλγόριθμο αναζήτησης πρώτα-κατά-εύρος· κατασκευάζει ένα γεννητικό δέντρο όπου κάθε κόμβος γνωρίζει τους γείτονές του τους οποίους μπορεί να επιστρέψει μέσω της διεπαφής *Neighborhood*. Το BFSM λοιπόν, παρέχει τις διαπαφές StdControl, Neighborhood και χρησιμοποιεί τις SendMsg, ReceiveMsg, Timer.

```
module BFSM{
  provides {
    interface StdControl;
    interface Neighborhood;
  }
  uses {
    interface SendMsg;
    interface ReceiveMsg;
    interface Timer as BfsTimer;
  }
}
```

Εντολές Κάθε κόμβος του δικτύου χαρακτηρίζεται από μια συγκεκριμένη διεύθυνση. Κατά την αρχικοποίηση του αλγορίθμου ορίζουμε τις αποστάσεις του κάθε κόμβου από τον κόμβο - αρχικοποιητή, που θα είναι η ρίζα του δέντρου. Έτσι, κατά την υλοποίηση της εντολής *StdControl.init*, ανατίθεται σε κάθε κόμβο η διεύθυνση του TOS_LOCAL_ADDRESS και επιπλέον, αν ο κόμβος έχει τη μηδενική διεύθυνση ορίζουμε την απόσταση d = 0 και τον parent ίσο με 0 επίσης, αλλιώς τη θέτουμε ίση με μια αρκετά μεγάλη τιμή· ομοίως και για τη μεταβλητή parent.

```
//initialize component
   command result_t StdControl.init(){
      atomic{
```

```
      m_ID = TOS_LOCAL_ADDRESS;
      m_totChildren = 0;

      if(m_ID == 0){
        d = 0;
        parent = m_ID;
      }
      else{
        d = 1000; //some max value
        parent = 1000;
      }
    }
    return SUCCESS;
  }
```

```
//initialize component
  command result_t StdControl.init(){
    atomic{
      m_ID = TOS_LOCAL_ADDRESS;
      m_totChildren = 0;

      if(m_ID == 0){
        d = 0;
        parent = m_ID;
      }
      else{
        d = 1000; //some max value
        parent = 1000;
      }
    }
    return SUCCESS;
  }
```

Στην υλοποίηση της εντολής *StdControl.start* εκκινούμε τον χρονιστή, ο οποίος θα

‘χτυπά’ κάθε 1000 msec.

```
//Start - Set Timer to fire after 1000ms
  command result_t StdControl.start(){
    return call BfsTimer.start(TIMER_REPEAT,1000);
  }
```

Κατόπιν, πρέπει να υλοποιηθεί η εντολή *StdControl.stop*. Εκεί, αυτό που γίνεται είναι η παύση του χρονιστή.

```
//Start - Set Timer to fire after 1000ms
  command result_t StdControl.stop(){
    return call BfsTimer.stop();
  }
```

Η εντολή *Neighborhood.getNeighbors* όπως αναφέρθηκε και παραπάνω, επιστρέφει τους γείτονες ενός κόμβου. Κατά τη διάρκεια εκτέλεσης του αλγορίθμου, όσοι γείτονες ανακαλύπτονται αποθηκεύονται στη δομή *children_array*, που είναι ουσιαστικά μια διασυνδεδεμένη λίστα από αντικείμενα τύπου NodeNeighbor. Η εντολή Neighborhood.getNeighbors χρησιμοποιεί τη βιβλιοθήκη *sglib* και με τη βοήθεια της μακροεντολής SGLIB_LIST_MAP_ON_ELEMENTS διατρέχει τη λίστα και τοποθετεί τους κόμβους σε ένα πίνακα, τον οποίο επιστρέφει η εντολή.

```
//Return all children-neighbors
  async command uint16_t * Neighborhood.getNeighbors(){
    uint16_t i=0;
    uint16_t *children;

    if(m_totChildren <1)
       return NULL;

    /* allocate memory */
    atomic children= (uint16_t*)malloc(sizeof(uint16_t)
    * m_totChildren);
    atomic memset(children, 0, sizeof(uint16_t)
    * m_totChildren);
```

```
    /* Update array */
    SGLIB_LIST_MAP_ON_ELEMENTS(NodeNeighbor, children_array,
    node,next_ptr, {children [i++] = node->id;});

    return children;
  }
```

Η επόμενη εντολή της διεπαφής Neighborhood που πρέπει να υλοποιηθεί στο BFS module είναι η *Neighborhood.getSize*, η οποία επιστρέφει τον αριθμό των γειτονικών κόμβων. Η τιμή αυτή ανανεώνεται κατά τη λειτουργία του αλγορίθμου.

```
//Return the number of children-neighbors
  async command uint16_t Neighborhood.getSize(){
    return m_totChildren;
 }
```

Γεγονότα Οι τρεις από τις διεπαφές που χρησιμοποιεί το BFS module όπως είδαμε ορίζουν τρία γεγονότα. Το γεγονός *BfsTimer.fired* προκαλεί την επαναλαμβανόμενη εκκίνηση του αλγορίθμου. Όταν ο χρονιστής 'χτυπήσει', ο αρχικοποιητής, που στη περίπτωσή μας είναι κόμβος 0, θα αναγκαστεί να στείλει ένα μήνυμα αναζήτησης στους γειτονικούς του κόμβους.

```
event result_t BfsTimer.fired(){
    if (m_ID==0)
        post sendSearchMsg();
    return SUCCESS;
}
```

Η λήψη ενός μηνύματος γίνεται με τη βοήθεια του γεγονότος *ReceiveMsg.receive*. Μόλις ένας κόμβος λάβει ένα μήνυμα ξεκινάει τους ελέγχους που ορίζει ο αλγόριθμος, καλώντας το αντίστοιχο task.

```
 // Process a message received
 event TOS_MsgPtr ReceiveMsg.receive(TOS_MsgPtr recv_packet) {
   ChannelsMsg * msgdata = (ChannelsMsg*) recv_packet->data;
```

```
 /* ...declarations... */

 dbg(DBG_TEMP, "Received msg from %d with distance %d\n",
  sender, sent_dist);

/* allocate memory */
newNeighbor = (NodeNeighbor*)malloc(sizeof(NodeNeighbor));
memset(newNeighbor, 0, sizeof(NodeNeighbor));

newNeighbor->id = sender;

//Check if it is message from a child
if(sender_parent == myID){
       //dbg(DBG_TEMP, "Trying to add child\n");
       SGLIB_SORTED_LIST_ADD_IF_NOT_MEMBER(NodeNeighbor,
   children_array, newNeighbor,Neighbor_COMPARATOR, next_ptr,
   member);

   if(member!=NULL){
         //dbg(DBG_TEMP, "Already in childrens\n");
       }else {
        atomic{
         m_totChildren = m_totChildren + 1;
         count = m_totChildren;
        }
       }
     }
  //Check if this is a message from our parent
  else if(my_parent == sender){
    //Check if parent has changed height
        if(my_dist != sent_dist + 1)
      //Update distance
```

```
            atomic d = sent_dist + 1;
        }
    //Check if this is a message from a node closer to the
    //root
    else if(sent_dist + 1<my_dist){
        SGLIB_SORTED_LIST_DELETE_IF_MEMBER(NodeNeighbor,
        children_array, newNeighbor, Neighbor_COMPARATOR,
        next_ptr, member);

            if (member != NULL) {
                atomic {
                  if(m_totChildren > 0)
                    m_totChildren = m_totChildren -1;
                }
            }
            atomic{
                d = sent_dist + 1;
        //Change parent (new parent is closer)
                parent = sender;
                count = m_totChildren;
                }
        }
    else{
      SGLIB_SORTED_LIST_DELETE_IF_MEMBER(NodeNeighbor,
      children_array, newNeighbor,Neighbor_COMPARATOR,
      next_ptr, member);

      // check if child already in list
      if (member != NULL) {
                atomic {
                  if(m_totChildren > 0)
                     m_totChildren = m_totChildren -1;
                     count = m_totChildren;
```

```
                }
            }
        }
        return recv_packet;
}
```

