ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ Η/Υ ΚΑΙ ΠΛΗΡΟΦΟΡΙΚΗΣ ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ ΠΑΤΡΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Σχεδιασμός και μελέτη κατανεμημένων αλγορίθμων επικοινωνίας για ασύρματα δίκτυα αισθητήρων και πειραματική αξιολόγηση στην πλατφόρμα SUN SPOT

Κονίνης Χρήστος , ΑΜ 2654

Επιβλέπων: Λέκτορας Νικολετσέας Σωτήρης

Συνεπιβλέπων: Χατζηγιαννάκης Ιωάννης

Οκτώβριος 2008

Ευχαριστίες

Θα ήθελα πρώτα από όλους να ευχαριστήσω τον Ιωάννη Χατζηγιαννάκη, οποίος έκανε τη συνεπίβλεψη αυτής της διπλωματικής, για την πολύτιμη καθοδήγησή του και τις συμβουλές του καθόλη τη διάρκεια εκπόνησής της. Επίσης θα ήθελα να ευχαριστήσω τον Γρηγόριο Πράσινο με τον οποίο συνεργάστηκα στο τελευταίο τμήμα της διπλωματικής, η συμβολή του υπήρξε καθοριστική για την επιτυχή ολοκλήρωσή της.

Θα ήθελα ακόμα να ευχαριστήσω τον Σωτήρη Νικολετσέα, Λέκτορα του Τμήματος Μηχανικών Η/Υ και Πληροφορικής, για την ευκαιρία και την εμπιστοσύνη που μου έδειξε με την ανάθεση αυτής της διπλωματικής εργασίας. Τον ευχαριστώ για τις πολύτιμες συμβουλές του, και τις καίριες υποδείξεις και προτροπές που με βοήθησαν να στην επίτευξη των στόχων μου.

Τέλος θα ήθελα ευχαριστήσω τους γονείς μου για την συμπαράστασή και την υποστήριξη που μου δίνουν σε όλες τις επιλογές της ζωής μου.

When a distinguished but elderly scientist states that something is possible, he is almost certainly right. When he states that something is impossible, he is very probably wrong.

-Arthur C. Clarke

All sorts of computer errors are now turning up. You'd be surprised to know the number of doctors who claim they are treating pregnant men.

-Isaac Asimov

Εισαγωγή στα ασύρματα δίκτυα αισθητήρων

Τα ασύρματα δίκτυα αισθητήρων(WSNs) είναι δίκτυα που αποτελούνται από μια συλλογή από μικρές, χαμηλής κατανάλωσης, και περιορισμένης επεξεργαστικής ισχύς συσκευές που διασκορπίζονται σε μια περιοχή και έχουν την δυνατότητα να εποπτεύουν το περιβάλλον , συλλέγοντας πληροφορίες μέσω ενσωματωμένων αισθητήρων. Οι συσκευές αυτές συνεργάζονται μεταξύ τους, επικοινωνώντας ασύρματα για την μετάδοση και συλλογή των παραπάνω πληροφοριών. Τα WSNs έχουν μια μεγάλη γκάμα εφαρμογών, όπως ανίχνευση φυσικών καταστροφών(πχ πυρκαγιές, σεισμοί, πλημμύρες κ.α.) σε μια περιοχή, έλεγχος κίνησης σε δρόμους και σε υποδομές για “έξυπνα” σπίτια και συστήματα ασφάλειας. Λόγο της πληθώρας των πρακτικών εφαρμογών, τα WSN παρουσιάζουν μεγάλο ερευνητικό ενδιαφέρον, συγκεκριμένα σύγχρονα ζητήματα έρευνας προσπαθούν να δώσουν απαντήσεις τόσο σε προβλήματα σε επίπεδο hardware όσο και σε επίπεδο software. Η πρώτη γενιά WSN-hardware ήταν ενεργοβόρα με χαμηλή υπολογιστική ικανότητα και περιορισμένο ρυθμό ασύρματης μετάδοσης δεδομένων. Η δεύτερη και τρίτη γενιά συσκευών έχει κατά μεγάλο βαθμό ωριμάσει, προσφέροντας γρήγορη και αξιόπιστη ασύρματη επικοινωνία και αξιοποιώντας νέους πιο γρήγορους, μικροελεγκτές. Επίσης ένα μεγάλο μέρος της προόδου που έχει γίνει τα τελευταία χρόνια οφείλεται στην καθιέρωση standard (όπως το LoWPAN[1] από την IETF και το zigbee[2]) που έχουν οδηγήσει στην δημιουργία ενός κοινού πλαισίου για τον σχεδιασμό συσκευών, ξεφεύγοντας από τις κλειστές ιδιωτικές λύσεις που χαρακτήριζαν την πρώτη γενιά . Αυτήν την στιγμή στην αγορά υπάρχει μια μεγάλη ποικιλία από WSN πλατφόρμες για ερευνητική αλλά και για εμπορική χρήση. Κάθε μια ακολουθεί διαφορετική προσέγγιση για να δώσει λύσεις στα προβλήματα των WSN δικτύων. Η Crossbow προσφέρει αρκετές WSN πλατφόρμες βασισμένες κυρίως στον Atmel AtMega 128 με IEEE 802.15.4 radio(MICAz [3], IRIS [4]) που μπορούν να συνδυαστούν με διαφορετικά sensor board. Επίσης προσφέρει συσκευές με υψηλή υπολογιστική ισχύ και γρήγορο radio, όπως το imote2 που βασίζεται στον επεξεργαστή Xscale. Η ETH Zurich διαθέτει *τα* Btnodes[5] που απευθύνονται κυρίως σε ερευνητές, είναι βασισμένα σε ένα AtMega ελεγκτή και διαθέτουν δυο radio. Η ScatterWeb GmbH έχει αναπτύξει μια μεγάλη ποικιλία από WSN-συσκευές, όπως τα ScatterNodes [6] και το MSB430 [7]. Η coalesenses GmbH έχει αναπτύξει την πλατφόρμα iSense [8]. Το iSense είναι η πρώτη συσκευή που βασίζεται σε ένα chip που περιλαμβάνει και το radio(IEEE 802.15.4) και έναν 32-bit ελεγκτή, και μπορεί να επεκταθεί προσθέτοντας sensor/interface boards. Πρόσφατα η Sun Microsystems ανέπτυξε την πλατφόρμα SPOT [9] προσπαθώντας να αντιμετωπίσει πολλές από τις προκλήσεις που παρουσιάζονται στην ανάπτυξη μικρών συσκευών με αισθητήρες όπως η δυσκολία προγραμματισμού και αποσφαλμάτωσης εφαρμογών για WSN συσκευές. Με αυτή την καινούργια πλατφόρμα της SUN ασχοληθήκαμε σε αυτήν την διπλωματική, υλοποιώντας νέα πρωτόκολλα δρομολόγησης που δεν υπήρχαν για αυτήν την πλατφόρμα και επεκτείνοντας την λειτουργικότητά του εξομοιωτή των SUN SPOTs.

Παραδείγματα WSN συσκευών καθώς και τα σχετικά πλεονεκτήματα / μειονεκτήματα τους παρουσιάζονται στον παρακάτω πίνακα μαζί με τα βασικά χαρακτηριστικά τους:

WSN nodes	**Γλώσσα προγραμμα-τισμού**	**Λειτουργικό σύστημα**	**Πλεονεκτήματα πλατφόρμας**	**Μειονεκτήματα πλατφόρμας**
Sun SPOT	Java	Java Squawk VM	Υψηλή υπολογιστική ισχύ, ευκολία προγραμματισμού	Υψηλή τιμή, μέτρια αυτονομία
BTnode rev 3	C, AVR-GCC TOOL	TinyOS	2 radio(Bluetooth και 433-915 MHz radio)	Υψηλή τιμή, μεγάλο βάρος συσκευής, δεν παρέχεται υποστήριξη
SHIMMER	NesC	TinyOS	Προερετικό 2ο radio(Bluetooth) και επεκτάσεις	Υψηλή τιμή, περιορισμένη διαθεσιμότητα
Firefly	C	Nano-RK	Πολύ μεγάλη αυτονομία	Αρκετά βαριά συσκευή (λόγο δυο AA μπαταριών)
eZ430-RF2500	C, IAR Kickstart	None	Μικρό μέγεθος, μεγάλη αυτονομία	Λίγη μνήμη και περιορισμένη επεξεργαστική ισχύ
MICAz	nesC	TinyOS	Χαμηλή τιμή, μεγάλη αυτονομία	Μικρή εμβέλεια radio και περιορισμένη επεξεργαστική ισχύ
Sensinode	NanoStack written in C	None	IP based	Λόγο απουσίας OS η εγκατάσταση των εφαρμογών γίνεται ξανα περνόντας το firmware
Sentilla	Java	None	Μικρό μέγεθος	Λόγο απουσίας OS η εγκατάσταση των εφαρμογών γίνεται ξανα περνόντας το firmware

1. Εισαγωγή στην πλατφόρμα SUNSPOT

Τα SUN SPOT (Small Programable Object Technology) είναι μικρές ασύρματες συσκευές με ενσωματωμένους αισθητήρες που λειτουργούν με μπαταρίες και είναι προγραμματιζόμενες σχεδόν εξολοκλήρου σε Java δίνοντας την δυνατότητα σε απλούς προγραμματιστές να αναπτύξουν εύκολα λογισμικό χωρίς να έχουν εξειδικευμένες γνώσεις σε embedded συστήματα. Επίσης περιλαμβάνει αρκετούς ενσωματωμένους αισθητήρες και την δυνατότητα να διασυνδεθεί με εξωτερικές συσκευές.

Τα SUN SPOTs χρησιμοποιούν μια υλοποίηση της Java ME που λέγεται Squawk [10] και υποστηρίζει CLDC 1.1 και MIDP 1.0. Τα SPOTs δεν έχουν κάποιο λειτουργικό σύστημα, αλλά τρέχει την Squawk VM απευθείας πάνω στον επεξεργαστή, και η VM παρέχει τις βασικότερες λειτουργίες ενός OS, επίσης όλοι οι drivers των συσκευών είναι γραμμένοι σε Java.

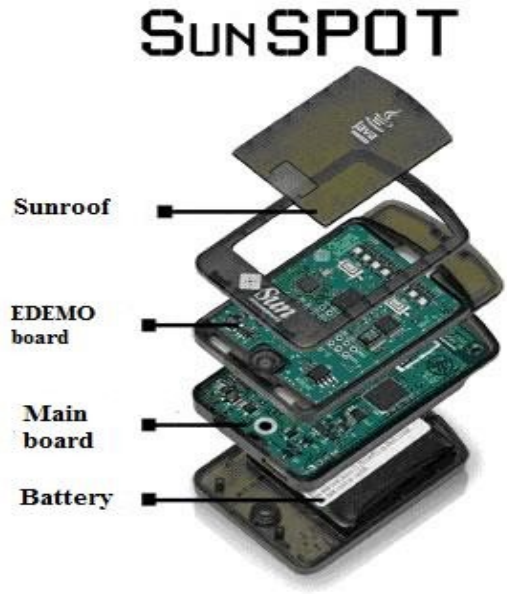
Το αναπτυξιακό σύστημα που δίνεται από την SUN και με το όποιο ασχοληθήκαμε σε αυτή την διπλωματική είναι η έκδοση v3(Purple) και περιλαμβάνει δύο eSPOTS ένα basestation, ένα USB καλώδιο για την σύνδεση των SPOTs/basestation στο υπολογιστή και ένα cd με το sdk για την ανάπτυξη και εγκατάσταση εφαρμογών στα SPOTS.

Πιο συγκεκριμένα:

- eSPOT - Πρόκειται για την τωρινή έκδοση του SUN SPOT και αποτελείται από ένα κύριο board με μπαταρίες λιθίου, επεξεργαστή, μνήμη, 802.15.4 radio και σύνδεσμο για προσθήκη κάρτας επέκτασης. Στην συγκεκριμένη έκδοση τα SPOTs έχουν μια κάρτα επέκτασης το eDEMO με επιταχυνσιόμετρο, μετατροπέα ADC, ψηφιακές εισόδους/εξόδους(GPIO), 2 κουμπιά και 8 led.
- Basestation – Το basestation είναι μια συσκευή που περιέχει το κύριο board του eSPOT χωρίς μπαταρίες και κάρτα επέκτασης. Η τροφοδοσία παρέχεται από ένα USB καλώδιο που συνδέεται με ένα υπολογιστή. Το basestation χρησιμοποιείται για να επικοινωνούν εφαρμογές που τρέχουν σε ένα υπολογιστή με τα SPOTS.

1.1 Τεχνικές προδιαγραφές των SPOTs

Όπως προαναφέραμε η τωρινή διαμόρφωση των SPOTs όπως φαίνεται στην παρακάτω εικόνα περιλαμβάνει το Main Board (cpu/radio) και το eDEMO Board που περιέχει τους αισθητήρες. Σε αυτό το κεφάλαιο θα αναφέρουμε τα βασικά στοιχεία του hardware που χρησιμοποιούνται και μια συνοπτική περιγραφή των χαρακτηριστικών τους. Επίσης θα αναλύσουμε τα σημαντικά υποσυστήματα της πλατφόρμας και την λειτουργικότητα του Sensor Board[11].



1.1.1 Στοιχεία του SPOT Main Board

Επεξεργαστής

Πρόκειται για τον ARM920T ARM Thumb processor της ATMEL που περιέχεται σε ένα SOC (System On Chip) κύκλωμα το AT91RM9200. Σε κανονική λειτουργία καταναλώνει 44mW και η μέγιστη ταχύτητα του ρολογιού φτάνει τα 180MHz. Το SOC ενσωματώνει 16Kbyte cache εντολών, και 64-way associative 16Kbyte cache δεδομένων. Η MMU (ARMv4) έχει ένα TLB buffer 64 στοιχείων για δεδομένα και άλλον ένα TLB 64 στοιχείων για μετάφραση εντολών. Η πρόσβαση στην εξωτερική μνήμη(flash, pSRAM) γίνεται

από το EBI δίαυλο, ο ελεγκτής του διαύλου είναι ρυθμισμένος ώστε να να εκκινεί το σύστημα(διαδικασία boot) από την flash όπου βρίσκεται η Squawk VM. Επίσης το SOC περιλαμβάνει μια μεγάλη συλλογή από interfaces για περιφερειακές συσκευές όπως θύρες USB host/devive , ethernet MAC, προγραμματιζόμενος ελεγκτής I/O (PIO), ελεγκτές SPI/USART/I2C/I2S, και 3 16-bit χρονιστές/μετρητές. Επιπλέον ενσωματώνεται και ένας DMA controller (PDC) για άμεσες και γρήγορες εγγραφές στην μνήμη και στους διαύλους USART/I2S/SPI. Λόγο του μικρού μεγέθους της συσκευής οι USB host και η μια USART θύρες δεν χρησιμοποιούνται όπως και τα TWI/I2S/Ethernet MAC interfaces. Επειδή όμως όλα τα σήματα υπάρχουν στο βύσμα του main board που το διασυνδέει με την κάρτα επέκτασης(eDEMO board), μπορούμε να χρησιμοποιήσουμε τα παραπάνω interfaces αν προσθέσουμε τις κατάλληλες φυσικές διασυνδέσεις και γράψουμε τους αντίστοιχους drivers.

Δίαυλοι επικοινωνίας του Main Board

Η επικοινωνία μεταξύ των SPOTs και workstation γίνεται κυρίως μέσω του διαύλου USB και για την διασύνδεση υπάρχει μια υποδοχή mini USB τύπου B. Η USB client συσκευή στα SPOTs είναι συμβατή με τα πρότυπα USB 1.1 και USB 2.0 και υποστηρίζει ACM modem για την σειριακή μετάδοση. Για την επικοινωνία ανάμεσα σε εσωτερικές συσκευές του main board και μεταξύ του main board και του eDEMO board(κάρτα επέκτασης/αισθητήρων) χρησιμοποιείται το SPI και το PIO. Το SPI είναι ένας σειριακός δίαυλος για την επικοινωνία με τον ασύρματο πομποδέκτη IC CC2420, τον power controller και τον έλεγχο των LEDs του eDEMO board. Το PIO interface ελέγχει το activity LED που βρίσκεται αριστερά της mini USB υποδοχής καθώς και τα σήματα ελέγχου και κατάστασης του ασύρματου πομποδέκτη, όπως για παράδειγμα ότι το κανάλι είναι ελεύθερο για μετάδοση ή ότι η RX ουρά είναι πλήρης. Τέλος μέσω του PIO μεταφέρονται τα σήματα ελέγχου του κυκλώματος που ρυθμίζει την τροφοδοσία ρεύματος στην USB θύρα.

Μνήμη

Η μνήμη στο Main Board είναι η Spansion S71PL032J40, και αποτελείται από 4Mbyte NOR flash και 512Kbyte pSRAM(pseudo-SRAM) που βρίσκονται στο ίδιο chip. Ο χρόνος πρόσβασης(access time) για την pSRAM είναι 70nsec και για την Flash 65nsec και έχουν 16- bit δίαυλο δεδομένων. Και οι δυο χρησιμοποιούν τροφοδοσία 3Volt και σε κανονικές συνθήκες λειτουργίας η κατανάλωση είναι 25ma για την pSRAM και 22ma για την Flash. Τα δεδομένα στην pSRAM διατηρούνται όσο το SPOT είναι συνδεδεμένο σε κάποια τροφοδοσία ή μπαταρία. Όταν στο SPOT είναι σε κατάσταση deep-sleep, που τα περισσότερα υποσυστήματα δεν τροφοδοτούνται για εξοικονόμηση ενέργειας η pSRAM καταναλώνει περίπου 8μA για την διατήρηση των δεδομένων της ενώ η flash απενεργοποιείται. Η flash είναι προγραμματισμένη ήδη από το εργοστάσιο και περιέχει τον bootloader, την Squawk VM, τις βασικές βιβλιοθήκες και μια προ εγκατεστημένη εφαρμογή (bounce demo).

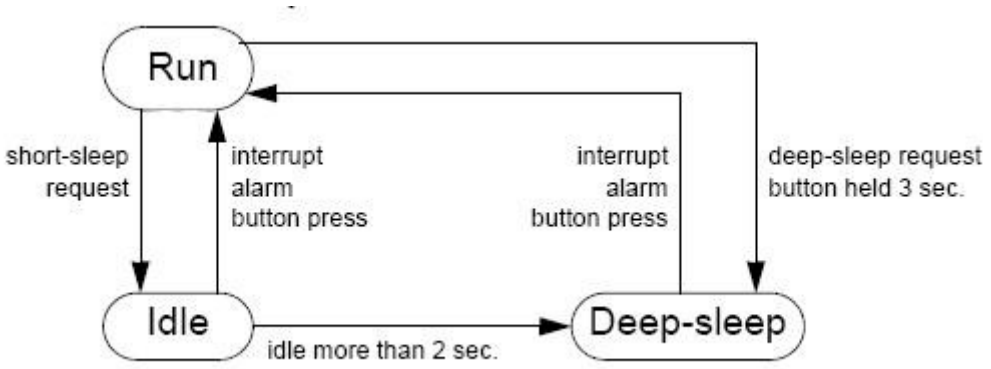
Κύκλωμα τροφοδοσίας

Το SPOT μπορεί να λειτουργήσει χρησιμοποιώντας οποιοδήποτε συνδυασμό από της εξής πηγές: την επαναφορτιζόμενη μπαταρία, USB Host ,είτε εξωτερική τροφοδοσία. Το κύκλωμα τροφοδοσίας είναι υπεύθυνο για να φορτίζει την ενσωματωμένη μπαταρία, να ρυθμίζει το ρεύμα που παρέχεται

στα υποσυστήματα του Main Board και του Sensor Board(eDEMO Board) είτε το SPOT βρίσκεται σε κανονική λειτουργία είτε σε deep-sleep. Το κύκλωμα αποτελείται από δύο τμήματα-κυκλώματα, κάθε ένα με διαφορετική λειτουργία LTC3455 και το TPS79730. Το LTC3455 έχει ενσωματωμένο, ένα κύκλωμα για την φόρτιση της μπαταρίας Li-ION ένα διαχειριστή ρεύματος για την USB και ένα διπλό σταθεροποιητή τάσης. Το LTC3455 διαχειρίζεται το ρεύμα που λαμβάνεται από την USB. Ανάλογα με τις απαιτήσεις της συσκευής, ο επεξεργαστής επιτρέπει την κατανάλωση περισσότερου ρεύματος από την USB. Το TPS79730 είναι ένας σταθεροποιητής τάσης και παρέχει μικρή ποσότητα ρεύματος στα 3Volt στην περίπτωση που το SPOT εισέλθει σε κατάσταση stand-by, επίσης παρέχει σταθερό ρεύμα στον Atmega88 και στην pSRAM και σε περίπτωση που η τάση πέσει κάτω από τα ασφαλή όρια λειτουργίας του επεξεργαστή τον απενεργοποιεί. Τα SPOT έχουν ειδικό firmware για εξοικονόμηση ενέργειας που μπορεί να θέσει την συσκευή σε τρεις καταστάσεις λειτουργίας:

- Run – Είναι η βασική κατάσταση στην οποία όλοι οι επεξεργαστές και το radio τροφοδοτούνται και λειτουργούν κανονικά. Η κατανάλωση σε αυτήν την κατάσταση φτάνει κυμαίνεται από 70mA ως 120mA, ενώ η κάρτα επέκτασης μπορεί να καταναλώνει μέχρι 400mA.
- Idle - Σε αυτή την κατάσταση το ρολόι του επεξεργαστή σταματάει και το radio απενεργοποιείται ενώ η κατανάλωση πέφτει στα 24mA.
- Deep-Sleep - Σχεδόν όλα τα κυκλώματα τροφοδοσίας απενεργοποιούνται εκτός από το κύκλωμα που δίνει ελάχιστο ρεύμα για την διατήρηση των δεδομένων της pSRAM. Η επαναφορά της συσκευής από την κατάσταση Deep-Sleep διαρκεί περίπου 2msec με 10msec.

Για να εισέλθει η συσκευή σε κατάσταση χαμηλής κατανάλωσης(Deep-Sleep) πρέπει το radio να είναι απενεργοποιημένο, να μην παρέχεται ρεύμα από εξωτερική συσκευή και να μην είναι ενεργοποιημένη η USB. Το SPOT εισέρχεται στις καταστάσεις Deep-Sleep και idle καλώντας κατάλληλες συναρτήσεις της βιβλιοθήκης. Επιπλέον μπορούσαμε να θέσουμε την συσκευή σε Deep-Sleep πατώντας το attention κουμπί για περισσότερα από 3 δευτερόλεπτα. Για να εξέλθει η συσκευή από Deep-Sleep πρέπει να χρησιμοποιήσουμε κάποιο εξωτερικό interrupt ή να πιέσουμε το attention κουμπί. Η παρακάτω εικόνα δείχνει τις μεταβάσεις που μπορεί να γίνουν μεταξύ των διαφορετικών καταστάσεων λειτουργίας.



Ελεγκτής τροφοδοσίας

Πρόκειται για τον 8-bit μικροελεγκτή Atmega88 της Atmel. Έχει ενσωματωμένο firmware που είναι υπεύθυνο για την λειτουργία του 64-bit ρολογιού, την επαναφορά της συσκευής σε περίπτωση που δεχτεί εξωτερικό interrupt και την επαναφορά ή είσοδό σε Deep-Sleep όταν πιεστεί το attention κουμπί. Η επικοινωνία με τον επεξεργαστή γίνεται μέσω του SPI διαύλου, από τον οποίο μεταφέρονται εντολές και δεδομένα κατάστασης από και προς τον Atmega88. Επίσης ο ελεγκτής μετράει και παρακολουθεί το φορτίο της μπαταρίας και τις τάσεις της USB, της μπαταρίας, και των εσωτερικών υποσυστημάτων χρησιμοποιώντας ένα 10-bit ACD. Ακόμα ο Atmega88 ελέγχει το power LED και δηλώνει διαφορετικές καταστάσεις(προβληματικές ή όχι) του SPOT μέσω ενδείξεων αυτού του LED. Για παράδειγμα όταν ανιχνεύσει ότι η μπαταρία έχει σχεδόν αποφορτιστεί ο ελεγκτής θα θέσει το power LED μόνιμα κόκκινο. Όλες οι πιθανές ενδείξεις παρουσιάζονται στον παρακάτω πίνακα.

Power State	Power LED Behavior
Powering up	One bright green pulse, sharp on, soft off
Powering down	Three bright red flashes
Charging the battery while CPU is active	Slowly alternate between a dim green and a bright green on a eight second cycle
Charging the battery when CPU is asleep	Slowly alternate between off and a dim green on an eight second cycle
External power supplied, but not charging, CPU active	Steady dim green
Battery low	Steady dim red. *This is a change from Release 1.0.*
Power fault	Two short red flashes
CPU going to sleep	Short red flash, short green flash
External interrupt or alarm, including button tap.	One short green flash

Μπαταρία

Η μπαταρία που χρησιμοποιείται στα SPOT είναι επαναφορτιζόμενη ιόντων λιθίου Li-ION στα 3.7V με χωρητικότητα 720mAH. Η μπαταρία ενσωματώνει κυκλώματα για την προστασία της από πλήρη αποφόρτιση, από υπερφόρτιση και από υψηλή τάση. Η φόρτιση μπορεί να γίνει είτε χρησιμοποιώντας ένα USB καλώδιο με βύσμα τύπου B είτε από οποιαδήποτε πηγή 5Volt (+/- 10%). Όταν δεν χρησιμοποιείται χάνει περίπου 2% της χωρητικότητας κάθε μήνα και σε περιπτώσεις υψηλής θερμοκρασίας ο ρυθμός αυτός αυξάνει. Τα κυκλώματα φόρτισης και διαχείρισης ρεύματος είναι ρυθμισμένα με ακρίβεια για να λειτουργούν με τον συγκεκριμένο τύπο μπαταρίας και για αυτό δεν πρέπει να αντικατασταθεί από άλλου τύπου.

Ασύρματος πομποδέκτης (wireless radio)

Τα SPOT για την ασύρματη μετάδοση δεδομένων ενσωματώνει τον ασύρματο πομποδέκτη CC2420. Το CC2420 συμμορφώνεται με το πρότυπο IEEE 802.15.4 και λειτουργεί σε συχνότητες από 2.4GHz ως 2.4835GHz(οι συχνότητες φαίνονται στο παρακάτω πίνακα), οι συχνότητες αυτές ανήκουν

στην ISM ζώνη και εξαιρούνται αδειοδότησης στην Ελλάδα σύμφωνα με τον νόμο 399/3-4-2006. Το κύκλωμα CC2420 εκτός από τον πομποδέκτη περιέχει δυο 128byte FIFOs για τα TX και RX δεδομένα, δυνατότητα για μέτρηση RSSI (received signal strength indication) με ευαισθησία 100db και ρύθμιση ισχύς του πομπού από -24dBm ως 0dBm(οι τιμές φαίνονται στο παρακάτω πίνακα). Ο πρακτικός ρυθμός μετάδοσης δεδομένων φτάνει τα 250Kbit/s ενώ η ευαισθησία του δέκτη είναι -90dBm. Για τα σήματα ελέγχου και δεδομένων από και προς το CC2420 στο Main Board χρησιμοποιούνται PIO θύρες και ο δίαυλος SPI. Στις PIO θύρες συνδέονται τα σήματα ελέγχου όπως reset, power down, start of frame(SFD) και σήματα κατάστασης όπως FIFO και FIFOP που ενημερώνουν αν η ουρά δεδομένων είναι άδεια ή αν έχουν ληφθεί δεδομένα. Ο δίαυλος SPI χρησιμοποιείται για την μεταβίβαση δεδομένων προς το CC2420. Το κύκλωμα καταναλώνει 20mA όταν ο δέκτης λαμβάνει δεδομένα και 18mA κατά την διάρκεια μετάδοσης με ισχύ 0dBm.

