ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΑΤΡΩΝ
ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ
ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΗΛΕΚΤΡΟΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μελετη και Αναπτυξη ενος Διαχυτου Συστηματος Ψυχαγωγιας με χρηση Ασυρματων Δικτυων αισθητήρων

Ορέστης Ακριβόπουλος
Α.Μ. : 3050

Επιβλέπων: Καθηγητής Παύλος Σπυράκης
Συνεπιβλέπων: Δρ. Ιωάννης Χατζηγιαννάκης

Πάτρα
Σεπτέμβριος 2009

Περίληψη

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον Καθηγητή του τμήματος Μηχανικών Ηλεκτρονικών Υπολογιστών και Πληροφορικής, και επιβλέποντα καθηγητή μου, κ. Παύλο Σπυράκη που μου έδωσε την ευκαιρία να ασχοληθώ με το αντικείμενο της παρούσας διπλωματικής και με τίμησε με την συνεργασία του.

Θερμές ευχαριστίες, επίσης απευθύνονται και στον Δρ. Ιωάννη Χατζηγιαννάκη, συνεπιβλέποντα της διπλωματικής μου, που με την εμπειρία του, τις γνώσεις και την μεθοδικότητα στον τρόπο σκέψης και δουλειάς ανέβασε το επίπεδο της εργασίας μου. Η ουσιαστική επικοινωνία και συνεργασία μαζί του, οι συμβουλές και υποδείξεις συνετελέσαν καθοριστικά στην περάτωση της διπλωματικής μου εργασίας.

Επιπλέον, κατά τη διάρκεια της ανάπτυξης της εφαρμογής και την συγγραφή της διπλωματικής μου εργασίας είχα την χαρά να συνεργαστώ με το φίλο και συνάδελφο Μάριο Λογαρά τον οποίο και ευχαριστώ πολύ για τον χρόνο που μου αφιέρωσε.

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

1.1 Κίνητρο και σημασία του θέματος

Τα τελευταία χρόνια η ανάπτυξη της τεχνολογίας και η διάδοση της ευνοεί την δημιουργία ετερογενών συστημάτων. Η χρήση κινητών συσκευών σχετίζεται πλέον με τις καθημερινές μας ανάγκες και οι τεχνολογικές εξελίξεις επηρεάζουν άμεσα και σε μεγάλο βαθμό την καθημερινή μας ζωή. Κινητά τηλέφωνα, notebooks, PDAs δημιουργούν διαφορετικά συστήματα με διαφορετικές δυνατότητες, διαφορετικές απαιτήσεις και διαφορετικά χαρακτηριστικά. Το Internet αποτελεί ήδη το πιο διαδεδομένο και το πλέον διαθέσιμο μέσο για την διασύνδεση των ετερογενών αυτών συστημάτων. Η διαδικασία αυτή της διασύνδεσής τους, ωστόσο, δεν είναι μία αυτόματη διαδικασία.

Ένας ερευνητικός τομέας που παρουσιάζει ιδιαίτερη άνθιση και εξελίσεται ραγδαία είναι αυτός των ασυρμάτων δικτύων αισθητήρων. Τυπικές εφαρμογές των δικτύων αυτών έχουν να κάνουν με την παρακολούθηση του φυσικού, βιομηχανικού και οικιακού περιβάλλοντος και τις συνθήκες που επικρατούν σε αυτό. Με την προσθήκη actuators (συσκευών ελέγχου) στα δίκτυα αυτά επιτυγχάνουμε τον απομακρυσμένο η ακόμη και τον αυτοματοποιημένο έλεγχο των περιβάλλοντων αυτών. Η χρήση των δικτύων αισθητήρων, ωστόσο, σπανίως συναντάται σε εφαρμογές ψυχαγωγίας και διασκέδασης. Οι μέχρι σήμερα εφαρμογές ψυχαγωγίας αφορούν συστήματα τα οποία κάνουν χρήση συμβατικών μέσων αλληλεπίδρασης (πληκτρολόγιο, ποντίκι, οθόνες). Επιπροσθέτως τα συστήματα αυτά είναι είτε κεντρικοποιημένα είτε αποσυνδεδεμένα μεταξύ τους. Επίσης σε ελάχιστα απο αυτά γίνεται χρήση αισθητήρων και έτσι δεν υπάρχει αλληλεπιδρασή με το φυσικό περιβάλλον. Η προσπάθεια απόκρυψης της ύπαρξης διαφορετικών συστημάτων και η παροχή ενός απλοποιημένου φαινομενικά συστήματος στους τελικούς χρήστες υλοποιεί κατά βάση την ιδέα ενός διάχυτου υπερσυστήματος. Η ανάπτυξη λογισμικού για τέτοια συστήματα παρουσιάζει πολλές προκλήσεις. Η επικοινωνία

διαφορετικών συσκευών μέσω διαφορετικών καναλιών, η μόνιμη αποθήκευση δεδομένων και αντικειμένων καθώς και ένα κοινό μοντέλο αρχιτεκτονικής αποτελούν τις σημαντικότερες απο αυτές.

1.2 Στόχοι της διπλωματικής εργασίας

Η παρούσα διπλωματική εργασία έχει ως στόχο την ανάπτυξη ενός διάχυτου συστήματος ψυχαγωγίας που θα χρησιμοποιεί τις υπάρχουσες υπολογιστικές δυνανοτήτες των σημερινών συστημάτων σε συνδυασμό με τα ασύρματα δίκτυα αισθητήρων. Η υλοποίηση ενός τέτοιου συστήματος απαιτεί την μελέτη της συμπεριφοράς των συσκευών που χρησιμοποιούνται σε ασύρματα δίκτυα αισθητήρων κάτω από συνθήκες υψηλής δυναμικότητας του δικτύου καθώς επίσης και θεμάτων επικοινωνίας και αλληλεπίδρασης με τα υπάρχοντα υπολογιστικά συστήματα. Ενας ακόμη στόχος, εξίσου σημαντικός με τους προηγούμενους, είναι η γνωριμία, και όσο το δυνατόν καλύτερη εξοικείωση, με σύγχρονες τεχνολογίες ανάπτυξης εφαρμογών που βασίζονται στη γλώσσα προγραμματισμού Θαα, και με συγκεκριμένες βιβλιοθήκες της οπως αυτή της Hibernate και της RMI (Remote Method Invocation).

Σημαντικότερο στόχο αποτελούσε ήταν η ανάπτυξη μιας κατανεμημένης εφαρμογής η οποία με την χρήση ενός κοινού μοντέλου αρχιτεκτονικής θα προσπαθούσε να επιλύσει το πρόβλημα του Data Persistence σε ετερογενή συστήματα.

1.3 Συνεισφορά της διπλωματικής εργασίας

Η παρούσα διπλωματική εργασία παρουσιάζει το σχεδιασμό και την υλοποίηση ενός κατανεμημένου διάχυτου συστήματος με την χρήση ασυρμάτων δικτύων αισθητήρων, βασισμένο σε σύγχρονες τεχνολογίες λογισμικού. Επιπλεόν, εξετάζει και αναλύει αρκετές πτυχές της διαδικασίας ανάπτυξης εφαρμογών και γενικά της τεχνολογίας και ανάπτυξης λογισμικού.