Το τελευταίο γεγονός που πρέπει να υλοποιηθεί είναι το *SendMsg.sendDone*, το οποίο απλώς επιστρέφει ‘Επιτυχία’, για να δείξει ότι ολοκληρώθηκε η αποστολή του μηνύματος· είναι η αντίδραση στην εντολή SendMsg.send. Όπως αναφέρθηκε στο προηγούμενο κεφάλαιο (δες παρ. 4.3), η εντολή SendMsg.send και το γεγονός SendMsg.sendDone αποτελούν τα δύο μέλη του split-phased πακέτου SendMsg.

```
//handles the sendDone event
event result_t SendMsg.sendDone(TOS_MsgPtr msg, bool success){
    return SUCCESS;
 }
```

Tasks Στα tasks γίνονται όλες οι υπολογιστικές εργασίες των αλγορίθμων. Τα tasks τοποθετούνται σε μια στοίβα και ακολουθείται FIFO πολιτική για την εκτέλεσή τους. Κάθε task εκτελείται ανεξάρτητα από τα υπόλοιπα και μπορεί να χρησιμοποιηθεί για την αποφυγή συνθηκών συναγωνισμού.

Στο BFS module η εντολή αποστολής ενός μηνύματος καλείται μέσα από το task *sendSearchMsg*, αφού συμπληρωθούν τα πεδία του μηνύματος

```
//Send message to Neighbors
task void sendSearchMsg(){
    result_t r;
    ChannelsMsg *msgdata = (ChannelsMsg*) m_msg.data;

    //set message contents
    atomic {
        msgdata->id = m_ID;
        msgdata->dist = d;
        msgdata->parent = parent;
```

```
    }

    //try to send the message
    r = call SendMsg.send(TOS_BCAST_ADDR, sizeof(ChannelsMsg),
    &m_msg);
    if(r)
        dbg(DBG_TEMP,"\nBFS: Sending my id (%d)
        with parent %d\n", msgdata->id, msgdata->parent);
  }
```

5.3.2 Το DFS module

Το module αυτό υλοποιεί τον αλγόριθμο αναζήτησης πρώτα-κατά-βάθος· κατασκευάζει ένα γεννητικό δέντρο όπου κάθε κόμβος γνωρίζει τους γείτονές του τους οποίους μπορεί να επιστρέψει μέσω της διεπαφής *Neighborhood*. Το DFSM λοιπόν, παρέχει τις διαπαφές StdControl, Neighborhood και χρησιμοποιεί τις SendMsg, ReceiveMsg, Timer. Επιπλέον, επειδή απαιτείται η γνώση της πληροφορίας της γειτονιάς για να ξεκινήσει ο αλγόριθμος, αρχικά ο κάθε κόμβος συλλέγει αυτή την πληροφορία χρησιμοποιώντας το NeighborsDetection module.

```
module DFSM {
  provides {
    interface StdControl;
    interface Neighborhood;
  }
  uses {
    interface Neighbors;
    interface SendMsg as SendSearch;
    interface ReceiveMsg as ReceiveSearch;
    interface Timer as SearchTimer;
    interface Timer as SendTimer;
    interface Timer as ReckonTimer;
  }
```

```
}
```

Όπως και στην υλοποίηση του αλγορίθμου BFS, ανταλλάσσονται μηνύματα μεταξύ των κόμβων και με διάφορους ελέγχους συλλέγεται τελικά η πληροφορία των παιδιών του κάθε κόμβου. Η πληροφορία αυτή επιστρέφεται στις εντολές *Neighborhood.getNeighbors* και *Neighborhood.getSize*, με τον ίδιο τρόπο όπως στο BFSM.

Εντολές Καταρχάς υλοποιείται η εντολή *StdControl.init*, η οποία στη συγκεκριμένη περίπτωση δεν επιτελεί καμιά ιδιαίτερη λειτουργία. Επιστρέφει πάντα 'Επιτυχία'.

```
//Initialize the component.
command result_t StdControl.init() {
  return SUCCESS;
}
```

Στην εντολή *StdControl.start* 'ξυπνάει' ο αρχικοποιητής του αλγορίθμου και γίνεται επίσης, η εκκίνηση των χρονιστών. Πιο κάτω θα δούμε τι γίνεται όταν 'χτυπάει' καθένας από τους χρονιστές.

```
//Start things up.  This just sets the rate for the clock
//component.
command result_t StdControl.start() {
/* Read neighborhood information from Reckon module
 * after DELAY_RECKON_RESULT ms
 * for the module to have enough time to collect all
 * the available information
 */
 call ReckonTimer.start(TIMER_ONE_SHOT, DELAY_RECKON_RESULT);

 if((TOS_LOCAL_ADDRESS==0)&&(m_parentID==DFS_MAX_NODE_ID))
 {
  /* DFS root wakes up */
  m_parentID = 0;

  /* Start exploring after DELAY_DFS_SEARCH ms
```

```
     * so that complete neighborhood information is available
     */
     call SearchTimer.start(TIMER_ONE_SHOT, DELAY_DFS_SEARCH);
    }

    call SendTimer.start(TIMER_REPEAT, INTERVAL_SEND_MSG);

    return SUCCESS;
  }
```

Η εντολή *StdControl.stop* απλά απενεργοποιεί τους δύο χρονιστές και παύει τη λειτουργία της εφαρμογής.

```
  /* Halt execution of the application.
  * This just disables the clock component.*/
  command result_t StdControl.stop() {
    call SendTimer.stop();
    return call SearchTimer.stop();
  }
```

