ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ

ΤΜΗΜΑ

ΗΛΕΚΤΡΟΝΙΚΩΝ ΜΗΧΑΝΙΚΩΝ &
ΜΗΧΑΝΙΚΩΝ Η/Υ

Αλγόριθμοι αναγνώρισης εξεχουσών τιμών σε Δίκτυα Αισθητήρων

Outliers Detection Algorithms in Sensor Networks

Συγγραφέας:
Αντώνιος Ιγγλεζάκης

Επιβλέπων:
Επίκ. Καθ. Αντώνιος Δεληγιαννάκης

Εξεταστική Επιτροπή:

Επίκ. Καθ. Αντώνιος Δεληγιαννάκης
Καθ. Μίνως Γαροφαλάκης
Επίκ. Καθ. Βασίλειος Σαμολαδάς

2 Απριλίου 2012

Περιεχόμενα

Ευχαριστίες

Πρώτα απ΄ όλα, ευχαριστώ τον επιβλέπων καθηγητή Δρ. Αντώνιο Δεληγιαννάκη, για την καθοδήγηση του, τις συμβουλές του, τις γνώσεις του, την εμπειρία του, καθώς και για την άψογη συνεργασία που είχαμε. Να ευχαριστήσω επίσης τους καθηγητές της εξεταστικής επιτροπής, κ. Γαροφαλάκη και κ. Σαμολαδά για το χρόνο που αφιέρωσαν στην ανάγνωση και αξιολόγηση της παρούσας εργασίας. Επιπλέον θα ήθελα να ευχαριστήσω τους καθηγητές κ. Λαγουδάκη, κ. Χαλκιαδάκη και κ. Σαμολαδά για τις χρήσιμες συμβουλές τους κατά την διάρκεια φοίτησης μου στο Πολυτεχνείο Κρήτης.

Τέλος, θα ήθελα να ευχαριστήσω τους γονείς μου, τα αδέρφια μου και τους φίλους μου, και τους συμφοιτητές μου, που με στήριξαν σε όλες τις εύκολες και δύσκολες στιγμές των φοιτητικών μου χρόνων, όπως και να τους ζητήσω συγνώμη, για τις όσες φορές τους στεναχώρησα ή/και τους θύμωσα.

Περίληψη

Το πρόβλημα τη ανίχνευσης εξεχουσών τιμών (outliers) σε δίκτυα αισθητήρων έχει υπάρξει αντικείμενο έρευνας σε πολλές εργασίες τα τελευταία χρόνια. Υπάρχουν πολλοί τρόποι για να οριστεί κάποια μέτρηση ή κάποιος κόμβος ως outlier, άλλα γενικά ως outlier μπορεί να θεωρηθεί ο κόμβος που οι μετρήσεις του αποκλίνουν σημαντικά από τις μετρήσεις άλλων ή γειτονικών του κόμβων. Η ανίχνευση μη φυσιολογικών μετρήσεων παρουσιάζει ενδιαφέρον καθώς με αυτόν τον τρόπο μπορούν να εντοπιστούν δυσλειτουργίες σε κόμβους είτε ενδιαφέροντα γεγονότα, όπως πυρκαγίες.
Στην εργασία αυτήν υλοποιήθηκαν σε TinyOS αλγόριθμοι, βασισμένοι στην γεωμετρική προσέγγιση, για ανίχνευση εξεχουσών τιμών σε δίκτυα αισθητήρων. Η εργασία αυτή αποτελεί την πρώτη υλοποίηση των αλγορίθμων αυτών σε TinyOS. Για την εκτέλεση των αλγορίθμων σε δίκτυο αισθητήρων του εργαστηρίου υλοποιήθηκαν παράλληλα μηχανισμοί διατήρησης και ανάκτησης στατιστικών και μετρήσεων από τους αισθητήρες, καθώς επίσης και δομή δικτύου στην οποία στηρίχθηκε η εκτέλεση των αλγορίθμων. Οι αλγόριθμοι αυτοί δοκιμάστηκαν και εκτελέστηκαν σε δίκτυο αισθητήρων του εργαστηρίου.

Κεφάλαιο 1

Εισαγωγή

Το πρόβλημα τη ανίχνευσης εξεχουσών τιμών (outliers) σε δίκτυα αισθητήρων έχει υπάρξει αντικείμενο έρευνας σε πολλές εργασίες τα τελευταία χρόνια. Υπάρχουν πολλοί τρόποι για να οριστεί κάποια μέτρηση ή κάποιος κόμβος ως outlier, αλλά γενικά ως outlier μπορεί να θεωρηθεί ο κόμβος που οι μετρήσεις του αποκλίνουν σημαντικά από τις μετρήσεις άλλων ή γειτονικών του κόμβων. Η ανίχνευση μη φυσιολογικών μετρήσεων παρουσιάζει ενδιαφέρον καθώς με αυτόν τον τρόπο μπορούν να εντοπιστούν δυσλειτουργίες σε κόμβους είτε ενδιαφέροντα γεγονότα, όπως πυρκαγιές

Στην εργασία αυτήν υλοποιήθηκαν σε TinyOS αλγόριθμοι, βασισμένοι στην γεωμετρική προσέγγιση, για ανίχνευση εξεχουσών τιμών σε δίκτυα αισθητήρων. Η εργασία αυτή αποτελεί την πρώτη υλοποίηση των αλγορίθμων αυτών σε TinyOS. Για την εκτέλεση των αλγορίθμων σε δίκτυο αισθητήρων του εργαστηρίου υλοποιήθηκαν παράλληλα μηχανισμοί διατήρησης και ανάκτησης στατιστικών και μετρήσεων από τους αισθητήρες, καθώς επίσης και δομή δικτύου στην οποία στηρίχθηκε η εκτέλεση των αλγορίθμων. Οι αλγόριθμοι αυτοί δοκιμάστηκαν και εκτελέστηκαν σε δίκτυο αισθητήρων του εργαστηρίου.

Η εργασία αυτή αποτελεί την πρώτη υλοποίηση αλγορίθμου σε TinyOS που έγινε έως τώρα στο τμήμα και εκτός από την υλοποίηση χρειάστηκε να ξεπεραστούν και άλλα προβλήματα, όπως η ελλιπής τεκμηρίωση τμημάτων του TinyOS, η ανύπαρκτη υλοποίηση τμημάτων για προσομοίωση (Storage, FTSP), δημιουργία δομών δικτύου καθώς και ελλειπή ή/και ανύπαρκτα εργαλεία για ανάπτυξη και αποσφαλμάτωση. Στην συνέχεια αυτού του κεφαλαίου γίνεται μια σύντομη περιγραφή των δικτύων αισθητήρων και του υλικού του αισθητήρα. Στο δεύτερο κεφάλαιο γίνεται μια εισαγωγή στον προγραμματισμό εφαρμογών σε TinyOS καθώς επίσης και μερικών βιβλιοθηκών και μηχανισμών του. Στο τρίτο κεφάλαιο περιγράφονται θεωρητικά οι αλγόριθμοι που υλοποιήθηκαν ενώ στο τέταρτο κεφάλαιο περιγράφεται η υλοποίηση και η σχεδίαση της εφαρμογής και τεκμηριώνονται οι σχεδιαστικές επιλογές. Στο πρώτο παράρτημα, συνο-

ψίζονται τα βασικά interfaces που χρησιμοποιήθηκαν από τις βιβλιοθήκες του TinyOS.

1.1 Ασύρματα δίκτυα αισθητήρων

Ένα δίκτυο αισθητήρων αποτελείται συνήθως από ένα μεγάλο αριθμό μικρών κόμβων αισθητήρων χαμηλού κόστους που κατανέμονται σε μια μεγάλη περιοχή, με έναν ή περισσότερους σταθμούς βάσης, συλλέγοντας τις μετρήσεις των αισθητήρων. Τα δίκτυα αισθητήρων αποτελούνται χαρακτηριστικά από τις μικρές συσκευές που εξοπλίζονται με μια πηγή ισχύος, μια μονάδα επεξεργασίας με περιορισμένη επεξεργαστική ισχύ και μνήμη, μία ή περισσότερες συσκευές αίσθησης (όπως για θερμοκρασία, υγρασία, ακτινοβολία, θερμότητα, πίεση, ήχο, δόνηση, τάση κ.α.) από τις οποίες λαμβάνουν μετρήσεις και μια μονάδα επικοινωνίας για να αναμεταδώσουν αυτές τις μετρήσεις ή το αποτέλεσμα της επεξεργασίας τους σε άλλους αισθητήρες που βρίσκονται στην εμβέλεια τους.

Οι πρόσφατες πρόοδοι στη μικροηλεκτρονική έχουν επιτρέψει την ανάπτυξη μεγάλης κλίμακας δικτύων αισθητήρων για ποικίλες εφαρμογές παρακολούθησης και ελέγχου. Τέτοιες εφαρμογές αποτελούν οι οικολογικές εφαρμογές (παρακολούθηση της πανίδας ή ενός οικολογικού συστήματος), εφαρμογές στην υγειονομική περίθαλψη (την υγεία και την ιατρική παρακολούθηση), στρατιωτικές εφαρμογές (ανίχνευση και παρακολούθηση οχημάτων πίσω από τις εχθρικές γραμμές, την επιτήρηση πεδίων μαχών), εφαρμογές για έλεγχο κυκλοφορίας, γεωργικών καλλιεργειών, έλεγχο παραγωγής και εφαρμογές ελέγχου καταστροφών (παρακολούθηση εκ των προτέρων ενός επικίνδυνου καιρικού φαινόμενου πχ τσουνάμι ή μιας πυρκαγιάς).

1.1.1 Επεξεργασία δεδομένων σε ασύρματα δίκτυα αισθητήρων

Η επεξεργασία δεδομένων αποτελεί σημαντικό κομμάτι των εφαρμογών στα δίκτυα αισθητήρων. Χωρίς την κατανεμημένη επεξεργασία των μετρήσεων, συλλέγονται μεγάλοι όγκοι δεδομένων, ως επί των πλείστων χωρίς να προσφέρουν επιπλέον χρήσιμη πληροφορία, στους σταθμούς βάσης ενώ γίνεται και σπάταλη χρήση των πόρων των αισθητήρων (μπαταρίες, επεξεργαστική ισχύ). Η επεξεργασία των δεδομένων τοπικά στο δίκτυο αισθητήρων δίνει την δυνατότητα αποφυγής αποστολής άσκοπων μηνυμάτων και την δυνατότητα αποστολής μόνο χρήσιμης πληροφορίας ή ενημέρωσης για ενδιαφέροντα γεγονότα. Τέτοιες εφαρμογές μπορούν να απαιτούν από τους αισθητήρες να συλλέγουν τις μετρήσεις που κρίνονται 'παρόμοιες' ή να ανιχνευθεί εάν οι μετρήσεις των αισθητήρων είναι 'παρόμοιες', με τις μετρήσεις των κοντινών αισθητήρων ([2]). Ενδιαφέρον

παρουσιάζει η διαδικασία ανίχνευσης της περίπτωσης όπου οι μετρήσεις των δύο κόμβων δεν είναι όμοιες, επειδή το γεγονός αυτό μπορεί να βοηθήσει στον εντοπισμό 1. είτε κάποιας δυσλειτουργίας του κόμβο 2. είτε έναν κόμβο που ξεκινάει να παρατηρεί ένα τοπικά ενδιαφέρον φαινόμενο (πχ, μια πυρκαγιά) . Η διαδικασία αυτή αναφέρεται συχνά ως ανίχνευση εξεχουσών τιμών (Outliers Detection).

Η ανίχνευση εξεχουσών τιμών μπορεί να αναφέρεται είτε για τοπικές αλλαγές σε περιορισμένη περιοχή ελέγχου, είτε για καθολικές αλλαγές που αφορούν όλο το δίκτυο. Οι διάφορες τεχνικές για ανίχνευση εξεχουσών τιμών βασίζονται σε χρονικές ή χωρικές συσχετίσεις ή συσχετίσεις πυκνότητας. Υπάρχουν διάφοροι τρόποι για την ταξινόμηση και τον εντοπισμό των ακραίων τιμών.

Σε αυτήν την εργασία υλοποιήθηκαν στο TinyOS, ένα λειτουργικό σύστημα για αισθητήρες ειδικά σχεδιασμένο για την ανάπτυξη εφαρμογών για δίκτυα αισθητήρων, τρεις αλγόριθμοι, με χρήση της γεωμετρική προσέγγισης, για ανίχνευση εξεχουσών τιμών. Οι αλγόριθμοι αυτοί περιγράφονται στο κεφάλαιο 3 της παρούσας εργασίας και αποτελούν μέρος της δημοσίευσης [1] των Α. Δεληγιαννάκη και Σ. Μπουρδάκη.

Η εργασία αυτή αποτελεί την πρώτη υλοποίηση αλγορίθμου σε TinyOS που έγινε έως τώρα στο τμήμα και εκτός από την υλοποίηση χρειάστηκε να ξεπεραστούν και άλλα προβλήματα, όπως η ελλιπής τεκμηρίωση τμημάτων του TinyOS, η ανύπαρκτη υλοποίηση τμημάτων για προσομοίωση (Storage, FTSP), δημιουργία δομών δικτύου, καθώς και ελλειπή ή/και ανύπαρκτα εργαλεία για ανάπτυξη και αποσφαλμάτωση. Στην συνέχεια αυτού του κεφαλαίου γίνεται μια σύντομη περιγραφή του υλικού του αισθητήρα. Στο δεύτερο κεφάλαιο γίνεται μια εισαγωγή στον προγραμματισμό εφαρμογών σε TinyOS καθώς επίσης και μερικών βιβλιοθηκών και μηχανισμών του. Στο τρίτο κεφάλαιο περιγράφονται θεωρητικά οι αλγόριθμοι που υλοποιήθηκαν ενώ στο τέταρτο κεφάλαιο περιγράφεται η υλοποίηση και η σχεδίαση της εφαρμογής και τεκμηριώνονται οι σχεδιαστικές επιλογές. Στο πρώτο παράρτημα συνοψίζονται τα βασικά interfaces που χρησιμοποιήθηκαν από τις βιβλιοθήκες του TinyOS.

1.2 Ο Αισθητήρας, σαν συσκευή

Όπως γίνεται αντιληπτό από τα εισαγωγικά για τα δίκτυα αισθητήρων, ένας αισθητήρας δεν αποτελείται απλά από ένα σύνολο αισθητηρίων. Η συσκευή του αισθητήρα αποτελείται από μικροελεγκτή ή μικροεπεξεργαστή, μονάδα ασύρματης επικοινωνίας, μονάδα μόνιμης μνήμης, σύνολο αισθητηρίων, μπαταρίες ή μονάδα παροχής ισχύος, και για τον προγραμματισμό τους και διασύνδεση με τον Η/Υ, μονάδα σειριακής επικοινωνίας.

Σχήμα 1.1: Φωτογραφία του αισθητήρα Memsic IRIS

Σε αυτήν την ενότητα περιγράφεται το υλικό του αισθητήρα και τα περιφερειακά του.

1.2.1 Κυρίως μέρος τους αισθητήρα

Το κύριο μέρος αποτελείται από τις βασικές μονάδες του αισθητήρα, μονάδα μικροελεγκτή, μονάδα ασύρματης επικοινωνίας, μονάδας μόνιμης μνήμης και μονάδα παροχής ισχύος (μερικές φορές). Αυτές οι μονάδες είναι οι απολύτως απαραίτητες για την λειτουργία του αισθητήρα μέσα στο δίκτυο και για την εκτέλεση του προγράμματος. Το κυρίως μέρος συνδέεται με τα περιφερειακά μέρη, όπως βάση προγραμματισμού και πλακέτα αισθητηρίων. Στην εικόνα 1.1 είναι μια φωτογραφία του κύριου μέρους του αισθητήρα Memsic IRIS. Αυτό το μοντέλο αισθητήρα διαθέτει και το εργαστήριο.

Μικροελεγκτής

Ο μικροελεγκτής είναι ο επεξεργαστής του αισθητήρα. Συνήθως είναι τύπου AVR, 8-bit ή 16-bit. Λειτουργεί σε πολύ χαμηλές συχνότητες ρολογιού και έχει χαμηλή κατανάλωση ενέργειας. Επίσης διαθέτει μικρή μνήμη προγράμματος (64KBytes ή 128KBytes) και μικρή προσωρινή μνήμη (4KBytes - 8KBytes).Στην εικόνα 1.2 φαίνεται ένας πίνακας με συνοπτική περιγραφή των χαρακτηριστικών μερικών μοντέλων αισθητήρων της εταιρείας Memsic.Στην εικόνα 1.3 φαίνεται ένας πίνακάς με την κατανάλωση των μερών μερικών αισθητήρων της Memsic ενώ στην εικόνα 1.4 φαίνεται το διάγραμμα μονάδων του αισθητήρα Memsic IRIS.Οι αισθητήρες Memsic IRIS που διαθέτει το εργαστήριο έχουν τον 8-bit μικροελεγκτή Atmel Atmega 1281 με μνήμη προγράμματος 128KB, προσωρινή μνήμη 8KB και συχνότητα λειτουργίας στα 7.37 MHz.

Mote Hardware Platform		IRIS	MICAz	MICA2	MICA2DOT
Models (as of April 2005)		XM2110	MPR2400	MPR400/410/420	MPR500/510/520
MCU	**Chip**	ATMega1281	ATMega128L		
	Type	7.37 MHz, 8 bit			4 MHz, 8 bit
	Program Memory (kB)	128			
	SRAM (kB)	8	4		
Sensor Board Interface	**Type**	51 pin			18 pin
	10-Bit ADC	7, 0 V to 3 V input			6, 0 V to 3 V input
	UART	2			1
	Other interfaces	DIO, I2C			DIO
RF Transceiver (Radio)	Chip	RF230	CC2420	CC1000	
	Radio Frequency (MHz)	2400		315/433/915	
	Max. Data Rate (kbits/sec)	250		38.4	
	Antenna Connector	MMCX			PCB solder hole
Flash Data Logger Memory	Chip	AT45DB014B			
	Connection Type	SPI			
	Size (kB)	512			
Default power source	Type	AA, 2×			Coin (CR2354)
	Typical capacity (mA-hr)	2000			560

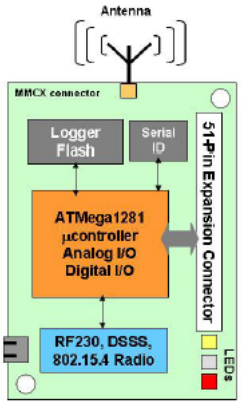
Σχήμα 1.2: Πίνακας χαρακτηριστικών αισθητήρων της Memsic

Operating Current (mA)	IRIS	MICAz	MICA2	MICA2DOT
Processor, full operation	8 (7.37 MHz)	12 (7.37 MHz)	12 (7.37 MHz)	6 (4MHz)
Processor, sleep	0.008	0.010	0.010	0.010
Radio, receive	16	19.7	7	7
Radio, transmit (1 mW power)	17	17	10	10
Radio, sleep	0.001	0.001	0.001	0.001
Serial flash memory, write	15			
Serial flash memory, read	4			
Serial flash memory, sleep	0.002			

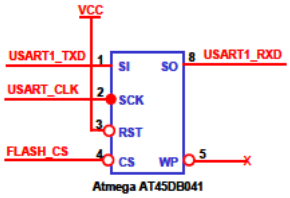
Σχήμα 1.3: Κατανάλωση ενέργειας μονάδων αισθητήρων της Memsic

Μονάδα ασύρματης επικοινωνίας

Οι μονάδες ασύρματης επικοινωνίας που χρησιμοποιούνται στους αισθητήρες είναι σχεδιασμένες για χαμηλή κατανάλωση ενέργειας και μπορούν να αλλάζουν συχνότητα λειτουργίας και ενέργεια μετάδοσης μέσω λογισμικού. Ο αισθητήρας Memsic IRIS χρησιμοποιεί το ολοκληρωμένο Atmel RF230, το οποίο λειτουργεί σε συχνότητες 2400MHz-2483.5MHz. Επίσης είναι συμβατό με το



Σχήμα 1.4: Διάγραμμα μονάδων του αισθητήρα Memsic IRIS



Σχήμα 1.5: Σχεδιάγραμμα σύνδεσης ολοκληρωμένου Atmega AT45DB041

πρωτόκολλο IEEE 802.15.4 και μπορεί να επιτύχει ταχύτητες μετάδοσης έως 250Kbps. Τα κανάλια, σύμφωνα με το IEEE 802.15.4 ,έχουν εύρος ζώνης 5 MHz και κατ΄ επέκταση το Atmel RF230 , μπορεί να λειτουργήσει στα κανάλια 11(2405 MHz)έως 26(2480 MHz). Στην εικόνα 1.6, φαίνονται οι στάθμες ενέργειας μετάδοσης στις οποίες μπορεί να ρυθμιστεί το RF230.

Μνήμη Flash

Οι μονάδες μόνιμης μνήμης που χρησιμοποιούνται από τους αισθητήρες είναι συνήθως αρκετές για την αποθήκευσης μερικών χιλιάδων μετρήσεων και για την χρήση τους για τον ασύρματο προγραμματισμό τους. Ο αισθητήρας Memsic IRIS , που διαθέτει το εργαστήριο, χρησιμοποιεί το ολοκληρωμένο Atmega AT45DB041 για την αποθήκευση 4-Mbit (512 KBytes) δεδομένων. Στην εικόνα 1.5 φαίνεται το σχεδιάγραμμά σύνδεσης του ολοκληρωμένου με τον μικροελεγκτή.

RF Power (dBm)	Power Register (code)
3.0	0
2.6	1
2.1	2
1.6	3
1.1	4
0.5	5
-0.2	6
-1.2	7
-2.2	8
-3.2	9
-4.2	10
-5.2	11
-7.2	12
-9.2	13
-12.2	14
-17.2	15

Σχήμα 1.6: Στάθμες ενέργειας αποστολής Atmega RF230

Mote Hardware Platform	Standard Battery (# required)	Typical Battery Capacity (mA-hr)	Practical Operating Voltage Range (V)
IRIS	AA (2)	2000, Alkaline	3.6 to 2.7
MICAz	AA (2)	2000, Alkaline	3.6 to 2.7
MICA2	AA (2)	2000, Alkaline	3.6 to 2.7
MICA2DOT	Coin	560, Li-ion	3.6 to 2.7

Σχήμα 1.7: Σχεδιάγραμμα σύνδεσης ολοκληρωμένου Atmega AT45DB041

Τροφοδοσία

Οι αισθητήρες είναι σχεδιασμένοι να τροφοδοτούνται κυρίως από μπαταρίες. Η εικόνα 1.7 δείχνει έναν πίνακα με τις μορφές τροφοδοσίας που απαιτούν μερικά μοντέλα αισθητήρων την εταιρείας Memsic. Οι περισσότεροι αισθητήρες παρέχουν και επαφές για σύνδεση με εξωτερική τροφοδοσία.
Στην εικόνα 1.8 φαίνεται ένα παράδειγμα υπολογισμού διάρκειας μπαταρίας ανάλογα τα χαρακτηριστικά ενός αισθητήρα.

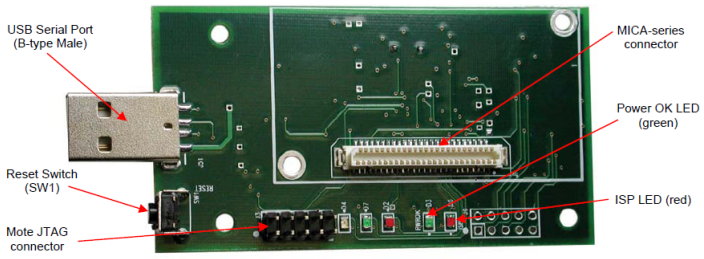
1.2.2 Περιφερειακά του αισθητήρα

Σαν περιφερειακά του αισθητήρα εννοούμε τις πλακέτες αισθητηρίων (Sensorboards), και τις βάσεις προγραμματισμού (Mote Interface Boards). Οι πλακέτες αισθητηρίων έχουν συνδεδεμένα τα αισθητήρια όργανα, ενώ οι βάσεις προγραμματισμού παρέχουν την συνδεσιμότητα με τον Η/Υ, και επιπλέον δυνατότητες επικοινωνίας με τον μικροελεγκτή για αποσφαλμάτωση(JTAG).
Το εργαστήριο, διαθέτει την βάση προγραμματισμού (MIB520CB 1.9) και sensorboard MDA100CB (1.10).