Στα πλαίσια της διπλωματικής περιγράφεται το περιβάλλον εργασίας της Hibernate. Η Hibernate είναι μία τεχνολογία που διευκολύνει σε μία εφαρμογή Java τη διαχείριση των δεδομένων προς αποθήκευση κάνοντας την αντιστοίχηση των αντικειμένων της Java σε μία σχεσιακή βάση δεδομένων.

Επίσης μελετήθηκαν εις βάθος οι συσκευές της Sun Micosystems, SunSPOTs, και ο τρόπος σχεδιασμού και ανάπτυξης λογισμικού για αυτά. Μελετήθηκαν θέματα επικοινωνίας και δρομολόγησης πακέτων σε κινητά δίκτυα αισθητήρων.

Ένα ακόμη θέμα με το οποίο υπήρξε ιδιαίτερη ενάσχοληση και προσπάθεια για επίλυση ήταν αυτό του Data Persistence σε ετερογενή συστήματα.

Επιπλέον, χρησιμοποιήθηκαν εργαλεία τα οποία διευκόλυναν την διαδικασία ανάπτυξης της εφαρμογής αλλά παράλληλα εξασφάλιζαν την ορθότητα και την

αποδοτικότητα του παραγόμενου κώδικα. Σημαντική επίσης ήταν η εμπειρία που αποκτήθηκε στο σχεδιασμό και ανάπτυξη μεγάλων συστημάτων λογισμικού όπου η συνεργασία παίζει πολύ σημαντικό ρόλο.

1.4 Δομή της διπλωματικής εργασίας

Σε αυτή την ενότητα παρουσιάζεται η δομή της παρούσας διπλωματικής εργασίας, η οποία χωρίζεται σε τέσσερις θεματικές ενότητες. Η πρώτη ενότητα περιλαμβάνει την εισαγωγή της διπλωματικής, όπου αναφέρονται η χρησιμότητα της διπλωματικής, το κίνητρο ενασχόλησης με το συγκεκριμένο θέμα καθώς και η σημασία του θέματος, οι στόχοι της διπλωματικής εργασίας και η συνεισφορά της, και τέλος η δομή της. Στο δεύτερο μέρος γίνεται η παρουσίαση τεχνολογιών που επιλύουν το πρόβλημα του Data Persistence σε ετερογενή συστήματα. Το τρίτο μέρος ασχολείται με την εφαρμογή που αναπτύχθηκε στα πλαίσια της διπλωματικής εργασίας, όπου υπάρχουν τέσσερα κεφάλαια που αφορούν το σχεδιασμό της, την υλοποίησή της, την πειραματική διαδικασία κατά την οποία ελέγχεται η συμπεριφορά του συστήματος και η λειτουργία των πρωτοκόλλων και στο τέλος μια εναλλακτική υλοποιήση για το ιδιο σύστημα με την χρήση Web Services. Στο τελευταίο μέρος καταγράφονται τα συμπεράσματα και ο απολογισμός της συγγραφής της διπλωματικής καθώς επίσης και η βιβλιογραφία που χρησιμοποιήθηκε.

Πιο αναλυτικά, στο **κεφάλαιο 2** ορίζεται το Data Persistence η αναγκαιότητα του και πως μπορεί να επιτευχθεί σε ετερογενή συστήματα. Αναφέρονται και αναλύονται καποιες υπάρχουσες τεχνικές όπως η Hibernate, Serialization, Spring κ.α.

Στο **κεφάλαιο 3** παρουσιάζεται ο σχεδιασμός μιας βιβλιοθήκης για ετερογενή συστήματα και αναλύεται η αρχιτεκτονική της. Κάθε επίπεδο της αρχιτεκτονικής του συστήματος αναλύεται ξεχωριστά.

Το **κεφάλαιο 4** περιέχει την υλοποιήση της αρχιτεκτονικής που παρουσιάστηκε στο κεφάλαιο 3. Περιγράφεται η υλοποίηση κάθε επιπέδου ξεχωριστά και σε κάθε επίπεδο υπάρχει ανάλυση της υλοποιήσης καθε υπηρεσίας και των πρωτοκόλλων που παρέχει. Επίσης σχολιάζονται οι σχεδιαστικές αποφάσεις πο ελήφθησαν κατά την υλοποίηση της αρχιτεκτονικής.

Στο **κεφάλαιο 5** περιγράφεται η πειραματική διαδικασία που εκτελέστηκε για τον έλεγχο της απόδοσης ολόκληρου του συστήματος. Επιπροσθέτως, έχει ελέγχεται η ορθή λειτουργία κάθε πρωτοκόλλου ξεχωριστά.

Στο **κεφάλαιο 6** παρουσιάζεται η υλοποίηση ενός συστήματος με παρόμοια χαρακτηριστικά αλλά με διαφορετικές τεχνλολογίες υλοποίησης. Στην συγκεκριμένη υλοποίηση χρησιμοποιούνται κυρίως Web Services.

Τέλος, στο **κεφάλαιο 7** παρουσιάζονται κάποια συμπεράσματα το οποία προέκυψαν από τη συγγραφή της διπλωματικής εργασίας, και γίνεται ο απολογισμός

όλων όσων επιτεύχθηκαν από αυτή την προσπάθεια.

Κεφάλαιο 2

Data Persistence

2.1 Ορισμος και Αναγκαιότητα του Data Persistence

Ο όρος Data Persistence (διατήρηση δεδομένων) στην επιστήμη των υπολογιστών αναφέρεται στα δεδομένα εκείνα που δημιουργούνται κατά την εκτέλεση μιας εφάρμογης και διατηρούνται και μετά την ολοκλήρωση και τον τερματισμό της. Χώρις αυτή τη δυνατότητα όλα τα δεδομένα θα ήταν αποθηκευμένα στην RAM και με τον τερμάτισμο του υπολογιστή δεν θα υπήρχαν πλέον στην μνήμη. Επομένως δεν θα μπορούσε να επιτευχθεί η εξαγωγή δεδομένων από κάποια εφαρμογή. Η διατήρηση των δεδομένων επιτυγχάνεται στην ουσία με την αποθήκευση τους στο σκληρό δισκο του υπολογιστή ή σε καποια flash memory. Για παράδειγμα οι επεξεργαστές κειμένου επιτυγχάνουν Data Persistence με την αποθήκευση των εγγράφων τους σε αρχεία στον σκληρό δίσκο έτσι ώστε να αποφευχθεί η απώλεια δεδομένων κατά τον τερματισμό του υπολογιστή.