Η εντολή *Neighborhood.getNeighbors* όπως αναφέρθηκε και παραπάνω, επιστρέφει τους γείτονες ενός κόμβου. Κατά τη διάρκεια εκτέλεσης του αλγορίθμου, όσοι γείτονες ανακαλύπτονται αποθηκεύονται στη δομή *m_children*, που είναι ουσιαστικά μια διασυνδεδεμένη λίστα από αντικείμενα τύπου DFSChild. Η εντολή Neighborhood.getNeighbors χρησιμοποιεί τη βιβλιοθήκη *sglib* και με τη βοήθεια της μακροεντολής SGLIB_LIST_MAP_ON_ELEMENTS διατρέχει τη λίστα και τοποθετεί τους κόμβους σε ένα πίνακα, τον οποίο επιστρέφει η εντολή.

```
//Return all children-neighbors
async command uint16_t * Neighborhood.getNeighbors(){
 uint16_t i=0;
 uint16_t *children;
 uint16_t totChild;
```

```
atomic totChild = m_totChildren;

if(totChild <1)
      return NULL;

/* allocate memory */
atomic children = (uint16_t*)malloc(sizeof(uint16_t)
* m_totChildren);

atomic memset(children, 0, sizeof(uint16_t)
* m_totChildren);

/* Update array */
SGLIB_LIST_MAP_ON_ELEMENTS(DFSChild, m_children,node,
next_ptr,{children [i++] = node->id;});

return children;
}
```

Η επόμενη εντολή της διεπαφής Neighborhood που πρέπει να υλοποιηθεί είναι η *Neighborhood.getSize*, η οποία επιστρέφει τον αριθμό των γειτονικών κόμβων. Η τιμή αυτή ανανεώνεται κατά τη λειτουργία του αλγορίθμου.

```
// Return the number of children-neighbors
async command uint16_t Neighborhood.getSize(){
      uint16_t totChild;

      atomic totChild = m_totChildren;
      return totChild;
}
```

Γεγονότα Τα γεγονότα που υλοποιούνται στο DFS module προκαλούνται είτε από τους χρονιστές του είτε από τη διεπαφή της αποστολής μηνυμάτων, όπως συμβαίνει και στο BFS module. Το γεγονός *ReckonTimer.fired* συλλέγει την αρχική πληροφορία

σχετικά με τους γείτονες του κάθε κόμβο. Αν το μέγεθος της γειτονιάς είναι μικρότερο από ένα κατώφλι, η διαδικασία συλλογής πληροφοριών επαναλαμβάνεται. Όλοι οι γείτονες τοποθετούνται στη λίστα με τους ανεξερεύνητους γείτονες και κατά την εκτέλεση του αλγορίθμου ένας ένας απομακρύνονται από αυτή.

```
// Respond to the <code>SearchTimer.fired</code> event
//by counting the children
event result_t ReckonTimer.fired() {
  uint16_t nsize = call Neighbors.getSize();

  if(nsize<MINIMUM_BEACON)
       return call ReckonTimer.start(TIMER_ONE_SHOT,
    DELAY_RECKON_RESEARCH);

  if (nsize > 0) {
    uint16_t *narray = call Neighbors.getNeighbors();
    uint16_t i = 0;

    /* Iterate through all neighbors **/
    for(i = 0; i < nsize; i++) {
      unexploredAdd(narray[i]);
    }

    dbg(DBG_USR1, "MyPlotName PLOT: add point x:
    %d y: %d plotStyle: dots\n", TOS_LOCAL_ADDRESS, nsize);

  } else
    /* Not enough data were collected
     * (or network is partitioned) **/
    return call ReckonTimer.start(TIMER_ONE_SHOT,
    DELAY_RECKON_RESEARCH);

  return SUCCESS;
}
```

Όταν χτυπήσει ο επόμενος χρονιστής *SearchTimer.fired* ξεκινάει η διαδικασία εξερεύνησης του αλγορίθμου.

```
// Respond to the <code>SearchTimer.fired</code>
//event by starting the DFS construction
event result_t SearchTimer.fired() {

  /* DFS root starts exploring **/
  post explore();

  return SUCCESS;
}
```

Στο σημείο αυτό πρέπει να αναφέρουμε ότι τα προς αποστολή μηνύματα τοποθετούνται σε μια λίστα. Έτσι, η δουλειά του τρίτου και τελευταίου χρονιστή *SendTimer.fired* είναι να διατάζει την αποστολή του μηνύματος που έχει σειρά, καλώντας το task sendNextMessage.

```
// Respond to the <code>SendTimer.fired</code>
//event by sending out an unsent message
event result_t SendTimer.fired() {

  if (m_unsent != NULL)
    post sendNextMessage();

  return SUCCESS;
}
```

Η λήψη ενός μηνύματος γίνεται με τη βοήθεια του γεγονότος *ReceiveMsg.receive*. Μόλις ένας κόμβος λάβει ένα μήνυμα ξεκινάει τους ελέγχους που ορίζει ο αλγόριθμος.

```
  /* Process a search message received
   *  from the neighborhood based on the state
   * of the local structure variables */
event TOS_MsgPtr ReceiveSearch.receive(TOS_MsgPtr recv_packet) {
  DFSMsgSearch * search = (DFSMsgSearch *) recv_packet->data;
```

```
/* Overhear nearby transmissions */
if (search->targetid != TOS_LOCAL_ADDRESS) {
 /* Remove node from list of unexplored nodes **/
 unexploredDel(search->srcid);
 return recv_packet;
}

/* Confirm received messages */
dbg(DBG_TEMP, "DFS-SEARCH-RCVD\t%d\t%d\n",
search->srcid, search->msgtype);

/* Examine message type **/
switch (search->msgtype) {
 /* Check if this is a search message **/
  case DFS_MSG_SEARCH:

 /* Node has not received a search message before **/
  if (m_parentID == DFS_MAX_NODE_ID) {
   /* Set parent to node id that sent the search message **/
   m_parentID = search->srcid;

   /* Add Edge in Debug-Graph (TOSSIM Plugin) **/
   dbg(DBG_USR1, "MyGraphName DIRECTED GRAPH: add edge
   %d direction:backward \n", m_parentID);

   /* Remove node from list of unexplored nodes **/
   unexploredDel(m_parentID);

   /* Start exploring neighborhood **/
   post explore();
  } else {
    /* Already in DFS tree **/
```

```
      messageAdd(search->srcid, DFS_MSG_ALREADY);

      /* Remove node from list of unexplored nodes **/
      unexploredDel(search->srcid);
    }

     break;

    /* Check if this is a parent message **/
    case DFS_MSG_PARENT:
      childAdd(search->srcid);
      post explore();
      break;

    /* Check if this is an already message **/
    case DFS_MSG_ALREADY:
      post explore();
      break;

    }
    return recv_packet;
  }
```

Στην παραπάνω υλοποίηση, για να διαγραφεί ένας κόμβος από τη λίστα των ανεξερεύνητων κόμβων χρησιμοποιείται η συνάρτηση *unexploredDel* ενώ για καλείται και η συνάρτηση *childAdd* για την προσθήκη ενός κόμβου στη λίστα με τα 'παιδιά' του κόμβου που εκτελεί τον αλγόριθμο.

