ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ Η/Υ ΚΑΙ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΑΤΡΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Δίκτυα έξυπνης σκόνης: Τεχνολογική μελέτη υπαρχόντων συστημάτων και συγκριτική αξιολόγηση πρωτοκόλλων για αποδοτικό εντοπισμό και διάδοση πληροφορίας. Ανάπτυξη και λειτουργία πειραματικού δικτύου

Γεώργιος Χ. Μυλωνάς, ΑΜ 2159
Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πάτρα
mylonasg@ceid.upatras.gr

Επιβλέπων: Σωτήρης Νικολετσέας
Λέκτορας Πανεπιστημίου Πατρών

26 Ιουνίου 2003

Κάθε αυθεντικό αντίτυπο φέρει την υπογραφή του συγγραφέα...

Αφιερωμένο στους γονείς μου, στον αδερφό μου,
στους φίλους μου και σε όλους όσους προσπαθούν να γίνουν καλύτεροι...

Πρόλογος

Τα δίκτυα έξυπνης σκόνης αποτελούν μια νέα κατηγορία δικτύων υπολογιστών, η οποία υπόσχεται με τα ιδιαίτερα χαρακτηριστικά της να αναθεωρήσει τη γενική εικόνα που έχουμε για τα δίκτυα ως σήμερα. Τέτοια δίκτυα αποτελούνται από μεγάλο πλήθος υπολογιστικών κόμβων μικροσκοπικού μεγέθους, εφοδιασμένων με πλήθος αισθητήρων και μονάδων ελέγχου. Σκοπός τους είναι η επίτευξη μιας δύσκολης, για τα δεδομένα του κάθε κόμβου, αποστολής μέσω της συνεργασίας μεταξύ τους.

Τα δίκτυα αυτά έγιναν τεχνικώς υλοποιήσιμα μόλις τα τελευταία 4 χρόνια, μετά από την αλματώδη πρόοδο που σημειώθηκε στους τομείς του VLSI και των ασύρματων τηλεπικοινωνιών. Το νεαρό της ηλικίας τους αντικατοπτρίζεται και στο πλήθος των ευκαιριών για επιστημονική έρευνα και δραστηριότητα. Επιλέξαμε να ασχοληθούμε με πρωτόκολλα επιπέδου δικτύου για διάδοση πληροφορίας σε αυτά τα δίκτυα, και κινηθήκαμε στους παρακάτω τρεις άξονες:

1. ΄Ερευνα αιχμής στα πρωτόκολλα δικτύου για δίκτυα έξυπνης σκόνης: Γίνεται μια εκτενής αναφορά στις βασικές έννοιες και στη θεωρία γύρω από τα δίκτυα αυτά, η οποία συνδυάζεται με μια περιγραφή 6 επιλεγμένων πρωτοκόλλων από τη σχετική βιβλιογραφία. Αναδύονται τα ιδιαίτερα χαρακτηριστικά των δικτύων έξυπνης σκόνης και ποια είναι τα κυρίαρχα θέματα που διέπουν τη σχεδίασή τους, ενώ βλέπουμε τις τάσεις που υπάρχουν στην έρευνα σήμερα. Με λίγα λόγια, ο ο αναγνώστης μπορεί να βρει συγκεντρωμένα όλα τα στοιχεία που αποτελουν το θεωρητικό υπόβαθρο για τη μελέτη των δικτύων αυτών.

2. Εξομοίωση πρωτοκόλλων για δίκτυα έξυπνης σκόνης και συγκριτική αξιολόγησή τους: Στα πλαίσια της εργασίας αυτής, αναπτύχθηκε ένας εξομοιωτής δικτύων έξυπνης σκόνης, τον οποίο καλούμε simDust, για τρία πρωτόκολλα (LTP, PFR, TEEN), ο οποίος παρέχει και δυνατότητες οπτικοποίησης (animation) δίκτυων έξυπνης σκόνης. Μαζί με την ανάλυση της υλοποίησης του simDust παρέχεται και μια συγκριτική αξιολόγηση δύο πρωτοκόλλων (TEEN και PFR), μέσω αποτελεσμάτων που πήραμε χρησιμοποιώντας τον simDust. Επίσης, *η κλίμακα της εξομοίωσης ξεπερνά σε μέγεθος όλες τις αντίστοιχες υλοποιήσεις που έχουν γίνει ως τώρα*.

3. Υλοποίηση εφαρμογών σε ένα πειραματικό δίκτυο έξυπνης σκόνης: Για να είναι η εργασία μας ολοκληρωμένη, ασχοληθήκαμε και με *εφαρμογές σε ένα*

πραγματικό πειραματικό δίκτυο. Για την υλοποίησή μας επιλέξαμε την πλατφόρμα TinyOS/Smart Dust, η οποία είναι η μονάδικη ολοκληρωμένη πρόταση στο χώρο. Μαζί με την υλοποίηση κάποιων εφαρμογών, παρέχεται και μια αναλυτική περιγραφή της πειραματικής πλατφόρμας, την οποία πιστεύουμε ότι ο αναγνώστης θα βρει εξαιρετικά χρήσιμη καθώς οι σχετικές με το θέμα αυτό πηγές είναι λιγοστές.

Η εργασία αυτή είναι *από τις πρώτες που γίνονται σχετικά με τα δίκτυα έξυπνης σκόνης, σε πανευρωπαϊκό επίπεδο*, και επίσης καλύπτει το θέμα από πολλές πλευρές. Τέλος, ένα μεγάλο μέρος της γράφηκε έχοντας υπόψη τον αναγνώστη που δεν έχει καμία προηγούμενη εμπειρία με τα δίκτυα έξυπνης σκόνης. Ελπίζουμε ότι θα αποτελέσει μια καλή εισαγωγή και ουσιαστική βοήθεια σε όσους επιλέξουν να ασχοληθούν με τα δίκτυα αυτά στο μέλλον. Καλή ανάγνωση!

Ευχαριστίες

Θα ήθελα πρώτα από όλους να ευχαριστήσω τους συνάδελφούς μου Θανάση Αντωνίου και Γιάννη Χατζηγιαννάκη, οι οποίοι στάθηκαν πολύτιμοι συνεργάτες και καλοί φίλοι. Ειδικά ο Γιάννης, ο οποίος έκανε τη συνεπίβλεψη της διπλωματικής, αφιέρωσε πολύτιμο γι' αυτόν χρόνο και ενέργεια και τον ευχαριστώ ιδιαίτερα για αυτό.

Θα ήθελα στη συνέχεια να ευχαριστήσω τους γονείς και τον αδερφό μου Βαγγέλη, για την υπομονή και την κατανόηση που δείχνουν απέναντί μου όλα αυτά τα χρόνια (ελπίζω να συνεχίσουν και στο μέλλον...).

Θα ήθελα ακόμα να ευχαρίστησω τον Σωτήρη Νικολετσέα, Λέκτορα του Τμήματος Μηχανικών Η/Υ και Πληροφορικής, για την ευκαιρία που μου έδωσε με την ανάθεση αυτής της διπλωματικής εργασίας και την καθοδήγησή του καθόλη τη διάρκεια εκπόνησής της. Τον ευχαριστώ για τις πολύτιμες συμβουλές του, που με βοήθησαν να ξεκαθαρίσω τους στόχους μου, για την εμπιστοσύνη που μου έδειξε και για την ελευθερία δράσης που μου έδωσε.

Τέλος, θα ήθελα να ευχαριστήσω τους δημιουργούς του South Park, Trey Parker και Matt Stone, καθώς και τους Monty Python, οι οποίοι αποτελούν για μένα φωτεινό παράδειγμα σκέψης και δημιουργικής φαντασίας.

"If dolphins are so smart, why do they live in igloos?" - Eric Cartman

Περιεχόμενα

Μέρος Ι

Πρωτόκολλα για δίκτυα έξυπνης σκόνης

Κεφάλαιο 1

Εισαγωγή στα δίκτυα έξυπνης σκόνης

1.1 Γενικά εισαγωγικά στοιχεία

Υπάρχουν φορές, όπου η παρατήρηση της φύσης γύρω μας, μας διδάσκει και μας δείχνει το δρόμο προς την καινοτομία. Ας δούμε το παράδειγμα της κοινωνίας των μυρμηγκιών. Ένα μυρμήγκι από μόνο του μάλλον μας είναι αδιάφορο. Εάν βλέπαμε μια γραμμή από μυρμήγκια στη σειρά μέσα στο σπίτι μας, ίσως να ενοχλούμασταν. Μια μυρμηγκοφωλιά όμως, σίγουρα προκαλεί μεγαλύτερη εντύπωση. Τα μυρμήγκια από μόνα τους είναι ασήμαντα, όταν όμως συνεργάζονται είναι ιδιαίτερα αποτελεσματικά και επιτελούν δύσκολα για το μέγεθός τους καθήκοντα.

Την απλή αυτή ιδέα,"δανείστηκαν" οι ερευνητές που πρώτοι οραματίστηκαν τα δίκτυα έξυπνης σκόνης (smart dust networks), ή αλλιώς δίκτυα αισθητήρων (sensor networks). Τα δίκτυα αυτά αποτελούνται από ένα πολύ μεγάλο αριθμό κόμβων, οι οποίοι έχουν πολύ μικρό μέγεθος, και έχουν κάποιους αισθητήρες ενσωματωμένους συν κάποιες δυνατότητες επεξεργασίας και ασύρματης επικοινωνίας. Οι κόμβοι αυτοί τοποθετούνται με κάποιο τρόπο μέσα σε ένα πεδίο και συνεργάζονται για να φέρουν εις πέρας μια δύσκολη, για τα δεδομένα ενός μόνο τέτοιου κόμβου, αποστολή. Υπάρχει πλειάδα εφαρμογών για δίκτυα έξυπνης σκόνης, τις οποίες αναφέρουμε σε επόμενη ενότητα.

Σε αντίθεση με την απλότητα της αρχικής ιδέας, η υλοποίηση των δικτύων αυτών δεν είναι καθόλου εύκολη υπόθεση. Μάλιστα οι προκλήσεις που παρουσιάζονται σε ερευνητικό επίπεδο είναι πάρα πολλές και επίσης, σε πολλά πεδία. Αρκεί να αναλογιστούμε ότι επιδιώκουμε να κατασκευάσουμε κόμβους δικτύου τόσο μικρούς, ώστε να μοιάζουν με σκόνη, να επικοινωνούν σε αποστάσεις ακόμα και των εκατοντάδων μέτρων, να αντέχουν σε αντίξοες συνθήκες, να καταναλώνουν ελάχιστη ενέργεια. Μόλις τα τελυταία πέντε χρόνια έγινε δυνατόν να κατασκευαστούν ολοκληρωμένα που να πληρούν κάποιες από αυτές τις προϋποθέσεις, χάρη στην πρόοδο στους τομείς του VLSI και των ασύρματων

επικοινωνιών.

Στην παρούσα εργασία θα χρησιμοποιήσουμε την ονομασία "δίκτυα έξυπνης σκόνης" για τα δίκτυα αυτά, η οποία αν και κάπως σουρεαλιστική, είναι μάλλον καλύτερη(τουλάχιστον στην ελληνική γλώσσα) από τα "δίκτυα αισθητήρων". Στη σχετική βιβλιογραφία συναντούμε και τις δύο ονομασίες.

Ακολουθεί μια σύντομη περιγραφή των διαφορών ανάμεσα στα δίκτυα έξυπνης σκόνης και τα ήδη υπάρχοντα δίκτυα, και των παραγόντων που επιδρούν συνολικά στη σχεδίαση ενός τέτοιου δικτύου. Ακολουθεί η γενική αρχιτεκτονική του δικτύου και μία αναφορά σε κάποιες χαρακτηριστικές εφαρμογές. Τέλος, περιγράφουμε δύο εφαρμογές κάπως πιο αναλυτικά, για να δούμε τις απαιτήσεις τους σε ενέργεια, καθώς και τη βιωσιμότητά τους.

΄Ενα εξαιρετικό survey, το οποίο μπορεί να χρησιμοποιήσει ο αναγνώστης ως εισαγωγή στο πεδίο των δικτύων έξυπνης σκόνης, είναι το [1] (το [2] είναι μία πιο πρόσφατη συντομευμένη εκδοχή του).

1.2 Διαφορές των δικτύων έξυπνης σκόνης με τους παραδοσιακούς αισθητήρες και τα κλασσικά ad-hoc δίκτυα

Τα δίκτυα έξυπνης σκόνης αποτελούν σημαντική βελτίωση σε σχέση με τους παραδοσιακούς αισθητήρες, οι οποίοι χρησιμοποιούνται με τους παρακάτω δύο τρόπους:

1. Τοποθετούνται σχετικά μακριά από το φαινόμενο προς παρατήρηση, και επομένως απαιτούνται ακριβοί αισθητήρες, οι οποίοι έχουν τη δυνατότητα να φιλτράρουν το θόρυβο από το περιβάλλον, ο οποίος υπεισέρχεται ακριβώς λόγω της απόστασης από το φαινόμενο προς μέτρηση, ώστε να δίνουν αξιόπιστα αποτελέσματα.

2. Τοποθετούνται σε μικρούς αριθμούς με κάποιο *προσχεδιασμένο* τρόπο και μεταδίδουν συνεχώς μετρήσεις ανά τακτά διαστήματα προς κάποιο κεντρικό σταθμό.

Σε αντίθεση με τους κλασσικούς αισθητήρες, ένα δίκτυο έξυπνης σκόνης:

1. Αποτελείται από πολύ μεγάλο αριθμό κόμβων και η ανάπτυξή του στο πεδίο ενδιαφέροντος μπορεί να γίνεται με εντελώς τυχαίο τρόπο.

2. Οι κόμβοι του δικτύου βρίσκονται *μέσα* στο πεδίο ενδιαφέροντος. Οι μετρήσεις που λαμβάνουμε είναι συνεπώς ακριβείς.

3. Αντί να στέλνει ο κάθε κόμβος κατευθείαν τις μετρήσεις που παίρνει σε κάποιο κεντρικό (βασικό) σταθμό, χρησιμοποιείται ένα πιο σύνθετο πρωτόκολλο δικτύου για να προωθηθεί η πληροφορία προς το βασικό σταθμό, σε περισσότερα από ένα βήματα.

Διαβάζοντας κάποιος τα προήγουμενα χαρακτηριστικά, μπορεί να συμπεράνει ότι τα δίκτυα έξυπνης σκόνης είναι απλά ad hoc δίκτυα και να αναρωτηθεί: "γιατί δεν εφαρμόζουμε απλά τις μεθόδους και τα πρωτόκολλα που έχουμε αναπτύξει τα τελευταία χρόνια;". Η απάντηση είναι ότι τα δίκτυα έξυπνης σκόνης, αν και παρουσιάζουν ομοιότητες με τα κλασσικά ad hoc δίκτυα, δεν ταυτίζονται μαζί τους. Επομένως, οι ήδη υπάρχουσες μέθοδοι για ad hoc δεν επαρκούν για τα δίκτυα έξυπνης σκόνης και παραθέτουμε τους κυριότερους λόγους για αυτό:

- Ο αριθμός των κόμβων σε ένα δίκτυο έξυπνης σκόνης είναι πολύ μεγαλύτερος από ότι σε ένα ad hoc δίκτυο.
- Η πυκνότητα των κόμβων στο πεδίο είναι πολύ μεγαλύτερη.
- Οι κόμβοι είναι επιρρεπείς σε αστοχίες (failures), κυρίως σε επίπεδο υλικού (hardware), και επομένως και η τοπολογία μπορεί να αλλάζει με μεγαλύτερη συχνότητα από ότι στα ad hoc δίκτυα.
- Χρησιμοποιούνται ως επί το πλείστον τεχνικές broadcast μετάδοσης, ενώ στα ad hoc δίκτυα χρησιμοποιούνται συνδέσεις σημείο-προς-σημείο.
- Υπάρχουν κυρίαρχοι περιορισμοί σε ισχύ, μνήμη και ενέργεια.
- Μια πολύ σημαντική διαφορά είναι ότι η επικοινωνία κόμβων και βασικού σταθμού είναι δεδομενο-κεντρική (data-centric). Αυτό σημαίνει ότι ο κεντρικός σταθμός ενδιαφέρεται να μάθει πληροφορία για ορισμένες περιοχές του δικτύου, και όχι από συγκεκριμένους κόμβους. Δηλαδή, το πιο πιθανό είναι να στέλνει αιτήσεις του στυλ "οι κόμβοι που βρίσκονται στην περιοχή x να στείλουν μετρήσεις", και όχι "ο κόμβος 1380 να στείλει μετρήσεις".

Αυτές ήταν εν συντομία οι κύριες διαφορές των δικτύων έξυπνης σκόνης με τους κλασσικούς αισθητήρες και τα ad hoc δίκτυα. Στη συνέχεια, θα δούμε πως αυτές οι διαφορές μεταφράζονται σε παράγοντες που επηρεάζουν τη σχεδίαση ενός τέτοιου δικτύου.

1.3 Παράγοντες οι οποίοι επηρεάζουν συνολικά το σχεδιασμό ενός δικτύου έξυπνης σκόνης

1.3.1 Προσαρμοστικότητα του δικτύου

΄Ενα δίκτυο έξυπνης σκόνης έχει να αντιμετωπίσει δυσκολίες όπως συχνές αστοχίες υλικού, συχνά προβλήματα στην επικοινωνία μεταξύ κόμβων και μεγάλο μέγεθος και πυκνότητα δικτύου. Οι αλγόριθμοι, οι οποίοι ρυθμίζουν την επικοινωνία μεταξύ των κόμβων πρέπει να έχουν μεγάλες ανοχές σε λάθη, να

αντιλαμβάνονται γρήγορα πότε μια σύνδεση δε λειτουργεί σωστά και να δημιουργούν καινούριες. Επιπλέον, πρέπει να αντιμετωπίσουν τη μεγάλη πυκνότητα του δικτύου και να μπορούν να εφαρμόζονται σε περιπτώσεις στις οποίες έχουμε αρκετές χιλιάδες κόμβων.

1.3.2 Τοπολογία δικτύου

Η τοπολογία ενός δικτύου έξυπνης σκόνης μπορεί να είναι εντελώς τυχαία ή να ακολουθεί κάποιο πρότυπο. Μπορεί να γίνει ρίψη συσκευών από αεροπλάνο (η συγκεκριμένη μέθοδος έχει ήδη δοκιμαστεί), από καταπέλτη, με τα χέρια, κτλ. Μετά την τοποθέτηση των κόμβων η τοπολογία μπορεί να αλλάξει λόγω θέσης, συνδεσιμότητας, έλλειψης ενέργειας. Επιπλέον κόμβοι μπορεί να τοποθετηθούν, για να αντικαταστήσουν παλιούς που τέθηκαν εκτός λειτουργίας.

1.3.3 Μέσο μετάδοσης

Η επικοινωνία μεταξύ των κόμβων του δίκτυου μπορεί να γίνει με ραδιοσυχνότητες, ακτίνες laser ή υπέρυθρες. Από τις τρεις αυτές επιλογές, περισσότερο χρησιμοποιείται η πρώτη, ακολουθεί η δεύτερη με χρήση σε συγκεκριμένες εφαρμογές και η τρίτη μέχρι στιγμής δεν έχει χρησιμοποιηθεί κάπου. Για τις ραδιοσυχνότητες πρέπει να επιλεγεί ζώνη μετάδοσης, η οποία θα είναι διαθέσιμη για εμπορική χρήση, π.χ οι συχνότητες που χρησιμοποιούνται από τα wireless LAN. Τα συστήματα που περιγράφονται στο κεφάλαιο 6 χρησιμοποιούν τη ζώνη των 916 MHz.

1.3.4 Περιορισμοί υλικού και κόστους παραγωγής

΄Ενας κόμβος δικτύου έξυπνης σκόνης αποτελείται από τέσσερα βασικά τμήματα, έναν αισθητήρα(ή και περισσότερους), μία CPU, ένα πομποδέκτη και μια μονάδα ισχύος. Επίσης, είναι δυνατόν να υπάρχουν κι άλλα τμήματα, όπως μονάδα GPS ή κινητήρας. ΄Ολες αυτές οι μονάδες πρέπει να χωρούν σε μια συσκευασία μεγέθους σπιρτόκουτου ή ακόμα μικρότερη. Ακόμα, πρέπει να καταναλώνουν ελάχιστη ενέργεια, να είναι αυτόνομες και να λειτουργούν ανεξάρτητες, να προσαρμόζονται στο περιβάλλον τους, να αντέχουν σε αντίξοες συνθήκες. Επομένως, υπάρχουν πολύ μεγάλες απαιτήσεις στον τομέα του υλικού, και πρέπει η έρευνα στον τομέα της κατασκευής ολοκληρωμένων κυκλωμάτων να υποστηρίξει αυτές τις ανάγκες.

Στα παραπάνω έρχεται να προστεθεί η απαίτηση για πολύ μικρό κόστος παραγωγής ενός κόμβου. ΄Ενα δίκτυο έξυπνης σκόνης προφανώς είναι συμφέρον, αν κάθε κόμβος στοιχίζει τόσο λίγο ώστε το συνολικό κόστος να μην είναι απαγορευτικό. Εφόσον μιλάμε για χιλιάδες κόμβων, κάθε κόμβος πρέπει να στοιχίζει πολύ κάτω από 1$. Συγκριτικά, μια συσκευή Bluetooth σήμερα κοστίζει κοντά στα 10$.

1.3.5 Κατανάλωση ενέργειας

Η λειτουργία ενός κόμβου-αισθητήρα συνοψίζεται ως εξής: ανίχνευση γεγονότος, επεξεργασία πληροφορίας, μετάδοση ή λήψη μηνύματος. Επομένως, η

ενέργεια ενός κόμβου ξοδεύεται με τρεις τρόπους.

Επικοινωνία: Η περισσότερη ενέργεια καταναλώνεται εδώ, στην οποία περιλαμβάνεται τόσο η μετάδοση όσο και η λήψη δεδομένων. Γενικά, για μικρές αποστάσεις και για τις δύο περιπτώσεις καταναλώνεται περίπου η ίδια ενέργεια. Επίσης, λαμβάνουμε υπόψη μας και την ενέργεια για την αρχικοποίηση του πομποδέκτη. Επομένως, πρέπει να βρούμε έναν έξυπνο τρόπο να διαχειριστούμε τον πομποδέκτη και να ελαχιστοποιήσουμε μεταδόσεις και λήψεις μηνυμάτων. Τυπικές τιμές για αυτήν την περιοχή είναι $50\ nJ/bit$.

Επεξεργασία πληροφορίας: Εδώ, η κατανάλωση ενέργειας είναι πολύ μικρότερη σε σχέση με την ενέργεια για επικοινωνία. Μπορούμε να εκμεταλλευτούμε την ισχύ του επεξεργαστή και να κάνουμε συμπίεση δεδομένων ή μπορούμε ακόμα και να επεξεργαστούμε τα μηνύματα που λαμβάνουμε και να αφαιρέσουμε περιττή πληροφορία (π.χ το ίδιο μήνυμα έχει φθάσει από δύο κόμβους), προκειμένου να γλιτώσουμε bit μεταδιδόμενης πληροφορίας. Τυπική τιμή για τους σύγχρονους επεξεργαστές είναι 1 pJ/εντολή.

Μέτρηση πεδίου: Η ενέργεια που καταναλώνεται εδώ είναι σταθερή και περιλαμβάνει την ενέργεια για λήψη δείγματος από τους αισθητήρες ενός κόμβου. Προφανώς, μπορούμε να ελαττώσουμε την ενέργεια που καταναλώνουμε, όσο πιο αραιά κάνουμε μετρήσεις της ποσότητας που μας ενδιαφέρει (π.χ θερμοκρασία). Τυπικές τιμές σε αυτό το πεδίο είναι 4 nJ ανά δείγμα (ακρίβεια 10 bit/δειγμα).

1.3.6 Πηγές ενέργειας

Υπάρχουν αρκετές τεχνολογικές προτάσεις ως υποψήφιες πηγές ενέργειας για χρήση σε δίκτυα έξυπνης σκόνης, καθεμιά με τα πλεονεκτήματα και τα μειονεκτήματά της. Προφανώς αυτό που ενδιαφέρει περισσότερο είναι η διάρκεια ζωής και η αυτονομία που προσφέρουν στους κόμβους του δικτύου, αλλά υπάρχουν και άλλες σημαντικές παράμετροι όπως το μέγεθος, η ευκολία στη σύνδεση με τους κόμβους, κ.α. Ακολουθούν κάποιες χαρακτηριστικές περιπτώσεις.

Η τεχνολογία η οποία, προς το παρόν, φαίνεται πως είναι η πιο πιθανή λύση, είναι οι μπαταρίες λιθίου. Οι τελευταίες γενιές των μπαταριών αυτών είναι επαναφορτιζόμενες, γεγονός το οποίο είναι πολύ χρήσιμο, και με δυνατότητα πυκνότητας ενέργειας γύρω στα 2000 $Joule/cc$. Μια άλλη πρόταση από το χώρο των μπαταριών, είναι οι νέες μπαταρίες λεπτού φιλμ οξειδίου βαναδίου και οξειδίου μολυβδαινίου. Οι χωρητικότητες τους είναι ανάλογες των μπαταριών λιθίου και επιπλέον έχουν το πλεονέκτημα ότι μπορούν να ενσωματωθούν στους κόμβους πιο εύκολα.

Μια άλλη πρόταση είναι η χρήση φωτοηλεκτρικών κυττάρων, τα οποία παράγουν ενέργεια από το φως. Το φως αυτό μπορεί να προέρχεται είτε από τον ΄Ηλιο, είτε από κάποια άλλη φωτεινή πηγή, όπως ο εσωτερικός φωτισμός. Βέβαια, στη δεύτερη περίπτωση η παραγόμενη ενέργεια είναι κλάσμα της πρώτης. Η λύση αυτή μπορεί να χρησιμοποιηθεί περισσότερο ως ένας τρόπος να

αναπληρώνεται η ενέργεια μιας μπαταρίας, η οποία θα είναι η κύρια πηγή ενέργειας. Αν αυτό μπορεί να γίνεται με έναν ικανοποιητικό ρυθμό, τότε μπορεί να αυξηθεί αρκετά ο χρόνος ζωής ενός δικτύου έξυπνης σκόνης.

Μια κάπως πιο εξωτική πρόταση είναι η παραγωγή ενέργειας από την κίνηση του κόμβου. ΄Ετσι, π.χ ένας κόμβος που είναι τοποθετημένος σε μια πόρτα που ανοιγοκλείνει αρκετά συχνά, μπορεί να εκμεταλλευτεί την κίνηση αυτή. Δυστυχώς, η παραγόμενη ενέργεια στις περισσότερες περιπτώσεις είναι μικρή. Γενικά πάντως, έχουν γίνει κι άλλες πρωτότυπες προτάσεις, όπως η χρήση πρωτεϊνών για την παραγωγή ενέργειας σε περίπτωση που μιλάμε για δίκτυο έξυπνης σκόνης μέσα σε ζωντανό οργανισμό.

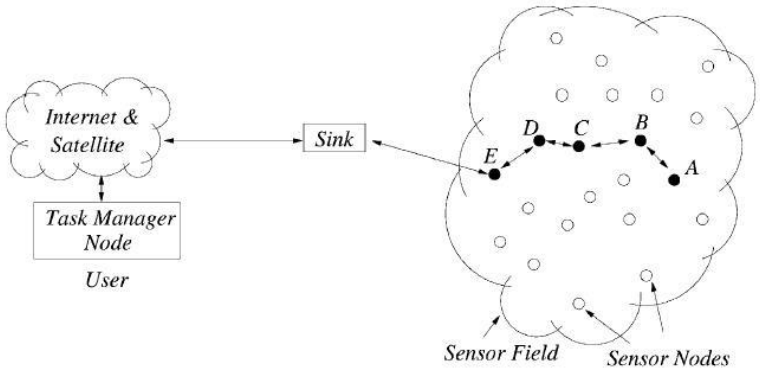
Μια ελπιδοφόρα τεχνολογία είναι οι μικροκυψέλες καυσίμου, οι οποίες επιπλέον είναι και φιλικές προς το περιβάλλον. Πρόκειται για κάψουλες που περιέχουν μικρές ποσότητες καυσίμου, όπως αιθανόλης, και μέσω κάποιου καταλύτη γίνεται μια αντίδραση η οποία παράγει ενέργεια. Το πρόβλημα εδώ είναι μάλλον το μέγεθος του συστήματος, το οποίο ίσως να είναι απαγορευτικό για τα δίκτυα έξυπνης σκόνης.

1.4 Αρχιτεκτονική δικτύου

΄Ενα δίκτυο έξυπνης σκόνης σε γενικές γραμμές αποτελείται από τους απλούς κόμβους του δικτύου (sensor nodes ή particles), οι οποίοι βρίσκονται διασκορπισμένοι σε μία περιοχή που μας ενδιαφέρει (sensor field), και από ένα βασικό σταθμό (sink, base station), ο οποίος συγκεντρώνει και αποθηκέυει ή προωθεί αλλού τις πληροφορίες που συλλέγουν οι κόμβοι μέσα στο πεδίο. Ο βασικός σταθμός μπορεί να είναι τοποθετημένος σε κάποια μικρή απόσταση από το πεδίο και θεωρούμε ότι έχει πολύ μεγαλύτερα αποθέματα ενέργειας, σε σχέση με τους απλούς κόμβους του δικτύου. Η ροή πληροφορίας προς το βασικό σταθμό επιτυγχάνεται μέσω της συνεργασίας των κόμβων του δικτύου (multihop επικοινωνία). Επίσης, θεωρούμε ότι ο βασικός σταθμός μπορεί να είναι συνδεδεμένος και με κάποιο άλλο δίκτυο, π.χ το Internet, με κάποιο τρόπο(ενσύρματο ή ασύρματο). Η οργάνωση αυτή φαίνεται στο σχήμα 1.1.

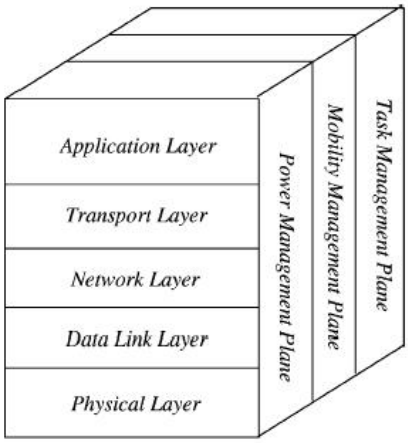
Η στοίβα με τα επίπεδα δικτύου φαίνεται στο σχήμα 1.2. Αποτελείται από τα επίπεδα δικτύου, όπως αυτά ορίζονται στο μοντέλο OSI αλλά χωρίς επίπεδο session, συν τρία νέα επίπεδα, τα οποία διατρέχουν ολόκληρη την ιεραρχία επιπέδων, τα power, mobility και task management επίπεδα. Εκτός των τριών τελευταίων, τα υπόλοιπα επίπεδα επιτελούν ακριβώς τις γνωστές λειτουργίες όπως τις γνωρίζουμε από τα IP δίκτυα. Τα τρία τελευταία παρακολουθούν την κατανομή κατανάλωσης ενέργειας και φόρτου εργασίας σε ένα δίκτυο έξυπνης σκόνης, καθώς και την διατήρηση της συνεκτικότητας του δικτύου.

Το *power management plane* διαχειρίζεται τον τρόπο με τον οποίο ένας κόμβος εκμεταλλεύεται τα αποθέματα ενέργειάς του. Π.χ ένας κόμβος μπορεί να κλείνει το δέκτη του για ένα διαστήμα αφού λάβει ένα μήνυμα, προκειμένου να μην το λάβει δύο φορές. Το *mobility management plane* παρακολουθεί την ύπαρξη γειτόνων στο δίκτυο ανά πάσα στιγμή, έτσι ώστε να υπάρχουν συνδέσεις



Σχήμα 1.1: Γενική άποψη ενός δικτύου έξυπνης σκόνης

μέσω των οποίων θα προωθηθεί πληροφορία προς το βασικό σταθμό. Γνωρίζοντας ποιοι είναι οι γείτονες ενός κόμβου μπορεί να κατανεμηθεί κι ο φόρτος εργασίας συνολικά στο δίκτυο. Δεν χρειάζεται να είναι ενεργοί όλοι οι κόμβοι που βρίσκονται σε μια μικρή περιοχή και να παίρνουν δείγματα ή να προωθούν μηνύματα συνεχώς. Το *task management plane* εξισορροπεί το φόρτο εργασίας και καθορίζει ποιος κόμβος είναι ενεργός ανά πάσα στιγμή. Π.χ οι κόμβοι που έχουν περισσότερη ενέργεια, αναλαμβάνουν περισσότερο ενεργό ρόλο από άλλους. Αυτά τα τρία επίπεδα χρειάζονται για τη σωστή συνεργασία μεταξύ των κόμβων και το διαμοιρασμό των πόρων, έτσι ώστε να επιμηκυνθεί η σωστή λειτουργία και η διάρκεια ζωής του δικτύου συνολικά.



Σχήμα 1.2: Τα επίπεδα δικτύου σε ένα δίκτυο έξυπνης σκόνης

1.5 Εφαρμογές δικτύων έξυπνης σκόνης

Αφού μελετήσαμε τα κύρια σήμεια που διέπουν το σχεδιασμό και τη λειτουργία ενός δικτύου έξυπνης σκόνης, θα δούμε μερικά σενάρια εφαρμογών αυτών των δικτύων, οι οποίες αναδεικνύουν τη χρησιμότητά τους, καθώς και το γεγονός ότι σε πολλές περιπτώσεις ανοίγουν νέα πεδία επιστημονικής έρευνας.

Λόγω της ποικιλίας των αισθητήρων που μπορεί να φέρει ένας κόμβος ενός δικτύου έξυπνης σκόνης, μπορούν να υπάρξουν πολλές εφαρμογές στις οποίες θα χρησιμοποιηθεί ένα τέτοιο δίκτυο. Βέβαια, καθώς η έρευνα και γενικά η ιδέα για τα δίκτυα έξυπνης σκόνης είναι ακόμα σε αρκετά πρώιμο στάδιο, δεν έχουν υλοποιηθεί ακόμα πολλές εφαρμογές, αλλά καθώς μεγαλώνει ο αριθμός των ομάδων που ασχολούνται με το θέμα, αυξάνονται και οι σχετικές υλοποιήσεις.

Ανάμεσα στα φαινόμενα που μπορούμε να παρακολουθήσουμε με τους υπάρχοντες αισθητήρες, ανήκουν τα παρακάτω:

- Θερμοκρασία
- Υγρασία
- Κίνηση αντικειμένου
- Πίεση
- Θόρυβος (ήχος γενικά)
- Χαρακτηριστικά κινούμενου αντικειμένου όπως κατεύθυνση, ταχύτητα και μέγεθος του αντικειμένου
- Ανίχνευση χημικών ουσιών

Στη συνέχεια παραθέτουμε αναφορικά κάποιες χαρακτηριστικές εφαρμογές, από τις οποίες μερικές έχουν ήδη υλοποιηθεί από ερευνητικές ομάδες. Επέλεξα να μην αναφερθώ σε στρατιωτικές εφαρμογές, σε αντίθεση με τα σχετικά surveys, καθώς θεωρώ ότι οι ωφέλειες από τις ειρηνικές εφαρμογές ενός τέτοιου δικτύου μπορούν εύκολα να υπερκεράσουν τις ωφέλειες από στρατιωτικές εφαρμογές.

Μικρο-γεωργία: Με τη χρήση ενός δικτύου έξυπνης σκόνης, μπορούμε να παρακολουθούμε τις συνθήκες που επικρατούν στις καλλιέργειες σε πραγματικό χρόνο, ενώ παράλληλα μπορούμε να μετράμε και τις ποσότητες μικροβιοκτόνων που περνούν στο πόσιμο νερό, το επίπεδο της διάβρωσης του εδάφους, αν σμήνη εντόμων επιτίθενται στις καλλιέργειες και άλλα πολλά χρήσιμα.

Καταγραφή της βιοποικιλότητας: Οι δορυφόροι που χρησιμοποιούνται για την παρατήρηση των δασικών εκτάσεων και του μεγέθους τους, δεν έχουν τη δυνατότητα να παρατηρήσουν τι γίνεται σε σχετικά μικρή κλίμακα. Με ένα δίκτυο έξυπνης σκόνης με δυνατότητες ανίχνευσης και αναγνώρισης αντικειμένων σε μικρή κλίμακα είναι δυνατή η καταγραφή της βιοποικιλότητας ακόμα

και των πιο μικροσκοπικών ζώων, π.χ των εντόμων. Αυτό είναι δυνατό γιατί κάθε κινούμενο αντικείμενο παράγει κάποια ηλεκτρομαγνητική διαταραχή στην περιοχή που κινείται καθώς και ένα ελαφρύ σεισμικό αποτύπωμα, από τα οποία είναι δυνατό έως βαθμό να γίνει μια αναγνώριση του αντικειμένου. Π.χ ένας ελέφαντας όταν βαδίζει προκαλεί διαφορετική διαταραχή από μια καμηλοπάρδαλη.

Παράδειγμα σχετικής εφαρμογής είναι το πείραμα στο νησί Great Duck στις ΗΠΑ, το οποίο είναι τόπος αναπαραγωγής ενός είδους πουλιού (Leach's storm petrel)[3]. Σε αυτό το νησί λοιπόν, ερευνητές τοποθέτησαν κόμβους έξυπνης σκόνης στις φωλιές των πουλιών για να παρακολουθούν τις συνθήκες που επικρατούν σε αυτές, καθώς επίσης και την αύξηση του πληθυσμού στο νησί. Στο σχήμα 1.3 φαίνεται ένας κόμβος έξυπνης σκόνης από αυτούς που χρησιμοποιήθηκαν στο συγκεκριμένο πείραμα, σε ειδική συσκευασία.

Σχήμα 1.3: ΄Ενας κόμβος έξυπνης σκόνης για το πείραμα στο Great Duck Island

Ανίχνευση πυρκαγιάς: Είναι δυνατόν να γίνει ρίψη κόμβων έξυπνης σκόνης από ένα αεροπλάνο ή ελικόπτερο ή και από ανθρώπους στο έδαφος, με κάποιο τυχαίο τρόπο σε μια δασική περιοχή. Δίνεται έτσι η δυνατότητα να ανιχνευθεί με ακρίβεια η τοποθεσία του μετώπου μιας πιθανής πυρκαγιάς σχεδόν αμέσως. Το πρόβλημα στην υλοποίηση ενός τέτοιου σεναρίου στη χώρα μας είναι ότι τα δάση μας είναι κυρίως πευκοδάση, που σημαίνει ότι οι πυρκαγιές παίρνουν πολύ γρήγορα μεγάλες διαστάσεις και επομένως είναι αμφισβητήσιμη η χρησιμότητα ενός τέτοιου δικτύου έξυπνης σκόνης. Επιπλέον, υπάρχει και το θέμα της ρύπανσης που προκαλούν οι κόμβοι του δικτύου όταν αχρηστευθούν, με τις

μπαταρίες που φέρουν.

Ανίχνευση επικίνδυνων ουσιών: Μετά τις τελευταίες εξελίξεις (επίθεση σε Twin Towers κτλ) αυτή η εφαρμογή απέκτησε μεγάλο ενδιαφέρον και πολλές ερευνητικές ομάδες ασχολούνται προς αυτή την κατεύθυνση. Στο πανεπιστήμιο του San Diego, στο τμήμα Χημικών Μηχανικών, εξετάζουν σενάρια στα οποία ένα δίκτυο έξυπνης σκόνης επιπλέει μέσα στο δίκτυο υδροδότησης, έτσι ώστε να μπορεί να ανιχνεύει πιθανές απόπειρες δηλητηρίασης του πληθυσμού μέσω του νερού.

Παρακολούθηση ιατρικών δεδομένων in vivo **και χορήγηση φαρμάκων:** Μια άλλη εντυπωσιακή εφαρμογή είναι η εισαγωγή κόμβων έξυπνης σκόνης στο ανθρώπινο σώμα, οι οποίοι θα παρακολουθούν και θα καταγράφουν συνεχώς τι συμβαίνει στο σώμα μας. Επίσης, θα μπορούν να χορηγούν φάρμακα τοπικά χωρίς να χρειάζεται να επιβαρυνθεί ολόκληρος ο οργανισμός, πράγμα πολύ χρήσιμο στην περίπτωση καρκινοπαθών που κάνουν χημειοθεραπεία. Το κατά πόσον είναι δυνατόν να υλοποιηθεί το σενάριο αυτό στο εγγύς μέλλον, είναι θέμα υπό συζήτηση.

Παρακολούθηση ηλικιωμένων σε γηροκομείο: Εκτός από τις προφανείς μετρήσεις θερμοκρασίας και πίεσης, οι οποίες μπορούν να αποσταλλούν σε ένα κεντρικό υπολογιστή από κόμβους έξυπνης σκόνης που θα προσδένονται σε ηλικιωμένους, θα μπορεί κάποιος επίσης να παρακολουθεί τις κινήσεις ενός ηλικιωμένου. Αυτό είναι εξαιρετικά χρήσιμο σε περίπτωσεις που ο συγκεκριμένος ηλικιωμένος πάσχει από την ασθένεια του Altsheimer ή αν συμβεί κάποιο ατύχημα (π.χ πτώση σε σκάλες), οπότε δεν θα χρειάζεται να υπάρχει συνεχώς κάποιος από το νοσηλευτικό προσωπικό για να επιβλέπει τους ηλικιωμένους.

Οικιακός αυτοματισμός: Είναι μια πολλά υποσχόμενη εφαρμογή, και ανάμεσα σε αυτά τα οποία υπόσχεται είναι και το ότι θα κάνει την τεχνολογία των δικτύων έξυπνης σκόνης σύντομα μέρος της καθημερινότητας μας. Αισθητήρες με δυνατότητες ελέγχου μπορούν να ενσωματωθούν σε οικιακές συσκευές, όπως φώτα, συστήματα ήχου και εικόνας, κλιματισμού, εξαερισμού, κτλ, και οι οποίες θα έχουν τη δυνατότητα επικοινωνίας μεταξύ τους καθώς και με ένα server, ώστε να ρυθμίζουν κατάλληλα τη λειτουργία τους. Όλα αυτά θα συνθέτουν αυτό που ονομάζεται "smart home environment". Ένα υποθετικό σενάριο είναι το παρακάτω: κάποιος/α γυρνά βράδυ στο σπίτι του από τη δουλειά, μετά από τις υπερωρίες που επιβάλλει το πρόγραμμα της εταιρίας στην οποία εργάζεται. Μόλις μπαίνει στο σπίτι, ανάβουν και σβήνουν τα φώτα αυτόματα από τα δωμάτια που περνά, το στερεοφωνικό αρχίζει και παίζει αυτόματα το τρίτο από τα Βραδεμβούργια κοντσέρτα του Μπαχ, κ.ο.κ. Αυτό το σενάριο δεν είναι τόσο μακριά όσο ίσως φαντάζεται ο αναγνώστης.

Έλεγχος περιβάλλοντος σε κτίρια γραφείων: Σήμερα ο κλιματισμός και ο εξαερισμός μεγάλων κτιρίων με γραφεία γίνεται συνήθως κεντρικά. Η θερμοκρασία μέσα στο ίδιο γραφείο μπορεί να έχει σημαντικές διακυμάνσεις και

η κυκλοφορία του αέρα να μην γίνεται ικανοποιητικά. Τα κτίρια αυτά επιπλέον έχουν προβλήματα λόγω του μεγέθους και του σχεδιασμού τους, ειδικά οι ουρανοξύστες παρουσιάζουν πολλά λειτουργικά προβλήματα και απαιτούν να είναι ανοιχτά τα φώτα και να λειτουργεί ο κλιματισμός καθ΄ όλη τη διάρκεια της ημέρας. Ένα σύστημα έξυπνης σκόνης μπορεί να διορθώσει αρκετά από τα προβλήματα αυτά, με τη σωστή διαχείριση των σχετικών συστημάτων. Σύμφωνα με μια πρόσφατη μελέτη, μόνο στις Ηνωμένες Πολιτείες μπορεί να εξοικονομηθεί ενέργεια αξίας 55 δισεκατομμυρίων δολαρίων ανά χρόνο ή αλλιώς, εκπομπές 35 εκατομμυρίων τόνων διοξειδίου του άνθρακα (που σημαίνει επιπλέον κέρδος, αφού σύμφωνα με την πρόσφατη συνθήκη του Κιότο εκπομπές καυσαερίων στο μέλλον ισοδυναμούν με πρόστιμο στην χώρα που τα παράγει).

Παρακολούθηση καιρικών φαινομένων: Σύμφωνα με το άρθρο [5], ένα δίκτυο έξυπνης σκόνης θα μπορούσε να αιωρείται μέσα στα σύννεφα για αρκετές μέρες και να δίνει λεπτομερή στοιχεία, απαραίτητα για την πρόβλεψη του καιρού από τους μετεωρολόγους. Αυτό με την προϋπόθεση ότι οι σχεδιαστές συστημάτων έξυπνης σκόνης θα μπορέσουν να κατασκευάσουν κόμβους πλάτους 100 μικρών και μήκους μερικών εκατοστών, το οποίο μέγεθος επιτρέπει την αιώρηση στην ατμόσφαιρα για μια εβδομάδα!

Παρακολούθηση εργαλειοθήκης: Ένα project που "τρέχει" την ώρα που μιλάμε, είναι το Ivy (κισσός) στο πανεπιστήμιο Berkeley [6]. Ο σκοπός του είναι να δημιουργηθεί ένα δίκτυο έξυπνης σκόνης για ερευνητικούς σκοπούς μέσα στο campus του πανεπιστημίου. Αποτελείται από τριών ειδών κόμβους:

1. βασικούς, οι οποίοι είναι σταθεροί και επιτελούν τη διασύνδεση ασύρματου και ενσύρματου δικτύου.

2. σταθερούς, οι οποίοι επιτελούν το έργο του ενδιάμεσου ανάμεσα στους κινητούς και τους βασικούς κόμβους.

3. κινητούς, οι οποίοι είναι προσδεμένοι σε διάφορα αντικείμενα.

Οι βασικοί κόμβοι επιπλέον συνδέονται σε μια βάση δεδομένων, της οποίας τα δεδομένα μπορούν να προσπελασθούν από την ιστοσελίδα του εργαστηρίου. Μέχρι στιγμής μια ενδιαφέρουσα εφαρμογή είναι ο εντοπισμός της θέσης των προβολέων (projectors),τους οποίους χρησιμοποιούν οι διδάσκοντες στις παραδόσεις τους. Μπορεί κάποιος να γνωρίζει ανά πάσα στιγμή ποιος έχει πάρει ένα προβολέα και σε ποια αίθουσα βρίσκεται. Όσοι έχουν ασχοληθεί με το θέμα, καταλαβαίνουν αμέσως τη χρησιμότητα της συγκεκριμένης εφαρμογής. Οι σχεδιαστές του συστήματος χρησιμοποίησαν τα smart dust motes που παρουσιάζονται σε επόμενη ενότητα, δηλαδή θεωρητικά το πείραμα θα μπορούσε να επαναληφθεί και στο τμήμα Μηχανικών Η/Υ εφόσον έχει αγοραστεί (σχεδόν) το ίδιο υλικό. Υπολογίζεται πως με δύο μπαταρίες ΑΑ σε κάθε κόμβο, το δίκτυο θα λειτουργήσει για ένα χρόνο!

1.6 Παρουσίαση εφαρμογής δύο σεναρίων για δίκτυα έξυπνης σκόνης

Ακολουθεί η ανάλυση για την κατανάλωση ενέργειας σε δύο σενάρια εφαρμογής δικτύων έξυπνης σκόνης, όπως αυτές παρουσιάζονται στο [4]. Πιο συγκεκριμένα, το πρώτο σενάριο αφορά στην παρακολούθηση των συνθηκών σε μεγάλα κτίρια και το δεύτερο στην ανίχνευση εισβολέων σε πεδίο. Θα κάνουμε τις παρακάτω παραδοχές, ώστε να μπορέσουμε να μελετήσουμε τα δυο σενάρια:

- *Ενέργεια*: Υποθέτουμε ότι χρησιμοποιούμε μπαταρίες λιθίου ως κύρια πηγή ενέργειας. Η ενέργεια που χωράει σε μια τέτοια μπαταρία είναι $2J/mm^3$. Χρησιμοποιούμε φωτοηλεκτρικά κύτταρα για να αναπληρώνουμε μέρος της ενέργειας που καταναλώνεται. Αν χρησιμοποιούμε ηλιακό φως στο ύπαιθρο η απόδοση των κυττάρων είναι $0.3mW/mm^2$, ενώ στο εσωτερικό ενός κτιρίου είναι $0.3\mu W/mm^2$ (το ένα χιλιοστό της πρώτης περίπτωσης).
- *Υπολογισμοί*: 1 pJ/εντολή.
- *Δειγματοληψία*: Για ένα δείγμα 10 bit ξοδεύουμε $4\ nJ$.
- *Επικοινωνία*: Για επικοινωνία με ραδιοσυχνότητες ξοδεύουμε $100\ nJ/bit$.

Παρακολούθηση συνθηκών σε κτίρια : υποθέτουμε πως έχουμε κόμβους οι οποίοι μετρούν τη θερμοκρασία, την υγρασία και το φωτισμό στο γραφείο που είναι εγκατεστημένοι. Κατόπιν επικοινωνούν ασύρματα με ένα δέκτη για να μεταδώσουν αυτές τις πληροφορίες, ο οποίος με τη σειρά του τις στέλνει σε ένα κεντρικό υπολογιστή, ο οποίος ελέγχει τον κλιματισμό, τον εξαερισμό και το φωτισμό για όλο το κτίριο.

Στην περίπτωση αυτή το δίκτυο είναι το πιο απλό που υπάρχει αφού η επικοινωνία με το δέκτη είναι single-hop. Προφανώς ο χρόνος ζωής των κόμβων επηρεάζεται από τη συχνότητα μετρήσεων και εκπομπών. Για τους τρεις αισθητήρες χρειαζόμαστε ενέργεια $12\ nJ$ για κάθε μέτρηση και $3\ \mu J$ για μετάδοση του σχετικού μηνύματος στο δέκτη του γραφείου. Η ενέργεια για υπολογισμούς είναι αμελητέα, καθώς απλά χρειάζεται εναλλαγή μεταξύ δύο καταστάσεων, η οποία υποθέτουμε ότι γίνεται κάθε δευτερόλεπτο και καθ' όλη τη διάρκεια της ημέρας. Έτσι, η απαιτούμενη ημερήσια ενέργεια είναι $300\ mJoule$. Μόνο με την ενέργεια της μπαταρίας ο χρόνος ζωής είναι περίπου 1 εβδομάδα/mm^3. Με την προσθήκη και μίας επιφάνειας φωτοκυττάρων $9\ cm^2$, το σύστημα έχει άπειρο χρόνο ζωής.

Ανίχνευση κίνησης οχημάτων: Η ομάδα Smart Dust του πανεπιστημίου Berkeley [20], έκανε μία επίδειξη το 2001 μιας τέτοιας εφαρμογής, στη διάρκεια της οποίας έριξαν έναν μικρό αριθμό από motes σε μια περιοχή της ερήμου για την ανίχνευση στρατιωτικών οχημάτων, τα οποία διέσχιζαν έναν δρόμο στην περιοχή αυτή. Κάθε κόμβος του δικτύου δειγματοληπτούσε με συχνότητα $30\ Hz$

με το μαγνητόμετρό του το πεδίο και έστελνε ένα μήνυμα μεγέθους 30 byte όταν ανίχνευε κάποιο όχημα να περνάει. Το μήνυμα κατευθυνόταν προς ένα κεντρικό σταθμό μέσω μια απλής τοπολογίας, συνολικά το πολύ τριών βημάτων. Η κίνηση στο δρόμο ήταν 100 οχήματα την ημέρα. Χρησιμοποιήθηκαν τα `Rene` motes, δηλαδή προηγούμενης γενιάς από αυτά που υπάρχουν διαθέσιμα σήμερα και είχαν την εξής κατανάλωση:

- 1 mJ ανά ημέρα για δειγματοληψία.
- 2 nJ ανά ημέρα για επικοινωνία με το σταθμό.
- 1 nJ ημέρα για υπολογισμούς.