Οι τρόποι με τους οποίους επιτυγχάνεται το Data Persistence χωρίζονται σε δυο κατηγορίεσ:

- **Orthogonal ή Transparent:** Η διατήρηση των δεδομένων λέγετε Orthogonal ή Transparent όταν υλοποιείται σαν εσώτερικη διεργασία του περιβάλλοντος στο οποίο εκτελείται κάποια εφαρμογή. Ένα Orthogonal Persistence περιβάλλον δεν απαιτέι καμία συγκεκριμένη ενέργεια από την εφαρμογή που εκτελείται και παρέχει την αποθήκευση και την ανάκτηση της κατάστασης (state) της εφαρμογής.

- **Non Orthogonal:** Στην περίπτωση ενός Non Orthogonal Persistence περιβάλλοντος, η εγγραφή και η ανάγνωση των δεδομένων από και προς το μέσο αποθήκευσης υλοποιείται από την εφαρμογή που εκτελείται.

Τα πλεονεκτήματα ένος Orthogonal σε σχέση με ένα Non Orthogonal Persistence περιβάλλον είναι οτι οι εφαρμογές είναι πιο απλες, καθώς το περιβάλλον

είναι υπεύθυνο για το Data Persistence και δεν χρειάζεται υλοποίηση κάποιας ιδιαίτερης λειτουργίας από την εφαρμογή. Όπως είναι κατανοήτο μειώνεται και η πιθανότητα εμφάνισης σφαλμάτων κατά το Data Persistence από την εφαρμογή.

Η πιο απλή υλοποίηση του Data Persistence είναι τα **Snapshots**. Ένα Snapshot είναι ένα αντίγραφο ολόκληρης της κατάστασης (state) μιας εφαρμογής στο μέσο αποθήκευσης των δεδομένων. Χαρακτηριστικό παράδειγμα είναι η λειτουργία της αδρανοποίησης (hibernate) των φορητών υπολογιστών οπου αποθηκέυετε ένα Snapshot ολόκληρης της RAM. Το συγκεκριμενο παράδειγμα ανήκει στην κατηγορία Orthogonal Persistence καθώς δεν απαιτείται κάποια συγκέκριμενη ενέργεια απο τις εφαρμογές που εκτελούνται στον υπολογιστή. Ένα παράδειγμα χρήσης του Non Orthogonal Snapshot είναι οι επεξεργαστές κειμένου καθώς εκτελουν συγκεκριμένες εντολές με σκοπό να αποθηκεύσουν ολόκληρο το κείμενο σε ένα αρχείο.

Η επόμενη απλούστερη τεχνική ειναι τα **Journals**. Σύμφωνα με την συγκεκριμένη τεχνική γίνετε καταγραφή ολών συμβάντων (events) σε κάποιο αρχείο στο οποίο διατηρείται το ιστορικό (log file) , πριν γίνει εφαρμογή τους στο σύστημα. Τέτοια log files καλούνται Journals. Κατά την εκκίνηση, γίνετε ανάγνωση του Journal και κάθε event εφαρμόζεται στο σύστημα. Με αυτόν τον τρόπο αποφεύγετε η απώλεια δεδομένων σε περίπτωση αποτύχιας του συστήματος ή τερματισμού.

Για παράδειγμα, ολόκληρο το ιστορίκο των εντολών του χρήστη (Undo/Redo) σε ένα πρόγραμμα επεξεργασίας εικόνων, όταν καταγράφεται σε ένα αρχείο, αποτελεί ενα Journals, δίνοντας στον χρήστη την δυνατότητα να επαναφέρει την επεξεργασμένη εικόνα σε οποιαδηπότε παλιότερη καταστάση επιθυμεί.

Τα Journals συνήθως συνδυάζονται με άλλες persistence τεχνικές έτσι ώστε να μην είναι απαραίτητη η επανεκτέλεση όλων των events του ιστορικού του συστήματος (ενδεχομένως πολύ μεγάλο) κατά την εκκίνηση.

2.2 Υπάρχουσες Τεχνικές

2.2.1 Hibernate

Η Hibernate είναι μια βιβλιοθήκη ελεύθερου λογισμικού για την γλώσσα προγραμματισμού Java η οποία αντιστοιχεί οντοκεντρικά μοντέλα σε σχήματα παραδοσικών σχεσιακών βάσεων δεδομένων. Είναι μία ολοκληρωμένη λύση στο πρόβλημα της διαχείρισης δεδομένων εκτελώντας όλες τις ενέργειες μεταξύ των εφαρμογών και των σχεσιακών βάσεων δεδομένων απαλλάσσοντας τον προγραμματιστή από επιπρόσθετο κόπο και τον κίνδυνο αστοχιών στην αποτύπωση σχέσεων μεταξύ κλάσεων και σχήματος βάσης.

Ο κεντρικός στόχος αυτής είναι η δημιουργία μίας διεπαφής (interface) μεταξύ των διαδεδομένων σχεσιακών βάσεων δεδομένων και του αντικειμενοστραφούς

προγραμματισμού. Με άλλα λόγια, επιτρέπει τη χρήση μίας σχεσιακής βάσης δεδομένων ως αντικειμενοστραφή. Για την επίτευξη αυτού, δημιουργούνται αντιστοιχίες μεταξύ των εννοιών του αντικειμενοστραφούς προγραμματισμού, όπως οι συσχετίσεις, η κληρονομικότητα και ο πολυμορφισμός, και των πινάκων και σχέσεων μεταξύ αυτών μίας σχεσιακής βάσης. Με αυτόν τον τρόπο ο προγραμματιστής βλέπει τελικά μία αντικειμενοστραφή βάση δεδομένων, παρόλο που στην ουσία χρησιμοποιεί μία σχεσιακή. Έτσι ο προγραμματιστής χρησιμοποιεί τα αντικείμενα της συγκεκριμένης εφαρμογής, τα τροποποιεί σχετικά με τη λογική της εφαρμογής που αναπτύσσει και τα αποθηκεύει (τροποποιεί, διαγράφει και αναζητά) στη βάση ως αντικείμενα.

Η Hibernate δηλαδή είναι το ενδιάμεσο επίπεδο ανάμεσα σε σχεσιακή βάση δεδομένων και σε εφαρμογή. Αναλαμβάνει να κατασκευάσει τον κατάλληλο κώδικα, μετατρέποντας τα αντίκειμενα της Java σε δεδομένα τα οποία αναγνωρίζοντε από την σχεσιακή βάση δεδομένων και τελικά τα στέλνει στη βάση δεδομένων. Παρέχει επίσης όλες τις λειτουργίες που χρειάζονται για να επικοινωνήσει η εφαρμογή με την σχεσιάκη βάση δεδομένων καθώς επίσης και τις βασικές συναρτήσεις που απαιτούνται για την διατήρηση των δεδομένων (persistence storage), δημιουργία, ανάγνωση, τροποποίηση και διαγραφή (CRUD - Create Read Update Delete).