Channel	Center Frequency
11	2405MHz
12	2410MHz
13	2415MHz
14	2420MHz
15	2425MHz
16	2430MHz
17	2435MHz
18	2440MHz

Channel	Center Frequency
19	2445MHz
20	2450MHz
21	2455MHz
22	2460MHz
23	2465MHz
24	2470MHz
25	2475MHz
26	2480MHz

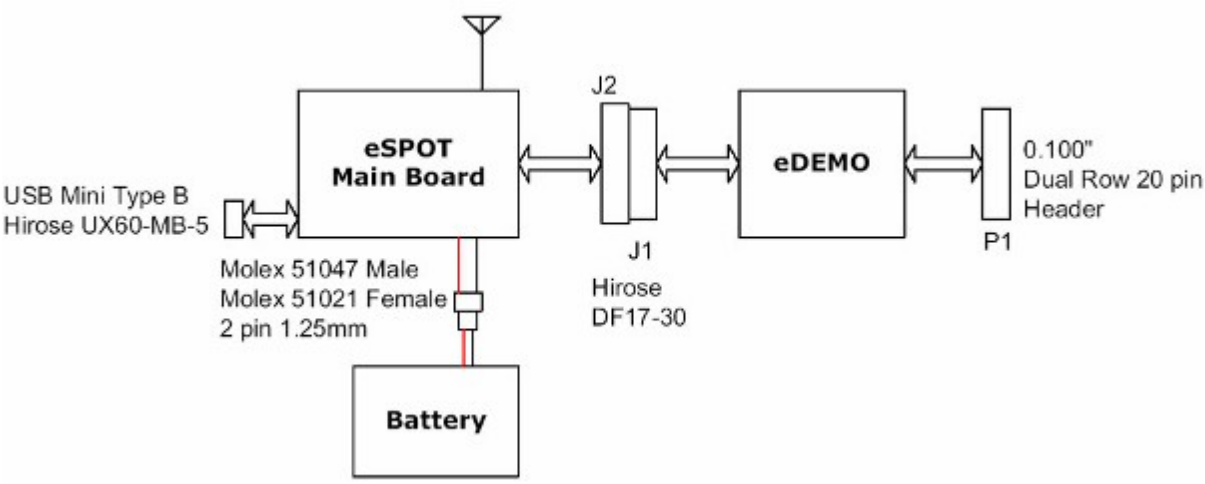
PA_LEVEL	Output Power
31	0dBm
27	-1dBm
23	-3dBm
19	-5dBm

PA_LEVEL	Output Power
15	-7dBm
11	-10dBm
7	-15dBm
3	-25dBm

Η κεραία του SPOT είναι τύπου inverted-F, τυπωμένη στην άνω επιφάνεια του PCB(Printed Circuit Board) του Main Board. Είναι σχεδιασμένη για να συντονίζεται στη συχνότητα 2450MHz με ωμική αντίσταση 115Ω. Λόγο της θέσης την κεραίας θα πρέπει να αποφεύγουμε την τοποθέτηση μεταλλικών αντικειμένων ή γραμμών τροφοδοσίας κοντά σε αυτήν. Σε εξωτερικό χώρο, κάτω από καλές καιρικές συνθήκες η εμβέλεια φτάνει τα 100m ενώ σε εσωτερικούς χώρους περιορίζεται στα 30m.

1.1.2 Στοιχεία του eDEMO Board

Το eDEMO Board είναι η κάρτα επέκτασης(daughterboard) του eSPOT. Αυτή είναι ενσωματωμένη στα SPOT που υπάρχουν στο αναπτυξιακό της SUN και προσφέρει μια ποικιλία από αισθητήρες και I/O θύρες. Στο eSPOT Main Board μπορούν να συνδεθούν και διαφορετικές κάρτες επέκτασης και αυτή την στιγμή είναι υπό σχεδίαση αρκετές κάρτες με πιο προηγμένες δυνατότητες και πιο ευαίσθητα όργανα. Προϋπόθεση για την προθήκη μιας κάρτας επέκτασης στο eSPOT είναι να συνδέεται με τον Main Board μέσω ενός βύσματος Hirose DF17-30, να υποστηρίζει το SPI interface αφού μέσω αυτού του διαύλου γίνεται η επικοινωνία και να περιέχει μια SPI flash για την αποθήκευση πληροφορίας σχετικά με τις παραμέτρους λειτουργίας της. Στο παρακάτω σχήμα βλέπουμε την διασύνδεση του eDEMO με τα υπόλοιπα στοιχεία του SPOT.



Το eDEMO Board αποτελεί την πλατφόρμα αισθητήρων(sensor board) του SPOT και περιέχει επιταχυνσιόμετρο που μετράει την επιτάχυνση και στους 3 άξονες, αισθητήρα φωτός και οκτώ LEDs τριών χρωμάτων. Επίσης περιλαμβάνει 2 κουμπιά, έξι αναλογικές εισόδους, τέσσερις αναλογικές εισόδους υψηλής τάσης, και πέντε ψηφιακές γενικού σκοπού I/O θύρες. Για τον χειρισμό των παραπάνω στοιχείων η SUN έχει κατάλληλους drivers και βιβλιοθήκες με κλάσεις για τον χειρισμό τους. Για παράδειγμα αν θέλουμε να ελέγξουμε το πράσινο LED πρέπει να χρησιμοποιήσουμε την κλάση Iled της βιβλιοθήκης:

```
Iled theLed = Spot.getInstance().getGreenLed();
και για τον χειρισμό του LED:
theLed.setOn();
theLed.setOff();
```

Περισσότερο αναλυτικά για τον προγραμματισμό των SPOTs θα αναφερθούμε στο επόμενο κεφάλαιο. Κλείνοντας την παρουσίαση του sensor board θα παραθέσουμε μερικά τεχνικά χαρακτηριστικά των υποσυστημάτων του:

Operating Temperature (with battery charging)	0 to 45C
Operating Temperature (with battery discharging)	-20 to 60C
Operating Temperature (without battery)	-20°C to +75°C
Storage Temperature (with battery)	-20°C to +35°C
Storage Temperature (without battery)	-40°C to +85°C
Voltage on any input pin	-0.1V to 3.5V
eDEMO DC Current per I/O pin	40.0ma
eSPOT DC Current per I/O pin	8.0ma
Maximum External/USB voltage	6.0V

2. Προγραμματισμός των SPOTs

Στο κεφάλαιο αυτό θα παρουσιάσουμε τα βασικά στοιχεία που χρειάζεται κάποιος να γνωρίζει για την ανάπτυξη εφαρμογών στην πλατφόρμα SUN SPOT. Θα παρουσιάσουμε χρήσιμα και απαραίτητα εργαλεία για την υλοποίηση και την εγκατάσταση των προγραμμάτων στα SPOTs που προσφέρονται από την SUN στο αναπτυξιακό kit. Επίσης θα αναφερθούμε στην δομή που πρέπει να έχουν οι εφαρμογές και θα δείξουμε πως μπορεί κάποιος να χρησιμοποιήσει τις πρότυπες βιβλιοθήκες του SUN SPOT για να δουλέψει με το radio και τους αισθητήρες του eDEMO Board. Τέλος θα παρουσιάσουμε παράδειγμα εφαρμογής και θα αναλύσουμε τα κύρια στοιχεία του.

Ο προγραμματισμός στην πλατφόρμα SUN SPOT [12] γίνεται με την γλώσσα Java. Συγκεκριμένα οι εφαρμογές ακολουθούν τις προδιαγραφές του MIDP(Mobile Information Device Profile) που κτίζεται πάνω στο CLDC και προσθέτει ένα επιπλέον API για εφαρμογές σε embedded συσκευές. Το MIDP χρησιμοποιείται σε πολλά embedded συστήματα και συσκευές όπως για παράδειγμα τα κινητά τηλέφωνα. Τα συγκεκριμένα προγράμματα που ακολουθούν τις παραπάνω προδιαγραφές καλούνται MIDlets και έχουν συγκεκριμένη δομή και περιορισμούς.

Τα MIDlets τρέχουν σε μια μικρή Java ME(J2ME) VM που λέγεται Squawk VM. Όπως είχαμε αναφέρει και σε προηγούμενη ενότητα τα SPOT δεν έχουν λειτουργικό σύστημα, αλλά τον ρόλο του OS τον αναλαμβάνει η Squawk VM. Μαζί με τα MIDlet του χρήστη αλλά σε “χαμηλότερο” επίπεδο τρέχουν και μια πλειάδα άλλων εφαρμογών που δεν είναι άμεσα “ορατές” στον χρήστη:

- Ο bootloader – που είναι υπεύθυνος για την USB σύνδεση, εκκινεί τις εφαρμογές και επικοινωνεί με τα ant scripts του PC που είναι συνδεδεμένο το SPOT.
- bootstrap suite – που περιλαμβάνει τις πρότυπες κλάσεις της Java ME.
- library suite – που περιλαμβάνει την βιβλιοθήκη με τις κλάσεις σχετικές με το SUN SPOT.

Η Squawk VM χρησιμοποιεί ανεξάρτητες περιοχές για εκτέλεση εφαρμογών, τα isolates. Κάθε isolate συνιστά ένα διαφορετικό σύνολο από threads και αντικείμενα που σχετίζονται με αυτά. Το SPOT έχει πάντα ένα master isolate, στο οποίο τρέχουν deamon threads που διαχειρίζονται βασικές λειτουργίες του. Αυτά τα threads είναι τμήμα της βασικής βιβλιοθήκης του SUN SPOT και φροντίζουν για την διαχείριση ενέργειας (απενεργοποιώντας υποσυστήματα που δεν χρησιμοποιούνται), παρακολουθούν την κατάσταση της USB και αποτελούν μέρος της υλοποίησης του radiostack. Τα υπόλοιπα isolate που μπορεί να δημιουργηθούν καλούνται child isolates. Η προκαθορισμένη(default) συμπεριφορά του SPOT είναι οι εφαρμογές του χρήστη να τρέχουν στο master isolate αν και αυτό δεν είναι υποχρεωτικό.

Τα threads χρησιμοποιούνται στα MIDlet μέσω της κλάσης Thread, και υπάρχει η δυνατότητα να ρυθμιστεί η προτεραιότητά τους από 1(Thread.MIN_PRIORITY) ως 10(Thread.MAX_PRIORITY). Επίσης υπάρχει η δυνατότητα να δοθούν και υψηλότερες προτεραιότητες όπως οι

“προτεραιότητες συστήματος”(system priorities), αυτές όμως αφορούν ορισμένα deamon threads και δεν προορίζονται για χρήση από τα MIDlets. Εδώ πρέπει να αναφέρουμε ότι για την σωστή λειτουργία των threads των βιβλιοθηκών του SPOT, οι προγραμματιστές θα πρέπει να αναθέτουν προτεραιότητα χαμηλότερη από 5(Thread.NORMAL) όταν οι εφαρμογές τους είναι cpu-bounded. Υψηλές προτεραιότητες μπορεί να προκαλέσουν προβλήματα στα threads της SPOT library, όπως την απώλεια broadcast μηνυμάτων.

Στην Java SE μια εφαρμογή θα πρέπει να περιέχει μία main() μέθοδο ή να υλοποιεί το Applet interface αν πρόκειται να εκτελεστεί από έναν browser. Όμως στην Java ME, που υλοποιεί η Squawk VM, κάθε εφαρμογή που υλοποιούμε πρέπει να είναι συμβατή με το πρότυπο MIDlet. Όλες οι εφαρμογές για τα SPOT πρέπει να κληρονομούν(extends) τα στοιχεία της κλάσης MIDlet και να υλοποιούν τις μεθόδους:

- startApp() - Η μέθοδος αυτή καλείται όταν πρόκειται να εκτελεστεί το MIDlet.
- PauseApp() - Η μέθοδος αυτή καλείται όταν πρόκειται να ανασταλεί η εκτέλεση του MIDlet.
- destroyApp() - Η μέθοδος αυτή καλείται όταν το MIDlet τερματίζεται από το σύστημα, όπως σε περιπτώσεις που το isolate που εκτελείται το MIDlet καταστραφεί με την μέθοδο ”Isolate.exit()” είτε τερματίσει η VM με την μέθοδο VM.stopVM() είτε το MIDlet προκαλέσει μια εξαίρεση εκτός της MIDletStateChangeException.

Προαιρετικά μπορεί να υπάρχει μια μέθοδος δημιουργός(constructor) χωρίς ορίσματα. Για τον τερματισμό ενός MIDlet από τον προγραμματιστή πρέπει να χρησιμοποιείται η μέθοδοςnotifyDestroyed(), οι εφαρμογές δεν πρέπει ποτέ να καλούν την μέθοδο System.exit(). Όλες λοιπόν οι εφαρμογές για τα SPOT έχουν την παρακάτω βασική δομή (εισάγουμε όλες τις κλάσεις που χρειαζόμαστε προκειμένου να έχουμε πλήρη λειτουργικότητα):

```
import com.sun.spot.peripheral.Spot;
import com.sun.spot.sensorboard.EDemoBoard;
import com.sun.spot.sensorboard.peripheral.ITriColorLED;
import com.sun.spot.peripheral.radio.IRadioPolicyManager;
import com.sun.spot.io.j2me.radiostream.*;
import com.sun.spot.io.j2me.radiogram.*;
import com.sun.spot.util.*;
import java.io.*;
import javax.microedition.io.*;
import javax.microedition.midlet.MIDlet;
import javax.microedition.midlet.MIDletStateChangeException;


public class SunSpotApplication extends MIDlet {
```

```
    protected void startApp() throws MIDletStateChangeException {
    }
    protected void pauseApp() {
    }
    protected void destroyApp(boolean unconditional)  throws
MIDletStateChangeException {
    }
}
```

Οι δομή των φακέλων όταν αναπτύσσουμε μια εφαρμογή για τα SPOT πρέπει να έχουν συγκεκριμένη δομή. Στο φάκελο κάθε εφαρμογής πρέπει να υπάρχουν 2 αρχεία, ένα αρχείο build.xml και ένα build.properties που χρησιμοποιούνται από τα ant scripts για την μετάφραση και την εκτέλεση των MIDlet. Επιπλέον πρέπει να έχουμε και 3 υποφακέλους, έναν με όνομα src που τοποθετούμε τους καταλόγους με τα αρχεία πηγαίου κώδικα και έναν με τον όνομα resources/META-INF που περιγράφεται στην επόμενη παράγραφο. Τέλος αν η εφαρμογή μας χρησιμοποιεί το NetBeans IDE τότε θα υπάρχει και ένας τρίτος φάκελος με όνομα nbproject, με τα περιεχόμενα του NetBeans project.

Στο φάκελο resources/META-INF υπάρχει το αρχείο MANIFEST.MF που περιέχει πληροφορίες που χρησιμοποιούνται από την Squawk VM για την εκκίνηση των εφαρμογών. Συγκεκριμένα περιέχει τα όνομα των αρχικών κλάσεων των MIDlets και ορισμένες ιδιότητες αυτών. Επιπλέον ο φάκελος μπορεί να περιλαμβάνει αρχεία που ορίζει ο προγραμματιστής και είναι διαθέσιμα στην εφαρμογή όταν εκτελείται. Η δομή ενός τυπικού MANIFEST.MF είναι:

```
MIDlet-Name: Air Text demo
MIDlet-Version: 1.0.0
MIDlet-Vendor: Sun Microsystems Inc
MIDlet-1: AirText, , org.sunspotworld.demo.AirTextDemo
MicroEdition-Profile: IMP-1.0
MicroEdition-Configuration: CLDC-1.1
```

Η σύνταξη κάθε γραμμής είναι *<property-name>:<space><property-value.* Η πιο σημαντική γραμμή για κάθε πρόγραμμα είναι η *MIDlet-1:<όρισμα 1>,<όρισμα 2>,<όρισμα 3>.* Το πρώτο όρισμα είναι μια περιγραφή της εφαρμογής και το τρίτο όρισμα είναι η κύρια κλάση του MIDlet. Το δεύτερο όρισμα είναι μια εικόνα που σχετίζεται με το MIDlet, αλλά στην παρούσα έκδοση δεν υποστηρίζεται αυτή η επιλογή. Μέσα από την εφαρμογή μπορούμε να διαβάσουμε τις τιμές για τις παραπάνω ιδιότητες χρησιμοποιώντας την μέθοδο <όνομα MIDlet>*.getAppProperty("<property-name>")*. Όλα τα αρχεία μέσα στον φάκελο resources είναι διαθέσιμα στην εφαρμογή κατά την διάρκεια που εκτελείται, μπορούμε να ανοίξουμε ένα stream εισόδου για να διαβάσουμε τα περιεχόμενά τους μέσω της μεθόδου *InputStream is = getClass().getResourceAsStream("/<όνομα αρχείου>").*

2.1 Βιβλιοθήκες του συστήματος

Στην ενότητα αυτή θα δούμε τα περιεχόμενα και την λειτουργία των βασικών βιβλιοθηκών του SUN SPOT, αυτές είναι: η βιβλιοθήκη συσκευών(device library), h βιβλιοθήκη για ασύρματη επικοινωνία(radio communication library) και τη βιβλιοθήκη για τον έλεγχο του sensor board. Επίσης θα δώσουμε παραδείγματα για το πως μπορούμε να χειριστούμε διαφορετικά υποσυστήματα του SUN SPOT και του sensor board μέσω κλάσεων αυτών των βιβλιοθηκών.

2.1.1 Device Library

Η βιβλιοθήκη συσκευών βρίσκεται στο spotlib_device.jar και στο spotlib_common.jar. Ο πηγαίος κώδικας της βιβλιοθήκης(των δυο παραπάνω jar) βρίσκεται στο spolib_source.jar στο φάκελο "*Sun\SunSPOT\sdk\src*" του sdk και περιέχει drivers για τις παρακάτω συσκευές:

- On-board LEDs
- Τους διαύλους PIO,USART και τα ρολόγια/μετρητές του επεξεργαστή
- Τον πομποδέκτη CC2420
- Το SPI interface
- Την ενσωματωμένη flash μνήμη

Επίσης περιλαμβάνει κλάσεις για τον χειρισμό του basestation(*com.sun.spot.peripheral.basestation*), κλάσεις για έλεγχο των SPOT μέσω του basestation, OTA(Over The Air), ένα framework για την επικοινωνία εφαρμογών που εκτελούνται σε διαφορετικά isolates , και ένα handler για το attention κουμπί. Κάθε στοιχείο και υποσύστημα ελέγχεται από τους παραπάνω drivers, ενώ στον προγραμματιστή δίνεται πρόσβαση στα υποσυστήματα μέσω των παρακάτω interfaces:

Υποσύστημα	**Interface**
LED	*ILed*
PIO	*IAT91_PIO*
AIC	*IAT91_AIC*
Timer-Counter	*IAT91_TC*
CC2420	*I802_15_4_PHY*
MAC layer	*I802_15_4_MAC*
RadioPolicyManager	*IRadioPolicyManager*
SPI	*ISpiMaster*
Flash memory	*IFlashMemoryDevice*
Power controller	*IPowerController*
OTA Command Server	*OTACommandServer*

Attention button handler	*FiqInterruptDaemon*

Κλάσεις που υλοποιούν τα παραπάνω interfaces δημιουργούνται από ένα αντικείμενο της κλάσης Spot που είναι υπεύθυνο για την δημιουργία των drivers και διαχείριση της πρόσβασης σε αυτόν. Για παράδειγμα για να εκλέξουμε το πράσινο LED του Main Board χρησιμοποιούμε τον ακόλουθο κώδικα:

```
Iled theLed = Spot.getInstance().getGreenLed();

theLed.setOn();
theLed.setOff();
```

2.1.2 Radio Communication Library

Πρόκειται για την βιβλιοθήκη που είναι υπεύθυνη για την δημιουργία και τον έλεγχο όλων των συνδέσεων και πρωτοκόλλων, όπως radiostream και radiogram. Οι κλάσεις που υλοποιούν τμήματα του radio stack πάνω από το MAC layer βρίσκονται στο *multihoplib_rt.jar* ο αντίστοιχος πηγαίος κώδικας στο *multihoplib_source.jar.* Η τωρινή έκδοση του SUN SPOT SDK χρησιμοποιεί το GCF(Generic Connection Framework) για την δημιουργία συνδέσεων και την ασύρματη επικοινωνία μεταξύ των SPOTs χρησιμοποιώντας δυο πρωτόκολλα:

- radiostream – Το radiostream παρέχει αξιόπιστη μετάδοση δεδομένων μεταξύ δυο κόμβων, επίσης χρησιμοποιεί buffers για καλλίτερη απόδοση. Το πρωτόκολλο αυτό είναι stream-based.
- Radiogram – Στο radiogram πρωτόκολλο η μεταφορά δεδομένων γίνεται με datagrams, και δεν παρέχει καμία εγγύηση ότι τα πακέτα θα παραλειφθούν σωστά ή ότι θα φτάσουν στον προορισμό με την σειρά που στάλθηκαν. Όταν ένα πακέτο στέλνεται μέσω άλλων κόμβων(περισσότερα του ενός hop), υπάρχει περίπτωση να χαθεί χωρίς να παρέχεται κάποια ειδοποίηση είτε να παραλειφθεί από τον προορισμό περισσότερες από μια φορές είτε να μην φτάσει με την σειρά που στάλθηκε. Ενώ στην περίπτωση που τα datagrams στέλνονται σε γειτονικό κόμβο(ένα hop), η μόνη περίπτωση είναι κάποια να παραλειφθούν περισσότερες από μια φορές.

Παράδειγμα χρήσης του radiostream

Για να συνδεθούμε με ένα κόμβο με το πρωτόκολλο radiostream χρησιμοποιούμε τον παρακάτω κώδικα για να “ανοίξουμε” μια σύνδεση:

```
RadiostreamConnection conn =
(RadiostreamConnection)Connector.open("radiostream://<destinationAddr>:<portNo>");
```

Το *<destinationAddr>* είναι η 64-bit διεύθυνση του προορισμού και η *<portNo>* είναι ένας αριθμός από 1 ως 255 που χαρακτηρίζει μοναδικά την

συγκεκριμένη σύνδεση. Ο προγραμματιστής μπορεί να επιλέξει οποιοδήποτε αριθμό port θέλει από την παραπάνω περιοχή εκτός των ports 1 ως 31 που είναι δεσμευμένα για χρήση από το σύστημα. Προκειμένου να μπορούν δυο κόμβοι να χρησιμοποιήσουν ένα radiostream για να επικοινωνήσουν μεταξύ τους πρέπει και οι δυο να ανοίξουν ένα *RadiostreamConnection* στην την ίδια port και με τις αντίστοιχες διευθύνσεις. Αφού έχουμε ανοίξει μια radiostream σύνδεση μπορούμε να πάρουμε το stream εισόδου και εξόδου αυτός της σύνδεσης για να μεταδώσουμε δεδομένα, για παράδειγμα:

```
DataInputStream dis = conn.openDataInputStream();
DataOutputStream dos = conn.openDataOutputStream();
```

Παρακάτω ακολουθεί να απλό παράδειγμα με δυο προγράμματα:

Program 1

```
RadiostreamConnection conn = (RadiostreamConnection)
Connector.open("radiostream://0014.4F01.0000.0006:100");
DataInputStream dis = conn.openDataInputStream();
DataOutputStream dos = conn.openDataOutputStream();
try {
dos.writeUTF("Hello up there");
dos.flush();
System.out.println ("Answer was: " + dis.readUTF());
} catch (NoRouteException e) {
System.out.println ("No route to 0014.4F01.0000.0006");
} finally {
dis.close();
dos.close();
conn.close();
}
```

Program 2

```
RadiostreamConnection conn = (RadiostreamConnection)
Connector.open("radiostream://0014.4F01.0000.0007:100");
DataInputStream dis = conn.openDataInputStream();
DataOutputStream dos = conn.openDataOutputStream();
try {
String question = dis.readUTF();
if (question.equals("Hello up there")) {
dos.writeUTF("Hello down there");
} else {
dos.writeUTF("What???");
}
dos.flush();
} catch (NoRouteException e) {
System.out.println ("No route to 0014.4F01.0000.0007");
} finally {
dis.close();
dos.close();
conn.close();
}
```

Σε αυτό το παράδειγμα τα δυο προγράμματα ανοίγουν ένα radiostream connection και τα αντίστοιχα stream εισόδου/εξόδου. Μετά το πρόγραμμα 2 αναμένει να λάβει δεδομένα από το stream και το πρόγραμμα 1 στέλνει το μήνυμα "Hello up there". Αν το μήνυμα παραλειφθεί σωστά το πρόγραμμα 2 απαντά “Hello down there” και η απάντηση τυπώνεται στο System.out stream από το πρόγραμμα 1. Τα δεδομένα στέλνονται ασύρματα όποτε γεμίσει το buffer του radiostream, αν θέλουμε να σταλούν τα δεδομένα άμεσα πρέπει να χρησιμοποιήσουμε την εντολή flush(). Επίσης αν το πρόγραμμα 2 ξεκινήσει πριν το 1 υπάρχει περίπτωση να χαθεί το μήνυμα "Hello up there", αφού η αποστολή θα γίνει πριν το πρόγραμμα 1 ζητήσει δεδομένα. Για σίγουροι ότι θα εκτελεστούν με την σωστή σειρά πρέπει να εφαρμόσουμε κάποιο είδος handshake ή να εισάγουμε μια καθυστέρηση πριν πρόγραμμα 2 στείλει το μήνυμα. Σε περίπτωση που δεν μπορεί να βρεθεί μια διαδρομή προς τον προορισμό προκαλείται μια εξαίρεση *NoRouteException*.

Σε κάθε μετάδοση ενός πακέτου το MAC layer περιμένει την αποστολή ενός ACK(πακέτο επιβεβαίωσης), που δηλώνει την επιτυχή λήψη του πακέτου. Σε περίπτωση που ο τελικός προορισμός βρίσκεται σε απόσταση ένα hop, το MAC-level ack είναι αρκετό για να διασφαλίσουμε την σωστή λήψη. Σε διαφορετική περίπτωση το radiostream θα ζητήσει από τον προορισμό να στείλει ένα πακέτο επιβεβαίωσης πίσω στον αρχικό αποστολέα. Για την βελτίωση της απόδοσης του πρωτοκόλλου όταν στέλνουμε δεδομένα, το output stream δεν περιμένει το ACK από τον προορισμό αλλά επιστρέφει άμεσα. Σε περίπτωση που δεν παραλάβουμε το πακέτο επιβεβαίωσης σε κάποιο προκαθορισμένο χρονικό διάστημα τότε ξαναστέλνεται αυτόματα. Μετά από ένα αριθμό αποτυχημένων προσπαθειών προκαλείται μια εξαίρεση *NoMeshLayerAckException* στην επόμενη προσπάθεια για αποστολή δεδομένων. Σε αυτή την περίπτωση δεν μπορούμε να ξέρουμε ποσά από τα δεδομένα που έχουμε στείλει έχουν πράγματι φτάσει στον προορισμό.