Με μια απλή μπαταρία 3V (όπως αυτές που χρησιμοποιούνται στα ρολόγια χειρός) το δίκτυο λειτούργησε κανονικά για τρεις ημέρες. Με τα σημερινά motes (`MICA`) τα οποία έχουν υποδεκαπλάσια κατανάλωση, μπορούμε να περιμένουμε λειτουργία για πάνω από ένα μήνα.

1.7 Συνεισφορά-οργάνωση της διπλωματικής

Η εργασία αυτή εστιάζει στα πρωτόκολλα επιπέδου δικτύου για διάδοση πληροφορίας σε δίκτυα έξυπνης σκόνης. Προσφέρει το θεωρητικό υπόβαθρο για να εμβαθύνει κάποιος στα δίκτυα έξυπνης σκόνης και να κατανοήσει τα ζητήματα που ανακύπτουν στη διάρκεια της σχεδίασης ενός πρωτοκόλλου για τέτοια δίκτυα. Καλύπτει από *όλες* τις πλευρές το συγκεκριμένο θέμα, καθώς εκτός από το θεωρητικό μέρος περιλαμβάνει και εξομοίωση δικτύων έξυπνης σκόνης, καθώς και πειραματική αξιολόγηση σχετικών πρωτοκόλλων. Επίσης, περιλαμβάνει υλοποίηση εφαρμογών σε ένα πραγματικό πειραματικό δίκτυο.

Από πλευράς οργάνωσης, η εργασία χωρίζεται στα παρακάτω τρία μέρη:

1. *Πρωτόκολλα για δίκτυα έξυπνης σκόνης:* Στο κεφάλαιο 2 περιγράφονται τα χαρακτηριστικά και τα μοντέλα για τα πρωτόκολλα δικτύου για δίκτυα έξυπνης σκόνης. Αναλύονται επισης κάποια χαρακτηριστικά πρωτόκολλα, τα οποία δίνουν το στίγμα της τρέχουσας έρευνας στον τομέα. Στη συνέχεια, στο κεφάλαιο 3, εξετάζουμε ξεχωριστά τα πρωτόκολλα TEEN και LEACH.

2. *Εξομοίωση πρωτοκόλλων για δίκτυα έξυπνης σκόνης:* Στα πλαίσια της εργασίας αυτής, αναπτύχθηκε ένας εξομοιωτής δικτύων έξυπνης σκόνης, τον οποίο καλούμε simDust, για τρία πρωτόκολλα (LTP, PFR, TEEN), ο οποίος παρέχει και δυνατότητες οπτικοποίησης (animation) δίκτυων έξυπνης σκόνης. Στο κεφάλαιο 4, περιγράφεται η υλοποίηση του simDust, ενώ στο επόμενο κεφάλαιο περιγράφονται κάποια πειράματα που έγιναν με τον εξομοιωτή αυτόν για τα πρωτόκολλα PFR και TEEN και τα σχετικά αποτελέσματα.

3. *Ανάπτυξη και λειτουργία πειραματικού δικτύου:* Στο κεφάλαιο 6 γίνεται μια αναλυτική περιγραφή της πλατφόρμας TinyOS-Smart Dust, τόσο σε επίπεδο υλικού όσο και λογισμικού. Τέλος, στο κεφάλαιο 7 γίνεται η περιγραφή της ανάπτυξης κάποιων εφαρμογών σε ένα πραγματικό πειραματικό δίκτυο.

Κεφάλαιο 2

Επιλεγμένη έρευνα στα πρωτόκολλα για δίκτυα έξυπνης σκόνης

Στο κεφάλαιο αυτό θα ασχοληθούμε αρχικά με κάποιους από τους παράγοντες που επηρεάζουν αποφασιστικά τη σχεδίαση πρωτοκόλλων για το επίπεδο δικτύου στα δίκτυα έξυπνης σκόνης, καθώς και με τα μοντέλα που χρησιμοποιούμε για την κατηγοριοποίησή τους. Κατόπιν, θα μελετήσουμε τέσσερα τέτοια πρωτόκολλα, συγκεκριμένα το Sleep-awake, το PFR, το Directed Diffusion και το STEM.

2.1 Μοντέλα για βασικά χαρακτηριστικά των δικτύων έξυπνης σκόνης

΄Ενα πρωτόκολλο επιπέδου δικτύου για δίκτυα έξυπνης σκόνης είναι υπεύθυνο για τη διεξαγωγή όλης της επικοινωνίας, πρώτα μεταξύ των κόμβων του δικτύου και έπειτα μεταξύ των κόμβων και του βασικού σταθμού. Η απόδοσή του επηρεάζεται σε μεγάλο βαθμό από τη δυναμική αλλαγής της τοπολογίας του δικτύου και από το μοντέλο που χρησιμοποιείται για τη συλλογή και παράδοση πληροφορίας από τους κόμβους του δικτύου. Γίνεται μια πρώτη κατηγοριοποίηση αυτών των παραγόντων και βλέπουμε τα μοντέλα που χρησιμοποιούνται σήμερα. Ο αναγνώστης που ενδιαφέρεται να εμβαθύνει στο θέμα μπορεί να ανατρέξει στο [7] για περισσότερες λεπτομέρειες.

2.1.1 Μοντέλα επικοινωνίας

Γενικά, η επικοινωνία που διεξάγεται ανάμεσα στους κόμβους του δικτύου μπορεί να χωρισθεί σε δύο κατηγορίες: επικοινωνία για μεταφορά πληροφορίας και αιτήσεων και επικοινωνία για δημιουργία συνδέσεων στο δίκτυο. Το πρωτόκολλο πρέπει να υλοποιεί και τους δύο τύπους επικοινωνίας.

Στον πρώτο τύπο επικοινωνίας περιλαμβάνουμε τα μηνύματα-αιτήσεις που

στέλνει ο βασικός σταθμός και τα μηνύματα-πληροφορίες που στέλνουν οι κόμβοι του δικτύου. Υπάρχουν δύο μοντέλα εδώ, το *συνεργατικό* και το *ατομικό*. Στο *συνεργατικό* μοντέλο, οι κόμβοι του δικτύου συνεργάζονται για να μεταφέρουν όσο το δυνατόν πιο ακριβή πληροφορία από το δίκτυο στον κεντρικό σταθμό. Μπορούν π.χ να συνεννοούνται για να κάνουν συντονισμένη δειγματοληψία ή οι κόμβοι να επεξεργάζονται τις πληροφορίες που λαμβάνουν από τους γείτονες τους για να τις προωθήσουν στον κεντρικό σταθμό και να προσθέτουν τις δικές τους πληροφορίες ή να συνενώνουν πολλά τέτοια μηνύματα(data aggregation). Στο *ατομικό* μοντέλο αντίθετα, δεν συμβαίνει κάτι τέτοιο.

Στο δεύτερο τύπο επικοινωνίας, περιλαμβάνονται όλα τα μηνύματα που ανταλλάσσονται μεταξύ των κόμβων του δικτύου για να εγκαταστήσουν συνδέσεις μεταξύ τους, ώστε να μπορέσει να περάσει η πληροφορία από τους κόμβους στο βασικό σταθμό. Πιο συγκεκριμένα, πρέπει οι κόμβοι να είναι σε θέση να βρίσκουν μονοπάτια προς άλλους κόμβους ή το βασικό σταθμό, ανεξάρτητα από το αν αλλάζει η τοπολογία του δικτύου. Η τοπολογία του δικτύου μοιραία αλλάζει σε κάποια στιγμή, ανεξάρτητα από το αν οι κόμβοι είναι κινητοί ή όχι, αφού εξαντλούν κάποτε τα αποθέματα ενέργειάς τους. Επομένως, χρειάζεται επικοινωνία εκ νέου για να διατηρηθεί η συνεκτικότητα και η ομαλή λειτουργία του δικτύου και επίσης, στο βαθμό που αυτό είναι δυνατόν, να βελτιστοποιηθεί η απόδοση του δικτύου. Ουσιαστικά αυτή η επικοινωνία είναι το overhead του πρωτοκόλλου και ο σκοπός είναι να το ελαχιστοποιήσουμε, διατηρώντας ταυτόχρονα ανεκτά επίπεδα απόδοσης.

Σε πολλά από τα πρωτόκολλα για δίκτυα έξυπνης σκόνης χρειάζεται μια φάση προεργασίας για την ομαλή λειτουργία του δικτύου, π.χ σε πρωτόκολλα με clusters όπως το LEACH και το TEEN, τα οποία εξετάζουμε στο επόμενο κεφάλαιο, υπάρχει μια φάση που δημιουργούνται οι clusters και γενικά μια ιεραρχία δρομολόγησης. Επίσης, άλλα πρωτόκολλα προκειμένου να βελτιστοποιήσουν την απόδοση του δικτύου χρησιμοποιούν μηνύματα, π.χ του τύπου ότι ο κόμβος x πρέπει να ξυπνήσει αμέσως κτλ.

Ο αριθμός των μηνυμάτων που ανταλλάσσεται στο δίκτυο είναι καταλυτικός για την ενέργεια που καταναλώνεται από τους κόμβους και επηρεάζεται σε πολύ μεγάλο βαθμό από το πρωτόκολλο δικτύου. Η επικοινωνία για μετάδοση πληροφορίας μειώνεται, αν θέσουμε τη μικρότερη συχνότητα δειγματοληψίας που θα ικανοποιεί τις απαιτήσεις μας για απόδοση του δικτύου, δεδομένων των συνδέσεων που υπάρχουν στο δίκτυο. Η επικοινωνία για εγκατάσταση συνδέσεων μεταξύ των κόμβων παράγεται από το πρωτόκολλο δικτύου, για να ικανοποιήσει τις απαιτήσεις για πληροφορία του κεντρικού σταθμού.

2.1.2 Μοντέλα ανάκτησης πληροφορίας

Τα δίκτυα έξυπνης σκόνης μπορούν να κατηγοριοποιηθούν ανάλογα με το μοντέλο ανάκτησης πληροφορίας που χρησιμοποιούν σε *συνεχή* (continuous), *οδηγούμενα από γεγονότα* (event-driven), *κεντροποιημένα* (observer-initiated) και *υβριδικά*. Αυτά τα μοντέλα καθορίζουν την κίνηση πληροφορίας που δημιουργείται μέσα στο δίκτυο.

Στο *συνεχές* μοντέλο (ή proactive όπως αναφέρεται στο [9]) οι κόμβοι του δικτύου δειγματοληπτούν με σταθερή συχνότητα και στέλνουν τα αποτελέσματά τους στο βασικό σταθμό σε προκαθορισμένα χρονικά διαστήματα. Παράδειγμα τέτοιου πρωτοκόλλου είναι το LEACH. Στο *οδηγούμενο από γεγονότα* μοντέλο (ή reactive όπως αναφέρεται στο [9]), οι κόμβοι δειγματοληπτούν συνεχώς αλλά στέλνουν αποτελέσματα μόνο όταν πληρούνται κάποιες προϋποθέσεις. Παράδειγμα τέτοιου πρωτοκόλλου είναι το TEEN. Στο *κεντροποιημένο* μοντέλο οι κόμβοι στέλνουν πληροφορία μόνο όταν τους το ζητηθεί από το βασικό σταθμό. Το πρωτόκολλο Directed Diffusion θα μπορούσε να χαρακτηρισθεί ως τέτοιο, αν και εκεί δεν υπάρχει η έννοια του βασικού σταθμού αφού κάθε κόμβος μπορεί να δράσει ως τέτοιος. Το *υβριδικό* μοντέλο συνδυάζει τα παραπάνω μοντέλα.

2.1.3 Μοντέλα δυναμικής του δικτύου

Ανάλογα με τη δυναμική ενός δικτύου έξυπνης σκόνης ξεχωρίζουμε τις εξής δύο περιπτώσεις:

- **Στατικά δίκτυα:** Στα στατικά δίκτυα οι κόμβοι του δικτύου, ο βασικός σταθμός και το φαινόμενο που παρατηρούν, δεν εμπλέκονται σε καμιά κίνηση. Κλασσικό παράδειγμα είναι ένα δίκτυο έξυπνης σκόνης που μετρά τη θερμοκρασία ενός πεδίου. Τα στατικά δίκτυα προφανώς είναι η πιο απλή περίπτωση που θα μπορούσαμε να έχουμε.

- **Δυναμικά δίκτυα:** Στα δυναμικά δίκτυα είτε οι ίδιοι οι κόμβοι του δικτύου, είτε ο βασικός σταθμός, είτε το παρατηρούμενο φαινόμενο είναι κινητά. Όταν ένας κόμβος μετακινηθεί από τη θέση του μπορεί η σύνδεση που χρησιμοποίησε σε κάποια προηγούμενη μετάδοση πληροφορίας να μην είναι λειτουργική πλέον. Σε αυτή την περίπτωση πρέπει να βρεθεί νέα διαδρομή.

Τα δυναμικά δίκτυα μπορούν να κατηγοριοποιηθούν παραπέρα λαμβάνοντας υπόψη την κίνηση των επιμέρους συστατικών τους. Αυτή η κίνηση είναι σημαντική γιατί προφανώς καθορίζει το είδος και το μέγεθος της επικοινωνίας μέσα στο δίκτυο. Γενικά, ανάλογα με το αν κινείται ο βασικός σταθμός ή οι κόμβοι ή το παρατηρούμενο φαινόμενο, φαίνεται πως χρειάζεται διαφορετική αντιμετώπιση από το πρωτόκολλο δικτύου.

Κινητός βασικός σταθμός: Σε αυτή την περίπτωση ο βασικός σταθμός δεν παραμένει σταθερός, οπότε οι κόμβοι του δικτύου πρέπει προφανώς να αλλάζουν συνεχώς τα μονοπάτια προς αυτόν. Κλασσικό παράδειγμα είναι ένα αεροπλάνο που λειτουργεί ως βασικός σταθμός και πετά πάνω από το πεδίο που βρίσκεται το δίκτυο έξυπνης σκόνης.

Κινητοί κόμβοι: Σε αυτή την περίπτωση οι κόμβοι κινούνται και ως προς το βασικό σταθμό και μεταξύ τους. Παράδείγμα τέτοιας εφαρμογής είναι ένα δίκτυο

έξυπνης σκόνης με κάθε κόμβο προσαρμοσμένο σε ένα ταξί. Οι κόμβοι πρέπει (και μπορούν) να επικοινωνούν μόνο με τους γείτονές τους, οπότε χρειάζεται άλλος τρόπος να δημιουργούνται μονοπάτια προς το βασικό σταθμό.

Κινητό παρατηρούμενο φαινόμενο: Εδώ το φαινόμενο που παρατηρούν οι κόμβοι του δικτύου αλλάζει θέση. Κλασσικό παράδειγμα είναι μια εφαρμογή ανίχνευσης κίνησης. Προφανώς ένα τέτοιο δίκτυο είναι προτιμότερο να είναι οδηγούμενο από συμβάντα (event-driven) γιατί έχουμε μεγάλη εξοικονόμηση ενέργειας και ουσιαστικά είναι ενεργοί μόνο οι κόμβοι που βρίσκονται στην περιοχή όπου έχουμε ένα συμβάν.

Πρέπει να τονιστεί ότι η κίνηση στα δίκτυα έξυπνης σκόνης αντιμετωπίζεται διαφορετικά από ότι στα συμβατικά ασύρματα δίκτυα. Στα συμβατικά δίκτυα μας ενδιαφέρει η εξυπηρέτηση των κινητών σταθμών και δίνουμε ιδιαίτερη σημασία σε αυτούς, για αυτό και υπάρχουν διαδικασίες ανάθεσης χρηστών, συχνοτήτων κτλ, που είναι πολύ βασικές σε αυτά τα δίκτυα. Αντίθετα, στα δίκτυα έξυπνης σκόνης οι κόμβοι από μόνοι τους δεν παίζουν ιδιαίτερο ρόλο ο καθένας. Είναι πιθανό να μην υπάρχουν καν μοναδικοί ID αριθμοί για τον κάθε κόμβο. Μας ενδιαφέρει να λειτουργεί σωστά το δίκτυο στην εκάστοτε περιοχή ανεξάρτητα από το ποιοι κόμβοι βρίσκονται σε αυτή, και να μπορούμε να παρατηρήσουμε τι συμβαίνει εκεί. Επομένως, το πρόβλημα πρέπει να αντιμετωπιστεί διαφορετικά από ότι σε ένα κλασσικό `ad-hoc` δίκτυο.

2.2 Παρουσίαση πρωτοκόλλων για δίκτυα έξυπνης σκόνης

2.2.1 Το πρωτόκολλο sleep-awake

Το πρωτόκολλο αυτό [12], είναι ουσιαστικά μια επέκταση του πρωτοκόλλου LTP, το οποίο περιγράφεται στο extended abstract [13] το οποίο εξέδωσαν οι ίδιοι συγγραφείς. Το μοντέλο δικτύου και η ορολογία που χρησιμοποιούνται για τη θεωρητική ανάλυση είναι ουσιαστικά ίδια, οπότε ο αναγνώστης που ενδιαφέρεται για τη θεωρητική ανάλυση του συγκεκριμένου πρωτοκόλλου μπορεί να ανατρέξει και στο [13] για περισσότερες λεπτομέρειες.

΄Ενα νέο πρωτόκολλο προτείνεται, αποκαλούμενο sleep-awake, το οποίο χρησιμοποιεί τα sleep mode που διαθέτουν όλοι οι σύγχρονοι επεξεργαστές, προκειμένου να διασωθεί η ενέργεια των κόμβων. Ακόμα, παράμετροι όπως η πυκνότητα του δικτύου, το μήκος της περιόδου στην οποία ο κόμβος βρίσκεται σε κατάσταση "ύπνου" και οι άγρυπνες χρονικές περίοδοι, έχουν σοβαρό αντίκτυπο στην αποδοτικότητα του πρωτοκόλλου. Η αλληλεπίδρασή τους μελετάται στο [12] μέσω θεωρητικής και πειραματικής ανάλυσης.

Μοντέλο δικτύου που χρησιμοποιείται

Κάνουμε αρχικά την υπόθεση ότι οι κόμβοι του δικτύου μετά την τοποθέτησή τους στο πεδίο, δεν μετακινούνται από τη θέση τους και ότι έχουν περιορισμένα αποθέματα ενέργειας. Η πληροφορία για κάποιο γεγονός που ανιχνεύεται

μέσα στο πεδίο από κάποιον κόμβο, πρέπει να μεταδοθεί και να φτάσει σε ένα κεντρικό σταθερό σταθμό, ο οποίος βρίσκεται εκτός του πεδίου.

Κάθε κόμβος έχει δύο modes επικοινωνίας. Το πρώτο, το οποίο καλείται *beacon mode*, είναι ουσιαστικά μια εκπομπή σε περιορισμένη ακτίνα και μέσα σε κάποια γωνία και χρησιμοποιείται για να βρεθούν οι γείτονες που μπορούν να προωθήσουν μηνύματα προς τον κεντρικό σταθμό στο επόμενο βήμα. Το δεύτερο mode είναι μια επικοινωνία *point-to-point* μεταξύ δύο κόμβων. Η πρώτη μέθοδος επικοινωνίας επιτυγχάνεται με κάποια κατευθυνόμενη κεραία ραδιοσυχνοτήτων και η δεύτερη με χρήση laser. Οι υποθέσεις αυτές είναι λογικές, καθώς υπάρχουν ήδη κόμβοι έξυπνης σκόνης, οι οποίοι έχουν τις δυνατότητες που αναφέρθηκαν. Ακόμα, κάθε κόμβος μπορεί να μεταπηδά από την κατάσταση λειτουργίας σε μια κατάσταση λήθαργου, κατά την οποία δεν είναι δυνατή η επικοινωνία με το υπόλοιπο δίκτυο και στην οποία καταναλώνει ελάχιστη ενέργεια, τουλάχιστον σε σχέση με την κατάσταση λειτουργίας.

Η πυκνότητα των κόμβων στο πεδίο συμβολίζεται με d, και η ακτίνα μετάδοσης στο *beacon mode* με R. Μια άλλη ιδέα που χρησιμοποιείται είναι του *τείχους (wall) W*, που είναι μια ευθεία γραμμή στα άκρα του πεδίου και συμβολίζει τον κεντρικό σταθμό. Χρησιμοποιείται για να απλοποιήσει τη θεωρητική ανάλυση. Τέλος, υποθέτουμε ότι γίνεται κάποια αρχικοποίηση στους κόμβους ώστε να ξέρουν τη γενική κατεύθυνση προς το τείχος, ώστε να προωθούν κατάλληλα τα μηνύματα που λαμβάνουν από τους γείτονές τους.

Το πρωτόκολλο αναλυτικότερα - πώς δουλεύει

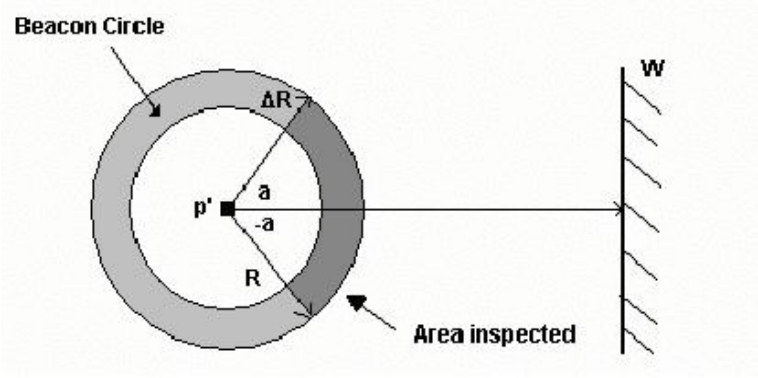
Όπως τα πρωτόκολλα στο [13], όταν λαμβάνει ένας κόμβος p τις πληροφορίες (e) από τον κόμβο p', θα ψάξει αρχικά για ένα κόμβο ο οποίος δεν βρίσκεται σε sleep mode προς την κατεύθυνση του τοίχου W. Υποθέτουμε ότι βρίσκει ένα τέτοιο κόμβο(τον p''), τότε στέλνει τις πληροφορίες (e) στο p' μέσω μιας απευθείας μετάδοσης και στέλνει έπειτα ένα μήνυμα επιτυχίας πίσω στο p, από το οποίο αρχικά έλαβε τις πληροφορίες. Η διαφορά με το πρωτόκολλο LTP[12] εδώ είναι ότι δηλώνεται ρητά ότι κάθε κόμβος θα μείνει άγρυπνος από τη στιγμή που διαβιβάζει κάποιες πληροφορίες έως ότου λαμβάνει ένα μήνυμα επιτυχίας ή αποτυχίας.

Και τώρα πάμε στις εσωτερικές εργασίες του πρωτοκόλλου. Κάθε κόμβος συνεχώς βρίσκεται μέσα σε ένα βρόχο μέσω μιας διαδικασίας που αλλάζει περιοδικά την κατάστασή του μεταξύ του ύπνου και της κανονικής λειτουργίας. Οι χρονικές περίοδοι ύπνου και κανονικής λειτουργίας είναι σταθερές και προκαθορισμένες. Όταν ένας κόμβος κοιμάται δεν μπορεί να επικοινωνήσει με το υπόλοιπο δίκτυο. Την αλλαγή μεταξύ καταστάσεων επιτελεί ένα σήμα διακοπής στον επεξεργαστή. Εάν ένας κόμβος είναι άγρυπνος και λάβει ένα μήνυμα, ένα μήνυμα διακοπής δημιουργείται και αναλόγως με τον τύπο μηνύματος καλείται ο αντίστοιχος handler (Receive Info, Fail, Success Handler).

Οι κόμβοι ξεκινούν αρχικά σε AWAKE mode. Το σημαντικό εδώ είναι να κοιμηθούν οι κόμβοι για πρώτη φορά σε διαφορετικούς χρόνους, που σημαίνει ότι δεν μπορούν να είναι άγρυπνοι για την ίδια χρονική περίοδο και να πάνε

όλοι ταυτόχρονα σε sleep mode, γιατί τότε προφανώς δε θα λειτουργούσε το δίκτυο καθόλου έως ότου ξυπνήσουν! Έτσι, πηγαίνουν στον ύπνο για πρώτη φορά με τυχαίο τρόπο, δηλαδή όχι όλοι ταυτόχρονα.

Υπάρχουν δύο δυαδικές μεταβλητές (`HOLDER`, `EXECUTING`) σε κάθε κόμβο. Η πρώτη μεταβλητή χρησιμοποιείται για να δείχνει εάν ο κόμβος έχει λάβει μήνυμα πληροφορίας από κάποιον γείτονά του. Η δεύτερη χρησιμοποιείται εάν ο κόμβος έχει διαβιβάσει τις πληροφορίες σε κάποιον γείτονά του και περιμένει μήνυμα επιτυχίας ή επιτυχίας από αυτόν. Επίσης, καταγράφουμε τον κόμβο από τον οποίο προήλθαν οι πληροφορίες σε μια μεταβλητή `PREVIOUS` και τους κόμβους που επέστρεψαν μηνύματα αποτυχίας σε μια λίστα `OUTF` (ώστε να γνωρίζουμε μελλοντικά ποιοι γείτονες εμφανίζουν πρόβλημα). Η φάση αναζήτησης υλοποιείται μέσω μιας διαδικασίας που ανιχνεύει τους κόμβους οι οποίοι βρίσκονται μέσα στην ακτίνα μετάδοσης, και επιλέγεται αυτός με τη μεγαλύτερη απόσταση. Πρακτικά, η περιοχή που επιθεωρείται μειώνεται από τη γωνία α και την ακτίνα ΔR (η οποία καθορίζεται μαζί με την ακτίνα R), όπως φαίνεται στην παρακάτω εικόνα. Η διαδικασία που εφαρμόζεται σε αυτήν την φάση επιστρέφει ένα σύνολο των γειτόνων που αποκρίθηκαν ταξινομημένο με βάση την απόσταση τους από τον κόμβο. Ο γείτονας με τη μεγαλύτερη απόσταση επιλέγεται ως το επόμενο βήμα στη διαδρομή προς το W.



Σχήμα 2.1: Η περιοχή στην οποία ένας κόμβος ψάχνει για γείτονες

Ανάλυση του πρωτοκόλλου

Στο [12] παρέχεται μια ανάλυση της αλληλεπίδρασης μεταξύ του ποσοστού επιτυχημένων μεταδόσεων, της διάρκειας των χρονικών περιόδων ύπνου και λειτουργίας και της πυκνότητας του δικτύου. Να σημειωθεί ότι είναι μια από τις ελάχιστες περιπτώσεις που συμβαίνει αυτό, καθώς στις περισσότερες εργασίες απλά παρουσιάζονται αποτελέσματα μιας εξομοίωσης χωρίς κάποιο θεωρητικό υπόβαθρο.

Αρχικά, βρίσκουμε τον αριθμό των κόμβων που βρίσκονται μέσα στην περιοχή η οποία ορίζεται από τη γωνία αναζήτησης α και τα R και ΔR, και ο

οποίος είναι:

$$N = \frac{d\alpha}{\pi}(\pi R^2 - \pi(R - \Delta R)^2) = 2\alpha \cdot R \cdot \Delta R \cdot d$$

Οι περίοδοι ύπνου και λειτουργίας έχουν αντίστοιχα διάρκεια S και W. Η πιθανότητα ότι κατά τη φάση αναζήτησης τουλάχιστον ένας κόμβος θα είναι σε λειτουργία είναι

$$P = 1 - (\frac{s}{s+w})^N$$

αφού η πιθανότητα ένας κόμβος να κοιμάται είναι $\frac{s}{s+w}$ και υψώνουμε σε δύναμη λαμβάνοντας υπόψη τους κόμβους που βρίσκονται στην περιοχή που εξετάζουμε.

Η πιθανότητα $P_{success}$ ένα μήνυμα που εκπέμπεται από κάποιο κόμβο να φθάσει τελικά στο W, είναι

$$P_{success} = \sum_{h_0}^{\infty} P_1^h P(h = h_0)$$

όπου $P(h = h_0)$ είναι η συνάρτηση πυκνότητας πιθανότητας του h (αθροίζουμε την πιθανότητα να είναι κάποιος κόμβος σε λειτουργία κατά μήκος του μονοπατιού προς το W).

Τέλος, έστω $en = \frac{s}{s+w}$ και $\beta = \frac{s}{w}$. Χρησιμοποιώντας πυκνότητα d τέτοια ώστε $2\alpha \cdot \Delta R \cdot d \approx n(1+\beta)$, παίρνουμε τελικά $P_1 = 1 - e - n$. Οι συγγραφείς αποδεικνύουν στην εργασία τους ότι η πιθανότητα αυτή είναι αυστηρά θετική όταν $n > \ln E(h)$.

2.2.2 Το πρωτόκολλο PFR

Το πρωτόκολλο PFR (Probabilistic Forwarding Protocol) [10], ήταν το επόμενο πρωτόκολλο που εξέδωσε η ομάδα του Sleep-Awake. Ως βασική ιδέα χρησιμοποιεί την προώθηση μηνυμάτων προς το βασικό σταθμό με κάποια πιθανότητα, μέσα σε μια στενή ζώνη από κόμβους του δικτύου. Στόχος είναι η ουσιαστική μείωση των μεταδόσεων, διατηρώντας παράλληλα ανεκτούς χρόνους διάδοσης προς το βασικό σταθμό. Στην ενότητα αυτή το PFR παρουσιάζεται με κάποιες αλλαγές σε σχέση με την αρχική δημοσίευσή του.

Μοντέλο δικτύου που χρησιμοποιείται.

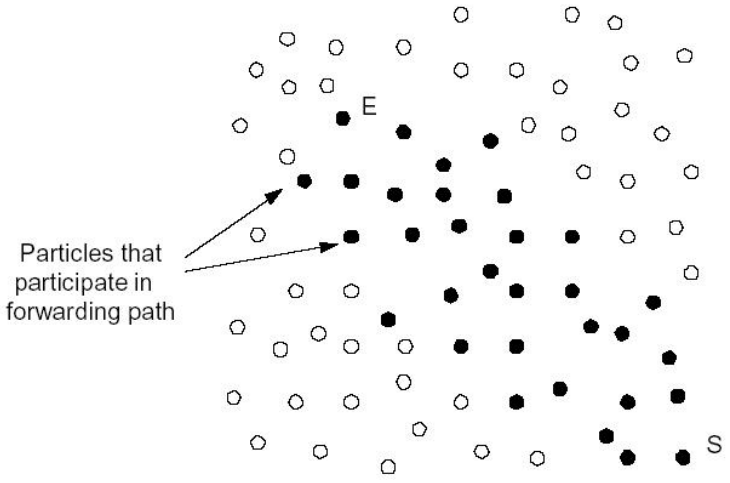
Το μοντέλο δικτύου που χρησιμοποιείται στο PFR είναι παρόμοιο με αυτό του Sleep-Awake. Μπορούμε να κάνουμε μετάδοση υπό γωνία(*beacon mode*) ή σημείου προς σημείο(*point-to-point*). Ακόμα, αντί για τείχος χρησιμοποιούμε ένα κόμβο στην άκρη του δικτύου ως βασικό σταθμό(sink). Επίσης, υποθέτουμε ότι οι κόμβοι γνωρίζουν τη θέση τους στο δίκτυο(π.χ με τη χρήση μιας μονάδας GPS) καθώς και τις συντεταγμένες του sink. Αυτό δεν σημαίνει όμως ότι γνωρίζουν και άλλες πληροφορίες, όπως πλήθος κόμβων στο δίκτυο ή θέση

των υπόλοιπων κόμβων. Μπορούμε να υποθέσουμε ακόμα και ότι δεν γνωρίζουν ακριβώς τη θέση του sink, αλλά μόνο τη γενική κατεύθυνση προς την οποία βρίσκεται. Τέλος, οι κόμβοι κάνουν εναλλάγες μεταξύ ενεργούς κατάστασης (sleep) και λήθαργου (awake) όπως στο Sleep-Awake.

Λειτουργία του πρωτοκόλλου.

Η λειτουργία του PFR γίνεται σε δύο φάσεις, τη φάση δημιουργίας ενός "μετώπου" διάδοσης πληροφορίας και τη φάση προώθησης μηνύματος με πιθανότητα. Οι δύο αυτές φάσεις εξελίσσονται σε κάθε κόμβο ξεχωριστά και δεν χρειάζεται κανενός είδους συχρονισμός των κόμβων.

Δημιουργία μετώπου διάδοσης πληροφορίας: Όπως αναφέραμε, αρχικά δημιουργείται ένα μέτωπο διάδοσης, όταν κάποιος κόμβος έχει πληροφορία να μεταδώσει. Αυτό πρακτικά σημαίνει ότι αρχικά δεν επιλέγουμε μόνο ένα γείτονα για να μεταδώσει την πληροφορία στο υπόλοιπο δίκτυο, αλλά τους κόμβους που περιέχονται σε μια ορισμένη γωνία α στην κατεύθυνση του βασικού σταθμού. Δημιουργούμε αυτό το "μέτωπο" για να υπερπηδήσουμε δυσκολίες, όπως σταθμοί που έπαψαν να λειτουργούν ή εμπόδια που δυσχεραίνουν την επικοινωνία μεταξύ των κόμβων του δικτύου, αφού με τον τρόπο αυτό είναι σαν να δημιουργούμε πολλαπλά μονοπάτια διάδοσης, απλά αυτά βρίσκονται σε μια περιορισμένη γεωγραφική περιοχή. Το σημαντικό εδώ είναι ότι γνωρίζοντας περίπου που θα δημιουργήσουμε τις διόδους για τη διάδοση της πληροφορίας, περιόριζουμε ουσιαστικά τα μηνύματα ελέγχου για τη δημιουργία συνδέσεων, επομένως γλιτώνουμε κατανάλωση ενέργειας.



Σχήμα 2.2: Η λεπτή ζώνη από κόμβους του δικτύου γύρω από τη νοητή γραμμή που συνδέει τον κόμβο E που ανιχνεύει ένα γεγονός και το βασικό σταθμό S

Πιο συγκεκριμένα, όταν ένας κόμβος ανιχνεύει ένα γεγονός και έχει πληρο-

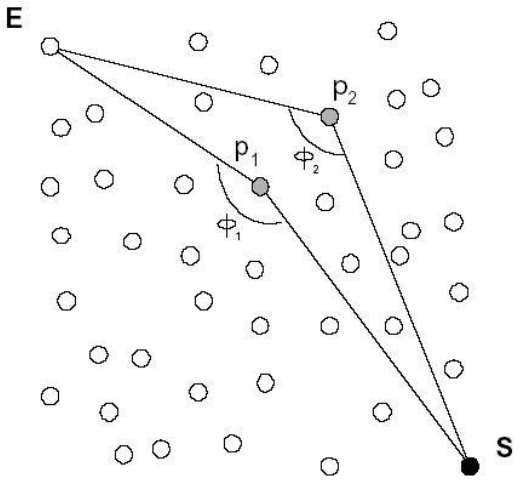
φορία να στείλει στο βασικό σταθμό, κάνει ένα "περιορισμένο" flooding στους γείτονές του, οι οποίοι βρίσκονται μέσα στη γωνία α. Μαζί με την πληροφορία, στο μήνυμα που στέλνεται υπάρχει και ένας μετρητής τον οποίο ονομάζουμε $beta$. Έστω ότι αρχικά έχει τιμή β.

Όταν ένας κόμβος λάβει ένα τέτοιο μήνυμα και το $beta$ είναι μεγαλύτερο του μηδενός, το μειώνει κατά 1 και ντετερμινιστικά το προωθεί στους δικούς του γείτονες που βρίσκονται μέσα στη γωνία α. Όταν ένας κόμβος λάβει μήνυμα με $beta = 0$, αρχίζει να εκτελεί τη δεύτερη φάση του πρωτοκόλλου.

Διάδοση πληροφορίας με πιθανότητα: Ενώ στην αρχική φάση τα μηνύματα προωθούνται προς το βασικό σταθμό ντετερμινιστικά, στη δεύτερη φάση κάνει προώθηση μηνύματος με μία πιθανότητα P_{fwd}. Η πιθανότητα αυτή ορίζεται ως

$$P_{fwd} = \frac{\phi}{\pi}$$

όπου ϕ είναι η γωνία που ορίζεται από την ευθεία που ενώνει τον κόμβο με τον κόμβο που μετέδωσε αρχικά την πληροφορία και την ευθεία που ενώνει τον κόμβο με το βασικό σταθμό, όπως φαίνεται στο επόμενο σχήμα. Προφανώς, αν ένας κόμβος δεν βρίσκεται στη ζώνη ανάμεσα στον αρχικό κόμβο και το βασικό σταθμό, θα έχει μικρότερη πιθανότητα προώθησης, όποτε οι μετάδόσεις περιορίζονται σε μια στένή ζώνη από κόμβους.



Σχήμα 2.3: Η γωνία ϕ για δύο διαφορετικούς κόμβους

Οι δύο διαφορές με την αρχική δημοσίευση του PFR, είναι ότι θεωρούμε ότι υπάρχει μια *cache* σε κάθε κόμβο και ότι οι κόμβοι κανούν μεταβάσεις μεταξύ sleep και awake καταστάσεων. Η *cache* υπάρχει για να αποθηκεύονται τα μηνύματα που λαμβάνει ένας κόμβος για πρώτη φορά, οπότε εάν λάβει το ίδιο μήνυμα σε κάποια στιγμή στο μέλλον δε θα το προωθήσει πολλές φορές. Τέλος,

οι εναλλαγές μεταξύ των δύο καταστάσεων γίνονται για να εξοικονομήσουμε ενέργεια.

Κάποιες λεπτομέρειες για την υλοποίηση του πρωτοκόλλου τώρα. Ένα μήνυμα που σηματοδοτεί κάποιο συμβάν $\mathcal{E}$, αποτελείται από μια τριάδα (*info($\mathcal{E}$), location($\mathcal{E}$),* β), όπου *info($\mathcal{E}$)* είναι η πληροφορία που πρεπεί να διαδοθεί μέσα από το δίκτυο, *location($\mathcal{E}$)* είναι οι συντεταγμένες του κόμβου που ανίχνευσε το συμβάν και το β αναλύθηκε πιο πριν.

Για κάθε συμβάν $\mathcal{E}$ χρησιμοποιούμε σε κάθε κόμβο μια δυαδική μεταβλητή, την οποία καλούμε `PARTICIPATED`, για να καταδείξουμε τη συμμετοχή ή όχι του κόμβου στη διάδοση μηνύματος προς το sink. Αρχικά, την θέτουμε `false` και όταν φθάσει ένα μήνυμα με πληροφορία ή ο κόμβος ο ίδιος ανιχνεύσει ένα συμβάν, τίθεται `true`. Επίσης, χρησιμοποιούμε δύο μεταβλητές, την S και την `INIT`β, για να αποθηκεύσουμε τη θέση του βασικού σταθμού και τη διάρκεια της φάσης δημιουργίας μετώπου διάδοσης αντίστοιχα. Αύτές τις δύο πληροφορίες μπορεί να τις αποκτήσει κάθε κόμβος με ένα broadcast του sink κατά τη διάρκεια της εκκίνησης της λειτουργίας του δικτύου.

Το PFR είναι ένα από τα πρωτοκόλλα που υλοποιήσαμε στον εξομοιωτή που περιγράφεται στο δεύτερο μέρος της εργασίας αυτής, και μάλιστα κάνουμε στη συνέχεια και μία πειραματική αξιολόγησή του και το συγκρίνουμε με το πρωτόκολλο TEEN, το οποίο περιγράφουμε στο επόμενο κεφάλαιο.

2.2.3 Directed Diffusion: ένα ευέλικτο πρότυπο λειτουργίας για δίκτυα έξυπνης σκόνης

Η δουλειά που διεξάγεται στο τμήμα Computer Science στο πανεπιστήμιο UCLA στην Καλιφόρνιαπάνω στα δίκτυα έξυπνης σκόνης, ανάμεσα στα άλλα είχε ως αποτέλεσμα και τη δημιουργία του προτύπου Directed Diffusion ([14], [15], [16]). Ο σχεδιασμός του βασίστηκε στα παρακάτω σημεία-κλειδιά:

- Όλη η επικοινωνία είναι δεδομένο-κεντρική.
- Δεν υπάρχει ουσιαστικά δρομολόγηση των πακέτων (με την έννοια που γνωρίζουμε από τα IP δίκτυα).
- Δεν απαιτούνται ξεχωριστοί ID αριθμοί για τους κόμβους τους δικτύου.
- Τα πακέτα υφίστανται επεξεργασία από τους ενδιάμεσους κόμβους για να συμπτυχθούν (aggregation).

Το πρότυπο παρουσιάζεται χρησιμοποιώντας το παράδειγμα μιας εφαρμογής παρακολούθησης ενός πεδίου μέσω ενός δικτύου έξυπνης σκόνης (ίσως όχι η καλύτερη επιλογή, οι συγγραφείς δέχτηκαν κριτική για την επιλογή τους αυτή). Το μοντέλο δικτύου που χρησιμοποιείται είναι *reactive*, το οποίο σημαίνει ότι δεν υπάρχει ένας προκαθορισμένος χρόνος δειγματοληψίας και αποστολής δειγμάτων και ότι κάθε κόμβος είναι ανεξάρτητος από τους υπόλοιπους. Επίσης, δεν υπάρχει ένας κεντρικός σταθμός με άπειρη ενέργεια. Οποιοσδήποτε

από τους κόμβους του δικτύου μπορεί να λειτουργήσει ως sink και να απαιτήσει πληροφορία από οποιονδήποτε άλλον κόμβο, σε αντίθεση με το μοντέλο που χρησιμοποιείται σε άλλα πρωτόκολλα όπως το LEACH.

Βασικές ιδέες στο Directed Diffusion - Πώς δουλεύει πραγματικά

Καθετί που μπορεί να γίνει αισθητό από τους κόμβους του δικτύου κατηγοριοποιείται, που σημαίνει ότι αν ένας κόμβος ανιχνεύει διαφορετικά είδη από κινούμενα αντικείμενα και θέλουμε να μάθουμε αν ένα φορτηγό (το οποίο *είναι* κινούμενο αντικείμενο) είναι σε μια συγκεκριμένη περιοχή, θα πρέπει να το δηλώσουμε στο query που θα στείλουμε στο δίκτυο. Έτσι, τα queries προς το δίκτυο γίνονται χρησιμοποιώντας ζεύγη γνωρίσματος-τιμής. Η περιοχή που μας ενδιαφέρει προσδιορίζεται σαν ένα ορθογώνιο και επίσης καθορίζουμε τη συχνότητα δειγματοληψίας. Αναφερόμαστε στον κόμβο που στέλνει μια τέτοια αίτηση ως το **sink** και στους κόμβους που ανταποκρίνονται σε αυτή την αίτηση και αρχίζουν τη δειγματοληψία ως τα **sources**. Καλούμε αυτή την αίτηση *data interest*. Παράδειγμα ενός data interest:

```
Type = four-legged animal
Interval = 1 sec
Rectangular = [-100, 200, 200, 400]
Timestamp = 01:20:40
Expires_at = 01:30:40
```

Η επόμενη βασική ιδέα είναι το *gradient*. Ας υποθέσουμε ότι ένας τυχαίος κόμβος παράγει μια αίτηση. Την καταγράφει στη μνήμη του και κατόπιν τη στέλνει στους γείτονες του, αρχικά με πολύ χαμηλή αιτούμενη συχνότητα δειγματοληψίας (γεγονός που εξηγείται αμέσως μετά). Όταν ένας κόμβος δέχεται μια τέτοια αίτηση ελέγχει αν ήδη υπάρχει στην cache του. Αν όχι, δημιουργεί μια νέα καταχώρηση και επίσης καταγράφει τον γείτονα από όπου προήλθε η αίτηση (και όχι τον κόμβο που δημιούργησε την αίτηση) και δημιουργεί ένα gradient και γι' αυτό το λόγο πρέπει να μπορεί να ξεχωρίζει τους γείτονές του. Όταν ένα gradient σε κάποια φάση εκπνέει λόγω timeout ή άλλου λόγου, το interest αφαιρείται από την cache.

Όταν ένας κόμβος λάβει ένα data interest, μπορεί να αποφασίσει να το μεταδώσει σε ένα υποσύνολο των γειτόνων του. Σε αυτούς φαίνεται ότι η αίτηση πήγασε από το γείτονά τους και όχι από το αρχικό sink. Φυσικά, υπάρχουν πολλοί τρόποι με τους οποίους μπορεί να γίνει η επιλογή των γειτόνων που θα αποσταλεί η αίτηση. Ο πιο απλός τρόπος είναι το flooding, δηλαδή να αποσταλεί σε όλους τους γείτονες. Ένας άλλος είναι αν υπάρχουν πληροφορίες GPS να σταλεί προς τους γείτονες που βρίσκονται στην ίδια γεωγραφική κατεύθυνση με το πεδίο που μας ενδιαφέρει.

Με τον τρόπο αυτό, το interest μεταδίδεται μέσω του δικτύου και φθάνει τελικά σε κάποιον κόμβο, ο οποίος βρίσκεται μέσα στο ορισμένο από το interest ορθογώνιο. Ένας τέτοιος κόμβος εκτός από το να αποφασίσει αν θα επαναμεταδώσει το interest στους γείτονές του, αρχίζει να δειγματοληπτεί το πεδίο

γύρω του. Αν ανιχνεύσει ότι υπάρχει κάτι στο πεδίο, ψάχνει την cache του για να δει αν υπάρχει αντίστοιχη αίτηση, αν ανιχνεύσει π.χ έναν ιπποπόταμο , ψάχνει να δει αν έχει αίτηση για τετράποδο ζώο. Αν ναι, αρχίζει και μεταδίδει στους γείτονές του ένα μήνυμα του στυλ:

```
Type = four-legged animal
Instance = elephant
Location = [125,200]
.....
timestamp = 01:20:40  /* local time when event was generated */
```

΄Ενας κόμβος που λαμβάνει τέτοιο μήνυμα ελέγχει την cache του για να δει αν υπάρχει αντίστοιχο interest. Αν δεν υπάρχει, ή αν πρόσφατα προώθησε τέτοιο μήνυμα στους γείτονές του, δεν κάνει τίποτα. Αλλιώς, προωθείται στους γείτονες για τους οποίους υπάρχει ένα αντίστοιχο gradient. Με τον τρόπο αυτό, τα μηνύματα οδεύουν προς το αρχικό sink πάνω από τα gradients που δημιουργήθηκαν καθώς το interest μεταδιδόταν στο δίκτυο.

Μια τελευταία βασική ιδέα είναι η *ενίσχυση* (reinforcement). Όπως αναφέρθηκε, αρχικά το sink είχε στείλει data interest με μικρή απαιτούμενη συχνότητα δειγματοληψίας και έτσι οι κόμβοι που λειτουργούν ως sources δειγματοληπτούν και στέλνουν μηνύματα με μικρή συχνότητα και το sink αρχίζει σε κάποια φάση να λαμβάνει αυτά τα μηνύματα. Επιλέγει τότε να ενισχύσει κάποιον από τους γείτονές του για να λαμβάνει πιο συχνά μηνύματα από αυτόν, ας πούμε επειδή φαίνεται πως από αυτόν τον γείτονα θα λαμβάνει πιο γρήγορα απαντήσεις. Πώς γίνεται όμως αυτό;

Το sink ξαναστέλνει το αρχικό interest αλλά με μεγαλύτερη απαιτούμενη συχνότητα δειγματοληψίας στο συγκεκριμένο γείτονα που θέλει να ενισχύσει. Ο κόμβος αυτός όταν λαμβάνει το νέο μήνυμα ελέγχει αν ήδη έχει gradient προς το sink. Προσέχει ότι τώρα του ζητείται μεγαλύτερη συχνότητα μηνυμάτων και με τη σειρά του αποφασίζει ποιον από τους γείτονές του πρέπει να ενισχύσει. Αυτή η απόφαση γίνεται τοπικά και ενισχύει το γείτονα με τη μικρότερη καθυστέρηση, δηλαδή αυτόν από τον οποίο ήρθε απάντηση γρηγορότερα. ΄Ετσι, επιλέγεται εμπειρικά ένα μονοπάτι μικρής καθυστέρησης.

Επιπλέον, μπορεί να ενισχυθούν παραπάνω από ένα μονοπάτια. Επομένως, χρειαζόμαστε εκτός από ένα μηχανισμό για ενίσχυση και ένα μηχανισμό για να αποδυναμώνουμε μονοπάτια και ουσιαστικά να τα καταργούμε. ΄Ενας τέτοιος μηχανισμός είναι να καταργούμε gradients με time-out αν δεν ενισχύονται. Μια άλλη μέθοδος είναι να στέλνουμε data interests με ακόμα χαμηλότερη συχνότητα δειγματοληψίας. Επίσης, ένα άλλο θέμα είναι το πώς επιλέγουμε τους γείτονες τους οποίους θα *αποδυναμώσουμε*. ΄Ενας τρόπος είναι να αποδυναμώνουμε τους γείτονες από τους οποίους δεν έχουμε δεχθεί μήνυμα μέσα σε κάποιο παράθυρο χρόνου.

Κατά μία έννοια, το Directed Diffusion είναι μια *reactive* τεχνική δρομολόγησης, αφού τα μονοπάτια σχηματίζονται κατά τη διάρκεια της διαδρομής μιας αίτησης στο δίκτυο. Διαφέρει όμως από τις ad-hoc τεχνικές στα παρακάτω:

1. Δεν γίνεται καμιά προσπάθεια να βρεθούν διαδρομές χωρίς βρόχους πριν να αρχίσει η μετάδοση πληροφορίας, αλλά αντιθέτως σχηματίζονται πολλαπλές διαδρομές.

2. Η ενίσχυση (reinforcement) προσπαθεί κατόπιν να περιορίσει τον αριθμό αυτών των μονοπατιών.

3. Χρησιμοποιείται μια cache για τα μηνύματα ώστε να αποφύγουμε επαναμεταδόσεις, βρόχους κτλ.

Εξομοίωση του Directed Diffusion, στόχοι, μετρικές και αποτελέσματα

Οι συγγραφείς του [15] χρησιμοποίησαν τον εξομοιωτή ns-2 για να κάνουν μια εξομοίωση ενός δικτύου έξυπνης σκόνης που τρέχει Directed Diffusion, χρησιμοποιώντας ένα 1.6 Mbps 802.11 ως MAC επίπεδο (αν και αναφέρουν ότι σίγουρα δεν είναι η καλύτερη επιλογή, απλά υπήρχε υλοποιημένο ήδη στον εξομοιωτή). Δημιούργησαν πέντε πεδία από 50 έως 250 κόμβους με σταθερή πυκνότητα. Ο σκοπός της εξομοίωσης αυτής ήταν η σύγκριση με το flooding και το omniscient multicasting, να εξετάσουν τη συμπεριφορά του δικτύου σε περιπτώσεις απώλειας κόμβων και την επίδραση του MAC επιπέδου σε αυτό.

Οι μετρικές που χρησιμοποιήθηκαν ήταν η μέση κατανάλωση ενέργειας, η μέση καθυστέρηση και το ποσοστό μηνυμάτων που στάλθηκαν από τυχαία sources και κατάφεραν τελικά να φτάσουν στο sink. Να σημειωθεί ότι το δίκτυο λειτούργησε κάτω από συνθήκες κανονικού φόρτου.

Τα αποτελέσματα έδειξαν ότι το Directed Diffusion είχε καλύτερη απόδοση από το flooding και το omniscient multicasting στην κατανάλωση ενέργειας. Το omniscient multicasting χρησιμοποιεί περίπου τη μισή ενέργεια από το flooding, ενώ το Directed Diffusion χρησιμοποίησε περίπου το 60% της ενέργειας που κατανάλωσε το omniscient multicasting. Επιπλέον, η μέση καθυστέρηση του Directed Diffusion ήταν άμεσα συγκρίσιμη με τα άλλα δύο πρωτόκολλα, των οποίων η καθυστέρηση είναι σχεδόν βέλτιστη.