Πλεονεκτηματα της Hibernate

Οι κυριότεροι λόγοι για την χρήση της Hibernate κατα την ανάπτυξη μιας Java εφαρμογής που απαιτείται αλληλεπίδραση με μία σχεσιακή βάση δεδομένων είναι οι εξής:

- **Αύξηση της παραγωγικότητασ:** Η Hibernate αυξάνει σε μεγάλο βαθμό την παραγωγικότητα καθώς εξαλείφει το μεγαλύτερο μέρος της συγγραφής κώδικα που σχετίζεται με την αποθήκευση δεδομένων σε μία βάση δεδομένων. Ο προγραμματιστής ασχολείται μόνο με το λειτουργικό κομμάτι της εφαρμογής και ενδιαφέρεται μόνο για τα αντικείμενα. Αυτά αποθηκεύονται στη βάση και ανακτώνται από αυτήν με ελάχιστο κόπο.

- **Βελτίωση συντηρησιμότητασ:** Με τη χρήση της Hibernate γράφονται σημαντικά λιγότερες γραμμές κώδικα και ο κώδικας είναι πιο κατανοητός και καλογραμμένος. Αυτό κάνει την συντήρηση της εφαρμογής ευκολότερη. Ακόμα, χρειάζονται πολύ λιγότερες γραμμές κώδικα για τις βασικές τροποποιήσεις των δεδομένων (CRUD - Create Read Update Delete) στη βάση δεδομένων. Αυτά, σε μεγάλης έκτασης εφαρμογές, αποδεικνύονται πολύτιμα τόσο όσον αφορά το χρόνο ανάπτυξης της εφαρμογής όσο και στην αποσφαλμάτωση του κώδικα κατα την ανάπτυξη της εφαρμογής

- **Βελτίωση της απόδοσης του συστήματος:** Η βελτίωση στην απόδοση πολύ δύσκολα θα επιτυγχανόταν εάν ο κώδικας που χρειαζόταν για τις CRUD λειτουργίες ήταν γραμμένος απο τον προγραμματιστή (hand-coding). Η Hibernate δημιουργεί πολύ αποδοτικά ερωτήματα, πράγμα που διασφαλίζει την απόδοση σε πολλές περιπτώσεις. Εκτός αυτού, όμως, υποστηρίζει μία πολύ έξυπνη και αποδοτική πολιτική πρώτου και δεύτερου επιπέδου caching. Με αυτόν τον τρόπο επιτυγχάνεται μεγάλη δυνατότητα κλιμάκωσης. Επίσης δίνει τη δυνατότητα στον προγραμματιστή να επιλέξει το επίπεδο caching που επιθυμεί, όντας πολύ έξυπνη στην πολιτική των write-backs. Επίσης, μπορεί να συνδυαστεί πολύ καλά με κάποια από τα σημαντικότερα λογισμικά caching τόσο ελεύθερου λογισμικού όσο και εμπορικά πακέτα.

- **Μεταφερσιμότητα εφαρμογής:** Η Hibernate διαχωρίζει την εφαρμογή απο την βάση δεδομένων και την διάλεκτο της. Υποστηρίζει παράλληλα έναν αριθμό απο διαφορετικές βάσεις δεδομένων. Η συμβατότητα αυτή καθώς επίσης και η δυνατότητα να σύνδεσης με διαφορετική βάση δεδομένων με ελάχιστες τροποποιήσεις επιτρέπει την πλήρη μεταφερσιμότητα της εφαρμόγης. Το γεγονός αυτό στερεί μεν από την Hibernate την εκμετάλλευση των ιδιαίτερων χαρακτηριστικών της χρησιμοποιούμενης βάσης, όμως, και σε αυτή την περίπτωση, δίνεται η δυνατότητα χρήσης πηγαίας SQL μέσα στη Hibernate που εκμεταλλεύεται τα ιδιαίτερα αυτά χαρακτηριστικά. Αυτό βέβαια μειώνει την ανεξαρτησία της Hibernate. Ένας ακόμη λόγος που η Hibernate προσφέρει μεταφερσιμότητα της εφαρμογής είναι το γεγονός οτι οι κλάσεις που αντιστοιχίζονται είναι απλές κλάσεις Java (Plain Old Java Objects POJOs) με αποτέλεσμα να μην χρειάζονται να είναι απόγονοι μιας πολύπλοκης υποχρεωτικής δομής.

2.2.2 Spring

Το Spring Framework είναι λογισμικό ανοιχτού κώδικα (ελεύθερο λογισμικό) που σκοπό έχει να διευκολύνει την ανάπτυξη J2EE λογισμικού σε μεγάλη έκταση και βασίζεται στη γλώσσα προγραμματισμού Java. Αναπτύχθηκε κατά κύριο λόγο, για να αντιμετωπίσει την πολυπλοκότητα ανάπτυξης enterprise εφαρμογών. Οποιαδήποτε εφαρμογή Java μπορεί να επωφεληθεί από τη χρήση της Spring σε σχέση με την απλότητα του κώδικα, τον αποτελεσματικό έλεγχο και τη χαλαρή διασύνδεση της.

Η Spring είναι ενα περιβάλλον εργασίας το οποίο δημιουργήθηκε από τον Rod Johnson και περιγράφτηκε στο βιβλίο του Expert One-on-One J2EE Design and Development που εκδόθηκε τον Οκτώβριο του 2002. Τα βασικότερα χαρακτηριστικά του Spring Framework είναι οτι αποτελεί έναν ελαφρύ (lightweight) aspect-

oriented container ο οποίος εφαρμόζει dependency injection, ενώ παράλληλα αποτελεί και περιβάλλον εργασίας.

Στη συνέχεια παρουσιάζονται αναλυτικότερα τα κυριότερα χαρακτηριστικά του Spring Framework :

- **Lightweight:** Ο συγκεκριμένος όρος αναφέρεται τόσο στο μέγεθος του Spring Framework όσο και στο επιπλέον φόρτο (overhead). Το Spring Framework μπορεί να συμπεριφθεί σε ένα απλό jar αρχείο μεγέθους 2,5 MB, ενώ ο επιπρόσθετος φόρτος εργασίας που απαιτείται είναι αμελητέος.

- **Dependency Injection:** Η Spring προάγει τη χαλαρή διασύνδεση χρησιμοποιώντας μία τεχνική που είναι γνωστή ως dependency injection (DI).Όταν εφαρμόζεται η DI, τα αντικείμενα λαμβάνουν παθητικά τις εξαρτήσεις τους (dependencies) αντί να τις δημιουργούν ή να τις αναζητούν μόνα τους.Είναι κάτι σαν ένα αντίστροφο Java Naming and Directory Interface (JNDI), αντί ένα αντικείμενο να ψάχνει μόνο του για τις εξαρτήσεις του σε έναν container, ο container δίνει τις εξαρτήσεις στο αντικείμενο χωρίς να περιμένει πρώτα να ερωτηθεί.

- **Aspect-Oriented:** Η Spring παρέχει πλούσια υποστηρίξη σε Aspect Oriented Programming (AOP) και επιτρέπει τη δημιουργία εφαρμογών με συνοχή, το οποίο επιτυγχάνεται από το διαχωρισμό της εφαρμογής σε λειτουργικά επίπεδα. Έτσι κάθε αντικείμενο της εφάρμογης υλοποιέι κάποιες συγκεκριμένες λειτουργίες χωρίς να χρειάζεται να γνωρίζει τις υπόλοιπες λειτουργίες της εφαρμογής.