Τέλος μια άλλη εξαίρεση που μπορεί να προκληθεί και στα στα δυο πρωτόκολλα (radiostream radiogram), είναι η ChannelBusyException. Αυτή η εξαίρεση είναι ένδειξη ότι το κανάλι είναι απασχολημένο, δηλαδή ότι μεταδίδουν άλλες συσκευές.

Παράδειγμα χρήσης του radiogram

Το πρωτόκολλο radiogram ακολουθεί το μοντέλο client-server, και παρέχει επικοινωνία μεταξύ δυο συσκευών μέσω datagrams.

Για να “ανοίξουμε” μια σύνδεση από την πλευρά του server χρησιμοποιούμε τον ακόλουθο κώδικα:

```
RadiogramConnection conn =
(RadiogramConnection)Connector.open("radiogram://:<portNo>");
```

Η <*portNo*> είναι ένας αριθμός από 1 ως 255 που χαρακτηρίζει μοναδικά την συγκεκριμένη σύνδεση. Ο προγραμματιστής μπορεί να επιλέξει οποιοδήποτε αριθμό port θέλει από την παραπάνω περιοχή εκτός των ports 1 ως 31 που είναι δεσμευμένα για χρήση από το σύστημα.

Για να “ανοίξουμε” μια σύνδεση από την πλευρά του client

χρησιμοποιούμε τον ακόλουθο κώδικα:

```
RadiogramConnection conn =
(RadiogramConnection)Connector.open("radiogram://<serveraddr>:<portNo>");
```

Το <*destinationAddr*> είναι η 64-bit διεύθυνση του server και η <*portNo*> είναι η port που έχει ανοίξει το radiogram connection ο server που θέλουμε να συνδεθούμε.

Τα δεδομένα μεταξύ του client και του server στέλνονται ως datagrams, προκειμένου να στείλουμε δεδομένα πρέπει να κατασκευάσουμε ένα αντικείμενο datagram από το connection που έχουμε ανοίξει, να γράψουμε σε αυτό τα δεδομένα που θέλουμε και μετά να το στείλουμε στον προορισμό μέσω του connection. Παρακάτω παραθέτουμε ένα παράδειγμα που έχει την ίδια λειτουργία με τα προγράμματα 1 και 2 της προηγούμενης ενότητας:

Client end

```
RadiogramConnection conn =
(RadiogramConnection)Connector.open("radiogram://0014.4F
01.0000.0006:100");
Datagram dg =
conn.newDatagram(conn.getMaximumLength());
try {
dg.writeUTF("Hello up there");
conn.send(dg);
conn.receive(dg);
System.out.println ("Received: " + dg.readUTF());
} catch (NoRouteException e) {
System.out.println ("No route to 0014.4F01.0000.0006");
} finally {
conn.close();
}
```

Server end

```
RadiogramConnection conn = (RadiogramConnection)
Connector.open("radiogram://:100");
Datagram dg =
conn.newDatagram(conn.getMaximumLength());
Datagram dgreply =
conn.newDatagram(conn.getMaximumLength());
try {
conn.receive(dg);
String question = dg.readUTF();
dgreply.reset(); // reset stream pointer
dgreply.setAddress(dg); // copy reply address from input
if (question.equals("Hello up there")) {
dgreply.writeUTF("Hello down there");
} else {
dgreply.writeUTF("What???");
```

```
}
conn.send(dgreply);
} catch (NoRouteException e) {
System.out.println ("No route to " + dgreply.getAddress());
} finally {
conn.close();
}nn.close();
}
```

Ορισμένα χαρακτηριστικά των datagrams που θα πρέπει να αναφέρουμε είναι:

- Τα datagrams που στέλνονται μέσω ενός datagram connection πρέπει να έχουν κατασκευαστεί από αυτό, χρησιμοποιώντας την μέθοδο newDatagram() του ανοιχτού connection. Δεν μπορούμε να στείλουμε datagrams σε ένα connection, τα οποία έχουν προέλθει από ένα άλλο.
- Ένα datagram connection που έχει ανοιχτεί για μια συγκεκριμένη διεύθυνση δεν μπορεί να χρησιμοποιηθεί για να σταλούν πακέτα σε άλλο προορισμό. Αν προσπαθήσουμε να στείλουμε ένα datagram σε διαφορετικό προορισμό θα προκληθεί εξαίρεση.
- Είναι δυνατό να έχουμε στην ίδια συσκευή να έχουμε ανοικτό ένα server και ένα client connection στην ίδια port. Τα datagrams που λαμβάνονται και προορίζονται για την συγκεκριμένη port θα προωθούνται στο server connection.
- Στην τωρινή υλοποίηση του πρωτοκόλλου στα SPOT όταν ο server κλείσει το connection, τότε κλείνει αυτόματα και το connection του client.

Τα radiograms connections μπορούν να χρησιμοποιηθούν για την αποστολή broadcast πακέτων. Η προκαθορισμένη συμπεριφορά είναι να αποστέλλονται τα broadcast σε ακτίνα 2 hop για να μπορούν να λαμβάνονται από κοντινές συσκευές που όμως βρίσκονται εκτός ακτίνας μετάδοσης. Ο κώδικας για τη αποστολή broadcasts πακέτων είναι ο εξής:

```
DatagramConnection conn =
(DatagramConnection)Connector.open("radiogram://broadcast:<portNo>");
```

Για την λήψη broadcasts πακέτων δεν μπορεί να χρησιμοποιηθεί το παραπάνω connection αλλά πρέπει να ορίσουμε ένα client connection ως εξής:

```
RadiogramConnection conn =
(RadiogramConnection)Connector.open("radiogram://:<portNo>");
```

Αν θέλουμε να τροποποιήσουμε τον αριθμό των hops που θα μεταβεί το broadcast μήνυμα τότε μπορούμε να χρησιμοποιήσουμε την παρακάτω μέθοδο στο “ανοιχτό” connection θέτοντας το n στον αριθμό των hops που θέλουμε :

```
((RadiogramConnection)conn).setMaxBroadcastHops(n);
```

Εδώ πρέπει να προσθέσουμε ότι ο μηχανισμός μετάδοσης broadcast δεν παρέχει καμία εγγύηση για σωστή παραλαβή, αφού δεν χρησιμοποιεί πακέτα επιβεβαίωσης, ούτε πραγματοποιούνται αναμεταδώσεις broadcast πακέτων.

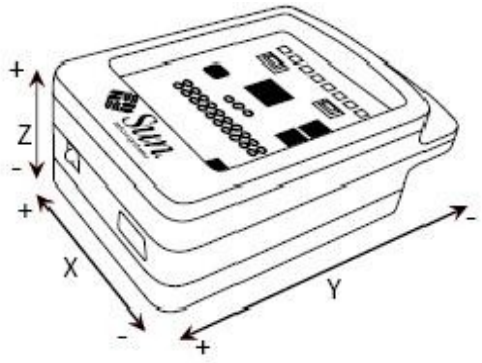
Στην περίπτωση που το μέγεθος των broadcast πακέτων είναι μεγάλο, τότε στέλνεται σε τμήματα(fragments) και αυξάνεται η πιθανότητα να συμβεί κάποιο σφάλμα σε τουλάχιστο ένα από αυτά. Το μέγιστο μέγεθος δεδομένων(payload) των broadcast πακέτων που μπορούμε να στείλουμε είναι 1260 bytes, ενώ το payload των πακέτων του 802.15.4 radio είναι περίπου 100 bytes. Στην περίπτωση που στέλνονται πολλά fragments τα SPOTs αντιμετωπίζουν και ένα επιπλέον πρόβλημα που οδηγεί σχεδόν σίγουρα στην απώλεια δεδομένων. Το πρόβλημα είναι ότι η συσκευή που δέχεται τα πακέτα έχει ένα περιορισμό στο πόσο γρήγορα μπορεί να αδειάζει τον packet buffer, κάτι που μπορεί να επιδεινώνεται σε περιπτώσεις που δέκτης έχει μεγάλο αριθμό από ενεργά threads ή τυχαίνει να καλεί συχνά τον gc(garbage collector). Για αυτό τον λόγο προτείνεται να μην στέλνουμε broadcast radiograms με περισσότερα από 200 bytes δεδομένων, δηλαδή το πολύ 2 radio packets για κάθε broadcast και να εισάγουμε μια καθυστέρηση 20ms μεταξύ διαδοχικών broadcast ώστε να προλαβαίνει ο δέκτης να τα παραλαμβάνει.

2.1.3 Sensor Board library

Πρόκειται για την βιβλιοθήκη που περιέχει όλες τις κλάσεις και τα interfaces που χρειαζόμαστε για την διαχείριση των συστημάτων και των αισθητήρων του eDEMO Board. Αυτές βρίσκονται στο αρχείο *transducerlib_rt.jar*. Σε αυτή την ενότητα θα παρουσιάσουμε συνοπτικά με παραδείγματα πως μπορούμε να χρησιμοποιήσουμε τις κλάσης αυτής της βιβλιοθήκης για να χειριστούμε το επιταχυνσιόμετρο, τα LED, τον αισθητήρα φωτεινότητας, τον αισθητήρα θερμοκρασίας, τα τις ψηφιακές θύρες εισόδου/εξόδου καθώς και τους διακόπτες. Για περισσότερες λεπτομέρειες προτρέπουμε τον αναγνώστη να διαβάσει τα αντιστοιχία τμήματα του javadoc και να μελετήσει τα παραδείγματα που βρίσκονται στον φάκελο “*[SpotSDKdirectory]/doc/AppNotes/”*.

Επιταχυνσιόμετρο

Το επιταχυνσιόμετρο μετράει την επιτάχυνση και στους τρεις άξονες. Ο Z-άξονας είναι κατακόρυφος σε σχέση με τα board του SPOT. Ο X-άξονας είναι παράλληλος στην σειρά των LEDs του sensor board, ενώ ο Y-άξονας είναι παράλληλος στην μεγάλη πλευρά του SPOT. Στην παρακάτω εικόνα βλέπουμε τους άξονες στους οποίους μετράει την επιτάχυνση ο αισθητήρας καθώς και την κατεύθυνση(σημειώνεται με +) προς την οποία όταν αυξάνεται η επιτάχυνση ο αισθητήρας δίνει μεγαλύτερες τιμές.



Για να χρησιμοποιήσουμε το επιταχυνσιόμετρο στις εφαρμογές μας, πρώτα δημιουργούμε ένα αντικείμενο από το αντίστοιχο interface της βιβλιοθήκης(*IAccelerometer3D*):

```
import com.sun.spot.sensorboard.EDemoBoard;
import com.sun.spot.sensorboard.IAccelerometer3d;
//....
IAccelerometer3D ourAccel =
EDemoBoard.getInstance().getAccelerometer();
```

Και έπειτα μπορούμε να πάρουμε της τιμές για την επιτάχυνση στους άξονες με τις ακόλουθες μεθόδους:

```
double x-accel = ourAccel.getAccelX();
double y-accel = ourAccel.getAccelY();
double z-accel = ourAccel.getAccelZ();
```

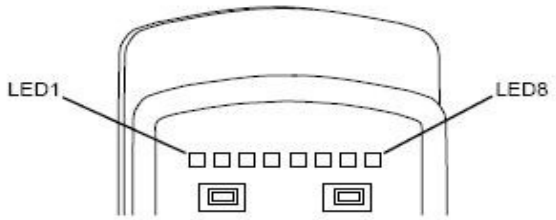
Οι τιμές που επιστρέφονται είναι σε μονάδες G(m^2/s). Επιπλέον υπάρχει η μέθοδος getAccel() που επιστρέφει την συνισταμένη επιτάχυνση από τους τρεις άξονες. Το interface του επιταχυνσιομέτρου παρέχει μεθόδους για την μέτρηση της επιτάχυνσης σε σχέση με μια ορισμένη προηγμένη τιμή της. Με αυτό τον τρόπο μπορούμε να αφαιρέσουμε από τις μετρήσεις μας τιμές που προτίθενται από την επιτάχυνσης της βαρύτητας.

```
ourAccel.setRestOffsets();
double z-relative-accel = ourAccel.getRelativeAccelZ();
```

Μια ακόμα λειτουργία που μας παρέχεται είναι ο υπολογισμός του προσανατολισμού του SPOT σε σχέση με την επιτάχυνσή του. Οι τιμές αυτές είναι σε radians και επιστρέφονται από τις μεθόδους, getTiltX(), getTiltY(), getTiltZ().

LEDs

Υπάρχουν οκτώ LEDs στο sensor board, διατεταγμένα σε μία σειρά ξεκινώντας από το LED1 στην αριστερή πλευρά και καταλήγοντας στο LED8 στην δεξιά. Κάθε LED έχει τρεις ενσωματωμένες πηγές φωτεινότητας, μια πράσινη, μια μπλε και μια κόκκινη. Η ισχύς κάθε πηγής μπορεί να κυμανθεί από 0 ως 255, τιμή 0 σημαίνει ότι η πηγή δεν εκπέμπει ενώ τιμή 255 ότι έχει πλήρη φωτεινότητα.



Παρακάτω παραθέτουμε πως μπορούμε να χρησιμοποιήσουμε τα LED και τον αντίστοιχο κώδικα.

1. Αρχικοποίηση των LED

```
Import com.sun.spot.sensorboard.EDemoBoard;
Import com.sun.spot.sensorboard.ITriColorLED;
///....
ITriColorLED[] ourLEDs = EDemoBoard.getInstance().getLEDs();
```

2. Ανάθεση χρώματος στα LEDs – Η ανάθεση χρώματος στα LEDs γίνεται χρησιμοποιώντας την μέθοδο *setRGB(int red, int green, int blue)*, κάθε όρισμα μπορεί να πάρει τιμές από 0 ως 255 ανάλογα με το πόσο έντονο θέλουμε να έχει το αντίστοιχο χρώμα στον τελικό συνδυασμό.

```
//Στα πρώτα δυο LEDs θέτουμε έντονο κόκκινο χρώμα, στα επόμενα δυο
//έντονο πράσινο, στα επόμενα δυο έντονο μπλε και στα τελευταία δυο λευκό

// έντονο κόκκινο
ourLEDs[0].setRGB(255,0,0); ourLEDs[1].setRGB(255,0,0);

// έντονο πράσινο
ourLEDs[2].setRGB(0,255,0); ourLEDs[3].setRGB(0,255,0);

// έντονο μπλε
ourLEDs[4].setRBG(0,0,255); ourLEDs[5].setRGB(0,0,255);

// λευκό
ourLEDs[6].setRGB(255,255,255); ourLEDs[7].setRGB(255,255,255);
```

3. Ενεργοποίηση των LEDs

```
for (int i = 0; i < 8; i++){
ourLEDs[i].setOn()
}
```

4. Απενεργοποίηση των LEDs

```
for (int i = 0; i < 8; i++){
ourLEDs[i].setOff()
}
```

Επίσης μπορούμε να εξετάσουμε την κατάσταση των LEDs χρησιμοποιώντας τις παρακάτω μεθόδους *isOn(), getRed(),getGreen(),* και *getBlue()*.

Κουμπιά – Switches

Το sensor board έχει δυο κουμπιά που διαχειρίζονται από την κλάση *Iswitch*. Υπάρχουν δυο τρόποι για να δουλέψουμε με τα κουμπιά του sensor board:

1. Ελέγχοντας συνεχώς την κατάσταση των switches(αν πιέζονται ή όχι) χρησιμοποιώντας ένα βρόγχο και τις μεθόδους waitForChange(); και isOpen().

```
Import com.sun.spot.sensorboard.EDemoBoard;
Import com.sun.spot.sensorboard.ISwitch;
ISwitch[] ourSwitches = EDemoBoard.getInstance().getSwitches();

if(ourSwitches[0].isOpen()){
// Αν ο διακόπτης ήταν ανοιχτός περιμένουμε για να πιεστεί
ourSwitches[0].waitForChange();
}
// Περιμένουμε για απελευθέρωση του διακόπτη
ourSwitches[0].waitForChange();
```

2. Σε περίπτωση που δεν θέλουμε να αναμένουμε για αναμένουμε για αλλαγή της κατάστασης του διακόπτη μπορούμε να χρησιμοποιήσουμε listeners. Συγκεκριμένα η εφαρμογή μας θα πρέπει να κάνει implements το IswitchListener interface και να περιλαμβάνει τις μεθόδους switchPressed(ISwitch sw) και switchReleased(ISwitch sw). Η πρώτη εκτελείται κάθε φορά που ο διακόπτης πιέζεται ενώ η δεύτερη κάθε φορά που ο διακόπτης απελευθερώνεται. Οι συναρτήσεις αυτές καλούνται με όρισμα τον διακόπτη που άλλαξε κατάσταση. Το παρακάτω παράδειγμα είναι ένα πλήρες MIDlet που τυπώνει “switch 1” όταν πιεστεί ο πρώτος διακόπτης και “switch 2” για τον δεύτερο.

```
Import com.sun.spot.sensorboard.EDemoBoard;
Import com.sun.spot.sensorboard.ISwitch;

public class BroadcastCount extends MIDlet implements ISwitchListener {

   protected void startApp() throws MIDletStateChangeException {
      ISwitch switches[] = EdemoBoard.getInstance().getSwitches();
      switches[0].addISwitchListener(this);
      switches[1].addISwitchListener(this);
  }
   protected void pauseApp() {
   }
   protected void destroyApp() {
   }

   public void switchReleased(ISwitch sw) {
   }

   public void switchPressed(ISwitch sw) {
      if (sw == switches[0]) {
         System.out.println(“switch 1”);
```

```
    } else {
        System.out.println("switch 2");
    }
}
```

Αισθητήρας φωτεινότητας

Ο μετρητής φωτεινότητας διαχειρίζεται από την κλάση *IlightSensor* του *EdemoBoard.* Για να πάρουμε μια τιμή για την στιγμιαία ένταση της φωτεινότητας του περιβάλλοντος μπορούμε να χρησιμοποιήσουμε την μέθοδο getalue(), όπως φαίνεται στο παρακάτω παράδειγμα:

```
Import com.sun.spot.sensorboard.EDemoBoard;
Import com.sun.spot.sensorboard.peripheral.ILightSensor;
ILightSensor ourLightSensor =
EdemoBoard.getInstance().getLightSensor();

int lightSensorReading = ourLightSensor.getValue();
```

Αν η πηγή που μετράμε δεν έχει σταθερή φωτεινότητα, όπως οι λάμπες φθορισμού, η παραπάνω μέθοδος δεν θα έχει τα αναμενόμενα αποτελέσματα. Αν και στο ανθρώπινο μάτι δεν είναι ορατές οι γρήγορες αλλαγές στην ένταση, ο αισθητήρας θα μας δίνει τιμές για τις γρήγορες διακυμάνσεις. Ένας τρόπος για να ξεπεράσουμε αυτό το πρόβλημα είναι να υπολογίζουμε τον μέσο όρο της φωτεινότητας παίρνοντας ένα ικανοποιητικό αριθμό δειγμάτων. Αυτό γίνεται αυτόματα με την μέθοδο *getAverageValue(n)* που επιστρέφει τον μέσο όρο n δειγμάτων που λαμβάνονται διαδοχικά χρησιμοποιώντας 1ms καθυστέρηση στις μεταξύ τους μετρήσεις. Η προκαθορισμένη τιμή για τα δείγματα αν δεν δοθεί όρισμα είναι 17. Οι τιμές που επιστρέφουν οι παραπάνω μέθοδοι κυμαίνονται από 0 ως 750, με 0 αναπαριστούμε το απόλυτο σκοτάδι. Στον παρακάτω πίνακα δίνουμε την αντιστοιχία των τιμών που δίνει ο αισθητήρας σε διαφορετικές συνθήκες φωτεινότητας:

Luminance	Sensor Reading
1000 lx	497
100lx	50
10lx	5

Αισθητήρας θερμοκρασίας

Πρόκειται για τον πιο απλό από όλους τους αισθητήρες, η κλάση που σχετίζεται με αυτόν παρέχει μεθόδους για την άμεση λήψη τιμών σε κλίμακα Celsius ή Fahrenheit. Όμως επειδή πρόκειται για έναν ενσωματωμένο αισθητήρα οι τιμές επηρεάζονται από πηγές θερμότητας στο εσωτερικό του SPOT, όπως cpu, κυκλώματα φόρτισης κλπ. Αν χρειαζόμαστε αξιόπιστες μερίσεις της θερμοκρασίας του περιβάλλοντος προτείνεται η χρήση ενός εξωτερικού μετρητή που μπορούμε να συνδέσουμε με το SPOT μέσω των GIO θυρών. Ακολουθεί ένα παράδειγμα μέτρησης θερμοκρασίας:

```
Import com.sun.spot.sensorboard.EDemoBoard;
Import com.sun.spot.sensorboard.io.ITemperatureInput;
ITemperatureInput ourTempSensor = EdemoBoard.getADCTemperature();

double celsiusTemp = ourTempSensor.getCelsius();
double fahrenheitTemp= ourTempSensor.getFahrenheit();
```

2.2 Εγκατάσταση και εκτέλεση εφαρμογών στα SPOTs

Σε αυτή την ενότητα θα περιγράψουμε την διαδικασία μετάφρασης εγκατάστασης και εκτέλεσης μιας εφαρμογής σε μια συσκευή SUN SPOT, συγκεκριμένα θα εγκαταστήσουμε την εφαρμογή BounceDemo που παρέχεται μαζί με το SDK. Όλες οι παραπάνω διαδικασίες γίνονται, όπως θα δούμε αυτοματοποιημένα με την χρήση ant scripts. Υποθέτουμε ότι έχουμε ήδη εγκαταστήσει τα εργαλεία και τα προγράμματα που υπάρχουν στο cd του αναπτυξιακού kit, όπως το SUN SPOT SDK και το ant στις προκαθορισμένες τοποθεσίες.

1. Μετάφραση της εφαρμογής και δημιουργία αρχείου jar, χρησιμοποιώντας την εντολή "*ant jar-app*" στο φάκελο "*C:\Sun\SunSPOT\Demos\BounceDemo\BounceDemo-OnSPOT*". Το jar, μετά την εκτέλεση της εντολής, δημιουργείται στο φάκελο suite και του δίνεται ένα όνομα της μορφής "*< MIDlet-Name>_<MIDlet-Version>.jar*". Οι τιμές αυτές βρίσκονται στο αρχείο MANIFEST.MF της εφαρμογής. Στην συγκεκριμένη περίπτωση το όνομα που θα προκύψει είναι "eSPOT Bounce Demo-OnSPOT_1.0.0.jar".
2. Σύνδεση του SUN SPOT με τον υπολογιστή μεσώ ενός mini-USB καλωδίου.
3. Το επόμενο βήμα είναι ο έλεγχος της σωστής επικοινωνίας του υπολογιστή με το SPOT. Αυτό γίνεται με την εντολή "*ant info*", η οποία τυπώνει πληροφορίες σχετικά με την συσκευή.
4. Το επόμενο βήμα είναι η εγκατάσταση της εφαρμογής στο SPOT, χρησιμοποιώντας την εντολή "*ant jar-deploy*". Με αυτή την εντολή μπορούμε να εγκαταστήσουμε οποιοδήποτε κατάλληλο jar έχουμε δημιουργήσει ή μας έχουν δώσει, δίνοντας απλώς την εντολή *"ant jar-deploy -Djar.file=<name>.jar" .*
5. Τέλος για να εκτελέσουμε την εφαρμογή χρησιμοποιούμε την εντολή "*ant run*".

Κατά την διάρκεια εκτέλεσης τα streams εξόδου(System.out και System.err) της εφαρμογής τυπώνονται στο τερματικό που δώσαμε την εντολή "*ant run*". Αντί για την παραπάνω διαδικασία θα μπορούσαμε να αντικαταστήσουμε την εντολή "*ant jar-app*" και "*ant jar-deploy*" με την εντολή "*ant deploy*", που έχει την ίδια λειτουργία με τις δυο προηγούμενες. Επιπλέον επειδή το ant δέχεται πολλαπλές εντολές στην σειρά, μπορούμε να ενσωματώσουμε και την εντολή "*ant run*" σε μια εντολή "*ant deploy run*" που μεταφράζει, εγκαθιστά και εκτελεί το την εφαρμογή.

Μια ακόμα δυνατότητα που μας δίνεται είναι να χρησιμοποιούμε κλάσεις από ήδη υπάρχοντα jars. Αυτά τα jar αναφέρονται ως utility jars και έχουν δημιουργήσει με την εντολή "*ant jar-app*". Αν θέλουμε να χρησιμοποιούμε τέτοια jar στην εφαρμογή μας πρέπει το δηλώσουμε με την επιλογή *"-Dutility.jars=<filename>"*. Για παράδειγμα "*ant deploy –Dutility.jars=util.jar*". Στην περίπτωση που χρειάζεται να προσθέσουμε περισσότερα jars, πρέπει να τα χωρίσουμε με τον χαρακτήρα ":" ή ";".

2.3 Ανάπτυξη πρότυπης εφαρμογής

Σε αυτή την ενότητα θα ασχοληθούμε με την ανάπτυξη μιας απλής εφαρμογής για τα SUN SPOT που θα χρησιμοποιεί το radio για μετάδοση μηνυμάτων, τα LED για ένδειξη παραλαβής μηνυμάτων από άλλα SPOT, και τα switches της συσκευής που θα προκαλούν την μετάδοση μηνυμάτων. Ο σκοπός αυτής της ενότητας είναι η εξοικείωση του αναγνώστη με την δομή των MIDlets και των βασικών κλάσεων της βιβλιοθήκης των SUN SPOT μέσα από ένα πλήρες λειτουργικό παράδειγμα. Η εφαρμογή που θα παρουσιάσουμε, θα χρησιμοποιεί το

Δημιουργία των κατάλληλων φακέλων του project

Αν έχουμε εγκαταστήσει το NetBeans IDE μπορούμε να δημιουργήσουμε μια νέα εφαρμογή για τα SUN SPOT από το μενού File -> New Project και επιλέγοντας "Sun SPOT Application" από την κατηγορία "General". Το NetBeans θα δημιουργήσει τους κατάλληλους φακέλους και αρχεία , και ένα βασικό MIDlet για να ξεκινήσουμε. Αν δεν έχουμε NetBeans μπορούμε να χρησιμοποιήσουμε σαν template το κώδικα που μας δίνεται από το SUN SPOT SDK στον φάκελο "C:\Sun\SunSPOT\Demos\CodeSamples\SunSpotApplicationTemplate", αντιγράφοντας τον κατάλογο αυτό σε μια άλλη θέση που θα αναπτύξουμε την εφαρμογή μας. Αν επιλέξουμε την δεύτερη επιλογή, θα πρέπει να αλλάξουμε στο αρχείο MANIFEST.MF τις τιμές που σχετίζονται με το όνομα της εφαρμογής μας.