Η συνάρτηση *unexploredDel* έχει ως εξής:

```
// Remove a node from the linked list of
// unexplored nodes
void unexploredDel(uint16_t cid) {
  DFSChild *child, *member;
```

```
  /* allocate memory **/
  child = (DFSChild*)malloc(sizeof(DFSChild));
  memset(child, 0, sizeof(DFSChild));

  /* fix parameters **/
  child->id = cid;

  // remove the element frome the list while keeping
  //it sorted

  SGLIB_SORTED_LIST_DELETE_IF_MEMBER(DFSChild,
  m_unexplored, child, DFSChild_COMPARATOR, next_ptr, member);
}
```

Η συνάρτηση *childAdd* έχει ως εξής:

```
// Add a new node in the linked list of children nodes
void childAdd(uint16_t cid) {
  DFSChild *child, *member;

  /* allocate memory **/
  child = (DFSChild*)malloc(sizeof(DFSChild));
  memset(child, 0, sizeof(DFSChild));

  /* fix parameters **/
  child->id = cid;

  // insert the new element into the list while keeping
  // it sorted

  SGLIB_SORTED_LIST_ADD_IF_NOT_MEMBER(DFSChild, m_children,
  child, DFSChild_COMPARATOR, next_ptr, member);

  /* check if child already in list **/
```

```
    if (member != NULL) {
        free(child);
    } else{
        /*update number of children**/
        atomic m_totChildren++;
    }
  }
```

Το τελευταίο γεγονός που πρέπει να υλοποιηθεί είναι το *SendMsg.sendDone*, το οποίο απλώς επιστρέφει 'Επιτυχία', για να δείξει ότι ολοκληρώθηκε η αποστολή του μηνύματος.

```
//handles the sendDone event
event result_t SendMsg.sendDone(TOS_MsgPtr msg,bool success){
  return SUCCESS;
}
```

Tasks Στο DFS module έχουν υλοποιηθεί δύο tasks. Το πρώτο αφορά την εξερεύνηση της γειτονιάς του κόμβου. Αν υπάρχουν κόμβοι που δεν έχουν εξερευνηθεί, επιλέγει κάποιον από αυτόν και τον 'μαρκάρει', απομακρύνοντάς τον από τη λίστα των ανεξερεύνητων κόμβων.

```
//Explore the neighborhood
  task void explore() {

    if (m_unexplored != NULL) {
       DFSChild *node = m_unexplored;

       /* Send message to node **/
      messageAdd(node->id , DFS_MSG_SEARCH);

       /* Remove node from m_unexplored **/
      m_unexplored = m_unexplored->next_ptr;

      if (m_parentID != TOS_LOCAL_ADDRESS) {
```

```
      /* We are done exploring our neighborhood
       * -- report back to parent **/
       messageAdd(m_parentID, DFS_MSG_PARENT);

     } else if (m_parentID == TOS_LOCAL_ADDRESS) {
       dbg(DBG_TEMP, "DFS-DONE\n");

     }else {
     }

  }
}
```

Όπως μπορεί να παρατηρήσει κανείς, τα μηνύματα που πρέπει να σταλούν κατά την εξερεύνηση τοποθετούνται σε μια λίστα χρησιμοποιώντας τη συνάρτηση *messageAdd*. Η συνάρτηση αυτή λαμβάνει σαν όρισμα τη διεύθυνση του κόμβου όπου πρέπει να σταλεί το μήνυμα και τον τύπο του μηνύματος. Ο τύπος του μηνύματος μπορεί να πάρει μία από τις τιμές DFS_MSG_SEARCH, DFS_MSG_PARENT ή DFS_MSG_ALREADY.

```
  // Add a new message in the linked list of unsent messages
  void messageAdd(uint16_t tid, uint8_t mtype) {
   DFSMsgQueue *msg, *oldmsg;

   /* allocate memory **/
   msg = (DFSMsgQueue*)malloc(sizeof(DFSMsgQueue));
   memset(msg, 0, sizeof(DFSMsgQueue));

   /* fix parameters **/
   msg->targetid = tid;
   msg->srcid = TOS_LOCAL_ADDRESS;
   msg->msgtype = mtype;

  /* insert the new element into the list while keeping
   * it sorted **/
```

```
SGLIB_SORTED_LIST_ADD_IF_NOT_MEMBER(DFSMsgQueue, m_unsent, msg,
DFSMsgQueue_COMPARATOR, next_ptr, oldmsg);
}
```

Το επόμενο task είναι αυτό που αποστέλει τα μηνύματα καλώντας την εντολή *SendSearch.send*. Αφού συμπληρώσει τα δεδομένα του μηνύματος, το στέλνει και το αφαιρεί από τη λίστα με τα προς αποστολή μηνύματα, χρησιμοποιώντας τη συνάρτηση *messageDel*.

Το *sendNextMessage* έχει ως εξής:

```
// Send next message from the queue
task void sendNextMessage() {
 DFSMsgSearch * search = (DFSMsgSearch *) m_msgSearch.data;

 // Fix message contents
 atomic {
  search->targetid = m_unsent->targetid;
  search->srcid = m_unsent->srcid;
  search->msgtype = m_unsent->msgtype;
  search->msgid = m_nextMsgId;
 }

  // Send the message to m_unsent->targetid
  if ((call SendSearch.send(search->targetid,
  sizeof(DFSMsgSearch),m_msgSearch)) != SUCCESS) {
   dbg(DBG_TEMP, "DFS: transmission <%d> to
   %d FAILED [%d]\n",
   search->msgtype, search->targetid, search->msgid);
  }else{

   /* Increase counter **/
   m_nextMsgId++;

   dbg(DBG_TEMP, "DFS: transmission <%s> to
```

```
     %d SUCCEEDED [%d]\n",
     msgTypeDescr(search->msgtype), search->targetid,
     search->msgid);

     /* drop message from the list **/
     messageDel(search->targetid, search->msgtype);
   }
 }
```

Η *messageDel* συνάρτηση έχει ως εξής:

```
//Remove a message from the linked list of unsent
//messages
void messageDel(uint16_t tid, uint8_t mtype) {
  DFSMsgQueue *msg, *oldmsg;

  /* allocate memory **/
  msg = (DFSMsgQueue*)malloc(sizeof(DFSMsgQueue));
  memset(msg, 0, sizeof(DFSMsgQueue));

  /* fix parameters **/
  msg->targetid = tid;
  msg->srcid = TOS_LOCAL_ADDRESS;
  msg->msgtype = mtype;

  //remove the element frome the list while keeping
  //it sorted
  SGLIB_SORTED_LIST_DELETE_IF_MEMBER(DFSMsgQueue, m_unsent,
  msg, DFSMsgQueue_COMPARATOR, next_ptr, oldmsg);

  /* check if child already in list **/
  while (oldmsg != NULL) {
   //remove the element frome the list while keeping
   //it sorted
```

```
    SGLIB_SORTED_LIST_DELETE_IF_MEMBER(DFSMsgQueue, m_unsent,
    msg, DFSMsgQueue_COMPARATOR, next_ptr, oldmsg);
   }
  }
```

5.3.3 Το NeighborsDetection module

Η λειτουργία του NeighborsDetection module είναι βοηθητική. Η υλοποίησή του είναι πολύ απλή. Χρησιμοποιεί έναν χρονιστή, ο οποίος χτυπάει επαναλαμβανόμενα. Κάθε φορά που προκαλείται ένα γεγονός από το χρονιστή, όλοι οι κόμβοι εκπέμπουν ένα μήνυμα αναζήτησης προς όλους τους γείτονές τους καλώντας τη συνάρτηση *SendMsg.send*. Όταν κάποιος κόμβος λάβει ένα μήνυμα αναζήτησης, προσθέτει τη διεύθυνση του αποστολέα στη λίστα με τους γειτονικούς του κόμβους.