- **Container:** Η Spring αποτελεί έναν ςονταινερ με την έννοια ότι κρατάει και διαχειρίζεται τον κύκλο ζωής και τη διαμόρφωση των αντικειμένων της εφαρμογής. Ο προγραμματιστής μπορεί να ορίσει πώς θέλει να διαμορφώνονται και να δημιουργούνται τα αντικείμενα, καθώς και ποιες θέλει να είναι οι σχέσεις μεταξύ τους.

- **Framework:** Η Spring δίνει τη δυνατότητα να δημιουργηθούν πολύπλοκες εφαρμογές με το συνδυασμό άλλων, λιγότερο πολύπλοκων, εφαρμογών που παρέχει. Στη Spring τα αντικείμενα δημιουργούνται μέσω δηλώσεων σε απλά XML αρχεία. Επίσης παρέχει πολλές δομικές λειτουργίες, όπως η διαχείριση συναλλαγών, επιτρέποντας στον προγραμματιστή να ασχοληθεί περισσότερο με τη λογική της εφαρμογής του.

Το περιβάλλον εργασίας της Spring αποτελείται απο αρκετά, καλά ορισμένα μέρη modules, τα οποία ως σύνολο δίνουν στον προγραμματιστή ό,τι χρειάζεται για την ανάπτυξη εφαρμογών enterprise. Δεν χρειάζεται όλη η εφαρμογή να βασίζεται στη Spring. Ο προγραμματιστής είναι ελεύθερος να επιλέξει εκείνα τα μέρη

που ταιριάζουν στην εφαρμογή του αλλά και να αναζητήσει άλλες επιλογές όταν η Spring δεν τον καλύπτει.

Δύο από τα modules της Spring, το DAO module και το ORM module λύνουν το πρόβλημα του persistence storage. Αναλυτικότερα :

- **DAO module:** Η δουλειά με JDBC συχνά περιλαμβάνει τη δημιουργία αρκετού κώδικα, ο οποίος προσφέρει σύνδεση με τη βάση δεδομένων, δημιουργία κάποιας εντολής (statement), επεξεργασία του αποτελέσματος και κλείσιμο της σύνδεσης με τη βάση. Μέσω του Data Access Object (DAO) module η Spring δίνει τη δυνατότητα απλοποίησης όλης αυτή της διαδικασίας περιορίζοντας τα προβλήματα που προκύπτουν. Επιπλέον, δημιουργεί ένα layer από εξαιρέσεις (exceptions) πάνω από τα μηνύματα σφάλματος που παράγουν αρκετοί εξυπηρετητές βάσεων δεδομένων.

- **ORM module:** Η Spring με το συγκεκριμένο module δίνει τη δυνατότητα χρησιμοποίησης object-relation mapping (ORM) αντί για την απευθείας χρήση του JDBC. Βασίζεται στη λειτουργία του DAO module παρέχωντας υποστηρίξη αρκετών εργαλείων που προσφέρουν object-relation mapping. Η Spring δεν παρέχει το δικό της ORM, υποστηρίζει αρκετά δημοφιλή περιβάλλοντα εργασίας, συμπεριλαμβανομένων των Hibernate, Java Persistence API και Java Data Objects. Το σύστημα διαχείρισης συναλλαγών της Spring υποστηρίζει εκτός από αυτά τα περιβάλλοντα εργασίας και το JDBC.

2.2.3 Serialization

Η διαδικασία του Serialization λύνει το πρόβλημα της διατήρησης των δεδομένων επιτρέπωντας την αποθήκευση και την ανταλλαγή αντικειμένων ως μια ακολουθία απο bits.

Με τον όρο serialization αναφερόμαστε στην διαδικασία μετατροπής ενός αντικειμένου σε μια ακολουθία απο bits με σκοπό να διατηρηθούν σε κάποιο μέσο αποθήκευσης ή να αποσταλούν μέσω κάποιου δικτύου. Η ακολουθία αυτη των bits όταν ξαναδιαβάζεται από το μέσο αποθήκευσης ή από τον δέκτη, μπορεί να δημιουργήσει ένα ακριβή αντίγραφο του αρχικού αντικειμένου. Για ιδιαίτερα περίπλοκα αντικείμενα τα οποία περιέχουν αναφορές σε άλλα αντικείμενα η διαδικασία αυτή δεν μπορεί να γίνει απευθείας.

Η διαδικασία του serialization καλείται επίσης deflating ή marshalling ενός αντικειμένου. Η αντίθετη διαδικασία, δηλαδή η δημιουργία μια συγκεκριμένης δομής δεδομένων από μία ακολουθία από bits, καλείται deserialization (είναι επίσης γνωστή και ως inflating ή unmarshalling).

Οι λειτουργίες που προσφέρει το serialization είναι οι ακόλουθες:

- Μετατρέποντας τα αντικείμενα της εφαρμόγης σε ακολουθία απο bits επιτρέπει την αποθήκευση τους, λύνοντας έτσι το πρόβλημα του Data Persistence.
- Καθιστά δυνατή την κλήση απομακρυσμένων μεθόδων (Remote Procedure Calls - RPC) όπως γίνετε και στο SOAP (Simple Object Access Protocol).
- Επιτρέπει τον διαμοιρασμό αντικειμένων αναμέσα σε διαφορετικές εφαρμογές, ειδικά σε εφαρμογές που έχουν αναπτυχθεί με COM, CORBA κτλ.
- Προσφέρει επίσης μεθόδους για την αναγνώριση τροποποιήσεων σε μεταβαλόμενα στο χρόνο δεδομένα. Εφόσον αυτά ειναι σε ακολουθία απο bits είναι εύκολο να εντωπιστούν οι αλλαγές.

Για να είναι δυνατή η εκμετάλευση των δυνατοτήτων που προσφέρει, θα πρέπει να διατηρηθεί ανεξαρτησία από την αρχιτεκτονική του συστημάτος στο οποίο εκτελείται η εφαρμογή. Για παράδειγμα, η επιτυχής ανταλαγή αντικειμένων μεταξύ δυο υπολιστών διαφορετικής αρχιτεκτονικής θα πρέπει να επιτρέπει την αξιόπιστη ανακατασκευή των αντικειμένων από την ακολουθία των bits. Αυτό σημαίνει πως η απλούστερη και γρηγορότερη λύση της απευθείας αντιγράφης συγκεκριμέων δεδομένων από την μνημη δεν είναι αξιόπιστη εφόσον πρόκειται για διαφορετικές αρχιτεκτονικές. Επομένως με το serialization των δεδομένων σε μια μορφή ανεξάρτητη από την αρχιτεκτονική αντιμετοπίζονται τα προβλήματα της ταξινόμησης των bytes (byte ordering), της διάταξης της μνήμης (memory layout) και της διαφορετικής αναπαράστασης συγκεκριμένων δομών δεδομένων ανάμεσα σε διαφορετικές γλώσσες προγραμματισμού.