Κώδικας και επεξήγηση της εφαρμογής

Στην αρχή περιλαμβάνουμε όλες τις μεταβλητές που είναι αναγκαίες για την εφαρμογή και στιγμιότυπα των κλάσεων που διαχειρίζονται τα LEDs και τα Switchs. Η μεταβλητή color χρησιμοποιείται σαν δείκτης για τον πίνακα colors, και παίρνει τιμές {1,2,3} , αυτές αντιστοιχούν στα χρώματα κόκκινο, πράσινο, μπλε. Η μεταβλητή αυτή δηλώνει το τρέχων χρώμα των LEDs του SPOT. Η μεταβλητή count, αποθηκεύει τον αριθμό που σχηματίζεται στα LEDs(ως δυαδική αναπαράσταση). Τέλος έχουμε μια μεταβλητή tx για τον broadcast connection και άλλη μια xdg για τα datagram πακέτα που θα στέλνουμε.

```
public class BroadcastCount extends MIDlet implements ISwitchListener {

    private static final int CHANGE_COLOR = 1;
```

```
private static final int CHANGE_COUNT = 2;

private ITriColorLED leds[] = EDemoBoard.getInstance().getLEDs();
private ISwitch switches[] = EDemoBoard.getInstance().getSwitches();
private int count = -1;
private int color = 0;
private LEDColor[] colors = { LEDColor.RED, LEDColor.GREEN, LEDColor.BLUE };
private RadiogramConnection tx = null;
private Radiogram xdg;
```

Ακολουθεί το κύριο σώμα της εφαρμογής. Στην αρχή θέτουμε τα LEDs στο προκαθορισμένο χρώμα(πράσινο) με την συνάρτηση showColor() , προσθέτουμε listeners για τα δυο switches και ανοίγουμε δυο broadcast connections, ένα tx για την αποστολή μηνυμάτων και ένα rx για την λήψη. Και τα δυο χρησιμοποιούν την port 123. Ακολουθεί ένας ατέρμων βρόγχος στον όποιο περιμένουμε για την λήψη ενός πακέτου. Κάθε πακέτο αποτελείται από 3 integers, ο πρώτος δηλώνει την εντολή(1 για αλλαγή χρώματος των LEDs και 2 για αλλαγή του αριθμού που σχηματίζουν), ο δεύτερος είναι ο αριθμός που πρέπει να θέσουμε στα LEDs, και ο τρίτος το χρώμα. Όταν παραλειφθεί ένα datagram διαβάζουμε αυτές τις τρεις τιμές και στην συνέχεια ανάλογα με την εντολή(αλλαγή χρώματος ή αριθμού), εκτελούμε την αντίστοιχη συνάρτηση(showColor(), showCount()).

```
protected void startApp() throws MIDletStateChangeException {
    System.out.println("Broadcast Counter MIDlet");
    showColor(color);
    RoutingPolicy rp = new RoutingPolicy(RoutingPolicy.ALWAYS);
    RoutingPolicyManager.getInstance().policyHasChanged(rp);
    switches[0].addISwitchListener(this);
    switches[1].addISwitchListener(this);

    try {
        tx = (RadiogramConnection)Connector.open("radiogram://broadcast:123");
        xdg = (Radiogram)tx.newDatagram(20);
        RadiogramConnection rx = (RadiogramConnection)Connector.open("radiogram://:123");
        Radiogram rdg = (Radiogram)rx.newDatagram(20);
        while (true) {
            try {
                rx.receive(rdg);
                    System.out.println("Received packet from " + rdg.getAddress());

                int cmd = rdg.readInt();
                int newCount = rdg.readInt();
                int newColor = rdg.readInt();
                if (cmd == CHANGE_COLOR) {
                    System.out.println("Received packet from " + rdg.getAddress());
                    showColor(newColor);
                } else {
                    showCount(newCount, newColor);
```

```
                }
            } catch (IOException ex) {
                System.out.println("Error receiving packet: " + ex);
                ex.printStackTrace();
            }
        }
    } catch (IOException ex) {
        System.out.println("Error opening connections: " + ex);
        ex.printStackTrace();
    }
}

protected void pauseApp() {
    // This will never be called by the Squawk VM
}

protected void destroyApp(boolean arg0) throws MIDletStateChangeException {
    // Only called if startApp throws any exception other than MIDletStateChangeException
}
```

Στην συνέχεια έχουμε τις συναρτήσεις για αλλαγή χρώματος και αριθμού στα LEDs, που καλούνται όταν παραλαμβάνουμε πακέτα με τις αντίστοιχες εντολές.

```
private void showCount(int count, int color) {
    for (int i = 7, bit = 1; i >= 0; i--, bit <<= 1) {
        if ((count & bit) != 0) {
            leds[i].setColor(colors[color]);
            leds[i].setOn();
        } else {
            leds[i].setOff();
        }
    }
}
private void showColor(int color) {
    for (int i = 0; i < 8; i++) {
        leds[i].setColor(colors[color]);
        leds[i].setOn();
    }
}
```

Τέλος οι παρακάτω δυο συναρτήσεις, υλοποιούν το IswitchListener interface και καλούνται όταν έχουμε κάποια αλλαγή στην κατάσταση του switch του SPOT. Αυτή η εφαρμογή ανταποκρίνεται στο πλήρες “πάτημα” ενός switch, δηλαδή όταν πιέσουμε και στην συνέχεια απελευθερώσουμε ένα διακόπτη. Όταν μια τέτοια ενέργεια ανιχνευθεί το SPOT καλεί την συνάρτηση switchReleased(). Σε αυτήν ελέγχουμε ποιος διακόπτης έχει απελευθερωθεί, αν πρόκειται για τον αριστερό τότε στέλνουμε την εντολή για αλλαγή χρώματος στα γειτονικά SPOT, ενώ σε αντίθετη περίπτωση στέλνουμε εντολή για αλλαγή του αριθμού που δείχνουν τα LEDs.

```
public void switchReleased(ISwitch sw) {
```

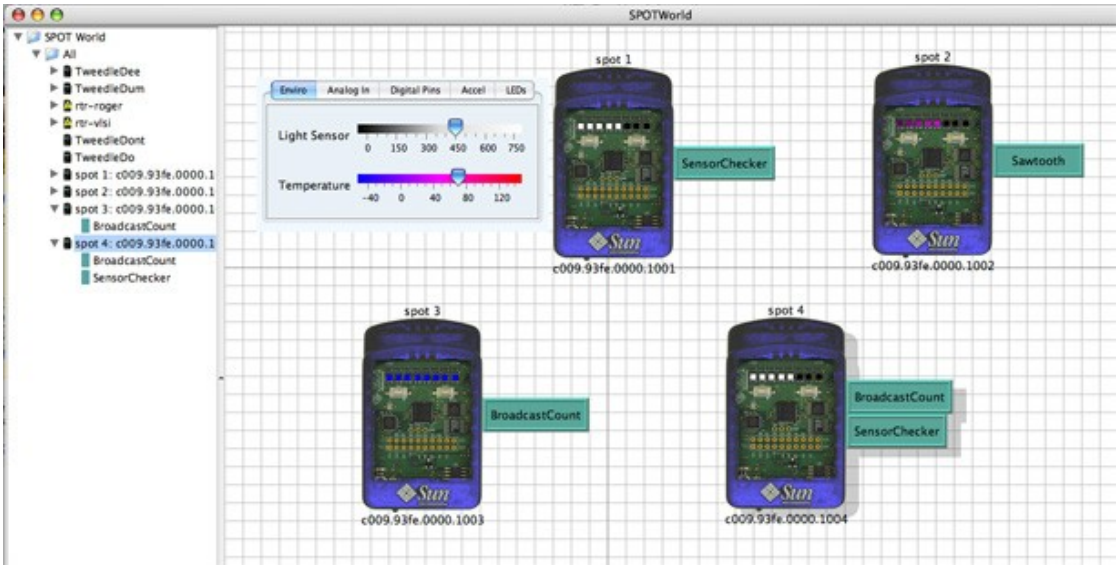
```
    int cmd;
    if (sw == switches[0]) {
        cmd = CHANGE_COLOR;
        if (++color >= colors.length) { color = 0; }
        count = -1;
    } else {
        cmd = CHANGE_COUNT;
        count++;
    }
    try {
        System.out.println("Sending packet to ");
        xdg.reset();
        xdg.writeInt(cmd);
        xdg.writeInt(count);
        xdg.writeInt(color);
        tx.send(xdg);
    } catch (IOException ex)  {
        System.err.println("Error sending packet: " + ex);
        ex.printStackTrace();
    }
}

public void switchPressed(ISwitch sw) {
}
```

3.0 Εξομοιωτής για τα SUN SPOT

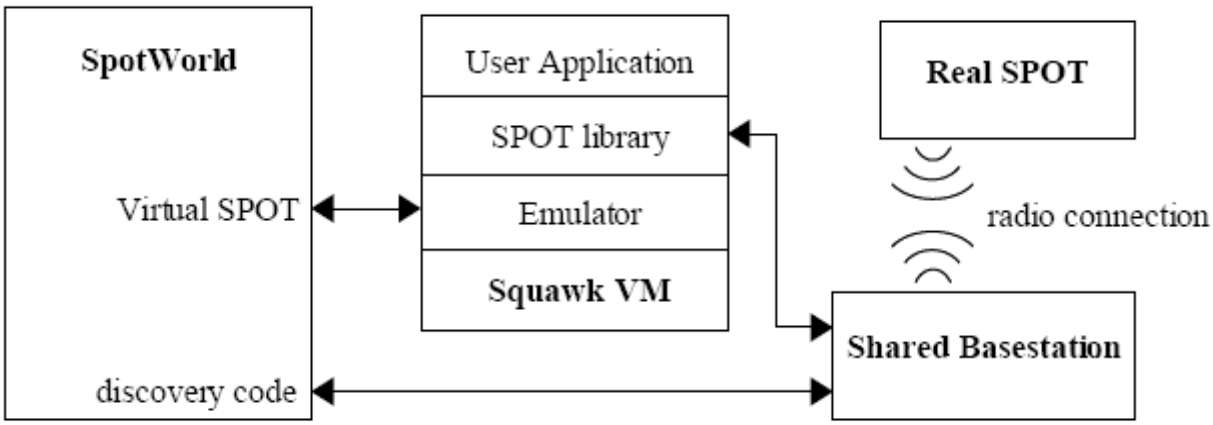
3.1 Το εργαλείο SPOTWORLD

Μαζί με τα SPOT η Sun παρέχει ένα εύχρηστο εργαλείο το SPOTWORLD [13] που περιλαμβάνει ένα εξομοιωτή, και μας επιτρέπει να εκτελέσουμε τις εφαρμογές μας στον υπολογιστή πριν τις εγκαταστήσουμε στα SPOT. Ακόμα και αν δεν έχουμε μια πραγματική συσκευή μπορούμε να χρησιμοποιήσουμε το SPOTWORLD για να τρέχουμε τα προγράμματά μας. Το SPOTWORLD μας επιτρέπει να χειριζόμαστε εικονικά SPOT, και μέσω ενός control panel να ελέγχουμε και το sensor board, για παράδειγμα μπορούμε να αλλάξουμε τις τιμές του επιταχυνσιομέτρου, του αισθητήρα θερμοκρασίας, των ψηφιακών εισόδων κλπ. Επίσης ο χρήστης μπορεί να ελέγξει με το ποντίκι τα δυο κουμπιά που εμφανίζονται στην φωτογραφία του εικονικού SPOT. Η εφαρμογή που τρέχει στα εικονικά SPOT έχει την δυνατότητα να αλλάζει το χρώμα των LED να διαβάζει τιμές από όλους τους αισθητήρες και να χρησιμοποιεί το radio όπως αν έτρεχε σε ένα κανονικό SPOT. Κάθε εικονικό SPOT έχει την δική του διεύθυνση για να μπορεί να στέλνει και να λαμβάνει δεδομένα από άλλες συσκευές. Επιπλέον αν έχουμε ένα basestation στον υπολογιστή που τρέχουμε το SPOTWORLD τότε τα εικονικά SPOT μπορούν ασύρματα να επικοινωνήσουν με τις πραγματικές συσκευές. Στην παρακάτω εικόνα βλέπουμε το SPOTWORLD με 4 εικονικά SPOT να τρέχουν διαφορετικές εφαρμογές.



3.2 Η λειτουργία του εξομοιωτή

Για κάθε εικονικό SPOT που δημιουργούμε το SPOWORLD εκκινεί μια Squawk VM και σε αυτή τρέχει τον Emulator. Ο κώδικας του εξομοιωτή, αφού τρέχει στην Squawk VM, είναι γραμμένος σε java ME και χρησιμοποιεί το GCF (Generic Connection Framework) για όλα τα socket connections που δημιουργεί. Ο Emulator συνδέεται μέσω socket στο SPOTWORLD για την αποστολή δεδομένων σχετικά με τις αλλαγές στην συσκευή, που προκαλούνται από την εφαρμογή που τρέχει. Για παράδειγμα αν η εφαρμογή αλλάξει το χρώμα ενός LED, ο Emulator θα στείλει αυτή την πληροφορία στο SPOTWorld ώστε να ενημερώσει την απεικόνιση του εικονικού SPOT. Αντίστοιχα όταν ο χρήστης αλλάξει κάποια τιμή στο sensor board μέσω του control panel,ή “πιέσει” με το ποντίκι του ένα από τα κουμπιά στο εικονικό SPOT, τότε αυτές οι αλλαγές θα σταλούν στον Emulator και θα είναι άμεσα διαθέσιμες στα τρέχουσα εφαρμογή. Επιπλέον ο Emulator προωθεί τα System.err και System.out streams της εφαρμογής στο SPOTWorld ώστε ο χρήστης να μπορεί να παρακολουθεί την εκτέλεσή της. Στο παρακάτω διάγραμμα βλέπουμε την αρχιτεκτονική του Emulator.



Πρωτόκολλο επικοινωνίας εξομοιωτή

Ο εξομοιωτής έχει ένα συγκεκριμένο πρωτόκολλο επικοινωνίας για την μετάδοση αλλαγών στην κατάσταση του εικονικού SPOT από και προς το SPOTWorld, αλλά και για τον έλεγχο της εξομοίωσης. Το πρωτόκολλο είναι αρκετά απλό, χρησιμοποιεί ένα σύνολο από εντολές της μορφής <εντολή> [<όρισμα 1> .. <όρισμα n>]. Συγκεκριμένα οι έγκυρες εντολές προς στον εξομοιωτή είναι:

- Switch 1/2 pressed/released
- InputPin index high/low
- Light val
- Temperature val

- Voltage index val
- Accelerometer x y z
- Deploy <file>
- Name <name>
- Run #midlet
- Kill #isolate
- System.in msg
- Exit

Η διαδικασία για την εκκίνηση μιας εφαρμογής στον εξομοιωτή, περιλαμβάνει την αποστολή του path του jar που περιέχει τα MIDlet της εφαρμογής(Deploy <file>) , και την επιλογή του MIDlet που θέλουμε να εκτελέσουμε.

Τα μηνύματα που στέλνει ο εξομοιωτής προς το SPOTWorld για την κατάσταση της εφαρμογής και του SPOT είναι:

- LED index r g b
- Red on/off
- Green on/off
- PinDirection index out/in
- OutputPin index high/low
- AccScale val
- System.out msg
- System.err msg
- Running #midlet #isolate
- Done #isolate
- Ready
- Goodbye

Δομή του Εξομοιωτή

Προκειμένου να παρέχεται η δυνατότητα για αλληλεπίδραση με το sensor board αλλά και επικοινωνία μέσω radio με άλλα SPOT ο emulator έχει ένα σύνολο κλάσεων που εξομοιώνουν την συμπεριφορά των περισσοτέρων υποσυστημάτων της συσκευής. Τα αρχεία του πηγαίου κώδικα του εξομοιωτή είναι διαθέσιμα στο C:\Sun\SunSPOT\sdk\src\emulator_source.jar. Η πιο σημαντική κλάση που παρέχεται είναι η SPOT.java που είναι στην κορυφή της ιεραρχίας της βασικής βιβλιοθήκης. Όλες οι κλάσεις της βιβλιοθήκης και των εφαρμογών μοιράζονται ένα κοινό αντικείμενο αυτής της κλάσης που διαχειρίζεται την πρόσβαση στα στοιχεία του hardware. Η κλάση SPOT είναι η πρώτη που εκκινείται από την Squawk VM.

Η κλάση Emulator είναι υπεύθυνη για την διατήρηση της πληροφορίας σχετικά με τις τιμές των διαφορετικών αισθητήρων, αυτές είναι διαθέσιμες

στο SPOTWorld μέσω του πρωτοκόλλου που περιγράψαμε στην προηγούμενη ενότητα αλλά και στις υπόλοιπες κλάσεις που εξομοιώνουν το hardware. Στην παρούσα έκδοση δεν υποστηρίζονται όλα τα υποσυστήματα, η εφαρμογές που τρέχουν στον εξομοιωτή έχουν πρόσβαση στα:

- LEDs
- ψηφιακές εισόδους/εξόδους
- αισθητήρα θερμοκρασίας
- επιταχυνσιόμετρο
- αισθητήρα φωτεινότητας
- Διακόπτες
- Ασύρματη επικοινωνίας

Η διαχείριση του hardware σε χαμηλό επίπεδο, και μερικά στοιχεία του sensor board(UART,PWM) δεν υποστηρίζεται.

Εξομοίωση της ασύρματης επικοινωνίας

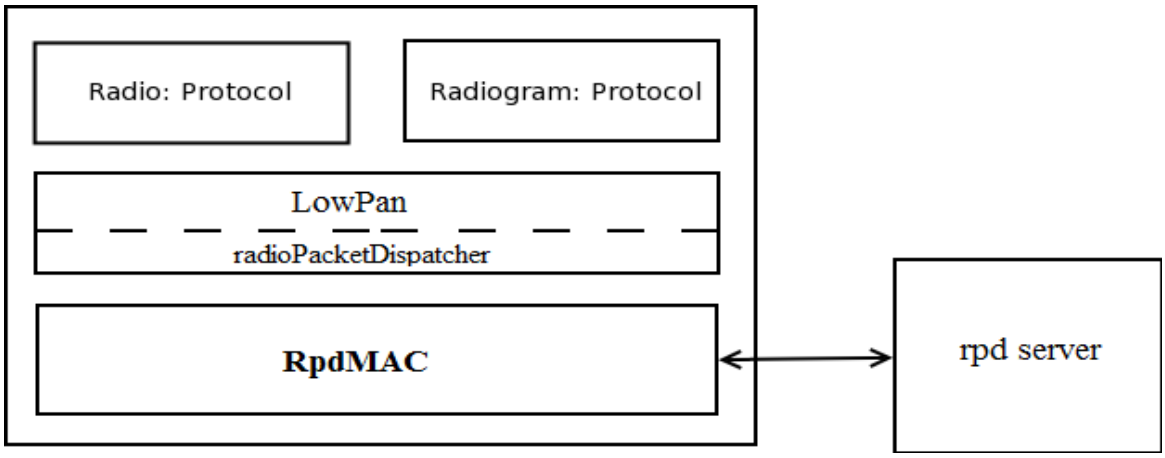
Όπως προείπαμε ο εξομοιωτής δίνει την δυνατότητα στις εφαρμογές να χρησιμοποιούν radio connections(πχ radiostream, radiodram), για την μετάδοση broadcast πακέτων ή για point-to-point συνδέσεις . Για να επιτύχει την αυτή την λειτουργία, ο εξομοιωτής τρέχει μια τροποποιημένη έκδοση της κλάσης της βασικής βιβλιοθήκης που διαχειρίζεται το MAC layer του 802_15_4 (την SocketMac), η οποία χρησιμοποιεί multicast sockets για την μετάδοση των broadcast πακέτων και unicast sockets για τα point-to-point radio connections. Τα πιο σημαντικά στοιχεία αυτής της κλάσης είναι δυο thread που ξεκινάνε κατά την αρχικοποίησή της, τα CommandRxThread και BroadcastPacketRxThread που λαμβάνουν πακέτα μέσω των multicast sockets που διαχειρίζονται οι αντίστοιχες κλάσεις commandBroadcastChannel και packetBroadcastChannel.

Για την αποστολή ενός πακέτου προς ένα συγκεκριμένο κόμβο, πρώτα αναζητούμε αν έχουμε ήδη ένα socket connection προς τον προορισμό. Αν δεν έχουμε(στέλνουμε πρώτη φορά πακέτα προς αυτόν), ζητάμε από τον προορισμό να συνδεθεί με ένα socket connection στον αποστολέα. Αυτό πραγματοποιείται στέλνοντας μέσω του commandBroadcastChannel την διεύθυνση του προορισμού, την port του serversocket για να συνδεθεί και την διεύθυνση του αποστολέα σε όλους του κόμβους. Όταν ένας κόμβος παραλάβει την παραπάνω πληροφορία εξετάζει είναι ο προορισμός και αν είναι συνδέεται στο αποστολέα. Μέσω αυτού του socket μεταδίδονται τα radiopackets καθώς και τα ack packets.

Για την broadcast μετάδοση ενός radiopacket χρησιμοποιούμε το packetBroadcastChannel για την αποστολή των bytes του πακέτου στο σύνολο των κόμβων που τρέχουν. Κατά την λήψη ενός broadcast ελέγχουμε ότι η διεύθυνση του αποστολέα δεν είναι ίδια με του παραλήπτη, και στην συνέχεια προσθέτουμε το πακέτο σε μια ουρά(την rxQueue), για την μετέπειτα παράδοση σε ανώτερα layers του radiostack(π.χ. lowpan).

3.3 Επέκταση της λειτουργικότητας του εξομοιωτή

Αν και ο SUN SPOT Emulator παρέχει μια βασική υποστήριξη για network communication, δεν υποστηρίζει network topology. Όλα τα emulated SPOTs συμπεριφέρονται σαν να βρίσκονται στην ίδια φυσική τοποθεσία και σχηματίζουν τοπολογία ενός πλήρη γράφου. Στα πλαίσια της διπλωματικής ένα από τα θέματα που ασχοληθήκαμε ήταν η επέκταση του εξομοιωτή ώστε να υποστηρίζει network topology. Για να επιτύχουμε την παραπάνω λειτουργικότητα χρειάστηκε να υλοποιήσουμε ένα σύστημα για την εξομοίωση της τοπολογίας(τον rpd server), και την τροποποίηση του MAC layer του εξομοιωτή ώστε να προωθεί και να λαμβάνει τα radio packets(τα πακέτα του 802.15.4) από τον rpd server. Στην παρακάτω εικόνα βλέπουμε την αρχιτεκτονική του radiostack του εξομοιωτή.



Τροποποίηση του του MAC layer

Η κλάση αυτή βρίσκεται στο αρχείο RdpMac.java και αποτελεί το τελευταίο επίπεδο στο radiostack πριν το physical layer. Τα πακέτα από τα υψηλού επιπέδου πρωτόκολλα (πχ radiograms), κερματίζονται σε radiopackets από το lowpan και παραδίδονται από το radioPacketDispatcher στο RdpMac για αποστολή. Κατά την αρχικοποίηση της κλάσης, συνδέεται στον rpd server μέσω socket connection και εκκινεί ένα thread που αναλαμβάνει την αποστολή και λήψη radiopacketsκαι ack packet. Το thread αυτό είναι το DispatcherServerSocketListener, οι λειτουργία του και οι κύριες μέθοδοι που χρησιμοποιεί παρουσιάζονται παρακάτω:

Κατά την δημιουργία της κλάσης ανοίγουμε τα stream εισόδου και εξόδου με τον rpd server και ενγράφουμε σε αυτόν την εφαρμογή στέλνοντας την διεύθυνση της εικονικής συσκευής. Επίσης δημιουργούμε μια ουρά(Queue) για τα πακέτα ack.

```
private class DispatcherServerSocketListener extends Thread {
    private static final long ACK_TIMEOUT = 1000;
    private InputStream in;
    private OutputStream out;
    private boolean closed;
    private StreamConnection clientSocket;
    private Queue ackQueue;
```

```
    private Integer sendLock = new Integer(0);

    public DispatcherServerSocketListener(StreamConnection clientSocket) throws IOException
{
        super("DispatcherServer socket listener");
        this.clientSocket = clientSocket;
        in = clientSocket.openInputStream();
        out = clientSocket.openOutputStream();
        ackQueue = new Queue();
        System.err.println("extenedAddress is ="+extendedAddress);

        byte destAddress[] = new byte[8];
        Utils.writeBigEndLong(destAddress, 0, extendedAddress);

        out.write(destAddress,0,destAddress.length);
        out.flush();
}
```

Οι παρακάτω συναρτήσεις χρησιμοποιούνται για την αποστολή radiopackets προς τον server. Όλα τα δεδομένα του radiopacket βρίσκονται σε ένα byte array που στέλνεται μέσω του socket. Η sendto() στέλνει πακέτα σε ένα συγκεκριμένο προορισμό ενώ η broadcast() σε όλους τους γείτονες. Πριν την αποστολή του πακέτου στέλνουμε την διεύθυνση προορισμού, ενώ στην broadcast περίπτωση η διεύθυνση αποστολής είναι FFFF(hex), όπως περιγράφεται από το πρωτόκολλο 802.15.4.
Η μέθοδος waitForAck() καλείται μετά την αποστολή ενός πακέτου και αναμένει την παραλαβή πακέτου επιβεβαίωσης από τον προορισμό. Επιστρέφει true αν λάβουμε το σωστό ack, ενώ σε περίπτωση που δεν έχουμε λάβει επιβεβαίωση μέσα σε ένα διάστημα ACK_TIMEOUT από την αποστολή επιστρέφει false.

```
    private void  sendto(long addr, byte[] buffer, int len) throws IOException{
        synchronized(sendLock)
        {
            byte destAddress[] = new byte[8];
            Utils.writeBigEndLong(destAddress, 0, addr);

            out.write(destAddress,0,destAddress.length);
            out.flush();
            out.write(buffer, 0, len);
            out.flush();
        }
    }

    public  void broadcast(byte[] byteArray) throws IOException {
        sendto((long)0xFFFF,byteArray,byteArray.length);
    }

    public boolean waitForAck(byte dataSequenceNumber) throws IOException {
        RadioPacket nextAck = (RadioPacket) ackQueue.get(1000);
        while (nextAck != null) {
            if (nextAck.getDataSequenceNumber() == dataSequenceNumber) {
```

```
                return true;
            }
            nextAck = (RadioPacket) ackQueue.get(ACK_TIMEOUT);
        }
            System.out.println("[SocketMAC] Timeout waiting for ack");

        return false;
    }
```