5.3.4 Το Echo module

Το Echo module όπως μαρτυρά και το όνομά του υλοποιεί τον αλγόριθμο Echo. Λειτουργεί σε συνεργασία με ένα από τα modules BFSM ή DFSM, από τα οποία αποκτά την αρχική γνώση του γεννητικού δέντρου που δημιουργούν, έτσι ώστε να στέλνει τα μηνύματά του πάνω σε αυτή τη δενδρική δομή και όχι να τα διαχέει παντού στο δίκτυο. Έτσι χρησιμοποιεί τη διεπαφή Neighborhood καθώς επίσης και τις διεπαφές SendMsg, ReceiveMsg, Timer, ενώ παρέχει τη διεπαφή StdControl.

```
module EchoM{
  provides {
      interface StdControl;
  }
  uses {
    interface Neighborhood;
    interface SendMsg as EchoSend;
    interface ReceiveMsg as EchoReceive;
    interface Timer as EchoStart;
    interface Timer as EchoGetNeighbors;
```

```
  }
}
```

Εντολές Στην εντολή *StdControl.init* πραγματοποιούνται οι αναγκαίες αρχικοποιήσεις μεταβλητών.

```
//Initialization
 command result_t StdControl.init(){
         uint8_t i;
         atomic {
           my_ID = TOS_LOCAL_ADDRESS;
           messages =0;
           rec = 0;
           finished = FALSE;
           if(my_ID==0)
                   parent=0;
           else
                   parent=DFS_MAX_NODE_ID;
           }

           for (i = 0; i < MESSAGE_QUEUE_SIZE; i++) {
                    msgqueue[i].pMsg = NULL;
           }

         enqueue_next = 0;
         dequeue_next = 0;
         fQueueIdle = TRUE;

         return SUCCESS;
 }
```

Στην εντολή *StdControl.start* αρχικοποιούνται οι χρονιστές του συστήματος.

```
//Start
 command result_t StdControl.start(){
```

```
        call EchoGetNeighbors.start(TIMER_ONE_SHOT, 25000);

        if (TOS_LOCAL_ADDRESS == 0)
                call EchoStart.start(TIMER_REPEAT,30000);
        return  SUCCESS;
 }
```

Στην εντολή *StdControl.stop* απενεργοποιούνται οι χρονιστές του συστήματος.

```
//Halt execution - disable clck
 command result_t StdControl.stop(){
        call EchoGetNeighbors.stop();
        return call EchoStart.stop();
}
```

Γεγονότα Το γεγονός *EchoGetNeighbors.fired* προκαλείται από τον αντίστοιχο χρονιστή. Για το χειρισμό του γεγονότος απλώς καλείται το task *getNeighborsList* που θα κάνει τις υπολογιστικές λειτουργίες. Ο χρονιστής αυτός χρησιμοποιείται για τη λήψη της πληροφορίας σχετικά με το γεννητικό δέντρο· δηλαδή κάθε κόμβος μαθαίνει ποια είναι τα παιδιά του.

```
event result_t EchoGetNeighbors.fired(){
        post getNeighborsList();

        return SUCCESS;
 }
```

Μόλις χτυπήσει ο δεύτερος χρονιστής, ο αρχικοποιητής ξεκινάει τον αλγόριθμο στέλνοντας το πρώτο μήνυμα.

```
 event result_t EchoStart.fired(){
        if(!(TOS_LOCAL_ADDRESS==0 && finished==TRUE))
                post sendEchoMsg();

        return SUCCESS;
 }
```

Όπως και στους προηγούμενους αλγορίθμους, όταν λαμβάνεται ένα μήνυμα προκαλείται το γεγονός *EchoReceive.receive*. Το μήνυμα τοποθετείται σε μια ουρά και καλείται το task *QueueServiceTask* για την εξυπηρέτησή της.

```
// Process a message received
event TOS_MsgPtr EchoReceive.receive(TOS_MsgPtr recv_packet){
   EchoMsg * msgdata = (EchoMsg*) recv_packet->data;

   if(recv_packet == NULL)
        dbg(DBG_TEMP, "\nEcho: Received null packet\n");
   else
        dbg(DBG_TEMP, "\nEcho: Received EchoMsg(%d) \n",
    msgdata->srcID);

   if (((enqueue_next + 1) % MESSAGE_QUEUE_SIZE)
   == dequeue_next) {
        // Fail if queue is full
        dbg(DBG_USR2, "QueuedSend: queue is full!\n");
        return NULL;
    }

    atomic{
        msgqueue[enqueue_next].address = msgdata->srcID;
        msgqueue[enqueue_next].id = msgdata->msgID;
        msgqueue[enqueue_next].pMsg = recv_packet;

        enqueue_next++; enqueue_next %= MESSAGE_QUEUE_SIZE;
    }

    dbg(DBG_USR1, "QueuedSend: Successfully queued msg to
    0x%x, enq %d, deq %d and queue idles = %d\n",
    msgdata->srcID, enqueue_next, dequeue_next, fQueueIdle);

    post QueueServiceTask();
```

```
    return recv_packet;
 }
```

Το τελευταίο γεγονός είναι το *EchoSend.sendDone* γεγονός.

```
//handles the sendDone event
event result_t EchoSend.sendDone(TOS_MsgPtr msg,bool success){
        dbg(DBG_TEMP, "send done \n");
        return SUCCESS;
 }
```

Tasks Στο Echo module υλοποιήθηκε καταρχάς ένα task για αποστολή μηνύματος σε ένα απλό κόμβο και ένα δεύτερο για αποστολή μηνύματος στον πατέρα του κόμβου.

Το *sendEchoMsg* task στέλνει ένα μήνυμα σε όλους τους γείτονες ενός κόμβου, εκτός από τον πατέρα του.

```
//Send echo message to neighbors
 task void sendEchoMsg(){

   /*....declarations...*/

   if(mID == 0){          //send echo to all neighbors
        SGLIB_LIST_MAP_ON_ELEMENTS(NodeNeighbor,
    children_array,node, next_ptr, {
        call EchoSend.send(node->id,sizeof(EchoMsg),&m_msg);
    });
   }else{
    //send echo to all neighbors except the father
    SGLIB_LIST_MAP_ON_ELEMENTS(NodeNeighbor, children_array,
    node, next_ptr, {
    if ((node->id)!=p) call EchoSend.send(node->id,
    sizeof(EchoMsg),&m_msg) ;
    });
   }
```

```
}
```

Με το *sendEchoParent* task ένας κόμβος στέλνει ένα μήνυμα στον πατέρα του.

```
//send echo answer to parent
 task void sendEchoParent(){
        result_t r;
        uint16_t p;
        uint16_t mID;

        EchoMsg *msgdata = (EchoMsg*) m_msg.data;

        //set message contents
        atomic{
                mID = my_ID;
                msgdata->srcID = my_ID;
                msgdata->msgID=messages+1;
                p = parent;
        }

        r = call EchoSend.send(p,sizeof(EchoMsg),&m_msg);
        if(r)
          dbg(DBG_TEMP,"\nEcho: Send to parent(%d) my
          id = %d \n", p, msgdata->srcID);
 }
```