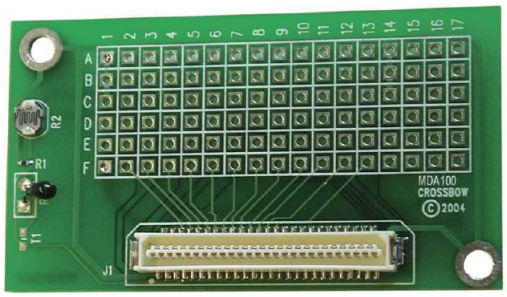
Το sensorboard MDA100CB παρέχει δύο αισθητήρια, μια φωτοαντίσταση

SYSTEM SPECIFICATIONS		
Currents		**Example Duty Cycle**
Processor		
Current (full operation)	8 **mA**	1
Current sleep	8 µA	99
Radio		
Current in receive	8 mA	0.75
Current transmit	12 mA	0.25
Current sleep	2 µA	99
Logger Memory		
Write	15 mA	0
Read	4 mA	0
Sleep	2 µA	100
Sensor Board		
Current (full operation)	5 mA	1
Current sleep	5 µA	99
Computed mA-hr used each hour		
Processor		0.0879
Radio		0.0920
Logger Memory		0.0020
Sensor Board		0.0550
Total current (mA-hr) used		**0.2369**
Computed battery life vs. battery size		
Battery Capacity (mA-hr)		Battery Life (months)
250		1.45
1000		5.78
3000		17.35

Σχήμα 1.8: Παράδειγμα υπολογισμού διάρκειας μπαταρίας.



Σχήμα 1.9: Βάση προγραμματισμού MIB520CB



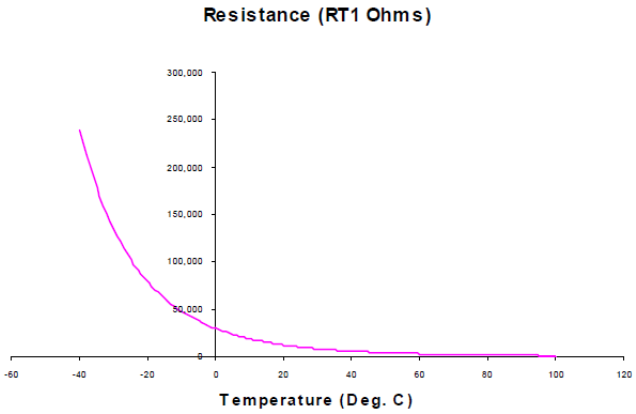
Σχήμα 1.10: Πλακέτα αισθητηρίων MDA100CB.

και ένα thermistor . Η συμπεριφορά του καθενός από αυτά τα αισθητήρια δεν είναι γραμμική και για τον υπολογισμό της μέτρησης σε μονάδες έντασης φωτός και μονάδες θερμοκρασίας απαιτεί περαιτέρω γνώση κατασκευαστικών χαρακτηριστικών. Επίσης παρέχει και χώρο στην πλακέτα για σύνδεση επιπλέον αισθητηρίων. Τα κανάλια του ADC (Analog to Digital Converter) έχουν ανάλυση 10-bit και κατ΄ επέκταση οι μετρήσεις που παίρνει ο μικροελεγκτής είναι τιμές στάθμης τάσης 0-1023 σε σχέση με την τάση αναφοράς (τροφοδοσίας). Για το thermistor του sensorboard MDA100CB ισχύει ο παρακάτω τύπος μετατροπής σε βαθμούς Kelvin.

$$\begin{aligned}
\frac{1}{T(K)} &= a + bln(R_{thr}) + c[ln(R_{thr})]^3 \\
R_{thr} &= R1(\frac{ADC_FS - ADC}{ADC}) \\
a &= 0.001010024, \\
b &= 0.000242127, \\
c &= 0.000000146, \\
R1 &= 10k\Omega, \\
ADC_FS &= 1023,
\end{aligned}$$

Η χαρακτηριστική καμπύλη του thermistor και ένας πίνακας με ενδεικτικές τιμές θερμοκρασίας φαίνονται στην εικόνα 1.11.

Temperature (°C)	Resistance (Ohms)	ADC5 Reading (% of VCC)
-40	239,800	4%
-20	78,910	11%
0	29,940	25%
25	10,000	50%
40	5592	64%
60	2760	78%
70	1990	83%



Σχήμα 1.11: Χαρακτηριστική καμπύλη thermistor MDA100CB,και ενδεικτικές τιμές θερμοκρασίας για ποσοστά τάσης αναφοράς.

Κεφάλαιο 2

Εισαγωγή στο TinyOS

Το TinyOS είναι ένα ανοικτού κώδικα λειτουργικό σύστημα σχεδιασμένο για να τρέχει σε μικρά συστήματα ασύρματων αισθητήρων, παρέχοντας στις εφαρμογές τους απαραίτητους μηχανισμούς αφαίρεσης (abstractions) από πλατφόρμα σε πλατφόρμα. Τα προγράμματα που τρέχουν στο TinyOS είναι γραμμένα στην γλώσσα προγραμματισμού NesC , η οποία αποτελεί μια διάλεκτο της C βασισμένη σε components. Σε αυτό το κεφάλαιο γίνεται μια σύντομη εισαγωγή στο TinyOS , και στην γλώσσα NesC , με σκοπό να γίνεται κατανοητή η λογική του σχεδιασμού μίας εφαρμογής καθώς επίσης και τα κομμάτια κώδικα που θα παρουσιαστούν σε επόμενα κεφάλαια.

2.1 Το TinyOS

Το TinyOS αποτελεί ένα μικρό λειτουργικό σύστημα ειδικά σχεδιασμένο για ασύρματους αισθητήρες χαμηλής κατανάλωσης. Το TinyOS διαφέρει από τα υπόλοιπα λειτουργικά συστήματα, στο ότι η σχεδίαση τους επικεντρώνεται στην εκμετάλλευση των χαμηλής κατανάλωσης και μικρών δυνατοτήτων μικροελεκτών που έχουν οι αισθητήρες, για να κάνει την δημιουργία εφαρμογών ασύρματων αισθητήρων ευκολότερη. Για να το πετύχει αυτό παρέχει ένα σύνολο υπηρεσιών και αφαιρέσεων για βασικές λειτουργίες, όπως η λήψη τιμών μέτρησης από αισθητήρα, η χρήση της flash memory, η επικοινωνία, και η μέτρηση χρόνου. Επιπλέον παρέχει μηχανισμούς για ταυτόχρονη χρήση πόρων όπως και μηχανισμούς αφαίρεσης για χρήση του ίδιου κώδικα σε διαφορετικές πλατφόρμες.

2.1.1 Εφαρμογές στο TinyOS

Οι εφαρμογές για το TinyOS καθώς επίσης και το ίδιο το TinyOS , είναι γραμμένα στην γλώσσα προγραμματισμού NesC . Η NesC είναι μια διάλεκτος της C με επιπλέον χαρακτηριστικά για μείωση χρήσης μνήμης RAM και μεγέθους κώδικα. Επιπλέον εισάγει σημαντικές βελτιστοποιήσεις και βοηθά στην αποφυγή σφαλμάτων χαμηλού επιπέδου, όπως race conditions. Μεταγλωττίζοντας μία εφαρμογή για το TinyOS παράγεται ένα εκτελέσιμο (binary image), το οποίο έχει τον πλήρη έλεγχο του υλικού του αισθητήρα. Κατ΄ επέκταση κάθε αισθητήρας τρέχει μόνο ένα εκτελέσιμο (binary image) ανά πάσα στιγμή. Η εφαρμογή και το κομμάτι του TinyOS που έχει μεταγλωττιστεί μοιράζονται τον ίδιο χώρο μνήμης, δηλαδή δεν υπάρχει διαχωρισμός μεταξύ μνήμης συστήματος και χρήστη καθώς επίσης δεν υπάρχει και υποστήριξη υλικού για αυτό
Όπως όλες οι εφαρμογές που γράφονται στην NesC , έτσι και οι εφαρμογές του TinyOS στηρίζονται σε ένα μοντέλο βασισμένο σε components. Κάθε εφαρμογή αποτελείται από ένα ή περισσότερα components συνδεδεμένα και υλοποιημένα έτσι ώστε να μπορεί να παραχθεί ένα εκτελέσιμο. Κάθε εκτελέσιμο αποτελείται μόνο από τα components που χρειάζεται η εφαρμογή. Κάθε component παρέχει ή/και χρησιμοποιεί interfaces , καθορίζοντας έτσι τον τρόπο που συνδέεται και αλληλεπιδρά με άλλα. Τα interfaces περιγράφουν τον τρόπο επικοινωνίας των components μεταξύ τους. Το κάθε interface αποτελείται από *α΄)* commands , τα οποία υλοποιούνται από το component που παρέχει το inteface, και *β΄)* events , τα οποία υλοποιούνται από το component που χρησιμοποιεί το inteface .

Υπάρχουν δύο είδη components στην NesC , *α΄)* modules , στα οποία συμπεριλαμβάνεται ο κώδικας που θα εκτελείται, *β΄)* configurations , στα οποία δηλώνεται ποια components συνδέονται με ποια καθώς επίσης και ποια components χρησιμοποιούν ή παρέχουν τα interfaces που παρέχει αυτό. . Η υλοποίηση κάθε component πρέπει να βρίσκεται σε ένα αρχείο με ονομασία <`όνομα του component`>`.nc` . Κάθε component αποτελείται από δύο μέρη: *α΄)* δήλωση των interfaces που χρησιμοποιεί ή παρέχει και *β΄)* την υλοποίηση των interfaces . Κάθε component μπορεί να παρέχει ή/και να χρησιμοποιεί κανένα, ένα ή παραπάνω interfaces , ακόμα και του ίδιου είδους. Για κάθε interface που παρέχει, υλοποιεί τα commands και για κάθε interface που χρησιμοποιεί, υλοποιεί τα events . Το σύνολο των interfaces που παρέχει ή χρησιμοποιεί ένα component , αποτελεί την περιγραφή του. Στο υψηλότερο επίπεδο μία εφαρμογή περιγράφεται από ένα configuration component , το οποίο δεν παρέχει, ούτε χρησιμοποιεί κάποιο interface .

2.1.2 Η εφαρμογή Blink

Στο κομμάτι κώδικα 2.2 βλέπουμε την υλοποίησή της εφαρμογής Blink από τα παραδείγματα του TinyOS source tree [10]. Η εφαρμογή αυτή δείχνει έναν μετρητή στα τρία leds του αισθητήρα. Το πρώτο led αλλάζει κατάσταση με συχνότητα 4 Hz , το δεύτερο με συχνότητα 2 Hz και το τρίτο με συχνότητα 1 Hz . Η εφαρμογή Blink αποτελείται από δύο components: *α΄*) ένα module component 2.2 , που ονομάζεται BlinkC και *β΄*) ένα configuration component 2.1, που ονομάζεται BlinkAppC . Το configuration component αυτό, αποτελεί την περιγραφή της εφαρμογής στο υψηλότερο επίπεδο. Παρατηρούμε ότι στο BlinkAppC component γίνεται η σύνδεση του BlinkC component με τα components που παρέχουν τα interfaces που χρησιμοποιεί. Σε αυτό το σημείο, γίνεται φανερό ο κύριος λόγος διαχωρισμού των components σε δύο κατηγορίες. Ο διαχωρισμός αυτός γίνεται για να μπορεί ο προγραμματιστής μιας εφαρμογής να χρησιμοποιήσει components βιβλιοθήκης ή άλλων εφαρμογών που δεν έχει υλοποιήσει ο ίδιος.

```
1  configuration BlinkAppC
2  {
3  }
4  implementation
5  {
6    components MainC, BlinkC, LedsC;
7    components new TimerMilliC() as Timer0;
8    components new TimerMilliC() as Timer1;
9    components new TimerMilliC() as Timer2;
10
11
12   BlinkC -> MainC.Boot;
13
14   BlinkC.Timer0 -> Timer0;
15   BlinkC.Timer1 -> Timer1;
16   BlinkC.Timer2 -> Timer2;
17   BlinkC.Leds -> LedsC;
18 }
```

Κώδικας 2.1: Κώδικας NesC του configuration component BlinkAppC

Το `module BlinkC` , χρησιμοποιεί τρία στιγμιότυπα του `interface Timer` (2.3) , για να χρησιμοποιήσει τρεις διαφορετικούς Timers , που μετράνε milliseconds , με ονόματα `Timer0 , Timer1 , Timer2,` έναν για κάθε led , το `interface Leds` (2.4) , για να δίνει εντολή ανάμματος και σβησίματος των leds και το `interface Boot` (2.5) , για να ξεκινάει την εκτέλεση μετά το άναμμα του αισθητήρα. Αυτό σημαίνει ότι στο `implementation` κομμάτι θα πρέπει να υλοποιήσει όλα τα events που είναι δηλωμένα σε όλα τα interfaces που χρησιμοποιεί, καθώς επίσης μπορεί να κάνει κλήσεις των commands που είναι δηλωμένες σε αυτά. Για τα τρία στιγμιότυπα του interface Timer , θα

```
1
2  #include "Timer.h"
3
4  module BlinkC @safe()
5  {
6    uses interface Timer<TMilli> as Timer0;
7    uses interface Timer<TMilli> as Timer1;
8    uses interface Timer<TMilli> as Timer2;
9    uses interface Leds;
10   uses interface Boot;
11 }
12 implementation
13 {
14   event void Boot.booted()
15   {
16     call Timer0.startPeriodic( 250 );
17     call Timer1.startPeriodic( 500 );
18     call Timer2.startPeriodic( 1000 );
19   }
20
21   event void Timer0.fired()
22   {
23     dbg("BlinkC", "Timer 0 fired @ %s.\n", sim_time_string());
24     call Leds.led0Toggle();
25   }
26
27   event void Timer1.fired()
28   {
29     dbg("BlinkC", "Timer 1 fired @ %s \n", sim_time_string());
30     call Leds.led1Toggle();
31   }
32
33   event void Timer2.fired()
34   {
35     dbg("BlinkC", "Timer 2 fired @ %s.\n", sim_time_string());
36     call Leds.led2Toggle();
37   }
38 }
```

Κώδικας 2.2: Κώδικας NesC του module component BlinkC

πρέπει να υλοποιήσει τρία ξεχωριστά `fired()` events .

Για το `interface Boot` , θα πρέπει να υλοποιηθεί το `event booted()` . Σε αυτό το event γίνεται η αρχικοποίηση της εφαρμογής. Για την εφαρμογή Blink, ξεκινάει τους τρεις timers να 'χτυπάνε' περιοδικά κάθε 250, 500, 1000 milliseconds αντίστοιχα. Το interface Leds δεν αποτελείται από κάποιο event , οπότε και δεν απαιτείται να υλοποιηθεί κάτι από την εφαρμογή. Όταν ο Timer0 γίνει fired, θα εκτελεστεί το `event Timer0.fired()` και μέσα από αυτό θα κλήση του `command Leds.led0Toggle()` για αλλαγή κατάστασης του led 0. Αντίστοιχα, το `event Timer1.fired()` θα κάνει αλλαγή κατάστασης στο led 1, και το `event Timer2.fired()` στο led 2.

Στο κομμάτι κώδικα 2.1, είναι η υλοποίηση του configuration component

```
#include "Timer.h"

interface Timer<precision_tag>
{  // basic interface
  command void startPeriodic(uint32_t dt);
  command void startOneShot(uint32_t dt);
  command void stop();
  /**
   * Signaled when the timer expires (one-shot) or repeats (periodic).
   */
  event void fired();
  // extended interface
  command bool isRunning();
  command bool isOneShot();
  /**
   * Set a periodic timer to repeat every dt time units. Replaces any
   * current timer settings. The <code>fired</code> will be signaled every
   * dt units (first event at t0+dt units). Periodic timers set in the past
   * will get a bunch of events in succession, until the timer "catches up".
   *
   * <p>Because the current time may wrap around, it is possible to use
   * values of t0 greater than the <code>getNow</code>'s result. These
   * values represent times in the past, i.e., the time at which getNow()
   * would last of returned that value.
   *
   * @param t0 Base time for timer.
   * @param dt Time until the timer fires.
   */
  command void startPeriodicAt(uint32_t t0, uint32_t dt);
  /**
   * Set a single-short timer to time t0+dt. Replaces any current timer
   * settings. The <code>fired</code> will be signaled when the timer
   * expires. Timers set in the past will fire "soon".
   *
   * <p>Because the current time may wrap around, it is possible to use
   * values of t0 greater than the <code>getNow</code>'s result. These
   * values represent times in the past, i.e., the time at which getNow()
   * would last of returned that value.
   *
   * @param t0 Base time for timer.
   * @param dt Time until the timer fires.
   */
  command void startOneShotAt(uint32_t t0, uint32_t dt);
  command uint32_t getNow();
  command uint32_t gett0();
  command uint32_t getdt();
}
```

Κώδικας 2.3: Δήλωση του interface Timer

της εφαρμογής. Στο `implementation` μέρος, γίνεται η δήλωση της σύνδεσης του `module BlinkC` με τα υπόλοιπα components παρέχουν τα interfaces που χρησιμοποιεί. Το `inteface Leds` παρέχεται από το `component LedsC`, το `interface Timer` για milliseconds παρέχεται από τα στιγμιότυπα του `component TimerMilliC`, ενώ το `interface Boot` παρέχεται από το `component MainC` . Με την λέξη κλειδί `components` δηλώνονται τα components που θα χρησιμο-

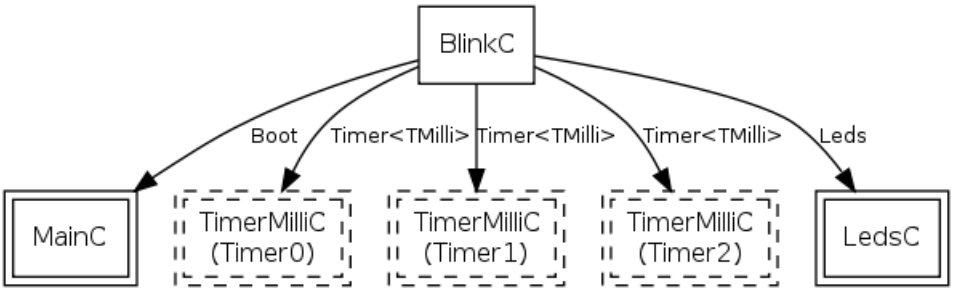
```
1  #include "Leds.h"
2
3  interface Leds {
4
5    async command void led0On();
6
7    async command void led0Off();
8
9    async command void led0Toggle();
10
11   async command void led1On();
12
13   async command void led1Off();
14
15   async command void led1Toggle();
16
17   async command void led2On();
18
19   async command void led2Off();
20
21   async command void led2Toggle();
22
23
24   /**
25    * Get the current LED settings as a bitmask.
26    * @return a bitmask describing which LEDs are on and which are off
27    */
28   async command uint8_t get();
29
30
31   /**
32    * Set the current LED configuration using a bitmask.
33    * @param  val   a bitmask describing the on/off settings of the LEDs
34    */
35    async command void set(uint8_t val);
36
37 }
```

Κώδικας 2.4: Δήλωση του interface Leds

ποιηθούν. Με τα <— και —>δηλώνεται η σύνδεση μεταξύ των interfaces των components, με το βελάκι να έχει φορά από το component που χρησιμοποιεί το interface στο component που το παρέχει. Μπορούμε να παραλείψουμε το όνομα του interface από την πλευρά του component-παρόχου, αν αυτό έχει το ίδιο όνομα με την πλευρά του component-χρήστη. Με την λέξη κλειδί `new` δηλώνουμε ότι θα δημιουργηθεί στιγμιότυπου του component. Components-στιγμιότυπα μπορούν να δημιουργηθούν μόνο όταν είναι δηλωμένα με την λέξη κλειδί `generic` (...παρακάτω 2.4). Όταν το configuration component παρέχει ή χρησιμοποιεί με την σειρά του interfaces, τότε αυτά τα interfaces θα πρέπει να συνδέονται με την χρήση του '=' αντί για τα —>και <— με interfaces από components που τα χειρίζονται με τον ίδιο τρόπο (όταν το interface χρησιμοποιείται από το configuration component, τότε αυτό με την σειρά του το συνδέει με ένα component που το χρησιμοποιεί, ομοίως αν το interface πα-

```
1  interface Boot {
2    /**
3     * Signaled when the system has booted successfully. Components can
4     * assume the system has been initialized properly. Services may
5     * need to be started to work, however.
6     *
7     * @see StdControl
8     * @see SplitConrol
9     * @see TEP 107: Boot Sequence
10    */
11   event void booted();
12 }
```

Κώδικας 2.5: Δήλωση του interface Boot.



Σχήμα 2.1: Το διάγραμμα της εφαρμογής BlinkAppC

ρέχεται, τότε το συνδέει με ένα component που το παρέχει) . Στο σχήμα 2.1, φαίνεται το διάγραμμα της εφαρμογής Blink,όπως αυτό προκύπτει από το configuration αρχείο.

2.2 Συμβάσεις ονομασίας των components στο TinyOS

Για την ονομασία των components ακολουθείται η σύμβαση στο πίνακα 2.1. Σύμφωνα με την σύμβαση αυτή, όταν το όνομα του component τελειώνει στο γράμμα `C`, τότε δηλώνει ότι αυτό είναι component και έχει νόημα να χρησιμοποιηθεί εκτός της βιβλιοθήκης ή της εφαρμογής που βρίσκεται, αντιθέτως όταν τελειώνει στο γράμμα `P` τότε δηλώνει ότι αυτό είναι μέρος βιβλιοθήκης και δεν έχει νόημα να χρησιμοποιηθεί εκτός. Όταν το όνομα του component τελειώνει σε `AppC` , τότε δηλώνει ότι αυτό το component είναι το configuration component υψηλότερου επίπεδου της εφαρμογής.

Foo.nc	Interface
FooC.nc	Public Component
FooP.nc	Private Component
FooAppC.nc	Application top-level Component
Foo.h	Header file

Πίνακας 2.1: Σύμβαση ονομασίας components.

2.3 Μεταβλητές και Σταθερές

Στην NesC , η δήλωση και η χρήση των μεταβλητών και των σταθερών γίνεται όπως και στην γλώσσα C . Υπάρχουν όμως και μερικοί τύποι μεταβλητών επιπλέον.

2.3.1 Τύποι μεταβλητών

Εκτός από τους τύπους μεταβλητών της C (`int, long, char` κλπ),υπάρχουν και επιπλέον τύποι μεταβλητών με αυστηρά καθορισμένο μέγεθος σε bytes . Οι τύποι αυτοί φαίνονται στο πίνακα 2.2 και αντιστοιχούν στους `unsigned int, int` και `long` της C . Στους τύπους μεταβλητών της C , το μέγεθος τους (σε bytes) αλλάζει από πλατφόρμα σε πλατφόρμα, γι΄ αυτό είναι προτιμότερο να γίνεται χρήση των αντίστοιχων από τον πίνακα 2.2. Επιπλέον σε αυτούς τους τύπους, υπάρχει και ο τύπος `bool` , ο οποίος παίρνει τιμές `TRUE,FALSE` . Οι περισσότερες πλατφόρμες υποστηρίζουν και τους τύπους μεταβλητών `float` και μερικές από αυτές και `double` .

2.3.2 Τύποι μεταβλητών δικτύου (network types)

Στην NesC , ορίζονται και τύποι μεταβλητών για εξωτερική χρήση (network types). Αυτοί οι τύποι παίρνουν ένα πρόθεμα `nx_` ή `nxle_` και ορίζονται έτσι για να έχουν την ίδια μεταχείριση απο πλατφόρμα σε πλατφόρμα στο ίδιο δίκτυο. Συνήθως με αυτούς τους τύπους δηλώνονται οι μεταβλητές μέσα στην δομή των μηνυμάτων. Το πρόθεμα `nx_` σημαίνει ότι ο τύπος αυτός είναι σε big endian μορφή, ενώ το `nxle_` σημαίνει ότι είναι σε little endian μορφή. Σε αντιστοιχία με το `struct`, ορίζεται και το `nx_struct` και `nxle_struct` για δομές εξωτερικού τύπου. Η ανάθεση τιμών σε μεταβλητές εξωτερικού τύπου, δεν διαφέρει από αυτήν εσωτερικού τύπου.

	8-bit	16-bit	32-bit	64-bit
signed	`int8_t`	`int16_t`	`int32_t`	`int64_t`
unsigned	`uint8_t`	`uint16_t`	`uint32_t`	`uint64_t`
nx signed	`nx_int8_t`	`nx_int16_t`	`nx_int32_t`	`nx_int64_t`
nx unsigned	`nx_uint8_t`	`nx_uint16_t`	`nx_uint32_t`	`nx_uint64_t`
nxle signed	`nxle_int8_t`	`nxle_int16_t`	`nxle_int32_t`	`nxle_int64_t`
nxle unsigned	`nxle_uint8_t`	`nxle_uint16_t`	`nxle_uint32_t`	`nxle_uint64_t`

Πίνακας 2.2: Επιπρόσθετοι τύποι μεταβλητών NesC

2.3.3 Σταθερές

Για την δήλωση σταθερών, ισχύει ό,τι ισχύει και στην γλώσσα C . Παρόλα αυτά όμως, είναι προτιμότερο οι σταθερές να δηλώνονται μέσα σε μια δήλωση `enum` . Η χρήση του `enum` για τον ορισμό των σταθερών, προτιμάται από την χρήση των `#define` διότι έτσι δεν αντικαθίσταται οποιαδήποτε σταθερά είναι δηλωμένη με το ίδιο όνομα ανεξαρτήτως από το που εμφανίζεται στον κώδικα, αλλά μόνο αυτές που είναι στην εμβέλεια της δήλωσης του `enum` .

```
1  #include "myHeader.h"
2
3  module myModuleC {
4      /* used and provided interfaces list*/
5  }
6  implementation {
7      enum{
8          MYCONSTANT=100
9      };
10     /* source code that may use the constant MYCONSTANT*/
11 }
```

Κώδικας 2.6: Παράδειγμα για εμβέλεια σταθεράς

Στο παράδειγμα 2.6, η σταθερά `MYCONSTANT` έχει εμβέλεια μόνο μέσα στο module . Όποιες σταθερές είναι δηλωμένες μέσα σε `enum` στο header file `myHeader.h` , έχουν εμβέλεια μόνο σε όποια components γίνεται `#include` το συγκεκριμένο header file .