Η μέθοδος run() περιμένει την λήψη radiopackets από τον server και αναλόγως τα τοποθετεί στις κατάλληλες ουρές. Κάθε radiopacket αποτελείται από 128 bytes τα οποία αποθηκεύουμε για κάθε πακέτο που λαμβάνουμε στο array data[]. Στην αρχή κάθε πακέτου αποστέλλεται ένα επιπλέον byte που χρησιμοποιείται να να διακρίνουμε το είδος του πακέτου. Αν το πρώτο byte είναι 1 τότε πρόκειται για broadcast ενώ σε διαφορετική περίπτωση έχουμε πακέτο επιβεβαίωσης ή δεδομένων. Αν λάβουμε πακέτο broadcast ελέγχουμε αν η διεύθυνση του αποστολέα είναι ίδια με την δικιά μας, σε αυτήν την περίπτωση το απορρίπτουμε αλλιώς το τοποθετούμε στην ουρά με τα εισερχόμενα πακέτα(rxQueue). Αν λάβουμε ένα πακέτο επιβεβαίωσης τότε το τοποθετούμε απευθείας στην ουρά των ack packets(ackQueue). Σε περίπτωση που λάβουμε πακέτο δεδομένων το τοποθετούμε στην ουρά εισερχομένων πακέτων(rxQueue) και κατασκευάζουμε ένα πακέτο επιβεβαίωσης με τον ίδιο ακολουθιακό αριθμό(Sequence Number) και το στέλνουμε στον αποστολέα.

```
    public void run() {
        while (true) {
            RadioPacket incoming = RadioPacket.getAckPacket();
            byte data[] = new byte[128];

            try {
                try {
                    int checkifpacketisbcast = in.read();
                    if(checkifpacketisbcast == 1)
                    {
                        RadioPacket rp = RadioPacket.getBroadcastPacket();
                        int len = in.read(data, 0, data.length);
                        for (int i = 0; i < len; i++) {
                            rp.buffer[i] = data[i];
                        }
                        rp.decodeFrameControl();
                        if (rp.getSourceAddress() != extendedAddress) {
                            if(verb)
                                System.err.println(extendedAddress+" Processing incoming broadcast data:
" + rp);
                            rxQueue.put(rp);
                        }
                        continue;
                    }

                    int lengthRead = in.read(incoming.buffer, 0, incoming.buffer.length);
                    if(lengthRead!=128)
```

```
                {
                 System.err.println("!!!!!!!!!!Read short incoming packet!!!!!!!!!!");

                }
                incoming.decodeFrameControl();
                if (incoming.isAck()) {
                                debug(extendedAddress+"","Getting ack packet ");
                   ackQueue.put(incoming);
                } else {

                   RadioPacket ackPacket = RadioPacket.getAckPacket();
                   ackPacket.setDSN(incoming.getDataSequenceNumber());
                   debug(extendedAddress+"","sending ack Header len= "+ackPacket.buffer[0]+1);
                   sendto(incoming.getSourceAddress(),ackPacket.buffer,ackPacket.buffer.length);

                   rxQueue.put(incoming);
                }
             } catch (IOException e) {
                e.printStackTrace();
                System.err.println("[DispatcherServerSocketListener] Closing (server down?)");
                return;
             }
          } catch (Exception e) {
                e.printStackTrace();
                return;
          }
       }
    }
```

Η κλάση RdpMac παρέχει δύο μεθόδους ώστε τα υψηλότερα layers να παραλαμβάνουν και να στέλνουν radiopackets, οι οποίες υπαγορεύονται από το I802.15.4MAC interface. Η μέθοδος mcpsDataRequest() καλείται από τον radioPacketDispatcher για την αποστολή ενός πακέτου από το RpdMAC. Πρώτα θέτει ένα νέο ακολουθιακό αριθμό(Sequence Number) στο πακέτο, και έπειτα εξετάζει αν πρόκειται για broadcast. Στην περίπτωση που η διεύθυνση αποστολής είναι 0xFFFF (broadcast) τότε αποστέλλεται μέσω της μεθόδου broadcast() του DispatcherServerSocketListener και επιστρέφεται άμεσα SUCCESS αφού στον broadcast μηχανισμό δεν έχουμε πακέτα επιβεβαίωσης. Σε διαφορετική περίπτωση μετά την αποστολή καλούμε την blocking μέθοδο waitForAck() περιμένοντας για πακέτο επιβεβαίωσης.

```
public synchronized int mcpsDataRequest(RadioPacket rp) {
   rp.setDSN(getDSN());
   int result = NO_ACK;
   try {

      if (rp.getDestinationAddress() == 0xFFFF) {
         packetRxThread.broadcast(rp.buffer);
         result = SUCCESS;
      } else {
```

```
            if (packetRxThread != null) {
              packetRxThread.sendto(rp.getDestinationAddress(),rp.buffer,rp.buffer.length);

                if (packetRxThread.waitForAck(rp.getDataSequenceNumber())) {
                   result = SUCCESS;
                }
              }
            }
        } catch (Exception e) {
           e.printStackTrace();
           throw new SpotFatalException("Got IOException while sending over SocketMac " +
e.getMessage());
        }
        return result;
    }
```

Η μέθοδος mcpsDataIndication() καλείται από τον radioPacketDispatcher για την παραλαβή ενός πακέτου από το RpdMAC. Η μέθοδος αυτή είναι blocking, εξαιτίας της υλοποίησης της μεθόδου get() της Queue στην βιβλιοθήκη του SUN SPOT, και καλείται για όσο υπάρχουν πακέτα στην ουρά.

```
    public void mcpsDataIndication(RadioPacket rp) {
       rp.copyFrom((RadioPacket) rxQueue.get());
    }
```

Υλοποίηση του rpd server

Ο rpd sever πρόκειται για ένα πρόγραμμα το οποίο διαμεσολαβεί στην επικοινωνία ανάμεσα στα εικονικά SPOT, παραλαμβάνει πακέτα από το RdpMAC και ανάλογα με την τοπολογία τα παραδίδει στον προορισμό. Τα αρχεία πηγαίου κώδικα υπάρχουν στον φάκελο radiopacketdispatcher. Ο διαβάζει την τοπολογία από ένα αρχείο με όνομα network.txt στο root directory. Ο τρόπος που ορίζουμε την τοπολογία στο αρχείο ο εξής:

- Κάθε γραμμή έχει μια ή περισσότερες διευθύνσεις σε δεκαεξαδική μορφή(hex dot)
- Η πρώτη διεύθυνση κάθε γραμμής είναι η διεύθυνση ενός κόμβου και οι υπόλοιπες των κόμβων που βρίσκονται εντός της εμβέλειάς του
- Κάθε κόμβος που θα συνδεθεί πρέπει να εμφανίζεται ακριβώς μια φορά στην αρχή κάποιας γραμμής
- Μια γραμμή με μόνο μία διεύθυνση στην αρχή δηλώνει ότι ο συγκεκριμένος κόμβος είναι απομονομένος (δεν συνδέεται με κανένα).
- Αν ένα link μεταξύ δυο κόμβων θέλουμε να είναι συμμετρικό τότε θα πρέπει να δηλώσουμε και τις δυο κατευθύνσεις.

Ο sever αποτελείται από πέντε κλάσεις, κάθε είναι υπεύθυνη για μια συγκεκριμένη υπηρεσία εκτός από την κλάση packet.

- Rdp – Είναι η κύρια κλάση του server, που αναλαμβάνει να αρχικοποιήσει

και να εκκινήσει τις υπόλοιπες κλάσεις. Μέτα την αρχικοποίηση δέχεται και καταχωρεί συνδέσεις από τα rdpMAC. Παρακάτω βλέπουμε τα βασικά βήματα για την εκκίνηση του Rdp από την main().

```
public static void main(String[] args) {

   rdp server = new rdp();
   //read setup of nodes topology
   server.readTopologyFromFile("network.txt");
   server.initPacketDispatcher();

   debug("main","starting accepting clients");
   server.startAcceptingClients();
   }
```

Η startAcceptingClients() διαχειρίζεται τις συνδέσεις με τα rdpMAC. Με το που συνδεθεί ένας emulator ξεκινάμε ένα νέο clientListener thread που αναλαμβάνει την περαιτέρω επικοινωνία.

```
void startAcceptingClients() {
   while(true) {
      try{

         StreamConnection clientSocket = serverSocket.acceptAndOpen();
         new clientListener(clientSocket,PacketDispatcher,this).start();

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

- clientListener – Πρόκειται για μια threaded κλάση που αναλαμβάνει την επικοινωνία με ένα συγκεκριμένο client-rdpMAC. Κάθε φορά που ένας client συνδέεται στον server εκκινείται ένα αντικείμενο αυτής της κλάσης. Περιλαμβάνει μεθόδους για την αποστολή και την λήψη radiopackets από το αντίστοιχο rdpMAC. Η προκαθορισμένη λειτουργίας της είναι να προσθέτει κάθε πακέτο που λαμβάνει στην ουρά του packetDispatcher, αυτό γίνεται γιατί η αποστολή και η εξέταση ύπαρξης link μπορεί να καταναλώνει αρκετό χρόνο(ειδικά αν πρόκειται για broadcast packet).
- packetDispatcher - Πρόκειται για μια threaded κλάση που εκκινείται με τον server και διατηρεί μια ουρά με πακέτα προς αποστολή. Τα πακέτα προτίθενται στην ουρά από τους clientListener. Για όσο υπάρχουν δεδομένα στην ουρά ο packetDispatcher τρέχει αλλιώς είναι blocked. Για κάθε πακέτο που αφαιρεί εξετάζει τον τύπο του και ανάλογα καλεί την sendBroadcast () ή την sendData (), οι οποίες ελέγχουν αν υπάρχει το κατάλληλο link, αν δεν υπάρχει το απορρίπτει ενώ σε αντίθετη περίπτωση βρίσκει τον κατάλληλο clientListener του προορισμού και μέσω αυτού το αποστέλλει.

```
   private void sendData(packet inpacket) throws IOException {
      clientListener cl =
RDP.getPacketDispatcherFor(inpacket.destAddress,inpacket.origAddress);
      if(cl != null) {

         cl.send(inpacket.data,inpacket.len);
         System.out.println(" OK");
      } else{
         System.out.println(" droped (no link)");
      }

   }
```

```
   private void sendBroadcast(packet inpacket) {

      for (Enumeration e = RDP.clientSockets.keys() ; e.hasMoreElements() ;) {
         Long addr = (Long)e.nextElement();
         System.out.print("Sending to: "+IEEEAddress.toDottedHex(addr.longValue()));

         if(NetworkTopology.LinkExist(addr ,inpacket.origAddress )) {
            if(addr.longValue() != inpacket.origAddress) {
               try {
                  ((clientListener)RDP.clientSockets.get(addr)).sendBcast(inpacket.data,inp
acket.len);
               } catch (IOException ex) {
                  ex.printStackTrace();
               }
               System.out.println(" OK");
            } else
               System.out.println("  droped");

         } else
            System.out.println("  droped");

      }
   }
```

- networkTopology – Η κλάση αυτή φορτώνει την τοπολογία από το network.txt και την αποθηκεύει σε ένα hash table για γρήγορη πρόσβαση, επίσης παρέχει μεθόδους για εξέταση ύπαρξης ενός link. Συγκεκριμένα χρησιμοποιούμε ένα hashable με hash key την διεύθυνση κάθε κόμβου που αναφέρεται στην αρχή κάθε γραμμής του αρχείου network.txt, και hash element μια linked list που περιέχει όλους τους γειτονικούς κόμβους.

```
class networkTopology {
   private Hashtable<Long,LinkedList> nodesLinks;

   networkTopology() {
      nodesLinks = new Hashtable<Long,LinkedList>();
```

```
    }

    boolean LinkExist(Long addr, long myaddr) {

        try{
            LinkedList l = (LinkedList)nodesLinks.get( new Long(myaddr));

            if(l!=null)
                for(int i=0;i<l.size();i++)
                    if(((Long)l.get(i)).longValue() == addr.longValue())
                        return true;
        } catch(Exception e) {
            e.printStackTrace();
            return false;
        }
        return false;
    }


    int readTopologyFromFile(String networkFile) {
        try {
            BufferedReader in = new BufferedReader(new FileReader(networkFile));
            String line;
            String nodesIEEEAddress[];

            while ((line = in.readLine()) != null) {
                nodesIEEEAddress = line.split(" ");
                LinkedList<Long> node = new LinkedList<Long>();
                System.out.print("adding node "+nodesIEEEAddress[0]+" with links to");
                for(int i=1;i<nodesIEEEAddress.length;i++) {
                    System.out.print(" "+nodesIEEEAddress[i]);
                    node.add(new Long(IEEEAddress.toLong(nodesIEEEAddress[i])));
                }
                System.out.println("");
                nodesLinks.put(new
Long(IEEEAddress.toLong(nodesIEEEAddress[0])),node);
            }
            in.close();
        } catch (Exception e) {
            System.err.println("Error reading network topology file: "+e.getMessage());
            return -1;
        }
        return 0;
    }

}
```

- packet – Η κλάση αυτή χρησιμοποιείται σαν wrapper για τα radiopackets. Περιλαμβάνει έναν byte array με τα δεδομένα του radiopacket καθώς και μεταβλητές για την αποθήκευση του αποστολέα, του προορισμού το μήκος των δεδομένων και τον τύπο του πακέτου(broadcast,data,ack).

```
class packet {
   public final static int BROADCAST = 1;
   public final static int DATA = 2;
   public final static int ACK = 3;

   byte data[];
   long origAddress;
   long destAddress;
   int len;
   int type;

   packet(byte[] data,long origAddress,long destAddress,int len,int type) {
      this.origAddress = origAddress;
      this.destAddress = destAddress;
      this.len = len;
      this.data = data;
      this.type = type;
   }

}
```

4. Ο εξομοίωσης ADAPT

Η πλατφόρμα ADAPT[14] είναι ένα εκτενές σύστημα για την εξομοίωση

κατανεμημένων αλγορίθμων. Υποστηρίζει την δημιουργία πολύπλοκων σεναρίων εξομοίωσης για την μοντελοποίηση διαφορετικών καταστάσεων, όπως κατάρρευση κόμβων, φυσικά εμπόδια κλπ. Επιπλέον έχει το χαρακτηριστικό ότι οι συσκευές που εξομοιώνονται μπορεί να μην είναι του ίδιου τύπου, δηλαδή μπορούμε να το χρησιμοποιήσουμε για την εξομοίωση ετερογενών δικτύων. Από τα πιο σημαντικά χαρακτηριστικά του ADAPT είναι ότι κάθε εφαρμογή που εξομοιώνεται τρέχει σαν κανονικό πρόγραμμα (process) στο λειτουργό σύστημα, και χρησιμοποιεί την ADAPT client βιβλιοθήκη για να επικοινωνεί, με το ADAPT Engine, δηλαδή την κεντρική process που διαχειρίζεται όλα να μηνύματα. Για την επικοινωνία χρησιμοποιείται ένα καλά ορισμένο πρωτόκολλο βασισμένο σε XML. Το ADAPT αποτελείται από τα εξής από τα εξής τρία υποσυστήματα:

ADAPT Engine

Πρόκειται για το κύριο υποσύστημα που διαχειρίζεται την εξομοίωση. Διατηρεί όλα τα χαρακτηριστικά, του περιβάλλοντος της εξομοίωσης και των αντικειμένων που συμμετέχουν, και τα χρησιμοποιεί για να διαχειρίζεται την επικοινωνία με τα processes και με το GUI.

Η Engine ακολουθεί το discrete-event μοντέλο εξομοίωσης. Σε αυτό το μοντέλο το σύστημα αναπαριστάται σαν μία ακολουθία από χρονικά διατεταγμένα events. Κάθε event είναι ένα γεγονός που συμβαίνει σε μια συγκεκριμένη χρονική στιγμή(θεωρείται ότι δεν έχει χρονική διάρκεια) και σηματοδοτεί μια αλλαγή στην κατάσταση(state) του συστήματος. Για κάθε γεγονός ή δράση που συμβαίνει μέσα στο σύστημα(όπως λήψη μηνυμάτων από τα processes), δημιουργείται ένα αντίστοιχο event με συγκεκριμένη χρονοσφραγίδα (timestamp). Όλα τα event τοποθετούνται σε μία ουρά και εκτελούνται από το κεντρικό module του Engine που στην ουσία είναι ένας διαχειριστής ουράς(queue handler). Τα events έχουν πλήρη έλεγχο του περιβάλλοντος της εξομοίωσης και μπορούν να περιγράψουν περίπλοκες καταστάσεις όπως αλλαγή στην τοπολογία.

Κάθε επικοινωνία στο ADAPT, είτε πρόκειται για μηνύματα από τις εφαρμογές της εξομοίωσης είτε για εντολές ελέγχου από το GUI, αναπαριστάται από ένα action. Για την διαχείριση αυτών είναι υπεύθυνο το action management framework που συμπληρώνει την λειτουργία του διαχειριστή events. Κατά την εκτέλεση των actions από το Engine μπορεί να έχουμε την δημιουργία events, για παράδειγμα ένα action από το GUI μπορεί να οδηγήσει στην δημιουργία ενός event για την παύση της εξομοίωσης ή για την αλλαγή κάποιου χαρακτηριστικού ενός κόμβου.

Τέλος η Engine κρατάει διάφορες μετρικές και στατιστικά για την εξομοίωση, όπως ο συνολικός αριθμός των μηνυμάτων που μεταδόθηκαν. Μέσω της client library δίνεται η δυνατότητα στον προγραμματιστή να προσθέσει και επιπλέον μετρικές που θα καταγράφει η Engine κατά την διάρκεια της εξομοίωσης.

Το πιο σημαντικό χαρακτηριστικό της Engine είναι το μοντέλο που χρησιμοποιεί για την εξομοίωση του περιβάλλοντος. Χρησιμοποιώντας ένα αφαιρετικό μοντέλο για την αναπαράσταση του περιβάλλοντος αποφεύγει ένα μεγάλο όγκο από λεπτομέρειες που προσθέτουν δυσανάλογο βάρος στην

πολυπλοκότητα της εξομοίωσης. Το μοντέλο που χρησιμοποιείται χωρίζεται σε τρία επίπεδα(layers):

- Topology layer – Αποτελείται από topology nodes που αναπαριστούν φυσικές τοποθεσίες, οι οποίες μπορεί να συνδέονται(να θεωρούνται γειτονικές).
- Computing layer – Κάθε computing node βρίσκεται πάνω από ένα topology node, και μπορεί να συνδέεται με άλλους γειτονικούς computing nodes. Επίσης μπορεί να μετακινείται σε διαφορετικά topology nodes ανάλογα με το περιβάλλον εξομοίωσης. Τα computing nodes αναπαριστούν τις φυσικές συσκευές που τρέχουν τους αλγόριθμους/εφαρμογές που εξομοιώνουμε.
- Process layer – Σε αυτό το layer περιέχονται όλες οι εφαρμογές που πρόκειται να τρέξουν στον εξομοιωτή. Κάθε εφαρμογή τρέχει πάνω σε ένα computing node, και ανάλογα με τις παραμέτρους της εξομοίωσης έχει την δυνατότητα να αλλάξει τα χαρακτηριστικά των χαμηλότερων επιπέδων(για παράδειγμα η μετακίνηση του computing node που βρίσκεται σε διαφορετικό topology node).

Το περιβάλλον της εξομοίωσης μπορεί να είναι στατικό ή να αλλάζει με την πάροδο του χρόνου σύμφωνα με ένα προκαθορισμένο σενάριο. Η περιγραφή των σεναρίων γίνεται μέσω μιας καλά ορισμένης XML. Τα σενάρια είναι μια σειρά από actions που πρόκειται να εκτελεστούν σε συγκεκριμένες χρονικές στιγμές κατά την διάρκεια της εξομοίωσης. Η Engine παρέχει την δυνατότητα να στείλουμε actions και από άλλες πηγές, όπως για παράδειγμα από το GUI, αυτά τα actions ακολουθούν το ίδιο πρωτόκολλο με αυτά που προέρχονται από το σενάριο. Τα actions που υποστηρίζονται από την Engine και μπορούμε να χρησιμοποιήσουμε στα σενάρια είναι:

- Δημιουργία ενός interrupt σε ένα κόμβο ή σε μια process σε μια συγκεκριμένη χρονική στιγμή. Αυτά τα interrupts εξυπηρετούνται από συγκεκριμένες συναρτήσεις που ο χρήστης κάνει register στην ADAPT client library.
- Κατάρρευση ενός κόμβου σε μια συγκεκριμένη χρονική στιγμή. Η κατάρρευση μπορεί να είναι μόνιμη ή προσωρινή.
- Δημιουργία φυσικών/περιβαλλοντολογικών μεγεθών/καταστάσεων που θα καταγράφονται από τους αισθητήρες των κόμβων μέσω στης ADAPT library.
- Αλλαγή της θέσης του κόμβου σε μια συγκεκριμένη χρονική στιγμή

Όλα τα παραπάνω actions έχουν παραμέτρους που μπορούμε να ορίσουμε να παίρνουν τυχαίες τιμές, όπως για παράδειγμα η χρονική στιγμή , η διάρκεια, και η πιθανότητα να συμβούν. Ακόμα έχουμε την δυνατότητα να προσθέσουμε εύκολα και επιπλέον actions.

Client library

Η client library είναι υπεύθυνη για την διασύνδεση των προγραμμάτων με την ADAPT Engine. Η βιβλιοθήκη αυτή παρέχει όλες τις συναρτήσεις που

χρειάζονται για την υλοποίηση κατανεμημένων αλγορίθμων, όπως συναρτήσεις για την αποστολή και λήψη μηνυμάτων(blocking και non-blocking). Επίσης δίνει την δυνατότητα για αλληλεπίδραση με το περιβάλλον μέσω συναρτήσεων που παρέχει για αλλαγή της θέσης των κόμβων, για αλλαγή και λήψη όλων των παραμέτρων του περιβάλλοντος κ.α. Επιπλέον παρέχει συναρτήσεις για τον έλεγχο της εξομοίωσης και του γραφικού περιβάλλοντος(για παράδειγμα αλλαγή του χρώματος ενός κόμβου).

Ο σχεδιασμός της βιβλιοθήκης είναι σχετικά απλός και το μεγαλύτερο τμήμα της επεξεργασίας το αναλαμβάνει η Engine, ώστε να μην επιβαρύνει το πρόγραμμα που τελικά θα χρησιμοποιήσει την client library. Το ADAPT περιλαμβάνει μια client library σε C++ για την ανάπτυξη εφαρμογών που τρέχουν είτε σε κανονικούς υπολογιστές είτε σε embedded συσκευές με linux περιβάλλον. Όλη η επικοινωνία μεταξύ της βιβλιοθήκης και της Engine γίνεται σε απλό XML format , οπότε η υλοποίησή της σε κάποια άλλη γλώσσα προγραμματισμού είναι σχετικά απλή υπόθεση. Κάθε πρόγραμμα που συμμετέχει στην εξομοίωση δεν χρειάζεται να χρησιμοποιεί την ίδια client library. Αυτό μας δίνει την δυνατότητα να εξομοιώνουμε ετερογενή δίκτυα όπου κάθε εφαρμογή που τρέχει σε διαφορετική πλατφόρμα θα χρησιμοποιεί και την αντίστοιχη βιβλιοθήκη.

GUI

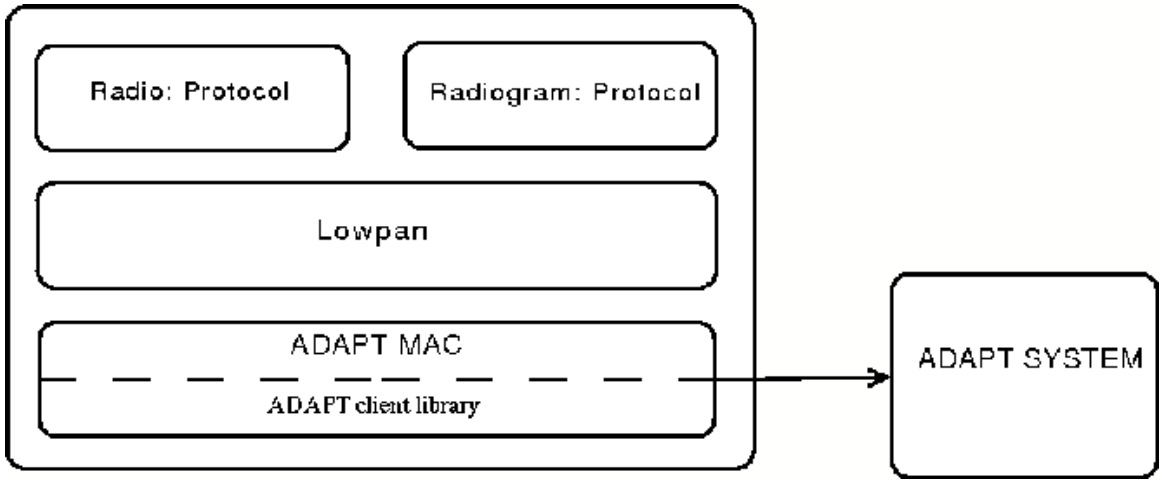
Το ADAPT παρέχει και μια γραφική διεπαφή για τον χρήστη ώστε να παρακολουθεί την εξομοίωση σε πραγματικό χρόνο. Ο χρήστης έχει την δυνατότητα να ελέγχει την εξομοίωση και μπορεί να την σταματάει και να την εκκινεί σε οποιαδήποτε φάση. Επιπλέον ο χρήστης μπορεί να αλλάζει τις παραμέτρους του περιβάλλοντος και να στέλνει νέα σενάρια εξομοίωσης στην Engine. Το GUI περιλαμβάνει ένα editor για την τοπολογία και έναν για τα σενάρια που απλοποιούν και βοηθούν τον καθορισμό των παραμέτρων της εξομοίωσης από τον χρήστη.

Το GUI είναι ξεχωριστό εκτελέσιμο από την Engine και επικοινωνεί με αυτήν μέσω sockets. Το GUI μπορεί να συνδεθεί και να αποσυνδεθεί στο Engine οποιαδήποτε στιγμή κατά την διάρκεια της εξομοίωσης, οπότε κατά την εκτέλεση χρονοβόρων εξομοιώσεων ο χρήστης μπορεί να συνδεθείτε και να παρακολουθεί την εξέλιξή της. Επιπλέον υπάρχει η δυνατότητα να συνδέονται περισσότερα από ένα GUI ταυτόχρονα, και κάθε ένα να παρακολουθεί διαφορετικό τμήμα της εξομοίωσης.

4.1 Διασύνδεση με το ADAPT

Όπως περιγράψαμε στο κεφάλαιο 3.3 για την προσθήκη στον SUN SPOT Emulator βασική υποστήριξη για network topology υλοποιήσαμε τον rpd server για την διαχείριση και αποστολή των πακέτων στους σωστούς προορισμούς και τροποποίησαμε το MAC layer του Emulator ώστε να στέλνει τα πακέτα στον rpd server. Αυτή η σχεδίαση παρέχει την δυνατότητα τα emulated SPOTs να συμπεριφέρονται σαν να βρίσκονται σε διαφορετική φυσική τοποθεσία αλλά δεν έχει υποστήριξη για πολύπλοκα σενάρια(πχ κινητικότητα κόμβων) και για λεπτομερή παρακολούθηση της εξομοίωσης(πχ logging). Για να επιτύχουμε την παραπάνω λειτουργικότητα διασυνδέσαμε τον SUN SPOT Emulator με τον εξομοιωτή ADAPT [15], ώστε να το ADAPT να διαχειρίζεται όλη την επικοινωνία. Έτσι μπορούμε να εκμεταλλευτούμε όλη την λειτουργικότητα του

ADAPT για την εκτέλεση κατανεμημένων αλγορίθμων στα SPOTs. Για την υλοποίηση της διασύνδεσης χρειάστηκε να υλοποιήσουμε μια client library σε Java 2 Micro edition για την επικοινωνία με το ADAPT Engine, και την τροποποίηση του MAC layer του εξομοιωτή ώστε να χρησιμοποιεί την client library για να προωθεί και να λαμβάνει τα radio packets από τον ADAPT. Στην παρακάτω εικόνα βλέπουμε την αρχιτεκτονική του radiostack του εξομοιωτή με το ADAPT.



4.2 Υλοποίηση client library

Η βιβλιοθήκη αυτή βρίσκεται στο adapt_client_j2me.jar και έχει όλες τις μεθόδους που χρειάζονται για την επικοινωνία με το ADAPT, και χρησιμοποιεί τον kxml pull parser για την διαχείριση των xml-based μηνυμάτων. Η κύρια κλάση που χρησιμοποιεί όποιο πρόγραμμα θέλει να συνδεθεί στο ADAPT είναι η DapSystem.java. Στην συνέχεια παραθέτουμε μια περιγραφή των κλάσεων της client library και αναλύουμε τις πιο σημαντικές μεθόδους τους.

DapSystem.java:

Στον δημιουργό αυτής της κλάσης περνάμε σαν όρισμα ένα αριθμό από παραμέτρους σχετικές με τον client, όπως το id του και σε πιο computing node βρίσκεται. Επιπλέον αρχικοποιούνται οι την κλάσεις ClientTranceiver, ProcessState, και KXMLParser. Τέλος στέλνουμε ένα μήνυμα στο ADAPT για να κάνουμε register τον συγκεκριμένο client. Μετά την αρχικοποίηση η εφαρμογή μπορεί να στείλει και να παραλάβει μηνύματα χρησιμοποιώντας τις παρακάτω μεθόδους:

- send(String content, String tag) – Αυτή η μέθοδος στέλνει ένα μήνυμα(content) σε όλες τις εφαρμογές που τρέχουν στον ίδιο computing node με το δικό μας πρόγραμμα. Το tag είναι μια προαιρετική επιλογή, και μπορούμε να την χρησιμοποιήσουμε για να φιλτράρουμε μηνύματα ώστε στην αντίστοιχη receive να δεχόμαστε μόνο τα μηνύματα με ένα συγκεκριμένο tag. Η send κωδικοποιεί το μήνυμα σε xml format και στην συνέχεια το στέλνει μέσω της μεθόδου send() του ClientTranceiver.
- send(Integer nodeID, Integer rcptID, String msg, String tag) - Αυτή η

μέθοδος στέλνει ένα μήνυμα(msg) σε μια συγκεκριμένη εφαρμογή(rcptID) σε έναν συγκεκριμένο computing node(nodeID). Η send κωδικοποιεί το μήνυμα σε xml format και στην συνέχεια το στέλνει μέσω της μεθόδου send() του ClientTranceiver.

- broadcast(String msg, String tag) – Αυτή η μέθοδος στέλνει ένα μήνυμα σε όλους τους γείτονες.
- receive(boolean block) – Η μέθοδος αυτή επιστρέφει ένα μήνυμα, αυτό περιέχει τις διευθύνσεις του αποστολέα, τα δεδομένα του μηνύματος το tag και εσωτερικές πληροφορίες που χρησιμοποιούνται από την βιβλιοθήκη. Η δομή του μηνύματος είναι:

```
public class Message
{
	public Integer fromID;
	public Integer toID;
	public String content;
	public String tag;
	public int ticks;
}
```

Τα μηνύματα αυτά παραλαμβάνονται μέσω της μεθόδου receive() του ClientTranceiver σε xml format και στην συνέχεια αποκωδικοποιούνται από την KXMLParser στην παραπάνω δομή.

ClientTranceiver.java:

Η κλάση αυτή είναι υπεύθυνη για την επικοινωνία μέσω stream sockets με το ADAPT. Κατά την αρχικοποίηση της κλάσης αυτής, συνδέεται στο ADAPT μέσω socket connections, η προκαθορισμένη λειτουργία είναι να συνδέεται στην port 8788 του localhost αλλά μπορούμε να αλλάζουμε αυτές τις ρυθμίσεις αν τα Emulated SPOTs τρέχουν σε διαφορετικό workstation από το ADAPT.

```
	public ClientTranceiver(String host, int port)
	{
		connect(host, port);
	}

	public void connect(String host, int port)
	{
		sock = null;
		out = null;
		in = null;
		try {
			sock = (StreamConnection)Connector.open("socket://" + host + ":" +
port);

			out = new PrintStream(sock.openOutputStream());
			in = new InputStreamReader(sock.openInputStream());
		} catch (IOException e) {
			System.err.println("Cannot get I/O for "
				+ "the connection to: " + host);
```

```
                e.printStackTrace();
                System.exit(1);
            }
        }
```

Η κλάση αυτή περιλαμβάνει και δυο επιπλέον μεθόδους για την αποστολή και την λήψη δεδομένων.

```
    public void send(String text)
    {
        out.println(text);
        out.flush();
    }
```

```
    public String receive() throws IOException
    {
        return readLine();
    }
```

```
    private String readLine()
    {
       String result = "";
       try {
          while (true) {
             int ch = in.read();
             if (ch == -1) {
                if (result.length() == 0) {
                   result = null;
                }
                break;
             } else if (ch == '\n' || ch == '\r') {
                break;
             } else {
                result += (char)ch;
             }
          }
       } catch (IOException ex) {
          System.err.println("Error reading input from socket: " + ex);
       }
    return result;
}
```

ProcessState.java:

Η κλάση αυτή είναι υπεύθυνη για να διατηρεί πληροφορίες σχετικά με την κατάσταση του client, όπως το node id στο όποιο βρίσκεται και το τελευταίο μήνυμα που έλαβε. Επίσης παρέχει μεθόδους για την την αλλαγή

αυτών των τιμών αλλά και για την ανάγνωσή τους από άλλες κλάσεις.

```
public class ProcessState
{
        public ProcessState(Integer n_pid, Integer n_hnID, int n_clock)
    {
            pid = n_pid;
            hostNodeID = n_hnID;
            curClock = n_clock;
            lastMsg = null;
    }

    Message popMessage()
    {
            return lastMsg;
    }

    public void putMessage(Message msg)
    {
            lastMsg = msg;
    }

        public void setPID(Integer n_pid) { pid = n_pid; }
        public Integer getPID() { return pid; }
        public void setHostNodeID(Integer nHNID) { hostNodeID = nHNID; }
        public Integer getHostNodeID() { return hostNodeID; }

        public void setClock(int ticks) { curClock = ticks; }
        public int getClock() { return curClock; }

        private Integer pid;
        private Integer hostNodeID;
        private int curClock;
    private Message lastMsg;
};
```

KXMLParser.java:

Αυτή η κλάση παρέχει την δυνατότητα για να αποκωδικοποιούνται τα xml μηνύματα που λαμβάνονται από την ADAPT Engine. Εσωτερικά χρησιμοποιεί τον Kxml parser για την αποκωδικοποίηση. Η μέθοδος που καλείται από το DapSystem για την αποκωδικοποίηση ενός μηνύματος είναι η *parse(String text),* που αρχικοποιεί και επιστρέφει μια κλάση action.

MsgRecvAction.java && SimCheckAction.java:

Πρόκειται για αντικείμενα που δημιουργούνται μετά από την αποκωδικοποίηση ενός μηνύματος από το ADAPT. Ανάλογα με τον τύπο του μηνύματος παράγεται και διαφορετικό action. Αν πρόκειται για μήνυμα

έλεγχου από το ADAPT τότε παράγεται ένα αντικείμενο της SimCheckAction ενώ αν πρόκειται για απλό μήνυμα επικοινωνίας τότε παράγεται ένα MsgRecvAction. Και τα δυο περιέχουν τις πληροφορίες που παρελήφθησαν και μια μέθοδο execute(), που αναλαμβάνει να διαχειριστεί την πληροφορία ανάλογα. Στην περίπτωση του MsgRecvAction η execute() παραδίδει το μήνυμα στην ProcessState (και μετέπειτα στην εφαρμογή), ενώ στην περίπτωση του SimCheckAction απλώς αλλάζει τις εσωτερικές παραμέτρους της κατάστασης του client.

4.3 Τροποποίηση του του MAC layer

Η κλάση αυτή βρίσκεται στο αρχείο adaptMAC.java και αποτελεί το τροποποιημένο MAC layer του εξομοιωτή των SPOTs. Τα πακέτα από τα υψηλού επιπέδου πρωτόκολλα (πχ radiograms), κερματίζονται σε radiopackets από το lowpan και παραδίδονται από το adaptMAC στο ADAPT χρησιμοποιώντας κλήσεις της client library. Κατά την αρχικοποίηση της κλάσης, συνδέεται στον ADAPT αστικοποιώντας το αντικείμενο DapSystem() της client library και εκκινεί ένα thread (το RxDapThread) που αναλαμβάνει την λήψη πακέτων από το ADAPT. Κατά την αρχικοποίηση χρησιμοποιούμε για id τα 16 least significant bits της εικονικής διεύθυνσης του SPOT. Παρακάτω βλέπουμε την μέθοδο που χρησιμοποιούμε για την αρχικοποίηση του MAC.

```
  public void mlmeStart(short panId, int channel) throws MAC_InvalidParameterException {

      try {
         if (adapt == null) {
            String ieeeAddress = System.getProperty("IEEE_ADDRESS");
            extendedAddress = IEEEAddress.toLong(ieeeAddress);

            int nodeID = (int)(extendedAddress & 0xFFFFL);
            System.out.print("Init connection to adapt system: nodeID== "+nodeID+"
extendedAddress== "+extendedAddress+" ieeeAddress== "+ieeeAddress);
            adapt = new DapSystem(new String[]{"--id",""+nodeID,"--sid",""+nodeID});
            RxThread = new RxDapThread();
            VM.setAsDaemonThread(RxThread);
            RxThread.start();
 }
      } catch (Exception e) {
         throw new SpotFatalException("Encountered Error Initialazing connection to adapt
system: " + e.getMessage());
      }
}
```

Το RxDapThread όπως είπαμε παραλαμβάνει πακέτα από το ADAPT και τα αποθηκεύει σε μια προσωρινή ούρα ώστε να τα παραλάβουν τα υψηλότερα επίπεδα. Τα πακέτα αυτά μπορεί να είναι unicast, broadcast ή ACKs. Για τον ευκολότερο διαχωρισμό του τύπου κάθε πακέτου χρησιμοποιούμε το πεδίο tag, στις μεθόδους send()/receive(), μαρκάροντας κατάλληλα τα πακέτα. Παρακάτω παραθέτουμε την μέθοδο run() του thread.

```
public void run() {
        while (true) {
           RadioPacket incoming = RadioPacket.getAckPacket();
           try {
                 DapMsg = adapt.receive(true);

                 if(DapMsg.tag.equals("data"))
                 {
                    System.arraycopy(DapMsg.content.getBytes(), 0, incoming.buffer,
0,DapMsg.content.getBytes().length);
                    incoming.decodeFrameControl();
                    if (incoming.isAck()) {
                       ackQueue.put(incoming);
                    } else {
                       RadioPacket ackPacket = RadioPacket.getAckPacket();
                       ackPacket.setDSN(incoming.getDataSequenceNumber());
                       adapt.send(getNodeID(incoming.getSourceAddress()),
getNodeID(incoming.getSourceAddress()), new String(ackPacket.buffer), "data");
                       rxQueue.put(incoming);
                    }
                 }
                 else if(DapMsg.tag.equals("bcast"))
                 {
                   RadioPacket rp = RadioPacket.getBroadcastPacket();
                    rp.decodeFrameControl();
                       rxQueue.put(rp);
                    }
                 }


           } catch (Exception e) {
              if (closed) {
                 break;
              } else {
                 e.printStackTrace();
              }
           }
        }
     }
```

Η κλάση adaptMAC παρέχει δύο μεθόδους ώστε τα υψηλότερα layers να παραλαμβάνουν και να στέλνουν radiopackets. Η μέθοδος mcpsDataRequest() καλείται για την αποστολή ενός πακέτου από το adaptMAC. Πρώτα θέτει ένα νέο ακολουθιακό αριθμό(Sequence Number) στο πακέτο, και έπειτα εξετάζει αν πρόκειται για broadcast. Στην περίπτωση που η διεύθυνση αποστολής είναι 0xFFFF (broadcast) τότε αποστέλλεται μέσω της μεθόδου broadcast() του DapSystem χρησιμοποιώντας σαν tag το ”bcast”. Σε διαφορετική περίπτωση μετά την αποστολή καλούμε send() με tag “data” και περιμένουμε για πακέτο επιβεβαίωσης. Όλα τα δεδομένα μετατρέπονται σε string πριν την αποστολή ενώ για διευθύνσεις(node id) προορισμού/αποστολέα χρησιμοποιούμε τα τελευταία 16 bit των εικονικών διευθύνσεων των SPOTs. Αυτό πρέπει να ληφθεί υπόψη κατά τον ορισμό του σεναρίου της εξομοίωσης καθώς όλες οι

διευθύνσεις των SPOT πρέπει να διαφέρουν στα τελευταία 2 bytes.

```
  public synchronized int mcpsDataRequest(RadioPacket rp) {
    rp.setDSN(getDSN());
    int result = NO_ACK;
    try {

      if (rp.getDestinationAddress() == 0xFFFF) {
        packetRxThread.broadcast(rp.buffer);
        result = SUCCESS;
      } else {

        if (packetRxThread != null) {
          packetRxThread.sendto(rp.getDestinationAddress(),rp.buffer,rp.buffer.length);

            if (packetRxThread.waitForAck(rp.getDataSequenceNumber())) {
              result = SUCCESS;
            }
        }
      }
    } catch (Exception e) {
      e.printStackTrace();
      throw new SpotFatalException("Got IOException while sending over SocketMac " +
e.getMessage());
    }
    return result;
  }
```

Η μέθοδος mcpsDataIndication() καλείται από τον radioPacketDispatcher για την παραλαβή ενός πακέτου από το adaptMAC. Η μέθοδος αυτή είναι blocking, εξαιτίας της υλοποίησης της μεθόδου get() της Queue στην βιβλιοθήκη του SUN SPOT, και καλείται για όσο υπάρχουν πακέτα στην ουρά.