Το *getNeighborsList* task καλεί τις εντολές *Neighborhood.getNeighbors* και *Neighborhood.getSize* που επιστρέφουν τα παιδιά ενός κόμβου και τον αριθμό αυτών αντίστοιχα, και τοποθετεί τα παιδιά σε μια λίστα.

```
task void getNeighborsList(){
  uint16_t i=0;
  NodeNeighbor *newNeighbor, *member;

  neighbors_array = call Neighborhood.getNeighbors();
  totNeighbors = call Neighborhood.getSize();
```

```
  if(!(totNeighbors <1)){
        for(i=0;i<totNeighbors;i++){

     /* allocate memory */
     newNeighbor = (NodeNeighbor*)malloc(sizeof(NodeNeighbor));
     memset(newNeighbor, 0, sizeof(NodeNeighbor));

         newNeighbor->id =neighbors_array[i];

         SGLIB_SORTED_LIST_ADD_IF_NOT_MEMBER(NodeNeighbor,
     children_array, newNeighbor, Neighbor_COMPARATOR,
         next_ptr, member);
        }
  }

 }
```

Το τελευταίο task είναι το *QueueServiceTask* task. Το task αυτό εξυπηρετεί ένα προς ένα τα μηνύματα που βρίσκονται στην ουρά. Η εξυπηρέτηση έγκειται στην παραγματοποίηση των ελέγχων που ορίζει ο αλγόριθμος Echo.

```
task void QueueServiceTask() {
    uint8_t id;
    uint16_t address;

    // Try to send next message (ignore xmit_count)
    if (msgqueue[dequeue_next].pMsg != NULL) {
        id = msgqueue[dequeue_next].id;
        address = msgqueue[dequeue_next].address;

        if(my_ID==0){
          rec++;
```

```
        //if received echo from all children, decide
        if(rec == totNeighbors){
                finished = TRUE;
                dbg(DBG_TEMP,"Finished!\n");
        }else{
                childDel(address);
        }
    }else{ // I am not the 0 node
        if(rec == 0 || parent == 1000) {
                parent = address;
                rec++;
                if (totNeighbors ==0){
                        post sendEchoParent();
                }else{
                        post sendEchoMsg();
                }
                }else if(address == parent){
                   if (totNeighbors ==0||(rec-1==totNeighbors)){
                    post sendEchoParent();
                   }else{
                    post sendEchoMsg();
                   }
                }else if(rec== totNeighbors) {
                   rec++;
                   childDel(address);
                   post sendEchoParent();
                }else{
                   rec++;
                   childDel(address);
                }
    } //end else

    msgqueue[dequeue_next].pMsg = NULL;
```

```
        dequeue_next++; dequeue_next %= MESSAGE_QUEUE_SIZE;

        dbg(DBG_USR1, "QueuedSend: Successfully dequeued msg
        from 0x%x, now enq %d, deq %d and queue idles = %d\n",
        address, enqueue_next, dequeue_next, fQueueIdle);
    }
    else{
        fQueueIdle = TRUE;
    }
}
```

5.4 Configurations

Για να συνεργαστούν μεταξύ τους τα modules θα πρέπει να προηγηθεί η μεταξύ τους σύνδεση. Με τις συνδέσεις που πραγματοποιούνται ουσιαστικά χτίζεται ολόκληρη η εφαρμογή. Συνεπώς, εφόσον οι εφαρμογές μας είναι δύο, πρέπει να γίνουν και οι αντίστοιχες συνδέσεις.

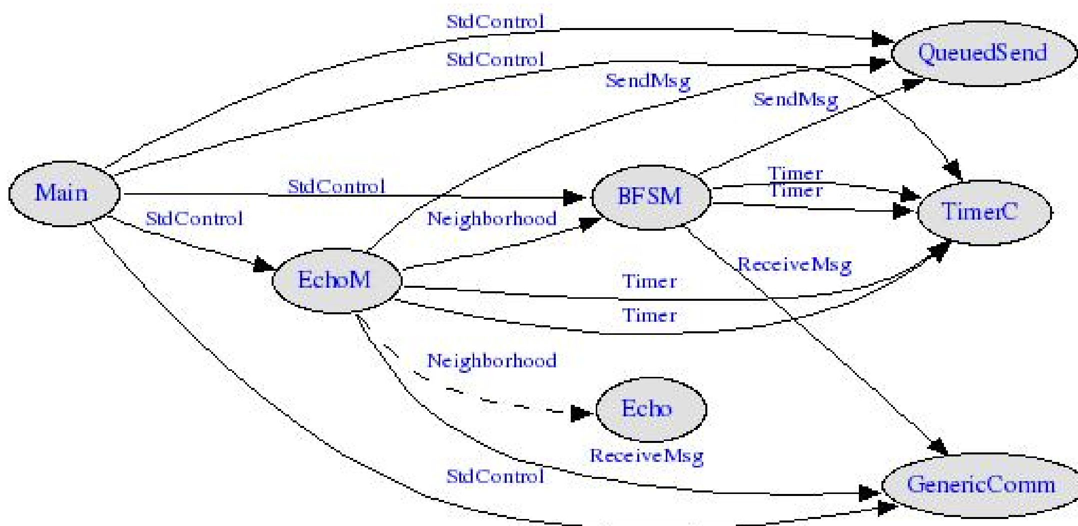
Έτσι έχουμε δύο configurations, ένα για τη σύνδεση του BFS module με το Echo module και ένα για τη σύνδεση του NeighborsDetection module με το DFS module και στη συνέχεια το δεύτερο με το Echo module.

BFSM και EchoM Ο παρακάτω κώδικας δείχνει πώς γίνονται οι συνδέσεις στο πρώτο configuration. Η διεπαφή από το πρώτο component συνδέεται με την αντίστοιχη διεπαφή του δεύτερου.

```
configuration Echo{
  uses interface Neighborhood;
}
implementation{
  components EchoM, QueuedSend, GenericComm, BFSM, TimerC, Main;

  Main.StdControl ->EchoM.StdControl;
  Main.StdControl ->BFSM.StdControl;
```

```
  Main.StdControl -> GenericComm;
  Main.StdControl -> QueuedSend;
  Main.StdControl -> TimerC.StdControl;

  EchoM.Neighborhood->BFSM.Neighborhood;

  EchoM.EchoSend->QueuedSend.SendMsg[AM_ECHOMSG ];
  EchoM.EchoReceive->GenericComm.ReceiveMsg[AM_ECHOMSG ];
  EchoM.EchoTimer -> TimerC.Timer[unique("Timer")];

  BFSM.BfsTimer -> TimerC.Timer[unique("Timer")];
  BFSM.StatusTimer -> TimerC.Timer[unique("Timer")];
  BFSM.SendMsg->QueuedSend.SendMsg[AM_BFSMSG];
  BFSM.ReceiveMsg->GenericComm.ReceiveMsg[AM_BFSMSG];

  Neighborhood = EchoM;
}
```

Γραφικά, το παραπάνω configuration περιγράφεται από το σχήμα 5.2.