Υπάρχουν αρκετές αντικειμενοστραφείς γλώσσες που παρέχουν απευθείας υποστήριξη του object serialization. Μερικές από αυτές είναι οι Ruby, Smalltalk, Python, PHP, Objective-C, Java και όλες οι .NET γλώσσες. Υπάρχουν επίσης βιβλιοθήκες που προσφέρουν object serialization σε γλώσσες που δεν το υποστηρίζουν.

Η Java συγκεκριμένα μέσω της διεπάφης java.io.Serializable που έχει υλοποιηθεί προσφέρει αυτοματοποιημένο object serialization. Ο προγραμματιστής ορίζει μέσω της διεπαφής ποιές κλάσεις είναι Serializable και η Java εκτέλει εσώτερικα όλες της απαραίτητες ενέργειες έτσι ώστε να επιτευχθεί το serialization. Στην διεπαφη έχουν οριστεί μέθοδοι που εκτελούν την διαδικασία του serialization της οποίες καλεί ο προγραμματιστής. Αλλά σε οποιαδήποτε κλάση που έχει οριστεί ως Serializable μπορεί να υλοποιηθούν μέθοδοι οι οποίες θεωρούνται μέρος της serialization/deserialization διαδικασίας. Επίσης δίνει τη δυνατότητα στον προγραμματιστή να μην χρησιμοποιήσει την προκαθορισμένη serialization διαδικασία, υλοποιώντας ο ίδιος την δικιά του διεπαφή μέσω της Externalizable διεπαφής η οποία περιέχει δύο ειδικές μεθόδους για την αποθήκευση και την επαναφορά της κατάστασης ενός αντικειμένου.

2.2.4 RDBMS

Το σχεσιάκο σύστημα διαχείρισης βάσεων δεδομένων Relational Database Management System (RDBMS) είναι ένα σύστημα διαχείρισης δεδομένων Database Management System (DBMS) το οποίο βασίζεται στο σχεσιακό μοντέλο. Πρώτη φορά περιγράφτηκε απο τον E. F. Codd στην IBM το 1970. Μία σχεσιακή βάση δεδομένων είναι μία συλλογή απο δεδομένα οργανωμένα σε καλά ορισμένους πίνακες όπου τα δεδομένα μπορούν να προσελαστούν με πολλούς διαφορετικούς τρόπους χωρίς να χρειάζεται αναδιοργάνωση των πινάκων. Οι συσχετίσεις των δεδομένων είναι και αυτές αποθηκεύμενες σε αντίστοιχους πίνακες. Οι πιο γνωστές εμπορικές και ανοιχτού κώδικα open source σχεσιακές βάσεις δεδομένων βασίζονται στο σχεσιακό μοντέλο.

Ο όρος ¨σχεσιακό μοντέλο¨ δημοσιεύτηκε από τον E. F. Codd στο "A Relational Model of Data for Large Shared Data Banks" το 1970. Στη συγκεκριμένη δημοσίευση όρισε την έννοια του σχεσιακού μοντέλου. Οι 12 κανόνες του Codd ορίζουν από τι αποτελείται ένα σχεσιακό μοντέλο. Παρόλα αυτά, οι πρώτες υλοποιήσεις του σχεσιακού μοντέλου δεν επαλήθευαν τους 12 κανόνες του Codd οπότε στην πραγματικότητα ο συγκεκριμένος όρος αναφέρεται σε μια μεγαλύτερη κατηγορία συστημάτων διαχείρισης βάσεων δεδομένων. Οι ελάχιστες απαιτήσεις ενός τέτοιου συστήματος είναι :

- σχεσιακή αναπαράσταση των δεδομένων στον χρήστη, δηλαδή αποθήκευση των δεδομένων σε διατεγμένους πίνακες που αποτελούνται απο γραμμές και στήλες
- παροχή σχεσιακών τελεστών για την επεξεργασία των δεδομένων που διατεταγμένα σε μορφή πινάκων

Η πρώτη υλοποίηση συστήματος σύμφωνα με το σχεσιακό μοντέλο έγινε το 1969 στο πανεπιστήμιου του Michigan και ονομαζόταν Micro DBMS και αντίστοιχα από το επιστημονικό κέντρο της IBM UK στο Peterlee υλοποιήθηκε το IS1 (1970-72) και στη συνέχεια το PRTV (1973-79).

Σήμερα, οι περισσότερες σχεσιακές βάσεις δεδομένων χρησιμοποιούν την SQL σαν γλώσσα επερωτήσεων (query language). Έχουν προταθέι και αναπτυχθεί και άλλες γλώσσες επερωτήσεων όπως η Ingres QUEL απο το Berkeley, οι οποίες χρησιμοποιούντε από εμπορικά συστήματα διαχείρισης. Όμως με την προτυποποίηση της SQL, έχει υοθετηθεί τόσο από εμπορικά όσο και από ανοιχτού κώδικα συστήματα διαχείρισης βάσεων δεδομένων.

Το σχεσιακό σύστημα διαχείρισης δεδομένων μπορεί να χρησιμοποιηθεί για την επιίλυση του προβλήματος της διατήρησης των δεδομένων. Όπως αναφέρθηκε μπορεί να αποθηκεύσει δεδόμενα από κάποια εφαρμογή καθώς επίσης και τις συσχετισείς τους σε πίνακες στην βάση δεδόμενων. Με τους σχεσιακούς τελεστές

που προσφέρει μπορεί να δημιουργήσει και να επεξεργάστει τα δεδομένα που έχουν αποθηκευτεί κατά την προηγούμενη εκτέλεση κάποιας εφαρμογής.

2.2.5 OODBMS

Με τον όρο OODBMS (Object-oriented database management system) αναφερόμαστε στις αντικειμενοστραφείς σχεσιακές βάσεις δεδομένων. Στο μοντέλο των συγκεκριμένων βάσεων δεδομένων οι πληροφορίες αναπαρίστανται με τη μορφή αντικειμένων οπώς και ακριβώς και στις αντικειμενοστραφείς γλώσσες προγραμματισμού.