```
  public void mcpsDataIndication(RadioPacket rp) {
    rp.copyFrom((RadioPacket) rxQueue.get());
  }
```

5. Εισαγωγή στο πρωτόκολλο DSR

Σε ένα WSN δεν υπάρχει σταθερή υποδομή και κατά αυτή την έννοια μοιάζει με ένα ad-hoc δίκτυο, σε αυτό περιβάλλον ένας κόμβος μπορεί να θέλει να μεταδώσει πακέτα πληροφορίας σε έναν άλλον που λόγο της περιορισμένης εμβέλειας εκπομπής δεν μπορεί να γίνει άμεσα. Σε μια τέτοια περίπτωση πρέπει να χρησιμοποιήσει ενδιάμεσους κόμβους που θα προωθήσουν τα πακέτα μέχρι τον προορισμό. Ένα τέτοιο παράδειγμα φαίνεται στην παρακάτω εικόνα(figure 1) όπου ο κόμβος Α θέλει να μεταδώσει στον κόμβο C που είναι εκτός εμβέλειας εκπομπής. Για να είναι εφικτή η επικοινωνία ο κόμβος Α μεταδίδει τα πακέτα πρώτα στον Β ο οποίος τελικά τα στέλνει στον C. Βέβαια στα πραγματικά ad-hoc δίκτυα η κατάσταση είναι περισσότερο περίπλοκη λόγο των χαρακτηριστικών των ασυρμάτων καναλιών και λόγο της κίνησης των κόμβων. Αλλά το βασικό πρόβλημα που καλείται να λύσει ένα πρωτόκολλο δρομολόγησης είναι η εύρεση μιας ακολουθίας από ενδιάμεσους κόμβους για την μετάδοση πληροφορίας προς ένα προορισμό.

Τα πιο σημαντικά πρωτόκολλα δρομολόγησης των ενσύρματων δικτύων βασίζονται κυρίως στους αλγορίθμους distance vector η link state, όμως και οι δυο εξαρτώνται από περιοδικές αποστολές στο δίκτυο πληροφορίας δρομολόγησης. Στους distance vector αλγορίθμους κάθε κόμβος στέλνει σε όλους τους γείτονες του τις αποστάσεις του από όλους τους κόμβους του δικτύου και έτσι με βάση αυτήν την πληροφορία κάθε κόμβος μπορεί να υπολογίσει το κοντινότερο μονοπάτι προς κάθε προορισμό στο δίκτυο. Στους link state αλγορίθμους κάθε κόμβος στέλνει περιοδικά σε όλο το δίκτυο την κατάσταση των link με τους γείτονες του όποτε κάθε κόμβος έχει μια συνολική εικόνα του δικτύου και μπορεί να υπολογίσει με βάση αυτήν το κοντινότερο μονοπάτι για κάθε προορισμό.

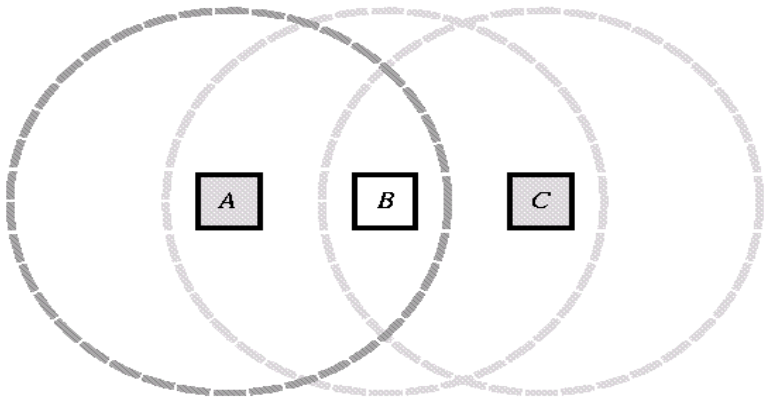


Figure 1 A simple ad hoc network of three wireless mobile hosts

Αν και τα πρωτόκολλα που βασίζονται στους παραπάνω αλγορίθμους δουλεύουν καλά σε ενσύρματα δίκτυα η απόδοση τους στα ασύρματα ad-hoc

δίκτυα είναι πολύ περιορισμένη. Αυτό συμβαίνει πρώτον γιατί οι περιοδικές αποστολές μηνυμάτων σε όλο το δίκτυο δημιουργούν συμφόρηση και καταναλώνουν πολύτιμο bandwidth και δεύτερον γιατί στο δυναμικό περιβάλλον των ad-hoc δικτύων η τοπολογία μπορεί να αλλάζει αρκετά γρήγορα με αποτέλεσμα οι αλγόριθμοι να μην προλαβαίνουν να συγκλίνουν σε σωστές διαδρομές.

5.2.1 Υποθέσεις του πρωτοκόλλου

Κατά τον σχεδιασμό του DSR [16] πρωτοκόλλου έχουν γίνει κάποιες βασικές υποθέσεις που πρέπει να ισχύουν για την σωστή λειτουργία του. Αυτές είναι:

- Υποθέτουμε ότι κάθε κόμβος που θέλει να μεταδώσει πακέτα στο δίκτυο, θα συμμετέχει πλήρως σε όλα τα πρωτόκολλα του δικτύου και κυρίως θα πρέπει να προωθεί πακέτα άλλων κόμβων στο δίκτυο. Αυτό είναι μια πολύ βασική υπόθεση που ισχύει για όλα τα multi-hop δίκτυα.
- Υποθέτουμε ότι το δίκτυο θα έχει σχετικά μικρή διάμετρο. Διάμετρο ενός δικτύου είναι ο αριθμός κόμβων που χρειάζεται να διατρέξει ένα πακέτο για να φτάσει από την μια άκρη του δικτύου στην άλλη. Αυτή η υπόθεση γίνεται γιατί κατά την μετάδοση πακέτων πληροφορίας επισυνάπτουμε και τις διευθύνσεις όλων των κόμβων που θα μεταβεί το πακέτο μέχρι να φτάσει στον προορισμό, οπότε σε δίκτυα με μεγάλη διάμετρο το overhead του πρωτοκόλλου γίνεται απαγορευτικό.
- Δεν κάνουμε καμιά υπόθεση για το πότε και πως θα κινούνται οι κόμβοι στο δίκτυο άλλα υποθέτουμε ότι η ταχύτητα των κόμβων στο δίκτυο είναι αρκετά μικρή σε σύγκριση με τον χρόνο που χρειάζεται για την μετάδοση των πακέτων και της ακτίνας μετάδοσης των ασυρμάτων κόμβων.
- Τέλος για την πλήρη αξιοποίηση των χαρακτηριστικών του θα πρέπει να μπορούνε οι κόμβοι να θέσουν τον ασύρματο δεκτή τους σε promiscuous mode. Αυτό πρακτικά σημαίνει ότι θεωρούμε πως μπορούν να λάβουν όλα τα μεταδιδόμενα πακέτα και όχι μόνο αυτά που απευθύνονται σε αυτούς Αν και αυτό το χαρακτηριστικό δεν είναι απαραίτητο για την λειτουργιά του πρωτοκόλλου, μπορούμε να το χρησιμοποίησουμε για να βελτιώσουμε την απόδοσή του. Πρέπει να προσθέσουμε εδώ ότι όταν ένα ασύρματο interface είναι σε promiscuous mode τότε θα έχουμε περισσότερα πακέτα για επεξεργασία και άρα περισσότερο CPU overhead στον κόμβο. Αυτό όμως δεν είναι σημαντικός παράγοντας γιατί στα περισσότερα ασύρματα δίκτυα ο βασικότερος περιοριστικός παράγοντας είναι η μικρή διαθεσιμότητα σε bandwith.

5.3 ΒΑΣΙΚΕΣ ΔΙΑΔΙΚΑΣΙΕΣ ΤΟΥ DSR

Το DSR πρωτόκολλο αποτελείται από δυο βασικές διαδικασίες που δουλεύουν μαζί και επιτρέπουν στην ανακάλυψη και την συντήρηση των source routes (διαδρομών πηγής) στο ad-hoc δίκτυο.

- Διαδικασία ανακάλυψης διαδρομής: Η διαδικασία αυτή επιτρέπει σε ένα κόμβο Α που θέλει να στείλει δεδομένα σε ένα προορισμό Β να αποκτήσει μια διαδρομή πηγής, αν δεν την έχει ήδη αποθηκευμένη.

- Διαδικασία συντήρησης διαδρομής: Κατά την διάρκεια που ένας κόμβος Α μεταδίδει δεδομένα σε ένα κόμβο Β, με αυτή την διαδικασία ο Α μπορεί να ανιχνεύσει τυχόν αλλαγές στην τοπολογία που έχουν ως αποτέλεσμα η διαδρομή που χρησιμοποιεί να μην είναι έγκυρη πλέον Αυτό συμβαίνει συνήθως όταν ένα link στο μονοπάτι της διαδρομής καταρρεύσει.

Αυτές οι δυο διαδικασίες λειτουργούν κατά απαίτηση, δηλαδή μόνο όταν τις χρειάζεται ένας κόμβος, έτσι δεν έχουμε καθόλου περιοδικές μεταδόσεις πακέτων από το πρωτόκολλο. Οι διαδρομές που ανακαλύπτει ένας κόμβος αποθηκεύονται σε μια μνήμη προσωρινής αποθήκευσης διαδρομών και σχετίζονται με ένα μετρητή χρόνου(timer) που διαγράφει τις καταχωρίσεις διαδρομών μετά από ένα χρονικό διάστημα.

5.3.1 Διαδικασία ανακάλυψης διαδρομής

Όταν ένας κόμβος θέλει να στείλει ένα πακέτο δεδομένων σε ένα προορισμό στο δίκτυο ψάχνει για την κατάλληλη διαδρομή(που αποτελείται από όλους τους ενδιάμεσους κόμβους που πρέπει να ακολουθήσει το πακέτο) στην μνήμη προσωρινής αποθήκευσης διαδρομών (route cache), αν δεν βρεθεί κατάλληλη διαδρομή τότε ξεκινά μια διαδικασία ανακάλυψης διαδρομής.

Κατά την διαδικασία ανακάλυψης διαδρομής ο κόμβος-αφετηρία δημιουργεί ένα πακέτο ROUTE REQUEST(RREQ) και το στέλνει σε όλους τους γειτονικούς τους κόμβους, δηλαδή σε όλους τους κόμβους που βρίσκονται μέσα στην εμβέλεια του. Το πακέτο RREQ περιέχει την διεύθυνση του κόμβου-αφετηρία την διεύθυνση του κόμβου-προορισμού και ένα μοναδικό request_id που τίθεται από τον κόμβο-αφετηρία. Το request_id σε συνδυασμό με την διεύθυνση της αφετηρίας προσδιορίζουν μοναδικά κάθε ROUTE REQUEST. Ακόμα περιέχεται ένα πεδίο που αποθηκεύονται όλοι οι ενδιάμεσοι κόμβοι από τους οποίους έχει περάσει το συγκεκριμένο ROUTE REQUEST πακέτο.

Όταν ένας κόμβος του δικτύου λάβει ένα RREQ πακέτο αρχικά ελέγχει για να διαπιστώσει μήπως το έχει ξαναλάβει σε μια προηγούμενη χρονική στιγμή. Αυτό γίνεται ελέγχοντας το request_id και την διεύθυνση του αποστολέα του RREQ που έλαβε με αυτά που έχει ήδη λάβει προηγουμένως. Αν

το έχει λάβει παλαιότερα τότε το απορρίπτει και δεν το προωθεί. Μέτα ελέγχει μήπως η δική του διεύθυνση αναφέρεται μέσα στο πεδίο που αποθηκεύονται οι ενδιάμεσοι κόμβοι. Αν αναφέρεται τότε απορρίπτει το RREQ. Έτσι αποφεύγονται οι δημιουργίες κύκλων. Τέλος αν ο κόμβος που έλαβε το RREQ δεν έχει απορρίψει το πακέτο ελέγχει μήπως η διεύθυνση που κόμβου-προορισμού είναι η ίδια με την δικιά του. Αν όχι τότε το προωθεί στους γείτονές του. Αν όμως τελικά είναι αυτός ο κόμβος-προορισμός τότε στέλνει ένα ROUTE REPLY(RREP) πακέτο πίσω στον κόμβο-αφετηρία. Το RREP θα περιέχει την πλήρη διαδρομή από ενδιάμεσους κόμβους όπως αυτή έχει καταγραφεί στο σχετικό πεδίο του RREQ.

Όταν ένας κόμβος αποφασίσει να στείλει ένα ROUTE REPLY πρώτα ελέγχει αν έχει αποθηκευμένη μια διαδρομή προς τον κόμβο-αφετηρία στην route cache. Αν υπάρχει μια τέτοια διαδρομή τότε μπορεί να την χρησιμοποιήσει. Αν δεν έχει αποθηκευμένη μια τέτοια διαδρομή τότε μπορεί να χρησιμοποιήσει την αντίστροφη διαδρομή από αυτή που υπάρχει στο πεδίο ενδιάμεσοι κόμβοι του ROUTE REQUEST. Αυτό βέβαια υποθέτει ότι οι συνδέσεις μεταξύ των κόμβων είναι συμμετρικές, αν κάτι τέτοιο δεν ισχύει τότε αυτή η διαδρομή δεν μπορεί να χρησιμοποιηθεί και ο κόμβος-προορισμός πρέπει να ξεκινήσει μια διαδικασία ανακάλυψης διαδρομής για τον κόμβο-αφετηρία. Αυτό όμως μπορεί να οδηγήσει ένα ατέρμονα κύκλο αποστολής ROUTE REQUEST πακέτων, για να αποφύγουμε αυτό το πρόβλημα ο κόμβος-προορισμός στο πακέτο ROUTE REQUEST που στέλνει ενσωματώνει(piggyback) και το πακέτο ROUTE REPLY. Εδώ μπορούνε να προσθέσουμε ότι είναι δυνατόν να χρησιμοποιήσουμε αυτήν την τεχνική για να ενσωματώσουμε και άλλα μικρά πακέτα δεδομένων στο ROUTE REQUEST όπως τα TCP SYN πακέτα.

5.3.2 Διαδικασία συντήρησης διαδρομής

Η διαδικασία συντήρησης διαδρομής ενημερώνει έναν κόμβο του δικτύου που χρησιμοποιεί μια διαδρομή για να στείλει πακέτα δεδομένων σε ένα κόμβο-προορισμό, ότι κάποιο link αυτής της διαδρομής έχει καταρρεύσει.

Η παραπάνω διαδικασία στηρίζεται σε ένα μηχανισμό επιβεβαίωσης από τον επόμενο κόμβο που λαμβάνει δεδομένα, δηλαδή κάθε φορά που ένας κόμβος στέλνει δεδομένα σε ένα γείτονα κόμβο τότε ο δεύτερος θα πρέπει να στείλει μια επιβεβαίωση στο αποστολέα ότι τα δεδομένα ελήφθησαν σωστά. Αυτός ο μηχανισμός επιβεβαίωσης μπορεί να παρέχεται από το MAC πρωτόκολλο(όπως το link level ack frame στο 802.11) , είτε να από την τεχνική της παθητικής επιβεβαίωσης. Η παθητική επιβεβαίωση(passive acknowledgment) μπορεί να γίνει όταν ένας κόμβος που έχει μεταδώσει ένα πακέτο σε έναν γειτονικό κόμβο τον ακούει μετά να το προωθεί και αυτός. Βέβαια οι παραπάνω τεχνικές δουλεύουν μόνο στην περίπτωση που τα link μεταξύ των κόμβων είναι συμμετρικά, σε αντίθετη περίπτωση ο κόμβος που μεταδίδει ένα πακέτο μπορεί να θέσει ένα συγκεκριμένο bit απαιτώντας από τον παραλήπτη να του στείλει μια επιβεβαίωση. Αν το link μεταξύ τους δεν είναι συμμετρικό τότε η επιβεβαίωση μπορεί να ακολουθήσει διαφορετικό μονοπάτι.

Όταν δεν ληφθεί επιβεβαίωση σε ένα κόμβο τότε το πακέτο επαναμεταδιδεται, αυτό γίνεται μέχρι ένα μέγιστο αριθμό φόρων. Αν τελικά

δεν έρθει επιβεβαίωση τότε ο κόμβος στέλνει ένα πακέτο ROUTE ERROR(RERR) στον κόμβο-αφετηρία. Αυτό το πακέτο περιέχει την ακμή του δικτύου από την οποία δεν ήταν δυνατή η αποστολή δεδομένων.

Όταν ο κόμβος-αφετηρία λάβει ένα τέτοιο μήνυμα τότε ενημερώνει την route cache, αφαιρώντας όλες τις διαδρομές που χρησιμοποιούν το link που έχει καταρρεύσει. Στην συνέχεια αν ο κόμβος-αφετηρία θέλει να μεταδώσει και άλλα πακέτα στο ίδιο προορισμό τότε θα πρέπει να εκκινήσει πάλι μια διαδικασία ανακάλυψης διαδρομής.

5.4 Βελτιστοποιήσεις του DSR

Πάνω στο βασικό DSR πρωτόκολλο που αναλύσαμε και πιο συγκεκριμένα στις δυο βασικές διαδικασίες (ανακάλυψης και συντήρησης διαδρομής) μπορούμε να εισάγουμε αρκετές βελτιστοποιήσεις που μειώνουν σημαντικά τον αριθμό των πακέτων που μεταδίδει το πρωτόκολλο και αυξάνουν την συνολική απόδοση του DSR.

1.4.1 Πλήρης χρησιμοποίηση της μνήμης αποθήκευσης διαδρομών

Οι διαδρομές στην route cache ενός κόμβου πρακτικά σχηματίζουν ένα δεντρό διαδρομών με ριζά τον κόμβο αυτόν, φύλλα τους προορισμούς και εσωτερικούς κόμβους του δέντρου τους ενδιάμεσους κόμβους των διαδρομών. Αυτή η δομή μας δίνει περισσότερες πληροφορίες που μπορούμε να εκμεταλλευτούμε. Για παράδειγμα στο σχήμα 2 βλέπουμε ένα δίκτυο με 5 κόμβους όπου ο κόμβος A έχει ολοκληρώσει μια διαδικασία ανακάλυψης διαδρομής προς τον κόμβο D. Αυτό σημαίνει ότι ο κόμβος A έχει αποθηκεύσει μια πλήρη διαδρομή προς τον D μεσώ των κόμβων B και C, άρα ο κόμβος A εκτός από μια διαδρομή για τον D γνωρίζει επιπλέον και διαδρομές για τους B,C. Αν αργότερα ο A ξεκινήσει μια άλλη διαδικασία ανακάλυψης διαδρομής για τον E και βρει ότι η διαδρομή αυτή έχει ενδιάμεσους κόμβους τους B,C λόγο την δεντρικής δομής των διαδρόμων θα χρειαστεί να αποθηκεύσει μόνο την επιπλέον πληροφορία ότι από τον C μπορεί να πάει άμεσα στον E.

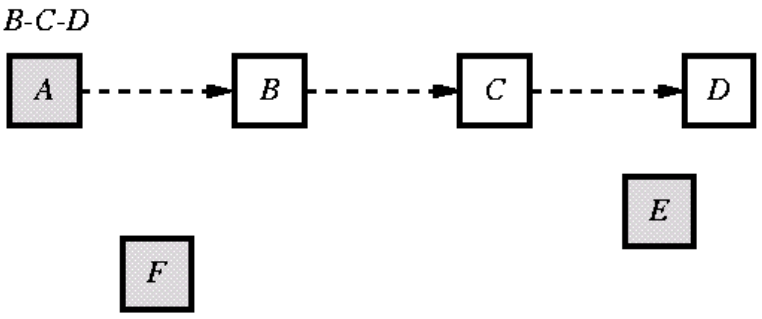


Figure 2 An example ad hoc network illustrating use of the route cache

Επιπλέον αν ο κόμβος C πλησιάσει στον κόμβο A και βρίσκεται στην ακτίνα μετάδοσης του, τότε ο A μπορεί να επιλέξει να μειώσει την απόσταση των διαδρόμων προς τον E και D αφαιρώντας τον κόμβο B από αυτές τις διαδρομές.

Μια ακόμα βελτίωση που θα μπορούσε να γίνει είναι να προσθέτει ένας κόμβος στην route cache του, ότι διαδρομές μαθαίνει από πακέτα που προωθεί. Αυτά μπορεί να είναι είτε πακέτα δεδομένων που στην επικεφαλίδα τους έχουν μια πλήρη διαδρομή, είτε πακέτα ROUTE REPLY. Επιπλέον αν μπορεί ένας κόμβος να θέσει το ασύρματο interface του σε promiscuous mode τότε μπορεί να προσθέτει και διαδρομές που μεταδίδουν άλλοι κόμβοι στην γειτονιά του.

Μια άλλη τεχνική για την μείωση των μηνυμάτων που χρειάζεται να στείλει ένας κόμβος στο δίκτυο για την εύρεση διαδρομών είναι να επιτρέπουμε σε ένα ενδιάμεσο κόμβο να απαντάει σε ένα ROUTE REQUEST πακέτο αν γνωρίζει μια διαδρομή προς τον κόμβο προορισμό. Ο ενδιάμεσος κόμβος που θα απαντήσει θα προσθέσει στο πεδίο των ενδιάμεσων κόμβων που θα πάρει από το ROUTE REPLY τους κόμβους από την διαδρομή προς τον κόμβο-προορισμό που έχει αποθηκευμένη. Για παράδειγμα αν στην εικόνα 2 ο κόμβος F στέλνει ένα ROUTE REQUEST για να βρει μια διαδρομή για τον κόμβο D τότε μπορεί να του απαντήσει άμεσα ο A στέλνοντας του την διαδρομή A,B,C,D.

Ένα σημαντικό πρόβλημα που παρουσιάζεται με την παραπάνω τεχνική είναι ότι όταν πολλοί κόμβοι λάβουν ένα ROUTE REQUEST από ένα κόμβο-αφετηρία και επειδή γνωρίζουν όλοι την ζητούμενη διαδρομή αρχίσουν να απαντούν γεμίζοντας το δίκτυο με μηνύματα και αυξάνοντας τις συγκρούσεις. Επιπλέον ένας κόμβος μπορεί να λάβει πολλές απαντήσεις από τους γείτονες που κάθε μια να έχει διαφορετικό μήκος.

Για να εξαλείψουμε τα παραπάνω προβλήματα απαιτούμε από κάθε κόμβο να εισάγει μια μικρή καθυστέρηση πριν στείλει μια απάντηση(ROUTE REPLY) όταν χρησιμοποιεί τις αποθηκευμένες διαδρομές του. Πιο συγκεκριμένα πριν απαντήσει ένας κόμβος σε αυτήν την περίπτωση ακολουθεί την παρακάτω διαδικασία:

1. Διαλέγει μια περίοδο καθυστέρησης d με d= H x (h-1+r) οποί h είναι το μήκος της διαδρομής μέχρι τον κόμβο που θα στείλει το ROUTE REPLY, r είναι ένας τυχαίος αριθμός από 0 ως 1, και H είναι μια μικρή σταθερά που είναι η καθυστέρηση που εισάγουμε ανά κόμβο.
2. Καθυστερεί την μετάδοση του ROUTE REPLY για χρονικό διάστημα d.
3. Σε όλη την διάρκεια που καθυστερεί μπαίνει σε promiscuous mode. Αν λάβει ένα ROUTE REPLY για την αίτηση του κόμβου-αφετηρία από ένα άλλο κόμβο μέσα σε αυτήν την περίοδο τότε ελέγχει αν το μήκος της διαδρομής στην απάντηση είναι μικρότερο από h. Αν είναι τότε δεν χρειάζεται να στείλει και δεύτερη απάντηση αφού κόμβος-αφετηρία θα λάβει την πρώτη απάντηση που είχε μικρότερο μήκος διαδρομής.

Ένα άλλο πρόβλημα που παρουσιάζεται όταν επιτρέπουμε ενδιάμεσους

κόμβους να στέλνουν απάντηση(ROUTE REPLY) χρησιμοποιώντας τις αποθηκευμένες διαδρομές τους, είναι η δημιουργία κύκλων στις διαδρομές που περιέχουν αυτές οι απαντήσεις. Αν και δεν καταχωρούνται διαδρομές που περιέχουν κύκλους αυτό το φαινόμενο είναι ανεπιθύμητο. Για παράδειγμα στο σχήμα 2 αν ο κόμβος B στείλει ένα ROUTE REQUEST για να βρει μια διαδρομή προς το D, τότε ο A μπορεί να του απαντήσει αμέσως συνενώνοντας την διαδρομή που γνωρίζει για τον D (A-B-C-D) με την διαδρομή προς τον B (A-B). Όμως αυτή η διαδρομή περιέχει κύκλους. Ένας απλός τρόπος για να το αποφύγουμε αυτό θα ήταν πριν ένας ενδιάμεσος κόμβος απαντήσει να ελέγχει την ύπαρξη κύκλων στην διαδρομή που θα στείλει ως απάντηση και αν υπάρχουν κύκλοι να μην την αποστείλει. Σε αυτήν την περίπτωση όμως ένας κόμβος θα απαντήσει μόνο αν βρίσκεται πάνω στο μονοπάτι από την αφετηρία στον προορισμό και συγκεκριμένα στο τέλος της διαδρομής που καταγράφεται στο ROUTE REQUEST και στην αρχή της διαδρομής που καταγράφεται στην route cache του. Όποτε στο παράδειγμα που δώσαμε πριν αν και ο γείτονας του B (ο A) ξέρει την διαδρομή προς τον D, δεν θα μπορεί να του την στείλει.

Τέλος μια ακόμα βελτίωση που μπορούμε να προσθέσουμε στον πρωτόκολλο είναι η τοποθέτηση ενός ορίου ενδιάμεσων κόμβων(hop limit) στο ROUTE REQUEST πακέτο. Αυτό το όριο περιορίζει το μήκος του μονοπατιού που μπορεί να προωθηθεί το ROUTE REQUEST πακέτο. Όποτε σε κάθε ROUTE REQUEST πακέτο έχουμε ένα αριθμό που μειώνεται καθώς αποστέλλεται από κόμβο σε κόμβο και όταν φτάσει μηδέν το πακέτο απορρίπτεται. Αυτός ο μηχανισμός χρησιμοποιείται για να στείλουμε ένα ROUTE REQUEST με όριο ενδιάμεσων κόμβων ίσο με 1, μαθαίνοντας με αυτόν τον τρόπο είτε αν ο κόμβος-προορισμός βρίσκεται στην γειτονιά του κόμβου-αφετηρία είτε αν κάποιος γείτονας του κόμβου-αφετηρία γνωρίζει μια διαδρομή προς τον προορισμό. Μια άλλη τεχνική που μπορούμε να εφαρμόσουμε είναι το expading ring search, δηλαδή όταν ένας κόμβος θέλει να ανακαλύψει μια διαδρομή για ένα προορισμό στέλνει πρώτα ένα ROUTE REQUEST με hop limit 1, αν δεν πάρει απάντηση μετά από ένα χρονικό διάστημα επαναλαμβάνει την διαδικασία με μεγαλύτερο hop limit μέχρι να καλύψει όλο το δίκτυο. Αυτή η τεχνική αν και μειώνει τα μηνύματα έλεγχου του πρωτοκόλλου είναι πιθανό να αυξάνει τον μέσο χρόνο απόκτησης μιας διαδρομής αφού αν ο προορισμός δεν βρίσκεται κοντά στον κόμβο-αφετηρία θα συμβούν αρκετά timeout μεταξύ των διαδοχικών αποστόλων ROUTE REQUEST.

5.4.2 Μείωση του μήκους των διαδρόμων

Είναι πολύ πιθανό λόγο της κινητικότητας των κόμβων κάποιοι κόμβοι σε μια διαδρομή να πλησιάσουν αρκετά κοντά και μπορούνε να επικοινωνήσουν άμεσα χωρίς την χρήση ενδιάμεσων κόμβων. Επειδή είναι επιθυμητό να διατηρούμε σε κάθε κόμβο τις μικρότερες δυνατές διαδρομές προς ένα προορισμό πρέπει να αφαιρούμε από την route cache τους περιττούς ενδιάμεσους κόμβους.

Αν οι κόμβοι μπορούνε να μπούνε σε promiscuous mode, τότε μπορούμε να εφαρμόσουμε μια τεχνική για την μείωση του μήκους των διαδρομών. Έστω ότι ο κόμβος B(σχήμα 3) μεταδίδει ένα πακέτο στον C και ο D είναι ο επόμενος ενδιάμεσος κόμβος που θα μεταδοθεί το πακέτο. Αν ο D πάρει το πακέτο(γιατί

είναι σε promiscuous mode) μπορεί να εξετάσει την επικεφαλίδα του για να δει ότι το πακέτο έφτασε από τον B άμεσα, όποτε μπορεί να υποθέσει ότι κινήθηκε κοντά στον B και είναι στην ακτίνα μετάδοσης του. Έτσι μπορεί να αποφασίσει να μειώσει την διαδρομή αφαιρώντας τον C από αυτήν. Μετά ο D στέλνει ένα ειδικό ROUTE REPLY στον κόμβο-αφετηρία του πακέτου ενημερώνοντας τον για αυτήν την αλλαγή. Αν και αυτή η τεχνική δεν έχει ενσωματωθεί στον πρωτόκολλο που χρησιμοποιήθηκε στις εξομοιώσεις είναι σίγουρο ότι θα διασφαλίζει τις μικρότερες δυνατές διαδρομές.

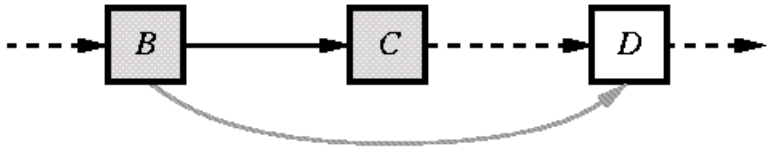


Figure 3 Mobile host *D* notices that the route can be shortened

5.4.4 Βελτιστοποίηση στον χειρισμό λαθών

Μια συχνή κατάσταση που πρέπει να λάβουμε υπ΄ όψη μας και να χειριστούμε στα ad-hoc δίκτυα είναι η περίπτωση το δίκτυο να διαιρεθεί σε τμήματα, δηλαδή σε ομάδες άπω κόμβους που κάθε κόμβος που βρίσκεται σε μια ομάδα δεν μπορεί να επικοινωνήσει με κανένα κόμβο άλλης ομάδας. Σε αυτήν την περίπτωση αν κάθε φορά που ένας κόμβος ήθελε να στείλει πακέτα σε ένα προορισμό σε μια άλλη ομάδα ξεκινούσε μια διαδικασία ανακάλυψης διαδρομής θα είχαμε συμφόρηση στο δίκτυο και μείωση του ωφέλιμου bandwidth. Για αυτό περιορίζουμε τον ρυθμό με τον οποίο μπορεί να γίνονται νέες διαδικασίες ανακάλυψης διαδρομής χρησιμοποιώντας ένα εκθετικό backoff διάστημα μεταξύ συνεχόμενων τέτοιων διαδικασιών. Αν ένας κόμβος προσπαθεί να στείλει πακέτα σε ένα προορισμό(για τον οποίο δεν έχουμε διαδρομή) με ρυθμό μεγαλύτερο από το το όριο που θέτουμε τότε αυτά τα δεδομένα μπορούμε να κρατάμε σε ένα buffer, μέχρι να λάβουμε ένα ROYTE REPLY.

Αν οι κόμβοι μπορούνε να λειτουργήσουν σε promiscuous mode, τότε θα μπορούσαμε να ακολουθήσουμε και την εξής τεχνική για τα ROUTE ERROR πακέτα. Όταν μεταδίδεται ένα ROUTE ERROR μήνυμα, όπως φαίνεται και στο σχήμα 4 από ένα κόμβο B σε ένα κόμβο A, τότε αν οι κόμβοι C,D,E είναι σε promiscuous mode μπορούνε να ακούσουνε το ROUTE ERROR και να μάθουν πιο link έχει καταρρεύσει. Οπότε μπορούνε να ελέγξουν την route cache τους για τις διαδρομές που χρησιμοποιούν αυτό το link, και να αφαιρέσουν όλους τους ενδιάμεσους κόμβους από το σημείο του υπάρχει το πρόβλημα στο link και μετά.

Αυτή η τεχνική παρουσιάζει όμως προβλήματα όταν λόγο μη συμμετρικών link το route error μεταδοθεί από διαφορετική διαδρομή και άρα οι κόμβοι C,D,E δεν μπορούνε να το λάβουν. Σε αυτή την περίπτωση μπορούμε να λύσουμε το πρόβλημα με το να απαιτήσουμε από τον κόμβο-αφετηρία που θα λάβει το route error να το ξαναστείλει πίσω στο B.

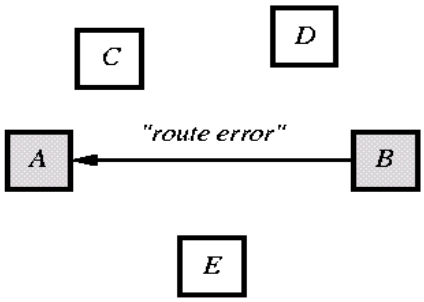


Figure 4 Mobile host *B* is returning a route error packet to *A*

Μια τελευταία βελτιστοποίηση στο χειρισμό προβληματικών καταστάσεων στο δίκτυο είναι η υποστήριξη για αποθήκευση αρνητικών πληροφοριών στην route cache ενός κόμβου. Με τον όρο αρνητική πληροφορία εννοούμε την πληροφορία για πρόσφατα χαμένα links στο δίκτυο. Ένα παράδειγμα για για το πως μπορεί να χρησιμοποιηθεί η αρνητική πληροφορία είναι το εξής(υποθέτουμε ότι οι παραπάνω βελτιστοποιήσεις δεν χρησιμοποιούνται). Αν στο σχήμα 4 ο A λάβει από τον B ένα route error και στείλει μετά ένα ROUTE REQUEST για να ανακαλύψει μια νέα διαδρομή είναι πολύ πιθανό να του απαντήσει ένας από τους C,D,E με την παλιά μη έγκυρη διαδρομή. Αν λοιπόν ο A κρατάει για ένα μικρο διάστημα την πληροφορία από το route error πακέτο για το πιο είναι το προβληματικό link μπορεί να απορρίπτει ROUTE REPLY πακέτα που περιέχουν διαδρομές που το χρησιμοποιούν.

6.1 Υλοποίηση του πρωτοκόλλου DSR στην πλατφόρμα SUNSPOT.

Στο μοντέλο που ακολουθεί η πλατφόρμα SUNSPOT το πρωτόκολλο δρομολόγησης βρίσκεται στο lowpan layer(lightweight ip layer). Το lowpan layer διαχειρίζεται την ανακατασκευή (reassembly) και κατάτμηση (fragmentation) των πακέτων, την παράδοση αυτών στα υψηλότερου επιπέδου πρωτόκολλα (όπως το radiostream) καθώς και την δρομολόγηση αυτών.

Για την υλοποίηση ενός πρωτοκόλλου δρομολόγησης στην πλατφόρμα SUNSPOT η sun παρέχει στην βιβλιοθήκη δικτύωσης multihoplib, πρότυπα interfaces που θα πρέπει να υλοποιεί το κάθε πρωτόκολλο. Τα interface που δίνονται είναι για την υλοποίηση hop-by-hop πρωτοκόλλων , δηλαδή οι ορισμοί των συναρτήσεων που δηλώνονται στο IRoutingManager δέχονται σαν ορίσματα και επιστρέφουν routes που ορίζονται από τον προορισμό το επόμενο hop, καθώς και τον συνολικό αριθμό των hops. Για την υλοποίηση του source routing πρωτοκόλλου DSR τροποποιήσαμε το παραπάνω interface για την υποστήριξη δομών που μπορούν να αναπαραστήσουν source routes.

Το lowpan layer για την αποδοτική λειτουργία του όποιου routing αλγόριθμου χρησιμοποιεί ένα call-back interface. Πιο συγκεκριμένα το σχήμα που χρησιμοποιείται είναι το εξής:

Όταν χρειάζεται να αναζητηθεί μια διαδρομή προς ένα προορισμό που δεν υπάρχει αποθηκευμένη, τότε στέλνετε από το lowpan layer μια αίτηση προς το routing manager(μέσω της findRoute()) που περιλαμβάνει την διεύθυνση του προορισμού καθώς και το instance του lowpan που περιμένει την απάντηση. Όταν μια διαδρομή προς τον προορισμό βρεθεί ή εξαντληθεί ένα συγκεκριμένο χρονικό όριο τότε καλείται από τον routing manager (χρησιμοποιώντας το instance που έχει αποθηκευμένο) η συνάρτηση routeFound() του lowpan.

Η αρχιτεκτονική του lowpan στην multihoplib είναι σχεδιασμένη για να συνεργάζεται με hop-by-hop routing managers. Προϋποθέτει ότι στον πίνακα δρομολόγησης για κάθε προορισμό αποθηκεύονται πληροφορίες σχετικά με τον επόμενο κόμβο(next-hop) την διαδρομής προς αυτόν τον προορισμό. Για αυτό, αν και οι λεπτομέρειες του αλγόριθμου δρομολόγησης μπορεί να διαφέρουν σε διαφορετικές υλοποιήσεις, υπάρχουν περιορισμοί ως προς δομή του αλγόριθμου. Επομένως για την υλοποίηση του DSR ήταν απαραίτητη και η τροποποίηση του lowpan layer προκειμένου να υποστηρίζει source routing αλγόριθμους. Πιο συγκεκριμένα η προκαθορισμένη λειτουργία του lowpan όταν λαμβάνει ένα πακέτο που δεν έχει τον ίδιο για προορισμό είναι να αναζητεί τον επόμενο κόμβο στον πίνακα δρομολόγησης. Στο DSR η πληροφορία σχετικά με την διαδρομή (τους ενδιάμεσους κόμβους) που πρέπει να ακολουθήσει το πακέτο δεν βρίσκεται κατανεμημένη στους κόμβους αλλά στην επικεφαλίδα του πακέτου ως μια διατεταγμένη λίστα από τις διευθύνσεις των ενδιάμεσων κόμβων που πρέπει να προωθήσουν το πακέτο. Επιπλέον εξαιτίας του προηγούμενου γεγονότος καταλαβαίνουμε πως έχουμε μεταβλητό μέγεθος επικεφαλίδας. Στη βιβλιοθήκη όμως το κάθε πακέτο του lowpan θεωρείται ότι έχει σταθερό μήκος επικεφαλίδας για το εκάστοτε πρωτόκολλο, οπότε στην υλοποίηση του DSR πρέπει να γίνουν οι κατάλληλες τροποποιήσεις όπου είναι αναγκαίο.

6.2 Περιορισμοί και παραδοχές στην υλοποίηση του DSR

Καταρχήν πρέπει να πούμε πως λόγο περιορισμών σε hardware δεν ήταν δυνατόν η πλήρη υλοποίηση όλων των χαρακτηριστικών του DSR. Πιο συγκεκριμένα:

- Λόγο του ότι έχουμε επιβεβαιώσεις σε επίπεδο MAC, δεν είναι δυνατό η ανακάλυψη μη συμμετρικών διαδρομών, δηλαδή να έχουμε διαφορετικές διαδρομές για την μετάδοση πακέτων από την πηγή προς τον προορισμό και αντίστροφα. Αυτό είναι προφανές γιατί σε περίπτωση που ένας κόμβος προσπαθήσει να στείλει ένα πακέτο πάνω σε ένα μια μη συμμετρική σύνδεση και δεν λάβει επιβεβαίωση για την αποστολή τότε θα θεωρήσει ότι το πακέτο χάθηκε και μετά από ένα όριο αναμεταδόσεων θα το δηλώσει ως αποτυχία (μέσω ενός RRER μυνήματος) ή αν πρόκειται για RREQ απλώς θα το αγνοήσει.
- Επιπλέον επειδή το radio στα SPOTs δεν μπορεί να μπει σε promiscuous mode, δεν ήταν δυνατόν η υλοποίηση ορισμένων χαρακτηριστικών όπως η λήψη πληροφοριών σχετικά με την κατάσταση των συνδέσεων που περιγράφεται σε μυνήματα RRER που δεν έχουν σαν τελικό ή ενδιάμεσο προορισμό τον κόμβο που το ακούει. Επίσης δεν ήταν δυνατό η υλοποίηση της τεχνικής μείωσης του μήκους των διαδρομών που περιγράφεται στο 1.4.2.

6.3 Λεπτομέρειες και ανάλυση της υλοποίησης

Αρχεία που περιλαμβάνουν τον κώδικα του DSR καθώς και αυτά της βιβλιοθήκης της sun που τροποποιήθηκαν βρίσκονται στον φάκελο multihoplib_source. Για να χρησιμοποιήσουμε το πρωτόκολλο σε κάποια εφαρμογή που τρέχει στο SPOTWorld αρκεί να αντιγράψουμε τα αρχεία που παρέχουμε, στον αντίστοιχο φάκελο του SUN SPOT SDK , να εκτελέσουμε “*ant jar-app*” για την μετάφραση του κώδικα και την δημιουργία του κατάλληλου jar της βιβλιοθήκης(multihoplib_rt.jar). Αν θέλουμε να την εγκαταστήσουμε σε μια πραγματική συσκευή τότε θα πρέπει να εκτελέσουμε την εντολή “*ant flashlibrary*” στο root φάκελο της multihoplib_source.

Όλα τα αρχεία πηγαίου κώδικα της υλοποίησης του DSR βρίσκονται στον υποφάκελο dsr του multihoplib_source. Παρακάτω αναλύουμε την βασική λειτουργία και παραθέτουμε τα βασικά τμήματα κώδικα κάθε ενός.

DSRManager.java:

Πρόκειται για την κύρια κλάση κάθε πρωτοκόλλου δρομολόγησης, που αναλαμβάνει την αρχικοποίηση των routing tables και την εκκίνηση άλλων κλάσεων για την διαχείριση των πακέτων του πρωτοκόλλου. Ακόμα περιλαμβάνει συναρτήσεις που υπαγορεύονται από το IroutingManager interface για την εκκίνηση, αρχικοποίηση αναστολή και τερματισμό του routing πρωτοκόλλου καθώς και βασικές συναρτήσεις για την εύρεση/ανάκτηση και διαγραφή διαδρομών. Κατά την αρχικοποίηση του lowpan δημιουργείται ένα αντικείμενο του DSRManager. Οι μέθοδοι αυτής της κλάσης χρησιμοποιούνται από το lowpan σε 3 περιπτώσεις.

1. Όταν ένα SPOT πρόκειται να στείλει ένα νέο πακέτο σε ένα προορισμό, οπότε και καλεί την getRouteInfo() που αναζητά στην route cache μια διαδρομή προς τον προορισμό. Η getRouteInfo() θα επιστρέψει μια δομή routeInfo που είτε θα περιέχει την πλήρη διαδρομή προς τον προορισμό είτε θα περιέχει την τιμή InvalidRoute που δηλώνει ότι δεν υπάρχει αποθηκευμένη μια κατάλληλη διαδρομή.