Για τη μελέτη των επιδράσεων από απώλειες κόμβων στο δίκτυο, προκλήθηκαν θάνατοι κόμβων πάνω στα μονοπάτια που ήταν πιο πιθανό να χρησιμοποιήσει το πρωτόκολλο και τυχαία στο υπόλοιπο δίκτυο. Το πρωτόκολλο κατάφερε να διατηρήσει σε ικανοποιητικά επίπεδα την απόδοση του δικτύου και να έχει ανεκτό ποσοστό παράδοσης μηνυμάτων στο sink.

Συμπεράσματα και σύγκριση του Directed Diffusion με άλλα πρωτόκολλα

Συνοψίζοντας, τα βασικά χαρακτηριστικά του Directed Diffusion είναι:

- Το πρωτόκολλο έχει δυνατότητες για σημαντική εξοικονόμηση ενέργειας.
- Παρουσιάζει σταθερή συμπεριφορά στις αλλαγές τοπολογίας που μπορούν να συμβούν στο δίκτυο ανά πάσα στιγμή.
- Το MAC επίπεδο πρέπει να επιλεγεί και να σχεδιασθεί με μεγάλη προσοχή.

Γενικά, το Directed Diffusion ξεχωρίζει ανάμεσα σε όλα τα πρωτόκολλα για δίκτυα έξυπνης σκόνης, γιατί προβάλλει ως μια γενική λύση για το πεδίο αυτό, σε αντίθεση με τα περισσότερα πρωτόκολλα που απευθύνονται σε συγκεκριμένες εφαρμογές. Παρόλα αυτά, δεν έχει γίνει μέχρι στιγμής σύγκριση με άλλα πρωτόκολλα για δίκτυα έξυπνης σκόνης εκτός από τα πολύ βασικά, και επιπλέον υπάρχουν κάποια ερωτήματα στα οποία μέχρι στιγμής οι εμπνευστές του Directed Diffusion δεν έχουν απαντήσει:

- Θα έχουν τη δυνατότητα όλοι οι κόμβοι να αποθηκεύουν στη μνήμη τους όλες τις αιτήσεις που βλέπουν (μπορεί να μην φθάνει η μνήμη);
- Αν έχουμε μια εφαρμογή με μεγάλα πακέτα, όπως μετάδοση εικόνας, δεν μπορούμε να αποθηκεύουμε αυτήν την πληροφορία σε κάθε κόμβο του δικτύου.
- Πόσο αξιόπιστα είναι τα αποτελέσματα του ns-2 (δεδομένου ότι τελευταία υπήρξαν επικρίσεις για τον εξομοιωτή αυτόν);

2.2.4 Το πρωτόκολλο STEM

Το STEM (Sparse Topology and Energy Manipulation, διαχείριση αραιής τοπολογίας και ενέργειας),[11], είναι μια τεχνική διαχείρισης της τοπολογίας ενός δικτύου έξυπνης σκόνης, η οποία χρησιμοποιεί ως παραμέτρους την μέση καθυστέρηση του δικτύου και την κατανάλωση ενέργειας. Επίσης, οι συγγραφείς του STEM ερευνούν την επίδραση της πυκνότητας του δικτύου στη διαχείριση τοπολογίας και συνδυάζουν το STEM με το πρωτόκολλο GAF (Geographical Adaptive Fidelity), το οποίο εκμεταλλεύεται την πυκνότητα ενός δικτύου, προκειμένου να επιτευχθούν ακόμη καλύτερα αποτελέσματα.

Διαχείριση τοπολογίας και βασικές έννοιες του STEM

Ο προφανής τρόπος να διασωθεί η ενέργεια ενός κόμβου σε ένα δίκτυο αισθητήρων είναι να κλείσει ο πομποδέκτης του κόμβου, αποσυνδέοντας τον κατά συνέπεια από το δίκτυο, και αλλάζοντας έτσι την τοπολογία του δικτύου (καταργούνται συνδέσεις, διαχείριση τοπολογίας). Ο στόχος της διαχείρισης τοπολογίας είναι να συντονιστεί ο τρόπος που οι κόμβοι κλείνουν τους πομποδέκτες τους και συγχρόνως να εξασφαλίζεται ότι η πληροφορία από τους κόμβους μπορεί να φθάσει σε έναν κεντρικό σταθμό.

Το STEM εκμεταλλεύεται τη χρονική διάσταση περισσότερο παρά τη διάσταση της πυκνότητας, η οποία είναι αυτό που εκμεταλλεύονται άλλα πρωτόκολλα (όπως το SPIN και το GAF). Στην πράξη τώρα, θεωρούμε ότι το δίκτυο βρίσκεται στο μεγαλύτερο μέρος του χρόνου σε μια κατάσταση που απλά παρακολουθεί τι γίνεται στο πεδίο και δε χρειάζεται να μεταδώσει καμιά πληροφορία. Έτσι, κλείνουμε τους πομποδέκτες των κόμβων. Υποθέστε ότι ένα γεγονός συμβαίνει και κάποιος κόμβος εκεί κοντά το ανιχνεύει. Ανοίγει τον πομπό του αλλά το πρόβλημα είναι ότι οι γύρω σταθμοί, από τους οποίους κάποιος πρέπει να προωθήσει το μήνυμα προς τον κεντρικό σταθμό, πιθανότατα

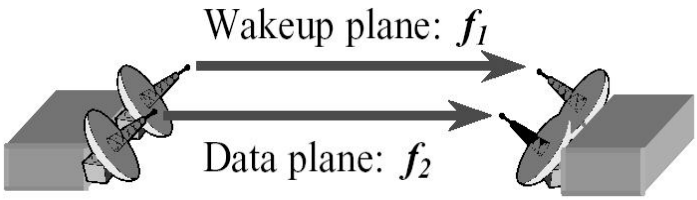
δεν έχουν ανιχνεύσει το γεγονός και επομένως έχουν κλειστούς τους δέκτες τους!

Η λύση που προτείνουν οι συγγραφείς είναι βασισμένη στη χρήση δύο διαφορετικών πομποδεκτών, ένα για να ξυπνούν κόμβους που κοιμούνται και ένα για την πραγματική μεταφορά στοιχείων. Η γενική ιδέα είναι η εξής: ο κόμβος που θέλει να επικοινωνήσει, ο "πομπός", στέλνει ένα σήμα αναγνωριστικών σημάτων με την ταυτότητα του κόμβου που προσπαθεί να ξυπνήσει (ο "στόχος") μέσω της wakeup ραδιοσυχνότητας. Οι κόμβοι που είναι κοιμισμένοι, ανοίγουν περιοδικά το wakeup δέκτη τους και εάν πιάσουν ένα σήμα με την ταυτότητά τους, αποκρίνονται και ανοίγουν έπειτα το δέκτη τους για μεταφορά πληροφορίας.

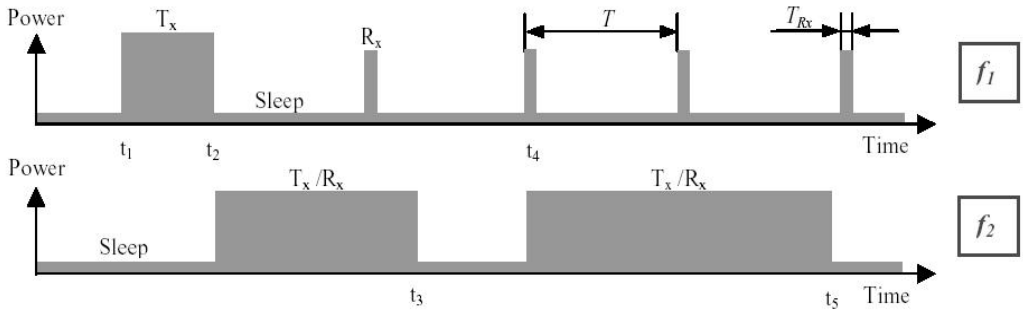
Για να ακούσει τουλάχιστον ένα wakeup σήμα, ο στόχος χρειάζεται να ανοίξει το wakeup δέκτη του για έναν αρκετά μακρύ διάστημα, το οποίο πρέπει να είναι τουλάχιστον όσο ο χρόνος μετάδοσης ενός πακέτου wakeup σημάτων συν το διάστημα μεταξύ δύο τέτοιων σημάτων. Ένα σενάριο μιας τέτοιας χαρακτηριστικής κατάστασης παρουσιάζεται στο σχήμα 2.5.

Οι μεταβάσεις στους δύο πομποδέκτες ενός κόμβου απεικονίζονται στο παραπάνω σχήμα. Στην αρχή, ο πρώτος κόμβος έχει κλειστά τα συστήματα επικοινωνίας του. Κατόπιν, ανιχνεύει ένα γεγονός και ανοίγει το wakeup πομπό του και στέλνει πακέτα wakeup. Μετά από κάποια στιγμή ένας άλλος κόμβος αποκρίνεται και έτσι ο κόμβος ανοίγει το data δέκτη του για να αρχίσει τη μεταφορά δεδομένων προς αυτόν τον κόμβο, αλλά ταυτόχρονα συνεχίζει να ανοίγει το wakeup δέκτη του περιοδικά, σε περίπτωση που κάποιος άλλος κόμβος επιθυμεί να επικοινωνήσει μαζί του.

Εάν επρόκειτο να χρησιμοποιήσουμε μια ραδιοσυχνότητα μόνο, θα υπήρχε σύγκρουση μεταξύ wakeup σημάτων και πληροφορίας. Η χρησιμοποίηση ενός πομποδέκτη ο οποίος θα αλλάζει μεταξύ δύο συχνοτήτων θα μπορούσε να λύσει αυτό το πρόβλημα, αλλά οι μεταδόσεις πληροφορίας θα έπρεπε να διακόπτονται περιοδικά. Φυσικά, ακόμα και στην περίπτωση δύο πομποδεκτών με διαφορετικές συχνότητες, οι συγκρούσεις με σήματα από άλλους κόμβους θα μπορούσαν να συμβούν σε κάθε περιοχή συχνότητας. Μερικές δευτερεύουσες αλλαγές πρέπει να γίνουν για να λύσουν αυτό το πρόβλημα.



Σχήμα 2.4: Οι δύο διαφορετικές συχνότητες που χρησιμοποιεί το STEM



Σχήμα 2.5: Η δραστηριότητα στις δύο συχνότητες του ίδιου κόμβου

΄Ενας κόμβος ανοίγει το πομποδέκτη δεδομένων του αν υπάρχει σύγκρουση στη wakeup συχνότητα (δηλαδή δύο κόμβοι προσπαθούν ταυτόχρονα να ξυπνήσουν τον ίδιο κόμβο), ακόμα και αν δεν είναι αυτός ο κόμβος που πρέπει να ξυπνήσει. Αν ένας κόμβος εντοπίσει μια τέτοια σύγκρουση, δεν στέλνει πίσω μια wakeup επιβεβαίωση αλλά παρόλα αυτά έχει ξυπνήσει, οπότε ο σκοπός μας επιτεύχθηκε. Ο πομπός περιμένει για μια wakeup επιβεβαίωση όμως. Αν αυτή έρθει μέσα σε ένα προκαθορισμένο χρονικό διάστημα T, αμέσως αρχίζει η μετάδοση πληροφορίας, αλλιώς περιμένει απλώς να περάσει το T. Οι κόμβοι που έχουν ξυπνήσει αλλά δεν είναι στόχοι, αλλάζουν κατάσταση και κοιμούνται μετά από κάποιο χρονικό διάστημα. Οι συγκρούσεις στη συχνότητα για μεταφορά δεδομένων αντιμετωπίζονται μέσω του `MAC` επιπέδου.

Θεωρητική ανάλυση του STEM

Οι δημιουργοί του STEM δίνουν και μια θεωρητική του ανάλυση. Δίνουν κάποια αποτελέσματα για την καθυστέρηση εγκατάστασης επικοινωνίας μεταξύ δύο κόμβων (setup latency) και για την εξοικονόμηση ενέργειας.

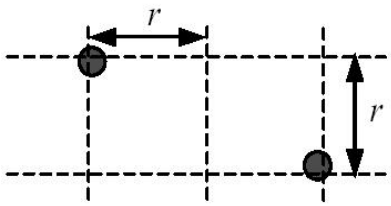
Η καθυστέρηση εγκατάστασης επικοινωνίας μεταξύ δύο κόμβων ορίζεται ως το διάστημα που μεσολαβεί μεταξύ της στιγμής που ο κόμβος-πομπός αρχίζει να στέλνει wakeup σήματα μέχρι τη στιγμή κατά την οποία και οι δύο κόμβοι έχουν ανοίξει τους πομποδέκτες δεδομένων τους. Μέσω μιας σχετικά επίπονης ανάλυσης, την οποία δεν έχει νόημα να παραθέσω εδώ, καταλήγουν ότι η setup latency T_S είναι περίπου

$$T_S = \frac{T + T_B}{2} + 2B_1 + B_2 - T_{R_x}$$

όπου T_{R_x} είναι ο ελάχιστος χρόνος κατά τον οποίο ο wakeup πομποδέκτης είναι αναμμένος, είναι ο χρόνος μεταξύ μεταδόσεων wakeup σημάτων και B_2 είναι ο χρόνος μετάδοσης μιας wakeup επιβεβαίωσης.

Η συνολική ενέργεια που ξοδεύεται ορίζεται ως

$$E_{node} = E_{wakeup} + E_{data}$$



Σχήμα 2.6: Οι εικονικοί κόμβοι στο GAF

όπου E_{wakeup} είναι η ενέργεια που ξοδεύεται όσο ο κόμβος σε κανονική λειτουργία και E_{data} είναι η ενέργεια όσο μεταδίδει δεδομένα. Καταλήγουν στο συμπέρασμα ότι η ενέργεια που ξοδεύεται προς την ενέργεια που θα ξοδευόταν αν ο κόμβος δεν πήγαινε πότε σε μετάδοση δεδομένων είναι:

$$\frac{E}{E_0} = \frac{1}{\beta} + f_S(t_{burst} + \frac{T}{2}) + 2\frac{P_{sleep}}{P}$$

όπου β είναι το αντίστροφο του χρόνου που ο κόμβος είναι απασχολημένος όταν λειτουργεί κανονικά, t_{burst} είναι η μέση διάρκεια μιας μετάδοσης δεδομένων και f_S είναι το αντίστροφο του χρόνου για εγκατάσταση σύνδεσης μεταξύ δύο κόμβων.

Συνδυάζοντας το STEM με το GAF - Συμπεράσματα

Αρχικά περιγράφεται σε συντομία η λειτουργία του πρωτοκόλλου GAF. Το GAF χωρίζει το πεδίο σε ένα πλέγμα από τετράγωνα διαστάσεων r επί r, όπως φαίνεται στην παρακάτω εικόνα. Οι διαστάσεις επιλέγονται ώστε οι κόμβοι που περιέχονται στο ίδιο τετράγωνο να είναι ισοδύναμοι από πλευράς δρομολόγησης. Με το GAF μόνο ένας κόμβος σε κάθε περιοχή είναι σε λειτουργία, ενώ οι υπόλοιποι κλείνουν τους πομποδέκτες τους. Μπορούμε να φανταστούμε κάθε τετράγωνο ως ένα εικονικό κόμβο. Αυτός ο κόμβος είναι υπεύθυνος για να προωθεί μηνύματα από και προς τους υπόλοιπους εικονικούς κόμβους στο δίκτυο. Στην ιδανική περίπτωση, αφού κάθε κόμβος λειτουργεί για ένα ποσοστό του χρόνου, έστω $\frac{1}{x}$, θα έπρεπε το δίκτυο να έχει x φορές μεγαλύτερη διάρκεια ζωής.

Το STEM και το GAF λειτουργούν κάπως συμπληρωματικά το ένα στο άλλο. Ο εικονικός κόμβος του GAF μπορεί να χρησιμοποιήσει το STEM για να επικοινωνήσει με τους υπόλοιπους εικονικούς κόμβους. Έτσι, το πρωτόκολλο μπορεί να αλλαχθεί και στη δρομολόγηση να χρησιμοποιούνται εικονικοί αντί για πραγματικοί κόμβοι. Πρέπει επίσης να γίνουν μερικές αλλαγές στον τρόπο που επιλέγεται αρχηγός μέσα σε κάθε περιοχή. Ένας κόμβος που θέλει να γίνει αρχηγός επικοινωνεί με τον ήδη υπάρχοντα αρχηγό και συμφωνούν σε μια διαδικασία αλλαγής αρχηγού. Αν δε μπορεί να γίνει αυτό, υποθέτει ότι ο αρχηγός έχει πεθάνει και γίνεται αυτόματα αρχηγός

Καταλήγοντας, το STEM δεν είναι καθαρά ένα πρωτόκολλο δρομολόγησης για δίκτυα έξυπνης σκόνης, και για το λόγο αυτό δεν μπορεί να συγκριθεί άμεσα με πρωτόκολλα όπως το LEACH ή το Directed Diffusion. Παρόλα αυτά, η ιδέα για δύο ξεχωριστές συχνότητες για επικοινωνία είναι ένα χαρακτηριστικό που μπορεί να χρησιμοποιηθεί και από άλλα πρωτόκολλα για καλύτερα αποτελέσματα.

Κεφάλαιο 3

Τα πρωτόκολλα TEEN και LEACH

Στο κεφάλαιο αυτό θα ασχοληθούμε με άλλα δύο πρωτόκολλα για δίκτυα έξυπνης σκόνης, το LEACH και το TEEN. Αποφάσισα να τους αφιερώσω ένα ξεχωριστό κεφάλαιο για δύο λόγους.

1. Σχετίζονται άμεσα μεταξύ τους, καθώς το TEEN είναι ουσιαστικά μια επέκταση του LEACH.

2. Με το TEEN θα ασχολήθουμε περισσότερο συστηματικά και στα επόμενα καφάλαια, καθώς είναι ένα από τα πρωτόκολλα που χρησιμοποιήσαμε στον εξομοιωτή simDust.

3.1 Το πρωτόκολλο LEACH

Το πρωτόκολλο LEACH (Low-energy Adaptive Clustering Hierarchy) [8] είναι ένα από τα πρώτα πρωτόκολλα για δίκτυα έξυπνης σκόνης που δημοσιεύθηκαν και είναι κατά κάποιο τρόπο σημείο αναφοράς. Ακολουθούν μια παρουσίαση των βασικών χαρακτηριστικών του, όπως αυτά παρουσιάζονται στο [8], αν και γενικά υπάρχουν ερωτηματικά για το πως θα μεταφερθεί το πρωτόκολλο στην πράξη.

3.1.1 Μοντέλο δικτύου που χρησιμοποιείται στην ανάλυση και στις πειραματικές μετρήσεις

Θα ασχοληθούμε πρώτα με το μοντέλο δικτύου που χρησιμοποίησαν οι συγγραφείς. Σε αυτό, ο βασικός σταθμός (base station) είναι σταθερός και βρίσκεται μακριά από το πεδίο που βρίσκονται οι αισθητήρες, οι οποίοι είναι όλοι ίδιοι μεταξύ τους και έχουν αρχικά τις ίδιες περιορισμένες ποσότητες ενέργειας.

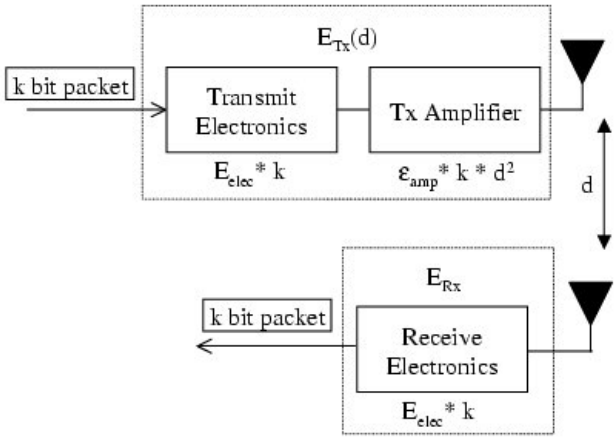
Ακόμα, γίνεται η υπόθεση ότι η ενέργεια που απαιτείται για μετάδοση και λήψη μηνύματος k-bit σε μια απόσταση d, είναι αντίστοιχα :

$$E_{T_x(k,d)} = E_{elec} \cdot k + e_{amp} \cdot k \cdot d^2$$

και

$$E_{R,r(k)} = E_{elec} \cdot k$$

όπου E_{elec} είναι η ενέργεια που χρειάζεται ο πομποδέκτης για να λειτουργήσει και e_{amp} η ενέργεια που απαιτεί ο ενισχυτής του πομπού για να έχουμε ανεκτό λόγο σήματος-θορύβου. Επίσης, η ενέργεια που απαιτείται για μετάδοση είναι ανάλογη του τετραγώνου της απόστασης d. Από τις σχέσεις αυτές, γίνεται φανερό ότι για μικρές αποστάσεις η ενέργεια που απαιτείται για λήψη ενός μηνύματος είναι άμεσα συγκρίσιμη με την ενέργεια που απαιτείται για τη μετάδοσή του! Επιπλέον, γίνεται η υπόθεση ότι το κανάλι είναι συμμετρικό, δηλαδή η ενέργεια που απαιτείται για μετάδοση μηνύματος από το σημείο A στο B, είναι η ίδια για μετάδοση από το B στο A. Το μοντέλο για την κατανάλωση ενέργειας συνολικά φαίνεται στο σχήμα 3.1.



Σχήμα 3.1: Μοντέλο κατανάλωσης ενέργειας

Τέλος, το δίκτυο των μικροαισθητήρων δειγματοληπτεί με σταθερό ρυθμό (π.χ 1 φορά το δευτερόλεπτο) και δεν στέλνουμε από τον κεντρικό σταθμό εντολές για δειγματοληψία, επομένως οι κόμβοι του δικτύου έχουν πάντα πληροφορία για να στείλουν. Το πρωτόκολλο αυτό, επομένως, κατατάσσεται στα συνεχούς ανάκτησης (continuous, proactive).

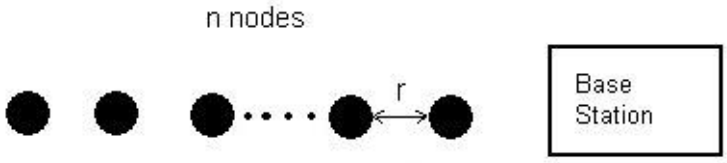
3.1.2 Παραδοσιακά πρωτόκολλα επικοινωνίας. Ανάλυση των μειονεκτημάτων τους στην περίπτωση των δικτύων έξυπνης σκόνης

Οι δημιουργοί του LEACH εξετάζουν τρεις περιπτώσεις. Η πρώτη είναι η απ' ευθείας επικοινωνία κάθε κόμβου του δικτύου με το βασικό σταθμό. Προφανώς η απόσταση από το βασικό σταθμό είναι ζωτικής σημασίας. Όσο μεγαλύτερη είναι αυτή η απόσταση, τόσο περισσότερη είναι και η ενέργεια που απαιτείται

για την επικοινωνία με το βασικό σταθμό. Από την άλλη πλευρά όμως, δεν υπάρχουν καθόλου λήψεις μηνυμάτων στους κόμβους του δικτύου.

Η δεύτερη περίπτωση είναι η δρομολόγηση των μηνυμάτων που στέλνει κάθε κόμβος του δικτύου προς το βασικό σταθμό να γίνεται μέσω των ίδιων των κόμβων του δικτύου. Γίνεται κάποια επιλογή συντομότερου μονοπατιού με κάποιο κριτήριο κατανάλωσης ενέργειας. Εδώ θεωρούμε την περίπτωση το βάρος σε κάθε ακμή του δικτύου να είναι η απαιτούμενη ενέργεια για μετάδοση (δηλαδή δεν λαμβάνεται υπόψη το κόστος της λήψης) από τον ένα κόμβο στον άλλο.

Οι συγγραφείς αποδεικνύουν το φαινομενικά παράδοξο ότι μπορεί τελικά να απαιτείται σε κάποιες περιπτώσεις δικτύων περισσότερη ενέργεια στην δεύτερη περίπτωση από ότι στην πρώτη! Έστω ότι έχουμε το απλό δίκτυο του σχήματος 3.2.



Σχήμα 3.2: Μια "ύποπτη" περίπτωση δικτύου

Στην πρώτη περίπτωση η ενέργεια που απαιτείται είναι

$$E_{direct} = E_{T_x(k,d=nr)} = E_{elec} \cdot k + e_{amp} \cdot k \cdot (nr)^2 = k(E_{elec} + e_{amp} n^2 r^2)$$

Στην δεύτερη περίπτωση (Minimum Transmission Energy , MTE routing) θα είναι :

$$E_{MTE} = nE_{T_x(k,d=r)} + (n-1) \cdot E_{R_x(k)} = \ldots = k((2n-1)E_{elec} + e_{amp} n r^2)$$

Από τις δύο σχέσεις προκύπτει ότι $E_{direct} < E_{MTE}$! Χρησιμοποιώντας τις προηγούμενες σχέσεις ως προσεγγίσεις της απαιτούμενης ενέργειας σε ένα τυχαίο δίκτυο 100 κόμβων, οι συγγραφείς έκαναν κάποια πειράματα στο MATLAB, τα οποία επιβεβαίωσαν τα προηγούμενα και δείχνουν ότι η αποτελεσματικότητα του πρωτοκόλλου επικοινωνίας εξαρτάται σε πολύ μεγάλο βαθμό από την τοπολογία του δικτύου.

Επίσης, στα πειράματά τους παρατήρησαν τα εξής:

- Στην περίπτωση της απ' ευθείας μετάδοσης οι κόμβοι που "πεθαίνουν" πιο γρήγορα είναι οι πιο απομακρυσμένοι από τον βασικό σταθμό. Τελευταίοι πεθαίνουν αυτοί που είναι πιο κοντά στον βασικό σταθμό.

- Στην περίπτωση του MTE routing, οι κόμβοι που πεθαίνουν πιο γρήγορα είναι αυτοί που βρίσκονται πιο κοντά στον βασικό σταθμό! Αυτό είναι

αναμενόμενο καθώς αυτοί οι κόμβοι κάνουν συνεχώς μεταδόσεις και λήψεις μηνυμάτων που στέλνουν οι άλλοι κόμβοι και τα προωθούν στον βασικό σταθμό, συνεπώς είναι συνεχώς απασχολημένοι. Αφού εξαντλήσουν τα αποθέματα ενέργειάς τους, τη θέση τους θα πάρουν οι κόμβοι που βρίσκονται αμέσως πιο μακριά από τον βασικό σταθμό, μόνο που αυτοί θα πεθάνουν ακόμα πιο γρήγορα, αφού βρίσκονται πιο μακριά και άρα θέλουν περισσότερη ενέργεια για να στείλουν μήνυμα!

Η τρίτη περίπτωση είναι να χωρίσουμε σε clusters το δίκτυο, σε καθένα από τους οποίους θα υπάρχει ένας σταθερός cluster-head κόμβος, ο οποίος θα στέλνει τα μηνύματα που λαμβάνει από τους υπόλοιπους κόμβους του δικτύου στον βασικό σταθμό. Προφανώς, οι πρώτοι σταθμοί που πεθαίνουν είναι οι cluster-head, και το δίκτυο δεν λειτουργεί σωστά έπειτα από σύντομο χρονικό διάστημα.

3.1.3 Χαρακτηριστικά του LEACH

Το LEACH βασίζεται στην ιδέα των clusters, την οποία προχωρά ένα βήμα παραπέρα. Έτσι, οι κόμβοι του δικτύου αυτο-οργανώνονται σε clusters, όπου υπάρχει ένας cluster-head μέσω του οποίου οι υπόλοιποι επικοινωνούν με το βασικό σταθμό. Η διαφορά στο LEACH είναι ότι οι cluster-head (CH) κόμβοι δεν είναι σταθεροί, δηλαδή οι κόμβοι αποφασίζουν σε κάθε χρονική στιγμή με κάποια τυχαιότητα αν θα είναι CH ή όχι. Πιο συγκεκριμένα, η λειτουργία του LEACH διαιρείται σε γύρους και κάθε γύρος διαιρείται σε δύο φάσεις:

1. μία φάση αρχικοποίησης, και
2. μία σταθερή φάση.

Φάση αρχικοποίησης

Αρχικά, κάθε κόμβος αποφασίζει αν θα είναι CH ή όχι. Θεωρούμε ότι υπάρχει ένας βέλτιστος αριθμός CH στο δίκτυο, τον οποίο γνωρίζουμε (έστω ότι έχει προκύψει μετά από θεωρητική ή πειραματική ανάλυση). Η απόφαση βασίζεται στον αριθμό αυτό και στο πόσες φορές μέχρι τώρα έχει γίνει κάποιος κόμβος CH μέχρι τη δεδομένη χρονική στιγμή. Σε κάθε κόμβο επιλέγεται τυχαία ένας αριθμός μεταξύ του 0 και του 1. Αν υπερβαίνει κάποιο κατώφλι T(n), τότε ο κόμβος γίνεται CH για τον τρέχοντα γύρο. Πιο συγκεκριμένα:

$$T(n) = \frac{P}{1 - P(r \bmod \frac{1}{P})}$$

αν ο κόμβος n ανήκει στους κόμβους που δεν έχουν γίνει CH τους τελευταίους $\frac{1}{P}$ γύρους, και $T(n) = 0$ αλλιώς. Έτσι, κάθε κόμβος γίνεται CH μέσα σε $\frac{1}{P}$ γύρους.

Αν κάποιος κόμβος διαλέξει να γίνει CH, διαφημίζει αυτή του την ιδιότητα στους γείτονες του. Οι μη-CH κόμβοι ακούν τα μηνύματα διαφήμισης και αποφασίζουν σε ποιον cluster ανήκουν από την ισχύ των μηνυμάτων που λαμβάνουν.

Κοινοποιούν έπειτα στον CH του cluster τους την απόφασή τους αυτή και έτσι ο cluster head γνωρίζει τους κόμβους που βρίσκονται στην περιοχή δικαιοδοσίας του.

Αφού όλοι οι κόμβοι αποφασίσουν ότι ανήκουν σε κάποιο cluster και στον αντίστοιχο CH, οργανώνεται κάποιο `TDMA` πρόγραμμα μετάδοσης από τον CH, το οποίο και μεταδίδεται στους υπόλοιπους κόμβους (δηλαδή ότι πρώτος μεταδίδει ο κόμβος x, μετά ο κόμβος y, κ.ο.κ). Έπειτα από αυτό, μπορεί να αρχίσει η σταθερή φάση.

Σταθερή φάση

Οι κόμβοι αρχίζουν να στέλνουν μηνύματα στον CH σύμφωνα με το πρόγραμμα που έχει καταρτιστεί για κάθε cluster. Ο CH έχει ανοιχτό τον δέκτη του καθ' όλη τη διάρκεια του γύρου, μέχρι να περάσει ένα προκαθορισμένο χρονικό διάστημα. Μετά από κάποια επεξεργασία (aggregation) μεταδίδει ένα σύνθετο μήνυμα στον βασικό σταθμό. Αυτή η διαδικασία επαναλαμβάνεται μέχρι να τελειώσει ο γύρος. Αυτό συμβαίνει μετά από κάποιο προκαθορισμένο χρονικό διάστημα το οποίο είναι γνωστό στους κόμβους. Έπειτα από αυτό το διάστημα, επαναλαμβάνεται η φάση αρχικοποίησης και σχηματίζονται εντελώς νέοι clusters.

Πολλαπλοί clusters και συγκρούσεις

Αν και στο [8] δεν είναι σαφή τα πράγματα στο MAC πεδίο, στην υλοποιήση που έκαναν οι συγγραφείς του LEACH στον ns-2, χρησιμοποιείται ένα νέο πρωτόκολλο MAC, το MacSensor, το οποίο είναι ένας συνδυασμός των CSMA (Carrier-Sense Multiple Access), TDMA (Time-Division Multiple Access), και απλής μορφής DS-SS (Direct Sequence Spread Spectrum). Το CSMA χρησιμοποιείται στη φάση αρχικοποίησης, ενώ τα TDMA και DS-SS χρησιμοποιούνται στη σταθερή φάση. Όταν ένας κόμβος αυτοεκλεγεί cluster-head διαλέγει τυχαία από μια σειρά κωδικών για να χρησιμοποιήσει στο DS-SS, και έτσι με κάποια πιθανότητα γειτονικοί clusters χρησιμοποιούν διαφορετικούς κωδικούς και περιορίζονται συνολικά οι συγκρούσεις. Παράλληλα, ο cluster-head πληροφορεί τους υπόλοιπους κόμβους για τους οποίους είναι υπεύθυνος για αυτόν τον κωδικό.

3.1.4 Αποτελέσματα - Συμπεράσματα

Οι δημιουργοί του LEACH έκαναν κάποια εξομοίωση, όπως αναφέρθηκε, στο MATLAB σε ένα τυχαίο δίκτυο 100 κόμβων για να συγκρίνουν την απόδοση των direct transmission, MTE routing και LEACH. Έθεσαν ως παράμετρο ότι το 5% των κόμβων μπορούν να είναι cluster-heads, και παραθέτουν τα ακόλουθα συμπεράσματα:

- Στο LEACH δεν παρατηρείται το φαινόμενο να πεθαίνουν πρώτα οι κόμβοι που βρίσκονται πιο κοντά ή πιο μακριά από τον βασικό σταθμό όπως

στα άλλα δύο πρωτόκολλα, αλλά υπάρχει μια ομοιόμορφη κατανομή σε ολόκληρη την επιφάνεια του δικτύου.

- Το LEACH μειώνει έως και 8 φορές την ενέργεια που καταναλώνεται στην επικοινωνία σε σχέση με τα άλλα δυο πρωτόκολλα
- Ο πρώτος κόμβος πεθαίνει στο LEACH έως και 8 φορές πιο αργά και ο τελευταίος κόμβος πεθαίνει έως και 3 φορές πιο αργά.

3.2 Το πρωτόκολλο TEEN

Το TEEN (Threshold sensitive Energy Efficient sensor Network protocol) [9], είναι ουσιαστικά μια επέκταση του LEACH για οδηγούμενα από γεγονότα (event-driven, reactive) δίκτυα. Επίσης, προχωρά παραπέρα την ιδέα του clustering με τη χρήση ιεραρχικού clustering. Πάνω από όλα όμως, το TEEN προορίζεται για χρήση σε δίκτυα έξυπνης σκόνης με διαφορετική φιλοσοφία από αυτά στα οποία απευθύνεται το LEACH.

Πιο συγκεκριμένα, το LEACH είναι κατάλληλο για δίκτυα όπου το ζητούμενο είναι να παίρνουμε συνεχείς μετρήσεις από τους κόμβους του δικτύου, δηλαδή δίκτυα συνεχούς (continuous) ανάκτησης πληροφορίας. Αντίθετα, στο TEEN θεωρούμε ότι πρέπει να πληρούνται κάποιες προϋποθέσεις προκειμένου να στείλει κάποιος κόμβος μήνυμα πληροφορίας προς το βασικό σταθμό. Ας δούμε τώρα πως δουλεύει το πρωτόκολλο.

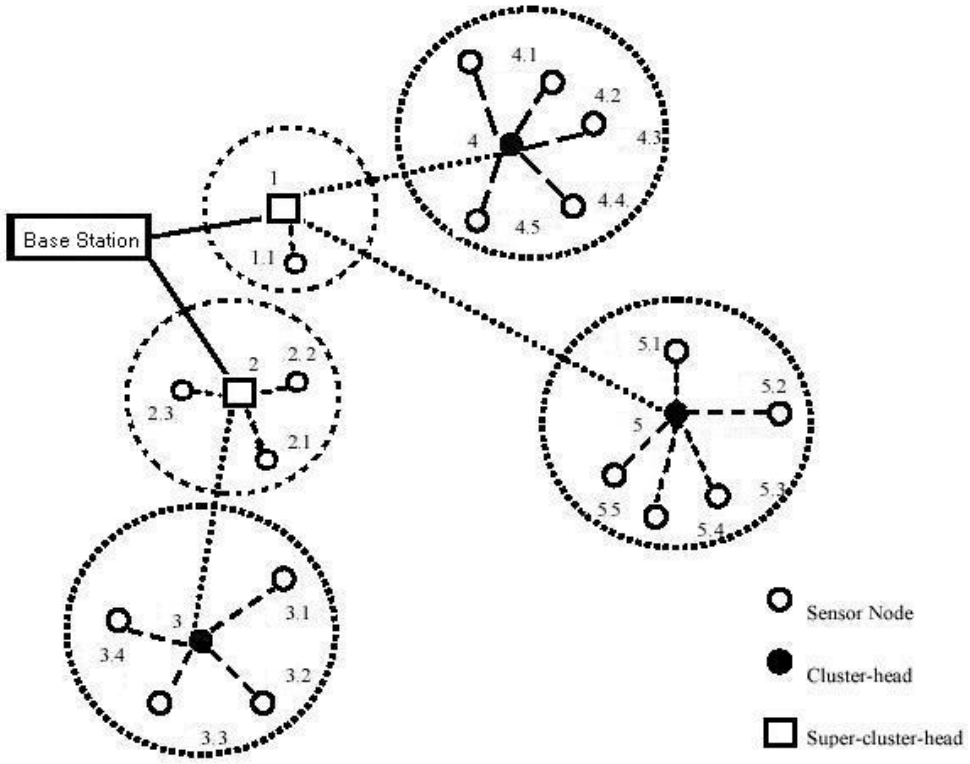
3.2.1 Μοντέλο δικτύου που χρησιμοποιείται στο TEEN

Ουσιαστικά, είναι το ίδιο με το LEACH (που είδαμε στην προηγούμενη ενότητα), εκτός από το ότι προχωρά την πρόταση που έκαναν οι συγγραφείς του LEACH στο τέλος της δημοσίευσης τους, για χρήση ιεραρχικού clustering. Πιο συγκεκριμένα, στο LEACH υπήρχαν δύο επίπεδα ιεραρχίας, απλοί κόμβοι και cluster-heads. Αφού οι απλοί κόμβοι αποφασίσουν σε ποιον cluster ανήκουν, όταν θέλουν να στείλουν μήνυμα στο βασικό σταθμό το στέλνουν στον cluster-head τους, ο οποίος με τη σειρά του το προωθεί στο βασικό σταθμό.

Στο TEEN αντίθετα, μπορούμε να έχουμε επιπλέον επίπεδα ιεραρχίας. Παίρνουμε το παράδειγμα του δικτύου στο σχήμα 3.3. Οι κόμβοι 4.1, 4.2, 4.3, 4.4, 4.5 και 4 αποτελούν έναν cluster, στον οποίο ο κόμβος 4 είναι ο cluster-head. Ομοίως, για τους κόμβους 5.1, 5.2, 5.3, 5.4, 5.5, 5. Οι δύο cluster-heads (κόμβοι 4 και 5) έχουν ως super-cluster-head τον κόμβο 1, ο οποίος είναι cluster-head του κόμβου 1.1. Μπορούμε να συνεχίσουμε αυτό το σχήμα οργάνωσης αν θέλουμε και τελικά να φτιάξουμε ένα δένδρο μεταδόσεων με το βασικό σταθμό ως ρίζα του.

Τα βασικά χαρακτηριστικά του σχήματος αυτού είναι:

1. Όλοι οι κόμβοι μεταδίδουν πληροφορία μόνο στους cluster-heads τους, όποτε γλιτώνουν αρκετή ενέργεια, αφού κάθε φορά που θέλουν να στείλουν πληροφορία δεν έχουν το overhead να ανακαλύψουν γείτονες, κτλ.



Σχήμα 3.3: Η ιεραρχία και οι clusters στο TEEN

2. Μόνο οι cluster-head χρειάζεται να κάνουν επεξεργασία (π.χ data aggregation), οπότε γλιτώνουμε και άλλη ενέργεια.

3. Οι cluster-heads αλλάζουν από γύρο σε γύρο (όπως και στο LEACH) και έτσι κατανέμεται ομοιόμορφα η ενέργεια που καταναλώνεται συνολικά στο δίκτυο.

Το αρνητικό βέβαια του σχήματος αυτού είναι ότι οι εμπνευστές του δεν αναφέρουν τίποτα (!) σχετικά με το πώς ακριβώς γίνεται η οργάνωση των κόμβων σε παραπάνω από δύο επίπεδα ιεραρχίας. Στο επόμενο κεφάλαιο, όπου αναλύεται ο εξομοιωτής simDust, ο οποίος αναπτύχθηκε ειδικά για τους σκοπούς της εργασίας αυτής, αναλύεται πώς ακριβώς έγινε η υλοποίηση του TEEN, αφού είναι ένα από τα πρωτόκολλα που εξομοιώθηκαν. Χρειάστηκε να γίνουν ορισμένες παραδοχές, για τις οποίες ο αναγνώστης πρέπει να ανατρέξει στην αντίστοιχη ενότητα.

3.2.2 Λειτουργία του TEEN

Το TEEN, όπως ακριβώς και το LEACH, λειτουργεί σε γύρους, στην αρχή των οποίων γίνεται η οργάνωση των κόμβων του δικτύου σε clusters. Για περισ-

σότερες λεπτομέρειες, ο αναγνώστης μπορεί να ανατρέξει στην ενότητα για το LEACH. Στο TEEN, στην αρχή κάθε γύρου κάθε cluster-head μεταδίδει επιπλέον τις εξής δύο πληροφορίες.

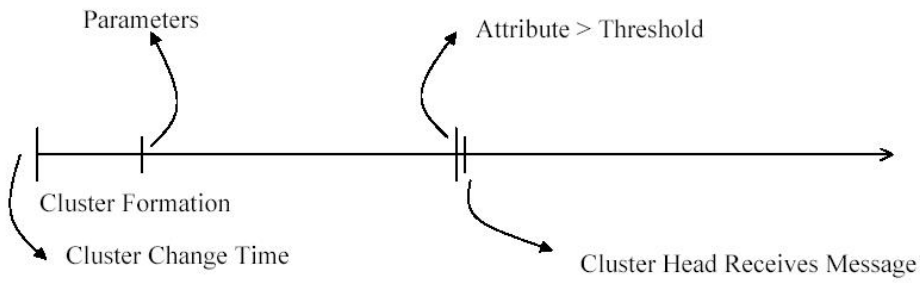
Μεγάλο φράγμα (Hard threshold): Αυτό είναι μια τιμή για το φαινόμενο που παρακολουθεί κάθε κόμβος του δικτύου. Αν ξεπεραστεί αυτή η τιμή, σημαίνει ότι εισερχόμαστε σε κρίσιμη κατάσταση. Τότε ο κόμβος ανάβει τον πομποδέκτη του και στέλνει μια σχετική αναφορά στον αντίστοιχο cluster-head, ενώ παράλληλα αρχίζει να καταγράφει τα δείγματα που παίρνει.

Μικρό φράγμα (Soft threshold): Αυτό είναι μια μικρή αλλαγή στην τιμή του μετρούμενου φαινομένου (σε σχέση με το μεγάλο φράγμα), η οποία θεωρείται όμως σημαντική. Αν έχει ξεπεραστεί το μεγάλο φράγμα και η τιμή του δείγματος που πήραμε από το πεδίο διαφέρει από την τιμή που είχαμε μετρήσει την προηγούμενη φορά που στείλαμε αναφορά στον βασικό σταθμό τουλάχιστον κατά το soft threshold, τότε πρέπει να στείλουμε νέα αναφορά.

Με άλλα λόγια, οι κόμβοι παίρνουν συνεχώς δείγματα από το πεδίο που βρίσκονται, αλλά τον περισσότερο καιρό έχουν κλειστούς τους πομποδέκτες τους. Όταν ξεπεραστεί για πρώτη φορά το μεγάλο φράγμα, τότε στέλνεται μια αναφορά προς το βασικό σταθμό και ξανακλείνει ο πομπός. Παράλληλα, καταγράφεται κάπου η τιμή που μετρήσαμε. Αν σε επόμενη μέτρηση η νέα τιμή διαφέρει από την τιμή που καταγράψαμε κατά το soft threshold, στέλνουμε νέα αναφορά κ.ο.κ. Αν η τιμή του μετρούμενου φαινομένου πέσει κάτω από το μεγάλο φράγμα, επανερχόμαστε στην αρχική κατάσταση. Επομένως, το μεγάλο φράγμα εμποδίζει τη μετάδοση τιμών που δε θεωρούνται κρίσιμες και το μικρό φράγμα εμποδίζει μικρές (μάλλον αδιάφορες) αυξομειώσεις των τιμών του μετρούμενου φαινομένου να περάσουν στο δίκτυο. Ως παράδειγμα μπορούμε να χρησιμοποιήσουμε την κλασσική εφαρμογή μέτρησης της θερμοκρασίας σε πεδίο. Θέτουμε ως μεγάλο φράγμα την τιμή $25\,^{\circ}\mathrm{C}$ και ως μικρό φράγμα την τιμή $0.5\,^{\circ}\mathrm{C}$. Μέχρι να ξεπεραστεί το μεγάλο φράγμα δεν στέλνουμε κανένα μήνυμα στο βασικό σταθμό. Από κει και πέρα, αν δεν μεταβληθεί η θερμοκρασία κατά το μικρό φράγμα είτε προς τα πάνω είτε προς τα κάτω, δεν στέλνουμε κανένα μήνυμα στο βασικό σταθμό.

Το χρονόδιάγραμμα της λειτουργίας του πρωτοκόλλου φαίνεται στο σχήμα 3.4. Διακρίνουμε δύο φάσεις σε κάθε γύρο όπως είπαμε, τη φάση αρχικοποίησης και τη σταθερή φάση, όπως και στο LEACH. Η διαφορά είναι ότι στο τέλος της φάσης αρχικοποίησης έχουμε τη μετάδοση από κάθε cluster-head των εξής δύο παραμέτρων:

- **Report time (T_R):** είναι ο χρόνος που μεσολαβεί μεταξύ δύο διαδοχικών μεταδόσεων μηνυμάτων από τους κόμβους του δικτύου.
- **Χαρακτηριστικά (Attributes):** είναι ένα σύνολο από φυσικές παραμέτρους που μπορεί να μετρήσει κάθε κόμβος του δικτύου, για τις οποίες ενδιαφέρεται ο τελικός χρήστης.



Σχήμα 3.4: Το χρονοδιάγραμμα λειτουργίας του TEEN

Σε κάθε T_R οι κόμβοι μέσα στους clusters μετράνε τα χαρακτηριστικά που επιθυμεί ο χρήστης και στέλνουν αναφορά στον cluster-head τους, ο οποίος συνενώνει όλα τα μηνύματα που παίρνει και τα στέλνει στα ανώτερα από αυτόν επίπεδα δρομολόγησης. Αφού οι παραπάνω δύο παράμετροι μεταδίδονται στην αρχή κάθε γύρου, μπορούμε να τις αλλάζουμε κατά βούληση και να ρυθμίζουμε τη λειτουργία του δικτύου. Στο LEACH αντί για το T_R έχουμε μια σταθερή περίοδο δειγματοληψίας.

3.2.3 Αποτελέσματα-Συμπεράσματα

Τα βασικά χαρακτηριστικά του πρωτοκόλλου, όπως περιγράφηκε, είναι τα εξής:

1. Η κρίσιμη πληροφορία φθάνει σχεδόν αμέσως στο βασικό σταθμό, λόγω του μεγάλου φράγματος, που σημαίνει ότι αν δεν υπάρχει συναγερμός σε όλο το πεδίο αλλά μόνο σε μέρος του δεν θα υπάρχει μεγάλη κίνηση στο δίκτυο, και του γεγονότος ότι δεν υπάρχουν πολλά hops ανάμεσα στο βασικό σταθμό και στους κόμβους του δικτύου.

2. Η κατανάλωση ενέργειας λόγω μηνυμάτων που περιέχουν πληροφορία για μετρήσεις είναι πολύ μικρότερη σε σχέση με ένα proactive δίκτυο.

3. Το μικρό φράγμα μπορεί να προσαρμοστεί κάθε φορά στις ανάγκες της εκάστοτε εφαρμογής, ανάλογα με το πόση ευαισθησία θέλουμε στις μετρήσεις μας. Προφανώς, μεγαλύτερη ευαισθησία σημαίνει μεγαλύτερη κατανάλωση ενέργειας.

4. Στην αρχή κάθε γύρου, τα φράγματα γίνονται εκ νέου γνωστά στους κόμβους του δικτύου, οπότε μπορούν να αλλάξουν κατά το δοκούν.

Το μειονέκτημα σε αυτήν την υπόθεση είναι ότι αν δεν παραβιαστούν τα φράγματα, οι κόμβοι δε θα στείλουν μετρήσεις στο βασικό σταθμό. Οπότε, το LEACH είναι μάλλον προτιμότερο για εφαρμογές στις οποίες θέλουμε να έχουμε συνολική εικόνα του δικτύου.

Τέλος, οι συγγραφείς του πρωτοκόλλου επιχείρησαν να κάνουν μία πειραματική σύγκρισή του με το LEACH, προκειμένουν να εξετάσουν τη συμπεριφορά του. Για το σκοπό αυτό τροποποίησαν τον κώδικα που είχε γραφεί για το LEACH στον ns-2, ώστε ο κόμβος να λειτουργεί σύμφωνα με τα hard και soft threshold. Παράλληλα, συνυπολόγισαν και την ενέργεια που καταναλώνει κάθε κόμβος όταν είναι idle και όταν χρησιμοποιεί τους αισθητήρες, οπότε τα αποτελέσματά τους μπορούν να θεωρηθούν κάπως πιο ακριβή από τα αντίστοιχα για το LEACH. Παρόλα αυτά, οι εξομοιώσεις τους περιορίζονται σε μικρό πλήθος κόμβων, δηλαδή 100, και μικρές διαστάσεις πεδίου. Επιπλέον, συνολικά έγιναν μόνο 5 επαναλήψεις (!).

Οι μετρικές στις οποίες βασίστηκαν ήταν η μέση κατανάλωση ενέργειας, δηλαδή η μέση ενέργεια που καταναλώνει κάθε κόμβος στο χρόνο, και ο συνολικός αριθμός "ενεργών" κόμβων στο δίκτυο, δηλαδή κόμβων με μη μηδενικά αποθέματα ενέργειας. Ως μετρούμενο φαινόμενο χρησιμοποιήθηκε η θερμοκρασία, ενώ το πεδίο χωρίστηκε σε τεταρτημόρια, στα οποία ανατιθόταν τυχαία μια τιμή κάθε 5 δευτερόλεπτα. Ως hard threshold χρησιμοποιήθηκε η τιμή $100\,^{\circ}\mathrm{F}$, ενώ ως soft threshold η τιμή $2\,^{\circ}\mathrm{F}$.

Τα αποτελέσματα της εξομοίωσης έδειξαν μία αισθητή βελτίωση, σε σχέση πάντα με το LEACH, της τάξης ακόμα και του 100%, πράγμα το οποίο ήταν αναμενόμενο. Το συμπέρασμα στο οποίο καταλήγουν είναι ότι το TEEN είναι αρκετά καλύτερο από το LEACH σε περιπτώσεις που θέλουμε όσο το δυνατόν μεγαλύτερη διάρκεια ζωής του δικτύου και άμεση αναφορά συμβάντων πίσω στο βασικό σταθμό, ενώ η φιλοσοφία του δεν ταιριάζει σε περιπτώσεις όπου απλά επιθυμούμε να έχουμε συνεχή επίβλεψη ενός πεδίου.

Μέρος ΙΙ

Εξομοίωση πρωτοκόλλων για δίκτυα έξυπνης σκόνης

Κεφάλαιο 4

Ο εξομοιωτής simDust

4.1 Λίγα λόγια γενικά για τον simDust

Ο simDust είναι ένας εξομοιωτής για δίκτυα έξυπνης σκόνης, ο οποίος αναπτύχθηκε στα πλαίσια της διπλωματικής αυτής. Δίνει τη δυνατότητα σε όποιον θελήσει να μελετήσει τα πρωτόκολλα που έχουν υλοποιηθεί (LTP, PFR, TEEN) και να δημιουργήσει δίκτυα έξυπνης σκόνης με πλήθος παραμέτρων. Οι παράμετροι αυτές περιλαμβάνουν πλήθος κόμβων δικτύου, διαστάσεις πεδίου, ακτίνα μετάδοσης του κάθε κόμβου, κατανομή κόμβων στο πεδίο, και άλλα πολλά σε συνάρτηση πάντα με το πρωτόκολλο που εξομοιώνεται κάθε φορά.