Κεφάλαιο 3

Σχεδιασμός μιας βιβλιοθήκης για ετερογενή συστήματα

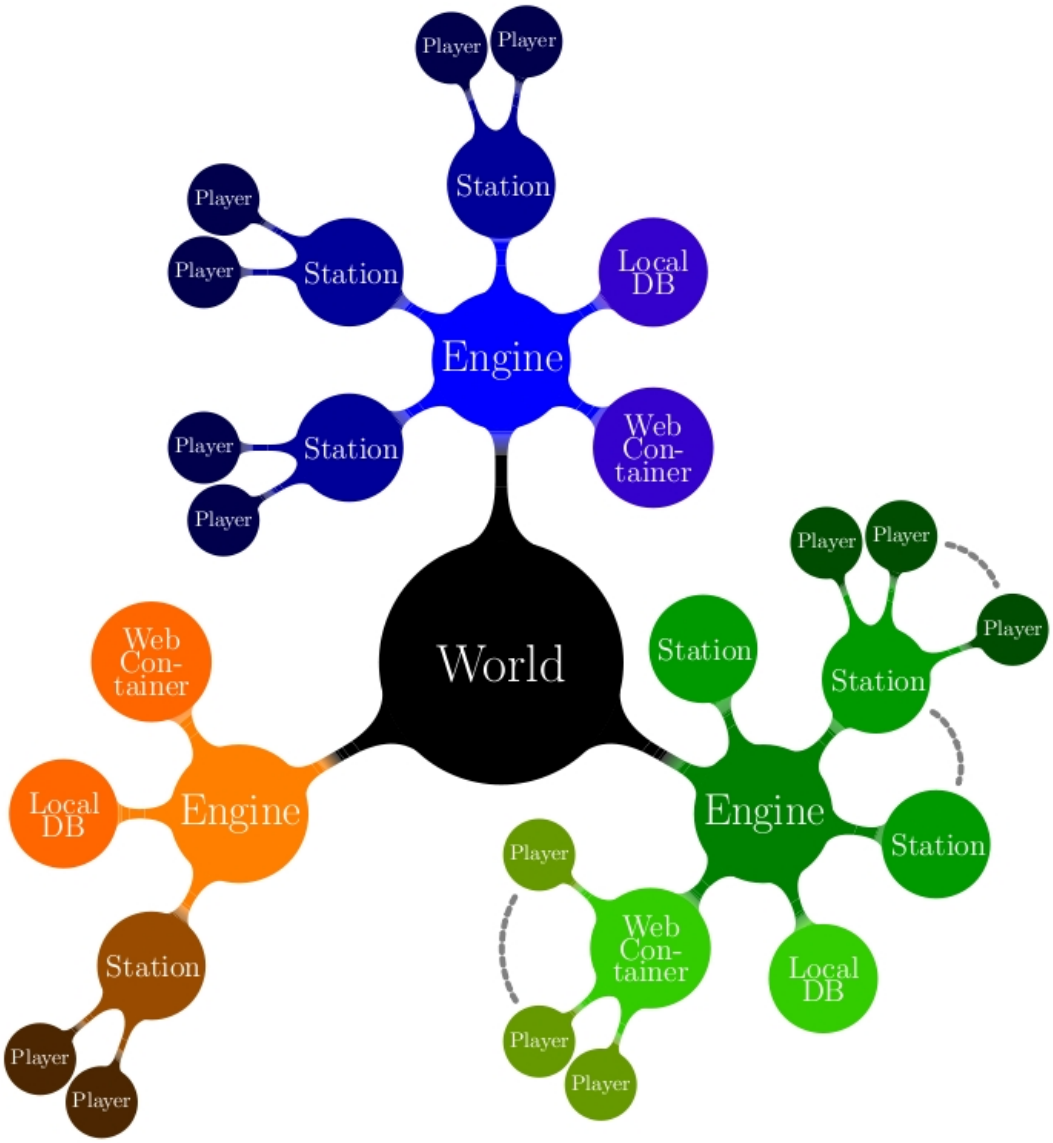
3.1 Εισαγωγή

Η ανάπτυξη της τεχνολογίας και η διάδοση της ευνοεί την δημιουργία ετερογενών συστημάτων. Η χρήση κινητών συσκευών (mobile phones, notebooks, PDAs, netbooks, sensors) γίνεται από ολοένα και αυξανόμενο αριθμό ανθρώπων. Τέτοια -ετερογενή- συστήματα αποτελούνται απο πληθώρα συσκευών, καναλιών και τρόπων επικοινωνίας καθώς επίσης και διαφορετικών αρχιτεκτονικών δομών. Η ανάπτυξη λογισμικού για τέτοια συστήματα παρουσιάζει πολλές προκλήσεις. Η επικοινωνία διαφοροτικών συσκευών μέσω διαφορετικών καναλιών, η μόνιμη αποθήκευση δεδομένων και αντικειμένων καθώς και ένα κοινό μοντέλο αρχιτεκτονικής αποτελούν τις σημαντικότερες απο αυτές.

Μια μεγάλη υποκατηγορία τέτοιων συστημάτων αποτελείται απο κινητά δίκτυα αισθητήρων τα οποία είναι διασυνδεδεμένα με υπολογιστικά συστήματα και αυτά με την σειρά τους είναι συνδεδεμένα στο διαδίκτυο (Internet). Οι εφαρμογές που μπορούν να βασιστούν και να αναπτυχθούν σε τέτοιου είδους συστήματα είναι αρκετές και καλύπτουν διάφορους τομείς. Ένας απο αυτούς είναι και ο τομέας της ψυχαγωγίας ο οποίος βρίσκεται στο επίκεντρο της παρούσας διπλωματικής.

3.2 Αρχιτεκτονική

Στην ενότητα αυτή προτίνεται και αναλύεται η αρχιτεκτονική ενός συστήματος ψυχαγωγίας που περιλαμβάνει κινητά δίκτυα αισθητήρων και υπολογιστικά συστήματα τα οποία είναι συνδεδεμένα μεταξύ τους καθώς επίσης και με το διαδίκτυο. Στο σχήμα 3.1 παρουσιάζεται η συγκεκριμένη αρχιτεκτονική.



Σχήμα 3.1: Αρχιτεκτονική του συστήματος.

Η αρχιτεκτονική του συστήματος βασίζεται σε μια ιεραρχία επιπέδων προσφέροντας επεκτασιμότητα καθώς επίσης και την χρήση διαφορετικών συσκευών (heterogeneity). Έχουν αναπτυχθεί πρωτόκολλα που προσφέρουν χωρικό προσδιορισμό (location awareness) ασυρμάτων συσκευών σε εσωτερικούς χώρους, εκμεταλεύονται τις δυνατότητες των αισθητήρων, συντονίζουν βασικές κατανεμημένες λειουργίες και προσφέρουν επίσης επικοίνωνια ανεκτική σε καθυστερήσεις (delay-tolerant communication). Επίσης στατιστικά συλλέγονται από τις κινητές συσκευές των παικτών, επεξεργάζονται και στη συνέχεια αποθηκεύονται σε μία

κεντρική βάση δεδομένων καθιστώντας δυνατή την προβολή τους σε οποιοδήποτε μελλοντικό χρόνο μέσω μίας ιστοσελίδας.