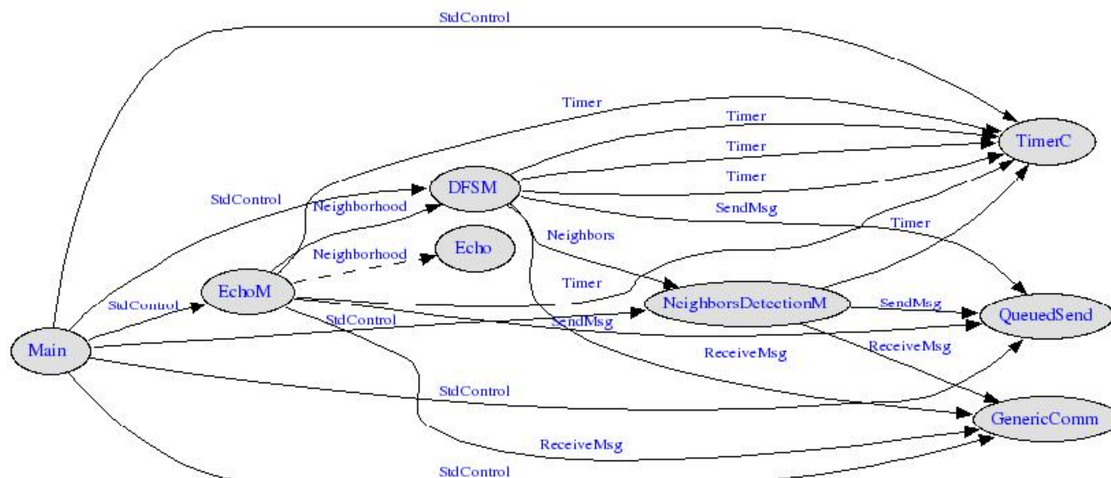
Σχήμα 5.2: Η σύνδεση των components για το Echo configuration

DFSM και EchoM Ο παρακάτω κώδικας δείχνει πώς γίνονται οι συνδέσεις στο δεύτερο configuration.

```
configuration Echo{
  uses interface Neighborhood;
}
implementation{
  components EchoM, QueuedSend, GenericComm as Comm, DFSM,
  NeighborsDetectionM, TimerC, Main;

  Main.StdControl ->EchoM.StdControl;
  Main.StdControl ->DFSM.StdControl;
  Main.StdControl -> NeighborsDetectionM.StdControl;
  Main.StdControl -> Comm;
  Main.StdControl -> QueuedSend;
  Main.StdControl -> TimerC.StdControl;

  EchoM.Neighborhood->DFSM.Neighborhood;

  EchoM.EchoSend->QueuedSend.SendMsg[AM_ECHOMSG ];
  EchoM.EchoReceive->Comm.ReceiveMsg[AM_ECHOMSG ];
  EchoM.EchoStart -> TimerC.Timer[unique("Timer")];
  EchoM.EchoGetNeighbors -> TimerC.Timer[unique("Timer")];

  NeighborsDetectionM.SendMsg->QueuedSend.SendMsg[AM_NEIGHMSG];
  NeighborsDetectionM.ReceiveMsg->Comm.ReceiveMsg[AM_NEIGHMSG];
  NeighborsDetectionM.NeighTimer -> TimerC.Timer[unique("Timer")];

  DFSM.Neighbors -> NeighborsDetectionM.Neighbors;
  DFSM.SendSearch -> QueuedSend.SendMsg[AM_DFS_SEARCH];
  DFSM.ReceiveSearch -> Comm.ReceiveMsg[AM_DFS_SEARCH];
  DFSM.SearchTimer -> TimerC.Timer[unique("Timer")];
  DFSM.SendTimer -> TimerC.Timer[unique("Timer")];
  DFSM.ReckonTimer -> TimerC.Timer[unique("Timer")];
```

```
  Neighborhood = EchoM;
}
```

Γραφικά, το παραπάνω configuration περιγράφεται από το σχήμα 5.3.



Σχήμα 5.3: Η σύνδεση των components για το δεύτερο Echo configuration

5.5 Πειράματα

Τοπολογίες Οι τοπολογίες που χρησιμοποιήθηκαν για τις δοκιμές των αλγορίθμων αποτελούνται από πέντε, εννέα και δεκαέξι κόμβους. Οι τοπολογίες αυτές φαίνονται στα σχήματα 5.4, 5.5 και 5.6 αντίστοιχα.

Μετρήσεις Τα αποτελέσματα που δώσανε οι αλγόριθμοι αναζήτησης πρώτα-κατά-εύρος και πρώτα-κατά-βάθος σχετικά με τον αριθμό μηνυμάτων που ανταλλάσσονται φαίνονται στους πίνακες 5.1 και 5.2 αντίστοιχα. Στον πίνακα 5.3 φαίνεται ο αριθμός των μηνυμάτων που ανταλλάσσονται όταν ο αλγόριθμος Echo εφαρμόζεται σε δέντρο αναζήτησης πρώτα-κατά-εύρος.

Αναζήτηση πρώτα-κατά-εύρος	
Τοπολογία	**Αριθμός μηνυμάτων**
Συνεκτικό δίκτυο 5 κόμβων	18
Δίκτυο 3x3 κόμβων	11
Δίκτυο 4x4 κόμβων	19

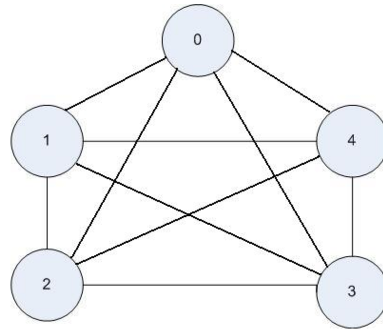
Πίνακας 5.1: Αναζήτηση πρώτα-κατά-εύρος

Αναζήτηση πρώτα-κατά-βάθος	
Τοπολογία	**Αριθμός μηνυμάτων**
Συνεκτικό δίκτυο 5 κόμβων	20
Δίκτυο 3x3 κόμβων	26
Δίκτυο 4x4 κόμβων	50

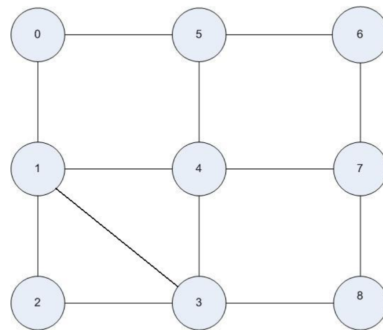
Πίνακας 5.2: Αναζήτηση πρώτα-κατά-βάθος

Echo σε BFS δέντρο	
Τοπολογία	**Αριθμός μηνυμάτων**
Συνεκτικό δίκτυο 5 κόμβων	8
Δίκτυο 3x3 κόμβων	30
Δίκτυο 4x4 κόμβων	53

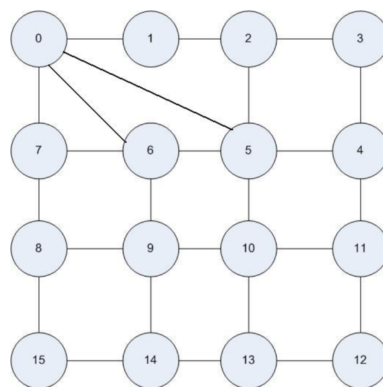
Πίνακας 5.3: Εφαρμογή του αλγορίθμου Echo σε BFS δέντρο



Σχήμα 5.4: Δίκτυο 5 κόμβων και 10 ακμών.