Εκτός από την εξομοίωση δικτύων έξυπνης σκόνης, ο simDust περιλαμβάνει και ένα εξίσου σημαντικό μέρος που αφορά στην οπτικοποίηση (animation) της λειτουργίας τέτοιων δικτύων. Μέσω της οπτικοποίησης αυτής, μπορεί κάποιος εύκολα και γρήγορα να δει πώς θα λειτουργήσει ένα δίκτυο έξυπνης σκόνης σε συνθήκες οι οποίες είναι παράμετροι εισόδου για το σύστημα.

Σε αυτό το σημείο οφείλω να αναφέρω ότι η υλοποίηση του εξομοιωτή αυτού έγινε σε συνεργασία με τους συνάδελφους Αντωνίου Αθανάσιο (προπτυχιακός φοιτητής του τμήματος) και Χατζηγιαννάκη Ιωάννη (υποψήφιος διδάκτορας του τμήματος). Συγκεκριμένα, η υλοποίηση του εξομοιωτή βασίστηκε αρχικά στον κώδικα για τον εξομοιωτή που έγραψε ο Ι. Χατζηγιαννάκης και χρησιμοποιήθηκε για τα πειράματα, τα οποία περιγράφονται στο [10]. Κατά τη διάρκεια της υλοποίησης έγιναν πολλές αλλαγές στην αρχική δομή του project, προστέθηκαν νέα πρωτόκολλα και δυνατότητες, και κυρίως προστέθηκε η δυνατότητα οπτικοποίησης των εξομοιούμενων πρωτοκόλλων.

Στις ενότητες της εργασίας αυτής που ακολουθούν, αναλύεται η λειτουργία και η σχεδίαση των διαφόρων μερών από τα οποία αποτελείται ο εξομοιωτής simDust.

4.1.1 Λόγοι που οδήγησαν στη δημιουργία του simDust

Ανάμεσα στους λόγους για την υλοποίηση του simDust ήταν ότι αφενός οι υπάρχοντες εξομοιωτές δικτύων, όπως ο **network simulator (ns-2)**, δεν υποστηρίζουν δίκτυα έξυπνης σκόνης, αφετέρου η προσθήκη ενός πρωτοκόλλου

για δίκτυα έξυπνης σκόνης στους ήδη υπάρχοντες εξομοιωτές είναι δύσκολη και χρονοβόρα και εξηγούμε αμέσως παρακάτω το γιατί.

Στον *ns-2* βέβαια, έχει γίνει η υλοποίηση κάποιων πραγμάτων όσον αφορά στα δίκτυα έξυπνης σκόνης, δηλαδή έχει υλοποιηθεί ένα πρότυπο ασύρματου δικτύου που έχει τα χαρακτηριστικά που βρίσκουμε στα δίκτυα έξυπνης σκόνης και επίσης έχουν υλοποιηθεί τα πρωτόκολλα LEACH και Directed Diffusion, αλλά αυτές οι υλοποιήσεις είναι κάπως αφαιρετικές έως ενός σημείου και επιπλέον έγιναν από ολόκληρες ομάδες σε πανεπιστήμια των ΗΠΑ, γεγονός που επιβεβαιώνει ότι το κόστος ενός τέτοιου εγχειρήματος ξεπερνά τους σκοπούς της εργασίας αυτής..

Επιπλέον, υπάρχει και το ζήτημα της αποδοτικής εξομοίωσης του δικτύου, δηλαδή οι υπάρχοντες εξομοιωτές δικτύων επειδή υποστηρίζουν πλήθος πρωτοκόλλων και χαρακτηριστικών, χάνουν σε απλότητα υλοποίησης και αποδοτικότητα. Κλασσικό παράδειγμα αποτελεί ο *ns-2*, ο οποίος υποστηρίζει πλήθος δικτυακών πρωτοκόλλων (TCP, UDP, κτλ), υλοποιήσεων σε επίπεδο `MAC` ενσύρματων(π.χ Ethernet) και ασύρματων δικτύων (π.χ IEEE 802.11), διαφορετικές τοπολογίες, και άλλα πολλά. Επιπλέον, έχει και τη δυνατότητα οπτικοποίησης ενός τέτοιου δικτύου με πολλές δυνατότητες.

Επομένως, είναι ένα αρκετά πολύπλοκο σύστημα και αυτό έχει αντίκτυπο στην απόδοσή του. Στα δίκτυα έξυπνης σκόνης μας ενδιαφέρει να κάνουμε εξομοιώσεις για μεγάλο αριθμό δικτυακών κόμβων και αυτή η εξομοίωση να δίνει αποτελέσματα σε σύντομο χρονικό διάστημα. Ο *ns-2* για μεγάλο αριθμό κόμβων δυστυχώς δεν μπορεί να δώσει τα αποτελέσματα που θέλουμε στο χρονικό περιθώριο που επιθυμούμε λόγω ακριβώς αυτής της πολυπλοκότητας στην υλοποίησή του.

Τέλος, η υλοποίηση αυτού του εξομοιωτή έδωσε την ευκαιρία για εμβάθυνση στο θέμα του σχεδιασμού πρωτοκόλλων για δίκτυα έξυπνης σκόνης και την καλύτερη κατανόησή τους. Μάλιστα, στην περίπτωση της υλοποίησης του πρωτοκόλλου TEEN έδωσε την ευκαιρία για να βελτιωθεί ο σχεδιασμός του και να υλοποιηθεί κάποια από τις προτάσεις που έκαναν στην εργασία τους οι αρχικοί σχεδιαστές του, **η οποία μέχρι τώρα δεν είχε υλοποιηθεί**(συγκεκριμένα το **ιεραρχικό clustering**).

4.1.2 Πλατφόρμα υλοποίησης

Ως πλατφόρμα υλοποίησης επιλέχθηκε το λειτουργικό σύστημα Linux και η αλγοριθμική βιβλιοθήκη `LEDA`. Πιο συγκεκριμένα, χρησιμοποιήθηκε η διανομή Mandrake 9.0 για το λειτουργικό σύστημα Linux και η έκδοση 4.3.1 της `LEDA` για compiler `gcc 3.2`.

Η χρήση της `LEDA` ήταν η προφανής λύση όσον αφορά στην επιλογή βιβλιοθήκης για την υλοποίηση του εξομοιωτή. Αυτό, γιατί αφενός παρέχει πλήθος αλγορίθμων και δομών δεδομένων με απλό και ενοποιημένο interface, κάνοντας εύκολη τη χρήση και κατανόησή τους, αφετέρου η υλοποίηση τους έχει γίνει με εξαιρετικά αποδοτικό τρόπο, ίσως τον πιο αποδοτικό από όλες τις σχετικές βιβλιοθήκες που υπάρχουν σήμερα διαθέσιμες. ΄Ετσι, δίνει τη δυνατότητα να

σχεδιαστεί και να υλοποιηθεί ένα πρόγραμμα που θα έχει εγγυημένα πολύ καλή απόδοση και όλα αυτά σε σχετικά σύντομο χρονικό διάστημα. Ο αναγνώστης που ενδιαφέρεται να κατανοήσει τον κώδικα για τον εξομοιωτή, θα χρειαστεί να ανατρέξει πολλές φορές στο εγχειρίδιο χρήσης της LEDA.

Το Linux από την άλλη πλευρά είναι ένα λειτουργικό σύστημα το οποίο είναι αξιόπιστο, αποδοτικά υλοποιημένο και διευκολύνει τη χρήση της LEDA. Άλλωστε, αν και η LEDA είναι διαθέσιμη για την πλατφόρμα Windows της Microsoft, η αρχική της υλοποίηση είναι για Linux και αυτό του δίνει ένα σχετικό πλεονέκτημα. Επιπλέον, το Linux είναι διαθέσιμο δωρεάν και έχει καθιερωθεί στη συνείδηση της επιστημονικής κοινότητας στο χώρο της Πληροφορικής. Επομένως, η πλειοψηφία των ανθρώπων που ασχολούνται ερευνητικά με την Πληροφορική μπορούν να δουν τον κώδικα για την υλοποίηση του εξομοιωτή και να τον τροποποιήσουν κατά το δοκούν.

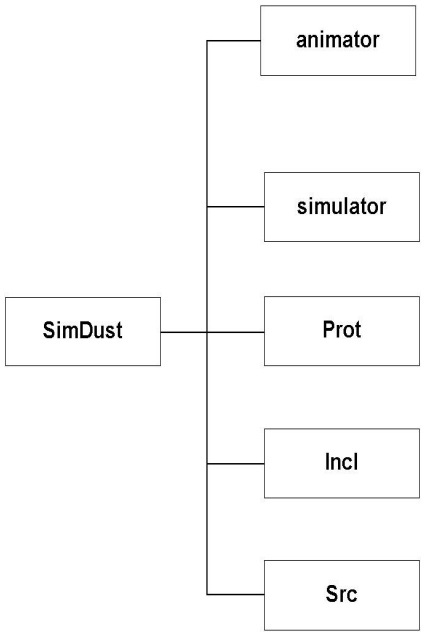
Ακόμα, αφού η LEDA είναι γραμμένη σε γλώσσα προγραμματισμού C++, η C++ ήταν η επόμενη προφανής επιλογή για την υλοποίηση του εξομοιωτή. Εκτός αυτού, η C++ είναι μια γλώσσα object-oriented, η οποία δίνει τη δυνατότητα για αντικειμενοστραφή σχεδίαση συνδυασμένη με αποδοτική υλοποίηση (συγκεκριμένα μια υλοποίηση σε C++ είναι ταχύτητα ελάχιστα πιο αργή από την αντίστοιχη σε C), που της δίνει ένα πλεονέκτημα σε σχέση με τις υπόλοιπες αντικειμενοστραφείς γλώσσες προγραμματισμού, όπως η Java. Γενικά, όταν μιλάμε για υλοποίηση δομών δεδομένων, εξομοιωτών, δικτυακών μοντέλων, η C++ είναι η γλώσσα που χρησιμοποιείται περισσότερο από τις υπόλοιπες σήμερα.

Τέλος, για την ομαλή συνεργασία κατά την ανάπτυξη του κώδικα για τον εξομοιωτή, χρησιμοποιήθηκε το πρόγραμμα CVS, το οποίο επιτρέπει εύκολα και γρήγορα την ταυτόχρονη εργασία μιας ομάδας προγραμματιστών στα αρχεία ενός κοινού project παρέχοντας ευκολίες όπως έλεγχο έκδοσης (version control), τήρηση αρχείων αλλαγών και άλλα πολλά χρήσιμα, που κάνουν ευκολότερη τη ζωή του προγραμματιστή.

4.2 Οργάνωση του κώδικα του εξομοιωτή simDust

Αρχικά να αναφέρω ότι ο κώδικας για τον εξομοιωτή είναι οργανωμένος σε αρχεία και καταλόγους που τονίζουν τη λογική του δομή. Η δομή αυτή παρουσιάζεται στην επόμενη εικόνα και εξηγείται αμέσως μετά. Υπάρχουν 5 κύριοι κατάλογοι όπως φαίνεται στο σχήμα 4.1.

Οι δύο πρώτοι κατάλογοι περιέχουν τον κώδικα για τα δύο εκτελέσιμα αρχεία από τα οποία αποτελείται ο εξομοιωτής, τον simulator και τον animator. Ο simulator είναι ο εξομοιωτής του οποίου η λειτουργία γίνεται μέσω της γραμμής εντολών και ο animator είναι το άλλο εκτελέσιμο αρχείο και ουσιαστικά είναι η οπτικοποίηση των αλγορίθμων που εκτελούνται στον εξομοιωτή. Ο animator είναι μια παραθυρική εφαρμογή, που εκμεταλλεύεται τις βιβλιοθήκες της LEDA για αναπαράσταση και animation δικτύων και τρέχει σε περιβάλλον X-Windows.



Σχήμα 4.1: Οργάνωση του κώδικα σε καταλόγους

Ο παραπάνω διαχωρισμός έγινε για να μπορεί να υλοποιηθεί πιο εύκολα ο animator χωρίς να κάνουμε πολύπλοκη τη λειτουργία του simulator. Από κει και πέρα και τα δύο modules χρησιμοποιούν τα ίδια αρχεία που υπάρχουν στους υπόλοιπους καταλόγους για την εκτέλεση των πρωτοκόλλων.

Πιο συγκεκριμένα, ο κατάλογος Incl περιέχει τα header αρχεία που χρησιμοποιούνται γενικά για την κατασκευή του δικτύου, κόμβων, μηνυμάτων, κτλ, ο κατάλογος Src περιέχει τις υλοποιήσεις που χρησιμοποιούν τα header αρχεία του Incl και ο κατάλογος Prot περιέχει Header αρχεία και υλοποίηση για τα πρωτόκολλα που εξομοιώνουμε στον simDust.

Πριν να αναλυθεί παραπέρα ο κώδικας και η οργάνωση του εξομοιωτή, αξίζει να αναφερθεί πώς γίνεται η μεταγλώττιση του κώδικα σε εκτελέσιμα αρχεία. Για το σκοπό αυτό υπάρχει ένα ειδικό αρχείο *Makefile* στον ανώτερο κατάλογο, το οποίο χρησιμοποιείται από το εργαλείο Make, το οποίο υπάρχει σε όλες τις διανομές Unix, για να καλέσει το μεταγλωττιστή (gcc). Το εργαλείο αυτό, έχοντας γράψει ένα κατάλληλο Makefile, κάνει έλεγχο στα αρχεία που υποδεικνύουμε και εξετάζει κατά πόσον έχουν μεταβληθεί από την τελευταία φορά που έχει γίνει μεταγλώττιση και κατά πόσο τα αρχεία που παράγονται είναι ενημερωμένα. Επίσης, μπορούμε να εισάγουμε εξαρτήσεις μεταξύ αρχείων για να γίνεται πιο συστηματικά αυτή η εργασία και σε λιγότερο χρόνο. Για περισσότερες λεπτομέρειες, ο αναγνώστης μπορεί να ανατρέξει στο εγχειρίδιο του Make.

Πιο συγκεκριμένα, αν είμαστε έστω στον κατάλογο */home/mylonasg/simDust*,

ο οποίος είναι ο κατάλογος στον οποίο βρίσκεται το αρχείο Makefile και οι υποκατάλογοι με τα αρχεία του κώδικα, αν δώσουμε στη γραμμή εντολών:

```
mylonasg@localhost> make
```

το εργαλείο make θα κάνει αυτόματα τη μεταγλώττιση του κώδικα και θα παράγει δύο εκτελέσιμα, τον animator και το simulator. Αν θέλουμε να μεταγλωττίσουμε μόνο τον animator δίνουμε make animator, και παρόμοια για το simulator.

Στη συνέχεια αναλύονται κάποιες τάξεις, οι οποίες είναι σημαντικές για τη λειτουργία του εξομοιωτή, και ακολουθεί μια λιγότερο λεπτομερής ανάλυση του κώδικα ανά αρχείο.

4.2.1 Σημαντικές τάξεις για τη λειτουργία του εξομοιωτή

Η τάξη Execution

Η υλοποίηση της τάξης αυτής γίνεται σε δύο αρχεία, τα Execution.h και Execution.cpp, τα οποία περιέχονται στους καταλόγους Incl και Src αντίστοιχα. Η τάξη αυτή είναι μια αρκετά αφαιρετική αναπαράσταση του πεδίου που βρίσκεται ένα δίκτυο έξυπνης σκόνης. Περιέχει πληροφορίες όπως οι διαστάσεις του πεδίου, δείκτες στους κόμβους που βρίσκονται στο πεδίο, τον αριθμό του γύρου στον οποίο βρισκόμαστε ανά πάση στιγμή, μια λίστα με τα γεγονότα που συμβαίνουν στον κάθε γύρο και θα εκτελεστούν στον τρέχοντα γύρο.

Η τάξη αυτή είναι βασική για τη λειτουργία του δικτύου, αφού κατά την δημιουργία της αρχικοποιείται το δίκτυο, δηλαδή ανάλογα με την κατανομή που έχουμε δώσει ως παράμετρο τοποθετούνται οι κόμβοι στο δίκτυο και έπειτα δημιουργείται ένας γράφος με αυτούς τους κόμβους. Επιπλέον, κάθε κόμβος χτίζει μια λίστα με τους γείτονές του, δηλαδή τους κόμβους που βρίσκονται σε μια ορισμένη απόσταση από αυτόν (ακτίνα μετάδοσης).

Από την τάξη αυτήν κληρονομούν οι τάξεις Experiment και Animation, οι οποίες είναι ουσιαστικά ο simulator και ο animator αντίστοιχα.

Η τάξη Particle

Η τάξη αυτή είναι ουσιαστικά ο κόμβος του δικτύου έξυπνης σκόνης. Η τάξη αυτή κληρονομεί από την τάξη point της `LEDA`, η οποία συμβολίζει ένα σημείο στο επίπεδο με συντεταγμένες X και Y. Προσθέτουμε πληροφορία όπως το ID του κόμβου, λίστα με γείτονες, διαθέσιμη ενέργεια, μεταβλητές που δείχνουν την κατάσταση του κόμβου (αν περιμένει να δεχθεί μήνυμα, αν είναι ενεργός),κτλ. Ο κώδικας για την τάξη αυτή βρίσκεται στα αρχεία Incl/Particle.h και Src/Particle.cpp.

Η τάξη αυτή είναι εικονική καθώς είναι πολύ γενική και χρησιμοποιείται για την παραγωγή τάξεων κόμβων που έχουν ειδικό ρόλο, όπως ο κόμβος - βάση (SINKParticle) και οι κόμβοι για καθένα από τα πρωτόκολλα που εξομοιώνουμε (LTPParticle, PFRParticle, TEENParticle), οι οποίοι έχουν τα εξειδικευμένα χαρακτηριστικά που χρειάζονται για κάθε πρωτόκολλο.

Επίσης, παρέχει ένα interface για να κάνουμε την εναλλαγή μεταξύ των καταστάσεων sleep και awake για τους κόμβους του δικτύου και έχει έναν δείκτη που δείχνει σε ένα αντικείμενο Network, το οποίο είναι η τάξη που αναλύεται αμέσως μετά.

Η τάξη Network

Η τάξη αυτή είναι μια αφαίρεση που χρησιμοποιούμε για να αναπαραστήσουμε το δίκτυο μεταξύ των κόμβων. Όταν λέμε δίκτυο εννοούμε τις δυνατότητες επικοινωνίας μεταξύ των κόμβων και όχι χαρακτηριστικά του στυλ διαστάσεις, που βρίσκονται στην τάξη Execution ή στην τάξη Particle.

Έτσι, εδώ υπάρχουν συναρτήσεις που διεκπεραιώνουν την επικοινωνία μεταξύ κόμβων, όπως αποστολή μηνύματος σε γείτονα, broadcast σε όλους τους γείτονες, παραλαβή μηνύματος κτλ. Μάλιστα υπάρχουν δύο διαφορετικές υλοποιήσεις της τάξης αυτής,μία για τον simulator (περιέχεται στο αρχείο simulator/Network_exp.cpp) και μία για τον animator (περιέχεται στο αρχείο animator/Network_anim.cpp), ενώ ο ορισμός της είναι στο αρχείο Incl/Network.h.

Ο λόγος πίσω από αυτό το σχεδιασμό είναι ότι τα δύο εκτελέσιμα καθώς εκτελούν ουσιαστικά τους ίδιους αλγόριθμους καλούν τις ίδιες συναρτήσεις. Στην περίπτωση του animator όμως, υπάρχει επιπλέον και η οπτικοποίηση της λειτουργίας των πρωτοκόλλων, οπότε πρέπει να χρησιμοποιηθούν επιπλέον και οι ανάλογες βιβλιοθήκες της `LEDA`. Για να μην γίνει υπερβολικά πολύπλοκη η υλοποίηση των συναρτήσεων της τάξης αυτής, επιλέξαμε αυτό το δρόμο.

Η τάξη Info

Η τάξη αυτή χρησιμοποιείται στα αντικείμενα της τάξης Message, η οποία αναλύεται αμέσως μετά. Ουσιαστικά αναπαριστά την πληροφορία που μεταφέρει ένα υποτιθέμενο μήνυμα και είναι απαραίτητη για τη λειτουργία του πρωτοκόλλου. Ο κώδικας για την τάξη αυτή βρίσκεται στα αρχεία Incl/Info.h και Src/Info.cpp.

Η πληροφορία αυτή περιλαμβάνει το ID του κόμβου που δημιούργησε την πληροφορία αυτή, τις συντεταγμένες του, τη χρονική στιγμή κατά την οποία δημιουργήθηκε και άλλες πληροφορίες που έχουν σχέση με το κάθε πρωτόκολλο, όπως αριθμός hops στο δίκτυο από τα οποία έχει περάσει μέχρι στιγμής , στο LTP αριθμός backtracks, κ.α.

Η τάξη Message

Η τάξη αυτή αναπαριστά τα μηνύματα που ανταλλάσσονται μεταξύ κόμβων. Όταν δημιουργείται ένα μήνυμα του αναθέτουμε κάποια πληροφορία (Info) και από κει και πέρα το μήνυμα μπορεί να είναι διαφόρων τύπων και να εξυπηρετεί διαφορετικούς σκοπούς ανάλογα με το πρωτόκολλο που εξετάζεται κάθε φορά. Μπορεί να είναι π.χ ένα μήνυμα για να ενημερώνει τους γείτονες ότι θέλει να στείλει ένα μήνυμα προς το base station, με τη σειρά τους να απαντούν ότι είναι πρόθυμοι να το μεταφέρουν με ένα άλλου τύπου μήνυμα, κ.ο.κ. Ο κώδικας για την τάξη αυτή βρίσκεται στα αρχεία Incl/Message.h και Src/Message.cpp.

Η τάξη Event

Η τάξη αυτή χρησιμοποιείται για να αναπαραστήσει ένα γεγονός, το οποίο συμβαίνει σε κάποιον κόμβο του δικτύου έξυπνης σκόνης σε κάποια χρονική στιγμή και επηρεάζει τη λειτουργία του. Αυτό το γεγονός μπορεί να περιλαμβάνει το triggering ενός κόμβου, που προσομοιώνει την κατάσταση που έχουμε κάποιο συμβάν στο πεδίο που επιβάλλει μέτρηση της αντίστοιχης ποσότητας από τον κόμβο και αποστολή μηνύματος προς το base station, ή λήψη μηνύματος από κάποιο γειτονικό κόμβο. Τα αρχεία που περιλαμβάνουν τον κώδικα για αυτή την τάξη είναι τα Incl/Event.h και Src/Event.cpp.

Η τάξη Experiment

Η τάξη αυτή κληρονομεί από την Execution και περιγράφει ουσιαστικά τον simulator. Με τη δημιουργία του αντικειμένου αυτού και με την κλήση της συνάρτησης μέλους του execute, ουσιαστικά αρχίζει η λειτουργία του simulator, γίνεται αρχικοποίηση του δικτύου, κτλ. Η execute είναι ένας βρόχος που επαναλαμβάνεται για όσους γύρους καθορίσουμε εμείς ή όταν φτάσουν τα μηνύματα που έχουμε δημιουργήσει σε διάφορους κόμβους στον base station. Ο κώδικας για την τάξη αυτή περιέχεται στα αρχεία Simulator/Experiment.h και Simulator/Experiment.cpp.

Η τάξη Animation

Η τάξη αυτή είναι το αντίστοιχο του Experiment για τον animator. Ουσιαστικά έχει επιπρόσθετα ένα αντικείμενο GraphWin της `LEDA`, για το οποίο αναφέρω αμέσως μετά, και διαφορετική υλοποίηση για το αντικείμενο Network. ΄Ετσι. Εκτελείται ουσιαστικά ο βρόχος που εκτελείται και στον simulator αλλά παράλληλα γίνεται κι ένα animation στο δίκτυο έξυπνης σκόνης που εξομοιώνεται. Ο κώδικας για την τάξη αυτή περιέχεται στα αρχεία Animator/Animation.h και Animator/Animation.cpp.

Η βάση για την οπτικοποίηση: η τάξη GraphWin της LEDA

Η τάξη GraphWin είναι μία τάξη της `LEDA` η οποία είναι εξαιρετικά χρήσιμη στην αναπαράσταση γραφοθεωρητικών αλγορίθμων και γενικά οτιδήποτε έχει σχέση με γραφήματα. Βασίζεται στην τάξη Window, επίσης της `LEDA`, η οποία είναι μια αναπαράσταση ενός παραθύρου στο περιβάλλον `X-Windows` και η οποία παρέχει πλήθος συναρτήσεων για να μπορεί κάποιος να μεταβάλλει αυτό που βρίσκέται μέσα στο συγκεκριμένο παράθυρο.

΄Ετσι, σε ένα αντικείμενο Window μπορεί κάποιος να σχεδιάσει γεωμετρικά σχήματα, να εμφανίσει κείμενο, να αλλάξει παραμέτρους όπως μέγεθος παραθύρου, χρώμα στο φόντο, και άλλα πολλά που δίνουν μεγάλη ελευθερία και ευκολία στον προγραμματιστή να εμφανίσει σε ένα παράθυρο αυτό που θέλει.

΄Ενα αντικείμενο GraphWin είναι ένα αντικείμενο Window συν ένα γράφημα και τις διάφορες λειτουργίες που μπορούμε να επιτελέσουμε σε αυτό, με την έννοια που αυτό υπάρχει στη `LEDA`. ΄Ετσι, υπάρχει ένα σύνολο κόμβων και

ακμών του γραφήματος που αντιστοιχίζεται με την αναπαράστασή τους σε ένα αντικείμενο Window. Μπορεί κάποιος από εκεί και πέρα να κάνει πρακτικά ότι θέλει σε αυτό το γράφημα: να του προσθέσει ακμές, να του προσθέσει κόμβους, να αλλάξει το σχήμα των κόμβων, να βάλει ετικέτες στα μέρη του γραφήματος, κτλ. Μπορούμε λοιπόν να κατασκευάσουμε ένα γράφημα για τις ανάγκες ενός αλγορίθμου και να τροποποιούμε ανάλογα τα χαρακτηριστικά του γραφήματος αυτού για να δείξουμε την πρόοδο του αλγορίθμου.

Βεβαίως, για να καταλάβει κάποιος τον τρόπο που λειτουργούν όλα αυτά πρέπει οπωσδήποτε να διαβάσει στο εγχειρίδιο χρήσης της LEDA τα κεφάλαια σχετικά με γραφήματα, το αντικείμενο Window και φυσικά το αντικείμενο GraphWin.

4.3 Σημαντικές συναρτήσεις και δηλώσεις τάξεων που περιλαμβάνονται στον κώδικα ανά αρχείο κώδικα και παρουσίασή τους εν συντομία.

Αρχικά να αναφέρουμε κάποιες παραδοχές που ακολουθήσαμε στη συγγραφή του κώδικα, όπως ότι τα μέλη μιας τάξης ως επί το πλείστον είναι protected και έχουν το πρόθεμα m_ στη δήλωσή τους, π.χ m_round, και υπάρχουν συναρτήσεις ειδικά για να επιστρέφουν και να θέτουν τις τιμές τους με ονόματα όπως setRound και getRound, με τις οποίες δε θα ασχοληθούμε ιδιαίτερα.

Ακολουθεί μια παρουσίαση του κώδικα ανά αρχείο. Τα header αρχεία παρουσιάζονται σχεδόν αυτούσια σε αρκετές περιπτώσεις, ενώ στα αντίστοιχα .cpp αρχεία γίνεται απλά ένας σχολιασμός για τις πιο ενδιαφέρουσες συναρτήσεις (σε αντίθετη περίπτωση θα χρειαζόμασταν 200 σελίδες μόνο για τον κώδικα!). Τέλος, ο αναγνώστης χρειάζεται να έχει διαβάσει και τα κεφάλαια 2 και 3, προκειμένου να έχει μια εικόνα για τα πρωτόκολλα που εξομοιώσαμε.

4.3.1 Incl/Cache_Info_Obj.h

Η τάξη Cache_Info_Obj αντιστοιχεί σε μία θέση της cache κάθε κόμβου του δικτύου, στην οποία αποθηκεύονται για κάποιο διάστημα τα μηνύματα που έχουν ληφθεί από τους γείτονες. Αυτό χρησιμεύει στην περίπτωση που το ίδιο μήνυμα στέλνεται από περισσότερους από έναν γείτονες. Αν υπάρχει αυτή η cache, δε θα προωθηθεί το ίδιο μήνυμα προς το βασικό σταθμό πολλές φορές. Χρησιμοποιείται στα πρωτόκολλα PFR και LTP.

```
class Cache_Info_Obj {

public:
  Cache_Info_Obj(Info *the_info ,double arrival_ts,
          double lifetime = LIFEOFCACHEOBJ);

  double   getlifetime();
  void     setlifetime(double life);
  double   getArrivalTimeStamp();
```

```
  Info*     getInfo();
  void      setSend(bool);
  bool      getSend();
  Particle* getPrevious();
  void      setPrevious(Particle *some_sender);
  void      putInBlackList(long particl_ID);
  bool      existsInBlackList(long particl_ID);


protected:
  double    m_arrival_ts;
  double    m_lifetime;
  Info*     m_storedInfo;
  Particle* m_previous;
  idList*   m_backtrack;
  bool      m_been_sent;
};
```

Σημαντικά μέλη είναι:

m _lifetime: Ο χρόνος που διατηρείται ένα τέτοιο αντικείμενο στην cache του κόμβου.
m_arrival_ts: Η στιγμή της άφιξης του μηνύματος στον κόμβο.
m_storedInfo: Η πληροφορία που κουβαλά το μήνυμα .
m_been_sent: Αν το μήνυμα έχει προωθηθεί σε άλλον κόμβο.

4.3.2 Incl/constants.h

Στο αρχείο αυτό περιέχονται οι περισσότερες από τις σταθερές που χρησιμοποιούμε στον εξομοιωτή, με τη μορφή διαδοχικών #define. Παραθέτουμε τις περισσότερες σταθερές που είναι σημαντικές και χρησιμοποιούνται στα περισσότερα αρχεία. Εξηγήσεις για κάθε μεταβλητή δίνονται στα σχόλια του κώδικα.

```
#define PI     3.14159265358979

// node distributions

#define SETUP_RANDOM    0
#define SETUP_POISSON   1
#define SETUP_LATTICE   2

// base station types. Wall, sink outside the field, or sink inside the field

#define SETUP_WALL  0
#define SETUP_SINK  1
#define SETUP_SINK_34  2

// Event types

#define EVENT_RECEIVE   1
#define EVENT_SENSE     2
```

```
// Message Types

#define MESSAGE_INFO    1
#define MESSAGE_LTP_B   2
#define MESSAGE_LTP_S   3
#define MESSAGE_LTP_LS1 4
#define MESSAGE_LTP_LS2 5
#define MESSAGE_TEEN_ADVERTISEMENT  6 // -- For TEEN, Advertisement
#define MESSAGE_TEEN_INFO   7     // -- TEEN, Actual info from nodes to Cluster Heads

// Particle types

#define PARTICLE_SINK   3   // -- Dummy Protocol (plays the role of the Sink/Wall)
#define PARTICLE_LTP    0   // -- LTP Protocol (Local Target Protocol)
#define PARTICLE_PFR    1   // -- PFR Protocol (Probabilistic Forwarding Protocol)
#define PARTICLE_TEEN 2   // -- TEEN Protocol

#define TEEN_SIMPLE_PARTICLE    1
#define TEEN_CLUSTER_HEAD   2
#define TEEN_SUPER_CLUSTER_HEAD 3

// TEEN specific

#define P_TEEN 0.05     // -- Probability that a node becomes cluster in TEEN
#define TEEN_EPOCH_DURATION 20  // -- How many rounds is an epoch in TEEN
protocol

#define LIFEOFCACHEOBJ 10000    // o xronos pou kratatai mia kataxwrhsh sthn cache
enos Particle
#define KEEP_FOREVER    -1  // Define gia na kratame gia panta mia kataxwrhsh sthn
cache

// ENERGY. See LEACH paper

#define ENERGY_ELEC 0.00000005
#define ENERGY_AMP  0.0000000001
#define BITS_FOR_TEEN_ADVERTISEMENT   128
#define BITS_FOR_INFORMING_CLUSTER_HEAD 40  //plhroforia ston clusterhead oti
mphkame ston cluster tou
            // 32 bit node ID + 1 byte yes
#define BITS_FOR_TDMA_SCHED   32

#define BITS_FOR_IDLE_STATE   1
#define BITS_FOR_POWERUP_STATE  3

#define INFO_SIZE   8192
#define DEBUG_TRACE 1   // Generate trace file
```

4.3.3 Incl/Event.h

Εδώ είναι ο ορισμός της τάξης Event της οποίας η σημασία αναλύθηκε πριν.

```
class Event {

public:
   Event(long type);
   ´Event();

   double  getTimeStamp();
   void    setTimeStamp(double ts);
   long    getType();
   void    setMessage(Message* msg);
   Message* getMessage();

protected:
   double    m_ts;
   long      m_type;
   Message*  m_message;
};
```

m_ts: είναι ο χρόνος που δημιουργήθηκε το event, το παίρνουμε με το κλειδί που αποθηκεύτηκε στην priority queue.
m_type: είναι ο τύπος event, δηλαδή sense ή receive.
m_message: είναι ένας δείκτης στο μήνυμα που αντιστοιχεί (αν υπάρχει τέτοιο) το event.
gettype: επιστρέφει απλά τον τύπο του Event.
getMessage: επιστρέφει το δείκτη σε μήνυμα του αντικείμένου.
setMessage: θέτει ως μήνυμα για το συγκεκριμένο Event το m_message.

4.3.4 Incl/Execution.h

Εδώ γίνεται ο ορισμός της βασικής τάξης Execution.

```
class Execution {

public:
   Execution(short setup, short controlType,
       short particleType, long totPoints,
       double X, double Y,
       long periodA, long periodS,
       double angleMax, double R, double deltaR,
       double lamda);
   virtual    Execution();
   void       triggerSource(int id = 0);
   Run*       getResults();
   virtual void    execute(d_array<int,set<long> > &cronojon_IDs_n_rounds) = 0;

protected:
   double  m_X, m_Y;
   long    m_round;
   short   m_controlType;
   long    m_TotalEvents;
```

```
  sdPlane m_particles;
  idList   m_particleList;
  Particle*  m_sink;
};
```

m_X, m_Y: είναι οι διαστάσεις του πεδίου στο οποίο βρίσκονται διεσπαρμένοι οι κόμβοι του δικτύου.
m_round: ο γύρος εκτέλεσης που βρισκόμαστε ανά πάσα στιγμή.
m_controlType: αν χρησιμοποιούμε βασικό σταθμό (sink) ή την ιδέα του τείχους (wall, ενότητα για το sleep-awake).
m_TotalEvents: τα events μέχρι στιγμής.
m_particles: ο αριθμός κόμβων στο δίκτυο.
m_particleList: μια ουρά με όλα τους κόμβους του δικτύου. **m_sink:** δείκτης στο βασικό σταθμό.

4.3.5 Incl/include.h

Αυτό το header αρχείο είναι από τα πλέον σημαντικά για τη μεταγλώττιση του κώδικα. Εδώ γίνεται η συμπερίληψη βιβλιοθηκών της LEDA.Επίσης, γίνονται κάποια σημαντικά typedef, τα οποία πρέπει να έχει υπόψη του ο αναγνώστης:

```
#include <LEDA/point.h>
#include <LEDA/p_queue.h>
#include <LEDA/d_array.h>
#include <LEDA/list.h>
#include <LEDA/random_source.h>
#include <LEDA/set.h>
#include <LEDA/color.h>

typedef list< long > idList;
typedef p_queue<double, Event* > eventQueue;
typedef list< Particle* > particleList;
typedef d_array<int, Particle*> sdPlane;
```

4.3.6 Incl/Info.h

Η χρήση της Info αναλύθηκε προηγουμένως, Εδώ έχουμε constructors για την τάξη αυτή.

```
class Info {

public:
  Info(long particleID, long hops = 0, long backtracks = 0, long beta = 0,double
start_timestamp=0.0,point initiator_point = point(0,0));

  long   getInitiator();
  double getInitiator_ts();
  point  getInitiator_trace();
  long   getHops();
```

```
   long   getBacktracks();    // Used by LTP
   void   incBacktracks();    // Used by LTP
   long   getBeta();      // Used by PFR
   void   decBeta();      // Used by PFR
   long   getBits();


protected:
   long    m_initiatorID;
   double  m_initiator_timestamp;
   point   m_initiator_trace;
   long    m_hops;
   friend class    Message;
};
```

m_initiator_timestamp: επιστρέφει την στιγμή που άρχισε να διαδίδεται η συγκεκριμένη πληροφορία.
m_initiator_trace(): επιστρέφει τις συντεταγμένες του κόμβου που δημιούργησε τη συγκεκριμένη πληροφορία (initiator).
m_initiatorID: το ID του initiator.
m_hops: ο αριθμός κόμβων από τους οποίους έχει περάσει ως τώρα η πληροφορία.

4.3.7 Incl/Message.h

΄Ενα αντικείμενο message, όπως είδαμε περιέχει μια πληροφορία (Info), και χρησιμοποιείται για να περάσει αυτή η πληροφορία από κόμβο σε κόμβο, ή εξυπηρετεί τις ανάγκες του πρωτοκόλλου που χρησιμοποιούμε.

```
class Message {

public:
   Message(long type = 0, Particle * p = nil, Info * i = nil);
   Message(Message * m);
   ΄Message();
   long    getType();
   Particle * getSender();
   Info *  getInfo();
   long    getHops();

protected:
   long       m_type;
   Particle *  m_sender;
   Info *  m_info;

   friend class    Network;

   void    doTransmit();
};
```

m_type: ο τύπος του μηνύματος, μπορεί να είναι απλό μήνυμα που περνά πληροφορία από τον ένα κόμβο στον άλλο ή ειδικό μήνυμα για τις ανάγκες

του αλγορίθμου.
m_sender: ο αποστολέας του μηνύματος (ο αποστολέας της πληροφορίας είναι ο initiator).
m_info: η πληροφορία που μεταφέρει το μήνυμα.

4.3.8 Incl/misc.h

Είναι ένα αρχείο, το οποίο περιέχει αρκετές συναρτήσεις, χρήσιμες κυρίως για την αρχικοποίηση του δικτύου. Παραθέτουμε τις σημαντικότερες.

newParticle: Δημιουργεί ένα νέο particle(δηλαδή κόμβο του δικτύου) ανάλογα με το πρωτόκολλο που τρέχουμε και δίνει τις ανάλογες παραμέτρους. Η υλοποίησή της βρίσκεται στο αρχείο Prot/protocols.h.

makeSetupRANDOM: Δημιουργεί κόμβους και τους κατανέμει στο πεδίο με τυχαίο τρόπο. Αντίστοιχα, οι makeSetupPOISSON και makeSetupLATTICE δημιουργούν δίκτυο με κατανομή Poisson και Lattice (πίνακας).

makeNeighbourhoods: Δημιουργεί ένα γράφο με τις συνδέσεις μεταξύ των κόμβων του δικτύου, ουσιαστικά ο κάθε κόμβος έχει μια λίστα με τους κόμβους οι οποίοι βρίσκονται μέσα στην ακτίνα μετάδοσής του. ΄Εχοντας αυτή την πληροφορία, σε κάθε γύρο των πρωτοκόλλων στέλνουμε μηνύματα κατευθείαν στους γείτονες χωρίς να χρειάζεται κάθε φορά να τους βρίσκουμε.

4.3.9 Incl/Network.h

Εδώ περιέχονται συναρτήσεις χρήσιμες για διαδικασίες δικτύου, όπως broadcast μηνύματος στους γείτονες και άλλα. Η υλοποίηση των συναρτήσεων αυτών είναι διαφορετική για τον simulator και τον animator (βρίσκεται στα αρχεία Simulator/Network_Exp.cpp και Animator/Network_anim.cpp αντίστοιχα).

```
class Network {

public:
  Network();

  void   setClock(long clockID);
  long   getClock();

  inline long getTotReceivedEvents() {
          return m_totReceivedEvents;
        }

  // FOR TEEN PURPOSES ONLY

  void   colorCluster(Particle* s, color cl);
  void   uncolorCluster(Particle* s);
  void   ClusterSetup(Particle* clusterHead);
  void   addNode2Cluster(Particle* me, color cl);
  void   sh_C_Head(Particle*, Particle*);
```

```
   void   colorSuperClusterHead(Particle* s, color cl);

protected:
   long   m_clockID;
   long   m_eventID;

   friend class Particle;
   friend class SINKParticle;
   friend class TEENParticle;

   void   broadcastMessage(Particle* s, Message* m);
   void   broadcastMessageAngle(Particle* s, Message* m);
   void   Network::broadcastTEENMessage(Particle *s);

   void   transmitMessage(Particle* s, Message* m, Particle* rcv);
   void   transmitMessageAngle(Particle* s, Message* m, Particle* rcv);

   void   no_transmitMessage(Particle* s); //new
   void   SinkLogsMessage(Event* thisEvent);

   void   aWaken(Particle* r);
   void   aSleep(Particle* r);

   void   receiveMessage(Particle* r, Message* m);

   void   generateSenseEvent(Particle* p);

   double getNextTimeStamp();

   long   m_totReceivedEvents;

};
```

Οι συναρτήσεις με τη λέξη broadcast είναι για μετάδοση μηνύματος στους γείτονες, ενώ η λέξη transmit υπονοεί μετάδοση σε συγκεκριμένους γείτονες (και όχι σε όλους). Οι συναρτήσεις που είναι ειδικά για το TEEN, γράφτηκαν ξεχωριστά γιατί η λειτουργία του απαιτεί γνώση του γύρου στον οποίο βρισκόμαστε. Η μεταβλητή m_clockID δείχνει τον γύρο στον οποίο βρισκόμαστε.

4.3.10 Incl/Particle.h

΄Ενα από τα πλέον σημαντικά αρχεία κώδικα., στο οποίο ορίζεται η πολύ σημαντική τάξη Particle.

```
class Particle : public point {

public:

   enum State {
      UNDEF_STATE,
      AWAKE,
```

```
    SLEEP,
    SENSING,
    ALWAYS_ON,
    DEAD
  };

  ...

  void   spendEnergy(double energy);
  ...
  void   doSetup();
  ...

  double clusterRadius;
  double clusterSize;


protected:
  long   m_id, m_type;
  State  m_state;
  long   m_periodA, m_periodS;
  double m_clock;
  double m_energy;

  point  m_WallPoint;
  Particle*  m_RealSink;
  bool   m_NextToWall;

  particleList*  m_neighbours;
  particleList*  m_neighbours_angle;
  eventQueue* m_events;

  Info*      m_info;

  list<Cache_Info_Obj * >  m_info_cache;

  long       m_totTrans, m_totRcvs;
  long       m_totParticles;
  bool       m_waiting, m_participated;

  Network*   m_network;

  friend class   Network;

  virtual void doExecute() = 0;
  void   doSleepAwake();
  void   doProcessEvent();

  virtual void handleSenseEvent(Event*) = 0;
  virtual void handleMessageEvent(Event*) = 0;

  void   broadcast(Message *m);
  void   broadcastAngle(Message *m);
  void   transmit(Message *m, Particle *p);
```

```
    void   transmitAngle(Message *m, Particle *p);
    void   no_transmit(); //new
};
```

Υπάρχει στην αρχή μία απαρίθμηση, η state, που δηλώνει τις καταστάσεις που μπορεί να βρίσκεται ο κόμβος (AWAKE σημαίνει ότι ο κόμβος λειτουργεί κανονικά, SLEEP το αντίθετο, ALWAYS_ON σημαίνει ότι ο κόμβος δεν κάνει εναλλαγές καταστάσεων, DEAD σημαίνει ότι έχει τελειώσει η ενέργειά του και δε μπορεί να κάνει τίποτα από εδώ και πέρα).

clusterRadius: αν ο συγκεκριμένος κόμβος έχει γίνει cluster-head στο TEEN, αυτή είναι η απόσταση του πιο απομακρυσμένου κόμβου μέσα στον cluster, για να ξέρουμε σε τι ακτίνα θα κάνουμε broadcast.
clusterSize: αν ο συγκεκριμένος κόμβος έχει γίνει cluster-head στο TEEN, πόσοι κόμβοι βρίσκονται μέσα στον cluster του. Χρειάζεται για να υπολογίζουμε το μέγεθος του `TDMA` schedule που κάνει broadcast.
m_id: ο αριθμός ID του κόμβου.
m_type: αν ο κόμβος είναι sink ή όχι.
m_state: η κατάσταση του κόμβου.
m_periodA, m_periodS: χρονικές περίοδοι που ο κόμβος είναι awake και asleep αντίστοιχα.
m_clock: το ρολόι του κόμβου.
m_energy: η διαθέσιμη ενέργεια του κόμβου. Κάθε κόμβος ξεκινά τη λειτουργία με ένα ορισμένο ποσό ενέργειας, το οποίο μειώνεται σε κάθε γύρο.
m_WallPoint: όταν έχουμε wall αντί για sink, αυτό δηλώνει τη γενική κατεύθυνση που βρίσκεται το wall.
m_RealSink: ο πραγματικός κόμβος sink, ακόμα και στην περίπτωση που έχουμε wall.
m_NextToWall: Boolean μεταβλητή, που δηλώνει αν είμαστε ένα βήμα πριν το wall, δηλαδή μπορούμε να μεταδώσουμε μήνυμα στο wall από αυτόν τον κόμβο.
m_neighbours: λίστα με τους κόμβους που βρίσκονται στην ακτίνα μετάδοσης μας.
m_neighbours_angle: λίστα όπως προηγουμένως, αλλά με κόμβους που βρίσκονται μέσα στη γωνία α, όπως αυτή ορίζεται στο PFR.
m_events: ουρά με τα events που περιμένουν για εξυπηρέτηση. Όταν εκτελείται το βήμα του πρωτοκόλλου, κοιτάμε σε αυτήν την ουρά αν υπάρχει event που περιμένει εξυπηρέτηση.
m_info:
m_info_cache:
m_totTrans, m_totRcvs: συνολικές μεταδόσεις και λήψεις μηνυμάτων από τον συγκεκριμένο κόμβο.
m_totParticles: συνολικός αριθμός κόμβων στο δίκτυο.
m_waiting, m_participated: χρησιμοποιούνται στο PFR.
m_network: δείκτης στο αντικείμενο Network, το οποίο είναι μια αφαίρεση για διαδικασίες δικτύου (π.χ αν θέλουμε να κάνουμε broadcast).

4.3.11 Incl/Run.h

΄Ενα αντικείμενο Run αντιπροσωπεύει ένα τρέξιμο του simulator, το οποίο μπορεί να αποτελείται από πολλούς γύρους, και χρησιμεύει για να αποθηκεύουμε στατιστικά στοιχεία για το τρέξιμο αυτό. Χρησιμοποιώντας τα στοιχεία από πολλά τέτοια Run, βγάζουμε τα στατιστικά στοιχεία που μας ενδιαφέρουν.

```
class Run {

public:
  Run(int isFail = 1, long rounds = 0,
    long hops = 0, long backtracks = 0,
    long totParticipants = 0,
    long totTrans = 0, long totRcvs = 0,
    double d = 0.0, double dx = 0.0, double dy = 0.0) {

    m_failure = isFail;

    m_rounds = rounds;
    m_hops = hops;
    m_backtracks = backtracks;
    m_participants = totParticipants;
    m_trans = totTrans;
    m_rcvs = totRcvs;

    m_d = d;
    m_dx = dx;
    m_dy = dy;
  }

private:
  long   m_rounds;
  long   m_hops, m_backtracks;
  long   m_participants;
  long   m_failure;
  long   m_trans, m_rcvs;
  double  m_d, m_dx, m_dy;
};
```

m_ rounds: ο αριθμός γύρων του Run.
m_participants: ο αριθμός κόμβων που συμμετείχαν στη διάδοση πληροφορίας προς το βασικό σταθμό.
m_failure: περιπτώσεις όπου ένας κόμβος δεν προώθησε μήνυμα.
m_trans και m_rcvs: σύνολο μεταδόσεων και λήψεων μηνυμάτων από όλους τους κόμβους του δικτύου.

4.3.12 source/Event.cpp

Ορίζουμε απλά τις συναρτήσεις κατασκευής και κατάργησης για την τάξη, και συναρτήσεις που επιστρέφουν την τιμή για τα protected μέλη.

4.3.13 source/Execution.cpp

Παραθέτουμε το αρχείο αυτό γιατί είναι σημαντικό. Όταν δημιουργείται ένα αντικείμενο Execution, γίνεται η αρχικοποίηση του δικτύου και οι κόμβοι τοποθετούνται στο πεδίο με κάποια κατανομή (makeSetup). Επίσης, δημιουργείται μια λίστα σε κάθε κόμβο με τους γείτονές του (makeNeighbourhoods).

```
Execution::Execution(short setup, short controlType,
        short particleType, long totPoints,
        double X, double Y,
        long periodA, long periodS,
        double angleMax, double R, double deltaR,
        double lamda)
{

   m_X = X;
   m_Y = Y;
   m_round = 0;
   m_TotalEvents = 0;

   m_controlType = controlType;

   makeSetup(setup, particleType, totPoints, m_particles, m_X, m_Y, periodA, periodS,
lamda, controlType);

   makeNeighbourhoods(controlType, m_X, m_Y, angleMax, R, deltaR, m_particles);
}
```

triggerSource: Με τη συνάρτηση αυτή δημιουργούμε ένα event σε έναν κόμβο.
getResults: Εδώ μετράμε τις συνολικές εκπομπές και λήψεις, καθώς και τον αριθμό των κόμβων που έχουν συμμετάσχει με κάποιο τρόπο. Ακόμα, ελέγχουμε αν υπάρχει μήνυμα στο δίκτυο, το οποίο δεν έχει φτάσει ακόμα στο βασικό σταθμό. Αν υπάρχει τέτοιο μήνυμα, βρίσκουμε το particle που είναι πιο κοντά στο βασικό σταθμό και παρέλαβε το μήνυμα.

4.3.14 source/Info.cpp

Εδώ πάλι έχουμε τους αναμενόμενους ορισμούς συναρτήσεων και όχι κάτι ιδιαίτερο.

4.3.15 source/Message.cpp

Και εδώ δεν έχουμε κάτι ιδιαίτερο. Στην doTransmit(), όταν μεταδίδεται ένα μήνυμα, αυξάνεται ο αριθμός των hops που έχει διανύσει η πληροφορία (Info), την οποία μεταφέρει το μήνυμα.

4.3.16 source/Particle.cpp

Θα δούμε τώρα κάποιες από τις σημαντικότερες συναρτήσεις που ορίζονται εδώ. Μερικές απλά καλούν συναρτήσεις-μέλη σε άλλα αντικείμενα, αλλά είναι

χρήσιμο να ξέρουμε τι κάνουν.

getDegree: επιστρέφει το πλήθος των γειτόνων του κόμβου.
getDegreeAngle: επιστρέφει το πλήθος των γειτόνων που βρίσκονται στη λίστα m_neighbors_angle.
setNeighbour(particle* , bool): Αν η boolean μεταβλητή είναι 1, ο κόμβος εισάγεται στη λίστα με τους γείτονες m_neighbors_angle, αλλιώς εισάγεται μόνο στη λίστα m_neighbors.
triggerEvent: καλεί την m_generateSenseEvent, για να δημιουργήσει ένα sense event στο συγκεκριμένο κόμβο.
doSetup: Χρησιμοποιείται για το PFR, έτσι ώστε στην αρχή της λειτουργίας του δίκτυου ο κομβος να κοιμηθεί σε τυχαία χρονική στιγμή μέσα σε ένα διάστημα.
doProtocolStep: Καλείται στην αρχή κάθε γύρου για κάθε κόμβο. Αποφασίζεται αν θα γίνει η εναλλαγή ανάμεσα στις καταστάσεις AWAKE,SLEEP, DEAD με την συνάρτηση doSleepAwake. Αν ο κομβος είναι AWAKE για τον επόμενο γύρο, καλείται η doProcessEvent για να επεξεργαστούμε τυχόν event και στη συνέχεια η doExecute, που είναι ουσιαστικά το βήμα του πρωτοκόλλου.
doSleepAwake: εκτελεί την εναλλαγή μεταξύ καταστάσεων. Υπάρχει η μεταβλητή m_clock που δείχνει πόσοι γύροι απομένουν μέχρι την επόμενη εναλλαγή κατάστασης. Εφόσον τα χρονικά διαστήματα SLEEP και AWAKE είναι σταθερά, απλά θέτουμε την m_clock στην αντίστοιχη τιμή και τη μειώνουμε σε κάθε γύρο.
doProcessEvent: Τα events προς επεξεργασία αποθηκεύονται σε μια ουρά, την m_queue, και από αυτήν σε κάθε γύρο επεξεργαζόμαστε ένα-ένα τα events. Αν έχουμε sense event τότε καλούμε την handleSenseEvent, αλλιώς την handleMessageEvent(thisEvent).
broadcast, broadcastAngle, transmitMessage, transmitMessageAngle: Καλούν τις ομώνυμες συναρτήσεις της τάξης Network.