2.4 Generic components και generic interfaces

Στο TinyOS μπορούν να δηλωθούν generic components και generic interfaces . Ένα τέτοιο component είναι και το `TimerMilliC` που είδαμε στο παράδειγμα του Blink (2.1), ενώ ένα generic interface είναι το `Timer` (2.3). Τα generic components επιτρέπουν σε πολλά άλλα components να χρησιμοποιούν ανεξάρτητα αντίγραφα της ίδια υλοποίησης, σε αντίθεση με τα απλά components που

δεν υπάρχουν σε παραπάνω από ένα στιγμιότυπο σε όλο το πρόγραμμα (singletons). Στο παράδειγμα του Blink (2.1), το `component BlinkC` χρησιμοποιεί τρία ανεξάρτητα αντίγραφα της ίδια υλοποίησης του `TimerMilliC` . Επιπλέον, τα generic components επιτρέπουν και την παραμετροποίηση μία υλοποίησης. Ένα παράδειγμα παραμετροποιήσιμου component είναι το `AMSenderC` (2.7),που χρησιμοποιείται για να την αποστολή μηνυμάτων. Τα generic interfaces χρησιμοποιούνται για την δήλωση παραμετροποιήσιμων interfaces . Το `interface Timer` παίρνει σαν παράμετρο τον τύπο ακρίβειας, για παράδειγμα `TMilli` για τα milliseconds . Τα generic components χρησιμοποιούνται με την λέξη κλειδί `new`, και με χρήση παρενθέσεων μετά το όνομα. Μέσα στις παρενθέσεις πρέπει να ορίζονται οι παράμετροι που δέχεται με την ίδια σειρά που είναι δηλωμένες. Για παράδειγμα, για να χρησιμοποιηθεί ένα στιγμιότυπο του AMSenderC,θα πρέπει να δηλωθεί στο configuration component με την γραμμή :

```
components new AMSenderC(AM_MSGTYPE);
```

```
1  #include "AM.h"
2
3  generic configuration AMSenderC(am_id_t AMId) {
4    provides {
5      interface AMSend;
6      interface Packet;
7      interface AMPacket;
8      interface PacketAcknowledgements as Acks;
9    }
10 }
11
12 implementation {
13
14 #if defined(LOW_POWER_LISTENING)
15   components new LplAMSenderC(AMId) as SenderC;
16 #else
17   components new DirectAMSenderC(AMId) as SenderC;
18 #endif
19
20   AMSend = SenderC;
21   Packet = SenderC;
22   AMPacket = SenderC;
23   Acks = SenderC;
24 }
```

Κώδικας 2.7: Το AMSenderC component.

2.5 Πολλαπλή σύνδεση σε interfaces, unique() και uniqueCount()

Κάθε interface ενός component μπορεί να συνδέεται σε παραπάνω από ένα άλλα interfaces ίδιου τύπου. Μία χρήση της πολλαπλής σύνδεσης ενός interface γίνεται από το component MainC (2.8). Όσα components πρέπει να αρχικοποιήθούν πριν ξεκινήσει η εκτέλεση της εφαρμογής, πρέπει να παρέχουν το `interface Init` (2.9), το οποίο χρησιμοποιεί το component MainC , ενώ όσα πρέπει να ξεκινήσουν ταυτόχρονα με την εφαρμογή κάνουν χρήση του `interface Boot`, το οποίο παρέχει το MainC . Όταν η MainC , κάνει κλήση του `command SoftwareInit.init()` , τότε όλα τα components που είναι συνδεδεμένα μέσω αυτού του interface στο MainC , θα εκτελέσουν τον κώδικα που είναι μέσα στο `command Init.init()` της υλοποίησής τους. Αντίστοιχα και με το `interface Boot` , όταν η MainC Μέτα την αρχικοποίηση του συστήματος κάνει `signal Boot.booted()` , τότε όλα τα components που χρησιμοποιούν το `interface Boot` , θα εκτελέσουν τον κώδικα που έχουν στην υλοποίηση του `event Boot.booted()` . Αυτό επιτρέπει σε μία υλοποίηση να είναι ανεξάρτητη από τον αριθμό των components από τα οποία εξαρτάται
Επιπλέον από αυτό, στο TinyOS υπάρχει μηχανισμός, ένα component -πάροχος να παρέχει αποδοτικότερα ένα interface σε πολλαπλά components -πελάτες, από ότι την πολλαπλή σύνδεση Το κάθε component -πελάτης συνδέεται στο component -πάροχο με ένα id . Αυτό μπορούμε να το φανταστούμε σαν ένα πίνακα από ίδιου τύπου interfaces με την ίδια υλοποίησή ή με υλοποίηση που να διαφοροποιείται με το id στην πλευρά του component -παροχου. Ένα τέτοιο παράδειγμα εφαρμογής αυτού του μηχανισμού είναι στο `component ActiveMessageC` (2.10) και ένα παράδειγμα component -πελάτη, είναι στο `AMReceiverC` component 2.12. Παρατηρούμε στο `AMReceiverC` component , ότι το `Receive` interface που παρέχει συνδέεται στο `Receive` interface του `ActiveMessageC` με `id=amId` . Με αυτόν τον μηχανισμό το component -παροχος, μπορεί να ξεχωρίσει ποιο component εξυπηρετεί και σε ποιο θα κάνει signal event , χωρίς να γίνονται signal ταυτόχρονα όλα τα events των components που είναι συνδεδεμένα στο ίδιο interface .
Κάνοντας χρήση αυτού του μηχανισμού, είναι απαραίτητο να υπάρχει μια `default` υλοποίηση των αντίστοιχων events απο την πλευρά του component -παρόχου. Αυτό είναι απαραίτητο, διότι τα `id` δεν απαιτείται να είναι συνεχόμενοι αριθμοί. Μία `default` υλοποίηση ενός event δηλώνεται με την λέξη κλειδί `default` . Μία κενή `default` υλοποίηση του event receive , δίνεται στο κώδικα 2.11. Συμπληρωματικά σε αυτόν τον μηχανισμό, υπάρχουν και δύο ειδικές συναρτήσεις του μεταγλωττιστή της NesC. Οι συναρτήσεις αυτές είναι οι `unique()` και `uniqueCount()` . Και οι δύο αυτές συναρτήσεις παίρνουν σαν όρισμα

μια συμβολοσειρά-κλειδί. Για αυτήν την συμβολοσειρά-κλειδί, η `unique()` επιστρέφει ένα μοναδικό `id` , ενώ η `uniqueCount()` επιστρέφει το πλήθος των μοναδικών id που έχει δώσει για αυτήν

```
 */

#include "hardware.h"

configuration MainC {
  provides interface Boot;
  uses interface Init as SoftwareInit;
}
implementation {
  components PlatformC, RealMainP, TinySchedulerC;

#ifdef SAFE_TINYOS
  components SafeFailureHandlerC;
#endif

  RealMainP.Scheduler -> TinySchedulerC;
  RealMainP.PlatformInit -> PlatformC;

  // Export the SoftwareInit and Booted for applications
  SoftwareInit = RealMainP.SoftwareInit;
  Boot = RealMainP;
}
```

Κώδικας 2.8: Το MainC component.

```

  /**
   * Initialize this component. Initialization should not assume that
   * any component is running: init() cannot call any commands besides
   * those that initialize other components.
   *
   * @return SUCCESS if initialized properly, FAIL otherwise.
   * @see TEP 107: Boot Sequence
   *
   */
  command error_t init();
}
```

Κώδικας 2.9: Το Init interface.

```
configuration ActiveMessageC {
    provides{
        interface Init;
        interface SplitControl;

        interface AMSend[uint8_t id];
        interface Receive[uint8_t id];
        interface Receive as Snoop [uint8_t id];
        /* rest interfaces provided */
    }
}
implementation {
    /* wiring*/
}
```

Κώδικας 2.10: Πολλαπλή σύνδεση interface με παράμετρο στο ActiveMessageC component

```
  default event message_t *
  Receive.receive[uint8_t id](message_t *msg, void *payload,uint8_t len) {
    return msg;
  }
```

Κώδικας 2.11: default υλοποίηση του event Receive.

```
#include "AM.h"

generic configuration AMReceiverC(am_id_t amId) {
  provides {
    interface Receive;
    interface Packet;
    interface AMPacket;
  }
}

implementation {
  components ActiveMessageC;

  Receive = ActiveMessageC.Receive[amId];
  Packet = ActiveMessageC;
  AMPacket = ActiveMessageC;
}
```

Κώδικας 2.12: Το AMReceiverC component.

2.6 Συναρτήσεις και Βιβλιοθήκες

Μέσα στο `implementation` κομμάτι ενός module, μπορούν να δηλωθούν, υλοποιηθούν και χρησιμοποιηθούν συναρτήσεις με τον ίδιο τρόπο που γίνεται στην γλώσσα C.Επιπλέον στο TinyOS source tree , υπάρχουν βιβλιοθήκες για πρωτόκολλα δικτύου, συγχρονισμό, link estimation , επικοινωνία με την σειριακή θύρα και αποστολή μηνυμάτων printf μέσω σειριακής. Οι υλοποιήσεις αυτών των βιβλιοθηκών υπάρχουν στον φάκελο `$TOSDIR/lib`.

2.7 Μοντέλο εκτέλεσης TinyOS

Το μοντέλο εκτέλεσης του TinyOS , βασίζεται σε εργασίες δύο φάσεων, tasks που εκτελούνται χωρίς διακοπές από άλλα, και σε διαχειριστές interrupts. Ο κώδικας της nesC διαχωρίζεται σε *α'*) σύγχρονο κώδικα, που μπορεί να εκτελεστεί μόνο μέσα από tasks και σε *β'*) ασύγχρονο κώδικα, που μπορεί να εκτελεστεί και από tasks και από διαχειριστές interrupts. . Ο ασύγχρονος κώδικας πρέπει να δηλώνεται από την λέξη κλειδί `async` και στα interfaces και στα module που υλοποιείται. Σε αυτήν την ενότητα θα περιγράφουν η ακολουθία εκκίνησης του αισθητήρα, οι εργασίες δύο φάσεων και τα tasks.

2.7.1 Ακολουθία εκκίνησης αισθητήρα

Ο μοναδικός τρόπος να γνωρίζει μια εφαρμογή πότε έχει γίνει εκκίνηση του αισθητήρα είναι μέσω του `event Boot.booted()`. Χρησιμοποιώντας αυτό το event θα πρέπει να αρχικοποιείται και να ξεκινάει η εκτέλεση της. Η διαδικασία εκκίνησης του TinyOS υλοποιείται μέσα στο component MainCκαι αποτελείται από τέσσερα βήματα :

1. Αρχικοποίηση χρονοπρογραμματιστή (Scheduler)
2. Αρχικοποίηση των components
3. Γίνεται `signal` το `Boot.booted()` event
4. Ξεκινάει η εκτέλεση του χρονοπρογραμματιστή (Scheduler).

Κατά την αρχικοποίηση του χρονοπρογραμματιστή, αρχικοποιείται η ουρά εκτέλεσης των tasks. Ακολουθεί η αρχικοποίηση των components που παρέχουν το `interface Init` και είναι συνδεδεμένα στο `interface SoftwareInit` του `component MainC`, και έπειτα γίνεται `signal` το `event Boot.booted()` , για να ξεκινήσει η εκτέλεση της εφαρμογής. Στο τέταρτο βήμα και τελευταίο βήμα

ξεκινάει η εκτέλεση του χρονοπρογραμματιστή. Στην πράξη, εφόσον ο χρονοπρογραμματιστής δεν εκτελούνται κατά την διάρκεια των τριών πρώτων βημάτων, δεν μπορούσε να εκτελεστεί οποιοδήποτε `task`.Η εκτέλεση όλων των tasks που τυχόν είχαν γίνει `post` κατά την διάρκεια της αρχικοποίησης των εφαρμογών, ξεκινάει μετά την έναρξη εκτέλεσης του χρονοπρογραμματιστή.

2.7.2 Tasks

Ο κώδικας του κάθε command και event εκτελείται σύγχρονα, εκτός αν δηλώνεται με την λέξη κλειδί `async`,και σε μία εκτέλεση (δεν υπάρχει preemption). Δηλαδή όταν ένα κομμάτι σύγχρονου κώδικα ξεκινήσει να εκτελείται δεν θα διακοπεί από άλλο κομμάτι σύγχρονου κώδικα μέχρι να ολοκληρωθεί η εκτέλεση του. Με αυτόν τον τρόπο ο χρονοπρογραμματιστής (scheduler) του TinyOS ελαχιστοποιεί την χρήση μνήμης RAM και διατηρείται ο σύγχρονος κώδικας απλός. Αυτό βέβαια, σημαίνει ότι όταν ένα κομμάτι εκτελείται για πολλή ώρα, αποτρέπει την εκτέλεση άλλων και κατ΄ επέκταση επηρεάζει την ανταπόκριση του συστήματος. Για παράδειγμα, τέτοιο κομμάτι, μπορεί αν αυξήσει τον χρόνο ανταπόκρισης του αισθητήρα στην λήψη ενός πακέτου. Τα περισσότερα components του συστήματος κάνουν `signal` ένα event σε ένα άλλο component το οποίο με την σειρά του καλεί ένα άλλο command και επιστρέφει. Αυτή η προσέγγιση δουλεύει καλά όταν πρόκειται για εργασίες που γίνονται γρήγορα. Επειδή ο σύγχρονος κώδικας εκτελείται χωρίς διακοπές, αυτή η προσέγγιση δεν θα δουλέψει καλά για εργασίες που απαιτούν χρόνο. Για αυτόν τον λόγο, το TinyOS παρέχει έναν μηχανισμό για την διάσπαση των μεγάλων εργασιών σε μικρότερα κομμάτια. Αυτός ο μηχανισμός ονομάζεται Tasks . Ένα task αποτελεί μία συνάρτηση και όταν γίνεται post , εισάγεται στην ουρά εκτέλεση του χρονοπρογραμματιστή του TinyOS και θα εκτελεστεί κάποια στιγμή στο μέλλον , όχι αμέσως.
Η δήλωση ενός task γίνεται με την ακόλουθη σύνταξη :

```
task void taskname() { /* task implementation code */}
```

Ένα task δεν μπορεί να πάρει καμία παράμετρο και δεν μπορεί να επιστρέψει κάποια τιμή. Για να κάνουμε post ένα task , χρησιμοποιούμε την σύνταξη :

```
post taskname();
```

Κάθε component μπορεί να κάνει post ένα task μέσα από ένα command , event, ή ακόμα και από ένα task . Επιπλέον, ένα task μπορεί να κάνει post τον εαυτό του, άλλα tasks ή να καλεί commands και να κάνει signal events. Μια βασική χρήση των tasks , είναι για να κάνουν signal ένα event.Κατά σύμβαση, αποφεύγετε μέσα στο σώμα ενός command να γίνεται signal event , για

να αποφεύγετε η επαναληπτικής εκτέλεση του ίδιου command ή event μέσω της αναδρομής. Όταν γίνεται post ένα task τότε αυτό εισάγεται σε μία ουρά εκτέλεσης tasks , η οποία επεξεργάζεται με την λογική FIFO. Όταν εκτελείται ενα task , πρέπει να τελειώσει για ξεκινήσει η εκτέλεση του επόμενου. Ένα task , μπορεί να διακοπεί από hardware interrupts . Η διαδικασία του post για ενα task μπορεί να αποτύχει αν είχε ήδη εισαχθεί στην ουρά εκτέλεσης πιο πριν, αλλά δεν εχει εκτελεστεί, ούτε έχει ξεκινήσει να εκτελείται, ακόμα.

2.7.3 Εργασίες δύο φάσεων (Split-phase operations)

Στο TinyOS , δεν υπάρχουν διαδικασίες που μπλοκάρουν μέχρι να τελειώσουν. Αντί για αυτές, κάθε διαδικασία που απαιτεί αρκετό χρόνο για να ολοκληρωθεί, γίνεται σε δυο φάσεις. Η έναρξη της πρώτης φάσης ξεκινάει με ένα command , το οποίο επιστρέφει αμέσως Μέτα τις απαραίτητες αρχικοποιήσεις και τα post των tasks που απαιτούνται, και όταν ολοκληρωθεί η διαδικασία κάνει signal το αντίστοιχο event (δεύτερη φάση). Αυτός ο τρόπος προσφέρει αρκετά πλεονεκτήματα:

- αποφυγή του να σταματήσει η εκτέλεση, να μπλοκάρουν όλα τα νήματα, σε διαδικασίες που μπλοκάρουν, διότι δεν υπάρχουν τέτοιες.
- επιτρέπει στην ταυτόχρονη έναρξη πολλών διαδικασιών δύο φάσεων 'ταυτόχρονα'
- χρήση λιγότερης μνήμης, διότι όταν ένα πρόγραμμα καλεί μια διαδικασία μπλοκαρίσματος, τότε όλη η κατάσταση που αποθηκεύετε στην στοίβα κλήσεων, πρέπει να αποθηκευτεί κάπου. Με αυτόν τον τρόπο δεν είναι ανάγκη να διατηρούνται στην στοίβα. Παρόλα αυτά, όταν υπάρχουν δεδομένα που πρέπει να διατηρηθούν μεταξύ των δύο φάσεων, αυτά πρέπει να αποθηκευτούν κάπου (συνήθως σαν μεταβλητές με εμβέλεια module).

Μερικές από τις πιο σημαντικές διαδικασίες δύο φάσεων, είναι η χρήση του `interface Timer` και η αποστολή μηνύματος.

Τα interfaces SplitControl και StdControl .

Στο TinyOS , χρησιμοποιούνται δυο interfaces για την αρχικοποίηση components και λειτουργιών. Αυτά τα interfaces είναι τα α') SplitControl (2.13) , για αρχικοποίηση δύο φάσεων και β') StdControl (2.14), για αρχικοποίηση μίας φάσης . Στο `interface SplitControl` , δηλώνονται δύο commands , α') το `start()` για την έναρξη λειτουργίας, και β') το `stop()` για την παύση λειτουργίας . Το καθένα από αυτά τα commands αρχικοποιεί την διαδικασία

και επιστρέφει. Όταν η διαδικασία (παύσης ή έναρξης) ολοκληρωθεί γίνεται `signal` το event *α΄*) `startDone()`,σε αντιστοιχία του `start()` command και το *β΄*) `stopDone()` , σε αντιστοιχία του `stop()` command . Στο `interface StdControl` , δηλώνονται μόνο τα δύο commands , *α΄*) το `start()` για την έναρξη λειτουργίας, και *β΄*) το `stop()` για την παύση λειτουργίας .

```
1  interface SplitControl
2  {
3    command error_t start();
4
5    event void startDone(error_t error);
6
7    command error_t stop();
8
9    event void stopDone(error_t error);
10 }
```

Κώδικας 2.13: Το SplitControl interface.

```
1  interface StdControl
2  {
3    command error_t start();
4    command error_t stop();
5  }
```

Κώδικας 2.14: Το StdControl interface.

2.7.4 Ατομικότητα

Η nesC παρέχει ένα τρόπο εκτέλεση ομάδας εντολών ατομικά. Αυτό είναι εφικτό με την χρήση την λέξης- κλειδί `atomic`. Η σύνταξη είναι η ακόλουθη:

```
atomic{ /* atomically executed statements */}
```

Με αυτόν τον τρόπο εξασφαλίζουμε ότι οι μεταβλητές, μέσα σε αυτήν την ομάδα εντολών, θα διαβαστούν και θα γραφτούν ατομικά. Αυτός ο τρόπος χρησιμοποιείται κυρίως στην υλοποίηση ασύγχρονου κώδικα.

2.8 Ασύρματη Επικοινωνία

Το TinyOS, παρέχει και έναν μηχανισμό αφαίρεσης για τις υπηρεσίες ασύρματης επικοινωνίας που λειτουργεί με τον ίδιο τρόπο από πλατφόρμα σε πλατφόρμα. Σε αυτήν την ενότητα θα περιγράφουν συνοπτικά τα interfaces και τα components του μηχανισμού αυτού.

2.8.1 Δομή του μηνύματος

Όλα τα components και interfaces του μηχανισμού αυτού χρησιμοποιούν μία κοινή δομή μηνύματος, το `message_t` (2.15). Η δομή του μηνύματος ορίζεται στο αρχείο `$TOSDIR/types/message.h`. Το κάθε πεδίο της δομής του μη-

```
1   typedef nx_struct message_t {
2       nx_uint8_t header[sizeof(message_header_t)];
3       nx_uint8_t data[TOSH_DATA_LENGTH];
4       nx_uint8_t footer[sizeof(message_footer_t)];
5       nx_uint8_t metadata[sizeof(message_metadata_t)];
6   } message_t;
```

Κώδικας 2.15: Δομή μηνύματος, message_t

νύματος, πρέπει να αλλάζει μόνο μέσω των interfaces `Packet` και `AMPacket`.Το αξιοσημείωτο με αυτήν την προσέγγιση είναι ότι το μέγεθος των δεδομένων, payload , παραμένει σταθερό μεταξύ των επιπέδων ζεύξης. Κάθε εφαρμογή τοποθετεί τα δεδομένα που θέλει να στείλει μέσα στο πεδίο `data`.

Την δομή που θα έχουν τα δεδομένα μέσα στο πεδίο `data` την καθορίζει η κάθε εφαρμογή. Αυτή τη δομή δηλώνεται με ένα `nx_struct` ή ένα `nxle_struct` . Όπως αναφέραμε και στην ενότητα 2.3, για τις δομές των μηνυμάτων χρησιμοποιούνται μεταβλητές εξωτερικού τύπου, συνεπώς όλα τα πεδία της δομής δεδομένων του μηνύματος πρέπει να είναι μεταβλητές εξωτερικού τύπου. Ένα παράδειγμα δήλωσης δομής δεδομένων μηνύματος είναι το ακόλουθο:

```
1          typedef nx_struct BlinkToRadioMsg {
2                nx_uint16_t nodeid;
3                nx_uint16_t counter;
4          } BlinkToRadioMsg;
```

2.8.2 interfaces και components

Τα interfaces τα οποία χρησιμοποιούνται για την ασύρματη επικοινωνία είναι τα ακόλουθα:

Packet A.7): Παρέχει τα απαραίτητα commands για πρόσβαση στα πεδία της δομής `message_t`.

Send A.9): Παρέχει την βασική λειτουργία αποστολής μηνύματος χωρίς-διεύθυνση. Επίσης, παρέχει και ένα event για την ολοκλήρωση αποστολής (επιτυχημένα ή όχι) του μηνύματος.

Receive A.8): Παρέχει τις βασικές λειτουργίες για την λήψη του μηνύματος Δηλαδή, ένα event receive για την λήψη, καθώς επίσης και commands για πρόσβαση στο payload του μηνύματος μέσω ενός pointer.

PacketAcknowledgements A.10): Παρέχει έναν μηχανισμό για την ζήτηση Acknowledgment για κάθε πακέτο.

RadioTimeStamping A.11): Παρέχει πληροφορίες χρόνου για την αποστολή και λήψη του μηνύματος.

AMPacket A.12): Παρέχει τα βασικά commands για την πρόσβαση πεδίων της δομής του μηνύματος που έχουν σχέση με τον μηχανισμό των Active Messages.Τέτοια πεδία είναι η διεύθυνση του αισθητήρα, η διεύθυνση του αποστολέα και του παραλήπτη, και ο τύπος του μηνύματος.

AMSend A.13): Παρέχει την βασική εντολή αποστολή μηνύματος μέσω του μηχανισμού ActiveMessage.Η κύριά διαφορά από το `Send` interface είναι ότι στο `AMSend` interface , το command `send` παίρνει όρισμα την διεύθυνση του παραλήπτη.

ActiveMessageAddress A.14): Παρέχει ένα μηχανισμό αλλαγής της διεύθυνσης του αισθητήρα. Η διεύθυνση του αισθητήρα καθορίζεται αρχικά κατά την εγκατάσταση της εφαρμογής.