Σχήμα 5.5: Δίκτυο 3x3 κόμβων και 11 ακμών.



Σχήμα 5.6: Δίκτυο 4x4 κόμβων και 21 ακμών.

Κεφάλαιο 6

Συμπεράσματα και Μελλοντική Εργασία

Στα προηγούμενα κεφάλαια μελετήσαμε μια βασική κατηγορία αλγορίθμων, τους Κυματικούς αλγορίθμους, με σκοπό τη χρησιμοποίησή τους σε εφαρμογές ασύρματων δικτύων αισθητήρων. Αφού λοιπόν ορίσαμε το πλαίσιο εργασίας και μελέτης, υλοποιήσαμε κάποιους κυματικούς αλγορίθμους – συγκεκριμένα τους Echo, αναζήτηση πρώτα-κατά-βάθος, αναζήτηση πρώτα-κατά-εύρος – και αναπτύξαμε μια εφαρμογή που βασίζεται σε αυτούς. Η ανάπτυξη έγινε στο λειτουργικό σύστημα TinyOS έτσι ώστε να αποκτήσουμε μια πρώτη γνώση ως προς τη συμπεριφορά των αλγορίθμων σε περιβάλλον προσομοίωσης ενός πραγματικού δικτύου αισθητήρων.

Η ανάλυση των αλγορίθμων σε θεωρητικό επίπεδο δίνει πολύ ικανοποιητικά αποτελέσματα. Παρουσιάζουν πολύ καλή συμπεριφορά στις χειρότερες περιπτώσεις, όσο αναφορά την πολυπλοκότητα χρόνου και επικοινωνίας, ένα στοιχείο που είναι ιδιαίτερα χρήσιμο στα δίκτυα αισθητήρων, λόγω των ειδικών χαρακτηριστικών τους.

Εντούτοις, σε πραγματικές εφαρμογές οι αλγόριθμοι ίσως να μη δώσουν τα θεωρητικώς αναναμενόμενα αποτελέσματα. Αυτό συμβαίνει γιατί ένα δίκτυο αισθητήρων είναι ένα μεταβαλλόμενο σύστημα, όπου η επικοινωνία μεταξύ των κόμβων δεν είναι αξιόπιστη, λόγω των σφαλμάτων που εμφανίζονται κατά τις μεταδόσεις. Ήδη από την προσομοίωση είδαμε ότι κάτι τέτοιο πράγματι συμβαίνει.

Μελλοντικά θα είχε ενδιαφέρον, να πραγματοποιηθούν πειράματα εφαρμόζοντας αυτή

τη φορά τους αλγορίθμους σε πραγματικές συσκευές, έτσι ώστε να παρατηρήσουμε κατά πόσο αποκλίνουν τα πραγματικά αποτελέσματα από τα αποτελέσματα της προσομοίωσης. Με τον τρόπο αυτό μπορούμε να μετρήσουμε την αξιοπιστία του εξομοιωτή, καθώς επίσης να δούμε κατά πόσο επηρεάζει ο παράγοντας του περιβάλλοντος τη λειτουργία των αλγορίθμων.

Επιπλέον, θα μπορούσαμε να ενσωματώσουμε τους αλγορίθμους αυτούς σε υπάρχοντα πρωτόκολλα, όπως αυτά που περιγράφησαν στο κεφάλαιο 2 και να παρατηρήσουμε το κατά πόσο οι κυματικοί αλγόριθμοι βελτιώνουν την απόδοση των πρωτοκόλλων αυτών.

Βιβλιογραφία

[1] Holger Karl, Andreas Willig. Protocols and Architectures for Wireless Sensor Networks, WILEY 2005

[2] Kazem Sohraby, Daniel Minoly, Taieb Znati. Wireless Sensor Networks, Technology, Protocols and Applications, WILEY 2007

[3] I.F. Akyildiz, W. Su, Y. Sankarasubramaniam, E. Cayirci. Wireless sensor networks: a survey, in the Journal of Computer Networks, Volume 38, pp. 393-422, 2002

[4] I.F. Akyildiz, W. Su, Y. Sankarasubramaniam, E. Cayirci, A survey on sensor networks, in the IEEE Communications Magazine, pp. 102-114, August 2002

[5] Kirk Martinez, Jane K.Hart, Royan Ong. Environmental sensor networks, in the IEEE Computer Magazine, August 2004

[6] Chalermek Intanagonwiwat, Ramesh Govindan, Deborah Estrin, John Heidemann and Fabio Silva. Directed Diffusion for Wireless Sensor Networking, in IEEE/ACM Transactions on Networking, Volume 11 , Issue 1 (February 2003), pp. 2 - 16, 2003

[7] I. Chatzigiannakis, T. Dimitriou, S. Nikoletseas and P. Spirakis: A Probabilistic Forwarding Protocol for Efficient Data Propagation in Sensor Networks, FLAGS Technical Report, FLAGS-TR-14, 2003

[8] Gerard Tel. Introduction to Distributed Algorithms, Cambridge University Press 2000

[9] TinyOS: A Component-based OS for the Network Sensor Regime. http://webs.cs.berkeley.edu/tos/, Fall 2007

[10] Philip Levis. TinyOS Programming, June 2006

[11] Manuel Lang. TinyOS, November 2006

[12] D. Gay, R. von Behren, M. Welsh, E. Brewer and D. Culler. The nesC Language: a holistic approach to networked embedded systems, Intel Research Berkeley, November 2002

[13] D. Gay, D. Culler and P. Levis. The nesC Language Reference Manual, September 2002

[14] The nesC Project, http://nescc.sourceforge.net.

[15] Αντωνίου Αθανάσιος. Δίκτυα Έξυπνης Σκόνης, Τεχνολογική μελέτη υπαρχόντων συστημάτων και σχεδιασμός και ανάλυση πρωτοκόλλων για αποδοτικό εντοπισμό και διάδοση πληροφορίας. Ανάπτυξη και λειτουργία πραγματικού πειραματικού δικτύου. Έμφαση σε πρωτόκολλα διαχείρισης του δικτύου. Διπλωματική εργασία, Ιούλιος 2003

[16] Μυλωνάς Γεώργιος. Δίκτυα Έξυπνης Σκόνης, Τεχνολογική μελέτη υπαρχόντων συστημάτων και συγκριτική αξιολόγηση πρωτοκόλλων για αποδοτικό εντοπισμό και διάδοση πληροφορίας. Ανάπτυξη και λειτουργία πραγματικού πειραματικού δικτύου. Διπλωματική εργασία, Ιούνιος 2003

[17] Πευκιανάκης Ιωάννης. Δίκτυα Έξυπνης Σκόνης (Smart Dust): Μελέτη και ανάλυση αρχιτεκτονικών, πρωτοκόλλων και τεχνολογιών. Το πρόβλημα της τοπικής ανίχνευσης και διάδοσης της πληροφορίας και το πρωτόκολλο MP-DFR.