4.3.17 Prot/protocols.h

Εδώ γίνεται ο ορισμός της newParticle, την οποία αναφέραμε πριν, και η οποία είναι ουσιαστικά ένα switch που δημιουργεί particles ανάλογο με το πρωτόκολλο που επιλέξαμε στην εκτέλεση μας, καλώντας τις αντίστοιχες συναρτήσεις.

4.3.18 Prot/SINKParticle.h

Ορίζουμε εδώ τον βασικό σταθμό του δικτύου. Απλά ο συγκεκριμένος σταθμός λαμβάνει μηνύματα, δεν εκτελεί κάποια άλλη λειτουργία και διαθέτει πολύ μεγάλη ενέργεια.

4.3.19 Prot/TEEN/TEENParticle.h

Στο αρχείο αυτό, γίνεται ο ορισμός του κόμβου που εκτελεί το πρωτόκολλο TEEN. Οι συναρτήσεις για το πρωτόκολλο θα παρουσιαστούν στο επόμενο αρχείο.

```
class TEENParticle : public Particle {

public:

   TEENParticle(long id, double x, double y,
         long periodA, long periodS,
         long totParticles);

   void    doSetup(point* p);

   int     doTEENSetup(double, int, double, idList&);
   void    addClusterHead(color cl);
   void    actualAddClusterHead(Particle *s, color cl);
   void    doClusterSetup();
   void    addNextLevelClusterHead(particleList& cL);
   void    actualAddNextLevelClusterHead(Particle *s, short s_type);
   void    showClusterHead();
   void    informClusterHead();
   void    broadcastTDMASchedule();
   void    receiveTDMASchedule();

   color   myColor;

protected:
   short   TEEN_type;
   bool    has_been_elected;
   short   rounds_left;
   Particle* myClusterHead;
   p_queue<double, Particle* > clusterHeadDistances;

   list<Particle *> clusterHeadsList;
   void doTEENProcessEvent(long, Particle*);

   bool am_i_a_cluster(double, double);
   void doExecute();
   void handleSenseEvent(Event* thisEvent);
   void handleMessageEvent(Event* thisEvent);

};
```

myColor: ένα αντικείμενο color της LEDA. Χρησιμεύει για την απεικόνιση του κόμβου στον animator. Όλοι οι κόμβοι που ανήκουν στον ίδιο cluster, έχουν το ίδιο χρώμα.
TEEN_type: δείχνει το ιεραρχικό επίπεδο στο οποίο βρίσκεται ο κόμβος. Όσο μεγαλύτερο τόσο ψηλότερα βρίσκεται στην ιεραρχία.
has_been_elected: αν ο κόμβος είναι από cluster-head και πάνω.
rounds_left: χρήσιμο για να υπολογίζουμε πότε ένας κόμβος που έχει γίνει cluster-head σε κάποιο προηγούμενο γύρο, έχει το δικαίωμα να ξαναγίνει cluster-head.

myClusterHead: δείκτης στον κόμβο που επιλέξαμε ως cluster-head μας.
clusterHeadDistances: είναι μια ουρά στην οποία καταχωρούμε δείκτες σε όλα τα particles, τα οποία έστειλαν διαφήμιση ότι είναι cluster-head, και την απόστασή τους από τον κόμβο μας. Χρησιμεύει για να αποφασίσουμε ποιος cluster-head είναι πιο κοντά.
am_i_a_cluster : συνάρτηση για να εκλέγουμε cluster-heads.

4.3.20 Prot/TEEN/TEEN.cpp

Θα εξετάσουμε το αρχείο αυτό σε τρία μέρη, τα οποία είναι αρχικοποίηση, διαχείριση event και δημιουργία επιπέδων ιεραρχίας. Στο πρώτο μέρος, υπάρχει η συνάρτηση **returnCurrentColor**, η οποία περιέχει μια στατική μεταβλητή για να υπολογίζουμε το χρώμα που θα δώσουμε σε κάποιο cluster. Αυτό προφανώς, χρησιμοποιείται στην οπτικοποίηση. Κάθε φορά που κάποιος κόμβος εκλέγεται cluster-head παίρνει το χρώμα του από αυτή τη συνάρτηση.

Προχωράμε έπειτα στην **doTEENSetup**, η οποία καλείται για κάθε κόμβο όταν αρχίζει ένας καινούριος γύρος. Εδώ αποφασίζουμε αν ένας κόμβος θα γίνει cluster-head. Αν ναι, καλούμε τις συναρτήσεις που συνδέουν nodes και cluster-heads και περιγράφονται στη συνέχεια.

```
// ------------------------------------------------------------
// ---- Constructor + Setup
// ------------------------------------------------------------

int returnCurrentColor() {
   static int col=0;
   if (col<15)
      col++;
   else col = 0;

   return col;
}


TEENParticle::TEENParticle(long id, double x, double y,
       long periodA, long periodS,
       long totParticles)
   :Particle(PARTICLE_TEEN, id, x, y, periodA, periodS, totParticles)
{
   has_been_elected=0;
   rounds_left = TEEN_EPOCH_DURATION;
   clusterRadius = 0;
   clusterSize = 0;

   // Do nothing
}

void TEENParticle::doSetup(point *p) {

   // Do basic stuff
   Particle::doSetup();
```

```
}

int TEENParticle::doTEENSetup(double T, int round, double
temp_random, idList& id_L) {

   if (!this->has_been_elected && this->am_i_a_cluster( T, temp_random)) {
      this->TEEN_type++;
      this->has_been_elected = 1;
      this->rounds_left = TEEN_EPOCH_DURATION;
      //colorTEENCluster(this);
      myClusterHead = nil;
      clusterSize = 0;
      clusterRadius = 0;

      // set color of the clusterHead and then the rest of the stuff...
      int r;

      r = returnCurrentColor();

      color kwlor(r);

      myColor = kwlor;

      colorTEENCluster(this, this->myColor);
      m_network->addNode2Cluster(this, this->myColor);

      this->addClusterHead(myColor);
      clusterHeadDistances.clear();
      return 1;  // CASE CLUSTER_HEAD
   }

   else if (this->rounds_left==TEEN_EPOCH_DURATION){
      this->rounds_left--;
      uncolorTEENCluster(this);
   }
   else if (this->rounds_left>1){
      this->rounds_left--;
   }

   else if (this->rounds_left==1){
      this->rounds_left=0;
      this->has_been_elected=0;
   }

   return 0;   // CASE SIMPLE NODE

}
```

Στη συνέχεια έχουμε συναρτήσεις για τα βήματα του πρωτοκόλλου μετα την αρχικοποίηση (σχηματισμος clusters). Έτσι, έχουμε τη **handleSenseEvent** για την περίπτωση που έχουμε ένα sense event να περιμένει στην ουρά με τα events ενός κόμβου, και την **handleMessageEvent** για την περίπτωση που

λαβάμε μήνυμα από κάποιον άλλο κόμβο.

```
// -------------------------------------------------------------
// ---- Execute Function
// -------------------------------------------------------------

void TEENParticle::doExecute() {
   // Do nothing
}


void TEENParticle::handleSenseEvent(Event* thisEvent) {
   // SenseEvent Handler

   // Construct info for new Event
   Info *info = new Info(getID(), 0, 0, 0, thisEvent->getTimeStamp(),
point(this->xcoord(),this->ycoord()));

   Message*   m_message;
   m_message = new Message(MESSAGE_INFO, this, info);

   // Transmit message to all neighbours

   if (clusterHeadDistances.size())
      m_network->transmitMessage(this, m_message, myClusterHead);
   else
      m_network->transmitMessage(this, m_message, m_RealSink);

   delete(m_message);
   m_participated = true;

   // giati me to pou ginetai triggered mpainei se SENSING STATE
   setState(ALWAYS_ON);
}

void TEENParticle::handleMessageEvent(Event* thisEvent) {
   Message* msg = thisEvent->getMessage();

   if (clusterHeadDistances.size())
      m_network->transmitMessage(this, msg, myClusterHead);
   else
      m_network->transmitMessage(this, msg, m_RealSink);

   m_participated = true;
}}
```

Στη συνέχεια έχουμε τις συναρτήσεις, οι οποίες περιέχουν τις λεπτομέρειες για την εκλογή cluster-heads. Η **doClusterSetup** απλά θέτει ένα κόμβο σε επίπεδο ιεραρχίας 0 και αδειάζει την ουρά με τις διαφημίσεις των cluster-heads. Η **am_i_a_cluster** είναι το σημείο που ένας κόμβος υπολογίζει μια τυχαία τιμή για το αν μπορεί να προχωρήσει πιο πάνω στην ιεραρχία. Ακολουθούν μετά συναρτήσεις που ορίζουν πως θα γίνει η επιλογή cluster-head από τους κόμβους του δικτύου, πως θα γίνει η εκλογή για τα ανώτερα από cluster-head επίπεδα

ιεραρχίας, μετάδοση TDMA schedule για μεταδόσεις, κτλ.

Πιο συγκεκριμένα, οι **addClusterHead** και **actualAddClusterHead** εκλέγουν cluster-heads. Οι **addNextLevelClusterHead** και **actualAddNextLevelClusterHead** είναι για τα επόμενα επίπεδα ιεραρχίας, γιατί η διαδικασία είναι κάπως διαφορετική. Η **informClusterHead** είναι για να υπολογίσουμε την ενέργεια που ξοδεύει κάθε κόμβος, όταν στέλνει μήνυμα στον cluster-head που θέλει να ανήκει, ενώ η **broadcastTDMASchedule** για την ενέργεια που ξοδεύει ο κόμβος όταν εκπέμπει το πρόγραμμα μεταδόσεων μέσα στον cluster.

```
// ------------------------------------------------------------
// ---- Cluster-related functions
// ------------------------------------------------------------

void TEENParticle::doClusterSetup() {
   clusterHeadDistances.clear();
   TEEN_type = 0 ;
}

bool TEENParticle::am_i_a_cluster(double T, double t_random) {
   if ( T > t_random) return true;
   else return false;

}

void TEENParticle::addClusterHead(color cl) {

   // Transmit message to all neighbours
   list_item item;

   forall_items(item, *(this->m_neighbours)) {
      Particle *p = this->m_neighbours->inf(item);
      p->actualAddClusterHead(this, cl);

   }

}

void TEENParticle::actualAddClusterHead(Particle * s, color cl) {
   if (TEEN_type == 0 ) {
   // an einai aplo node pros8etei to cluster -head sto p_queue tou kai to sygkrinei me ta
ypoloipa.
   // epilegei ws diko toy cluster ayto pou apexei th mikroterh apostash apo ton eayto toy.

      double temp_dist;
      temp_dist = this->distance(*s);
      this->clusterHeadDistances.insert(temp_dist, s);

      pq_item min_distance_c_head;

      min_distance_c_head=this>clusterHeadDistances.find_min();

      this->myClusterHead = this->clusterHeadDistances.inf(min_distance_c_head); //
```

```
set the cluster head

      if (s == this->clusterHeadDistances.inf(min_distance_c_head)) {
         this->myColor = cl; // same color with cluster head
         m_network->addNode2Cluster(this, this->myColor); // change the node in
GraphWin
      }
   }

}

void TEENParticle::addNextLevelClusterHead(particleList& cL) {

   // Transmit message to other cluster Heads in the same level
   list_item item;
   TEEN_type++;
   myClusterHead = nil;


   forall_items(item, cL) {
      Particle* p = cL.inf(item);
      if ( p != this)
         p->actualAddNextLevelClusterHead(this, this->TEEN_type);
   }

   m_network->colorSuperClusterHead(this, this->myColor);
   clusterHeadDistances.clear();


}

void TEENParticle::actualAddNextLevelClusterHead(Particle* s,
short s_type) {
   if (this->TEEN_type < s_type ) { // an einai se katwtero epipedo apo to s den an to
einai o swstos cluster Head tou
      double temp_dist;
      temp_dist = this->distance(*s);
      this->clusterHeadDistances.insert(temp_dist, s);

      /* an einai aplo node
       * pros8etei to cluster-head sto p_queue tou
       * kai to sygkrinei me ta ypoloipa.
       * epilegei ws diko toy cluster ayto pou apexei
       * th mikroterh apostash apo ton eayto toy.
       */
      pq_item min_distance_c_head;
      min_distance_c_head = this->clusterHeadDistances.find_min();

      this->myClusterHead = this->clusterHeadDistances.inf(min_distance_c_head);
   }
}

void   TEENParticle::showClusterHead() {
   m_network->sh_C_Head(this, this->myClusterHead);
```

```
}


void TEENParticle::informClusterHead() {
   if ( this->TEEN_type ==0 && this->myClusterHead != nil) {
      double d = this->distance(*(this->myClusterHead));
      this->spendEnergy((ENERGY_ELEC*BITS_FOR_INFORMING_CLUSTER_HEAD) +
            (ENERGY_AMP * BITS_FOR_INFORMING_CLUSTER_HEAD)*pow(d,2));
      this->incTrns();
      (this->myClusterHead)->
          spendEnergy(ENERGY_ELEC*BITS_FOR_INFORMING_CLUSTER_HEAD);
      (this->myClusterHead)->incRcvs();

      if ( d > this->myClusterHead->clusterRadius){
      // gia na ypologizoume to range gia to broadcast tou TDMA schedule

         this->myClusterHead->clusterRadius = d;
         this->myClusterHead->clusterSize++;
      }
   }
}

void TEENParticle::broadcastTDMASchedule() {
   this->spendEnergy( ENERGY_ELEC *BITS_FOR_TDMA_SCHED*clusterSize +
ENERGY_AMP *BITS_FOR_TDMA_SCHED*clusterSize*pow(this->clusterRadius, 2));
   this->incTrns();
}

void TEENParticle::receiveTDMASchedule() {
   if ( this->TEEN_type ==0 && this->myClusterHead != nil) {
      double cSize = this->myClusterHead->clusterSize;
      this->incRcvs();
      this->spendEnergy( ENERGY_ELEC * 32 * cSize );
   }
}
```

4.3.21 Animator/Animation.h

Το animation είναι ένα αντικείμενο που κληρονομεί από την τάξη Execution, για την περίπτωση του animator. Αφού δημιουργηθεί ένα αντικείμενο animation, το χρησιμοποιούμε για να αρχίσουμε να τρέχουμε γύρους του animator. Η αντίστοιχη τάξη για τον simulator είναι η Experiment. Η συνάρτηση **execute** είναι πολύ σημαντική και ουσιαστικά ξεκινά τη λειτουργία του animator.

```
class Animation : public Execution {

public:
   Animation(short setup, short controlType,
        short particleType, long totPoints,
        double X, double Y,
        long periodA, long periodS,
        double angleMax,double R, double deltaR,
```

```
        double lamda);

    ´Animation();

    void    execute(d_array<int,set<long>> &cronojon_IDs_n_rounds);
    void    triggerSource(int id = 0);

protected:
    Network* m_network;
    short    pType;
};
```

4.3.22 Animator/Animation.cpp

Στο αρχείο αυτό περιέχεται ο ορισμός της πολυ βασικής για τη λειτουργία του animator συνάρτησης, της execute. Στη συνάρτηση αυτή περιέχεται ένας βρόχος, ο οποίος εκτελεί τη λειτουργία του δικτύου σε γύρους.

Αρχικά γίνονται οι απαραίτητες αρχικοποίησεις (μέσω των doSetup και setNetwork) και μετά αρχίζει η λειτουργία της οπτικοποίησης μέχρι τη χρονική στιγμή που θέλουμε να σταματήσουμε. Γίνεται ένας έλεγχος αν υπάρχουν events, τα οποία ορίσαμε εμείς, που πρέπει να μοιραστούν στους κόμβους του δικτύου.

Στη συνέχεια, αν το πρωτόκολλο που "τρέχει" στο δίκτυο είναι το TEEN, και αν είμαστε σε γύρο με αριθμό πολλαπλάσιο του 24, γίνεται η φάση αρχικοποίησης του TEEN, με την εκλογή cluster-heads και τη δημιουργία ιεραρχικών επιπέδων δρομολόγησης. Πολλές από αυτές τις λειτουργίες επιλέξαμε να γίνονται μέσα σε αυτήν τη συνάρτηση, για να απλοποιήσουμε και να επιταχύνουμε την εκτέλεση του εξομοιωτή. ΄Ενας πιο σωστός σχεδιασμός θα ήταν να γίνοταν μέσα σε κάποιες άλλες συνάρτησεις, αλλά υπήρχε και πίεση χρόνου.

Στη συνέχεια, εξετάζουμε κάθε particle ξεχωριστά και εξετάζουμε αν υπάρχει event στην ουρά του κόμβου και εκτελούμε τα βήματα του πρωτοκόλλου. Σε κάθε γύρο κάνουμε και μια μετάθεση στη σειρά με την οποία εξετάζουμε τους κόμβους.

4.3.23 Animator/animator.h

Εδώ ορίζεται η doAnimation_multiple, η οποία πρώτα δημιουργεί ένα αντικείμενο Animation και έπειτα καλεί τη συνάρτηση execute της τάξης Animation.

```
extern GraphWin gw;
extern bool DRAW_CIRCLE;
extern point* the_circle_center;
extern int RANGE_COUNTER;

void doAnimation_multiple(short setup, short control,
        short particleType, long totPoints,
        double X, double Y,
        long periodA, long periodS,
        double angleMax,double R, double deltaR,
```

```
        double lamda, d_array<int,set<long> > &cronojon_IDs_n_rounds)

  Animation* thisAnim = new Animation(setup, control, particleType,
        totPoints, X, Y, periodA, periodS,
        angleMax, R, deltaR, lamda);

  gw.wait("Constructed the sensor field. Please press done to continue");
  thisAnim->execute(cronojon_IDs_n_rounds);
```

4.3.24 Animator/animator.cpp

Εδώ περιέχεται η main συνάρτηση του animator, οπότε από εδώ αρχίζει η εκτέλεση του animator. Βασικά, δημιουργείται ένα αντικείμενο GraphWin που είναι το παράθυρο που βλέπει ο χρήστης όταν ξεκινήσει την εφαρμογή. Επίσης, εδώ ορίζεται και το interface με το χρήστη, δηλαδή panels, buttons, συναρτήσεις που θα κληθούν, κτλ. Εδώ δηλαδή είναι το σημείο εκκίνησης της εφαρμογής, και περιέχεται η συνάρτηση doAnimation, η οποία δημιουργεί ένα αντικείμενο animation και καλεί έπειτα την συνάρτηση execute της τάξης animation για να αρχίσει η εκτέλεση γύρων.

Υπάρχουν επίσης και δηλώσεις κάποιων εξωτερικών μεταβλητών σε αυτό το αρχείο. Η πιο σημαντική είναι η **gw**, η οποία είναι το αντικείμενο GraphWin που χρησιμοποιούμε σε όλο τον animator.

Ο αναγνώστης που ενδιαφέρεται μπορεί να διαβάσει το manual της LEDA, προκειμένου να κατανοήσει πως ακριβώς ορίζονται τα αντικείμενα GraphWin και τα Panels.

4.3.25 Animator/Network_Anim.cpp

Το αρχείο αυτό περιέχει την υλοποίηση συναρτήσεων για την τάξη Network, οι οποίες χρησιμοποιούνται στον Animator. Προφανώς, ένα μεγάλο μέρος του κώδικα αναλώνεται στην απεικόνιση του δικτύου, στις αλλαγές της τοπολογίας και τα βήματα του πρωτοκόλλου. Σημαντικές συναρτήσεις είναι οι:

setClock: Θέτουμε το clock του Network (αριθμός γύρων).
broadcastMessage: Στέλνουμε ένα μήνυμα m σε καθένα από τους γείτονες του κόμβου. Καλείται η m->doTransmit() για τη μετάδοση του μηνύματος και στη συνέχεια η receiveMessage σε καθένα από τους κόμβους αυτούς.
broadcastMessageAngle: τα ίδια με την προηγούμενη συνάρτηση, απλά οι γείτονες στους οποίους στέλνεται το μήνυμα περιέχονται σε κάποια ορισμένη γωνία.
generateSenseEvent: Δημιουργούμε ένα sense event και το ενθέτουμε στην ουρά κάποιου κόμβου. Του δίνουμε ως timestamp έναν αριθμό ανάμεσα στον τρέχοντα και τον επόμενο γύρο (getNextTimeStamp).
addNode2Cluster, sh_head: Χρήσιμες για να χρωματίζουμε κόμβους που ανήκουν στον ίδιο cluster με το ίδιο χρώμα και να δείχνουμε την ιεραρχία των επιπέδων δρομολόγησης.

4.3.26 Simulator/Experiment.h

Το αντίστοιχο της τάξης animation για τον simulator. Υπάρχουν κάποια μέλη παραπάνω, γιατί στο simulator κρατάμε εγγραφές από τα αποτελέσματα ενός γύρου, αφού μας ενδιαφέρουν τα στατιστικά στοιχεία (στον animator αυτό δε μας ενδιαφέρει, απλά να δείξουμε τη λειτουργία των πρωτοκόλλων.

m_totRounds: ο συνολικός αριθμός γύρων που εκτελέστηκαν ως τώρα.
m_printRate: κάθε πόσους γύρους υπολογίζουμε στατιστικά στοιχεία.
m_cType: ο τύπος βασικού σταθμού (sink, Wall).
m_pType: ο τύπος κόμβου (PFRParticle, κτλ).
m_R: η ακτίνα μετάδοσης ενός κόμβου.

4.3.27 Simulator/Experiment.cpp

Κατά τα γνωστά από το Animation.cpp, έχουμε ένα βρόχο που εκτελεί τη λειτουργία του δικτύου σε γύρους. Η διαφορά είναι πως εδώ δεν έχουμε κώδικα για την οπτικοποίηση του δικτύου, και κάθε m_printRate γύρους εξάγουμε κάποια στοιχεία για τη λειτουργία του δικτύου. Μερικά από αυτά τα στοιχεία είναι:

alivePart: το πλήθος των κόμβων που βρίσκονται ακόμα σε λειτουργία, δηλαδή έχουν αποθέματα ενέργειας.
totEnergy: το άθροισμα των αποθεμάτων ενέργειας των κόμβων στο δίκτυο.
avgEnergy: η μέση ένεργεια στο δίκτυο.

4.3.28 Simulator/simulator.h

Η τάξη simulator είναι το αντίστοιχο της τάξης animator, δηλαδή χρησιμεύει για να ξεκινήσει την όλη διαδικασία μετρήσεων. Οι σημαντικές συναρτήσεις που περιέχονται εδώ είναι:

set_required_rounds: Η συνάρτηση αυτή ελέγχει αν οι γύροι που δίνουμε ως όρισμα στη γραμμή εντολών, φτάνουν για να εκτελεστούν όλα τα events που δημιουργούμε. Αν όχι, ο simulator τερματίζει με ένα μήνυμα που λέει πόσοι γύροι τουλάχιστον θα χρειαστούν για να γίνει κανονική εκτέλεση.
schedule_cronojon: Η συνάρτηση αυτή καλείται για να κατανείμει events σε γύρους εκτέλεσης.
doExperiment_multiple: Η συνάρτηση αυτή αρχίζει τη διαδικασία των πειραμάτων. Για κάθε τρέξιμο του simulator καλεί την προηγούμενη συνάρτηση για να δημιουργήσει ένα πρόγραμμα με τα events προς εκτέλεση και μετά καλεί την execute για να γίνουν τα πειράματα.

4.3.29 Simulator/simulator.cpp

Εδώ βρίσκεται η main συνάρτηση του simulator, οπότε απλά έχουμε έλεγχο ορισμάτων από τη γραμμή εντολών, και στη συνέχεια καλείται η doExperiment_multiple για να αρχίσει τη διαδικασία των πειραμάτων.

4.3.30 Simulator/Network_Exp.cpp

Οι συναρτήσεις που ορίζονται εδώ, είναι για την υλοποίηση της τάξης Network και χρησιμοποιούνται από το simulator. Είναι κατά βάση οι ίδιες που περιέχονται και στο αρχείο Network_Anim.cpp, αλλά χωρίς οπτικοποίηση.

4.4 Εισαγωγή στη χρήση του simDust

Θα περάσουμε τώρα σε κάτι πιο χειροπιαστό και θα δούμε πως μπορούμε να χρησιμοποιήσουμε τα δύο μέρη του εξομοιωτή simDust, τον Animator και το Simulator. Αυτό θα γινει γίνει μέσω παραδειγμάτων και screenshots, τα οποία θα είναι κυρίως γύρω από την εξομοίωση του πρωτοκόλλου TEEN.

4.4.1 Χρήση του animator

Ο animator είναι ένα πρόγραμμα με τη βοήθεια του οποίου μπορούμε να δημιουργήσουμε ένα δίκτυο έξυπνης σκόνης σε ένα δισδιάστατο πεδίο, στο οποίο μπορούμε να καθορίσουμε πολλές παραμέτρους και ταυτόχρονα να βλέπουμε στην οθόνη μας τη λειτουργία του δικτύου ανά πάσα στιγμή. Αρχικά να σημειώσουμε ότι το εκτελέσιμο αρχείο animator, παράγεται εισάγωντας στη γραμμή εντολών (ενώ βρισκόμαστε στον κατάλογο simDust):

```
> make animator
```

Το εκτελέσιμο αρχείο περιέχεται στον κατάλογο simdust/Out και το καλούμε δίνοντας στη γραμμή εντολών ./animator, οπότε εμφανίζεται μπροστά μας το παρακάτω παράθυρο:

Σχήμα 4.2: Η εισαγωγική οθόνη του animator

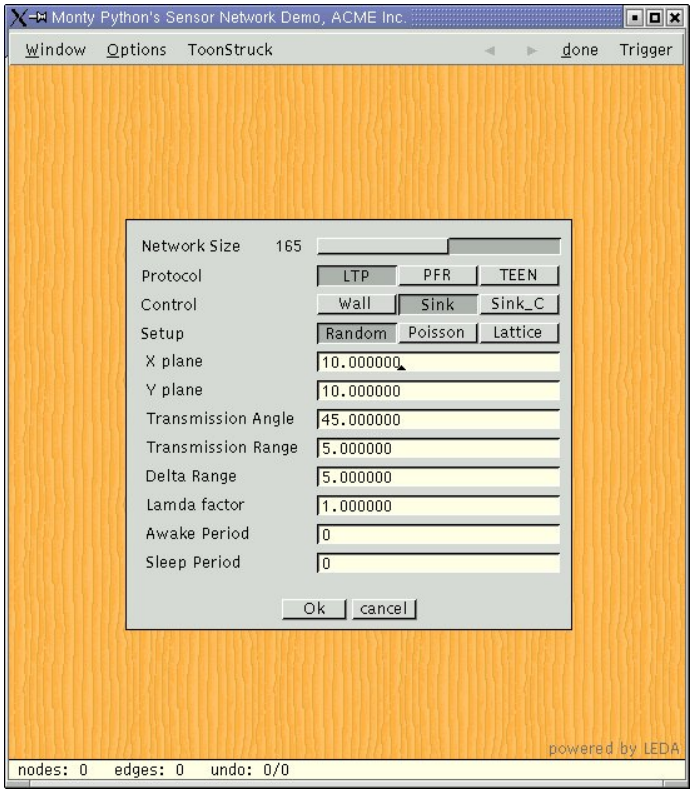
Παρουσιάζονται τέσσερις επιλογές στο panel του παραθύρου αυτού, οι οποίες από αριστερά προς τα δεξιά είναι:

- **Run:** Πατώντας αυτό το κουμπί αρχίζει η λειτουργία του δικτύου.
- **Setup:** Από εδώ μπορούμε να καθορίσουμε τις παραμέτρους του δικτύου.
- **Credits:** Λίγη δόξα και για μας!
- **Exit:** Τερματισμός του animator.

Ενδιαφέρον παρουσιάζει το setup, στο οποίο έχουμε αρκετές επιλογές, όπως φαίνεται στο σχήμα 4.3, και οι οποίες είναι:

- **Network Size:** το πλήθος κόμβων του δικτύου. Μπορεί να κυμαίνεται από 10 έως 300.
- **Protocol:** το πρωτόκολλο με το οποίο επικοινωνούν οι κόμβοι του δικτύου μεταξύ τους και με τον βασικό σταθμό. Οι δυνατές επιλογές είναι LTP, PFR και TEEN.
- **Control:** ο τύπος βασικού σταθμού. Οι δυνατές επιλογές είναι Wall(τείχος), Sink(βασικός σταθμός) και Sink_C(βασικός σταθμός αλλά μέσα στο δίκτυο, σε πειραματικό στάδιο ακόμα).
- **Setup:** ο τύπος κατανομής των κόμβων του δικτύου στο πεδίο. Οι δυνατές επιλογές είναι Random(τυχαία κατανομή) και Lattice (διάταξη πίνακα). Η κατανομή Poisson δεν έχει υλοποιηθεί ακόμα.
- **X, Y plane:** οι διαστάσεις του πεδίου.
- **transmission angle:** μία σταθερή γωνία με την οποία γίνεται θεωρούμε ότι γίνεται η μετάδοση υπο γωνία. Χρήσιμο στο PFR και στο LTP, ενώ στο TEEN δεν κάνουμε μετάδοση υπό γωνία.
- **Delta Range και Lamda Factor:** χρησιμοποιούνται στην αρχική έκδοση του PFR.
- **Awake και Sleep period:** η διάρκεια των περιόδων ύπνου και κανονικής λειτουργίας των κόμβων του δικτύου. Αν δώσουμε τιμές 0 και 0, οι κόμβοι λειτουργούν χωρίς διακοπή.

Αφού επιλέξουμε να δουλέψουμε με το πρωτόκολλο TEEN από το setup, πατάμε το κουμπί Run για να αρχίσει η λειτουργία του δικτύου. Στον πρώτο γύρο γίνεται η αρχικοποίηση του δικτύου, και δημιουργούνται cluster και επίπεδα ιεραρχίας. Στο σχήμα 4.4, φαίνεται ένα δίκτυο με 165 κόμβους. Μετά την αρχικοποίηση έχουν δημιουργηθεί τρία επίπεδα ιεραρχίας, απλοί κόμβοι,

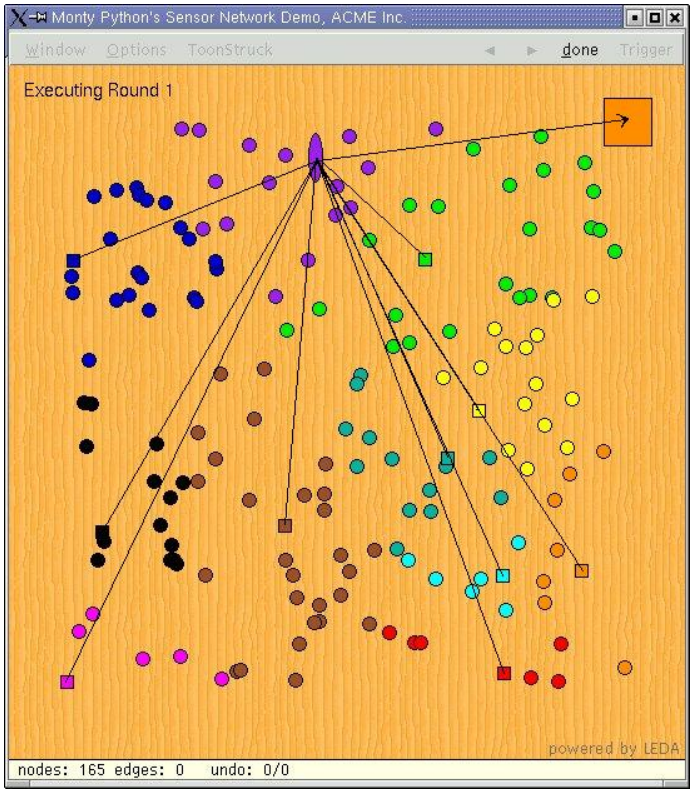


Σχήμα 4.3: Οι επιλογές στο setup panel του animator

cluster-heads και super-cluster-heads. Οι απλοί κόμβοι απεικονίζονται ως μικροί κύκλοι, των οποίων το χρώμα εξαρτάται από τον cluster στον οποίο ανήκουν. Οι cluster-heads εικονίζονται ως τετράγωνα στο μέγεθος των απλών κόμβων. Οι super-cluster-heads εικονίζονται ως ελλείψεις, αρκετά μεγαλύτερες από τους cluster-heads. Τέλος, ο βασικός σταθμός φαίνεται στο δεξί άνω άκρο του δικτύου, σαν ένα μεγάλο πορτοκαλί τετράγωνο. Προφανώς, όλα τα μηνύματα απευθύνονται τελικά στο βασικό σταθμό.

Αφού τελειώσει η αρχικοποίηση, δείχνουμε την ιεραρχία των επιπέδων με βέλη που ξεκινούν από τους cluster-heads και πηγαίνουν στους super-cluster-heads, και συνεχίζουμε στα ανώτερα επίπεδα. Στο δίκτυο της εικόνας, επειδή έχει μικρό μέγεθος, έχουμε ένα μόνο super-cluster-head, πάνω στον οποίο πέφτουν όλοι οι cluster-heads, και ο οποίος μεταδίδει κατευθείαν στο base station. Ακόμα και οι κόμβοι που βρίσκονται δίπλα στο βασικό σταθμό, μεταδίδουν μέσω του super-cluster-head.

Προχωράμε τώρα στην επόμενη εικόνα(σχήμα 4.5). Μετά από 24 γύρους, το δίκτυο θα ξαναμπεί σε φάση αρχικοποίησης και θα σχηματιστούν εκ νέου clusters. Το αποτέλεσμα τώρα είναι να δημιουργηθούν μόνο cluster-heads και κανένας super-cluster-head, οπότε πέφτουν όλοι κατευθείαν στο βασικό σταθμό.

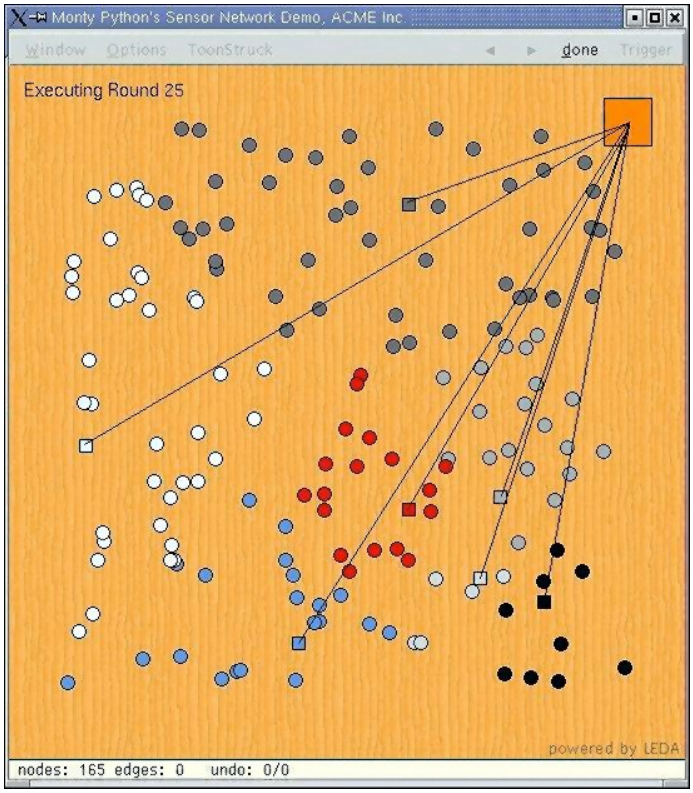


Σχήμα 4.4: Δίκτυο με δύο επίπεδα ιεραρχίας

Βλέπουμε δηλαδή ότι οι cluster-heads που δημιουργούνται κάθε φορά είναι πολύ διαφορετικοί μεταξύ τους και τελικά μπορεί η κατανομή των μεταδόσεων να μην είναι και η πιο οικονομική, από άποψη κατανάλωσης ενέργειας. Οι clusters θα ξανασχηματιστούν μετά από άλλους 24 γύρους (ο αριθμός αυτός είναι fixed στην υλοποίησή μας).

Προχωράμε τώρα σε ένα άλλο δίκτυο, με 300 κόμβους αυτή τη φορά (σχήμα 4.6). Αυτό το πλήθος κόμβων είναι αρκετό για να δημιουργηθούν 4 επίπεδα ιεραρχίας, με έναν hyper-cluster-head, στον οποίο πέφτουν όλοι οι super-cluster-heads. Η ιεραρχία των επιπέδων φαίνεται πάλι από τα βέλη που ξεκινούν από τους cluster-heads προς τους super-cluster-heads. Τα βέλη προς το ανώτερο επίπεδο είναι λίγο πιο παχιά, και καταλήγουν στον ίδιο hyper-cluster-head. ΄Ενα μήνυμα που ξεκινά από έναν απλό κόμβο θα μεταδωθεί πρώτα στον αντίστοιχο cluster-head, έπειτα στον super-cluster-head και μετά στον hyper-cluster-head. Κάθε τέτοια μετάδοση γίνεται σε διαφορετικό γύρο, δηλαδή δεν μπορεί ένα μήνυμα στον ίδιο γύρο να διαβεί δύο επίπεδα ιεραρχίας.

Να αναφέρουμε ότι στο ανω αριστερό άκρο της εικόνας εμφανίζονται διάφορα μηνύματα, όπως “Executing round 25”, και παρέχουν πληροφορία για την κατάσταση του δικτύου. Συνήθως, απαίτειται ο χρήστης να πατήσει το κουμπι



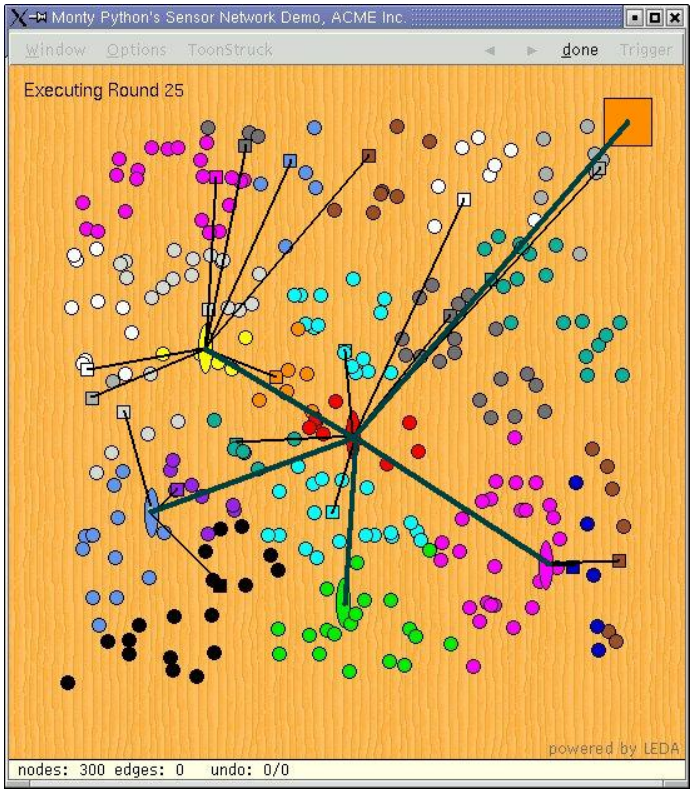
Σχήμα 4.5: Δίκτυο με ένα επίπεδο ιεραρχίας

Done, το οποίο βρίσκεται επάνω αριστερά στα menu, για να προχωρήσει η διαδικασία.

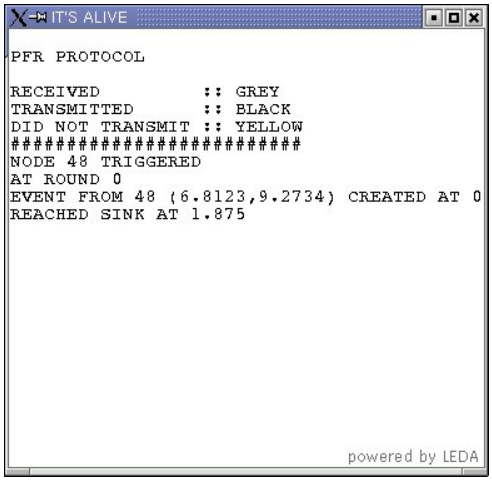
Εκτός από τα μηνύματα που εμφανίζονται στο κυρίως παράθυρο, υπάρχουν και αρκετά μηνύματα σε ένα βοηθητικό παράθυρο για να είναι πιο εύκολη η ανάγνωσή τους. Το παράθυρο αυτό χρησιμεύει για να δηλώνονται οι ακριβείς χρόνοι δημιουργίας ενός event σε κάποιο κόμβο του δικτύου, και της λήψης του από το βασικό σταθμό. Επίσης, εμφανίζονται πληροφορίες κατά τη φάση αρχικοποίησης του TEEN, όπως ποιοι κόμβοι έχουν γίνει cluster-heads, που διευκολύνουν το χρήστη στην κατανόηση της λειτουργίας των πρωτοκόλλων.

'Ενα παράδειγμα της λειτουργίας του παραθύρου αυτού φαίνεται στο σχήμα 4.7. Εκτελούμε το PFR και σε κάποια φάση(συγκεκριμένα στο γύρο 0) προκαλούμε ένα event σε έναν κόμβο του δικτύου. Εμφανίζεται το αντίστοιχο μήνυμα στο παράθυρο. Ο κομβος αντιδρά στον ίδιο γύρο και έστω ότι σε δύο βήματα το μήνυμα φτάνει στο βασικό σταθμό. Εμφανίζεται μια ειδοποίηση στο παράθυρο ότι ο βασικός κόμβος (sink) έλαβε το μήνυμα στη χρονική στιγμή 1.875, δηλαδή κάπου μεταξύ γύρου 1 και γύρου 2.

Τέλος, αναφέρουμε πως γίνεται το triggering των events. Πηγαίνουμε με το ποντίκι πάνω από έναν κόμβο του δικτύου και κάνουμε click έχοντας ταυ-



Σχήμα 4.6: Δίκτυο με τρία επίπεδα ιεραρχίας



Σχήμα 4.7: Χρήσιμες πληροφορίες για τη λειτουργία του δικτύου

τόχρονα πατημένο το πλήκτρο Ctrl στο πληκτρολόγιο. Προσέχουμε ότι ο κέρσορας αλλάζει φορά αν έχουμε πατημένο το Ctrl. Αυτόματα ο κόμβος αλλάζει μέγεθος και χρώμα για να ξεχωρίζει από τους υπόλοιπους στο δίκτυο. Η σχετική οδηγία εμφανίζεται αν πατήσουμε το κουμπί Trigger πάνω άκρη δεξιά στα menu. Το triggering μπορεί να γίνει σε οποιαδήποτε στιγμή της λειτουργίας του δικτύου.

4.4.2 Χρήση του simulator

Ας δούμε τώρα πως μπορούμε να χρησιμοποιήσουμε τον simulator. Το εκτελέσιμο αρχείο περιέχεται στον ίδιο κατάλογο με το εκτελέσιμο του animator. Δίνοντας στη γραμμή εντολών `./simulator`, εμφανίζονται στην οθόνη μας οδηγίες, σχετικές με το πώς καλούμε το πρόγραμμα. Οι παράμετροι αυτές που δίνουμε στο πρόγραμμα από τη γραμμή εντολών παρουσιάζονται στον ακόλουθο πίνακα, με τη σειρά που πρέπει να δωθούν.

Πρωτόκολλο	Το πρωτόκολλο που τρέχουν οι κόμβοι του δικτύου (LTP, PFR ή TEEN)
Κατανομή	Κατανομή των κόμβων του δικτύου στο πεδίο, τυχαία ομοιόμορφη (R) ή πίνακα (L)
Κέντρο ελέγχου	Τύπος βασικού σταθμού, βασικός σταθμός (S) ή τείχος (W)
Πλήθος κόμβων	Το πλήθος των κόμβων του δικτύου
Διάρκεια εξομοίωσης	Πλήθος γύρων εξομοίωσης
Επαναλήψεις	Πλήθος επαναλήψεων πειραμάτος
Διάσταση X	Η διάσταση X του πεδίου
Διάσταση Y	Η διάσταση Y του πεδίου
Ακτίνα μετάδοσης	Εμβέλεια ενός κόμβου
ΔR	Χρησιμοποιείται στο PFR, συνήθως το θέτουμε ίσο με την ακτίνα
Γωνία α	Η γωνία μετάδοσης
Awake period	Η διάρκεια περιόδου κανονικής λειτουργίας
Sleep period	Η διάρκεια περιόδου "ύπνου"
Παράμετρος γεγονότων X	Προαιρετική παράμετρος. Αν τη θέσουμε, είναι το πλήθος των γεγονότων στη διάρκεια της εξομοίωσης
Παράμετρος γεγονότων Y	Προαιρετική παράμετρος. Αν έχει τεθεί η προηγούμενη παράμετρος, αν δώσουμε τιμή <1 είναι η πιθανότητα να συμβεί γεγονός σε οποιονδήποτε γύρο, αλλιώς πλήθος γεγονότων στη σειρά

Πίνακας 4.1: Παράμετροι γραμμής εντολών για τον simulator

΄Εστω τώρα, ότι θέλουμε να τρέξουμε μια εξομοίωση ενός δικτύου έξυπνης σκόνης, στο οποίο οι κόμβοι θα τρέχουν το πρωτόκολλο PFR, θα έχουν τυχαία κατανομή στο πεδίο, θα έχουμε βασικό σταθμό, πλήθος κόμβων ίσο με 1000, η εξομοίωση θα γίνει μία φορά για 300 γύρους, το πεδίο θα έχει διαστάσεις 100x100 και οι κόμβοι εμβέλεια 30. Δεν θα υπάρχει sleep-awake και θα συμβούν

20 γεγονότα, με πιθανότητα 0.3 σε οποιονδήποτε γύρο. Τότε, θα δώσουμε στη γραμμή εντολών:

```
> ./simulator PFR R S 1000 300 1 100 100 30 30 90 0 0 20 0.3
```

Τα αποτελέσματα από αυτήν την εξομοίωση καταγράφονται σε δύο αρχεία, τα οποία μπορούμε να χρησιμοποιήσουμε αργότερα για να εξάγουμε συμπεράσματα, γραφικές παραστάσεις, κτλ. Τα αρχεία αυτά περιέχονται στον κατάλογο simdust/out/results και είναι το trace.out και το results.out.

trace.out: Σε αυτό το αρχείο καταγράφονται αναλυτικά στατιστικά στοιχεία από την τελευταία φορά που χρησιμοποιήσαμε τον simulator. Τα στοιχεία αυτά υπολογίζονται και καταγράφονται ανά 10 γύρους. Αναφορικά τα στοιχεία που μπορούμε να δούμε είναι:

- Τα ορίσματα που δώσαμε από την γραμμή εντολών, ώστε να καταλαβαίνουμε τα χαρακτηριστικά του δικτύου σε κάθε εκτέλεση.
- **Round:** Αύξων αριθμός γύρου.
- **TotEvnt, RcvEvnt:** Πλήθος συμβάντων και συμβάντων που έχουν φτάσει στο βασικό σταθμό.
- **SucRate:** Ποσοστό επιτυχίας, δηλαδή το ποσόστο των συμβάντων που έφτασε με επιτυχία στο βασικό σταθμό.
- **TotEnrg:** Συνολική ενέργεια στο δίκτυο, δηλαδή αθροιστικά η ενέργεια όλων των κόμβων του δικτύου.
- **MinEnrg:** Το ποσό ενέργειας του κόμβου με τη λιγότερη ενέργεια στο δίκτυο.
- **AvgEnrg:** Η μέση ενέργεια ενός κόμβου στο δίκτυο (μέσος όρος ενεργειών κόμβων που έχουν θετική ενέργεια).
- **MaxEnrg:** Το ποσό ενέργειας του κόμβου με την περισσότερη ενέργεια στο δίκτυο.
- **AliveP:** Πλήθος "ζωντανών" κόμβων, δηλαδή κόμβων που έχουν ακόμα διαθέσιμη ενέργεια.
- **Awake:** Πόσοι κόμβοι είναι σε κανονικη λειτουργία, δηλαδή σε awake mode.
- **TotTR:** Συνολικός αριθμός μεταδόσεων.
- **TotRCV:** Συνολικός αριθμός λήψεων.

results.out: Στο αρχείο αυτό, υπάρχουν τα συγκεντρωτικά στοιχεία από όλες τις φορές που χρησιμοποιήσαμε τον simulator, δηλαδή δεν υπάρχει η λεπτομέρεια που υπάρχει στο trace.out. Όπου αναφέρουμε μέσο όρο, εννοούμε μέσο όρο από τις επαναλήψεις της εκτέλεσης ενός πειράματος. Μπορούμε π.χ να εκτελέσουμε ένα πείραμα 500 γύρων 10 φορές. Εκτός από τα ορίσματα της γραμμής εντολών, εδώ μπορούμε να δούμε:

- **AvgHops:** Το μέσο πλήθος hops μέχρι να φτάσει ένα μήνυμα από την πηγή του στο βασικό σταθμό.
- **AvgBack:** Ο μέσος αριθμός backtracks μέχρι να φτάσει ένα μήνυμα στο βασικό σταθμό (δεν χρησιμοποιείται στο TEEN).
- **AvgPart:** Το μέσο πλήθος κόμβων που συμμέτέχουν ενεργά στη μετάδοση ενός μηνύματος.
- **AvgTran:** Ο μέσος αριθμός μεταδόσεων.
- **AvgRcv:** Ο μέσος αριθμός λήψεων.
- **RcvEvnt:** Γεγονότα που λήφθησαν από το βασικό σταθμό.
- **TotEvnt:** Γεγονότα που δημιουργήθηκαν μεσα στο δίκτυο.
- **E-Rate:** Η παράμετρος για τα συμβάντα.

Κεφάλαιο 5

Συγκριτική αξιολόγηση των πρωτοκόλλων TEEN και PFR

Στο κεφάλαιο αυτό θα επιχειρήσουμε να κάνουμε μια σύγκριση μεταξύ των πρωτοκόλλων TEEN και PFR, μέσω των αποτελέσματων που πήραμε από τα πειράματα που εκτελέσαμε με τον εξομοιωτή simDust. Επίσης, θα προσπαθήσουμε να αναδείξουμε τα συγκριτικά πλεονεκτήματα και μειονεκτήματά τους, καθώς και να αναζητήσουμε πιθανά υβριδικά πρωτόκολλα. Πρώτα θα δούμε κάποιες λεπτομέρειες για το πρωτόκολλο TEEN, στη συνέχεια ακολουθεί μια σύντομη αναφορά στις μετρικές που χρησιμοποιήσαμε και έπειτα εισερχόμαστε στο πιο ενδιαφέρον κομμάτι του κεφαλαίου, τη σύγκριση των δύο πρωτοκόλλων. Τα αποτελέσματα αυτά περιγράφονται αναλυτικά στην ερευνητική εργασία [17].