Τα αντίστοιχα components που παρέχουν τα παραπάνω interfaces είναι τα ακόλουθα:

AMReceiverC 2.12): Παρέχει τα interfaces `Receive`, `Packet`, `AMPacket` που είναι απαραίτητα για την λήψη μηνυμάτων

AMSenderC 2.7): Παρέχει τα interfaces `AMSend`, `AMPacket`, `Packet`, `PacketAcknowledgements` για την αποστολή μηνυμάτων

AMSnooperC : Παρέχει τα interfaces `Receive`, `Packet`, `AMPacket` για την λήψη μηνυμάτων που δεν προορίζονται για αυτόν τον αισθητήρα.

AMSnoopingReceiverC : Παρέχει τα interfaces `Receive`, `Packet`, `AMPacket` για την λήψη μηνυμάτων ανεξάρτητα αν αυτά προορίζονται για αυτόν τον αισθητήρα.

ActiveMessageAddressC : Παρέχει το `interface ActiveMessageAddress` για την αλλαγή διεύθυνσης του αισθητήρα.

ActiveMessageC : Αυτό το component παρέχει τον μηχανισμό αφαίρεσης για τις διάφορες πλατφόρμες. Αυτό το component χρησιμοποιούν τα υπόλοιπα για να παρέχουν τα interfaces που παρέχουν. Αυτό το component παρέχει το interface `SplitControl` για την έναρξη και παύση λειτουργίας της ασύρματης επικοινωνίας (και των αντίστοιχων μονάδων στο hardware)

2.8.3 Αποστολή-Λήψη μηνύματος

Για την αποστολή μηνύματος, ένα component χρησιμοποιεί τα interfaces `Packet`, `AMPacket` και `AMSend`. Αυτά τα interfaces παρέχονται από τα στιγμιότυπα του component `AMSenderC` . Για την λήψη μηνύματος, ένα component χρησιμοποιεί τα interfaces `Packet, AMPacket` και `Receive`. Αυτά τα interfaces παρέχονται από τα στιγμιότυπα του component `AMReceiverC` .

Επιπλέον,η εφαρμογή θα πρέπει να ορίζει κάπου και την δομή δεδομένων του κάθε μηνύματος που θα στέλνεται, καθώς επίσης και ένα Active Message type . Η δήλωση του Active Message type , είναι απαραίτητη από τον μηχανισμό των Active Messages για τον διαχωρισμό των μηνυμάτων από component σε component. Δηλαδή, αποτελεί ένα είδος πολυπλεξίας για τα μηνύματα που στέλνονται, ώστε να υπάρχει δίκαιη εξυπηρέτηση μεταξύ των components της εφαρμογής (και αυτών των βιβλιοθηκών που χρησιμοποιεί) και να παρέχεται και ένα επίπεδο αφαίρεσης για την κοινή χρήση του hardware για την αποστολή/λήψη μηνυμάτων. Τα μηνύματα που στέλνονται μέσω του component `AMSenderC` με συγκεκριμένο Active Message type , λαμβάνονται μέσω του component `AMReceiverC` που χρησιμοποιεί το ίδιο Active Message type . Η δήλωση του Active Message type καθώς επίσης και των στιγμιότυπων του component `AMSenderC` γίνεται με τον ακόλουθο τρόπο:

```
1       /* Active Message type declaration*/
2       enum{
3           AM_MYAPPMSG=12
4       };
5       /* AMReceiverC component instance */
6       components new AMReceiverC(AM_MYAPPMSG);
7       /* wiring of AMReceiverC with user-component*/
8       /* AMSenderC component instance*/
9       components new AMSenderC(AM_MYAPPMSG);
10      /* wiring of AMSenderC with user-component */
```

Η αποστολή/λήψη μηνυμάτων απαιτεί και την χρήση του απαραίτητου hardware . Για την αρχικοποίηση και το σταμάτημα όλου του μηχανισμού αποστολής μηνυμάτων καθώς επίσης και του hardware απαιτείται η χρήση του `interface SplitControl` το οποίο παρέχεται από το component `ActiveMessageC`. Η αρχικοποίηση ξεκινάει με το `command start()` και ολοκληρώνεται με το `event startDone()` . Αντίστοιχα και σταμάτημα ξεκινάει με το `command stop()` και ολοκληρώνεται με το `event stopDone()`.

Για την αποστολή μηνύματος, χρειάζεται η δήλωση δύο μεταβλητών με εμβέλεια module. Η μία μεταβλητή θα είναι τύπου `message_t` για το μήνυμα που θα χρησιμοποιείται σαν buffer μεταξύ της εφαρμογής και του επιπέδου ζεύξης, και η άλλη μεταβλητή θα είναι τύπου `bool` και θα γίνεται αληθής όσο ο buffer χρησιμοποιείται από το επίπεδο ζεύξης Ένα κομμάτι κώδικα για την αποστολή μηνύματος είναι το ακόλουθο:

```
1       /* declaration of message data type*/
2       typedef nx_struct myAppMsg{
3           nx_uint16_t data;
4       }myAppMsg_t;
5
6       /* declaration of variables */
7       message_t pkt;
8       bool pktBusy;
9       /* task for sending pkt */
10      task void msgSendTask(){
11          /*. . . */
12          if(!pktBusy){
13              myAppMsg_t* m = (myAppMsg_t*)
14              (call Packet.getPayload(&pkt,sizeof(myAppMsg_t)));
15              /* setting message data*/
16              m->data=TOS_NODE_ID; // setting data as node's id
17                  if(call AMSend.send(AM_BROADCAST_ADDR, &pkt,sizeof(myAppMsg_t))==SUCCESS)
18                  {
19                      pktBusy = TRUE;
20                  }
21            }
22      }
23
24      /* sendDone event */
25      event void AMSend.sendDone(message_t* msg, error_t error) {
26          if (&pkt == msg) {
27                pktBusy = FALSE;
28          }
29        }
```

Για την λήψη μηνύματος, απλά χρησιμοποιείται το event `receive`. Μέσα στην υλοποίηση του event `receive` πρέπει να αντιγράφεται το μήνυμα σε τοπική μεταβλητή για να χρησιμοποιηθεί αργότερα για τυχόν επεξεργασία. Το event `receive` επιστρέφει έναν `message_t` pointer . Η θέση που δείχνει ο pointer αυτός, θα χρησιμοποιηθεί από το επίπεδο ζεύξης για την λήψη του επόμενου μηνύματος, και επιπλέον αυτός ο pointer μπορεί να δείχνει σε διαφορετική θέση μνήμης από ότι το msg που έχει σαν όρισμα. Ο μηχανισμός αυτός δίνει την ευχέρεια στην εφαρμογή να χρησιμοποιήσει κάποιο είδος pool. Ένα παράδειγμα χρήσης είναι το ακόλουθο:

```
1       // declaration of local msg for more processing
2       message_t recPkt;
3
4       event message_t* Receive.receive(message_t* msg, void* payload, uint8_t len) {
5             if (len == sizeof(myAppMsg_t)) {
6           myAppMsg_t* m = (myAppMsg_t*)payload;
7           /* code for short decisions depending in msg fields*/
8           /* copy of message, using memcpy()
9           memcpy(&recPkt,msg,sizeof(message_t));
10            }
11            return msg;
12      }
```

Serial	Radio
SerialActiveMessageC	ActiveMessageC
SerialAMSenderC	AMSenderC
SerialAMReceiverC	AMReceiverC

Πίνακας 2.3: Αντιστοιχία components ασύρματης-σειριακής επικοινωνίας.

Πλατφόρμα	Ταχύτητα (baud rate)
telos	115200
telosb	115200
tmote	115200
micaz	57600
mica2	57600
iris	57600
mica2dot	57600
eyes	115200
intelmote2	115200

Πίνακας 2.4: Αντιστοιχία components ασύρματης- σειριακής επικοινωνίας.

2.9 Επικοινωνία μέσω σειριακής θύρας

Για την επικοινωνία της εφαρμογής μέσω της σειριακής θύρας, δεν διαφέρει η λογική και ο τρόπος χρήσης του μηχανισμού. Η βασική διαφορά βρίσκεται στα components τα οποία παρέχουν τα interfaces . Η αντιστοιχία με τα components για την ασύρματη επικοινωνία φαίνεται στο πίνακα 2.3.

2.9.1 Εργαλεία επικοινωνίας με τον Η/Υ

Το TinyOS παρέχει μια βιβλιοθήκη και μερικά εργαλεία, για την ανάπτυξη εφαρμογών Η/Υ, για την επικοινωνία του αισθητήρα που είναι συνδεδεμένος με τον υπολογιστή. Η βιβλιοθήκη που παρέχει, είναι για τις γλώσσες προγραμματισμού Python, Java και C.

Για την επικοινωνία του Η/Υ με τον αισθητήρα, απαιτείται ο καθορισμός της ταχύτητας του διαύλου (baud rate) και την σειριακής θύρας που είναι συνδεδεμένος ο αισθητήρας. Η ταχύτητα διαύλου εξαρτάται από την πλατφόρμα του αισθητήρα και διαφέρει από πλατφόρμα σε πλατφόρμα, ενώ η σειριακή θύρα από το λειτουργικό σύστημα του υπολογιστή. Ο πίνακας 2.4, δείχνει ενδεικτικά τις ταχύτητες για μερικές γνωστές πλατφόρμες που υποστηρίζει το TinyOS . Όλα τα εργαλεία για την επικοινωνία μέσω σειριακής θύρας, παίρνουν μία παράμετρο για τον καθορισμό της σειριακής θύρας και της ταχύτητα διαύλου Η παράμετρο

αυτή είναι της μορφής `-comm serial@<port>:<speed>` . Το εργαλείο Listen , τυπώνει όλα τα μηνύματα που θα λάβει από την σειριακή θύρα σε 16-bit πεδία. Η εκτέλεση του γίνεται με την εντολή

```
java net.tinyos.tools.Listen -comm serial@/dev/ttyUSB1:iris
```

. Το εργαλείο MsgReader , τυπώνει τα μηνύματα ενός συγκεκριμένου τύπου, αφού πρώτα κάνει την αντιστοίχηση στην μορφή δεδομένων του μηνύματος Η μορφή δεδομένων του μηνύματος του δίνεται σαν όρισμα. Η μορφή, μπορεί να εξαχθεί από τον κώδικα nesC μέσω του εργαλείου mig . Η εντολή για την εξαγωγή της μορφής του μηνύματος του παραδείγματος της προηγούμενης ενότητας είναι

```
mig java -target=null -java-classname=MyAppMsg myApp.h myAppMsg
  -o MyAppMsg.java
```

. Αφού δημιουργηθεί το .class αρχείο με την εντολή

```
javac MyAppMsg.java
```

, εκτελούμε το MsgReader με την εντολή

```
java net.tinyos.tools.MsgReader
-comm serial@/dev/ttyUSB1:iris MyAppMsg
```

Το εργαλείο SerialForwarder , προωθεί τα μηνύματα που λαμβάνει από την σειριακή θύρα σε socket . Δηλαδή δουλεύει σαν ενδιάμεσος server μηνυμάτων από και προς άλλες εφαρμογές του υπολογιστή. Η εντολή

```
java net.tinyos.sf.SerialForwarder -port <serverport>
 -comm serial@/dev/ttyUSB1:iris
```

ξεκινάει τον SerialForwarder να προωθεί μηνύματα από/προς την σειριακή ακούγοντας αιτήσεις στην port ¡serverport¿ του Η/Ψ. Εκτός από την σειριακή θύρα, μπορεί να δοθεί σαν όρισμα `-comm` στα υπόλοιπα εργαλεία του TinyOS , και ο SerialForwarder . Αυτό γίνεται με το όρισμα `-comm sf@HOST:PORT`.

2.10 Μετρήσεις αισθητηρίων

Η πιο σημαντική λειτουργία των αισθητήρων, είναι η συλλογή μετρήσεων από τα αισθητήρια που είναι συνδεδεμένα πάνω τους. Σε αυτήν την ενότητα περιγράφεται ο τρόπος με τον οποίο η εφαρμογή μπορεί να διαβάζει τις τιμές των αισθητηρίων. Δεν περιγράφεται ο τρόπος λειτουργία των ADC (Analog to Digital Converters) και δεν περιγράφεται η μεθοδολογία που ακολουθείται για την σύνδεση επιπλέον αισθητηρίου και ανάγνωση τιμών του, στον αισθητήρα. Για αυτά, ανατρέξτε στο βιβλίο [5].

2.10.1 interfaces και components

Για την ανάγνωση μετρήσεων από τα αισθητήρια, η εφαρμογή πρέπει να χρησιμοποιεί το `interface Read` (2.16). Η ανάγνωση τιμών του αισθητηρίου, γίνεται σε δύο φάσεις. Στην πρώτη φάση, η εφαρμογή καλεί το command `read()` και όταν ολοκληρωθεί η μέτρηση, εκτελείται το event `readDone()` επιστρέφονται και το αποτέλεσμα της μέτρησης στο όρισμα `val`. Τα components που παρέχουν το `interface Read` , διαφέρουν από αισθητήριο σε αισθητήριο. Για το sensorboard mda100cb του εργαστηρίου, το component για την μέτρηση θερμοκρασίας είναι το `TempC` , ενώ για την μέτρηση έντασης του φωτός, το `PhotoC` . Η υλοποίηση τους μέσα στο TinyOS source tree βρίσκεται στον φάκελο `$TOSDIR/sensorboards/mda100/`.

```
1  interface Read<val_t> {
2
3    command error_t read();
4
5    event void readDone( error_t result, val_t val );
6  }
```

Κώδικας 2.16: Το Read interface.

2.11 Χρήση μόνιμης μνήμης (Flash memory)

Το TinyOS παρέχει και έναν μηχανισμό αφαίρεσης για χρήση της μόνιμης μνήμης (flash memory) του αισθητήρα, ανεξαρτήτως του ολοκληρωμένου κυκλώματος που το παρέχει. Η μόνιμη μνήμη, διατηρεί τα δεδομένα που έχουν εγγραφεί επιτυχώς σε αυτήν, ακόμα και αν ο αισθήτήρας κλείσει από μπαταρία ή κάποιο άλλο λόγο Για επιπλέον ευκολία, παρέχει και τρεις κατηγορίες

χρήσης, α') BlockStorage, για γενική αποθήκευση μεγάλων αντικειμένων ή δομών, β') LogStorage, για αποθήκευση καταγραφών, όπως μετρήσεις, γ') ConfigStorage, για αποθήκευση ρυθμίσεων, όπως calibration data αισθητηρίων και παράμετροι εφαρμογών.

2.11.1 Δομή μνήμης, volumes

Το TinyOS χωρίζει το συνολικό μέγεθος της μνήμης σε ένα ή παραπάνω σταθερού μεγέθους τομείς (volumes). Οι τομείς αυτοί δηλώνονται σε ένα XML αρχείο, το οποίο λαμβάνεται υπόψιν από τον μεταγλωττιστή κατά την μεταγλώττιση της εφαρμογής. Αυτό το αρχείο περιέχει την περιγραφή του volume table και επιτρέπει στον προγραμματιστή να καθορίζει το όνομα, το μέγεθος και την διεύθυνση έναρξης για κάθε volume . Το μέγεθος για κάθε volume θα πρέπει να είναι πολλαπλάσιο του μονάδας διαγραφής (erase unit) του ολοκληρωμένου κυκλώματος μνήμης. Κάθε volume μπορεί να χρησιμοποιηθεί σαν έναν τύπο από τους τρεις που παρέχει το TinyOS. Ένα παράδειγμα αρχείου XML για την δήλωση του volume table φαίνεται στο 2.17.

```
1  <volume_table>
2    <volume name="LOGTEST" size="262144"/>
3    <volume name="CONFIGTEST" size="4608"/>
4  </volume_table>
```

Κώδικας 2.17: Παράδειγμα δήλωσης volume table.

Το αρχείο δήλωσης του volume table πρέπει να τοποθετείται μέσα στο κύριο φάκελο της εφαρμογής και πρέπει το όνομά του να ακολουθεί την εξής μορφή `volumes-CHIPNAME.xml`, όπου το CHIPNAME θα αντικαθίσταται από το όνομα του ολοκληρωμένου που χρησιμοποιεί η πλατφόρμα. Για κάθε διαφορετικό ολοκληρωμένο που χρησιμοποιείται στις πλατφόρμες που σκοπεύουμε να τρέχει η εφαρμογή μας, θα πρέπει να παρέχουμε και ένα ξεχωριστό αρχείο δήλωσης volume table. Για παράδειγμα, οι αισθητήρες Memsic Iris που διαθέτει το εργαστήριο, χρησιμοποιούν το ολοκληρωμένο at45db και κατ΄ επέκταση το αρχείο δήλωσης του volume table πρέπει να ονομάζεται `volumes-at45db.xml`.

2.11.2 interfaces και components

Για να χρησιμοποιήσει η εφαρμογή τον μηχανισμό για την μόνιμη μνήμη, θα πρέπει να χρησιμοποιήσει τα απαραίτητα interfaces που παρέχονται από τα αντίστοιχα components. Τα components που παρέχουν αυτά τα interfaces είναι τα ακόλουθα:

BlockStorageC : Παρέχει τα interfaces `Mount`, `BlockRead`,και `BlockWrite` για την διαχείριση των volumes τύπου BlockStorage.

LogStorageC : Παρέχει τα interfaces `LogRead` και `LogWrite` για την διαχείριση των volumes τύπου LogStorage.

ConfigStorageC : Παρέχει τα interfaces `ConfigStorage` και `Mount` για την διαχείριση των volumes τύπου ConfigStorage.

LogStorage

Παρέχει έναν αξιόπιστο τρόπο καταγραφής δεδομένων και μικρών αντικειμένων με ατομικότητα ανάμεσα στα components. Το αρχείο καταγραφής μπορεί να λειτουργεί γραμμικά , δηλαδή όταν γεμίσει σταματάει να γράφει, ή κυκλικά, δηλαδή όταν γεμίσει θα ξεκινήσει να διαγράφει τα παλαιότερα Κάθε κλήση του `command LogWrite.append()` δημιουργεί μια νέα εγγραφή. Σε περίπτωση κλεισίματος του αισθητήρα (πχ. λόγω μπαταρίας), θα έχουν χαθεί μόνο ολόκληρες εγγραφές από το τέλος του αρχείου. Το πρώτο βήμα για να χρησιμοποιήσουμε το LogStorage , είναι να δηλώσουμε την δομή της κάθε εγγραφής στο αρχείο. Ένα παράδειγμα τέτοιας δήλωσης είναι το ακόλουθο:

```
1       enum{
2           MAX_MEAS_LEN=2
3       };
4       typedef nx_struct logentry{
5           nx_uint8_t len;
6           nx_uint16_t Measurements[MAX_MEAS_LEN];
7       }logentry_t;
```

Ενα παράδειγμα για ανάκτηση εγγραφών από το αρχείο είναι το ακόλουθο:

```
1       logentry_t lentry;
2       bool entryBusy;
3       /* . . .*/
4       if( call LogRead.read(&lentry,sizeof(logentry_t))==SUCCESS)
5       {
6           entryBusy=TRUE;
7       }else{
8       // error handling
9       }
10
11      /*. . . */
12      event void LogRead.readDone(void* buf, storage_len_t len, error_t err) {
13             if ( (len == sizeof(logentry_t)) && (buf == &lentry) ) {
14                 /*succeeded read*/
15                 entryBusy=FALSE;
16            }
17              else {
18                 entryBusy=FALSE;
19                 //empty log or loss of sync
20                   if (call LogWrite.erase() != SUCCESS) {
```

```
21                          // Handle error.
22                  }
23                }
24        }
```

και ένα παράδειγμα εγγραφής στο αρχείο είναι το ακόλουθο:

```
1       logentry_t lentry;
2       bool entryBusy;
3       /* . . .*/
4       if(!entryBusy)
5       {
6           entryBusy=TRUE;
7           /* setting up lentry's fields */
8           if( call LogWrite.append(&lentry,sizeof(logentry_t))!=SUCCESS)
9           {
10          // error handling
11              entryBusy=FALSE;
12          }
13      }
14      /*. . . */
15       event void LogWrite.appendDone(void* buf, storage_len_t len,
16                                      bool recordsLost, error_t err) {
17          entryBusy = FALSE;
18        }
```

ConfigStorage

Το ConfigStorage χρησιμοποιείται κυρίως για την αποθήκευση ρυθμίσεων της εφαρμογής ή calibration data των αισθητηρίων. Αυτού του είδους τα δεδομένα, τυπικά έχουν τις παρακάτω ιδιότητες: α') μικρό μέγεθος, μερικές δεκάδες bytes,β') η τιμή τους δεν είναι απαραίτητο να είναι ίδια για κάθε αισθητήρα στο δίκτυο (πχ. εξαρτάται από το harware), γ') πρέπει να διατηρούνται μεταξύ των εκκινήσεων του αισθητήρα ή/και των επαναπρογραμματισμών του.

Για την χρήση του ConfigStorage , απαιτείται από το component πριν από την πρώτη χρήση του να έχει ολοκληρωθεί το Mount , μέσω της αντίστοιχης εντολής στο interface Mount.

Ένα παράδειγμα χρήσης του ConfigStorage, υπάρχει στο TinyOS source tree κάτω από τον φάκελο `$TOSROOT/apps/tutorials/BlinkConfig/`.

BlockStorage

Το BlockStorage χρησιμοποιείται κυρίως για αποθήκευση μεγάλων δομών δεδομένων που δεν μπορούν εύκολα να χωρέσουν στην RAM. Είναι χαμηλού επιπέδου μηχανισμός και απαιτεί προσοχή στον χειρισμό του. Σε γενικές γραμμές ακολουθείται ένα μοντέλο εγγραφής μίας φοράς, και για να εγγραφούν δεδομένα πάνω από τις παλιές εγγραφές απαιτεί πρώτα διαγραφή, η οποία είναι μια χρονικά ακριβή διαδικασία και συμβαίνει σε μεγάλη έκταση (256 Bytes – 64

KBytes) και μπορεί να γίνει σε περιορισμένο αριθμό φορών (πχ 10000–100000 φορές ανάλογα το ολοκληρωμένο). Όπως και ο μηχανισμός για το ConfigStorage , απαιτεί να έχει γίνει πρώτα Mount το volume.

2.12 Προσομοίωση στο TinyOS (TOSSIM)

Το TinyOS προσφέρει και έναν προσομοιωτή εφαρμογής. Τον προσομοιωτή τον ονομάζει TOSSIM. Το πρόγραμμα για την προσομοίωση της εφαρμογής δημιουργείται δίνοντας μια επιπλέον παράμετρο στο πρόγραμμα make για την μεταγλώττιση της εφαρμογής. Η μεταγλώττιση της εφαρμογής για τον προσομοιωτή γίνεται μόνο για την πλατφόρμα micaz , προς το παρόν. Η εντολή είναι η ακόλουθη:

```
make micaz sim
```

Επιπλέον, για να προσομοιωθεί η εφαρμογή, απαιτούνται να οριστούν από τον προγραμματιστή, η τοπολογία του δικτύου, η εξασθένιση του σήματος για κάθε ζεύξη και το μοντέλο θορύβου που θα υπάρχει κατά την αποστολή μηνυμάτων για κάθε αισθητήρα.