```
public RouteInfo getRouteInfo(long address) {
        RouteInfo info = routingTable.getNextHopInfo(address);
        return info;
}
```

2. Όταν χρειάζεται να αποσταλεί ένα πακέτο σε ένα προορισμό και δεν βρεθεί μια διαδρομή στη τοπική route cache. Οπότε και το lowpan καλεί την findRoute(), που εκκινεί μια διαδικασία εύρεσης διαδρομής

```
public boolean findRoute(long address, RouteEventClient eventClient, Object
uniqueKey) {
        return sender.sendNewRREQ(address, eventClient, uniqueKey);
}
```

3. Όταν αποτύχει να προωθήσει ένα πακέτο από σε ένα άλλο κόμβο(δεν λάβει πακέτο επιβεβαίωσης ACK). Οπότε κάλει την invalidateRoute() που αφερεί απο την route cache την άκυρη διαδρομή και στέλνει στον κόμβο αφετηρια ένα μύνημα RERR.

```
public boolean invalidateRoute(long originator, long destination,long
brokenLink,long[] SourceRoute) {
        if (originator == ourAddress) {
                routingTable.removeRoute(destination,new long[]
{originator,brokenLink});
                return true;
        }
        return sender.sendNewRERR(originator,destination,brokenLink,SourceRoute);
}
```

Receiver.java:

Αυτή η κλάση υλοποιεί την λογική για την λήψη και την διαχείριση των πακέτων του DSR πρωτοκόλλου. Στην αρχικοποίηση της receiver καταχωρεί (register) στο lowpan την κλάση με τον μοναδικό αριθμό του DSR πρωτοκόλλου(*103)*. Οι κλάσεις πρωτοκόλλων που γίνονται register στο lowpan πρέπει να υλοποιούν το IprotocolManager interface. Μετά την καταχώριση όλα τα πακέτα που λαμβάνει το lowpan, και έχουν τον συγκεκριμένο αριθμό του πρωτοκόλλου, τα προωθεί στη receiver καλώντας την συνάρτηση processIncomingData(), που υλοποιεί το protocol manager interface. Στην συνάρτηση αυτή αναγνωρίζεται ο τύπος του μυνήματος και στην συνέχεια τοποθετείται σε μία ουρά(*messageQueue*) για μετέπειτα επεξεργασία. Για να αποφύγουμε σπατάλη μνήμης και επεξεργαστικής ισχύς στην περίπτωση που έχουμε λάβει ένα πακέτο τύπου RREQ πριν το τοποθετήσουμε στην ουρά εξετάζουμε αν το έχουμε λάβει ήδη,δηλαδή αν υπάρχει στο requestTable. Σε περίπτωση που υπάρχει το αγνοούμε, αλλιώς το εισάγουμε. Επιπλέον έλεγχοι και επεξεργασία γίνονται μετέπειτα, αφού σε αυτή την μέθοδο πρέπει να περιορίσουμε στο ελάχιστο τους χρονοβόρους υπολογισμούς για να μην καθυστερεί το lowpan στην παράδοση πακέτων άλλων πρωτοκόλλων.

```
public void processIncomingData(byte[] payload, LowPanHeaderInfo headerInfo) {
    byte DSRMessageType = payload[0];

    switch (DSRMessageType) {
        case Constants.RREQ_TYPE:
            RREQ rreqMessage = new RREQ(payload);
            if (!requestTable.hasRequest(rreqMessage)) {
                if (messageQueue.size() < MAX_REQUESTS_OUTSTANDING) {
                    requestTable.addRREQ(rreqMessage, null, null);
                    messageQueue.put(new ReceivedPacket(rreqMessage,
headerInfo.sourceAddress));
                }
            }
            break;
        case Constants.RREP_TYPE:
            RREP rrepMessage = new RREP(payload);
            messageQueue.put(new ReceivedPacket(rrepMessage,
headerInfo.sourceAddress));
            break;
        case Constants.RERR_TYPE:
            RERR rerrMessage = new RERR(payload);
            messageQueue.put(new ReceivedPacket(rerrMessage,
headerInfo.sourceAddress));
            break;
        default:
            System.err.println("receiveData: bad packet from"
                    + new IEEEAddress(headerInfo.sourceAddress).asDottedHex());
            break;
    }
}
```

Για την επεξεργασία των εισερχόμενων πακέτων γίνεται στην μέθοδο run() της threaded κλάσης Receiver. Σε αυτή έχουμε ένα loop(while) που σε κάθε επανάληψη αφαιρεί ένα πακέτο από την ουρά εξετάζει τον τύπο του πακέτου και ανάλογα καλεί την κατάλληλη μέθοδο για περαιτέρω επεξεργασία. Πιο συγκεκριμένα οι συναρτήσεις που χειρίζονται τα RERR RERQ και RRER είναι αντίστοιχα οι:

- *handleRERRMessage()* - Η μέθοδος αυτή δέχεται ένα RERR πακέτο αυξάνει το αριθμό των κόμβων που έχει διανύσει και μετά εξετάζει αν έχει φτάσει στον προορισμό του. Στην περίπτωση που βρίσκεται στον τελικό προορισμό του τότε αφαιρούμε από την route cache την μη έγκυρη διαδρομή, σε διαφορετική περίπτωση προωθούμε το πακέτο στον επόμενο κόμβο προς τον προορισμό.

```
private void handleRERRMessage(RERR message) {

    if (!mhRouteListeners.isEmpty()) {
       Enumeration en = mhRouteListeners.elements();
       while (en.hasMoreElements()) {
          ((IMHEventListener) en.nextElement()).RERRReceived(message.getOrigAddress(),
message.getDestAddress());
       }
    }
    message.incrementHopCount();

    if (message.getOrigAddress() == ourAddress){
       routingTable.removeRoute(message.getDestAddress(),message.getBrokenLink());
    } else {
       sender.forwardDSRMessage(message);
    }
  }
```

- *handleRREQMessage()* – Σε αυτή την μέθοδο επεξεργαζόμαστε τα RREQ πακέτα του πρωτοκόλλου. Στην αρχή εξετάζουμε αν πρόκειται για ένα πακέτο που έχει προέλθει από αυτόν τον κόμβο, οπότε και το απορρίπτουμε. Μετά αυξάνουμε τον αριθμό των hop που έχει διανύσει και αν υπερβαίνει ένα όριο το απορρίπτουμε. Τέλος ελέγχουμε αν η αίτηση έφτασε τον προορισμό της, οπότε και στέλνουμε για απάντηση ένα RREP πακέτο με την μέχρι τώρα διαδρομή που έχει διανύσει το RREQ. Επιπλέον προσθέτουμε την διαδρομή μέχρι την αφετηρία στην route cache. Αν δεν είμαστε ο τελικός προορισμός το προωθούμε στο δίκτυο μέσω της κλάσης sender.

```
private void handleRREQMessage(RREQ message, long lastHop) {

    if (message.getOrigAddress() != ourAddress) {
       message.incrementHopCount();
       if (message.getHopCount() > Constants.NET_DIAMETER) return;
```

```
        if (message.getDestAddress() == ourAddress) {
            routingTable.addRoute(lastHop, message);
            RREP routeFoundMessage = new RREP(message);
            sender.sendNewRREP(routeFoundMessage);

        } else {
           if (!rpm.isEndNode()) {
              sender.forwardDSRMessage(message);
           }
        }
     }
  }
```

- *handleRREPMessage*() - Στην κλάση αυτή διαχειριζόμαστε τα RREP που λαμβάνει ένας κόμβος. Πρώτα ελέγχουμε αν το πακέτο έχει φτάσει στην αφετηρία(που έστειλε το αντίστοιχο RREQ). Σε αυτή την περίπτωση προσθέτουμε στην route cache την διαδρομή, και μέσω του call back interface ειδοποιούμε όσους είχαν ζητήσει αυτό το route(lowpan). Σε διαφορετική περίπτωση το προωθούμε στον επόμενο κόμβο.

```
   private void handleRREPMessage(RREP message, long lastHop) {

      message.incrementHopCount();

      if (message.getOrigAddress() == ourAddress) {
            routingTable.addRoute(lastHop, message);

         while (requestTable.hasRequest(message)) {
            RequestEntry rqe = requestTable.getRequestEntryAndRemove(message);
            if(rqe.client!=null)
            {
               RouteInfo info = new
RouteInfo(message.getDestAddress(),lastHop,message.getHopCount(),message.getPropSour
ceRoute());
               rqe.client.routeFound(info,rqe.uniqueKey);
            }
            else
            {
               System.err.println("handleRREPMessage: null client asssociated");
            }
         }
      } else {
         if (!rpm.isEndNode()) { // Only forward RREP if not an end node
            sender.forwardDSRMessage(message);
         }
      }
   }
```

Sender.java:

Πρόκειται για την κλάση που είναι υπεύθυνη για την αποστολή και την προώθηση όλων των μηνυμάτων ελέγχου του DSR, δηλαδή των μηνυμάτων RERR,RREP,RREQ. Αυτά μπορεί να προέρχονται είτε από το ίδιο το SPOT(πχ από την receiver) είτε να είναι μηνύματα άλλων SPOTs που θα τα προωθήσει κατάλληλα. Τα μηνύματα αυτά δεν στέλνονται απευθείας, άλλα τοποθετούνται πρώτα σε κατάλληλες ουρές(queues). Δηλαδή όταν μια άλλη κλάση χρειάζεται να στείλει ένα μήνυμα ελέγχου καλεί μια συνάρτηση του sender που τοποθετεί το μήνυμα προσωρινά σε μια ουρά για μελλοντική αποστολή. Ο λόγος που χρησιμοποιούμε αυτή την σχεδίαση είναι για βέλτιστη απόδοση της υλοποίησης, επειδή οι συναρτήσεις για αποστολή πακέτων είναι blocking και οι ταυτόχρονες αιτήσεις θα είχαν ως αποτέλεσμα την αναμονή πολλών τμημάτων του λογισμικού στο ίδιο σημείο κάτι που πρέπει να αποφύγουμε. Σε αυτήν την κλάση έχουμε τρεις queues για την διαχείριση των μηνυμάτων ελέγχου του πρωτοκόλλου, σε αυτές προσθέτουμε μυνήματα που θέλουμε η sender να προωθήσει:

1. outgoingDSRMessageQueue
   Σε αυτήν τοποθετούμε RERR,RREP,RREQ μηνύματα που έχουμε λάβει από άλλα SPOT και δεν έχουν φτάσει στον προορισμό τους ακόμα, και πρόκειται να τα προωθήσουμε.
2. RouteWantedQueue
   Σε αυτή την ουρά προσθέτουμε RREQ μηνύματα για την εύρεση διαδρόμων προς προορισμούς που για χρειάζονται για τον ίδιο κόμβο.
3. routeErrorQueue
   Αυτή η ουρά είναι για τα μυνήματα RRER. Όταν ένας κόμβος δεν λάβει μήνυμα επιβεβαίωσης αποστολής για ένα πακέτο που έστειλε τότε θεωρεί αυτήν την διαδρομή άκυρη και στέλνει ένα μήνυμα λάθους διαδρομής πίσω στην αφετηρία(τον κόμβο που έστειλε αρχικά το μήνυμα).

Η λειτουργία της threaded κλάσης sender συνοψίζεται στην συνάρτηση run() όπου εκτελείται όσο υπάρχουν πακέτα στις ουρές αλλιώς παραμένει blocked περιμένοντας να προστεθούν. Υπάρχουν τρεις συναρτήσεις για την επεξεργασία και αποστολή πακέτων από τις παραπάνω ουρές ανάλογα με τον τύπο τους.

1. *SendRREQ*() - Η μέθοδος αυτή αποστέλλει πακέτα RREQ σε όλους τους γειτονικούς κόμβους. Για να αποφύγουμε συγκρούσεις με τις αναμεταδόσεις των RREQ από άλλους κόμβους καθυστερούμε για ένα μικρό διάστημα την διαδοχική αποστολή πακέτων. Επίσης προσθέτουμε κάθε μήνυμα και την κλάση που ζήτησε την διαδρομή σε ένα πίνακα που τον διαχειρίζεται η requestTable. Αυτό γίνεται για να αποφύγουμε μελλοντικές αναμεταδόσεις του ίδιου RREQ και επιπλέον για να μπορούμε να ενημερώσουμε την κλάση(lowpan) που ζήτησε την διαδρομή όταν αυτή είναι διαθέσιμη.

```
private void sendRREQ(RREQ message, RouteEventClient eventClient,
    Object uniqueKey) throws ChannelBusyException, NoAckException {
```

```
    try {
       int r = randomGen.nextInt(50);
       Thread.sleep(r * 5);
    } catch (InterruptedException ie) {
    }

    byte[] buffer = message.writeMessage();
    if (!requestTable.hasRequest(message))
       requestTable.addRREQ(message, eventClient, uniqueKey);
    lowPan.sendBroadcast(Constants.DSR_PROTOCOL_NUMBER, buffer, 0,
          buffer.length, 0);

       }
    }
  }
```

2. *SendRREP*() - Η μέθοδος αυτή στέλνει ή προωθεί ένα RREP μήνυμα πίσω στην αφετηρία. Η διαδρομή κατασκευάζεται αντιστρέφοντας την διαδρομή του RREQ που έφτασε τον προορισμό. Σε περίπτωση που δεν λάβουμε πακέτο επιβεβαίωσης σε μια μετάδοση, τότε επαναλαμβάνουμε την αποστολή μέχρι MAX_RETRIES(3) φορές πριν το εκλάβουμε σαν αποτυχία. Σε περίπτωση που αποτύχει η μετάδοση στέλνουμε ένα RERR μήνυμα στον προορισμό για να αφαιρέσει την διαδρομή από την route cache του.

```
  private void sendRREP(RREP message) throws ChannelBusyException {
    byte[] buffer = message.writeMessage();

    long destinationAddress = message.getNextHop();


      for (int i=0; i< MAX_RETRIES; i++) {
         try {
            lowPan.sendWithoutMeshingOrFragmentation(Constants.DSR_PROTOCOL_N
UMBER,
                  destinationAddress, buffer, 0, buffer.length);
            break;
         } catch (NoAckException e) {
            sendNewRERR(message.getDestAddress(),
message.getOrigAddress(),destinationAddress,message.getRSRoute());
         }
  }
```

3. *SendRERR*() - Η *SendRERR()* χρησιμοποιείται για να στείλει ή να προωθήσει ένα RERR μήνυμα σε ένα κόμβο. Τυπικά καλείται όταν το lowpan αποτύχει στην αποστολή ενός πακέτου προς ένα γειτονικό κόμβο, για να ενημερώσει την αφετηρία ότι η διαδρομή δεν ισχύει πια. Όμως όπως είδαμε στην προηγούμενη μέθοδο μπορεί να χρησιμοποιηθεί για να ενημερώσει τον προορισμό ότι μια διαδρομή που έμαθε από ένα RREQ μήνυμα είναι άκυρη.

```
      private void sendRERR(RERR message) throws ChannelBusyException {

    byte[] buffer = message.writeMessage();
    long destinationAddress = message.getNextHop();

      for (int i=0; i< MAX_RETRIES; i++) {
          try {
             lowPan.sendWithoutMeshingOrFragmentation(Constants.DSR_PROTOCOL_N
UMBER,
                    destinationAddress, buffer, 0, buffer.length);
             break;
          } catch (NoAckException e) {
          }
    } else {
        Debug.print("sendRERR: can't find a next hop for RERR", 1);
    }

  }
```

RoutingTable.java:

Περιέχει την κλάση RoutingTable που είναι υπεύθυνη για την αποθήκευση και την διαχείριση των routes. Οι διαδρομές(routes) καταχωρούνται ως στοιχεία της κλάσης RoutingEntry, η οποία περιέχει την πλήρη διαδρομή ως ένα array από διευθύνσεις κόμβων και ένα timestamp που χρησιμοποιείται για την διαγραφή των routes που μένουν αχρησιμοποίητα για κάποιο διάστημα. Τα RoutingEntrys αποθηκεύονται σε ένα hashtable με hash key την διεύθυνση του προορισμού. Επειδή όλες οι κλάσεις μοιράζονται ένα κοινό στιγμιότυπο του routingtable με την μέθοδο getInstance(), φροντίζουμε να συγχρονίζουμε την πρόσβαση στο hashtable με locks. Κατά την αρχικοποίηση της κλάσης εκκινείται ένα thread(το RoutingTableCleaner) που περιοδικά ελέγχει τον πίνακα για τα παλαιότερα routes που πρέπει να διαγραφούν. Το πόσο παλαιό θεωρείται ένα route σχετίζεται με την κινητικότητα των κόμβων, σε δίκτυα με υψηλή κινητικότητα αναμένουμε τα links να καταρρέουν συχνά οι διαδρομές θα θεωρούνται άκυρες μετά από μικρό σχετικά διάστημα. Η προεπιλεγμένη τιμή βρίσκεται στο αρχείο Constants.java και είναι 30sec.

RequestTable.java:

Η κλάση RequestTable έχει παρόμοια λειτουργία με την RoutingTable, αντί όμως για routes αποθηκεύει μηνύματα RREQ. Τα μηνύματα αποθηκεύονται σε ένα hashtable και σε μια λίστα. Η λίστα έχει τα στοιχεία διατεταγμένα σύμφωνα με την χρονική στιγμή που τα εισάγαμε, και διαγράφονται μετά από ένα διάστημα PATH_DISCOVERY_TIME. Την διαγραφεί αναλαμβάνει η RequestTableCleaner που εκκινείται με την αρχικοποίηση του RequestTable.

Constants.java:

Περιλαμβάνει ένα σύνολο από σταθερές που χρησιμοποιούνται για να καθορίσουν τις παραμέτρους του αλγόριθμου DSR καθώς και ειδικές σταθερές σχετικές με την υλοποίηση. Οι πιο σημαντικές είναι οι:

```
int NET_DIAMETER = 10;
byte DSR_PROTOCOL_NUMBER = 103;
```

που ορίζουν την μέγιστη διάμετρο του ad-hoc δικτύου και το αναγνωριστικό αριθμό του πρωτοκόλλου. Ο περιορισμός στο μέγιστο μήκος διαδρομής τίθεται για 2 λόγους. Πρώτον γιατί για κάθε κόμβο που προωθείται ένα μήνυμα εύρεσης διαδρομής προστίθεται η διεύθυνση του στην επικεφαλίδα του, οπότε το μέγιστο μήκος μια διαδρομής περιορίζεται από ωφέλιμο payload του radiogram του 802.15.4(αφού δεν μπορούμε να έχουμε fragmentation κατά την αποστολή που μυνήματος απάντησης στον αρχικό κόμβο). Και δεύτερον λόγο όσο μεγαλύτερη είναι η διαδρομή που πρέπει να ακολουθήσει ένα πακέτο προς τον προορισμό τόσο περισσότερο χρόνο χρειάζεται για να έρθει η απάντηση πίσω στην αφετηρία, όμως λόγο της κινητικότητας των κόμβων η πιθανότητα να μετακινηθεί ένας κόμβος κατά μήκος της και να διασπαστεί αυξάνεται με τον χρόνο οπότε κρατάμε την διάμετρο μικρή ώστε να αποφύγουμε συχνές αποτυχίες. Αυτό σχετίζεται και με τις ειδικές σταθερές που ορίζουμε:

```
AVERAGE_RANDOM_BACKOFF = 200;
NODE_TRAVERSAL_TIME = 30;
NET_TRAVERSAL_TIME=900
PATH_DISCOVERY_TIME=8400
ACTIVE_ROUTE_TIMEOUT = 30000;
```

Πιο αναλυτικά οι τιμές είναι σε milliseconds και προκύπτουν ως εξής:

Η μεταβλητή random_backoff είναι μέση αναμονή ανάμεσα σε δύο αναμεταδόσεις RREQ πακέτων. Είναι αναγκαίο να εισάγουμε μια καθυστέρηση ανάμεσα στις αναμεταδόσεις για να αποφύγουμε συγκρούσεις. Για παράδειγμα αν ένας κόμβος μεταδόσεις ένα μήνυμα RREQ και λάβουν 2 ή περισσότεροι κόμβοι που βρίσκονται στην ίδια περιοχή τότε αν προσπαθήσουν να το προωθήσουν αμέσως τότε με μεγάλη πιθανότητα θα έχουμε σύγκρουση αυτών των πακέτων. Οπότε σε περιπτώσεις που ο DSR routing manager μεταδίδει διαδοχικά broadcast μηνύματα ελέγχου εφαρμόζει τον παραπάνω μηχανισμό.

Η τιμή NODE_TRAVERSAL_TIME είναι ο χρόνος που χρειάζεται ένα SPOT για να επεξεργαστεί και να μεταδώσει ένα πακέτο και μαζί με την random_backoff χρησιμοποιείται για να υπολογίσουμε τον χρόνο NET_TRAVERSAL_TIME(ο μέσος χρόνος που χρειάζεται ένα πακέτο για να διανύσει την διάμετρο του δικτύου και να επιστρέψει στην αφετηρία) και τον χρόνο PATH_DISCOVERY_TIME(τον χρόνο που χρειάζεται ένα RREQ μήνυμα για να διανύσει την διάμετρο του δικτύου και να επιστρέψει στην αφετηρία). Αυτές οι τιμές χρησιμοποιούνται για να αποφασίσουμε τον μέγιστο χρόνο αναμονής για μια απάντηση σε μια αίτηση εύρεσης διαδρομής.

Βιβλιογραφία

[1] Hyoung Jun Kim,Won Jay Song, Sang Ha Kim: Light-Weighted Internet Protocol Version 6 for low-Power Wireless Personal Area Networks, 2007
[2] Evans-Pughe C. : ZigBee wireless standard, IEEE Review Volume 49, Issue 3, March 2003 Pages: 28-31
[3] Crossbow MICAz 2.4GHz.
http://www.xbow.com/Products/productdetails.aspx?
sid=164.
[4] Crossbow IRIS 2.4GHz.
http://www.xbow.com/Products/productdetails.aspx?
sid=264.
[5] BTnodes - A Distributed Environment for Prototyping
Ad Hoc Networks. http://www.btnode.ethz.ch/.
[6] ScatterWeb ScatterNode.
http://www.scatterweb.de/content/products/
industry line/scatternode.de.html.
[7] ScatterWeb.
http://www.scatterweb.de/content/products/
research line/.
[8] Coalesenses iSense - A modular hardware and software
platform for wireless sensor networks.
http://www.coalesenses.com/isense.
[9] Sun Microsystems,
http://www.sunspotworld.com/products
[10] squawk vm architecture,
https://squawk.dev.java.net/index-more.html
[11] SunSPOT Theory Of Operation,
http://www.sunspotworld.com/docs/Purple/SunSPOT-TheoryOfOperation.pdf
[12]Spot Developers Guide,
http://www.sunspotworld.com/docs/Purple/spot-developers-guide.pdf
[13]Sun SPOT Emulator,
http://www.sunspotworld.com/docs/Purple/SunSPOT-Emulator.pdf
[14] I. Chatzigiannakis, A. Kinalis, A. S. Poulakidas,
G. Prasinos, and C. D. Zaroliagis. Dap: A generic
platform for the simulation of distributed algorithms.
In Annual Simulation Symposium, pages 167{177.
IEEE Computer Society, 2004.
[15]Chatzigiannakis, Ioannis, Koninis, Christos, Prasinos, Grigorios and
Zaroliagis, Christos, Distributed Simulation of Heterogeneous Systems of
Small Programmable Objects and Traditional Processors, in: 6th ACM
Workshop on Mobility Management and Wireless Access (MOBIWAC 08), ACM,
ACM, Vancouver, Canada, 2008.
[16]David B. Johnson & David A. Maltz, Dynamic Source Routing in Ad Hoc Wireless Networks, 1996 .