5.1 Λεπτομέρειες για την υλοποίηση του TEEN στο simDust

Στην ενότητα αυτή θα περιγράψουμε κάποια χαρακτηριστικά του πρωτοκόλλου TEEN, έτσι όπως τα μεταφέραμε στο simDust. Πριν προχωρήσουμε όμως στις λεπτομέρειες της υλοποίησης του TEEN, θα αναφερθούμε σε κάποια γενικά χαρακτηριστικά του εξομοιωτή, τα οποία ισχύουν για όλα τα πρωτόκολλα. Αρχίζουμε με το μοντέλο κατανάλωσης ενέργειας, το οποίο είναι ουσιαστικά το ίδιο με αυτό που χρησιμοποιήθηκε στα [8] και [9].

Πιο συγκεκριμένα, κάνουμε την υπόθεση ότι η ενέργεια που απαιτείται για μετάδοση και λήψη μηνύματος k-bit σε μια απόσταση d, είναι αντίστοιχα :

$$E_{T_x(k,d)} = E_{elec} \cdot k + e_{amp} \cdot k \cdot d^2$$

και

$$E_{R,r(k)} = E_{elec} \cdot k$$

όπου E_{elec} είναι η ενέργεια που χρειάζεται ο πομποδέκτης για να λειτουργήσει και e_{amp} η ενέργεια που απαιτεί ο ενισχυτής του πομπού για να έχουμε ανεκτό λόγο σήματος-θορύβου.

Γενικά ο εξομοιωτής simDust λειτουργεί σε γύρους, χωρίς αυτό να σημαίνει απαραίτητα ότι επηρεάζει τη λειτουργία των πρωτοκόλλων που εξομοιώνονται. Αυτό έγινε αφενός για να μας διευκολύνει στην υλοποίησή μας και να κάνουμε πιο συγχρονισμένη τη λειτουργία του δικτύου, και αφετέρου αυτό επιταχύνει αρκετά τη διαδικασία των πειραματικών μετρήσεων. Κάθε κόμβος ξεκινά τη λειτουργία του με $2J$ ενέργεια και σε κάθε γύρο που είναι σε κανονική κατάσταση λειτουργίας, υποθέτουμε ότι καταναλώνει $50pJ$ σε επξεργασία πληροφορίας (λειτουργίες CPU). Όταν είναι σε κατάσταση sleep θεωρούμε ότι καταναλώνει μηδενική ενέργεια.

Περνάμε τώρα στα ενδότερα του TEEN. Καταρχήν, η λειτουργία του είναι εξαρτημένη από γύρους, αφού έτσι περιγράφεται στο [9], και υπάρχει μια φάση αρχικοποίησης κάθε 24 γύρους. Επομένως, κάθε 24 γύρους επανασχηματίζονται οι clusters και τα επίπεδα ιεραρχίας. Για το πρώτο επίπεδο ιεραρχίας, η διαδικασία εκλογής είναι ίδια με αυτή που περιγράφεται στο [8] (και στην ενότητα για το LEACH. Για τα επομενα επίπεδα αποφασίσαμε να αλλάξουμε λίγο τη συγκεκριμένη διαδικασία.

Συγκεκριμένα, αν και στη δημοσίευση του TEEN υπάρχει η πρόταση για ιεραρχική δρομολόγηση, δεν υπάρχει περιγραφή του πως θα γίνει αυτό στην πράξη, ούτε και υπάρχουν κάποιες σχετικές μετρήσεις. Αποφασίσαμε να υπάρχουν μέχρι 4 επίπεδα ιεραρχίας στον simDust, τα οποία στην πράξη είναι υπεραρκετά όπως θα δούμε. Αφού λοιπόν εκλεγούν οι cluster-heads και σχηματιστούν οι clusters, υποθέτουμε ότι γίνεται ένα timeout. Οι κόμβοι που έχουν γίνει cluster-heads, έχουν ακούσει όλες τις διαφημίσεις των ομοίων τους, οπότε ξέρουν ποιο είναι το πλήθος τους. Αν ξεπερνά τους 8 cluster-heads, αποφασίζουν πάλι με μια πιθανότητα $P_2 = 0.1$ (ενώ στο πρώτο επίπεδο είναι $P_1 = 0.05$) αν θα περάσουν στο επομενο επίπεδο ιεραρχίας.

Αν απόφασίσουν πως θα γίνουν super-cluster-heads θα κάνουν πάλι μια διαφήμιση του γεγονότος αυτού σε όλο το δίκτυο με ένα broadcast. Αυτοί που παραμένουν cluster-heads θα υπολογίσουν από τα μηνύματα που άκουσαν, ποιος είναι ο πιο κοντινός super-cluster-head. Η διαδικασία αυτή συνεχίζεται για τα επόμενα επίπεδα, ώσπου να σχηματιστεί ένα δένδρο μεταδόσεων με ρίζα το βασικό σταθμό. Τονίζουμε και πάλι ότι ο αριθμός των ιεραρχικών επιπέδων εξαρτάται από τον αριθμό των κομβων του δικτύου και δεν είναι σταθερός από γύρο σε γύρο.

Κάνουμε ακόμα την υποθεση ότι κάθε κόμβος έχει ένα ID, το οποίο έχει μέγεθος 32 bits. Κάθε μήνυμα προς το βασικό σταθμό περιέχει πληροφορία μεγέθους 8 KByte. Τα μηνύματα διαφήμισης που στέλνουν οι cluster-heads έχουν μέγεθος 128 bits. Να κάνουμε σε αυτό το σημείο τη διευκρίνηση ότι, οι διαφημίσεις αυτές γίνονται σε ολόκληρο το δίκτυο και όχι σε μια περιορισμένη περιοχή. Η απάντηση που στέλνει ένας απλός κόμβος στον cluster-head ότι από εδώ και πέρα θα ανήκει στη δικαιοδοσία του, έχει μέγεθος 40 bits.

Το `TDMA` πρόγραμμα μεταδόσεων που μεταδίδει κάθε cluster-head στους κόμβους του cluster του, έχει μέγεθος ανάλογο με το πλήθος των κόμβων αυτών. Στην περίπτωση αυτή δεν έχουμε broadcast σε ολόκληρο το δίκτυο, όπως στην

διαφήμιση του cluster-head, αλλά μετάδοση σε ακτίνα που να περιλαμβάνει τον πιο απομακρυσμένο κόμβο του cluster. Η επικοινωνία μετά με τον cluster-head θεωρούμε ότι είναι point-to-point.

Η διαδικασία αρχικοποίησης, όπως αναφέρθηκε, επαναλαμβάνεται κάθε 24 γύρους, ενώ όταν ένας κόμβος γίνει cluster-head, αποκτά το δικαίωμα να ξαναγίνει cluster-head μετά από άλλες 20 φάσεις αρχικοποίησης, και όχι νωρίτερα. Τέλος, κατά τη διάρκεια της λειτουργίας του δικτύου, ένας απλός κόμβος μπορεί να στείλει μόνο ένα μήνυμα στον cluster-head του στη διάρκεια ενός γύρου, ενώ οι κόμβοι στα ανώτερα επίπεδα μπορούν να στείλουν όσα θέλουν.

Τέλος, για τον εξομοιωτή χρησιμοποιούμε την έννοια του συμβάντος(event), η οποία πρακτικά σημαίνει ότι ένας κόμβος σε κάποια στιγμή έχει πληροφορία για να στείλει στον βασικό σταθμό. Δεν χρησιμοποιούμε τη λογική του TEEN με τα φράγματα (thresholds) και επίσης ένα event αφορά μόνο σε έναν κόμβο και όχι σε μια ομάδα κόμβων που βρίσκονται σε μια περιοχή. Αυτό σημαίνει ότι τα πρωτόκολλα συγκρίνονται με βάση το ίδιο πλήθος συμβάντων και επομένως, η σύγκρισή τους είναι δίκαιη.

5.2 Πειράματα-Αποτελέσματα

5.2.1 Μετρικές που χρησιμοποιήσαμε

Οι μετρικές που χρησιμοποιήσαμε στα πειράματά μας, αναφορικά, είναι το ποσοστό επιτυχίας, η μέση διαθέσιμη ενέργεια ενός κόμβου, το πλήθος ενεργών κόμβων και ο ρυθμός δημιουργίας συμβάντων. Ακολουθούν οι σχετικοί ορισμοί:

Ποσοστό επιτυχίας(success rate): Έστω ότι n συμβάντα δημιουργήθηκαν κατά τη διάρκεια της λειτουργίας του εξομοιωτή, σε διάφορους κόμβους του δικτύου και σε διάφορες χρονικές στιγμές, και έστω ότι k συμβάντα έγιναν τελικώς αντιληπτά από το βασικό σταθμό. Ορίζουμε ως ποσοστό επιτυχίας P_s το λόγο $P_s = \frac{n}{k}$.

Μέση διαθέσιμη ενέργεια κόμβου(average available energy): Έστω ότι E_i είναι η ενέργεια που απομένει σε κάποιον κόμβο i. Ορίζουμε ως μέση διαθέσιμη ενέργεια κόμβου την ποσότητα

$$E_{avg} = \frac{\sum_i^n E_i}{n}$$

δηλαδή είναι η μέση ενέργεια σε κάθε κόμβο του δικτύου, όπου n είναι το πλήθος των κόμβων.

Πλήθος ενεργών κόμβων(number of "active" particles): Ορίζουμε ως πλήθος ενεργών κόμβων h_A, τον αριθμό των κόμβων του δικτύου, τα αποθέματα ενέργεια των οποίων δεν ακόμα εξαντληθεί.

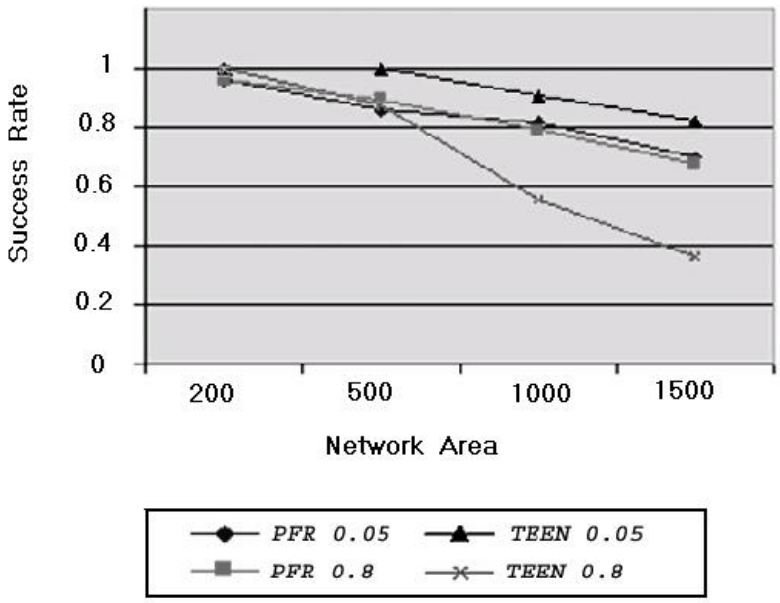
Ρυθμός δημιουργίας συμβάντων(injection rate): Ορίζουμε ως ρυθμό δημιουργίας συμβάντων I_s, την πιθανότητα να δημιουργηθεί από τον εξομοιωτή ένα συμβάν στη διάρκεια οποιουδήποτε γύρου.

Προφανώς, μας ενδιαφέρει να έχουμε ένα υψηλό ποσοστό επιτυχίας καιμεγάλη μέση διαθέσιμη ενέργεια στους κόμβους, για όσο το δυνατόν μεγαλύτερο χρονικό διάστημα. Επίσης, μας ενδιαφέρει ποιο είναι το πλήθος ενεργών κόμβων στο δίκτυο και ποιος είναι ο ρυθμός θανάτων σε κάθε στιγμή, και όλα αυτά μαζί δεδομένου κάποιου ρυθμού δημιουργίας συμβάντων.

5.2.2 Πειραματικά αποτελέσματα

Αρχικά, να αναφέρουμε ότι ονομάζουμε τις εκδόσεις του PFR και του TEEN, τις οποίες υλοποιήσαμε στον εξομοιωτή, SW-PFR(Sleep Awake PFR) και H-TEEN(Hierarchical TEEN) αντίστοιχα.

Η σύγκριση της απόδοσης των πρωτοκόλλων έγινε με τη διεξαγωγή πειραμάτων σε πλήθος πεδίων, των οποίων οι διαστάσεις κυμαίνονται από 200x200m έως 1500x1500m. Στα πεδία αυτά τοποθετήσαμε δίκτυα με πλήθος κόμβων από 500 έως 3000 κόμβους. Οι κόμβοι αυτοί μοιράστηκαν στο πεδίο με τυχαία ομοιόμορφη κατανομή. Πρέπει να σημειώσουμε ότι είναι η πρώτη φορά που γίνεται εξομοίωση με τόσο μεγάλο αριθμό κόμβων και σε τόσο μεγάλα πεδία. Μέχρι τώρα, οι εξομοιώσεις δικτύων έξυπνης σκόνης περιορίζονταν σε δίκτυα 200 κόμβων και πεδία 100x100m, κυρίως γιατί γίνονταν με τον `ns-2`.



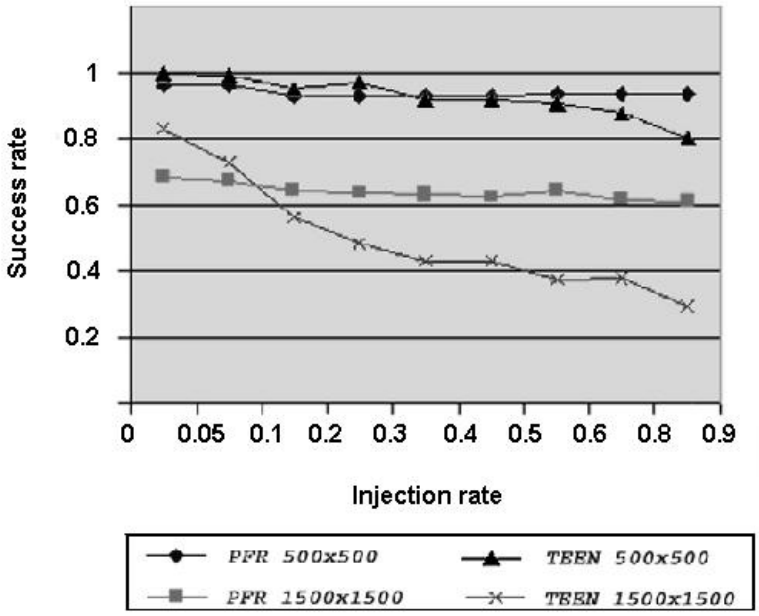
Σχήμα 5.1: Ποσοστά επιτυχίας για για διαφορετικές διαστάσεις δικτύου και injection rate 0.05 και 0.8

Στο σχήμα 5.1, φαίνεται η συμπεριφορά των πρωτοκόλλων σε δύο ακραίες τιμές του ρυθμού δημιουργίας συμβάντων. Πιο συγκεκριμένα, για $I_{s_1} = 0.05$ και $I_{s_2} = 0.8$, βλέπουμε πως μετάβαλλεται το ποσοστό επιτυχίας για τα δύο πρωτόκολλα καθώς μεταβάλλουμε τις διαστάσεις του πεδίου.

΄Ενα πρώτο σχόλιο που μπορεί να κάνει κάποιος, είναι ότι για μικρές διαστάσεις δικτύου και τα δύο πρωτόκολλα συμπεριφέρονται αρκετά καλά, με το H-TEEN να υπερτερεί του SW-PFR. Καθώς όμως οι διαστάσεις του πεδίου μεγαλώνουν, η απόδοση του H-TEEN πέφτει αρκετά γρήγορα, ενώ το SW-PFR δεν παρουσιάζει τόσο μεγάλη αλλαγή στη συμπεριφορά του.

Στην επόμενη γραφική παράσταση (σχήμα 5.2) μπορούμε να δούμε πώς μεταβάλλεται το ποσοστό επιτυχίας, καθώς μεταβάλλουμε το injection rate, για δύο διαφορετικές διαστάσεις πεδίου. Αυτό που παρατηρούμε είναι ότι στο μικρό πεδίο και τα δύο πρωτόκολλα συμπεριφέρονται πολύ καλά, γενικά όμως το H-TEEN υπερτερεί, ενώ στο μεγάλο πεδίο καθώς αυξάνουμε το injection rate, το H-TEEN αρχίζει να μην αποδίδει καλά. Συγκεκριμένα, πέφτει από ένα ποσοστό επιτυχίας 85% στο 30%, ενώ το SW-PFR παραμένει σταθερό, αν και αρχικά έχει χειρότερη επίδοση (70%).

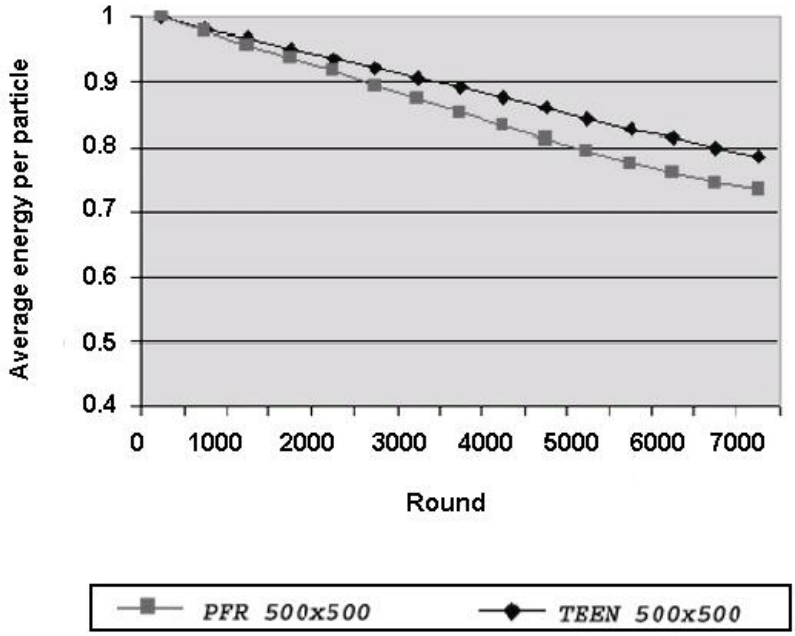
Αυτό μπορεί να εξηγηθεί ως εξής: όσο αυξάνουμε το injection rate, τόσο περισσότεροι κόμβοι στέλνουν αναφορές στον cluster-head τους, οπότε αυτός πρέπει να τα προωθήσει στα ανώτερα από αυτόν επίπεδα ιεραρχίας. Αν αυτό



Σχήμα 5.2: Ποσοστά επιτυχίας για διάφορα μεγέθη injection rate και διαστάσεις δικτύου 500x500, 1500x1500

συνδυαστεί με μεγάλες διαστάσεις δικτύου, είναι αρκετά πιθανό ένας cluster-head να εξαντλήσει τα αποθέματα ενέργειάς του γρήγορα, πριν να ξαναγίνει αρχικοποίηση του δικτύου, με αποτέλεσμα τα μηνύματα που απευθύνονται σε αυτόν να χάνονται από τη στιγμή εκείνη και έπειτα. Το SW-PFR αντίθετα, φαίνεται μάλλον ανεπιρρέαστο από τις μεταβολές του injection rate.

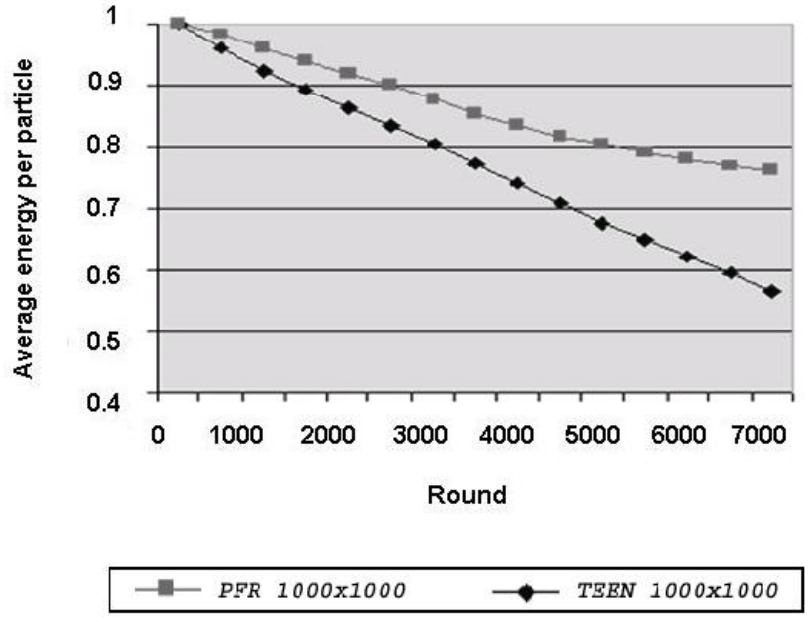
Εξετάζουμε στη συνέχεια πως μεταβάλλεται η μέση διαθέσιμη ενέργεια στους κόμβους σε συνάρτηση με το χρόνο. Έχουμε δύο γραφικές παραστάσεις, μία για ένα πεδίο 500x500m (σχήμα 5.3)και μία για 1000x1000m (σχήμα 5.4).



Σχήμα 5.3: Μέση διαθέσιμη ενέργεια ανά κόμβο για $I_s = 0.05$ και διαστάσεις δικτύου 500x500

Παρατηρούμε και πάλι ότι το H-TEEN είναι καλύτερο σε μικρά πεδία, ενώ το SW-PFR φαίνεται πιο οικονομικό στο μεγάλο πεδίο. Αυτό συμβαίνει γιατί στο H-TEEN ο αριθμός των βημάτων για τη μετάδοση ενός μηνύματος είναι περίπου σταθερός, και στο [8] είχε αποδειχθεί ότι για μικρά πεδία αυτός ο τρόπος μετάδοσης είναι προτιμότερος από μια απλή multihop μετάδοση (όπως στο SW-PFR). Καθώς όμως οι διαστάσεις του δικτύου αυξάνονται, στο H-TEEN η ενέργεια που απαιτείται για μια μετάδοση αυξάνεται εκθετικά, ενώ στο SW-PFR απλά προσθέτουμε hops, δηλαδή έχουμε σχεδόν γραμμική αύξηση.

Συνεχίζουμε με μετρήσεις για τον αριθμό των ενεργών κόμβων του δικτύου σε συνάρτηση με το χρόνο. Πάλι έχουμε δύο διαφορετικά πεδία, ένα μικρό και ένα μεγάλο. Όπως ήταν αναμενόμενο, στο πρώτο πεδίο (σχήμα 5.5) το H-TEEN υπερτερεί και μάλιστα παρατήρουμε ότι ενώ στο πρωτόκολλο αυτό η ενέργεια

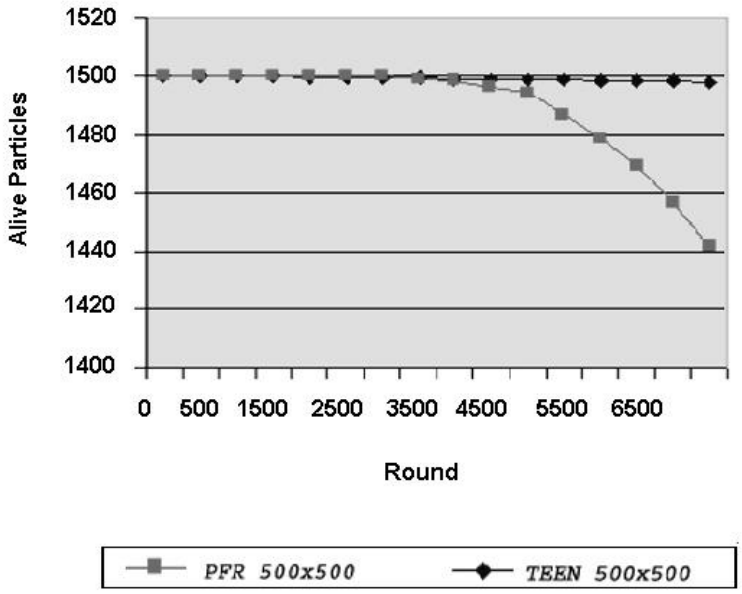


Σχήμα 5.4: Μέση διαθέσιμη ενέργεια ανά κόμβο για $I_s = 0.05$ και διαστάσεις δικτύου 1000x1000

καταναλώνεται με ένα ομοιόμορφο τρόπο και οι κόμβοι του δικτύου πεθαίνουν με κάποιον σταθερό ρυθμό, στο SW-PFR σε κάποιο χρονική στιγμή έχουμε απότομα πλήθος θανάτων. Αυτό εξηγείται ως εξής: στο πρωτόκολλο αυτό επιβαρύνονται πολύ οι κόμβοι που βρίσκονται κοντά στο βασικό σταθμό, γιατί αυτοί μεταφέρουν ουσιαστικά τα μηνύματα από το υπόλοιπο δίκτυο. Επομένως, σε κάποια στιγμή θα πεθάνουν αυτοί οι κόμβοι που είναι πολύ κοντά στο βασικό σταθμό και αυτοί οι θάνατοι θα συμβούν μάλλον με μικρή χρονική απόσταση μεταξύ τους. Επίσης, στο μεγάλο πεδίο (σχήμα 5.6) βλέπουμε, όπως περιμέναμε, ότι υπερτερεί το SW-PFR.

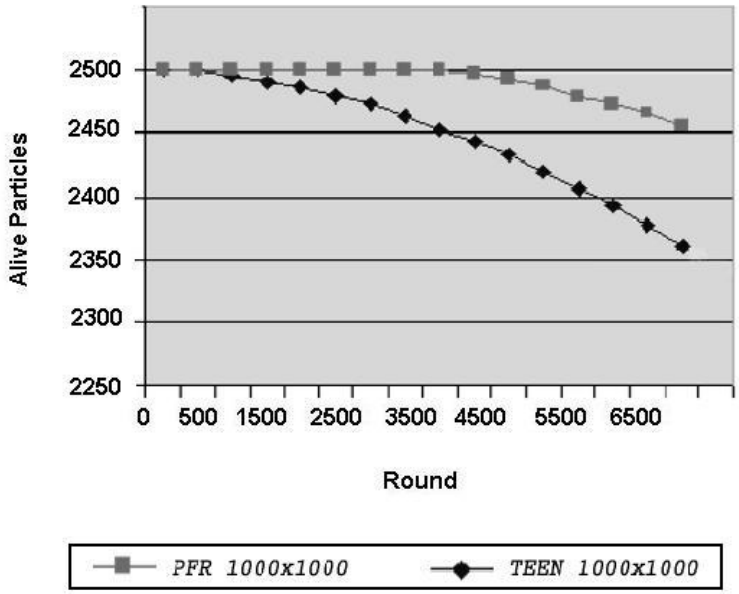
Τέλος, παραθέτουμε μια γραφική παράσταση ((σχήμα 5.7), στην οποία φαίνεται πώς μεταβάλλεται το ποσοστό επιτυχίας σε συνάρτηση με την ακτίνα μετάδοσης στο SW-PFR για ένα πεδίο 1500x1500m με 1500 κόμβους. Από αυτήν φαίνεται ότι η επιλογή της κατάλληλης ακτίνας μετάδοσης παίζει πολύ σημαντικό ρόλο για τη λειτουργία του πρωτοκόλλου.

Από όλα τα προηγούμενα αποτελέσματα, διαφαίνεται μία υπεροχή του PFR σε μεγάλα δίκτυα, ενώ το TEEN είναι καλύτερο σε δίκτυα με σχετικά μικρές διαστάσεις. Ίσως ένας συνδυασμός των δύο πρωτοκόλλων, όπου οι περισσότεροι κόμβοι θα τρέχουν PFR, ενώ κοντά στο sink θα τρέχουν TEEN, θα έχει ακόμα καλύτερα αποτελέσματα από καθένα από τα δύο πρωτόκολλα ξεχωριστά συνδυάζοντας τα προτερήματά τους. Βέβαια, αυτή η προσέγγιση απαιτεί μια

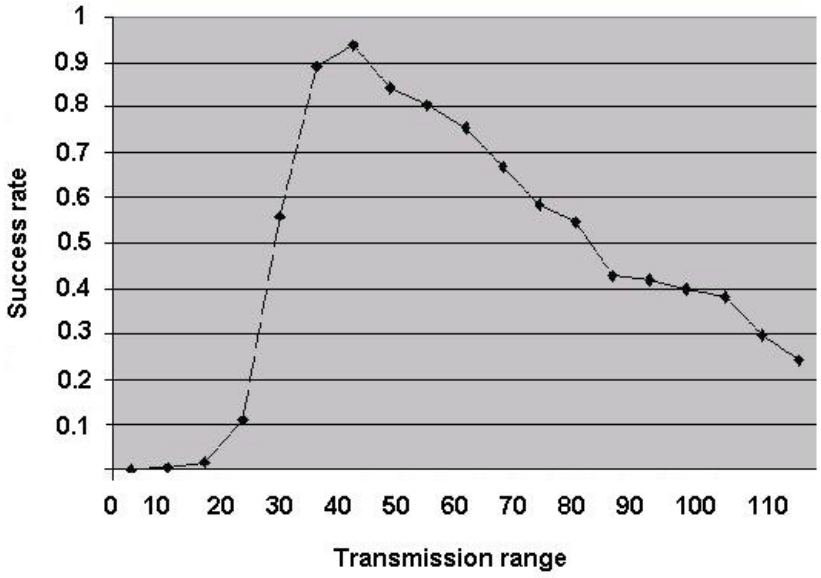


Σχήμα 5.5: Πλήθος ενεργών κόμβων στο χρόνο, για $I_s = 0.05$ και διαστάσεις δικτύου 500x500

πιο ολοκληρωμένη αντιμετώπιση από μια απλή παράθεση ιδεών, προς το παρόν όμως αποτελεί μία ενδιαφέρουσα ερευνητική πρόταση.



Σχήμα 5.6: Πλήθος ενεργών κόμβων στο χρόνο, για $I_s = 0.05$ και διαστάσεις δικτύου 1000x1000



Σχήμα 5.7: Ποσοστά επιτυχίας για διάφορες τιμές της ακτίνας μετάδοσης

Μέρος ΙΙΙ

Ανάπτυξη και λειτουργία πειραματικού δικτύου

Κεφάλαιο 6

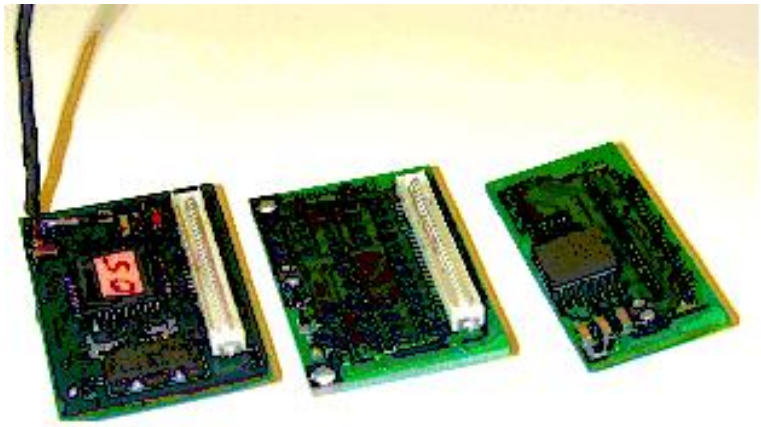
Η πειραματική πλατφόρμα

Στο κεφάλαιο αυτό θα δούμε περισσότερες λεπτομέρειες σχετικά με την πειραματική πλατφόρμα που χρησιμοποιεί η ομάδα που ασχολείται με τα δίκτυα έξυπνης σκόνης στο τμήμα Μηχανικών Η/Υ του Πανεπιστημίου Πατρών. Προτού όμως ασχοληθούμε με τις σχετικές λεπτομέρειες, θα είναι χρήσιμο να κάνουμε μια αναδρομή στην εξέλιξη της πλατφόρμας αυτής μέχρι σήμερα, αφενός μεν γιατί επικρατεί μια μικρή σύγχυση λόγω των ταχύτατων εξελίξεων στο θέμα και αφετέρου για να δείξουμε ακριβώς τη δυναμική του χώρου αυτού, όπως αυτή επιβεβαιώνεται από την ταχύτητα των αλλαγών. Ακολουθούν οι περιγραφές για τα processor board MPR300CB, τα sensor board MTS310CA και το programming board MIB300CA, τα οποία είναι ουσιαστικά τα συστήματα με τα οποία δουλέψαμε για να πραγματοποιήσουμε κάποιες μικρές εφαρμογές, οι οποίες περιγράφονται στο επόμενο κεφάλαιο. Τέλος, γίνεται μια αναφορά στο TinyOS και στα χαρακτηριστικά του, καθώς και στη γλώσσα προγραμματισμού nesC.

6.1 Ιστορική αναδρομή στην εξέλιξη της πλατφόρμας Smart Dust

Καταρχήν να αναφέρουμε ότι το όνομα Smart Dust (έξυπνη σκόνη) είναι το όνομα ενός σχετικού project [20] στο πανεπιστήμιο της Καλιφόρνια στο Berkeley. Προφανώς, είναι μια καλύτερη ονομασία από το sensor networks (δίκτυα αισθητήρων) και επιπλέον δείχνει τη διαφοροποίηση που υπάρχει από τα κλασσικά δίκτυα αισθητήρων που υπήρχαν ως σήμερα. Κύρια μέλη αυτής της ομάδας είναι οι καθηγητές David Culler και Kris Pister. Οι δύο αυτοί άνθρωποι αντιπροσωπεύουν και τις δύο κατευθύνσεις που υπάρχουν μέχρι στιγμής στο Smart Dust.

΄Ετσι, ο David Culler είναι υπεύθυνος για τη σχεδίαση της πλατφόρμας που χρησιμοποιούμε στα πειράματά μας και είναι ουσιαστικά οι κόμβοι ενός δικτύου έξυπνης σκόνης, και οι οποίοι αποτελούνται από ηλεκτρονικά μέρη εκ των οποίων τα περισσότερα είναι ευρέως διαθέσιμα στο εμπόριο, επομένως το κόστος κατασκευής τους είναι σχετικά χαμηλό και επιπλέον υπάρχουν διαθέ-



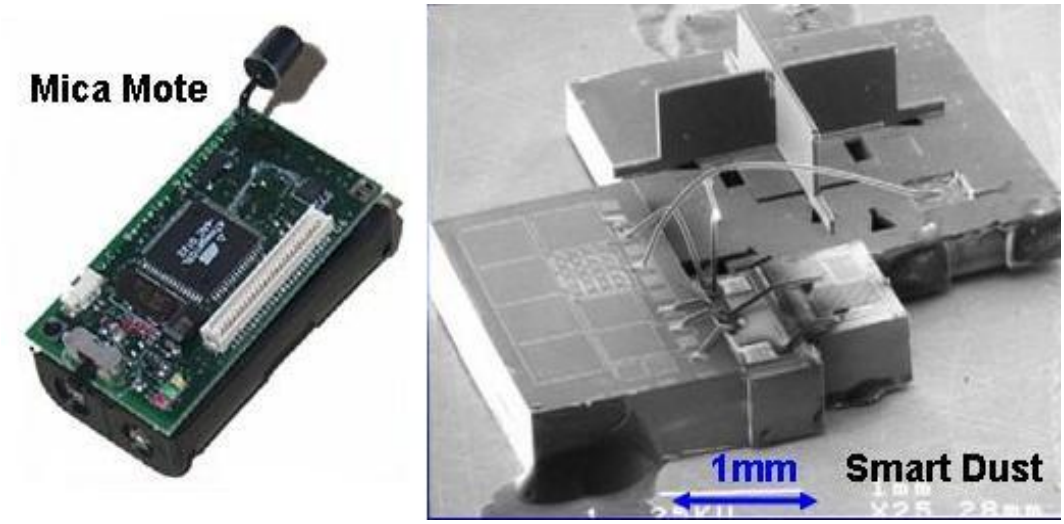
Σχήμα 6.1: Το rene mote και το αντίστοιχο sensor board

σιμα τα σχέδια για πλακέτες κτλ στο Διαδίκτυο, οπότε μπορεί ο οποιοσδήποτε να κατασκευάσει τις δικές του πλακέτες ή να τροποποιήσει τα υπάρχοντα σχέδια.

Συγκεκριμένα, οι κόμβοι που έχουμε διαθέσιμους ονομάζονται MICA motes (mote = κόμβος δικτύου σκόνης) και είναι η τέταρτη γενιά motes που προέκυψε από αυτή την ομάδα. Χρονολογικά προηγήθηκαν τα COTS dust motes, το weC mote και το Rene mote, το οποίο ήταν αρκετά κοντά στην τέταρτη γενιά. Τη στιγμή που γράφεται η εργασία αυτή (Ιούνιος 2003), έχει ήδη βγει πέμπτη γενιά motes, η Mica2. Οι διαφορές εντοπίζονται κυρίως στη μνήμη flash που είναι διαθέσιμη σε κάθε mote και στο εύρος μετάδοσης που έχει το mote.Οι γενιές από το Rene mote και μετά, κατασκευάζονται από την εταιρία CrossBow, οπότε ο αναγνώστης που ενδιαφέρεται να εντρυφήσει περισσότερο στην πλατφόρμα smart Dust, μπορεί να ανατρέξει στην ιστοσελίδα της εταιρίας [22]. Στο σχήμα 6.1 φαίνεται το Rene mote (άκρη αριστερά) μαζί με δύο πλακέτες στις οποίες υπάρχουν διάφοροι αισθητήρες και οι οποίες μπορούν να συνδεθούν με το mote.

Βέβαια, ένα πρόβλημα με τα motes αυτά είναι ότι αν και αρκετά μικρά σε διαστάσεις (1 x 1.5 ίντσες), δύσκολα θα μπορούσαν να χαρακτηρισθούν σκόνη, τουλάχιστον από άποψης διαστάσεων. Από την άλλη όμως δεν είναι αυτός ο σκοπός τους, ο οποίος είναι να δοθούν στην επιστημονική κοινότητα εργαλεία με τα οποία θα μπορεί να κάνει έρευνα στο πεδίο των δικτύων έξυπνης σκόνης όσο πιο κοντά γίνεται στο πραγματικό πεδίο και παράλληλα να έχουν ένα λογικό κόστος. ΄Ενα MICA mote φαίνεται στο αριστερό μέρος του σχήματος 6.2.

Ο Kris Pister είναι αντίθετα υπεύθυνος για τη σχεδίαση της πλατφόρμας που ουσιαστικά είναι η έξυπνη σκόνη και η οποία προς το παρόν δεν είναι ακόμα διαθέσιμη στο ευρύ κοινό. Μια υλοποίηση φαίνεται στα δεξιά της παραπάνω



Σχήμα 6.2: Στα αριστερά ένα MICA mote, στα δεξιά μεγέθυνση σε μικροσκόπιο ενός smart dust particle

εικόνας, οι διαστάσεις της οποίας είναι 3 x 3 mm. Σύμφωνα με τις τελευταίες ανακοινώσεις της ομάδας θα είναι σύντομα διαθέσιμη μια έκδοση με ακόμα μικρότερο μέγεθος (1x1 mm!). Επομένως, αυτό που ήταν επιστημονική φαντασία πριν από λίγα χρόνια τώρα γίνεται πραγματικότητα.

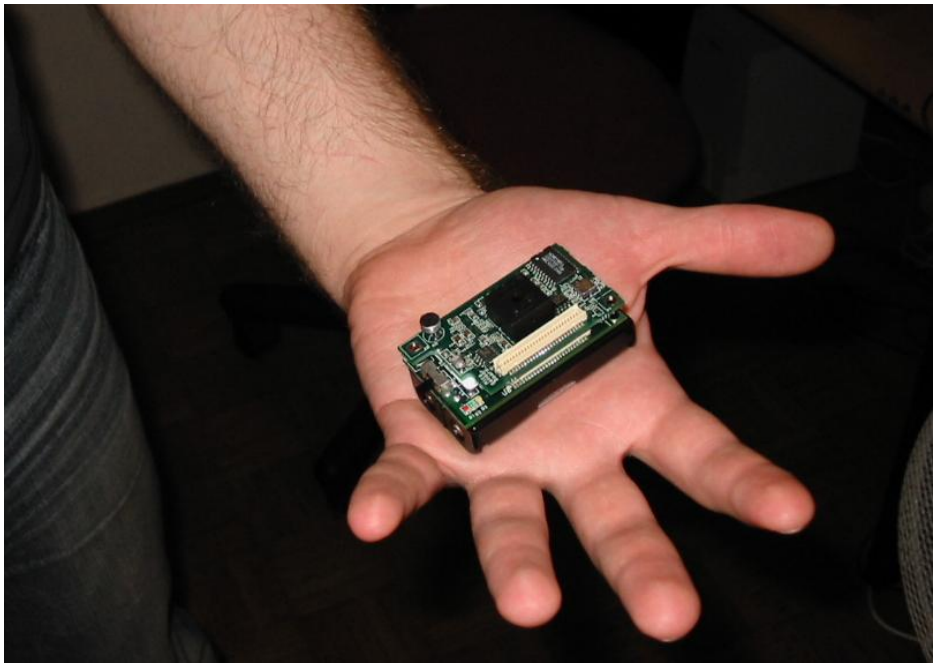
Αυτό που δίνει ζωή στο σύστημα είναι το TinyOS [21], ένα λειτουργικό που σχεδιάστηκε επίσης από την ομάδα Smart Dust, και είναι Open source, οπότε μπορεί οποιοσδήποτε να συνεισφέρει στον κώδικά του. Περισσότερες λεπτομέρειες σε επόμενη ενότητα.

6.2 Hardware

6.2.1 Το Mica mote (mote processor board MPR300CB)

Τα MICA motes MPR300/310 είναι ουσιαστικά οι κόμβοι του δικτύου έξυπνης σκόνης. Το μέγεθος τους είναι λίγο μεγαλύτερο από τις διαστάσεις δύο μπαταριών AA, από τις οποίες τροφοδοτούνται με ενέργεια. Ένα τέτοιο mote φαίνεται στο σχήμα 6.3, για να καταλάβουμε το μέγεθός του σε σχέση με το χέρι ενός ανθρώπου. Στο κάτω δεξί άκρο του mote της εικόνας αυτό που μοιάζει με πυκνωτής είναι εξωτερική κεραία, έτσι ώστε το σήμα του κόμβου να είναι πιο δυνατό και να φτάνει σε μεγαλύτερες αποστάσεις. Τα mote που έχουμε στη διάθεση μας δεν έχουν εξωτερική κεραία και περιοριζόμαστε στις δυνατότητες που έχει ο ενσωματωμένος πομποδέκτης του συστήματος.

Στο σχήμα 6.4 φαίνεται η εσωτερική οργάνωση του mote. Ως κεντρικός επεξεργαστής χρησιμοποιείται ο ATmega 128L της Atmel, ο οποίος είναι ειδικά σχεδιασμένος για συστήματα που έχουν ως στόχο τη χαμηλή κατανάλωση,

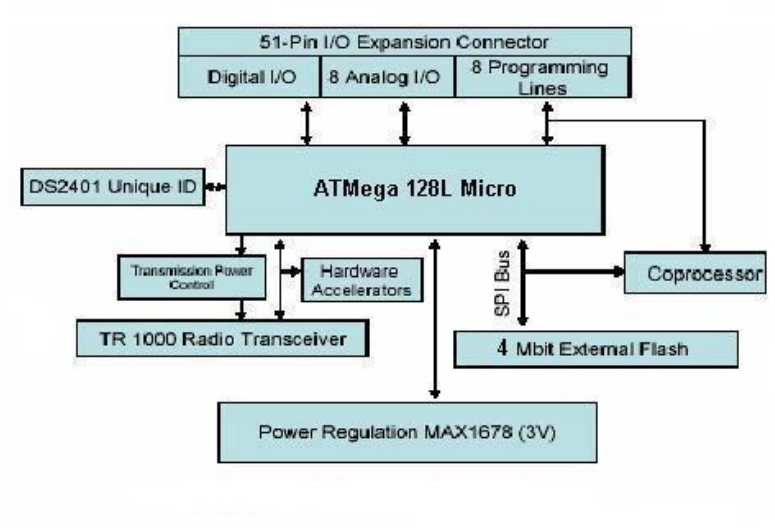


Σχήμα 6.3: ΄Ενα MICA mote χωράει άνετα στην παλάμη του χεριού μας!

διαθέτει 128 KB εσωτερικής flash μνήμης και μετατροπέα αναλογικού σήματος σε ψηφιακό εύρους 10 bit. Επίσης, διαθέτει ενσωματωμένη μονάδα για σειραϊκή επικοινωνία. Στα πρώτα συστήματα Mica που είχαν βγει στο εμπόριο αντί για τον ATmega 128L χρησιμοποιούταν ο ATmega 103L, ο οποίος έχει κάπως πιο περιορισμένες δυνατότητες (κυρίως στο μέγεθος της μνήμης flash).

Στον επεξεργαστή συνδέονται μια σειραϊκή μνήμη flash μεγέθους 512 KB, ένα ID chip με μνήμη 64 bit για την αναγνώριση του mote μέσα στο δίκτυο, ένας πομποδέκτης για ραδιοσυχνότητες ο οποίος λειτουργεί σε συχνότητα 916 MHz (το mote MPR310CB λειτουργεί στα 433 MHz) και μια μονάδα για εξωτερική τροφοδοσία. Να σημειώσουμε ότι η ακτίνα μετάδοσης του πομποδέκτη φθάνει τα 100 πόδια, με άλλα λόγια είναι σχετικά περιορισμένη. Η συνδεσμολογία που περιγράφηκε φαίνεται στο επόμενο σχήμα.

Οι γραμμές I/O του επεξεργαστή που είναι ελεύθερες συνδέονται με ένα σύνδεσμο I/O μεγέθους συνολικά 51 pin (στα schematics και στα spreadsheets του συστήματος αναφέρεται ως Big Guy). Ο σύνδεσμος αυτός ουσιαστικά επιτρέπει τη σύνδεση με ένα sensor board, όπως το MTS310CA που περιγράφεται στην επόμενη ενότητα, και δίνει στο mote τις δυνατότητες για παρακολούθηση ενός πεδίου, αφού το mote από μόνο του δεν έχει αισθητήρες πάνω του. Ακόμα, αυτός ο σύνδεσμος χρησιμεύει για τη σύνδεση με ένα programming board, όπως το MIB300CA που περιγράφεται σε επόμενη ενότητα, για προγραμματισμό της flash μνήμης του.



Σχήμα 6.4: Διάγραμμα με τις συνδέσεις πάνω στο MICA mote

Ο σύνδεσμος αυτός συγκεκριμένα έχει pin για τροφοδοσία και γείωση για το sensor board που θα συνδεθεί μέσω αυτού, pin που λειτουργούν ως αναλογικά inputs για να παίρνουμε τις μετρήσεις από τους αισθητήρες (και οι οποίες οδηγούνται στον A/D μετατροπέα ο οποίος εμπεριέχεται στον επεξεργαστή), pin τα οποία λειτουργούν ως ψηφιακά input και outputs για τον έλεγχο των αισθητήρων και άλλα.

Η μνήμη flash 4Mbit η οποία είναι διαθέσιμη, μπορεί να χρησιμεύσει ως χώρος αποθήκευσης δειγμάτων από τις μετρήσεις που κάνει το mote. Το λειτουργικό που συνοδεύει το mote, το TinyOS, έχει κάποιο υποτυπώδες file system οπότε μπορούμε να διαχειριστούμε σχετικά εύκολα τη μνήμη αυτή. Συνολικά, μπορούμε να αποθηκεύσουμε πάνω από 100,000 δείγματα από μετρήσεις. Τα χαρακτηριστικά του MPR300CB συνολικά, φαίνονται στον πίνακα της επόμενης σελίδας.

Οι διαφορές με την επόμενη γενιά motes, τα Mica2, εντοπίζονται κυρίως στην κατανάλωση ρεύματος, η οποία στα Mica2 είναι σχεδόν η μισή, και στην ακτίνα μετάδοσης, η οποία στα Mica2 φθάνει τα 500 πόδια.

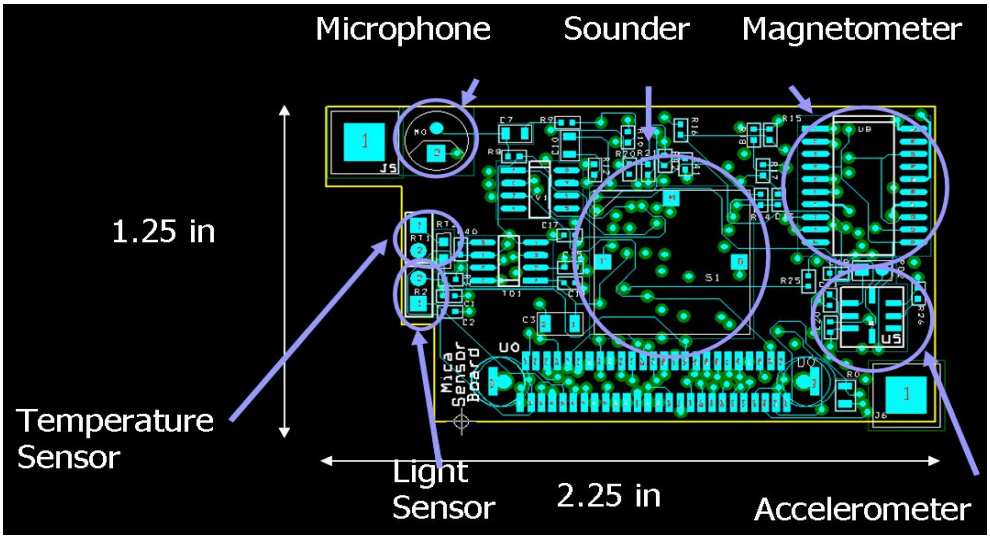
Για περισσότερες λεπτομέρειες, ο αναγνώστης μπορεί να αναζητήσει τα σχηματικά και τα datasheets που υπάρχουν στο site της CrossBow [22].

6.2.2 Το sensor board MTS310CA

Όπως αναφέρθηκε προηγουμένως, το sensor board είναι αυτό που δίνει στο σύστημα τις δυνατότητες για παρακολούθηση του περιβάλλοντος. Συγκεκριμένα, έχει ενσωματωμένους αισθητήρες για *ανίχνευση φωτός, θερμοκρασίας, μαγνητόμετρο* (magnetometer), *επιταχυνσιόμετρο* (accelerometer), *μικρόφωνο* και ένα μικρό *βομβητή* (buzzer). Εκτός από το MTS310CA υπάρχει και το MTS300CA

Επεξεργαστής	
Μοντέλο	Atmel ATmega 128L στα 4MHz
Μνήμη flash	128 Kbytes
SRAM	4Kbytes
EEPROM	4Kbytes
Serial Port	UART
Εύρος A/D μετατροπέα	10 bit
Ένταση ρεύματος που καταναλώνει ο επεξεργαστής	5.5 mA σε κανονική λειτουργία <20μA σε sleep mode
Serial flash	4Mbit (= 512Kbytes)
Πομποδέκτης	
Συχνότητα πομποδέκτη	916MHz
Ταχύτητα μεταφοράς δεδομένων	40 Kbits/sec (maximum)
Ισχύς	0.75mW
Ένταση ρεύματος που καταναλώνει ο πομποδέκτης	12 mA για μετάδοση δεδομένων <1μA για λήψη δεδομένων
Ακτίνα μετάδοσης	100 πόδια (με εξωτερική κεραία) 15 πόδια (χωρίς εξωτερική κεραία)
Γενικά χαρακτηριστικά	
Τροφοδοσία	2 μπαταρίες AA
Εξωτερική τροφοδοσία (προαιρετική)	3 Volt
User interface	3 LEDs
Εξωτερικές συνδέσεις	Σύνδεσμος 51 pin για σύνδεση με sensor board ή programming board

Πίνακας 6.1: Τα χαρακτηριστικά του sensor board MPR300CB



Σχήμα 6.5: Το sensor board MTS310CA

sensor board, το οποίο είναι το ίδιο σύστημα εκτός από το ότι δεν έχει μαγνητόμετρο και επιταχυνσιόμετρο.