2.12.1 Προετοιμασία εφαρμογής

Για να μπορέσουμε να παρακολουθήσουμε την εκτέλεση της εφαρμογής, υπάρχει η δυνατότητα να τυπώνουμε μηνύματα μέσα από τον κώδικα της εφαρμογής. Οι συναρτήσεις αυτές αγνοούνται, όταν πρόκειται να μεταγλωττιστεί η εφαρμογή για να εγκατασταθεί σε αισθητήρα. Ο προσομοιωτής διαχωρίζει τα μηνύματα αυτά σε κανάλια. Το χαρακτηριστικό του κάθε καναλιού είναι μία συμβολοσειρά. Για κάθε διαφορετική συμβολοσειρά, δηλώνεται και ένα διαφορετικό κανάλι. Η επιλογή καναλιού από το οποίο θα παρακολουθούμε τα μηνύματα, δηλώνεται μέσα στο python script που αρχικοποιεί και τρέχει τον προσομοιωτή. Οι συναρτήσεις για τα μηνύματα είναι οι ακόλουθες :

dbg(char* stringID, const char* format, ...): τυπώνει το μήνυμα στο κανάλι που δηλώνεται από το `stringID` . Το μηνυμα δημιουργείται με τον ίδιο τρόπο που το δημιουργεί η `printf()` στην γλώσσα C,ενώ προσθέτει και το χαρακτηριστικό 'DEBUG (<nodeid>):' στην αρχή, για να ξεχωρίζει ποιος κόμβος το τύπωσε.

dbg_clear(char* stringID, const char* format, ...): Κάνει ότι και η `dbg()` με την διαφορά ότι δεν τυπώνει το 'DEBUG (<nodeid>):'

dbgerror(char* stringID, const char* format, ...): τυπώνει το μήνυμα στο κανάλι που δηλώνεται από το `stringID` . Το μήνυμα δημιουργείται με τον ίδιο τρόπο που το δημιουργεί η `printf()` στην γλώσσα C,ενώ προσθέτει και το χαρακτηριστικό 'ERROR (<nodeid>):' στην αρχή, για να ξεχωρίζει ποιος κόμβος το τύπωσε.

dbgerror_clear(char* stringID, const char* format, ...): Κάνει ότι και η `dbgerror()` με την διαφορά οτι δεν τυπώνει το 'ERROR (<nodeid>):'

2.12.2 Δημιουργία Python script για την προσομοίωση

Η αρχικοποίηση και η εκτέλεση του προσομοιωτή για την εφαρμογή γίνεται μέσα από ένα python script ή ενός προγράμματος σε C++ .Σε αυτήν την ενότητα θα περιγραφή η διαδικασία μόνο για τον python script. Οι συναρτήσεις και η διαδικασία που ακολουθείται είναι αντίστοιχη για το πρόγραμμά σε C++. Αφού μεταγλωττιστεί η εφαρμογή για την προσομοίωση, έχουν παραχθεί τα απαραίτητα αρχεία για τον προσομοιωτή. Στο αρχείο `TOSSIM.py` περιέχονται οι απαραίτητες δηλώσεις αντικειμένων και δομών για την αρχικοποίηση του προσομοιωτή. Ένα python script για την προσομοίωση της εφαρμογής Βλινκ, είναι στο 2.18:

Στο αρχείο `topology.txt` , δηλώνεται η τοπολογία του δικτύου και η εξασθένιση σήματος σε κάθε ζεύξη, δηλαδή ποιοι κόμβοι επικοινωνούν με ποιους και με τι εξασθένιση σήματος Η κάθε γραμμή στο αρχείο αυτό, είναι της μορφής

```
<nodeid1> <nodeid2> <gain>
```

Το αρχείο `topology.txt` που διαβάζεται από το παραπάνω python script είναι το ακόλουθο:

```
0 1 -50.0
1 0 -50.0

1 2 -50.0
2 1 -50.0

2 3 -50.0
3 2 -50.0

3 1 -50.0
1 3 -50.0
```

Στο αρχείο `meyer-heavy.txt` , υπάρχει μια ακολουθία δειγμάτων θορύβου, που βάσει αυτής η μέθοδος, `createNoiseModel()` δημιουργεί το μοντέλο θορύβου για κάθε κόμβο του δικτύου στην προσομοίωση. Μερικές ακολουθίες δειγμάτων θορύβου, υπάρχουν στον φάκελο `$TOSDIR/lib/tossim/noise`.

```
#!/usr/bin/python

from TOSSIM import *
import sys ,os
import random

t=Tossim([])
f=sys.stdout #open('./logfile.txt','w')
SIM_END_TIME= 1000 * t.ticksPerSecond()
N_nodes=4

# channel registration
t.addChannel("BlinkC",f)
#definition of bootTimes for each node
for i in range(0,N_nodes):
        m=t.getNode(i)
        m.bootAtTime(10*t.ticksPerSecond() + i)

#topology file reading
topo = open("topology.txt", "r")

if topo is None:
        print "Topology file not opened!!! \n"

r=t.radio()
lines = topo.readlines()
for line in lines:
  s = line.split()
  if (len(s) > 0):
    print " ", s[0], " ", s[1], " ", s[2];
    r.add(int(s[0]), int(s[1]), float(s[2]))

#noise trace reading
mTosdir = os.getenv("TOSDIR")
noiseF=open(mTosdir+"/lib/tossim/noise/meyer-heavy.txt","r")
lines= noiseF.readlines()

for line in  lines:
        str1=line.strip()
        if str1:
                val=int(str1)
                for i in range(0,N_nodes):
                        t.getNode(i).addNoiseTraceReading(val)
noiseF.close()
#noise model creation
for i in range(0,N_nodes):
        t.getNode(i).createNoiseModel()

#simulation event loop

h=True
while(h):
        try:
                h=t.runNextEvent()
                #print h
        except:
                print sys.exc_info()
#                e.print_stack_trace()

        if (t.time()>= SIM_END_TIME):
                h=False
```

Κώδικας 2.18: Το simulation script για το Blink.

2.13 Μεταγλώττιση εφαρμογής και προγραμματισμός αισθητήρων

Για την μεταγλώττιση της εφαρμογής, το TinyOS χρησιμοποιεί μία ακολουθία από Makefiles .Το βασικό `Makefile` , είναι αυτό που δηλώνεται με την μεταβλητή περιβάλλοντος `MAKERULES`. Ο προγραμματιστής της εφαρμογής, γράφει ένα `Makefile` για την εφαρμογή, δηλώνοντας το κύριο configuration component της εφαρμογής, τυχόν sensorboard που συνδέονται στους αισθητήρες και ότι επιπλέον `#include` και `#define` χρειάζεται η εφαρμογή και για τις βιβλιοθήκες που χρησιμοποιεί. Ένα παράδειγμα `Makefile` από την εφαρμογή Blink είναι στο 2.19

2.13.1 Μεταγλώττιση και εγκατάσταση εφαρμογής

Για την μεταγλώττιση της εφαρμογής εκτελούμε την εντολή

```
make <platform>
```

και για να εγκαταστήσουμε την εφαρμογή σε αισθητήρα που είναι συνδεδεμένος στον υπολογιστή, εκτελούμε την εντολή:

```
make <platform> reinstall,<nodeid> <progmethod>,<serialport>
```

Για του αισθητήρες του εργαστηρίου, εκτελούμε τις εντολές:

```
make iris
make iris reinstall,<nodeid> mib520,/dev/ttyUSB0
```

Η μεταγλώττιση της εφαρμογής για χρήση με τον προσομοιωτή TOSSIM του TinyOS γίνεται με την χρήση του ορίσματος `sim` . Προς το παρόν, η προσομοίωση της εφαρμογής είναι εφικτή μόνο για την πλατφόρμα micaz. Η εντολή είναι η ακόλουθη:

```
make micaz sim
```

```
1  COMPONENT=BlinkAppC
2  include $(MAKERULES)
```

Κώδικας 2.19: Το Makefile για την εφαρμογή Blink.

Κεφάλαιο 3

Αναγνώριση εξεχουσών τιμών με την γεωμετρική προσέγγιση

Η γεωμετρική προσέγγιση χρησιμοποιείται για την παρακολούθηση ροών δεδομένων, σε σχέση με ένα σύνολο περιορισμών, σε κατανεμημένα συστήματα. Σκοπός είναι να διαχωρίζονται τοπικά μετρήσεις- δεδομένα, τα οποία δεν επηρεάζουν το τελικό αποτέλεσμα και κατ΄ επέκταση να αποφεύγεται το επιπλέον κόστος επικοινωνίας. Ο διαχωρισμός αυτός, γίνεται καθορίζοντας τοπικούς περιορισμούς σε κάθε κόμβο, τέτοιους ώστε, όσο ικανοποιούνται οι περιορισμοί αυτοί, με σιγουριά ικανοποιούνται και οι ολικοί περιορισμοί.
Στη συνέχεια περιγράφονται βασικά σημεία της γεωμετρικής προσέγγισης

3.1 Η γεωμετρική προσέγγιση

Σύμφωνα με τον αλγόριθμο της γεωμετρικής προσέγγισης ([1], [4]), ο κάθε κόμβος S_i διατηρεί ένα τοπικό d-διάστατο διάνυσμα τιμών $\overrightarrow{v_i}$, που ονομάζεται τοπικό διάνυσμα στατιστικών. Το κάθε j-οστό στοιχείο του τοπικού διανύσματος στατιστικών, συμβολίζεται με $\overrightarrow{v_{j,i}}$. Όλοι οι κόμβοι διατηρούν διανύσματα ίδιας διάστασης. Το ολικό διάνυσμα στατιστικών, $\overrightarrow{v}$, υπολογίζεται σαν ο μέσος όρος, όλων των τοπικών διανυσμάτων στατιστικών. Το j-οστό στοιχείο του, συμβολίζεται με $\overrightarrow{v_j}$, και υπολογίζεται, $\overrightarrow{v_j} = \frac{1}{n}\sum_{i=1}^{n}\overrightarrow{v_{j,i}}$. Στο πίνακα 3.1, συνοψίζεται ο συμβολισμός που χρησιμοποιείται.

Έστω μία συνάρτηση παρακολούθησης $f : R^d \rightarrow R$,ορισμένη στο ολικό διάνυσμα στατιστικών. Η συνάρτηση αυτή μπορεί να είναι και μή γραμμική. Σκοπός είναι να καθορίζεται, ανά πάσα στιγμή, το αν η τιμή της συνάρτησης $f(\overrightarrow{v(t)})$ υπερβαίνει ένα προκαθορισμένο όριο T.

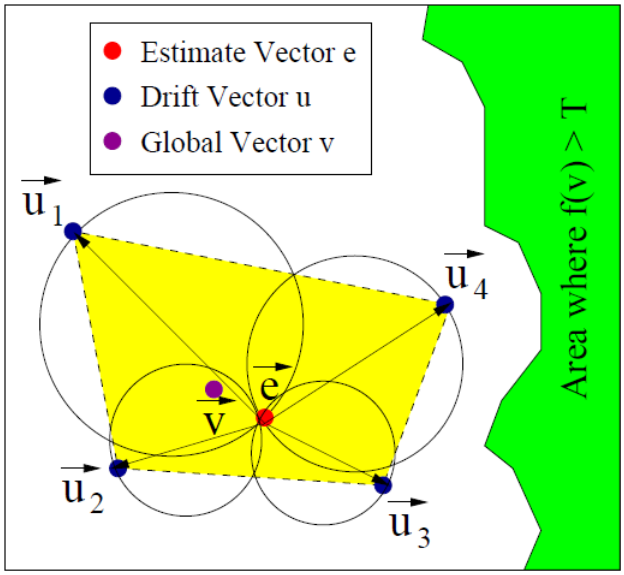
Η γεωμετρική προσέγγιση διαχωρίζει την διαδικασία παρακολούθησης σε ένα σύνολο περιορισμών, που μπορούν να ελεγχθούν τοπικά. Για να επιτευχθεί

Symbol	Definition
S_i	The i-th sensor node
CN_i	The Comparison Neighborhood of S_i: With which nodes does S_i compute its similarity with?
W	Dimensionality of the measurements vector
d	Dimensionality of the local statistics vector
T	The similarity threshold
$\vec{e}$	The estimate vector. Its dimensionality is d
$\vec{v}_i$	The local statistics vector of S_i. Its dimensionality is d
$\vec{v}$	The true (not known by the sites) global statistics vector
$\Delta\vec{v}_i$	The delta vector of S_i. Calculated as the difference of the current local statistic vector from the last local statistic vector that S_i has transmitted.
$\vec{u}_i$	The drift vector of S_i. Equal to $\vec{e} + \Delta\vec{v}_i$
$B(\vec{e}, \vec{u}_i)$	The sphere (ball) having $\vec{e}$ and $\vec{u}_i$ as its diameter
$Conv(\vec{e}, \vec{u}_1, \ldots, \vec{u}_n)$	The convex hull determined by vectors $\vec{e}$, $\vec{u}_1, \ldots, \vec{u}_n$

Σχήμα 3.1: Πίνακας συμβολισμών (πηγή:[1])

αυτό, κατά την εκτέλεση του αλγορίθμου κάθε κόμβος διατηρεί α') ένα διάνυσμα εκτίμησης (estimate vector) $\overrightarrow{e(t)}$, το οποίο υπολογίζεται σαν ο μέσος όρος των τοπικών διανυσμάτων στατιστικών που διατηρεί ο κάθε κόμβος για τους υπόλοιπους, $\overrightarrow{e_i(t)} = \frac{\sum_{j=1}^{n} \overrightarrow{v_j}}{n}$ β') ένα διάνυσμα διαφοράς (delta vector) $\Delta\overrightarrow{v_i}$, το οποίο παριστάνει την διαφορά του τοπικού διανύσματος στατιστικών από την τελευταία μετάδοση του κόμβου S_i ,γ') ένα διάνυσμα ολίσθησης (drift vector) $\overrightarrow{u_i} = \overrightarrow{e} + \Delta\overrightarrow{v_i}$,το οποίο υπολογίζεται από τις προηγούμενες δύο ποσότητες .

Ο διανυσματικός χώρος R^d, αντιπροσωπεύει όλες τις πιθανές θέσεις του ολικού διανύσματος στατιστικών, ανά πάσα στιγμή. Έστω όλα τα σημεία στον R^d , όπου $f(\overrightarrow{v}) \leq T$ είναι χρωματισμένα με το ίδιο χρώμα, ενώ τα υπόλοιπα σημεία είναι χρωματισμένα με ένα διαφορετικό χρώμα. Επειδή, ο κάθε κόμβος δεν μεταδίδει τις μετρήσεις του σε κάθε χρονική περίοδο, το ολικό διάνυσμα στατιστικών, $\overrightarrow{v}$, δεν είναι γνωστό στους κόμβους. Ωστόσο, αυτό που εγγυάται

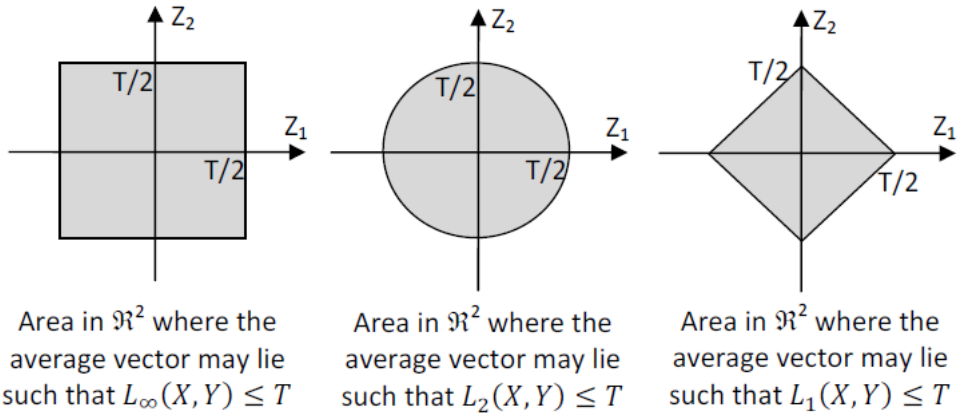


Σχήμα 3.2: Αναπαράσταση περιορισμών με την γεωμετρική μέθοδο (πηγή:[1]).

η γεωμετρική προσέγγιση, είναι ότι το $\overrightarrow{v}$, βρίσκεται πάντα εντός του convex hull $Conv(\overrightarrow{e}, \overrightarrow{u_1}, \cdots, \overrightarrow{u_n})$ που ορίζεται από τα drift vectors και το estimate vector. Κατά συνέπεια, εάν το convex hull $Conv(\overrightarrow{e}, \overrightarrow{u_1}, \cdots, \overrightarrow{u_n})$, είναι μονοχρωματικό, τότε όλοι οι κόμβοι είναι σίγουροι για το χρώμα της συνάρτησης f ,δεδομένου ότι αυτό συμπίπτει με το χρώμα τη συνάρτησης $f(\overrightarrow{e})$. Μία αναπαράσταση του χώρου διανυσμάτων και των περιορισμών φαίνεται στο σχήμα 3.2.

Αξίζει να παρατηρηθεί, ότι αν ο κάθε κόμβος S_i παρακολουθεί την σφαίρα $B(\overrightarrow{e}, \overrightarrow{u_i})$, που κατασκευάζεται με διάμετρο από το $\overrightarrow{e}$ μέχρι το drift vector $\overrightarrow{u_i}$, τότε η ένωση αυτών των σφαιρών, καλύπτει το convex hull. Κατά συνέπεια, αρκεί ο κάθε κόμβος να ελέγχει εάν η σφαίρα του είναι μονοχρωματική ή όχι. Εάν όλες οι σφαίρες είναι μονοχρωματικές, τότε και το convex hull είναι και αυτό μονοχρωματικό, και επομένως έχουν ίδιο χρώμα το $f(\overrightarrow{v})$ και το $f(\overrightarrow{e})$.

Σε περίπτωση παραβίασης κάποιων τοπικών περιορισμών, οι κόμβοι ανταλλάσσουν τα τοπικά διανύσματα στατιστικών τους και υπολογίζουν εκ νέου το διάνυσμα εκτίμησης (estimate vector). Για να ελεγχθεί αν μία σφαίρα είναι μονοχρωματική ή όχι, υπολογίζεται η μέγιστη και ελάχιστη τιμή της f στην σφαίρα $B(\overrightarrow{e}, \overrightarrow{u_i})$. Στο σχήμα 3.3, φαίνεται η αναπαράσταση του ορίου στον 2-διαστατο χώρο, για τις μετρικές ομοιότητας L_∞, L_1, L_2.



Σχήμα 3.3: Αναπαράσταση περιορισμών στον 2-διαστατο χώρο, για τις μετρικές ομοιότητας L_∞, L_1, L_2(πηγή:[1])

3.2 Παρουσίαση προβλήματος

Έστω ένα σταθμός βάσης, ο οποίος παρακολουθεί σε ζεύγη ομοιότητες W-διάστατων διανυσμάτων μετρήσεων, που έχουν συλλεχθεί από τους αισθητήρες. Αν και οι τεχνικές που παρουσιάζονται, δεν επικεντρώνονται στην μέθοδο συλλογής των μετρήσεων, δύο πιθανοί μέθοδοι είναι: α') το διάνυσμα να περιέχει τις τελευταίες W μετρήσεις μίας ποσότητας, ή β') το διάνυσμα να περιέχει τις τωρινές μετρήσεις W ποσοτήτων. .

Από κάθε κόμβο S_i ζητείται να συγκρίνει συνεχώς το διάνυσμα μετρήσεων του με ένα υποσύνολο των υπολοίπων κόμβων του δικτύου. Αυτό το υποσύνολο των υπόλοιπων κόμβων του δικτύου, το ονομάζουμε γειτονιά σύγκρισης (comparison neighborhood) (CN) του κόμβου S_i. Ο S_i, μπορεί να επικοινωνεί με τους κόμβους της γειτονιάς σύγκρισης είτε άμεσα (εντός εμβέλειας ασύρματου), είτε έμμεσα (μέσω ενδιάμεσων κόμβων).

Σκοπός είναι ο σταθμός βάσης της εφαρμογής, να γνωρίζει με ακρίβεια (θεωρώντας ότι δεν υπάρχουν απώλειες μηνυμάτων), ποιος κόμβος είναι όμοιος ή ανόμοιος με ποιόν άλλον στην γειτονιά σύγκρισης τους. Αν ο κόμβος S_i, ανιχνεύσει ότι η ομοιότητα με τον κόμβο $S_j \in CN_i$ έχει αλλάξει, τότε ο ένας από τους δύο χρειάζεται να ενημερώσει τον σταθμό βάσης. Για την αποστολή των ειδοποιήσεων αυτών στον σταθμό βάσης, μπορεί να χρησιμοποιηθεί κάποια δομή δικτύου, όπως είναι το aggregation tree .

3.3 Συναρτήσεις ομοιότητας

Σε αυτή την ενότητα περιγράφεται πώς μπορούν να μετασχηματιστούν οι συναρτήσεις ομοιότητας έτσι ώστε να χρησιμοποιηθούν με την γεωμετρική προσέγγιση

Θεωρούμε ότι θέλουμε να υπολογίσουμε την ομοιότητα μεταξύ δύο κόμβων με διανύσματα μετρήσεων X και Y, αντίστοιχα. Επειδή μερικοί από τους μετασχηματισμούς δεν χειρίζονται τα διανύσματα μετρήσεων με συμμετρικό τρόπο, θεωρούμε ότι το διάνυσμα μετρήσεων X ανήκει στον κόμβο με το μικρότερο node id.

Για να χρησιμοποιηθεί η γεωμετρική προσέγγιση, πρέπει κάθε συνάρτηση ομοιότητας να εκφραστεί σαν συνάρτηση πάνω στα τοπικά διανύσματα στατιστικών X' και Y', με τα στοιχεία των X' και Y' να υπολογίζονται από τα στοιχεία των X και Y αντίστοιχα. Σε αυτό το σημείο χρειάζεται να τονίσουμε τα παρακάτω σημεία:

- Γενικά, τα X' και Y' μπορεί να έχουν διαφορετική διάσταση από τα X και Y
- Οι διαστάσεις του υποχώρου που οι κάθε κόμβος παρακολουθεί, ανταποκρίνονται στην διάσταση των X' και Y' και όχι των X και Y.
- Κατά την διάρκεια της παρατήρησης, κάθε κόμβος χρειάζεται να υπολογίσει την τιμή της συνάρτησης ομοιότητας πάνω σε κάθε σημείο Z της ζώνης παρατήρησης. Κάθε τέτοιο σημείο Z, αναπαριστάνει μια πιθανή θέση του ολικού διανύσματος στατιστικών, και δεν πρέπει να συγχέεται με το πώς η συνάρτηση υπολογίζεται για τα διανύσματα X,X',Y,Y'. Η τελευταία στήλη της εικόνας 3.4, δείχνει πώς υπολογίζεται η τιμή της συνάρτησης πάνω σε κάθε σημείο Z.

Παρακάτω εξετάζονται οι συναρτήσεις ομοιότητας L_∞, L_1, L_2 και L_k νορμών. Οι συναρτήσεις ομοιότητας και οι μετασχηματισμοί τους φαίνονται στο πίνακα 3.4.

Για την περίπτωση της νόρμας L_∞ έχουμε:

$$\begin{aligned} L_\infty(X,Y) &= \max_{i=1\cdots W}\{|X_i - Y_i|\} \\ &= 2\max_{i=1\cdots W}\{|\frac{X_i - Y_i}{2}|\} = 2\max_{i=1\cdots W}\{|\frac{X_i + (-Y_i)}{2}|\} \end{aligned}$$

Άρα, απλά θέτοντάς $X' = X$ και $Y' = -Y$, η νόρμα $L_\infty(X,Y)$ μπορεί να υπολογιστεί σαν συνάρτηση του διανύσματος μέσου όρου $\frac{1}{2}(X'+Y')$. Οι μετασχηματισμοί για τις υπόλοιπες νόρμες, είναι αντίστοιχοι. Παρακάτω παρουσιάζονται οι μετασχηματισμοί για τις νόρμες L_1, L_2 και L_k.