Τα παραπάνω εξαρτήματα μας δίνουν τη δυνατότητα να αναπτύξουμε εφαρμογές για δίκτυα έξυπνης σκόνης σε πολλά πεδία, όπως ανίχνευση κίνησης οχημάτων (μαγνητόμετρο), παρακολούθηση θερμοκρασίας (θερμόμετρο), συναγερμούς για ιδιωτική και εταιρική χρήση (μικρόφωνο και αισθητήρας φωτός), σεισμική δραστηριότητα ή κίνηση γενικότερα (επιταχυνσιόμετρο), εφαρμογές ελέγχου στη ρομποτική, και άλλα πολλά. Το board και η θέση των διαφόρων εξαρτημάτων του φαίνεται στο σχηματικό που ακολουθεί (σχήμα 6.5).

Ας δούμε τώρα ένα-ένα τα διάφορα εξαρτήματα του MTS310CA. Αρχίζουμε με το *μικρόφωνο*, του οποίου η χρησιμότητα δεν είναι τόσο προφανής. Η πρώτη εφαρμογή που υπάρχει για το μικρόφωνο είναι αυτή που μπορεί να σκεφθεί ο καθένας, δηλαδή εγγραφή και καταγραφή ήχου σε ένα πεδίο. Μπορούμε να χρησιμοποιήσουμε το μικρόφωνο για να καταγράφουμε τα επίπεδα του θορύβου σε ένα πεδίο, οπότε από αυτά να βγάλουμε συμπεράσματα για τα επίπεδα της δραστηριότητας στο χώρο, ή απλά να ηχογραφήσουμε ήχο και να τον αποθηκεύσουμε στη μνήμη flash του mote.

Μια άλλη πιο έξυπνη χρήση είναι η εκτίμηση της απόστασης μεταξύ των κόμβων του δικτύου (*acoustic ranging*). Αυτό γίνεται με τη βοήθεια του βομβητή που υπάρχει διαθέσιμος. Παράγοντας μια συχνότητα με το βομβητή και στέλνοντας παράλληλα ένα μήνυμα broadcast στο πεδίο, μπορεί κάποιο mote που είναι σε θέση να ακούσει το μήνυμα και το βομβητή να υπολογίσει πόση απόσταση απέχει από το αρχικό mote. Αφού το μήνυμα θα διαδοθεί σχεδόν στιγμιαία και κάποιο mote θα το λάβει, τότε αυτό αρχίζει και μετρά το χρόνο με κάποιο μετρητή μέχρι τη στιγμή που θα ακούσει τη συχνότητα που εκπέμπει το

αρχικό mote. Τότε, λαμβάνοντας υπόψη την ταχύτητα διάδοσης του ήχου στον αέρα, μπορεί να γίνει μια εκτίμηση της απόστασης που διένυσε το ακουστικό σήμα.

Ο *βομβητής* (sounder, buzzer) είναι απλά ένα εξάρτημα που παράγει μια συχνότητα 4KHz και το μόνο που μπορούμε να κάνουμε είναι να το κλείσουμε, να το ανοίξουμε και να ρυθμίσουμε την ένταση του ήχου.

Οι *ανιχνευτές θερμοκρασίας και φωτός* βρίσκονται δίπλα πάνω στο sensor board, μάλιστα η λειτουργία τους είναι παρόμοια. Και οι δύο παράγουν κάποια αντίσταση της οποίας η τιμή μεταβάλλεται ανάλογα με τη τιμή του μετρούμενου φαινομένου. Π.χ όταν μετριέται η ανώτατη τιμή που μπορεί να μετρηθεί, η αντίσταση είναι πολύ μικρή και η τάση που περνά στην έξοδο είναι σχεδόν ίση με την τάση της τροφοδοσίας, ενώ όταν μετριέται η κατώτατη τιμή του φαινομένου η αντίσταση γίνεται πολύ μεγάλη και η τάση είναι ίση με την τάση της γείωσης. Οι δύο αισθητήρες χρησιμοποιούν τις ίδιες γραμμές εξόδου και για αυτό δεν πρέπει να χρησιμοποιούνται μαζί.

Το *μαγνητόμετρο* είναι μια συσκευή που ανιχνεύει μαγνητικά πεδία. Είναι αρκετά ευαίσθητο για να ανιχνεύσει το μαγνητικό πεδίο της Γης. Μπορεί να χρησιμεύσει στην ανίχνευση κινούμενων οχημάτων. Μετρήσεις έδειξαν ότι πιάνει παρεμβολές από οχήματα σε απόσταση 5 μέτρων. Το *επιταχυνσιόμετρο* από την άλλη, μπορεί να ανιχνεύει είτε κίνηση του αντικειμένου πάνω στο οποίο βρίσκεται το mote, είτε να ανιχνεύει σεισμικά κύματα, τα οποία μπορεί να προκαλούνται π.χ από κάποιο φορτηγό που περνά από την περιοχή.

Το sensor board MTS310CA μπορεί να χρησιμοποιηθεί τόσο με τα MICA motes όσο και με τα MICA2 motes.Για περισσότερες λεπτομέρειες, ο αναγνώστης μπορεί να αναζητήσει τα σχηματικά και τα datasheets που υπάρχουν στο site της CrossBow [22].

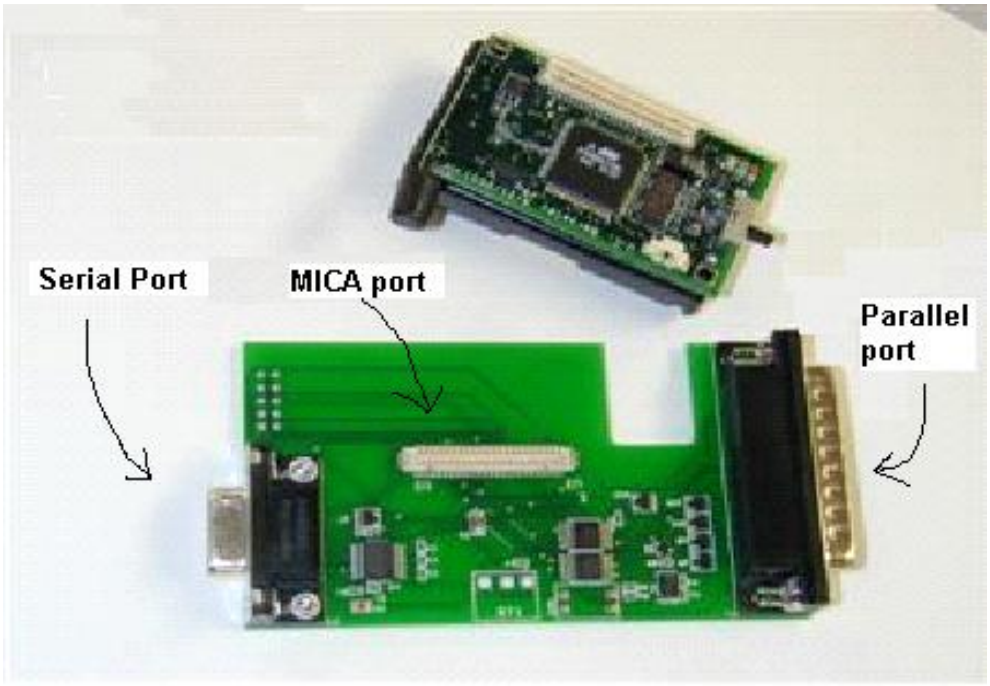
6.2.3 Το programming board MIB300CA

Συνεχίζουμε με το programming board MIB300CA. Η συσκευή αυτή έχει μια σειραϊκή (RS-232) και μία παράλληλη (parallel) θύρα για σύνδεση με υπολογιστή, ο οποίος μπορεί να τρέξει την αντίστοιχη έκδοση του TinyOS. Έχει επίσης μία υποδοχή 51 pin όμοια με αυτή που έχουν τα MICA motes, έτσι ώστε ένα mote μπορεί να συνδεθεί με ένα programming board. Γενικά, το MIB300CA χρησιμοποιείται πάντα σε συνδυασμό με κάποιο mote, αλλιώς από μόνο του δεν έχει καμία χρησιμότητα.

Οι λειτουργίες του MIB300CA είναι δύο:

1. Μέσω αυτού μπορούμε να προγραμματίζουμε τα motes και να περνάμε στη flash τις εφαρμογές που γράφουμε. Ο τρόπος που γίνεται αυτό περιγράφεται στο επόμενο κεφάλαιο. Χρησιμοποιούμε την παράλληλη θύρα του υπολογιστή μας για αυτό το σκοπό.

2. Μπορούμε να επικοινωνούμε με το mote και να διαβάζουμε δεδομένα που στέλνει μέσω του big guy. Χρησιμοποιούμε τη σειραϊκή θύρα του υπολογιστή μας για αυτή τη διαδικασία.



Σχήμα 6.6: Το programming board MIB300CA

Το MIB300CA μπορεί να παίρνει τροφοδοσία είτε από το mote, είτε από εξωτερική τροφοδοσία 3V με έξοδο σε βύσμα jack. Επίσης, υπάρχει και ένα πράσινο LED πάνω στην πλακέτα, το οποίο δείχνει αν η τροφοδοσία είναι σωστή και μπορεί να γίνει η επικοινωνία με το mote, οπότε είναι αναμμένο, αλλιώς είναι σβηστό.

Στο σχήμα 6.6 φαίνεται το MIB300CA δίπλα σε ένα MICA mote.

6.3 Το TinyOS και η γλώσσα προγραμματισμού nesC

Η επιτυχία και η δημοτικότητα που γνωρίζει γενικά η ιδέα της έξυπνης σκόνης τα τελευταία χρόνια οφείλεται σε μεγάλο βαθμό και στο TinyOS, το λειτουργικό σύστημα που συνοδεύει τις συσκευές αυτές.

Το TinyOS μπορεί να θεωρηθεί εξίσου υψηλή τεχνολογία με τα smart dust motes. Είναι το σύστημα που επιτρέπει τη λειτουργία των motes δίνοντάς μας μεγάλη ευελιξία και ευκολία στη δημιουργία εφαρμογών δικτύων έξυπνης σκόνης, παρά τις περιορισμένες δυνατότητες των motes.

Αυτό όμως που έχει μεγαλύτερη σημασία είναι όχι τόσο οι υπηρεσίες τις οποίες μας προσφέρει το TinyOS, οι οποίες εκ των πραγμάτων δεν μπορούν να είναι πολλές, αλλά το γεγονός ότι η φιλοσοφία και ο σχεδιασμός του είναι εναρμονισμένα με το πλαίσιο στο καλούνται να λειτουργήσουν τα motes.

Η τελευταία έκδοσή του είναι η 1.0.1, η οποία κυκλοφόρησε τον Οκτώβρη

του 2002. Πριν από αυτή, κυκλοφόρησαν η έκδοση 1.0 τον Μάιο του 2002 και η έκδοση 0.6.1 το 2001. Η έκδοση 1.0 ήταν μια μεγάλη βελτίωση σε σχέση με την 0.6.1 και περιέχει αρκετές αλλαγές. Ουσιαστικά, η έκδοση 1.0 κυκλοφόρησε για να υποστηρίξει καλύτερα την καινούρια γενιά motes, δηλαδή τα Mica2, τα οποία κυκλοφόρησαν λίγο αργότερα. Επίσης, μια βασική αλλαγή είναι ότι ολόκληρο το σύστημα γράφτηκε ξανά στη γλώσσα nesC, οπότε έχει αλλάξει και το προγραμματιστικό παράδειγμα για εφαρμογές στο TinyOS. Περισσότερα για τη nesC σε επόμενη ενότητα του κεφαλαίου αυτού.

6.3.1 Σχεδίαση του TinyOS

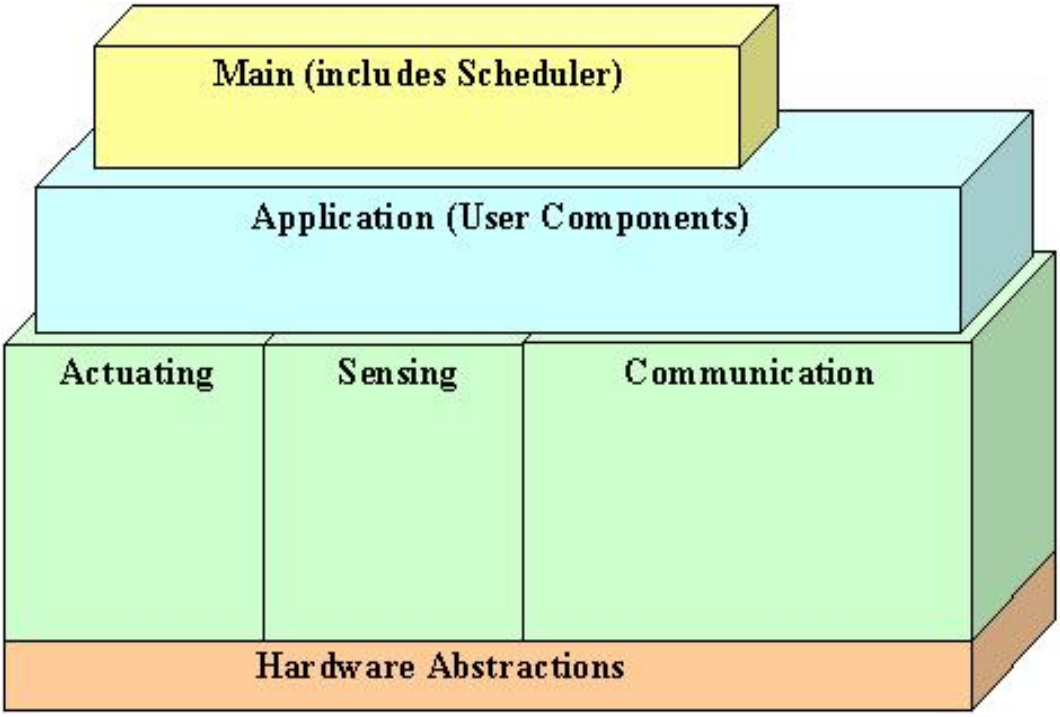
Ας δούμε τώρα πιο αναλυτικά τη σχεδίαση του TinyOS. Οι προκλήσεις που είχε να αντιμετωπίσει και να λύσει το TinyOS ήταν:

- Η χαμηλή κατανάλωση ενέργειας.
- Ιδιαίτερα υψηλές απαιτήσεις για συγχρονισμό:
  1. Ροή πληροφορίας από πολλές πηγές (αισθητήρες, πομποδέκτης)
  2. Μικρή μνήμη→δεν μπορεί να γίνει buffering→πρέπει να επεξεργαστούμε γρήγορα τα μηνύματα που δεχόμαστε, αλλιώς τα χάνουμε!
- Μικρό μέγεθος συνολικά του συστήματος, οπότε δεν μπορούμε να έχουμε controllers για το hardware!
- Η σχεδίαση να είναι modular για να μπορούμε να φτιάξουμε γρήγορα και έυκολα εφαρμογές.

Μια απλοποιημένη εκδοχή της αρχιτεκτονικής του TinyOS που προέκυψε φαίνεται στο σχήμα 6.7:

Όταν λέμε ότι ένα mote τρέχει TinyOS, εννοούμε ότι έχει εγκατεστημένο στη flash μνήμη του ένα binary εκτελέσιμο image με τις βιβλιοθήκες του TinyOS που χρειαζόμαστε, συνδεδεμένες με την εφαρμογή που θέλουμε να εκτελέσουμε. Το image αυτό από εδώ και πέρα θα το αναφέρουμε ως εφαρμογή TOS (TinyOS application). Το TinyOS από μόνο του δεν εκτελεί κάποια ιδιαίτερη λειτουργία και ούτε έχει κάποιο user interface (όπως π.χ το shell στο Unix), οπότε δεν έχει κανένα νόημα να το εγκαταστήσουμε μόνο του σε ένα mote.

Επίσης, στο TinyOS δεν υπάρχει καθόλου η έννοια της διεργασίας (process) όπως την έχουμε συνηθίσει στα σύγχρονα λειτουργικά συστήματα. Αντίθετα, υπάρχει η κυρίαρχη έννοια του component, το οποίο είναι κατά κάποιο τρόπο η αφαίρεση ενός λειτουργικού module του συστήματος. Κατ' ουσία είναι ένα πεπερασμένο αυτόματο και θυμίζει τα module που συναντάμε σε γλώσσες περιγραφής υλικού όπως η Verilog και η VHDL, αν και υπάρχουν αρκετές διαφορές. Αν και τα περισσότερα component είναι software modules, μερικά απλά χρησιμοποιούνται ως ένα απλό interface για το hardware του συστήματος.



Σχήμα 6.7: Η αρχιτεκτονική του TinyOS

Ακόμα, ο αναγνώστης μπορεί από τώρα να ξεχάσει έννοιες όπως kernel, διαχείριση διαδικασιών (process management), εικονική μνήμη (virtual memory), δυναμική κατανομή μνήμης (dynamic memory allocation) και software σήματα (signals). Αφού δεν υπάρχει kernel πρέπει να γίνει απευθείας διαχείριση του hardware, υπάρχει μόνο μια διαδικασία στο σύστημα, υπάρχει γραμμικός χώρος διευθύνσεων και η μνήμη ανατίθεται στατικά την ώρα που γίνεται compile η εφαρμογή στο PC μας. Αυτά όλα βέβαια έχουν σαν αποτέλεσμα το πολύ μικρό μέγεθος του TinyOS, το οποίο δεν ξεπερνά τα 3400 bytes αν συμπεριληφθούν όλα τα components του συστήματος..

Προχωράμε τώρα στην έννοια του event. Τα events χρησιμοποιούνται για την επικοινωνία μεταξύ των components, κάτι σαν software interrupt δηλαδή, και οι αντίστοιχοι handlers που υπάρχουν μέσα στα components μεταβάλλουν την εσωτερική κατάσταση των components τους. Τα events μπορούν να είναι δύο ειδών:

1. Εξωτερικά, προκαλούνται από hardware interrupts. Τέτοια interrupts έχουμε μόνο από τον timer και τον πομποδέκτη.

2. Εσωτερικά, προκαλούνται από event handlers μέσα στα components τα οποία αρχικά ξύπνησαν από κάποιο εξωτερικό event και στη συνέχεια έστειλαν ένα event σε κάποιο άλλο component.

Μια εφαρμογή TOS αποτελείται από ένα scheduler και ένα γράφημα από components, η διασύνδεση των οποίων δείχνει την επικοινωνία μεταξύ τους και τη ροή των events. Αυτή η διασύνδεση ονομάζεται wiring specification και είναι ανεξάρτητη από τα components. Η εφαρμογή συνδέει μόνο τα components τα οποία χρειάζεται για να λειτουργήσει και έτσι στο image που φορτώνουμε στη flash περιέχονται μόνο αυτά και όχι όλα τα components του συστήματος. Οι συνδέσεις αυτές μεταξύ components, οι οποίες ονομάζονται interfaces, είναι διπλής κατεύθυνσης (bidirectional).

Ο scheduler είναι μια απλή FIFO στοίβα και περιέχει δύο ειδών αντικείμενα, events και tasks. Τα tasks χρησιμοποιούνται για εργασίες οι οποίες δεν είναι απαραίτητο να εκτελεστούν αμέσως. Όπως είδαμε, είναι σημαντικό να μην χάνουμε μηνύματα και μετρήσεις από τους αισθητήρες. Για το λόγο αυτό, προσπαθούμε να γράφουμε όσο το δυνατόν πιο απλά components, ώστε η λογική τους να εκτελείται πολύ γρήγορα. Αν υπάρχουν εργασίες που μπορούν να εκτελεστούν χωρίς να μας ενδιαφέρει το πότε θα γίνει αυτό, τις γράφουμε ως tasks και τις στέλνουμε στη στοίβα αυτή. Τα tasks μπορούν να καλέσουν άλλες συναρτήσεις ή να προκαλέσουν ένα event.

Τώρα, ένα event έχει μεγαλύτερη σημασία από ένα task, οπότε αν υπάρχουν μόνο tasks στη στοίβα και ξαφνικά έρθει ένα event, μπορεί να διακοπεί η εκτέλεση του task και να αρχίσει η εκτέλεση του event. Με άλλα λόγια, ένα task μπορεί να γίνει pre-empted από ένα event, δεν μπορεί να γίνει όμως το αντίθετο. Η εκτέλεση των events είναι ατομική μεταξύ τους, όπως ατομική είναι και η εκτέλεση μεταξύ διαφορετικών tasks.

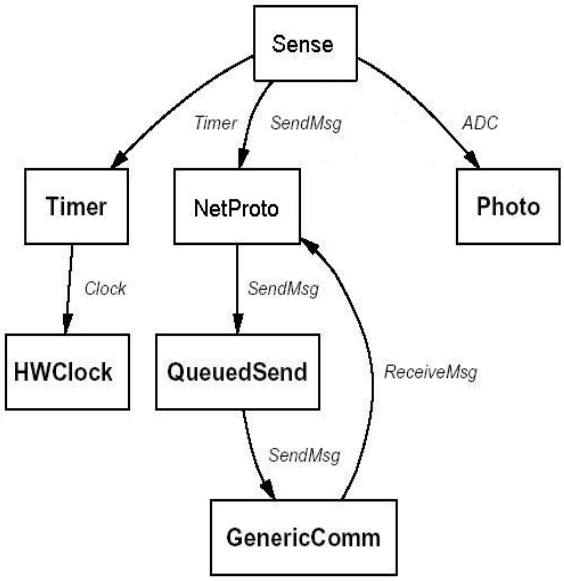
Ας δούμε τώρα ένα παράδειγμα μιας εφαρμογής στο TinyOS. Έστω ότι έχουμε μια εφαρμογή, την οποία ονομάζουμε Sense, και κάνει τα εξής δύο απλά πράγματα:

1. Παίρνει δείγματα από τον αισθητήρα φωτός σε περιοδικά χρονικά διαστήματα.

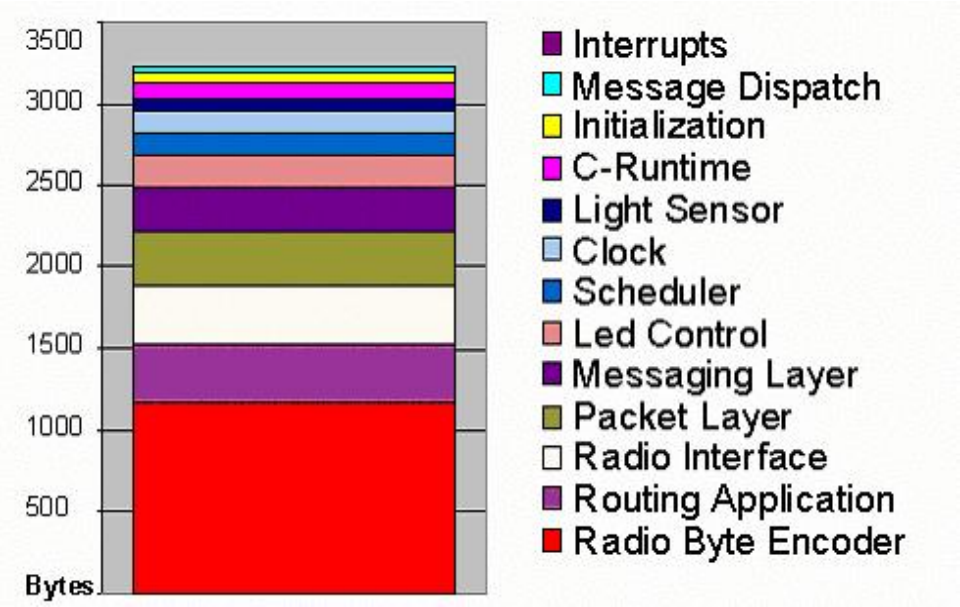
2. Με κάποιο πρωτόκολλο στέλνει τις μετρήσεις που παίρνει από το πεδίο που βρίσκεται σε ένα βασικό σταθμό, μέσω μηνυμάτων που στέλνει με τον πομπό του.

Στο σχήμα 6.10 βλέπουμε το γράφημα με τα components για το οποίο μιλήσαμε προηγουμένως. Αμέσως βλέπουμε ότι υπάρχουν μόνο όσα components χρειαζόμαστε. Από κει και πέρα, βλέπουμε ότι το sense component συνδέεται με τρία components, το timer, το netproto και το photo. Τα ονόματα επάνω στα βέλη είναι τα ονόματα των interfaces που χρησιμοποιεί. Όπως είπαμε τα interfaces είναι ανεξάρτητα από τα components και μπορούμε να χρησιμοποιήσουμε το ίδιο interface για διαφορετικά components (π.χ οι αισθητήρες φωτός και θερμοκρασίας λειτουργούν με τον ίδιο τρόπο). Το sense component λοιπόν είναι ένα αυτόματο που θέτει αρχικά τον hardware timer χρησιμοποιώντας το αντίστοιχο interface. Όταν έρθει το event που δηλώνει ότι το χρονικό διάστημα που θέσαμε πέρασε, το sense χρησιμοποιεί το ADC interface για να πάρει μια μέτρηση από τον αισθητήρα. Ένα αναλυτικό παράδειγμα ακολουθεί στην επόμενη ενότητα.

Στο σχήμα 6.9 φαίνεται η κατανομή της μνήμης ανάμεσα στα διάφορα μέρη από τα οποία αποτελείται το TinyOS. Όπως αναφέρθηκε, το συνολικό μέγεθος δεν ξεπερνά τα 3400 bytes.



Σχήμα 6.8: Το διάγραμμα της εφαρμογής Sense



Σχήμα 6.9: Τα τμήματα του TinyOS και ο χώρος που καταλαμβάνουν

6.3.2 Η γλώσσα προγραμματισμού nesC

Η nesC (προφέρεται Nessie, όπως το τέρας στη λίμνη του Λοχ Νες) είναι μια νέα γλώσσα προγραμματισμού, η οποία αναπτύχθηκε στο πανεπιστήμιο της Καλιφόρνια στο Berkeley σε συνεργασία με την ομάδα που δημιούργησε το TinyOS. Όπως το TinyOS, έτσι και η nesC είναι ένα open-source project και όποιος επιθυμεί να δει τον κώδικα για το μεταγλωττιστή της nesC για το TinyOS ή να τον τροποποιήσει, μπορεί να επισκεφθεί τη σχετική ιστοσελίδα [23] για να προμηθευθεί τα αντίστοιχα αρχεία.

Η nesC χρησιμοποιείται ως το επίσημο προγραμματιστικό παράδειγμα για το TinyOS από την έκδοση 1.0 του λειτουργικού και μετά. Μάλιστα, το TinyOS 1.0 γράφτηκε από την αρχή σε nesC. Πριν από τη nesC χρησιμοποιούταν ένα μίγμα από καθαρή C και πολλές μακρο-εντολές. Το αποτέλεσμα ήταν ότι υπήρχαν προβλήματα ασάφειας και debugging και επίσης ο κώδικας που γραφόταν δεν ήταν ιδιαίτερα κατανοητός. Αντίθετα, η nesC αντιμετωπίζει το πρόβλημα της συγγραφής κώδικα για εφαρμογές TinyOS με ένα αρκετά κομψό και αποδοτικό τρόπο. Για το λόγο αυτό δε θα ασχοληθούμε καθόλου με το προηγούμενο προγραμματιστικό παράδειγμα για το TinyOS. Βέβαια, υπάρχει μια περίοδος μέχρι ο προγραμματιστής να προσαρμοστεί στη νέα γλώσσα και επίσης, ένα πρόβλημα είναι ότι δεν υπάρχουν πολλές πηγές (τουλάχιστον ως τώρα) που να ασχολούνται με το θέμα. Δύο καλές (αν όχι οι μοναδικές) πηγές είναι το Reference manual της nesC [24] και η δημοσίευση των δημιουργών της σχετικά με το θέμα [25], η οποία ήρθε κάπως αργά αν λάβουμε υπόψη ότι το TinyOS 1.0 κυκλοφόρησε ένα εξάμηνο πιο πριν!

Μερικές από τις αρχές που διέπουν τη σχεδίαση της nesC είναι:

1. Η nesC βασίζεται στη C Αυτή η επιλογή έγινε διότι οι μεταγλωττιστές της C παράγουν αποδοτικό κώδικα για όλους του micro που μπορούν να χρησιμοποιηθούν ως επεξεργαστές σε συστήματα έξυπνης σκόνης (προς το παρόν χρησιμοποιούνται μόνο οι AVR της Atmel). Επίσης, ένα μεγάλο μέρος των προγραμματιστών σε embedded συστήματα χρησιμοποιεί τη C ως γλώσσα προγραμματισμού.

2. Ολική ανάλυση προγράμματος κατά τη διάρκεια της μεταγλώττισης (compiling) Η ξεχωριστή μεταγλώττιση μερών ενός προγράμματος δεν είναι διαθέσιμη ως επιλογή στη nesC. Αυτό γίνεται για να είναι πιο ακριβής ο έλεγχος για λάθη και συγχρονισμό (race conditions) στο πρόγραμμα και για πιο αποδοτικό κώδικα (μικρότερο μέγεθος), το οποίο είναι αρκετά βολικό, δεδομένου του μικρού μεγέθους της διαθέσιμης μνήμης.

3. Στατικός κώδικας Η μνήμη κατανέμεται στατικά σε μια εφαρμογή TOS κατά τη διάρκεια της μεταγλώττισης και επίσης το γράφημα διασυνδέσεων μεταξύ των διάφορων component είναι σταθερό και γνωστό. Το μοντέλο των component εξαλείφει την ανάγκη για δυναμική δέσμευση μνήμης και ενθαρρύνει ένα ευέλικτο σχεδιασμό.

4. Η nesC υποστηρίζει και αντικατοπτρίζει τη φιλοσοφία του TinyOS αφού βασίζεται και αυτή στην ιδέα των components και της επικοινωνίας μεταξύ τους μέσω των events.

Προχωράμε τώρα στα ενδότερα της γλώσσας. Όπως είπαμε, μια εφαρμογή TOS αποτελείται από διάφορα components, τα οποία συνδέονται μεταξύ τους με κάποιο συγκεκριμένο τρόπο. Υπάρχουν μόνο δύο είδη component, τα πρώτα ονομάζονται module και τα δεύτερα configuration. Κάπου εδώ η κατάσταση περιπλέκεται, οπότε ζητώ την κατανόηση του αναγνώστη. Οι τρεις βασικές έννοιες στις οποίες πρέπει να δώσει σημασία και να κατανοήσει είναι module, configuration και interface.

Τα module λοιπόν υλοποιούν τη λογική που αλλάζει την κατάσταση του αυτόματου, στο οποίο αντιστοιχεί ουσιαστικά ένα component. Αυτό γίνεται με κώδικα που μοιάζει πολύ με C. Πρακτικά αυτό σημαίνει ότι υλοποιούν τα interface, τα οποία έχουν δηλώσει ότι χρησιμοποιούν. Πριν να αναλύσουμε την έννοια του interface, να πούμε ότι η αντιστοίχηση interface στα modules, δηλαδή ποια interface χρησιμοποιεί ένα module, γίνεται με ένα configuration. Προχωράμε τώρα να αποσαφηνίσουμε τις παραπάνω έννοιες.

Ένα interface χρησιμοποιείται για να ορίσει τον τρόπο που επικοινωνούν δύο modules. Από τα δύο modules, το ένα ονομάζεται χρήστης (user) και το άλλο παροχέας (provider). Αυτή η διάκριση γίνεται για να δείξει μάλλον τη ροή του ελέγχου. Για να γίνει αυτό πιο κατανοητό, έστω ότι έχουμε ένα module πολύ γενικό και αφηρημένο. Για να μπορέσει να εκτελέσει μια λειτουργία χρειάζεται τις υπηρεσίες από άλλα modules, τα οποία είναι λιγότερο γενικά και πιο κοντά ας πούμε στο hardware. Έτσι, το πρώτο module θα χρησιμοποιήσει το interface με το οποίο συνδέεται με ένα module πιο κάτω στην ιεραρχία. Το πρώτο module θα γίνει user και το δεύτερο provider. Στο παράδειγμα που είδαμε στην προηγούμενη ενότητα, το module Sense θα χρησιμοποιήσει το interface ADC, το οποίο του παρέχει το module Photo. Η κατεύθυνση του βέλους κάνει σαφή τη διάκριση αυτή.

Τα interface είναι ο μόνος τρόπος αλληλεπίδρασης μεταξύ των modules και περιέχουν εντολές (commands) και γεγονότα (events). Οι εντολές πρέπει να υλοποιηθούν από τη μεριά του provider ενώ τα events από τη μεριά του user. Οι εντολές μας επιτρέπουν να ελέγξουμε ένα module, ενώ τα events είναι οι αντιδράσεις που προκύπτουν από την εκτέλεση των εντολών. Αυτό ίσως να μην φαίνεται λογικό με μια πρώτη ματιά αλλά σκεφθείτε το ως εξής: Τα interface είναι ανεξάρτητα από τα components. Δεν περιέχουν κώδικα ο οποίος υλοποιεί κάποια λογική. Απλά καθορίζουν τον τρόπο επικοινωνίας ανάμεσα στα modules.

Ακολουθούν οι ορισμοί των interfaces που χρησιμοποιεί το module Sense. Βλέπουμε ότι απλά ορίζουν τις εντολές και τα events που χρησιμοποιούν.

```
interface Timer {
```

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

interface SendMsg {
    command result_t send(TOS_Msg *msg, uint16_t length);
    event result_t sendDone(TOS_Msg *msg, result_t success);
    }

interface ADC {
    command result_t getData();
    event result_t dataReady(uint16_t data);
    }
```

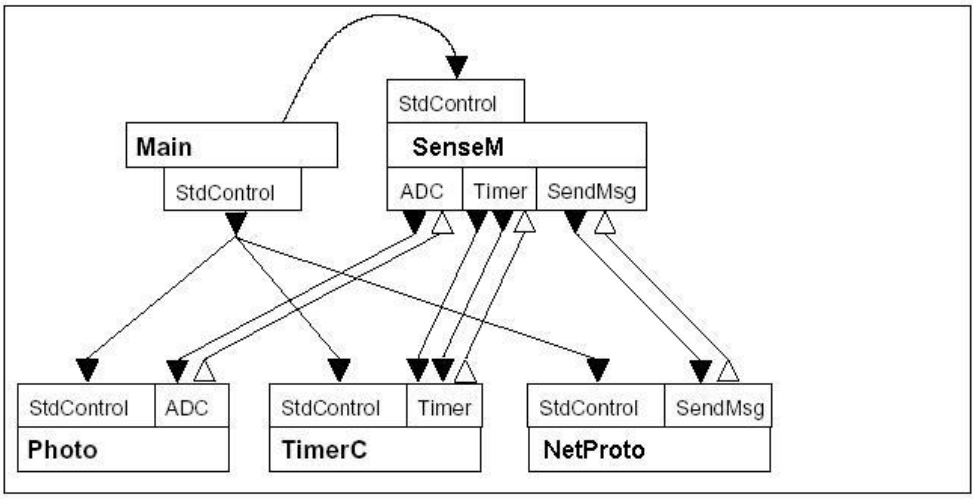
Για το παράδειγμά μας, στο interface ADC η εντολή getData() θα πρέπει να υλοποιηθεί από το module Photo και το event dataReady θα πρέπει να υλοποιηθεί από το module Sense. Ακολουθεί μέρος του ορισμού των module αυτών.

```
module Sense {
    uses interface ADC;
    uses interface Timer;
    uses interface Send;
    }
implementation {
    uint16_t sensorReading;
    event result_t Timer.fired() {
        call ADC.getData(); return SUCCESS;
    }
    event result_t ADC.dataReady(uint16_t data) {
        sensorReading = data; ...
        return SUCCESS;
    }
}

module Photo{
    provides interface ADC;
}
implementation {
    …
    command result_t getData() {
    ….
    }
}
```

Τώρα η εύλογη απορία είναι πώς καλούνται αυτά τα events και οι εντολές που παρέχει ένα interface. Ας ξαναδούμε την διαδικασία που περιγράψαμε στο παράδειγμα με το Sense.

Επιστρέφουμε τώρα στην έννοια του configuration. ΄Ενα configuration είναι ένα component που λεει στον μεταγλωττιστή ποια interface χρησιμοποιούμε στην εφαρμογή μας, μεταξύ ποιων modules χρησιμοποιούνται και ποιοι είναι



Σχήμα 6.10: Αναλυτικό διάγραμμα των components της εφαρμογής Sense

οι ρόλοι, user ή provider. Λογικά, ένα configuration σχηματίζει ένα γράφημα όπως αυτό στο σχήμα που ακολουθεί. Έστω ότι το module sense αποφασίζει ότι θέλει ένα δείγμα από τον αισθητήρα φωτός (module Photo). Θα καλέσει την εντολή getData(), ως εξής:

```
call ADC.getData();
```

δηλαδή αναφέρεται στο interface που χρησιμοποιεί (ADC) και στην εντολή getData ως συνάρτηση μέλος ενός αντικειμένου ADC. Όταν ληφθεί το δείγμα, πρέπει να ειδοποιηθεί το module Sense για να το καταγράψει σε μία μεταβλητή. Αυτό θα γίνει με τη χρήση ενός event, το οποίο θα κληθεί μέσα στο module Photo ως εξής:

```
signal ADC.dataReady( SensorReadings);
```

Το διάγραμμα που αντιστοιχεί στο configuration του Sense φαίνεται στο σχήμα 6.10. Από το γράφημα αυτό, ο μεταγλωττιστής ξέρει ότι το ADC συνδέει τα Sense και Photo, οπότε μπορεί να καταλάβει πως θα γίνει η σύνδεση των αντίστοιχων συναρτήσεων.

Ακόμα, στο σχήμα αυτό περιέχεται και ένα module για το οποίο δεν έχουμε μιλήσει ακόμα, το Main, το οποίο περιέχεται σε κάθε εφαρμογή TOS και χρησιμοποιεί το interface StdControl. Αυτό το interface χρησιμοποιείται για να αρχικοποιήσει όλα τα modules της εφαρμογής, είναι κάτι σαν ένα γενικό κουμπί reset με άλλα λόγια.

Κάτι άλλο που πρέπει να παρατηρήσουμε είναι η σύμβαση που χρησιμοποιείται σε τέτοια γραφήματα από τους συγγραφείς του TinyOS, δηλαδή ότι τα μαύρα τρίγωνα σε ένα module δηλώνουν commands, ενώ τα λευκά δηλώνουν

events. Παρατηρούμε ότι το interface StdControl έχει μόνο μία εντολή, αφού απλά χρειάζεται να κάνει reset στα modules.

Θα δούμε τώρα πως ορίζεται το configuration για την εφαρμογή Sense. ΄Εχουμε:

```
configuration Sense { }
implementation {
    components Main, SenseM, Photo, TimerC, NetProto;

    Main.StdControl -> SenseM.StdControl;
    Main.StdControl -> Photo.StdControl;
    Main.StdControl -> TimerC.StdControl;
    Main.StdControl -> NetProto.StdControl;

    SenseM.ADC -> Photo.ADC;
    SenseM.Timer-> TimerC.Timer;
    ...
}
```

Παρατηρούμε ότι ένα configuration δεν περιέχει καθόλου κώδικα για κάποια εφαρμογή και ότι στο implementation μέρος έχει μια σειρά από δηλώσεις που δείχνουν τη σχέση μεταξύ δύο components. Αρχικά, δηλώνουμε ποια components θα χρησιμοποιήσουμε στην εφαρμογή μας. Όπως αναφέρθηκε, υπάρχει πάντοτε ένα component με το όνομα Main, το οποίο χρησιμοποιείται απλά για να αρχικοποιήσουμε το σύστημα μέσω του interface StdControl. Προφανώς το όνομα δείχνει τη σχέση με τη συνάρτηση main() που χρησιμοποιείται στη C.

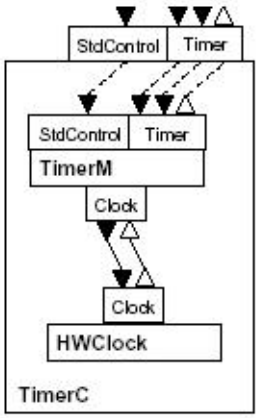
Το βέλος που χρησιμοποιείται στις δηλώσεις μέσα σε ένα configuration δείχνει ποιος είναι ο provider και ποιος ο user ανάμεσα σε δύο components. Στα αριστερά του βέλους είναι το component που χρησιμοποιεί το interface, ενώ στα δεξιά αυτό που το παρέχει.

Θα προχωρήσουμε τώρα ένα βήμα παραπέρα και θα ανακαλύψουμε ότι στην πραγματικότητα, το component TimerC είναι configuration και όχι module! Συγκεκριμένα, τα configuration μπορούν και αυτά να παρέχουν και να χρησιμοποιούν interface! Στο σχήμα που ακολουθεί φαίνεται ότι αποτελείται από δύο module, το TimerM και το HWClock, τα οποία συνδέονται μεταξύ τους με ένα clock interface.

Ο ορισμός του configuration TimerC είναι ο εξής:

```
configuration TimerC {
    provides {
                interface StdControl;
       interface Timer;
    }
} implementation {
    components TimerM, HWClock;

    StdControl = TimerM.Stdcontrol;
    Timer = TimerM.Timer;
```



Σχήμα 6.11: Το configuration TimerC

```
    TimerM.Clk -> HWClock.Clock;
}
```

Παρατηρούμε ότι σε δύο δηλώσεις αντί για το βέλος που είδαμε πριν, χρησιμοποιείται το ίσον. Αυτό γίνεται επειδή θέλουμε να αντιστοιχήσουμε τα interface που παρέχει το component στα interface τα οποία παρέχει το TimerM. Δεν παρεμβάλλεται κάποιο component μεταξύ τους, οπότε είναι ουσιαστικά τα ίδια interface.

Τελειώνοντας αυτήν την ενότητα, να τονίσω ότι σκοπός μου ήταν απλά να παρουσιάσω τα βασικά χαρακτηριστικά της γλώσσας και όχι να αποτελέσει αυτή η ενότητα μια κανονική εισαγωγή στον προγραμματισμό σε nesC. Υπάρχουν πολλά πράγματα ακόμα να ανακαλύψει κάποιος στη nesC και συνιστώ σε όποιον επιθυμεί να ασχοληθεί σοβαρά με το αντικείμενο, να αφιερώσει πρώτα αρκετό χρόνο στο reference manual της γλώσσας. Αυτό που πρέπει να κάνει έπειτα είναι να κοιτάξει τα tutorials που περιέχονται στη διανομή του TinyOS (υπάρχουν και στη σελίδα της ομάδας στο Berkeley), στα οποία αναλύεται βήμα προς βήμα πως γράφεται μια εφαρμογή TOS. Η nesC είναι σχετικά μικρή σε μέγεθος γλώσσα, αλλά έχει πολλά λεπτά σημεία που απαιτούν γνώση των χαρακτηριστικών της σε βάθος.

Κεφάλαιο 7

Πειραματικές εφαρμογές

Σε αυτό το κεφάλαιο θα περιγράψουμε κάποιες απλές εφαρμογές για να δείξουμε κάποιες από τις δυνατότητες των motes. Προτού γίνει αυτό όμως, θα περιγράψουμε τον τρόπο με τον οποίο μεταγλωττίζουμε κώδικα σε nesC για την πλατφόρμα μας και επίσης πως επικοινωνεί το PC μας με τα motes, προκειμένου να δούμε τι γίνεται στο δίκτυο έξυπνης σκόνης.

7.1 Προγραμματισμός των motes-Χρήσιμα εργαλεία

Καταρχήν, μαζί με τη διανομή του TinyOS παρέχονται και όλα τα απαραίτητα εργαλεία για το σκοπό μας, μαζί με ένα πλήθος από έτοιμες εφαρμογές για δίκτυα έξυπνης σκόνης. ΄Ετσι, αφού κάποιος εγκαταστήσει το TinyOS στο PC του, κάτω από τον κατάλογο *TinyOS-1.x/apps* μπορεί να βρει έτοιμο κώδικα σε nesC, ο οποίος μπορεί εύκολα να μεταγλωττιστεί και να "τρέξει" σε smart dust motes, ή να τον αλλάξει κατά το δοκούν.

΄Εστω τώρα ότι θέλουμε να εγκαταστήσουμε την έτοιμη εφαρμογή Blink σε ένα mote. Η Blink δεν κάνει κάτι ιδιαίτερο, απλά αναβοσβήνει ένα από τα LEDs του mote κάθε δευτερόλεπτο. Η εφαρμογή αυτή περιέχεται στον κατάλογο *TinyOS-1.x/apps/Blink*, όπου ήδη περιέχεται ένα makefile για τη μεταγλώττιση του κώδικα. Δίνουμε στη γραμμή εντολών:

```
>make mica
>make mica install
```

Η πρώτη εντολή δημιουργεί ένα εκτελέσιμο αρχείο (*Blink.exe*) και αμέσως μετά παράγει ένα binary image για εγκατάσταση στη flash του mote. Η δεύτερη εντολή γράφει το image αυτό στο mote, μέσω του programming board. Εάν θέλουμε να γράψουμε μια δικιά μας εφαρμογή, χρειάζονται ελάχιστες αλλαγές στο υπάρχον makefile για να μεταγλωττίζουμε εύκολα τον κώδικά μας.

Προτού κάνουμε τη μεταγλώττιση, πρέπει να συνδέσουμε το mote στο programming board, όπως φαίνεται στην επομενη εικόνα(σχήμα 7.1). ΄Επειτα συνδέουμε το board στην παράλληλη θύρα του υπολογιστή μας. Φυσικά, αντί να το συνδέσουμε κατευθείαν στη θύρα, μπορούμε να χρησιμοποιήσουμε ένα

Σχήμα 7.1: Το MICA board συνδεδεμένο στο programming board

καλώδιο για παράλληλη, αλλά αυτό πρέπει να έχει μήκος το πολύ 1 μέτρο για να δουλέψει σωστά. Όταν το mote είναι συνδεδεμένο στο board, το board μπορεί να παίρνει ενέργεια είτε από τις μπαταρίες του mote (το οποίο πρέπει να είναι ON) ή μέσω μετασχηματιστή με έξοδο 3V σε βύσμα jack.

Χρησιμοποιούμε επομένως την παράλληλη θύρα για να προγραμματίσουμε τα motes. Για να επικοινωνήσουμε μαζί τους (να παίρνουμε μετρήσεις ή να δίνουμε εντολές) πρέπει το board να συνδεθεί στη σειραϊκή θύρα του υπολογιστή μας. Γενικά όμως, δεν συνίσταται ο χρήστης να έχει συνδεδεμένες στο board ταυτόχρονα σειραϊκή και παράλληλη θύρα.

Η τελευταία συνδεσμολογία(με τη σειραϊκή δηλαδή), προϋποθέτει ότι το mote στο board έχει εγκατεστημένη κάποια εφαρμογή που στέλνει πακέτα στη θύρα επικοινωνίας με το board. Από εκεί και πέρα, υπάρχουν δύο εφαρμογές που μας βοηθούν να διαβάσουμε τις πληροφορίες που στέλνει το mote μέσω της σειραϊκής θύρας.

Η πρώτη τέτοια εφαρμογή λέγεται **ListenRaw** και βρίσκεται στον κατάλογο *tools/java/net/tinyos/tools*. Την εκτελούμε δίνοντας στη γραμμή εντολών (έστω ότι βρισκόμαστε στον κατάλογο *tools/java* και ότι το board είναι συνδεδεμένο στην σειραϊκή θύρα COM1)

```
java net.tinyos.tools.ListenRaw COM1
```

Αν υπάρχει στο mote εφαρμογή που στέλνει μηνύματα, στην οθόνη μας θα αρχίσουν να εμφανίζονται ένα-ένα τα μήνυματα αυτά, με τα περιεχόμενά τους.

Η δεύτερη λέγεται **SerialForward** και η λειτουργία της είναι να παίρνει αυτά που διαβάζει από τη σειραϊκή θύρα και να τα προωθεί σε κάποιο TCP port(συνήθως το 9000). Οι εφαρμογές που επικοινωνούν με το *Serialforward* παίρνουν από εκεί τα μηνύματα και τα χρησιμοποιούν όπως θέλουν. Μια τέτοια

εφαρμογή είναι το **Oscilloscope** (*tools/java/net/tinyos/oscope*), η οποία επικοινωνεί με το *SerialForward* σε κάποιο port, λαμβάνει τα περιεχόμενα των μηνυμάτων του mote, τα οποία υποτίθεται ότι είναι μετρήσεις από το sensor board, και τα απεικονίζει σε μια γραφική παράσταση. Οι δυο αυτές εφαρμογές θα μας φανούν ιδιαίτερα χρήσιμες για τα απλά πειράματα που θα περιγράψουμε στις ενότητες που ακολουθούν.

7.2 Παρουσίαση εφαρμογών

7.2.1 Μετάδοση μηνύματος μέσω radio και ρύθμιση εμβέλειας

Στην ενότητα αυτή θα στηριχθούμε σε δύο έτοιμες εφαρμογές, τις *CntToRfmAndLeds* και *RfmToLeds*, τις οποίες στη συνέχεια θα τροποποιήσουμε. Επίσης, θα δούμε αναλυτικά τον κώδικα για αυτές τις δυο εφαρμογές προκειμένου να γίνει κατανοητή η διαδίκασια της σχεδίασης μιας εφαρμογής TOS. Πριν να συνεχίσουμε, να σημειώσουμε ότι χρησιμοποιούμε το λειτουργικό σύστημα Linux στο PC μας.

Ξεκινούμε με την *CntToRfmAndLeds*, η οποία περιέχεται στον κατάλογο *TinyOS-1.x/apps/CntToRfmAndLeds* και η λειτουργία της είναι να εμφανίζει τα τελευταία 3 bit από ένα counter 16 bit στα 3 LEDs που υπάρχουν διαθέσιμα στα mica motes, και να στέλνει ένα μήνυμα με την τιμή του counter σε όλους τους κόμβους γύρω του (broadcast). Σύμφωνα με τη σύμβαση που ακολουθείται στο TinyOS, το wiring για την εφαρμογή περιέχεται σε ένα αρχείο με όνομα ίδιο με αυτό της εφαρμογής, δηλαδή στο *CntToRfmAndLeds.nc*. Η συγκεκριμένη εφαρμογή είναι απλά ένα configuration, το οποίο συνδέει έτοιμα components από τη βιβλιοθήκη του TinyOS, συγκεκριμένα τα *Counter*, *IntToLeds*, *IntToRfm* και *TimerC*. Ο αντίστοιχος κώδικας είναι:

```
configuration CntToLedsAndRfm {
}
implementation {
  components Main, Counter, IntToLeds, IntToRfm, TimerC;

  Main.StdControl   -> Counter.StdControl;
  Main.StdControl   -> IntToLeds.StdControl;
  Main.StdControl   -> IntToRfm.StdControl;
  Counter.Timer     -> TimerC.Timer[unique("Timer")];
  Counter.IntOutput -> IntToLeds;
  Counter.IntOutput -> IntToRfm;
}
```

Παρατηρούμε ότι δηλώνουμε και το component *Main*, το οποίο όπως είδαμε στο προηγούμενο κεφάλαιο είναι το component από το οποίο ξεκινά η εκτέλεση και αρχικοποιεί τα υπόλοιπα components μέσω πολλαπλής σύνδεσης του *Main.StdControl* interface. Ας δούμε τώρα τι κάνει το κάθε component ξεχωριστά.

Το component *TimerC* είναι μια αφαίρεση του hardware timer που διαθέτει ο AVR, και μπορούμε σε μια εφαρμογή να έχουμε 256 διαφορετικά στιγμιότυπα

του component αυτού. Για το λόγο αυτό, στη δήλωση του wiring χρησιμοποιούμε το όρισμα unique για να είμαστε σίγουροι ότι κάθε φορά που το καλούμε δημιουργούμε ένα καινούριο στιγμιότυπο (προφανώς εδώ αυτό δεν έχει ιδιαίτερη χρησιμότητα, σε άλλες εφαρμογές όμως με πολλαπλούς timers τα πράγματα είναι διαφορετικά). Παρέχει ένα απλό interface που αρχίζει αντίστροφη μέτρηση από κάποια τιμή και όταν τελειώσει επιστρέφει ένα event.

Το event αυτό το χρησιμοποιεί το *Counter* component, για να αυξήσει μια 16 bit τιμή, την οποία και στέλνει στα components *IntToLeds* και *IntToRfm*, μέσω του interface IntOutput, που επιτελεί ακριβώς αυτήν την εργασία. Το *IntToLeds* απλώς διαβάζει αυτή την τιμή και εμφανίζει τα τελευταία 3 bit στα LEDs του mica mote.

Το *IntToRfm* αντίθετα κάνει κάτι πολύ πιο ενδιαφέρον: στέλνει την τιμή που διαβάζει από τον Counter στο δίκτυο με ένα radio broadcast. Πρέπει όμως πρώτα να δούμε πώς γίνεται η επικοινωνία με μηνύματα μεταξύ των κόμβων.

΄Ενα μήνυμα στο TinyOS αντιπροσωπεύεται από μία δομή, την *TOS_Msg*, η οποία ορίζεται στο αρχείο *tos/system/AM.h*. Περιέχει πεδία για τη διεύθυνση του παραλήπτη, ένα πεδίο που ονομάζεται handlerID, ένα groupID, μήκος μηνύματος και payload, δηλαδή την πληροφορία που περιέχει το μήνυμα. Για τη διεύθυνση του παραλήπτη έχουμε τρεις επιλογές:

1. Να βάλουμε ως διεύθυνση το moteID ενός συγκεκριμένου κόμβου στον οποίο θέλουμε να μεταδώσουμε το μήνυμα. Το moteID προγραμματίζεται σε έναν κόμβο κατά τον προγραμματισμό της flash του, π.χ αν θέλουμε να εγκαταστήσουμε την εφαρμογή lala σε ένα κόμβο και να θέσουμε το moteID του ίσο με 13, εισάγουμε >make mica install.13

2. Να βάλουμε ως διεύθυνση την *TOS_BCAST_ADDR*, που σημαίνει broadcast στο δίκτυο οπότε το μήνυμα απευθύνεται σε όλους τους κόμβους.

3. Να βάλουμε τη διεύθυνση *TOS_UART_ADDR*, με άλλα λόγια μετάδοση από τη σειραϊκή θύρα.

Επίσης, υπάρχει και η global μεταβλητή *TOS_LOCAL_ADDR*, που είναι το moteID του κόμβου μας.

Το handlerID από την άλλη, είναι κάτι αντίστοιχο με τα TCP ports, δηλαδή όταν ένας κόμβος λάβει ένα μήνυμα, θα το προωθήσει στο component που δήλωσε ότι δέχεται μηνύματα με αυτό το handlerID. Δίνεται με αυτόν τον τρόπο η δυνατότητα να έχουμε ταυτόχρονα διάφορα είδη μηνυμάτων, τη διαχείριση των οποίων αναλαμβάνουν αυτόματα διαφορετικά μέρη μιας εφαρμογής TOS. Ας δούμε πώς γίνεται αυτό στον κώδικα για το configuration *IntToRfm*:

```
configuration IntToRfm
{
  provides interface IntOutput;
  provides interface StdControl;
}
```

```
implementation
{
  components IntToRfmM, GenericComm as Comm;

  IntOutput = IntToRfmM;
  StdControl = IntToRfmM;

  IntToRfmM.Send -> Comm.SendMsg[AM_INTMSG];
  IntToRfmM.StdControl -> Comm;
}
```

Το *GenericComm* είναι το component που αναλαμβάνει τη μετάδοση μηνυμάτων, είτε αυτή είναι μέσω radio είτε μέσω του serial port. Συνδέουμε το module IntToRfmM με το GenericComm μέσω του interface Send, το οποίο αρχικοποιείται ως instance με τον αριθμό AM_INTMSG, ο οποίος είναι μια σταθερά σε ένα include αρχείο, και είναι συγκεκριμένος για την εφαρμογή μας. Τα μηνύματα που στέλνονται μέσω του interface αυτού θα έχουν ως handlerID τον αριθμό AM_INTMSG.

Το groupID επιτρέπει σε επιλεγμένους κόμβους να "ακούν" το ίδιο μήνυμα, ουσιαστικά ένα broadcast σε περιορισμένο σύνολο κόμβων. Το groupID ενός κόμβου προγραμματίζεται θέτοντας την επιθυμητή τιμή στο Makefile της εφαρμογής. Π.χ αν θέλουμε να δώσουμε σε ένα mote groupID 13, εισάγουμε στο Makefile τη γραμμή DEFAULT_LOCAL_GROUP = 0x0D. Η default τιμή είναι 0x7D (δηλαδή 125).

Στο παρακάτω τμήμα κώδικα βλέπουμε πως σχηματίζουμε ένα μήνυμα προς μετάδοση. Χρησιμοποιούμε μια δομή IntMsg για το payload μέρος του μηνύματος, η οποία δομή απλά περιέχει μια τιμή του counter και το moteID του πομπού. Καλούμε έπειτα την εντολή send του interface.Send για να κάνουμε bradcast το μήνυμα.

```
 bool pending;
  struct TOS_Msg data;

  /* ... */

  command result_t IntOutput.output(uint16_t value) {
    IntMsg *message = (IntMsg *)data.data;

    if (!pending) {
      pending = TRUE;

      message->val = value;
      message->src = TOS_LOCAL_ADDRESS;

     if (call Send.send(TOS_BCAST_ADDR, sizeof(IntMsg), &data))
        return SUCCESS;
      pending = FALSE;
    }
    return FAIL;
  }
```

Η δεύτερη εφαρμογή που θα μας απασχολήσει σε αυτή την ενότητα είναι η *RfmToLeds*, που παίζει το ρόλο του δέκτη των μηνυμάτων που στέλνει ένα mote με την εφαρμογή *CntToRfmAndLeds*. Η λειτουργία που επιτελεί είναι ουσιαστικά η λήψη των μηνυμάτων αυτών και να τα στέλνει στο component που διαχειρίζεται τα μηνύματα με handlerID AM_INTMSG, συγκεκριμένα στο component RfmToInt. Αυτό με τη σειρά του, εξάγει από τα μηνύματα την τιμή του counter που περιέχουν και την προωθεί σε άλλο component, με το οποίο μπορούμε να εμφανίσουμε την τιμή αυτή στα LEDs του mote. Οπότε, προγραμματίζοντας ένα mote με την εφαρμογή CntToRfmAndLeds και ένα άλλο με την *RfmToLeds*, έχουμε αμέσως ασύρματη επικοινωνία!

΄Ενα ενδιαφέρον σημείο είναι η εμβέλεια (range) των motes. Αυτό που συμβαίνει είναι ότι το range είναι προγραμματιζόμενο και μάλιστα ο προγραμματισμός είναι αρκετά εύκολη υπόθεση (αν υπήρχε και το σχετικό documentation θα ήταν ακόμα πιο εύκολη...). Πάνω στο mica mote υπάρχει ένα ποτενσιόμετρο, το οποίο ρυθμίζει την τάση λειτουργίας του πομποδέκτη και μάλιστα μπορούμε να το προγραμματίσουμε από το TinyOS. Πιο συγκεκριμένα, υπάρχει ένα component, το *PotC*, το οποίο παρέχει το interface *Pot*, και μπορούμε να το συνδέσουμε στην εφαρμογή μας με κάποιο component και να αλλάζουμε την τιμή της αντίστασης του ποτενσιομέτρου.

Το ποτενσιόμετρο παίρνει τιμές από 0 έως 99, όπου το 0 αντιστοιχεί σε πολύ μικρή αντίσταση και επομένως μεγάλη τάση στον πομποδεκτή και μέγιστη εμβέλεια, και το 99 σημαίνει μεγάλη αντίσταση και μιρή εμβέλεια. Αν δεν αρχικοποιήσουμε εμείς το ποτενσιόμετρο, παίρνει μια default τιμή, η οποία είναι αρκετά μεγάλη και άρα έχουμε μικρή εμβέλεια (περίπου 1 πόδι = 30 εκατοστά). Αυτό που κάναμε εμείς είναι να συνδέσουμε το PotC στην εφαρμογή μας στο *IntToRfmM* (*tos/lib/IntToRfmM.nc*) μέσω του interface Pot, όπως φαίνεται στο παρακάτω τμήμα κώδικα.

```
includes IntMsg;

configuration IntToRfm
{
  provides {
    interface IntOutput;
    interface StdControl;
  }
}
implementation
{
  components IntToRfmM, GenericComm as Comm, PotC;

  IntOutput = IntToRfmM;
  StdControl = IntToRfmM;

  IntToRfmM.Send -> Comm.SendMsg[AM_INTMSG];
  IntToRfmM.SubControl -> Comm;
  IntToRfmM.Pot -> PotC;
}
```

Το δεύτερο που κάναμε ήταν στην υλοποίηση του *IntToRfm*, κατά τη διάρκεια της αρχικοποίησής του, δηλαδή όταν η Main χρησιμοποιήσει την εντολή init μέσω του StdControl, να αρχικοποιείται και το ποτενσιόμετρο, εδώ θέτοντας την τιμή του ίση με 50.

```
command result_t StdControl.init() {
    pending = FALSE;
    call Pot.init(50);
    return call SubControl.init();
  }
```

Παρατηρήσαμε ότι *η εμβέλεια ενός mote κυμαίνεται από περίπου 2 εκατοστά έως 1.5 μέτρο (5 πόδια)* στην καλύτερη περίπτωση, δηλαδή σε λεία επίπεδη επιφάνεια πάνω από το έδαφος, χωρίς εμπόδια μεταξύ των motes και χωρίς σημαντικές ηλεκτρομαγνητικές παρεμβολές στο χώρο. Αυτό πρακτικά σημαίνει ότι *η εξωτερική κεραία είναι απαραίτητη για να λειτουργήσουν ικανοποιητικά τα motes*. Μία άλλη παρατήρηση είναι ότι η μεταβολή της εμβέλειας του πομπού, καθώς μεταβάλλεται η αντίσταση του ποτενσιόμετρου, δεν είναι γραμμική.

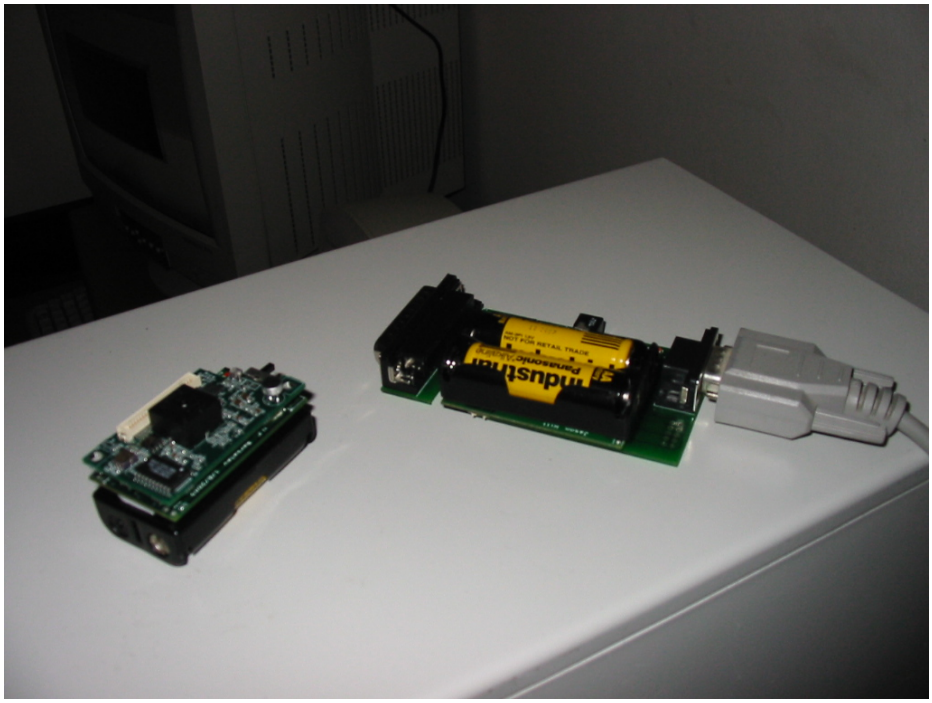
7.2.2 Μέτρηση διάρκειας ζωής ενός mote

Το επόμενο πείραμα που κάναμε ήταν να μετρήσουμε για πόσο χρόνο μπορεί να λειτουργήσει ένα mote σε συνθήκες φόρτου με δύο φορτισμένες μπαταρίες τύπου AA. Για το σκοπό αυτό χρησιμοποιήσαμε την εφαρμογή *OscillopeRF*, η οποία υπάρχει στη διανομή του TinyOS, την οποία τροποποιήσαμε έτσι ώστε να είναι ο μεγαλύτερος ο φόρτος που προκαλεί στο mote, και επομένως να καταναλώνει περισσότερη ενέργεια.

Η εφαρμογή αυτή χρησιμοποιεί τον αισθητήρα φωτός για να παίρνει δείγματα από το περιβάλλον και έπειτα τα κάνει broadcast στο υπόλοιπο δίκτυο μέσω radio μετάδοσης. Κάθε μέτρηση από τον αισθητήρα έχει ακρίβεια 10 bit, οπότε αποθηκεύεται σε 2 bytes. Όταν έρθουν 10 μετρήσεις, το mote στέλνει μήνυμα που περιέχει τις μετρήσεις αυτές. Η δειγματοληψία κανονικά γίνεται με συχνότητα 8 Hz, εμείς τη θέσαμε στα 32 Hz για να εξαντληθεί πιο γρήγορα η ενέργεια του κόμβου. Επίσης, με τη διαδικασία που περιγράψαμε στην προηγούμενη ενότητα θέσαμε στη μέγιστη τιμή της την εμβέλεια του mote. Να συνυπολογιστεί ότι παράλληλα ο κόμβος κάθε φορά που έστελνε μήνυμα άναβε και τα LEDs του. Αφού προγραμματίσαμε ένα mote με την εφαρμογή μας, το αφήσαμε ανοιχτό και περιμέναμε πότε θα σταματήσει να στέλνει μηνύματα.

Για τους σκοπούς της εφαρμογής αυτής, χρειάστηκε να χρησιμοποιήσουμε και την εφαρμογή *GenericBase*, η οποία περιέχεται στη διανομή του TinyOS 1.0 στον κατάλογο *TinyOS-1.x/apps/GenericBase*. Κατά τα γνωστά, κάνουμε compile και "φορτώνουμε" την εφαρμογή σε ένα mote, το οποίο αφήνουμε συνδεδεμένο με το programming board. Αποσυνδέουμε το board από την παράλληλη θύρα και το συνδέουμε στη σειραϊκή θύρα. Εμείς το συνδέσαμε με ένα καλώδιου μήκους περίπου 1.5 m, καλώδιο μεγαλύτερου μήκους δεν είναι σίγουρο ότι θα δουλέψει σωστά.

Αυτό που κάνει η *GenericBase* είναι να λαμβάνει πακέτα που γίνονται broadcast στο δίκτυο και να τα προωθεί μέσω της σειραϊκής θύρας του mote (UART) στο programming board, το οποίο όπως είπαμε είναι συνδεδεμένο στον υπολογιστή μας. Μέσω του SerialForward εάν θέλουμε μπορούμε να παρακολουθήσουμε τα πακέτα που έρχονται από τα motes. Η συνδεσμολογία που ακολουθήσαμε φαίνεται στο σχήμα 7.2.



Σχήμα 7.2: Συνδεσμολογία για τη μέτρηση της διάρκειας ζωής ενός mote.

Αφού λοιπόν εγκαταστήσαμε τις εφαρμογές μας, εφοδιάσαμε το mote με καινούριες μπαταρίες AA και άρχισε τη λειτουργία του. Το αποτέλεσμα ήταν ότι το mote κατάφερε να αντέξει *5 ολόκληρες μέρες*, χωρίς διακοπή, γεγονός αρκετά εντυπωσιακό από μόνο του. Ας προσπαθήσουμε να υπολογίσουμε τον όγκο δεδομένων που έστειλε το mote κατά τη διάρκεια της λειτουργίας του.

- Κάθε μέτρηση όπως είπαμε είναι 10 bit και αποθηκεύεται σε 2 bytes.
- Κάθε μήνυμα περιέχει 10 μετρήσεις, στα οποία έρχονται να προστεθούν τα 6 bytes του header, συνολικά δηλαδή κάθε μήνυμα έχει μέγεθος 26 bytes.
- ΄Εχουμε 32 μετρήσεις το δευτερόλεπτο, οπότε στέλνουμε 3.2 μηνύματα το δευτερόλεπτο.

- Σε 5 μέρες έχουμε στείλει 1382400 μηνύματα από 26 bytes το καθένα, που είναι συνολικά **34.27 Mbytes**!

Βέβαια, δεν μετρήσαμε όταν τέλειωναν τα αποθέματα ενέργειας του mote πόση ήταν η εμβέλειά του, η οποία πιθανότατα ήταν μειωμένη, οπότε ο υπολογισμός για τη διάρκεια ζωής του mote δεν είναι ιδιαίτερα ακριβής, αλλά *σε κάθε περίπτωση μεταφορά δεδομένων της τάξης των 30 Mbytes είναι άκρως εντυπωσιακή*. Τα πράγματα είναι ακόμα καλύτερα στα Mica2 motes, λόγω της διαφορετικής διαμόρφωσης που χρησιμοποιείται στην ασύρματη επικοινωνία και της μικρότερης κατανάλωσής τους συνολικά (επίσης, το range είναι αρκετά βελτιωμένο).

7.2.3 Μέτρηση θερμοκρασίας και μετάδοση πληροφορίας στο sink

Για την εφαρμογή αυτή χρησιμοποιήσαμε τις εφαρμογές *OscilloscopeRF* και *GenericBase* για motes και το Java applet *Oscilloscope*, το οποίο αναφέραμε στην αρχή του κεφαλαίου. Το *Oscilloscope*, συνδεέται στο port 9000 και επικοινωνεί με την εφαρμογή *SerialForward* για να διαβάσει τα πακέτα που στέλνουν τα motes που τρέχουν την εφαρμογή *OscilloscopeRF*. Τα πακέτα αυτά περιέχουν ένα header και 10 μετρήσεις από τους αισθητήρες φωτός των motes. Το *Oscilloscope* επεξεργάζεται αυτά τα πακέτα και εμφανίζει τα περιεχόμενά τους σε μια γραφική παράσταση.

Αντί για μέτρηση του φωτός, θελήσαμε να κάνουμε μέτρηση θερμοκρασίας χρησιμοποιώντας το thermistor των motes, οπότε έπρεπε να κάνουμε τις ανάλογες τροποποιήσεις στον ήδη υπάρχοντα κώδικα του *OscilloscopeRF*. Το βασικό σημείο που έπρεπε να προσέξουμε ήταν ότι *οι αισθητήρες φωτός και θερμοκρασίας μοιράζονται το ίδιο A/D channel στον επεξεργαστή*, δηλαδή οι έξοδοί τους χρησιμοποιούν τα ίδια pins για να συνδεθούν με τον επεξεργαστή του mote, όπου γίνεται η μετατροπή A/D. Επομένως, πρέπει με κάποιο τρόπο να κλείσουμε τον έναν από τους δύο αισθητήρες ώστε να λειτουργεί μόνο ένας.

Το component που αντιπροσωπεύει τον αισθητήρα θερμότητας στο TinyOS είναι το *Temp*, το οποίο θα πρέπει να συνδέσουμε στο configuration του *OscilloscopeRF*, μαζί με το component *Photo* (που είναι για τον αισθητήρα φωτός). Για πρακτικούς λόγους, ονομάζουμε την εφαρμογή μας από εδώ και πέρα *OscilloRFTemp*. Έτσι, θα έχουμε:

```
configuration Oscilloscope { }
implementation
{
  components Main, OscilloscopeM, ClockC, TimerC, LedsC,
   Photo, Temp, GenericComm as Comm, PotC;

  Main.StdControl -> OscilloscopeM;

  OscilloscopeM.Clock -> ClockC;
  OscilloscopeM.Leds -> LedsC;
```

```
  OscilloscopeM.Photo -> Photo.PhotoADC;
  OscilloscopeM.PhotoControl -> Photo.StdControl;
  OscilloscopeM.Temp -> Temp.TempADC;
  OscilloscopeM.TempControl -> Temp.StdControl;

  OscilloscopeM.TimerControl -> TimerC;
  OscilloscopeM.Timer -> TimerC.Timer[unique("Timer")];

  OscilloscopeM.CommControl -> Comm;
  OscilloscopeM.ResetCounterMsg ->
    Comm.ReceiveMsg[AM_OSCOPERESETMSG];
  OscilloscopeM.DataMsg -> Comm.SendMsg[AM_OSCOPEMSG];
  OscilloscopeM.Pot -> PotC;
}
```

Όπως βλέπουμε στον κώδικα, συνδέουμε τα δύο components με τα interface PhotoADC και TempADC, τα οποία είναι ουσιαστικά interface ADC. Ένα άλλο σημείο που πρέπει να προσέξουμε είναι ότι χρειάζεται κάποιο διάστημα μέχρι να δειγματοληπτήσει σωστά ο αισθητήρας θερμοκρασίας, το οποίο είναι περίπου 20 ms. Επομένως, όταν δώσουμε εντολή για δειγματοληψία μέσω του PhotoADC interface, θα πρέπει να περιμένουμε 20 ms ώστε να είναι σωστή η είσοδος στον A/D converter. Για αυτό το λόγο στην υλοποίηση του module χρησιμοποιούμε έναν επιπλέον timer σε σχέση με την αρχική υλοποίηση. Επίσης, όταν γίνεται αρχικοποίηση του συστήματος, κλείνουμε τον αισθητήρα φωτός για να μην μπερδεύονται οι δύο έξοδοι. Οι αλλαγές που κάναμε φαίνονται παρακάτω:

```
module OscilloscopeM
{
  provides interface StdControl;
  uses {
    interface Clock;
    interface Leds;

    interface StdControl as TempControl;
    interface StdControl as PhotoControl;
    interface StdControl as TimerControl;

    interface ADC as Photo;
    interface ADC as Temp;

    interface Timer;

    interface StdControl as CommControl;
    interface SendMsg as DataMsg;
    interface ReceiveMsg as ResetCounterMsg;
    interface Pot;
  }
}
...
  command result_t StdControl.init() {
```

```
    call Leds.init();
    call Leds.yellowOff();
    call Leds.redOff();
    call Leds.greenOff();

    call TempControl.init();
    call PhotoControl.init();
    call PhotoControl.stop();
    call CommControl.init();
    call Pot.init(0);
    call Clock.setRate(TOS_I5PS, TOS_S5PS);
    currentMsg = 0;
    packetReadingNumber = 0;
    readingNumber = 0;

    return SUCCESS;
  }

...

  task void waitForTemp() {
    if (call TempControl.start() == SUCCESS)
    call Timer.start(TIMER_ONE_SHOT, 20);
  }

  task void getTemp()
  {
    call Temp.getData();
  }

...

  event result_t Clock.fire() {
    post waitForTemp();
    return SUCCESS();
  }

  event result_t Timer.fired() {
    post getTemp();
    return SUCCESS;
  }
```

Πρακτικά αυτό που κάνει ο κώδικας είναι ότι κάθε φορά που έρχεται η ώρα να δειγματοληπτήσουμε, ανάβουμε τον αισθητήρα θερμοκρασίας και περιμένουμε 20 ms μέχρι να επεξεργαστούμε την έξοδό του. Παρατηρήσαμε ότι παρόλο που κλείνουμε τον αισθητήρα φωτός, όταν πέφτει φως πάνω στο mote επηρεάζεται η έξοδος του αισθητήρα θερμοκρασίας, οπότε τελικά λειτουργούν και οι δύο μαζί. Αυτό ίσως είναι κάποιο bug του TinyOS, μπορούμε απλά να απομονώσουμε τον αισθητήρα αυτόν καλύπτοντάς τον με κάτι ώστε να μην περνά καθόλου φως και δεν θα έχουμε πρόβλημα.

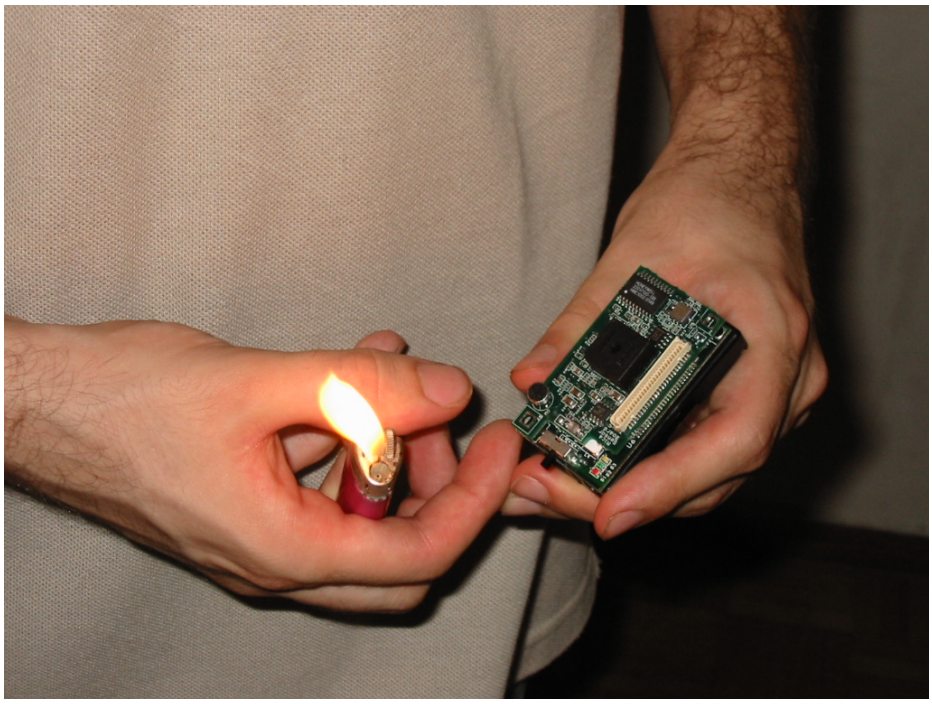
Τρέχουμε λοιπόν στο PC στο background τα δύο applets *SerialForward* και *Oscilloscope*, συνδέουμε στο programming board ένα mote που τρέχει το Gener-

icBase, συνδέουμε το board στη σειραϊκή θύρα όπως κάναμε στην προηγούμενη εφαρμογή και ανάβουμε το mote που έχουμε προγραμματίσει με το *OscilloRFTemp*. Στο πείραμα που κάναμε χρησιμοποιήσαμε έναν αναπτήρα για να ανεβάσουμε αρχικά τη θερμοκρασία απότομα, όπως φαίνεται στο σχήμα 7.3, και κατόπιν το βάλαμε δίπλα σε ένα κλιματιστικό, για να κατεβάσουμε απότομα τη θερμοκρασία. Η γραφική παράσταση που πήραμε φαίνεται στο σχήμα 7.4.

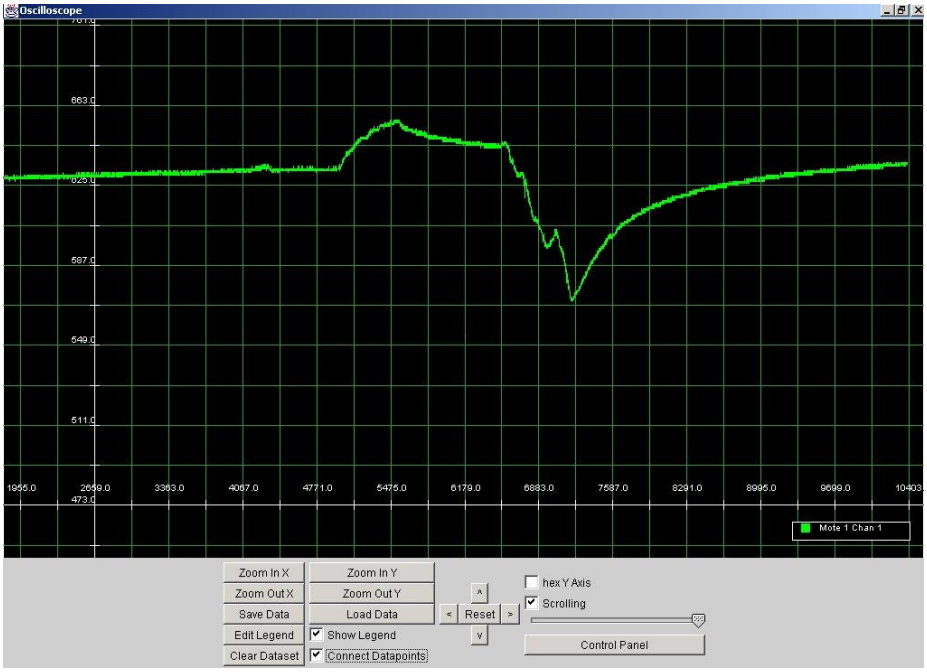
7.2.4 Η εφαρμογή TinyDB

Το TinyDB είναι η τελευταία εφαρμογή που θα μας απασχολήσει σε αυτήν την εργασία. Στην ενότητα αυτή γίνεται μια σύντομη εισαγωγή στο TinyDB και περιγράφονται κάποια απλά πειράματα, μέσω των οποίων διαφαίνονται οι μεγάλες δυνατότητες και ευκολίες που προσφέρει αυτή η εφαρμογή.

Το TinyDB γράφτηκε από μία ομάδα στο Πανεπιστήμιο της Καλιφόρνια στο Berkeley σε συνεργασία με την ομάδα του TinyOS. Η βασική σκέψη πίσω από τη δημιουργία του είναι να γίνει όσο το δυνατόν ευκολότερη η ζωή του προγραμματιστή που ασχολείται με δίκτυα έξυπνης σκόνης. Αφού όλες οι εφαρμογές κάνουν κάποιου είδους ανάκτηση πληροφορίας από τα motes του δικτύου, το λογικό θα ήταν να βρεθεί ένας τρόπος να αυτοματοποιηθεί αυτή η διαδικασία όσο πιο πολύ γίνεται. Το TinyDB κάνει ακριβώς αυτό: επιτρέπει στον προγραμματιστή να έχει πρόσβαση στην πληροφορία που περιέχεται στο δίκτυο



Σχήμα 7.3: Προκαλούμε απότομη άνοδο της θερμοκρασίας με ένα αναπτήρα



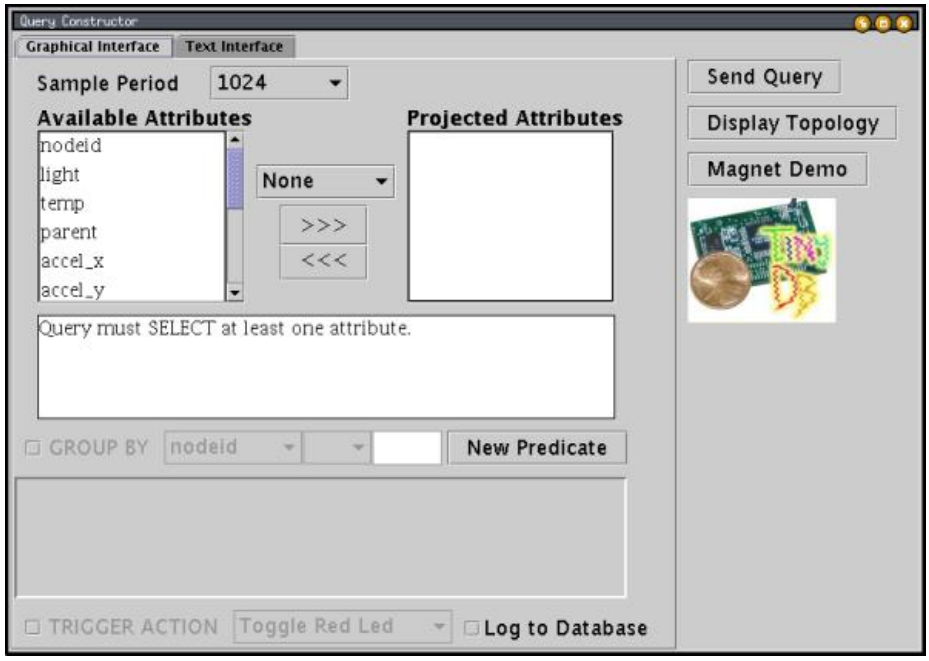
Σχήμα 7.4: Μέτρηση θερμοκρασίας όπως φαίνεται μέσα από την εφαρμογή Oscilloscope

μέσω ενός SQL-like interface, δήλαδή περίπου όπως σε μια βάση δεδομένων.

Ο προγραμματιστής μπορεί να καθορίσει παραμέτρους όπως πληροφορία προς ανάκτηση (θερμοκρασία, φως, κτλ), κόμβοι που στέλνουν πληροφορία, ρυθμός δειγματοληψίας, κτλ. Έτσι, η διαδικασία γίνεται πολύ πιο απλή και σύντομη. Δύο πολύ καλές πηγές για το TinyDB είναι το tutorial και το manual που περιέχονται στη διανομή του TinyOS 1.0. Υποθέτουμε από εδώ και πέρα ότι ο αναγνώστης έχει εγκαταστήσει το TinyOS (η διαδικασία περιγράφεται στο αντίστοιχο tutorial). Το GUI για το TinyDB ξεκινά με την κληση (είμαστε πάντα στον κατάλογο *TinyOS-1.x/tools/java)*:

```
> java net.tinyos.tinydb.TinyDBMain
```

Εμφανίζεται καταρχήν το παράθυρο του σχήματος 7.5, από το οποίο μπορούμε να ξεκινήσουμε να εργαζόμαστε με το TinyDB. Υπάρχει το κεντρικό panel, μέσω του οποίου μπορούμε να στείλουμε SQL queries στα motes του δικτύου, τα οποία όλα έχουν στη μνήμη τους την εφαρμογή TinyDB. Υπάρχει ένα mote συνδεδεμένο στο programming board, το οποίο τρέχει και αυτό TinyDB, και το οποίο είναι προγραμματισμένο με MoteID 0 (θεωρούμε ότι το mote που είναι συνδεδεμένο στο board πρέπει να έχει ID 0). Τα queries που δίνουμε μέσω του κεντρικού panel γίνονται broadcast στο δίκτυο μέσω αυτού του mote, και διαδίδονται στη συνέχεια από τα ίδια τα motes. Δεν θεωρήσαμε σκόπιμο να

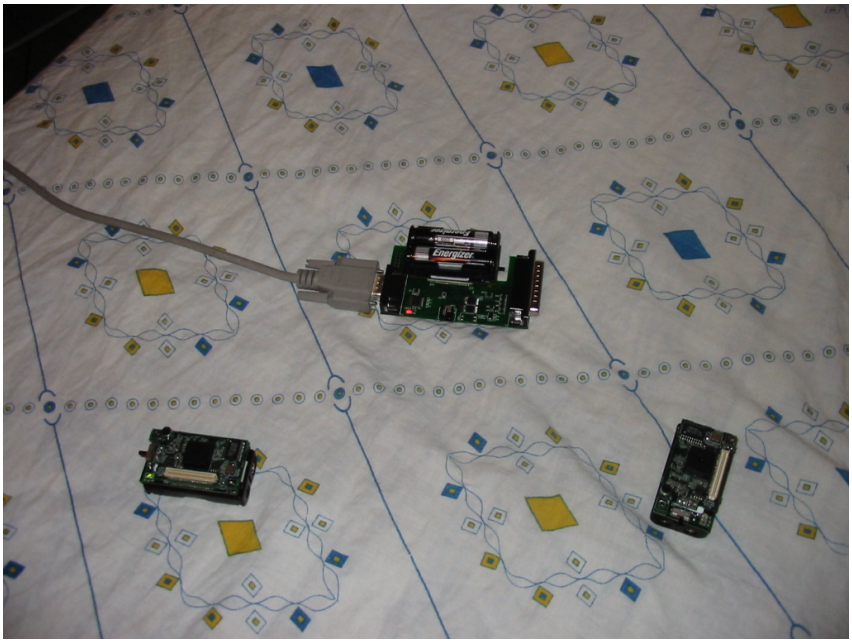


Σχήμα 7.5: Το βασικό παράθυρο της εφαρμογής TinyDB.

δείξουμε πώς μπορεί κάποιος να στείλει queries στο δίκτυο, καθώς το σχετικό tutorial καλύπτει πολύ καλά το συγκεκριμένο θέμα.

Μια πολύ ενδιαφέρουσα και ομολογουμένως εντυπωσιακή δυνατότητα του TinyDB είναι η δύνατότητα οπτικοποίησης της τοπολογίας του δίκτύου έξυπνης σκόνης στο οποίο είμαστε συνδεδεμένοι, με παράλληλη εμφάνιση στην οθόνη συμβάντων μέσα στο δίκτυο. Αφού συνδέσουμε κατα τα γνωστά το programming board στη σειραϊκή και σε αυτό το mote με ID 0, τοποθετούμε και άλλα motes μέσα στην εμβέλεια του πρώτου. Για τις ανάγκες του παραδείγματος, χρησιμοποιήσαμε άλλα δύο motes και είχαμε την απλή τοπολογία που φαίνεται στο σχήμα 7.6. Για να δούμε την τοπολογία μέσα από το TinyDB επιλέγουμε Diplay Topology από το κεντρικό παράθυρο της εφαρμογής. Το αποτέλεσμα φαίνεται στο σχήμα 7.7, όπου το PC μας εικονίζεται ως laptop και οι κόμβοι του δικτύου ως πράσινοι κύκλοι, ενώ οι ακμές δηλώνουν τις συνδέσεις του δικτύου.

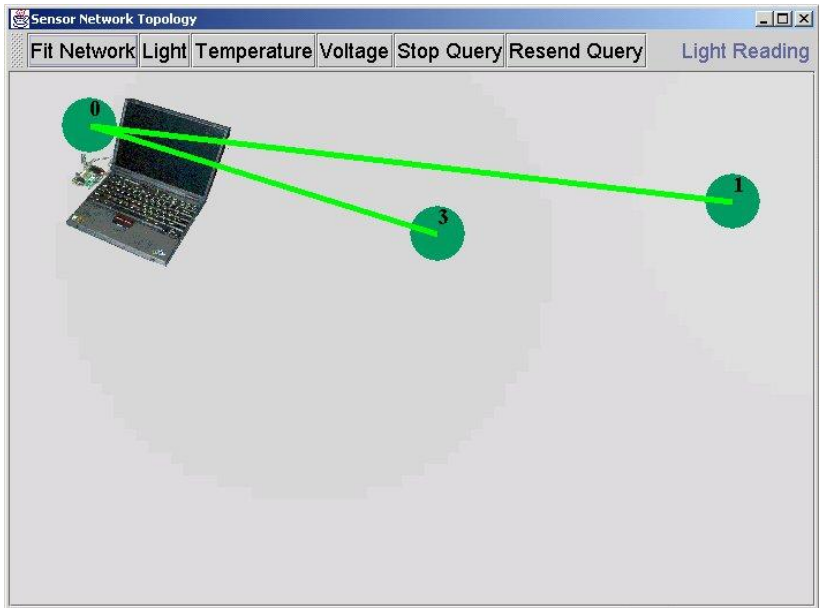
Προφανώς, η απεικόνιση της θέσης των motes εξαρτάται μόνο από την απόστασή τους από τον βασικό σταθμό, και δεν ανταποκρίνεται στην πραγματική τους θέση. ΄Ενα άλλο ενδιαφέρον χαρακτηριστικό του TinyDB είναι ότι μπορεί να χτίσει ένα δένδρο δρομολόγησης από μόνο του (χρησιμοποιεί κάποιο κλασσικό ad hoc πρωτόκολλο). Στην πράξη είδαμε ότι αυτό γίνεται με κάποια δυσκολία και πάντως γίνεται με μια σχετική καθυστέρηση. Είχαμε αρχικά την τοπολογία που φαίνεται στο σχήμα 7.8, με ένα mote μέσα στην εμβέλεια του βασικού σταθμού και ένα άλλο έξω από αυτή, αλλά μέσα στην εμβέλεια



Σχήμα 7.6: Μια απλή τοπολογία με τρία motes που τρέχουν TinyDB.

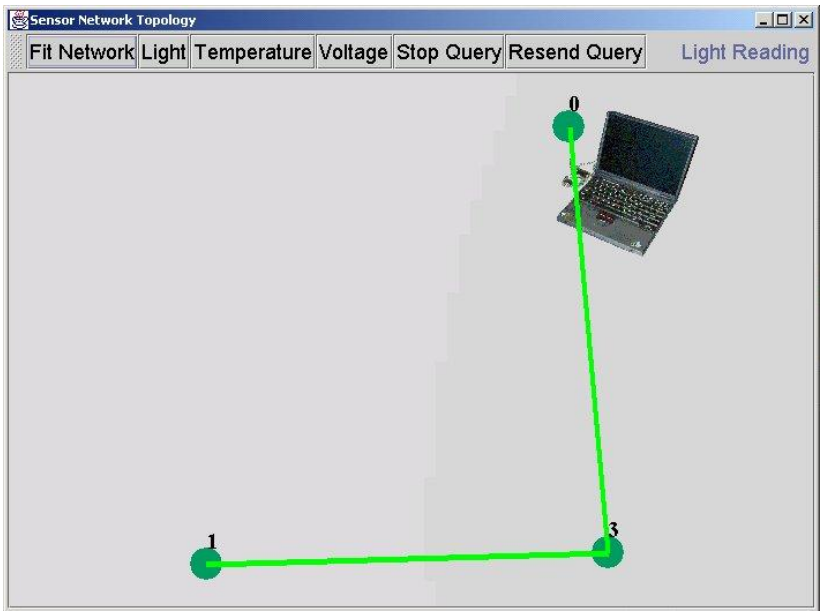
του δεύτερου mote. Μετά από λίγο το TinyDB έχτισε το δίκτυο που φαίνεται στο σχήμα 7.9, στο οποίο φαίνεται ξεκάθαρα ότι γίνεται multihop μετάδοση μηνυμάτων.

Το επόμενο που κάναμε ήταν να προκαλέσουμε ένα συμβάν μέσα στο δίκτυο και να δούμε πώς αυτό θα καταγραφεί από την εφαρμογή. Συγκεκριμένα, επιστρατεύσαμε πάλι έναν αναπτήρα και τον ανάψαμε δίπλα σε ένα mote, ώστε αυτός να καταγράψει την αλλαγή θερμοκρασίας, όπως φαίνεται στο σχήμα 7.10. Η αντίδραση ήταν πολύ εντυπωσιακή και άμεση, όπως καταγράφεται στο σχήμα 7.11. Εκτός της θερμοκρασίας, μπορούν να απεικονιστουν συμβάντα σχετικά με το φως και τη διαθέσιμη ενέργεια των κόμβων του δικτύου.

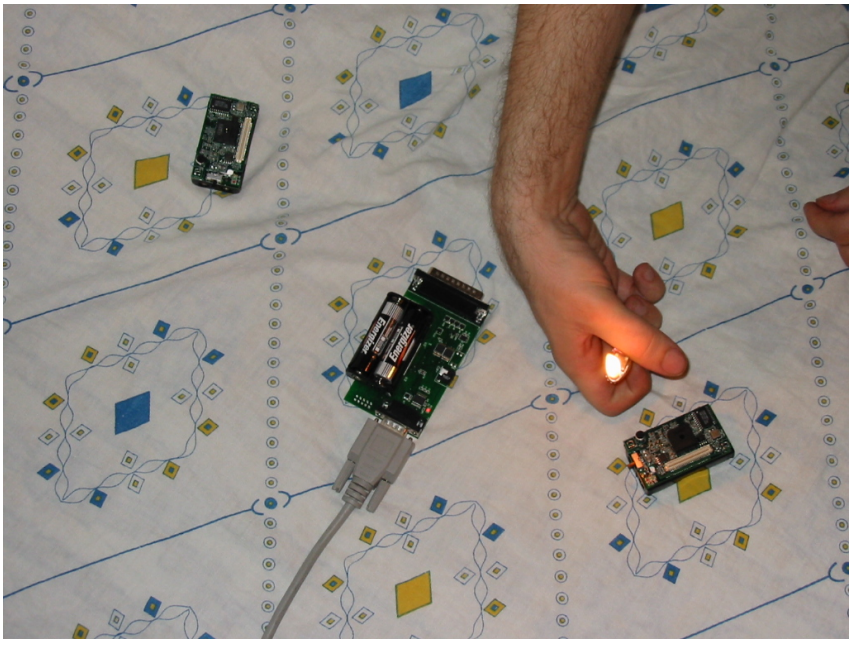


Σχήμα 7.7: Η ίδια τοπολογία μέσα από την εφαρμογή.

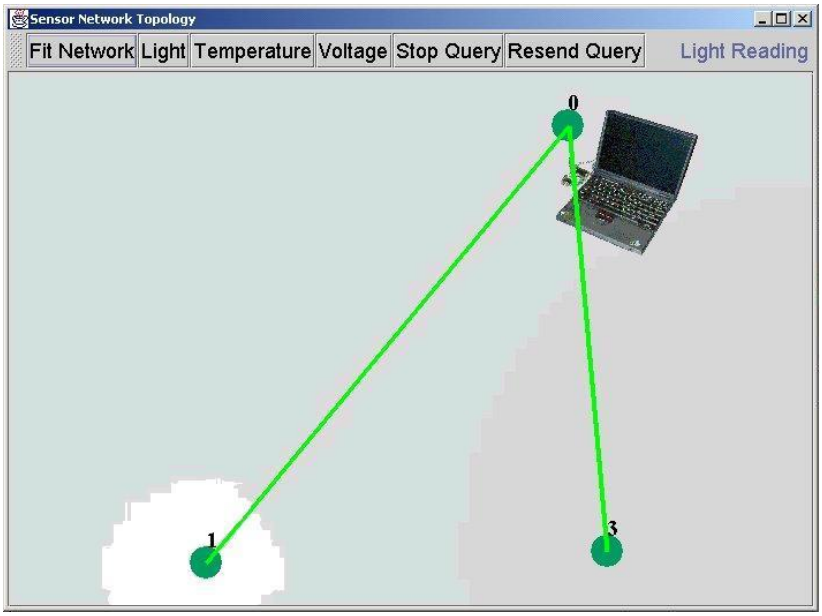
Σχήμα 7.8: Τοπολογία όπου μόνο ένας κόμβος βρίσκεται μέσα στην εμβέλεια του sink.



Σχήμα 7.9: Multihop τοπολογία όπως φαίνεται μέσα από την εφαρμογή TinyDB.



Σχήμα 7.10: Προκαλούμε ένα συμβάν μέσα στο δίκτυο.



Σχήμα 7.11: Καταγραφή της ανόδου της θερμόκρασίας στο δίκτυο (όσο πιο ανοιχτό χρώμα, τόσο πιο αυξημένη θερμοκρασία).

Βιβλιογραφία

[1] 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.

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

[3] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, J. Anderson, Wireless Sensor Networks for Habitat Monitoring, in *WSNA'02*, September 28, 2002, Atlanta, Georgia.

[4] L. Doherty, B.A. Warneke, B.E. Boser and K.S.J. Pister, Energy and performance considerations for smart dust, in the *International Journal of Parallel and Distributed systems and Networks, Vol. 4*, No. 3, 2001.

[5] United Press International, Dust-sized sensors could monitor weather, *www.upi.com/view.cfm?StoryID=20021030-031519-2481r*, October 30th, 2002.

[6] Ivy, a sensor network infrastructure for the college of Engineering, *http://www-bsac.eecs.berkeley.edu/projects/ivy/*

[7] S. Tillak, N.B. Abu-Ghazaleh and W. Heinzelman, A taxonomy of Wireless Micro-Sensor Network Models in the *Mobile Computing and communications review, Volume 1, Number 2*.

[8] W. R. Heinzelman, A. Chandrakasan and H. Balakrishnan: Energy-Efficient Communication Protocol for Wireless Microsensor Networks. In *Proc. 33rd Hawaii International Conference on System Sciences* -- HICSS'2000.

[9] A. Manjeshwar and D.P. Agrawal: TEEN: A Routing Protocol for Enhanced Efficiency in Wireless Sensor Networks. In *Proc. 2nd International Workshop on Parallel and Distributed Computing Issues in Wireless Networks and Mobile Computing*, satellite workshop of *16th Annual International Parallel & Distributed Processing Symposium* -- IPDPS'02.

[10] 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.

[11] C. Schurgers, V. Tsiatsis, S. Ganeriwal and M. Srivastava: Topology Management for Sensor Networks: Exploiting Latency and Density. In *MOBIHOC'02, June 2002, Lausanne, Switzerland.*

[12] I. Chatzigiannakis and Sotiris Nikoletseas: A Sleep-Awake Protocol for Information Propagation in Smart Dust Networks. In *Proc. 3rd Workshop on Mobile and Ad-Hoc Networks (WMAN), IPDPS Workshops, IEEE Press, 2003.*

[13] I. Chatzigiannakis, S. Nikoletseas and P. Spirakis: Smart Dust Protocols for Local Detection and Propagation. In *Proc. 2nd ACM International Workshop on Principles of Mobile Computing* -- POMC'2002. Also FLAGS Technical Report, FLAGS-TR-1.

[14] D. Estrin, R. Govindan, J. Heidemann and S. Kumar: Next Century Challenges: Scalable Coordination in Sensor Networks. In *Proc. 5th ACM/IEEE International Conference on Mobile Computing* -- MOBICOM'1999.

[15] C. Intanagonwiwat, R. Govindan and D. Estrin: Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. In *Proc. 6th ACM/IEEE International Conference on Mobile Computing* -- MOBICOM'2000.

[16] C. Intanagonwiwat, D. Estrin, R. Govindan and J. Heidemann: Impact of Network Density on Data Aggregation in Wireless Sensor Networks. *Technical Report 01-750, University of Southern California Computer Science Department, November, 2001.*

[17] S. Nikoletseas, I. Chatzigiannakis, A. Antoniou, G. Mylonas, H. Euthimiou, A. Kinalis: Energy efficient protocols for sensing multiple events in smart dust networks, CTI Technical Report, July 2003.

[18] I. Chatzigiannakis, T. Dimitriou, M. Mavronicolas, S. Nikoletseas and P. Spirakis: A Comparative Study of Protocols for Efficient Data Propagation in Smart Dust Networks, Distinguished Paper, in the *Proceedings of the International Conference on Parallel and Distributed Computing, (EUROPAR 2003)*.

[19] J.M. Kahn, R.H. Katz and K.S.J. Pister: Next Century Challenges: Mobile Networking for Smart Dust. In *Proc. 5th ACM/IEEE International Conference on Mobile Computing*, pp. 271-278, September 1999.

[20] The Smart Dust Project, *http://robotics.eecs.berkeley.edu/∼pister/SmartDust*.

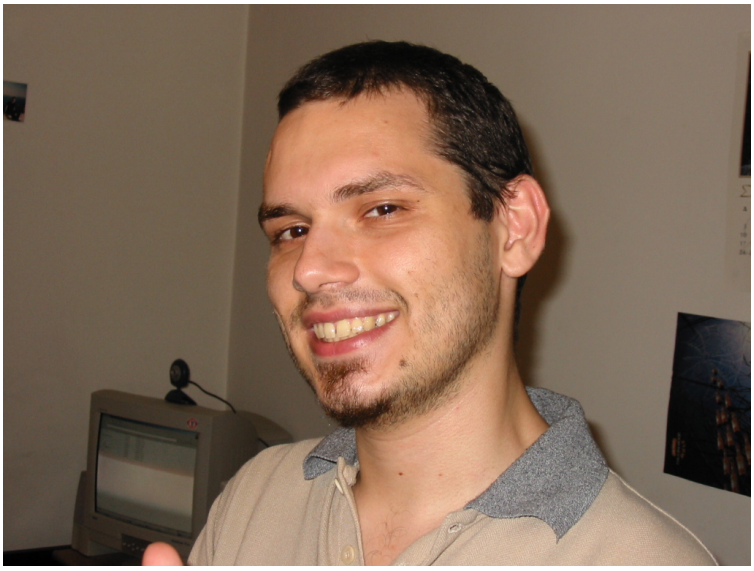
[21] TinyOS: A Component-based OS for the Network Sensor Regime. http://webs.cs.berkeley.edu/tos/, October, 2002.

[22] The Crossbow company, http://www.xbow.com.

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

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

[25] 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.



Σχήμα 7.12: Bonus Picture: Επιτέλους τελείωσε!