Similarity Metric	*Function based on vectors X, Y of two nodes*	*Transforming the Function Calculation into a Function over Average Quantities that each node may compute individually*	*Local Statistic Vectors X', Y' (may contain more elements than X, Y)*	*Value of function on any point Z*
$\|\mathbf{X}-\mathbf{Y}\|_\infty$	$dist(X,Y)=\max_i\|X_i-Y_i\|$	$2\max_i\left(\left\|\frac{X_i-Y_i}{2}\right\|\right)$	$X'=\begin{bmatrix}X_1\\ \dots\\ X_W\end{bmatrix}\quad Y'=\begin{bmatrix}-Y_1\\ \dots\\ -Y_W\end{bmatrix}$	$2\max_i(\|Z_i\|)$
$\|\mathbf{X}-\mathbf{Y}\|_1$	$dist(X,Y)=\sum_{i=1}^{W}\|X_i-Y_i\|$	$2\sum_{i=1}^{W}\left\|\frac{X_i-Y_i}{2}\right\|$	$X'=\begin{bmatrix}X_1\\ \dots\\ X_W\end{bmatrix}\quad Y'=\begin{bmatrix}-Y_1\\ \dots\\ -Y_W\end{bmatrix}$	$2\sum_{i=1}^{W}\|Z_i\|$
$\|\mathbf{X}-\mathbf{Y}\|_2$	$dist(X,Y)=\sqrt{\sum_{i=1}^{W}(X_i-Y_i)^2}$	$\sqrt{4\sum_{i=1}^{W}\left(\frac{X_i-Y_i}{2}\right)^2}$	$X'=\begin{bmatrix}X_1\\ \dots\\ X_W\end{bmatrix}\quad Y'=\begin{bmatrix}-Y_1\\ \dots\\ -Y_W\end{bmatrix}$	$2\sqrt{\sum_{i=1}^{W}Z_i^2}$
$\|\mathbf{X}-\mathbf{Y}\|_k$	$dist(X,Y)=\sqrt[k]{\sum_{i=1}^{W}\|X_i-Y_i\|^k}$	$\sqrt[k]{2^k\sum_{i=1}^{W}\left(\frac{X_i-Y_i}{2}\right)^k}$	$X'=\begin{bmatrix}X_1\\ \dots\\ X_W\end{bmatrix}\quad Y'=\begin{bmatrix}-Y_1\\ \dots\\ -Y_W\end{bmatrix}$	$2\sqrt[k]{\sum_{i=1}^{W}\|Z_i\|^k}$

Σχήμα 3.4: Πίνακα μετασχηματισμών συναρτήσεων ομοιότητας L_∞, L_1, L_2, L_k (πηγή:[1])

$$
\begin{aligned}
L_1(X,Y) &= \sum_{i=1}^{W}|X_i-Y_i| \\
&= 2\sum_{i=1}^{W}|\frac{X_i-Y_i}{2}| = 2\sum_{i=1}^{W}|\frac{X_i+(-Y_i)}{2}| \\
L_2(X,Y) &= \sqrt{\sum_{i=1}^{W}(|X_i-Y_i|)^2} \\
&= \sqrt{2^2\sum_{i=1}^{W}(|\frac{X_i-Y_i}{2}|)^2} = 2\sqrt{\sum_{i=1}^{W}(|\frac{X_i+(-Y_i)}{2}|)^2} \\
L_k(X,Y) &= \sqrt[k]{\sum_{i=1}^{W}(|X_i-Y_i|)^k} \\
&= \sqrt[k]{2^k\sum_{i=1}^{W}(|\frac{X_i-Y_i}{2}|)^k} = 2\sqrt[k]{\sum_{i=1}^{W}(|\frac{X_i+(-Y_i)}{2}|)^k}
\end{aligned}
$$

3.4 Λειτουργία κόμβων

Σύμφωνα με τους αλγόριθμους που παρουσιάστηκαν, η παρακολούθηση για παραβιάσεις των περιορισμών, γίνεται σε ζεύγη κόμβων. Σε αυτήν την ενότητα

Algorithm 1 Node i: Operation under Simple Mode

Require: Threshold T, Similarity Function F

1: Maintain $\vec{v'}_i$: last transmitted local statistics vector
2: Maintain $\vec{e}$: Estimate vector
3: Maintain $\vec{v'}_j$: last received local statistics vector from S_j
4: **while** Asked to Monitor Similarity to S_j **do**
5: Obtain new measurements and form local statistics vector $\vec{v}_i$
6: Compute delta vector $\Delta\vec{v}_i = \vec{v}_i - \vec{v'}_i$
7: Compute drift vector $\vec{u}_i = \vec{e} + \Delta\vec{v}_i$
8: **if** Acting Second in Pair **then**
9: **if** MessageWait(S_j, $\vec{v'}_j$, $\vec{e'}$) = true **then**
10: **if** $\vec{e'}$ and $\vec{e}$ are bi-chromatic **then**
11: Notify base station about global violation
12: **end if**
13: Send local measurements vector to S_j
14: $\vec{v'}_i = \vec{v}_i$, $\vec{e} = \vec{e'}$
15: Continue
16: **end if**
17: **end if**
18: {Acting first, or acting second but did not receive a message}
19: localViolation = checkIfViolation($\vec{e}$, $\vec{u}_i$, F, T)
20: **if** localViolation = true **then**
21: Send local measurements vector to S_j
22: $\vec{v'}_i = \vec{v}_i$
23: MessageWait(S_j, $\vec{v'}_j$, $\vec{e'}$) {Will definitely arrive}
24: Compute $\vec{e} = \frac{1}{2}(\vec{v'}_j + \vec{v}_i)$
25: Continue {If global violation, S_j will send notification}
26: **end if**
27: **if** Acting first **then**
28: **if** MessageWait(S_j, $\vec{v'}_j$, $\vec{e'}$) = true **then**
29: goto 10
30: **end if**
31: **end if**
32: **end while**

Σχήμα 3.5: Ψευδοκώδικας περιγραφής λειτουργίας κόμβου i (αλγόριθμος 1) (πηγή:[1]).

περιγράφεται η λειτουργία των κόμβων, ώστε να επιτυγχάνονται παρατηρήσεις παραβιάσεων από όλο το δίκτυο.

Όπως ήδη αναφέρθηκε, ο κάθε κόμβος διατηρεί στατιστικά για κάθε άλλον κόμβο που ανήκει στην γειτονιά σύγκρισης του, CN. Για την εκτέλεση του αλγορίθμου, δεν απαιτείται κάποια συγκεκριμένη αλληλουχία δράσεων των κόμβων στο ζεύγος σύγκρισης Παρόλα αυτά, για ευκολία παρουσίασης, υποθέτουμε ότι υπάρχει μια σειρά δράσης στο ζεύγος κόμβων, η οποία δεν είναι απαραίτητο να παραμένει ίδια από εποχή σε εποχή

Λέμε ότι υπάρχει τοπική παραβίαση περιορισμών, όταν παραβιάζονται οι τοπικοί περιορισμοί σε έναν κόμβο. Δηλαδή, όταν η σφαίρα ελέγχου του κόμβου αυτού είναι διχρωματική. Ολική παραβίαση υπάρχει, όταν ένας κόμβος σιγουρευτεί ότι η ομοιότητά του με τον άλλον κόμβο έχει αλλάξει χρώμα. Ο κόμβος που ανιχνεύσει την ολική παραβίαση, ενημερώνει τον σταθμό βάσης.

Ο αλγόριθμος 1 (3.5) παριστάνει την λειτουργία κάθε κόμβου. Και οι δύο κόμβοι, σε κάθε εποχή ανανεώνουν τα διανύσματα των μετρήσεων τους και υπολογίζουν τα delta vector και drift vector τους. Ξεχωρίζουμε τις παρακάτω τρεις περιπτώσεις όπου υπάρχει ανταλλαγή μηνυμάτων:

Algorithm 2 Node i: MessageWait Subroutine

Require: Paired node S_j, last received local statistics vector $\vec{v'}_j$, vector $\vec{e'}$

1: Wait for message from S_j
2: **if** Message Arrived, containing vector of measurements **then**
3: Compute local statistics vector $\vec{v}_j$ of S_j
4: $\vec{v'}_j = \vec{v}_j$
5: Compute $\vec{e'} = \frac{1}{2}(\vec{v'}_j + \vec{v}_i)$
6: RETURN true
7: **end if**
8: RETURN false

Σχήμα 3.6: Ψευδοκώδικας περιγραφής αναμονής μηνύματος κόμβου i (αλγόριθμος 2) (πηγή:[1]).

Algorithm 3 Node i: checkIfViolation Subroutine

Require: estimate vector $\vec{e}$, drift vector $\vec{u}_i$, function F, threshold T

1: **if** $F(\vec{e}) > T$ **then**
2: Find min value testVal in ball B($\vec{e}$, $\vec{u}_i$)
3: **else**
4: Find max value testVal in ball B($\vec{e}$, $\vec{u}_i$)
5: **end if**
6: **if** $F(\vec{e})$ and testVal are monochromatic **then**
7: Return false
8: **end if**
9: Return true

Σχήμα 3.7: Ψευδοκώδικας περιγραφής ελέγχου παραβίασης κόμβου i (αλγόριθμος 3) (πηγή:[1]).

1. Όταν ο κόμβος S_i δρα δεύτερος και λάβει μήνυμα τοπικής παραβίασης από τον S_j . Τότε ο S_i ελέγχει αν υπάρχει ολική παραβίαση περιορισμών (αλγόριθμός 3 (3.7)). Για να ελέγξει για ολική παραβίαση, δεν είναι ανάγκη να κατασκευάσει την σφαίρα ελέγχου από το drift vector, καθώς ήδη μπορεί να υπολογίσει το καινούριο ολικό διάνυσμα στατιστικών $\vec{e'}$, εφόσον έχε λάβει το πιο πρόσφατο διάνυσμα στατιστικών του κόμβου S_j .Δηλαδή, αρκεί να ελέγξει αν η συνάρτηση ομοιότητας για το $\vec{e'}$ και για το $\vec{e}$, είναι διαφορετικού χρώματος. Αν υπάρχει ολική παραβίαση περιορισμών, τότε ο κόμβος S_i , ενημερώνει τον σταθμό βάσης.

2. Όταν ο S_i δρα δεύτερος, και εντοπίσει τοπική παραβίαση περιορισμών ενώ ο S_j δεν είχε εντοπίσει. Σε αυτήν την περίπτωση ο S_i στέλνει ένα μήνυμα τοπικής παραβίασης, που περιέχει και το διάνυσμα μετρήσεων του και περιμένει να λάβει το αντίστοιχο διάνυσμα από τον S_j, για να ενημερώσει τα στατιστικά που διατηρεί για αυτόν (αλγόριθμος 2 (3.6)).

3. Όταν ο S_i δρα πρώτος, και δεν ανιχνεύσει τοπική παραβίαση, αλλά ο S_j ανιχνεύσει. Σε αυτήν την περίπτωση γίνεται ότι και ο στην προηγούμενη περίπτωση.

Κεφάλαιο 4

Σχεδιασμός Εφαρμογής

Σε αυτό το κεφάλαιο περιγράφεται η σχεδίαση και η υλοποίησή της εφαρμογής για το Outliers Detection. Όπως περιγράφηκε σε προηγούμενο κεφάλαιο, σκοπός της εφαρμογής είναι να ενημερώνεται ο σταθμός βάσης για τις μεταβολές των μετρήσεων των αισθητήρων, οι οποίες ξεπερνούν ένα όριο. Η σχεδίαση της εφαρμογής, έγινε με γνώμονα την εύκολη μεταφορά των components της σε μελλοντικές υλοποιήσεις, την ευκολία συντήρησης και την απλοποίηση της λειτουργικότητας του κάθε component. Επίσης αποφεύχθηκε η χρήση components , που έχουν υλοποιηθεί για συγκεκριμένες πλατφόρμες, με εξαίρεση αυτά των αισθητηρίων και της δήλωσης volume table για χρήση της μόνιμης μνήμης, όπου ήταν αδύνατο να αποφευχθεί.

4.1 Σχεδίαση εφαρμογής

Στο σχεδιάγραμμα 4.1 φαίνεται το διάγραμμα συνδέσεων των διαφόρων components που χρησιμοποιεί η εφαρμογή για το Outliers Detection. Το κύριο module , που εκτελεί τις λειτουργίες για το Outliers Detection, είναι το `OutliersDetectionP`. Στην θέση του `OutliersDetectionP` μπορεί να συνδεθεί οποιοδήποτε από τα *α΄*) `Linf_OutliersDetectionP`, το οποίο χρησιμοποιεί την συνάρτηση ομοιότητας $L\inf$, *β΄*) `L1_OutliersDetectionP`, το οποίο χρησιμοποιεί την συνάρτηση ομοιότητας L1 ,*γ΄*) `L2_OutliersDetectionP`, το οποίο χρησιμοποιεί την συνάρτηση ομοιότητας L2 . Η σύνδεση του component που θέλουμε να χρησιμοποιηθεί γίνεται στο configuration component `OutlierDetectionC` και επιλέγεται κατά την μεταγλώττιση της εφαρμογής, δηλώνοντας ένα από τα *α΄*) `SIMFUNC_L_INF` ,όταν θέλουμε την συνάρτηση ομοιότητας $L\inf$,*β΄*) `SIMFUNC_L_1`, όταν θέλουμε την συνάρτηση ομοιότητας L1 ,*γ΄*) `SIMFUNC_L_2`, όταν θέλουμε την συνάρτηση ομοιότητας L2 . Αυτό το component με την σειρά του χρησιμοποιεί τις υπηρεσίες *α΄*) διατήρηση δεδομένων

γειτόνων , `NeighborHoodC` , `NeighborHoodClientC` components ,β') συγχρονισμό με τους υπόλοιπους αισθητήρες στο δίκτυο, `TimeSyncC` component, γ') αποστολή μηνυμάτων στο σταθμό βάσης, `CollectionC` , `CollectionSenderC` components, δ') επικοινωνία με τον υπολογιστή, `SerialActiveMessageC` , `SerialAMSenderC` components ,αν πρόκειται για αισθητήρα που λειτουργεί σαν σταθμός βάσης, και ϵ') διατήρηση στατιστικών , `StatsKeeperC` component . Το component `AppManagerC`, λαμβάνει μήνυμα από την σειριακή θύρα, όταν πρόκειται για σταθμό βάσης, και διαδίδει την τιμή αυτή σε όλο το δίκτυο χρησιμοποιώντας τον μηχανισμό του TinyOS, Dissemination. Το μήνυμα αυτό αφορά το τι δεδομένα θα αποστείλει ο κάθε αισθητήρας (στατιστικά ή αρχείο μετρήσεων), στον σταθμό βάσης, ο οποίος με την σειρά του θα τα προωθήσει στην σειριακή θύρα. Το component `MeasLoggerC` καταγράφει στην μόνιμη μνήμη του αισθητήρα, τις μετρήσεις του αισθητήριου, όποτε αυτές διαβάζονται από τον component `OutliersDetectionP`. Επιπλέον, το component `OutliersDetectionP` αρχικοποιεί και ξεκινάει όλα τα υποσυστήματα που είναι απαραίτητα από τα components της εφαρμογής.

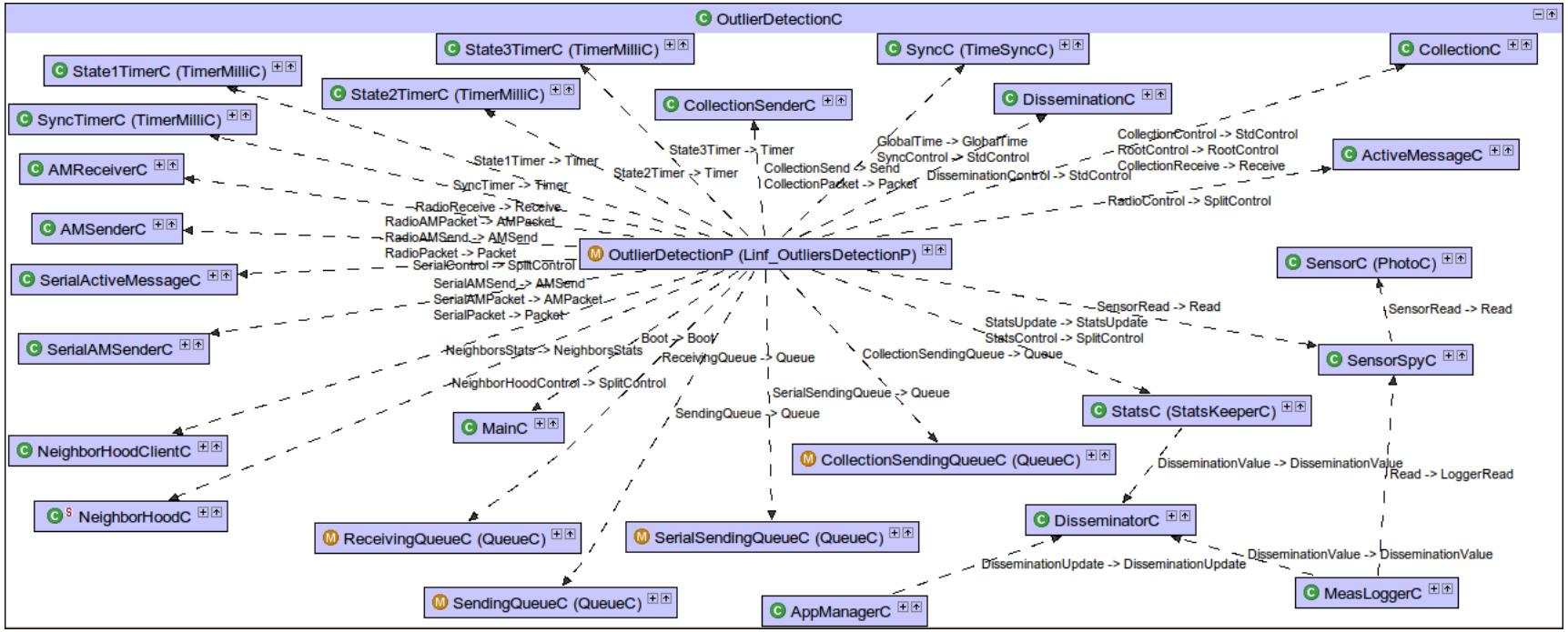
4.2 OutliersDetectionP

Όπως περιγράψαμε και στην προηγούμενη ενότητα, το module `OutliersDetectionP` , είναι το module μέσα στο οποίο υλοποιείται όλη η λογική της εφαρμογής. Σε αυτήν την ενότητα θα περιγράψουμε την ροή εκτέλεσης, και τα μηνύματα που ανταλλάσσει με τους γείτονες και τον σταθμό βάσης.

4.2.1 Επικοινωνία με γείτονες και σταθμό βάσης

Όπως περιγράφηκε και στον αλγόριθμο, για την μέθοδο Outliers Detection που υλοποιήσαμε, απαιτούνται τρία είδη μηνυμάτων, α') ένα για τοπική παραβίαση ορίου μεταξύ δύο κόμβων (`LocalViolationMsg` (4.1)), β') ένα για ενημέρωση στατιστικών ενός κόμβου σε άλλον, ενώ δεν υπάρχει ολική παραβίαση (`StatsUpdateMsg` (4.2)), και γ') ένα για ενημέρωση ολικής παραβίασης τον σταθμό βάσης (`GlobalViolationMsg` (4.3)) .

Με τα μηνύματα τοπικής παραβίασης και τα μηνύματα ανανέωσης στατιστικών, γίνεται ανταλλαγή του διανύσματος τελευταίων μετρήσεων μεταξύ των δύο κόμβων. Το πεδίο `versionNum` στο μήνυμα τοπικής παραβίασης, συμπληρώνεται με τον αριθμό εποχής του κόμβου που το στέλνει, ενώ στο μήνυμα ανανέωσης στατιστικών συμπληρώνεται με τον αριθμό εποχής που είχε ο κόμβος από το οποίο λήφθηκε το μήνυμα τοπικής παραβίασης. Στα μηνύματα ολικής παραβίασης, τα πεδία `senderVersionNum` και `partnerVersionNum`, περιέχουν τον αριθμό εποχής του κόμβου που εντόπισε την ολική παραβίαση και του κόμ-



Σχήμα 4.1: Το διάγραμμα της εφαρμογής OutlierDetectionC

```
typedef nx_struct ODMsgHeader
{
    nx_uint8_t msgType;
    nx_uint8_t simFunc;
} ODMsgHeader_t;

typedef nx_struct LocalViolationMsg{
    ODMsgHeader_t header;
    nx_uint16_t MVector[STATS_VECTOR_LEN];
    nx_uint16_t versionNum;
} LocalViolationMsg_t;
```

Κώδικας 4.1: Δήλωση του LocalViolationMsg.

```
typedef nx_struct StatsUpdateMsg{
    ODMsgHeader_t header;
    nx_uint16_t MVector[STATS_VECTOR_LEN];
    nx_uint16_t versionNum;
} StatsUpdateMsg_t;
```

Κώδικας 4.2: Δήλωση του StatsUpdateMsg.

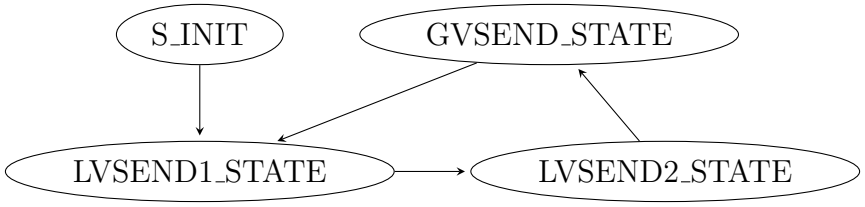
βου που εντόπισε την τοπική παραβίαση, αντίστοιχα. Το πεδίο `msgType`, στην επικεφαλίδα των μηνυμάτων, υποδηλώνει τον τύπο του μηνύματος, απαραίτητο διότι τα μηνύματα χρησιμοποιούν το ίδιο Active Message type.Το πεδίο `simFunc`, δηλώνει τον τύπο της μετρικής ομοιότητας που χρησιμοποιεί το component που στέλνει το μήνυμα. Τα πεδία `sender` και `partner` στα μηνύματα ολικής παραβίασης, συμπληρώνονται με τις διευθύνσεις (node id) του αποστολέα και του κόμβου που εντόπισε την τοπική παραβίαση. Το πεδίο `similarity`, συμπληρώνεται με το αποτέλεσμα της συνάρτησης μετρικής ομοιότητας με τιμές τύπου `float`. Ο τύπος στην δομή του μηνύματος δηλώνεται σαν 32-bit μη προσημασμένος ακέραιος, διότι δεν υπάρχει στην nesC εξωτερικός τύπος μεταβλητής για αριθμούς κινητής υποδιαστολής.

4.2.2 Ροή εκτέλεσης, μηχανή καταστάσεων

Η εκτέλεση του αλγορίθμου γίνεται σε τέσσερις φάσεις, *α'*) `S_INIT`, για τις αρχικοποιήσεις των μεταβλητών του αλγορίθμου και των υποσυστημάτων, καθώς επίσης και για τον συγχρονισμός των κόμβων του δικτύου, *β'*) `LVSEND1_STATE`, όπου παίρνει μέτρηση από το αισθητήριο, ενημερώνει το διάνυσμα μετρήσεων του και στέλνει μηνύματα για τυχόν τοπικές παραβιάσεις σε γείτονες με διεύθυνση (node id) μικρότερη από την διεύθυνση του κόμβου (`TOS_NODE_ID`). *γ'*) `LVSEND2_STATE`, όπου στέλνει μηνύματα για τυχόν τοπικές παραβιάσεις σε γείτονες με διεύθυνση (node id) μεγαλύτερη από την διεύθυνση του κόμβου (`TOS_NODE_ID`),και απο τους οποίους δεν έχει ληφθεί ήδη μήνυμα τοπική παραβίασης. *δ'*) `GVSEND_STATE`, όπου στέλνει μηνύματα για τυχόν ολικές παρα-

```
1  typedef nx_struct GlobalViolationMsg{
2      ODMsgHeader_t header;
3      nx_uint16_t sender;
4      nx_uint16_t partner;
5      nx_uint16_t senderVersionNum;
6      nx_uint16_t partnerVersionNum;
7      nx_uint32_t similarity;
8  }GlobalViolationMsg_t;
```

Κώδικας 4.3: Δήλωση του GlobalViolationMsg.



Σχήμα 4.2: Διάγραμμα ροής εκτέλεσης Outlier Detection module.

βιάσεις με γείτονες από τους οποίους έχει ληφθεί μήνυμα τοπική παραβίασης, καθώς επίσης στέλνει και μηνύματα ενημέρωσης στατιστικών στους γείτονες από τους οποίους έλαβε μήνυμα τοπικής παραβίασης. Το διάγραμμα ροής εκτέλεσης φαίνεται στο σχήμα 4.2.2. Για κάθε φάση του αλγορίθμου έχει οριστεί ένας Timer, εκτός από την φάση `S_INIT`. Ο κάθε Timer αρχικοποιείται κατά τον συγχρονισμό για να κάνει `fired()` με περίοδο `ROUND_DURATION`, η οποία ορίζεται σαν τον άθροισμα της διάρκεια των τριών καταστάσεων συν ένα διάστημα, `SLEEP_TIME`, όπου παραμένει αδρανής.

4.3 Γειτονιά, διατήρηση στατιστικών γειτόνων

Ο αλγόριθμος για την ανίχνευση εξεχουσών τιμών που υλοποιήθηκε, απαιτεί την διατήρηση στατιστικών δεδομένων για κάθε γείτονα κόμβο. Γείτονας κόμβος, θεωρείται ο κάθε κόμβος με τον οποίο μπορεί να επικοινωνήσει αμφίδρομα ο αισθητήρας μέσω της δομής δικτύου που χρησιμοποιεί, και θα συγκρίνει με αυτόν τις μετρήσεις του. Στην υλοποίηση αυτή, σαν δομή δικτύου χρησιμοποιήθηκε η άμεση επικοινωνία χωρίς ενδιάμεσους κόμβους (Active Messages). Κατ΄ επέκταση, με τον όρο γειτονιά, εννοούμε το σύνολο των κόμβων με τους οποίους ο αισθητήρας θα ελέγχει για εξέχουσες τιμές, και θα μπορεί να έχει αμφίδρομη επικοινωνία . Την υπηρεσία της δημιουργίας, συντήρησης της γειτνίασης, καθώς επίσης και της διατήρησης χώρου για τα στατιστικά δεδομένα για κάθε γείτονα, την προσφέρει μέσω του `interface NeighborsStats` το

component NeighborHoodC.Με αυτόν τον τρόπο δημιουργείται ένα επίπεδο αφαίρεσης για την έννοια της γειτονιάς και για την διατήρηση των στατιστικών για τους γείτονες, από τις εφαρμογές που το χρησιμοποιούν. Σε αυτήν την ενότητα περιγράφεται η λειτουργία του component NeighborHoodC (4.3) και το πώς αλληλεπιδρά με τις εφαρμογές που το χρησιμοποιούν.

4.3.1 interface NeighborsStats

Η δήλωση του interface NeighborsStats, φαίνεται στον κώδικα 4.4. Το interface NeighborsStats παρέχεται από τα component NeighborHoodC και NeighborHoodClientC. Το component NeighborHoodClientC (4.3) συνδέεται από την κάθε εφαρμογή που χρησιμοποιεί το interface NeighborsStats για πρόσβαση στα δεδομένα που αποθηκεύονται από αυτήν και όχι στα δεδομένα που μπορεί να έχουν αποθηκευτεί από άλλα components.

Το interface NeighborsStats παρέχει τα παρακάτω commands.

command bool empty() : επιστρέφει TRUE αν υπάρχουν γείτονες, αλλιώς FALSE,

command uint8_t size() : επιστρέφει πόσοι διαφορετικοί γείτονες ήταν ή είναι ενεργοί μέσα στην λίστα γειτόνων που διατηρεί,

command uint8_t maxSize() : επιστρέφει το μέγιστο μέγεθος γειτόνων για τους οποίους μπορεί να διατηρήσει στατιστικά,

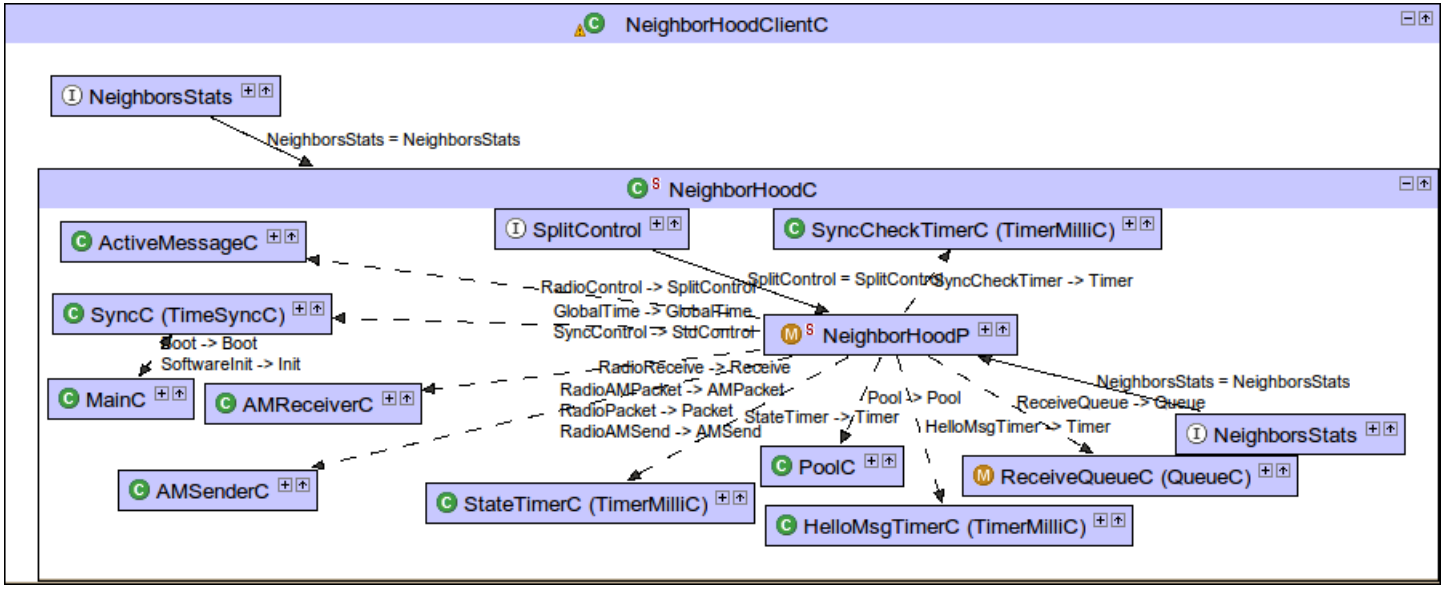
command uint8_t maxStatsSize() : επιστρέφει το μέγιστο μέγεθος σε bytes για την δομή των στατιστικών που μπορεί να χρησιμοποιήσει η εφαρμογή,

component bool isActive(uint16_t id) : επιστρέφει TRUE , αν ο κόμβος με διεύθυνση id είναι ενεργός στην γειτονιά,

command uint8_t activeCount() : επιστρέφει το πλήθος των ενεργών γειτόνων,

command void* getStatsById(uint16_t nodeId) : επιστρέφει έναν pointer στην δομή των στατιστικών για τον γείτονα με διεύθυνση nodeId. Αν δεν υπάρχει στην λίστα γειτόνων ή είναι ανενεργός, επιστρέφει NULL.

command void getIdList(uint16_t* idList) : επιστρέφει στην λίστα idList , η οποία πρέπει να είναι μεγέθους maxSize(), μια ταξινομημένη ως προς την διεύθυνση των γειτόνων, λίστα ενεργών γειτόνων.



Σχήμα 4.3: Το διάγραμμα των components NeighborHoodC και NeighborHoodClientC.

```
1  interface NeighborsStats{
2          command bool empty();
3          command uint8_t size();
4          command uint8_t maxSize();
5          command uint8_t maxStatsSize();
6          command bool isActive(uint16_t id);
7          command uint8_t activeCount();
8
9          /**
10          * Can be used both for data read and write
11          * @param nodeId
12          * @return pointer to data
13          */
14         command void* getStatsById(uint16_t nodeId);
15
16         /**
17          * @param idList
18          * @return array[MAX_NEIGHBORS] with neighbors' ids
19          */
20         command void getIdList( uint16_t* idList);
21 }
```

Κώδικας 4.4: Δήλωση του interface NeighborsStats.

4.3.2 Δημιουργία γειτονιάς, φάσεις εκτέλεσης

Η λειτουργία του component `NeighborHoodC` γίνεται σε δύο φάσεις, *α'*) `UPDATE_PHASE` , *β'*) `STABLE_PHASE`. Η έναρξη της διαδικασίας εύρεσης γειτόνων, ξεκινάει ταυτόχρονα σε όλους τους κόμβους δικτύου, για αυτόν τον λόγο απαιτείται να συγχρονίζονται πριν την ανταλλαγή μηνυμάτων τύπου `HelloMsg`.

Στην πρώτη φάση, `UPDATE_PHASE`, αφού έχουν συγχρονιστεί οι κόμβοι, στέλνονται μηνύματα τύπου `HelloMsg` σε όλους τους κόμβους που είναι στην εμβέλεια (`BROADCAST`) του ασύρματου. Όταν λάβει ένας κόμβος ένα μήνυμα τύπου `HelloMsg`, τότε ελέγχει αν υπάρχει ήδη στην λίστα γειτόνων του. Αν υπάρχει σαν ανενεργός, τον θέτει σαν ενεργό, ενώ αν δεν υπάρχει καθόλου στην λίστα τον προσθέτει, διαγράφοντας αν χρειαστεί αυτόν που ήταν ανενεργός τις περισσότερες πρόσφατες εποχές γειτνίασης. Σαν εποχή γειτνίασης ορίζουμε το χρονικό διάστημα από την μία αρχή της `UPDATE_PHASE` μέχρι την επόμενη αρχή της. Αυτό το χρονικό διάστημα ορίζεται από την σταθερά, `ROUND_PERIOD`, ενώ η διάρκεια της φάσης `UPDATE_PHASE`, ορίζεται από την σταθερά `UPDATE_DURATION`. Η συχνότητα τον μηνυμάτων τύπου `HelloMsg` που στέλνονται κατά την φάση `UPDATE_PHASE`, εξαρτάται από τον λόγο,

$$\frac{HELLOMSG\_PERIOD}{UPDATE\_DURATION}$$

Η σταθερά `HELLOMSG_PERIOD` δηλώνει την περίοδο που θα στέλνονται μηνύματα `HelloMsg` κατά την διάρκεια του `UPDATE_PHASE`.

Στην δεύτερή φάση, STABLE_PHASE, δεν ανταλλάσσονται μηνύματα από το component NeighborHoodC, και η λίστα γειτόνων δεν αλλάζει (ούτε προσθέτονται, ούτε αφαιρούνται γείτονες). Κατά την διάρκεια αυτής της φάσης, όλα τα μηνύματα τύπου HelloMsg τα οποία τυχόν ληφθούν, απορρίπτονται.

4.3.3 Αποθήκευση- Διατήρηση δεδομένων για κάθε γείτονα

Για κάθε γείτονα, διατηρείται μια δομή NeighborRecord, η δήλωση της οποίας φαίνεται στο κώδικα 4.3.3.

```
1  typedef struct NeighborStats{
2      int16_t LastSend[STATS_VECTOR_LEN];
3      int16_t LastRec [STATS_VECTOR_LEN];
4      float E[STATS_VECTOR_LEN];
5      float DriftV[STATS_VECTOR_LEN];
6      bool LVReceived; // when LVReceived... LastRec[] has new values.
7      bool LVSend;// when LVSend==TRUE, LastSend[] is not updated
8      uint16_t versionNum;// the versionNum that was on LV message received
9  }NeighborStats_t;
```

Κώδικας 4.5: Δήλωση δομής στατιστικών ανα γείτονα NeighborStats.

```
1  typedef struct NeighborRecord{
2      uint16_t nodeId;
3      uint8_t activity;
4      uint8_t roundNum;
5      uint8_t NeighborStats[NEIGHBORHOOD_CLIENTS][NEIGHBOR_STATS_LENGTH];
6  }NeighborRecord_t;
```

Κώδικας 4.6: Δήλωση δομής NeighborRecord.

Στο πεδίο nodeId, κρατείται η διεύθυνση του γείτονα, στο πεδίο roundNum κρατείται ο αριθμός περιόδου γειτνίασης κατά την διάρκεια της οποίας παρατηρήθηκε τελευταία φορά ο κόμβος αυτός, και το πεδίο activity, χρησιμοποιείται σαν bitvector για την δραστηριότητα του κόμβου. Το bit στην θέση μηδέν του activity, θέτεται σε 1, όταν ο κόμβος είναι ενεργός για την τρέχουσα περίοδο γειτνίασης. Σε κάθε αλλαγή περιόδου, το activity bitvector για κάθε εγγραφή στην λίστα γειτόνων, μετατοπίζεται κατά μία θέση αριστερά.
Στο πεδίο NeighborStats , δεσμεύεται μνήμη για τα στατιστικά που θα αποθηκεύει κάθε εφαρμογή που χρησιμοποιεί την δομή γειτονιάς. Η σταθερά NEIGHBORHOOD_CLIENTS ισούται με το πλήθος των στιγμιότυπων του component NeighborHoodClientC, ενώ η σταθερά NEIGHBOR_STATS_LENGTH δηλώνει το μέγιστο μέγεθος μιας εγγραφής στατιστικών που μπορεί να κρατήσει μια εφαρμογή για κάθε γείτονα. Η σταθερά NEIGHBOR_STATS_LENGTH,

έχει τιμή 24, και μπορεί να οριστεί με `#define` σε αρχείο εφαρμογής ή με παράμετρο στον μεταγλωττιστή Το ίδιο ισχύει και για τον μέγιστο αριθμό γειτόνων, `MAX_NEIGHBORS` , από το οποίο εξαρτάται το μέγεθος της λίστας γειτόνων. Η δομή των στατιστικών, ορίζεται από την εφαρμογή και δεν επηρεάζει το component `NeighborHoodC`, όσο απαιτεί λιγότερο χώρο από το `NEIGHBOR_STATS_LENGTH` .Η λίστα γειτόνων, διατηρείται ταξινομημένη με αύξουσα σειρά ως προς την διεύθυνση του κόμβου που εισάγεται. Αυτό γίνεται για να επιστρέφεται ταξινομημένη λίστα των διευθύνσεων γειτόνων από το αντίστοιχο command στο interface `NeighborsStats`.Για την αποφυγή αντιγραφής μεγάλων δομών στην μνήμη, χρησιμοποιείται ένα pool (`PoolC` component) με μνήμη που έχει πρόδεσμευτεί. Αντί για αντιγραφή μνήμης, γίνεται αντιγραφή διευθύνσεων (pointers), ενώ κατά την εισαγωγή νέου γείτονα, και διαγραφή ανενεργού, χρησιμοποιούνται οι λειτουργίες `get` και `put` του pool.

4.4 Συγχρονισμός Αισθητήρων

Ό συγχρονισμός των αισθητήρων αποτελεί βασική προϋπόθεσή για την σωστή εκτέλεση του αλγορίθμου και την ελαχιστοποίηση διπλής αποστολής μηνυμάτων τοπικής παραβίασης. Επιπλέον, βοηθάει στην σταθερότητα της λίστας γειτόνων, καθώς έτσι ξεκινάνε όλοι μαζί την φάση ανανέωσης.
Για την εφαρμογή, χρησιμοποιήθηκε μία υλοποίησή του αλγορίθμου Flooding Time Synchronization Protocol [3], που υπάρχει στην βιβλιοθήκη του TinyOS. Σε αυτήν την ενότητα θα γίνει μια σύντομη περιγραφή του αλγορίθμου και το πώς επιτυγχάνουμε τον συγχρονισμό.

4.4.1 Σύντομη περιγραφή αλγορίθμου FTSP

Ο αλγόριθμος FTSP,συγχρονίζει τους κόμβους του δικτύου, ακόμα και πολλαπλών επιπέδων, με σφάλμα της τάξης των $1.48\mu s$, μεταξύ δύο κόμβων, και με επιπλέον σφάλμα της τάξης των $0.5\mu s$ ανά επίπεδο δικτύου, στην περίπτωση πολλαπλών επιπέδων δικτύου. Για τον συγχρονισμό, απαιτεί περίπου ένα μήνυμα ανά κόμβο δικτύου, το οποίο στέλνει περιοδικά κάθε `TIMESYNC_RATE` δευτερόλεπτα. Ο αλγόριθμός αυτός, χρησιμοποιεί το μήνυμα αυτό για να εκτιμήσει τους χρόνους καθυστερήσεων που υπάρχουν κατά την αποστολή του μηνύματος (offset), καθώς επίσης και την διαφορά συχνότητας του ρολογιού (clock skew) που υπάρχει από αισθητήρα σε αισθητήρα. Η ακριβής περιγραφή του αλγορίθμου υπάρχει στο paper [3]. Για την εκτίμηση του global time , χρησιμοποιείται η παρακάτω σχέση :

$$globalTime = localTime + offset + skew * (localTime - syncPoint) \quad (4.1)$$

4.4.2 interface GlobalTime

Το component `TimeSyncC` που υλοποιεί τον συγχρονισμό, παρέχει το `interface GlobalTime` 4.7. Το `interface GlobalTime` , παρέχει τα παρακάτω commands :

uint32_t getLocalTime() : επιστρέφει την τοπική ώρα. Ότι επιστρέφει και το `command Timer.getNow()`.

error_t getGlobalTime(uint32_t *time) : επιστρέφει την global time στην θέση μνήμης του ορίσματος. Επιστρέφει `SUCCESS` όταν ο κόμβος είναι συγχρονισμένος με τους υπόλοιπους στο δίκτυο, και επιστρέφει `FAIL` όταν δεν είναι.

error_t local2Global(uint32_t *time) : μετατρέπει την ώρα που παίρνει σαν όρισμα σε global time.

error_t global2Local(uint32_t* time) : μετατρέπει την ώρα που παίρνει σαν όρισμα σε τοπική ώρα.

4.4.3 Ο συγχρονισμός στην εφαρμογή

Για να συγχρονίσουμε τους κόμβους στο δίκτυο, ώστε να αλλάζουν ταυτόχρονα καταστάσεις, χρησιμοποιούμε τα commands του `interface GlobalTime` για να υπολογίσουμε μια κοινή χρονική στιγμή από την οποία θα ξεκινήσουν να μετράνε οι Timers,για κάθε διαφορετική φάση. Για τον υπολογισμό της κοινής χρονικής στιγμής, παίρνουμε την global time , χρησιμοποιώντας το `command getGlobalTime` και κατόπιν υπολογίζουμε το $\lfloor(\frac{globaltime}{ROUND\_DURATION}) * ROUND\_DURATION\rfloor$. Έπειτα μετατρέπουμε το αποτέλεσμα του υπολογισμού σε local time. Με αυτόν τον τρόπο, ο κάθε κόμβος έχει υπολογίσει την αντιστοιχία σε τοπική ώρα, της κοινής αρχής της περιόδου. Αυτήν την χρονική στιγμή την χρησιμοποιεί σαν αναφορά για τον υπολογισμό της έναρξης του κάθε timer.

4.5 Collection και Dissemination

Στην εφαρμογή γίνεται χρήση και δύο επιπλέον ειδών πρωτοκόλλων επικοινωνίας, των Collection και Dissemination. Και τα δύο είδη αποτελούν πρωτόκολλα χωρίς διευθυνσιοδότηση, και υπάρχουν μερικές υλοποιήσεις τους στο

```
1  #include "Timer.h"
2
3  interface GlobalTime<precision_tag>
4  {
5          /**
6           * Returns the current local time of this mote.
7           */
8          async command uint32_t getLocalTime();
9
10         /**
11          * Reads the current global time. This method is a combination
12          * of <code>getLocalTime</code> and <code>local2Global</code>.
13          * @return SUCCESS if this mote is synchronized, FAIL otherwise.
14          */
15         async command error_t getGlobalTime(uint32_t *time);
16
17         /**
18          * Converts the local time given in <code>time</code> into the
19          * corresponding global time and stores this again in
20          * <code>time</code>. The following equation is used to compute the
21          * conversion:
22          *
23          *        globalTime = localTime + offset + skew * (localTime - syncPoint)
24          *
25          * The skew is normalized to 0.0 (1.0 is subtracted) to increase the
26          * machine precision. The syncPoint value is periodically updated to
27          * increase the machine precision of the floating point arithmetic and
28          * also to allow time wrap.
29          *
30          * @return SUCCESS if this mote is synchronized, FAIL otherwise.
31          */
32         async command error_t local2Global(uint32_t *time);
33
34         /**
35          * Converts the global time given in <code>time</code> into the
36          * correspoding local time and stores this again in
37          * <code>time</code>. This method performs the inverse of the
38          * <code>local2Global</clode> transformation.
39          *
40          * @return SUCCESS if this mote is synchronized, FAIL otherwise.
41          */
42         async command error_t global2Local(uint32_t *time);
43 }
```

Κώδικας 4.7: Δήλωση του interface GlobalTime.

TinyOS source tree. Οι διαφορές μεταξύ των υλοποιήσεων του κάθε ενός είδους πρωτοκόλλου, περιορίζονται στην απόδοση τους και στον αλγόριθμο που χρησιμοποιούν. Ο τρόπος που χρησιμοποιείται η κάθε υλοποίηση, είναι κοινός με τις υπόλοιπες του ίδιου είδους πρωτοκόλλου Η επιλογή των υλοποιήσεων που χρησιμοποιεί η εφαρμογή γίνεται από την επιλογή των φακέλων που θα γίνουν `include` στα ορίσματα του μεταγλωττιστή.
Τα πρωτόκολλα Collection, προσφέρουν ένα μηχανισμό αποστολή μηνυμάτων από οποιονδήποτε κόμβο του δικτύου στην πρώτη ρίζα στην διαδρομή. Η επικοινωνία, μπορεί να είναι πολυεπίπεδη. Σε αυτόν τον μηχανισμό, ένας κόμ-

βος μπορεί να δηλώσει τον εαυτό του ρίζα, χρησιμοποιώντας το `interface RootControl` , ενώ μπορεί να στείλει ένα μήνυμά χρησιμοποιώντας το `interface Send`. Ό,τι μήνυμα στέλνεται, λαμβάνεται μόνο από την πρώτη ρίζα στην διαδρομή, οι τυχόν ενδιάμεσοι κόμβοι απλά προωθούν το μήνυμα Υπάρχουν δύο υλοποιήσεις τέτοιου είδους πρωτοκόλλων στο TinyOS source tree, οι `CTP` και `LQI`. Στην εφαρμογή επιλέχτηκε η χρήση του πρωτοκόλλου `CTP`, για την αποστολή μηνυμάτων ολικής παραβίασης από τους κόμβους στην ρίζα. Τα πρωτόκολλα Dissemination, προσφέρουν έναν μηχανισμό διάδοσης τιμών σε όλους τους κόμβους του δικτύου. Όταν ο κόμβος- παραγωγός αλλάξει την τιμή, τότε οι κόμβοι- πελάτες θα ενημερωθούν με ένα event `changed()` ότι η τιμή ανανεώθηκε. Υπάρχουν τρεις διαφορετικές υλοποιήσεις στο TinyOS source tree, οι `drip, dip` και `dhv`. Για την εφαρμογή, επιλέχτηκε η χρήση του πρωτοκόλλου `dip`.Ο μηχανισμός αυτός χρησιμοποιείται για την διάδοση μιας τιμής για την έναρξη προώθησής στατιστικών ή αρχείου μετρήσεων.

4.6 Το AppManagerC component

Το `AppManagerC` component είναι υπεύθυνο για την λήψη των μηνυμάτων σειριακής θύρας, και την διάδοση τους, μέσω του Dissemination protocolστο υπόλοιπο δίκτυο. Η λειτουργία του αφορά μόνο τον κόμβο σταθμό βάσης. Η σύνδεση του component φαίνεται στο σχήμα 4.4. το μήνυμα που λαμβάνει από τον Η/Υ, είναι τύπου `FwdActive_t` (4.6), το ίδιο και το μήνυμα που διαδίδει.

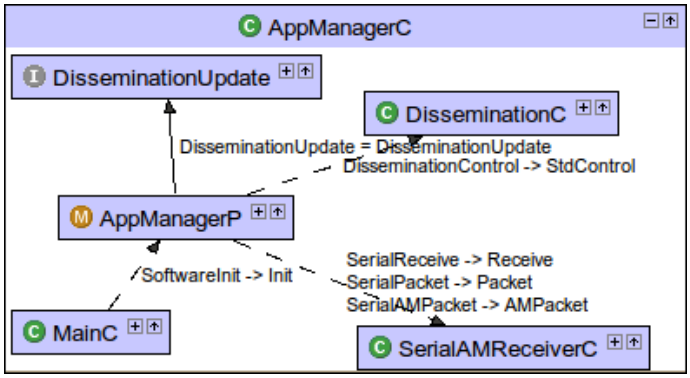
```
1  typedef nx_struct FwdActive
2  {
3      nx_uint8_t activeFwd;
4  }FwdActive_t;
```

Κώδικας 4.8: Δομή του μηνύματος, FwdActive_t.

Στο μήνυμα, το πεδίο `activeFwd` , δηλώνει τι ζητάει να προωθηθεί μέσω δικτύου στον σταθμό βάσης. Όταν πάρει την τιμή `FWD_NONE` , δεν προωθεί τίποτα, όταν πάρει την τιμή `FWD_ODSTATS`, το component `StatsKeeperC` προωθεί τα στατιστικά του Outliers Detection , ενώ όταν πάρει την τιμή `FWD_MEASLOG`, το component `MeasLoggerC`, προωθεί τις μετρήσεις απο το αρχείο καταγραφής. Οι σταθερές αυτές δηλώνονται στο αρχείο `AppManager.h`.

4.7 Διατήρηση στατιστικών εφαρμογής

Η διατήρηση των στατιστικών της εφαρμογής, καθώς και η αποστολής τους στον σταθμό βάσης, γίνεται από το component `StatsKeeperC`. Η ενημέρωση



Σχήμα 4.4: Το διάγραμμα του component AppManagerC

για το αν θα προωθήσει ή όχι τα στατιστικά στον σταθμό βάσης, γίνεται μέσω του πρωτόκολλο Dissemination που χρησιμοποιεί η εφαρμογή. Σε αυτήν την ενότητα περιγράφεται ο τρόπος ενημέρωσης των στατιστικών από την εφαρμογή και ο τρόπος αποστολής των στατιστικών αυτών στον σταθμό βάσης.

```
1  interface StatsUpdate<stats_t>
2  {
3          command error_t update(stats_t newVal);
4          command error_t sync();
5          event void syncDone(error_t result);
6  }
```

Κώδικας 4.9: Δήλωση του interface StatsUpdate.

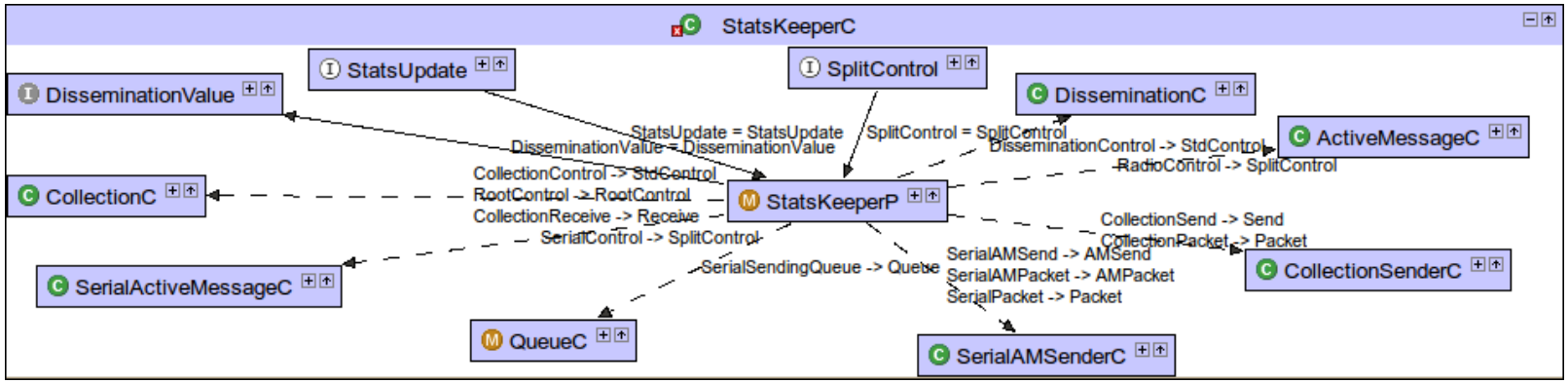
4.7.1 Ενημέρωση στατιστικών

Η εφαρμογή ενημερώνει τα στατιστικά μέσω του `interface StatsUpdate` 4.9. Η δομή των στατιστικών δηλώνεται στο `nx_struct StatsEntry` 4.7. Η δομή των στατιστικών, δίδεται σαν παράμετρος στο `interface StatsUpdate`. Το interface παρέχει τα παρακάτω δύο commands και ένα event .

command error_t update(stats_t newVal): για να ενημερώνει τα στατιστικά,

command error_t sync(): για να συγχρονίζει τον buffer , που κρατάει στην μόνιμη μνήμη. Δεν χρησιμοποιήθηκε στην εφαρμογή λόγω πολλών αποτυχημένων εγγραφών στην μνήμη.

event void syncDone(error_t result): για να ενημερώνει την εφαρμογή για την ολοκλήρωση του συγχρονισμού με την μόνιμη μνήμη.



Σχήμα 4.5: Το διάγραμμα του component StatsKeeperC

```
1  typedef nx_struct StatsEntry
2  {
3      nx_uint16_t version;
4      nx_uint8_t simFunc;
5      nx_uint32_t measurementPeriod;// or RoundDuration
6      nx_uint16_t roundRun;//
7      nx_uint16_t SentMsgCount; // counts only LV and StatsUpdate
8      nx_uint16_t ReceivedMsgCount;
9      nx_uint16_t LocalViolationsCount;
10     nx_uint16_t GlobalViolationsCount;
11 }StatsEntry_t;
12
13 typedef nx_struct StatsMsg
14 {
15     nx_uint8_t statsType;
16     nx_uint16_t origin;
17     StatsEntry_t statsEntry;
18 }StatsMsg_t;
```

Κώδικας 4.10: Δήλωση της δομής στατιστικών StatsEntry, και δομή μηνύματος StatsMsg.

.

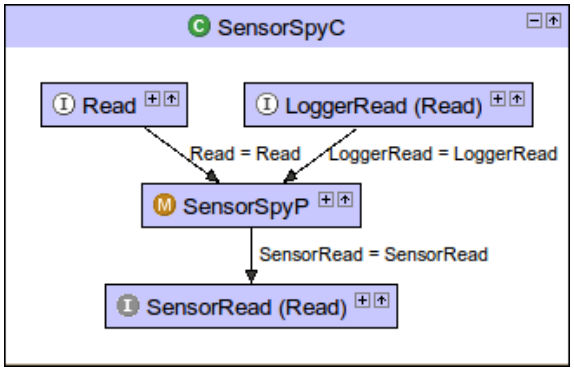
Για την διατήρηση των στατιστικών, το component `StatsKeeperC`, χρησιμοποιεί μια μεταβλητή τύπου `StatsEntry_t` και εμβέλειας εντός του component.

4.7.2 Αποστολή στατιστικών στο σταθμό βάσης

Για την αποστολή των στατιστικών στο σταθμό βάσης, χρησιμοποιείται το πρωτόκολλο Collection. Όταν αλλάξει η τιμή του `FwdActive_t` μέσω του πρωτοκόλλου Dissemination, και πάρει την τιμή `FWD_ODSTATS`, τοτε το component `StatsKeeperC` στέλνει ένα μήνυμα `StatsMsg_t` 4.7, στον σταθμό βάσης με το τελευταίο στιγμιότυπο των στατιστικών. Το component `StatsKeeperC` στον σταθμό βάσης, προωθεί τα μηνύματα αυτά στην σειριακή θύρα.

4.8 Καταγραφή μετρήσεων αισθητήρα

Η καταγραφή των μετρήσεων του αισθητηρίου, καθώς επίσης και η αποστολή τους στον σταθμό βάσης γίνονται από το component `MeasLoggerC`. Και αυτό το component, όπως και το component `StatsKeeperC`, χρησιμοποιεί τα πρωτόκολλα δικτύου Collection και Δισσεμινατιον για την αποστολή προς τον σταθμό βάσης και λήψη `FwdActive_t` μηνύματος, αντίστοιχα. Το component `MeasLoggerC`, καταγράφει τις μετρήσεις του αισθητηρίου στην μόνιμη μνήμη του αισθητήρα με χρήση της δομής αποθήκευσης LogStorage.



Σχήμα 4.6: Το διάγραμμα του component SensorSpyC

4.8.1 Παρακολούθηση αναγνώσεων αισθητήρα

Η παρακολούθηση των αναγνώσεων του αισθητηρίου, γίνεται μέσω του interface `Read`. Το `component MeasLoggerC`, δεν καλεί ποτέ το command `Read.read()`, για την ανάγνωση της τιμής του αισθητηρίου, απλά χρησιμοποιεί το `event Read.readDone()`. Για να γίνεται `signal` αυτό το event από το στιγμιότυπο του component του αισθητηρίου, χρειάστηκε να υλοποιηθεί ένα component το οποίο παίζει ρόλο 'κατασκόπου'. Αυτό το component είναι το `SensorSpyC` (4.6). Το component `SensorSpyC`, παρέχει δύο `interfaces Read` , ένα για την εφαρμογή που χρησιμοποιεί την μέτρηση και καλεί το `command Read.read()` και ένα για το component `MeasLoggerC` που απλά καταγράφει την μέτρηση. Επίσης χρησιμοποιεί ένα `interface Read`, για να παίρνει την μέτρηση από το αισθητήριο.

Όταν η εφαρμογή καλεί το `command Read.read()` , το `SensorSpyC` component καλεί και αυτό με την σειρά του το `command Read.read()` για ανάγνωση μέτρησης του αισθητηρίου. Όταν το `event Read.readDone()` γίνει `signal` , τοτε το component `SensorSpyC` , με την σειρά του κάνει `signal` τα `event Read.readDone()` και για το interface που είναι συνδεδεμένο με την εφαρμογή και για αυτό που είναι συνδεδεμένο με το `MeasLoggerC` component. Με αυτό τον τρόπο, όποτε η εφαρμογή καλεί το `command Read.read()`, και διαβάζει μια μέτρηση από το αισθητήριο, γίνεται και μια καταγραφή μέτρησης και από το `MeasLoggerC` component.

4.8.2 Καταγραφή μετρήσεων στην μόνιμη μνήμη

Για την καταγραφή των μετρήσεων στην μόνιμη μνήμη, χρησιμοποιείται ο τύπος αποθήκευσης LogStorage. Σε κάθε ανάγνωση μέτρησης του αισθητήριου, δημιουργείται και μία νέα καταγραφή στην μόνιμη μνήμη, τύπου `MeasLogEntry`

(4.8.2) . Η δομή αυτή αποτελείται από τα παρακάτω πεδία :

round : ο αύξοντας αριθμός ανάγνωσης για την μέτρηση του αισθητηρίου που καταγράφηκε

Mcount : Το πλήθος των μετρήσεων που έχουν καταγραφεί μέσα στο διάνυσμα του επόμενου πεδίου,

Meas[MAX_MEASLOG :] Το διάνυσμα μετρήσεων που έχουν καταγραφεί. Η σταθερά `MAX_MEASLOG` , ισούται με 1 για την εφαρμογή αυτή. Το ίδιο και το πεδίο `Mcount`.

```
1   typedef nx_struct MeasLogEntry
2   {
3       nx_uint16_t round;
4       nx_uint8_t Mcount;
5       nx_uint16_t Meas[MAX_MEASLOG];
6   }MeasLogEntry_t;
7
8   typedef nx_struct MeasLogEntryMsg
9   {
10      nx_uint16_t origin; // node that sends the measurement
11      MeasLogEntry_t MeasEntry; // entry stored in origin's log file
12  } MeasLogEntryMsg_t;
```

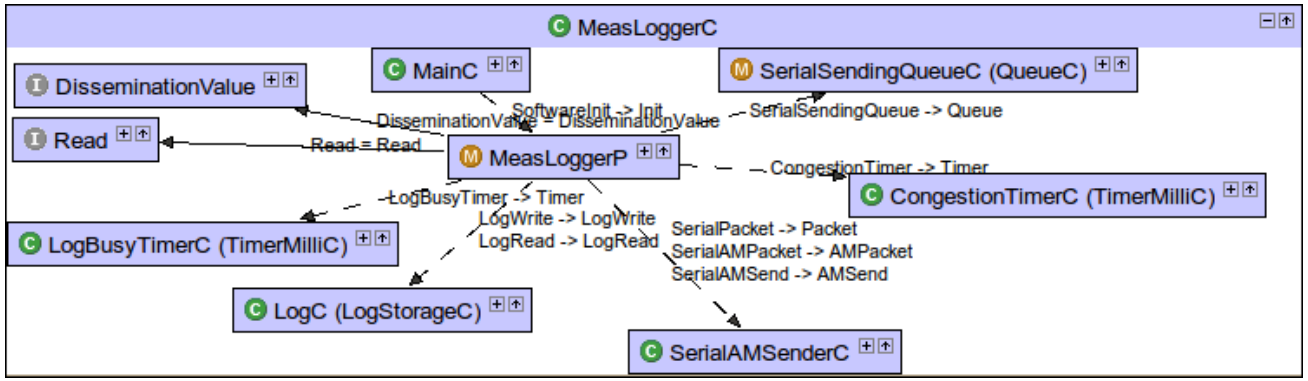
Κώδικας 4.11: Δομή εγγραφής αρχείου μετρήσεων, MeasLogEntry, και δομή μηνύματος MeasLogEntryMsg.

4.8.3 Αποστολή αρχείου καταγραφής στον σταθμό βάσης

Για την αποστολή του αρχείου καταγραφής μετρήσεων στον σταθμό βάσης, χρησιμοποιείται το πρωτόκολλο Collection.Όταν η τιμή του `FwdActive_t` αλλάξει και γίνει `FWD_MEASLOG` , τότε το component `MeasLoggerC` ξεκινάει να διαβάζει μία-μία καταγραφή από την αρχή του αρχείου και να την στέλνει με μήνυμα στον σταθμό βάσης. Η δομή του μηνύματος `MeasLogEntryMsg` φαίνεται στο κομμάτι κώδικα 4.8.2. Σε κάθε μήνυμα συμπεριλαμβάνεται και μια καταγραφή αρχείου. Το component `MeasLoggerC` στον σταθμό βάσης προωθεί τα μηνύματα αυτά στην σειριακή θύρα.

4.9 Επικοινωνία Η/Υ με σταθμό βάσης

Όπως περιγράφηκε στις προηγούμενες ενότητες, τόσο το κύριο component `OutlierDetectionP`, όσο και τα components `MeasLoggerC` , `AppManagerC` ,



Σχήμα 4.7: Το διάγραμμα του component MeasLoggerC

StatsKeeperC, είτε στέλνουν είτε λαμβάνουν μηνύματα στην σειριακή θύρα. Σε αυτήν την ενότητα θα περιγραφούν τα προγράμματα που τρέχουν στον υπολογιστή-βάση, και επικοινωνούν με αυτά τα components.

4.9.1 Λήψη μηνυμάτων ολικής παραβίασης

Μηνύματα ολικής παραβίασης, προωθούνται στην σειριακή θύρα από το component OutlierDetectionP στον σταθμό βάσης. Για την ανάγνωση των μηνυμάτων αυτών, μέσω σειριακή, υλοποιήθηκε ένα πρόγραμμα σε γλώσσα python, το οποίο τυπώνει το μήνυμα στο τερματικό και καταγράφει όλα τα μηνύματα σε ένα αρχείο κειμένου. Το πρόγραμμα αυτό χρησιμοποιεί την βιβλιοθήκη του TinyOS για την επικοινωνία μέσω σειριακής.

4.9.2 Λήψη αρχείων καταγραφών μετρήσεων

Μηνύματα καταγραφής μετρήσεων, προωθούνται στην σειριακή θύρα από το component MeasLoggerC στον σταθμό βάσης. Για να ξεκινήσουν να στέλνονται μηνύματα με τις εγγραφές των μετρήσεων, θα πρέπει ο Η/Υ να στείλει ένα μήνυμα τύπου FwdActive_t στην σειριακή θύρα, και το πεδίο activeFwd να έχει τιμή FWD_MEASLOG.
Για την αποστολή και λήψη των καταγραφών μετρήσεων μέσω σειριακής, υλοποιήθηκε ένα πρόγραμμα σε γλώσσα python, το οποίο στέλνει ένα μήνυμα FwdActive_t στην σειριακή θύρα και έπειτα λαμβάνει και τυπώνει τα μηνύματα με τις εγγραφές στο τερματικό και καταγράφει όλες τις μετρήσεις σε ένα αρχείο εντολών matlab/octave , για πιο εύκολη μετέπειτα χρήση των μετρήσεων.

4.9.3 Λήψη στατιστικών εφαρμογής

Τα μηνύματα στατιστικών της εφαρμογής, προωθούνται στην σειριακή θύρα από το component StatsKeeperC στον σταθμό βάσης. Για να ξεκινήσει η προώθηση των στατιστικών από τους αισθητήρες στο δίκτυο, πρέπει ο Η/Υ να στείλει ένα μήνυμα τύπου FwdActive_t στην σειριακή θύρα, και το πεδίο activeFwd να έχει τιμή FWD_ODSTATS.
Για την αποστολή και ανάγνωση των μηνυμάτων αυτών, υλοποιήθηκε ένα πρόγραμμα σε γλώσσα προγραμματισμού python, το οποίο στέλνει ένα μήνυμα τύπου FwdActive_t και έπειτα λαμβάνει, τυπώνει τα μηνύματα στο τερματικό και τα καταγράφει σε ένα αρχείο κειμένου.

4.9.4 Αποστολή αίτησης προώθησης μετρήσεων ή προώθησης στατιστικών

Το component `AppManagerC`, στο σταθμό βάσης, λαμβάνει μηνύματα από την σειριακή θύρα και τα διαδίδει μέσω του πρωτοκόλλου Dissemination σε όλους τους αισθητήρες του δικτύου. Αυτό το μήνυμα, λειτουργεί σαν αίτημα προς τους αισθητήρες του δικτύου για να προωθήσουν μία φορά το αρχείο μετρήσεων τους ή για να προωθήσουν τα στατιστικά τους προς τον σταθμό βάσης. Για την αποστολή του μηνύματος αίτησης στον σταθμό βάσης μέσω σειριακής θύρας.

4.10 Επεκτασιμότητα εφαρμογής

Όπως αναφέρθηκε και στην εισαγωγή του κεφαλαίου, ο σχεδιασμός και η υλοποίηση της εφαρμογής έγινε με σκοπό την εύκολη επέκταση της εφαρμογής, και την επαναχρησιμοποίηση τμημάτων της σε άλλες. Σε αυτήν την ενότητα περιγράφονται, ο τρόπος για χρήση της εφαρμογής με άλλη δομή δικτύου, και ο τρόπος για χρήσης της γειτονιάς (`NeighborHoodC` component), σε άλλη εφαρμογή.

4.10.1 Λειτουργία με άλλες δομές δικτύου

Για να χρησιμοποιηθεί το component `OutlierDetectionP`,με άλλη δομή δικτύου, πρέπει η άλλη δομή δικτύου να παρέχει αμφίδρομη επικοινωνία μεταξύ των κόμβων, και να χρησιμοποιείται μέσω των interfaces `AMSend, AMPacket, Packet` και `Receive`. Κατόπιν, συνδέονται τα components της νέας δομής δικτύου, που παρέχουν αυτά τα interfaces , στα components `OutlierDetectionP` και `NeighborHoodP` (αλλάζοντας το configuration component `NeighborHoodC`). Αυτές οι αλλαγές αρκούν για να χρησιμοποιηθεί μια άλλη δομή δικτύου από την εφαρμογή. Οι άλλες δομές δικτύου, μπορούν να παρέχουν αμφίδρομη επικοινωνία μεταξύ αισθητήρων ακόμα και αν δεν είναι ο ένας στην εμβέλεια ασυρμάτου του άλλου. Για την περίπτωση όπου η δομή δικτύου χρησιμοποιηθεί, υλοποιεί ή παρέχει την λίστα των κόμβων με τους οποίους υπάρχει επικοινωνία, τότε αρκεί να υλοποιηθεί ένα component το οποίο θα παρέχει το interface `NeighborsStats` και θα χρησιμοποιεί αυτήν την λίστα, αντί να χρησιμοποιηθεί το component `NeighborHoodC` που ενημερώνει την λίστα γειτόνων στέλνοντας ένα επιπλέον μήνυμα

4.10.2 Χρήση Γειτονιάς σε άλλη εφαρμογή

Το component `NeighborHoodC`, χρησιμοποιεί τον μηχανισμό άμεσης επικοινωνίας του TinyOS , Active Messages, για την δημιουργία λίστας γειτόνων που βρίσκονται στην εμβέλεια του αισθητήρα. Για την χρήση των υπηρεσιών που παρέχει μέσω του `interface NeighborsStats` , αρκεί η εφαρμογή να ξεκινάει την ασύρματη επικοινωνία μέσω του `interface SplitControl` του `ActiveMessageC` component και να χρησιμοποιεί το interface `NeighborsStats`.

Παράρτημα Α΄

TinyOS interfaces, components, libraries

```
1  #include "Storage.h"
2
3  interface BlockWrite {
4
5    command error_t write(storage_addr_t addr, void* buf, storage_len_t len);
6
7    event void writeDone(storage_addr_t addr, void* buf, storage_len_t len,
8                         error_t error);
9
10   command error_t erase();
11
12   event void eraseDone(error_t error);
13
14   command error_t sync();
15
16   event void syncDone(error_t error);
17 }
```

Κώδικας Α΄.1: Το BlockWrite interface

```
#include "Storage.h"

interface BlockRead {

  command error_t read(storage_addr_t addr, void* buf, storage_len_t len);

  event void readDone(storage_addr_t addr, void* buf, storage_len_t len,
                      error_t error);

  command error_t computeCrc(storage_addr_t addr, storage_len_t len,
                             uint16_t crc);

  event void computeCrcDone(storage_addr_t addr, storage_len_t len,
                            uint16_t crc, error_t error);

  command storage_len_t getSize();
}
```

Κώδικας Α΄.2: Το BlockRead interface

```
#include "Storage.h"

interface LogWrite {

  command error_t append(void* buf, storage_len_t len);

  event void appendDone(void* buf, storage_len_t len, bool recordsLost,
                        error_t error);

  command storage_cookie_t currentOffset();

  command error_t erase();

  event void eraseDone(error_t error);

  command error_t sync();

  event void syncDone(error_t error);
}
```

Κώδικας Α΄.3: Το LogWrite interface

```
#include "Storage.h"

interface LogRead {

  command error_t read(void* buf, storage_len_t len);

  event void readDone(void* buf, storage_len_t len, error_t error);

  command storage_cookie_t currentOffset();

  command error_t seek(storage_cookie_t offset);

  event void seekDone(error_t error);

  command storage_len_t getSize();
}
```

Κώδικας Α′.4: Το LogRead interface

```
interface Mount {
  /**
   * Mount a particular volume. This must be done before the volume's
   * first use. <code>mountDone</code> will be signaled if SUCCESS is
   * returned.
   * @return SUCCESS if mount request is accepted, FAIL if mount has
   *   already been attempted.
   */
  command error_t mount();

  /**
   * Report success or failure of mount operation. If the mount failed,
   * no operation should be perfomed on the volume. Note that success
   * should not be used to indicate that the volume contains valid data,
   * rather failure indicates some major internal problem that prevents
   * the volume from being used.
   *
   * @param error SUCCESS if the mount succeeded, FAIL otherwise.
   */
  event void mountDone(error_t error);
}
```

Κώδικας Α′.5: Το Mount interface

```
#include "Storage.h"

interface ConfigStorage {

  command error_t read(storage_addr_t addr, void* buf, storage_len_t len);

  event void readDone(storage_addr_t addr, void* buf, storage_len_t len,
                      error_t error);

  command error_t write(storage_addr_t addr, void* buf, storage_len_t len);

  event void writeDone(storage_addr_t addr, void* buf, storage_len_t len,
                       error_t error);

  command error_t commit();

  event void commitDone(error_t error);

  command storage_len_t getSize();

  command bool valid();
}
```

Κώδικας Α΄.6: Το ConfigStorage interface

```
#include <message.h>

interface Packet {


  command void clear(message_t* msg);

  command uint8_t payloadLength(message_t* msg);

  command void setPayloadLength(message_t* msg, uint8_t len);

  command uint8_t maxPayloadLength();

  command void* getPayload(message_t* msg, uint8_t len);

}
```

Κώδικας Α΄.7: Το Packet interface.

```
#include <TinyError.h>
#include <message.h>

interface Receive {


  event message_t* receive(message_t* msg, void* payload, uint8_t len);

}
```

Κώδικας Α΄.8: Το Receive interface.

```
interface Send {

  command error_t send(message_t* msg, uint8_t len);

  command error_t cancel(message_t* msg);

  event void sendDone(message_t* msg, error_t error);

  command uint8_t maxPayloadLength();

  command void* getPayload(message_t* msg, uint8_t len);

}
```

Κώδικας Α΄.9: Το Send interface.

```
interface PacketAcknowledgements {


  async command error_t requestAck( message_t* msg );

  async command error_t noAck( message_t* msg );

  async command bool wasAcked(message_t* msg);

}
```

Κώδικας Α΄.10: Το PacketAcknowledgements interface.

```
interface RadioTimeStamping
{

  async event void transmittedSFD( uint16_t time, message_t* p_msg );

  async event void receivedSFD( uint16_t time );
}
```

Κώδικας Α΄.11: Το RadioTimeStamping interface.

```
#include <message.h>
#include <AM.h>

interface AMPacket {

  command am_addr_t address();
  command am_addr_t destination(message_t* amsg);
  command am_addr_t source(message_t* amsg);
  command void setDestination(message_t* amsg, am_addr_t addr);
  command void setSource(message_t* amsg, am_addr_t addr);
  command bool isForMe(message_t* amsg);
  command am_id_t type(message_t* amsg);
  command void setType(message_t* amsg, am_id_t t);
  command am_group_t group(message_t* amsg);
  command void setGroup(message_t* amsg, am_group_t grp);
  command am_group_t localGroup();
}
```

Κώδικας Α΄.12: Το AMPacket interface.

```
#include <TinyError.h>
#include <message.h>
#include <AM.h>

interface AMSend {


  command error_t send(am_addr_t addr, message_t* msg, uint8_t len);

  command error_t cancel(message_t* msg);

  event void sendDone(message_t* msg, error_t error);


  command uint8_t maxPayloadLength();

  command void* getPayload(message_t* msg, uint8_t len);

}
```

Κώδικας Α΄.13: Το AMSend interface.

```
#include "AM.h"

/**
 * @author Phil Levis
 * @author David Moss
 */
interface ActiveMessageAddress {

  /**
   * Set the active message address of this node
   * @param group The node's group ID
   * @param addr The node's active message address
   */
  async command void setAddress(am_group_t group, am_addr_t addr);

  /**
   * @return the active message address of this node
   */
  async command am_addr_t amAddress();

  /**
   * @return the group address of this node
   */
  async command am_group_t amGroup();

  /**
   * Notification that the address or group settings changed.
   */
  async event void changed();

}
```

Κώδικας Α′.14: Το ActiveMessageAddress interface.

Λίστα απο κώδικες

Βιβλιογραφία

[1] *Detecting Outliers in Sensor Networks using the Geometric Approach* Sabbas Burdakis, Antonios Deligiannakis

[2] *Another Outlier Bites the Dust: Computing Meaningful Aggregates in Sensor Networks* In ICDE, 2009. A. Deligiannakis, Y. Kotidis, V. Vassalos, V. Stoumpos, and A. Delis.

[3] *The Flooding Time Synchronization Protocol* Miklos Maroti, Branislav Kusy, Gyala Simon Akos Ledeczi

[4] *A Geometric Approach to Monitoring Threshold Functions Over Distributed Data Streams* Izchak Sharfman, Assaf Schuster , Daniel Keren

[5] *TinyOS Programming* Philip Levis, David Gay CAMBRIDGE UNIVERSITY PRESS ISBN-13 978-0-511-50730-4

[6] *Crossbow MTS/MDA Sensor Board Users Manual* Revision B PN: 7430-0020-04

[7] *Memsic IRIS wireless module Datasheet* `http://www.memsic.com/support/documentation/wireless-sensor-networks/category/7-datasheets.html?download=135%3Airis`

[8] *Memsic MIB520 Gateway Datasheet* `http://www.memsic.com/support/documentation/wireless-sensor-networks/category/7-datasheets.html?download=144%3Amib520cb`

[9] *nesC 1.3 Language Reference Manual* David Gay, Philip Levis, David Culler, Eric Brewer, July 2009

[10] *TinyOS Tutorial — TinyOS Documentation wiki* `http://docs.tinyos.net/tinywiki/index.php/TinyOS_Tutorials`