Η συγκεκριμένη αρχιτεκτονική επιτρέπει την ανάπτυξη παιχνιδιών απόκρισης πραγματικού χρόνου, παιχνιδιών που απαιτούν χωρικό προσδιορισμό, παιχνιδιών που εμπλουτίζουν τον πραγματικό χώρο (augmented reality) καθώς επίσης και παιχνιδιών που δεν απαιτούν σύνδεση με το διαδίκτυο. Ο κύριος στόχος ήταν η ανάπτυξη μίας πλατφόρμας λογισμικού η οποία θα επέτρεπε την εύκολη ενσωμάτωση της σε συστήματα διαφορετικου υλικού και γιαυτό το λόγο η γλώσσα που επιλέχθηκε ήταν η Java. Η Java είναι μια ευρέως διαδεδομένη γλώσσα προγραμματισμού η οποία μπορεί χρησιμοποιηθεί σε διάφορες συσκευές, π.χ. κινητές συσκευές, αισθητήρες, καθώς επισής και σε οποιηδήποτε υπολογιστικό σύστημα. Οι βιβλιοθήκες που προσφέρει επιτρέπουν την διασύνδεση , επικοινωνία των εφαρμογών με βάσεις δεδομένων και την ανταλλαγή πληροφοριών μεταξύ συσκευών μέσω διάφορων καναλιών επικοινωνίας. Επίσης οι εφαρμογές που έχουν αναπτυχθεί σε Java μπορούν να είναι ανεξάρτητες του υλικού των συσκευών στις οποίες θα εκτελούνται. Για την μελέτη της συμπεριφοράς του συστήματος έχουν αναπτυχθεί διάφορα παιχνίδια τα οποία παρουσιάζουν τις δυνατότητες του. Κάθε ένα από αυτά εστιάζει σε διαφορετικά χαρακτηριστικά του συστήματος. Στη συνέχεια αναλύονται όλα τα επίπεδα της αρχιτεκτονικής ξεχωριστά και οι σχεδιαστικές αποφασείς που ελήφθησαν με σκοπό να λύσουν τις βασικές προκλήσεις ενός τέτοιου ετερογενούς συστήματος.

3.2.1 Guardian

Το επίπεδο του Guardian είναι το χαμηλώτερο επίπεδο στην αρχιτεκτονική του συστήματος. Αποτελείται από τις συσκευές που χρησιμοποιούν οι παίκτες κατά την διάρκεια των παιχνιδιών. Οι Guardians είναι το λογισμικό που εκτελείται στις συσκευές των παικτών και χρησιμοποιεί τους αισθητήρες της με σκοπό την δημιουργία μίας διεπαφής για την αλληλεπίδραση με τον χρήστη, την επικοινωνία κτλ. Προσφέρει επίσης ένα πρωτόκολλο για την εύρεση γειτόνων, την επικοινωνία με την υπόλοιπη υποδομή και με τους υπόλοιπους Guardians (echo protocol service). Όταν ανκαλυφθεί ένας νέος Guardian ο παίχτης έχει την δυνατότητα να προβεί σε περαιτέρω ενέργειες, είτε χρησιμοποιώντας τους αισθητήρες είτε τα κουμπιά της συσκεύης. Για την παρακολούθηση της εξέλιξης του παιχνιδιού κάθε ενέργεια η οποία σχετίζεται με το παιχνίδι αναπαρίσταται απο ένα Event.

Στους Guardians επίσης έχει υλοποιηθέι μια υπηρεσία που προσφέρει την αλληλεπίδραση με άλλους Guardians ακόμη και όταν αυτοί είναι αποσυνδεδεμένοι από την υπόλοιπη υποδομή του συστήματος για μεγάλα χρονικά διαστήματα. Συγκεκριμένα, όταν δημιουργέιται ένα νέο Event, ο Guardian το αποθηκεύει τοπίκα στην μνήμη του και μόλις η επικοινωνία με την υπόλοιπη υποδομή είναι δυνατή, όλα τα αποθηκευμένα Events προωθούνται σε ανώτερα επίπεδα της αρχιτεκτονι-

κής (delay tolerant communication service). Επίσης οι Guardians παρέχουν ενα υποσύστημα το οποίο επεξεργάζεται τα δεδομένα που συλλέγονται από το επιταχυνσιόμετρο και ανγνωρίζει τις κινήσεις που έγιναν με την συσκευή (gestures) και τις αντιστοιχεί σε ενέργειες σχετικές με το παιχνίδι. Οι παίκτες δηλαδή κατά την διάρκεια του παιχνιδιού δεν ενεργούν μόνο με το πάτημα κάποιων κουμπιών της συσκευής αλλά κυρίως κάνοντας κινήσεις με την ίδια την συσκευή.

3.2.2 Station

Το επίπεδο των Stations είναι υπεύθυνο για της επικοινωνία και την διασύνδεση των συσκευών που έχουν οι παίκτες με την υπόλοιπη υποδομή του δικτύου του συστήματος και είναι πολύ σημαντικό αλλά όχι απαραίτητο για όλα τα παιχνίδια που έχουν υλοποιηθεί. Προσφέρει υπηρεσίες context awareness και μέσω των Stations τα δεδόμενα μεταφέρονται από και προς τα υψηλότερα επίπεδα της αρχιτεκτονικής με σκοπό τον συντονισμό των παιχνιδιών και την μόνιμη αποθήκευση των δεδομένων. Κάθε Station ελέγχει μία συγκεκρίμενη φυσική περιοχή.

Τα Stations επικοινωνούν με τις συσκευές των χρηστών είτε μέσω τοπικών adhoc δικτύων είτε μέσω personal area non-IP δικτύου και λειτουργούν ως προεπιλεγμένες πύλες, επιτρέπωντας στην ουσία την επικοινωνία των Guardians με το αμέσως υψηλότερο επίπεδο, που είναι το Game Engine. Πολλαπλά Stations μπορούν να είναι συνδεδεμένα σε ένα Game Engine αυξάνοντας με αυτόν τον τρόπο την περιοχή στην οποία μπορούν να εξελιχθούν τα παιχνίδια. Κατά την αρχικοποίηση των παιχνιδιών τα Stations επικοινωνούν με το Game Engine και ανακτούν δεδομένα όπως οι παίκτες που είναι εγγεγραμμένοι για το συγκεκριμένο παιχνίδι το οποίο εξελίσεται, τις συσχετίσεις ανάμεσα στις οντότητες του διαγράμματος συσχέτισης (ER) του σχήματος 3.2 και τους εγγεγραμμένους Guardians. Τα Stations είναι υπεύθυνα για την αρχικοποίηση των Guardians και για την προώθηση όλων των δεδομένων που δημιουργούνται κατά την εξέλιξη ενός παιχνιδιού στο Game Engine.

Υπάρχει επίσης ακόμη μια επιλογή για τον τρόπο λειτουργίας των Stations κατά την διάρκεια ενός παιχνιδιού που δεν σχετίζεται με τα παραπάνω. Σε αυτήν την περίπτωση ονομάζονται mobile Stations και λειτουργούν με διαφορετικό τρόπο. Δεν προσφέρουν επικοινωνία με τα υψηλότερα επίπεδα της αρχιτεκτονίκης παρά μόνο κάλυψη μεγαλυτερής περιοχής και location-aware υπηρεσίες. Ενημερώνουν τους Guardians αποστέλλοντας Events για την mobile λειτουργία τους και αυτοί με την σειρά τους είναι υπεύθυνοι μέσω των σταθερών Stations να ενημερώσουν το Game Engine.

3.2.3 Engine

Κάθε παιχνίδι που αρχικοποιείται και ετοιμαζέται να ξεκινήσει ανατίθεται σε ένα συγκεκριμένο Game Engine από το οποίο συντονίζεται. Το Engine είναι υπεύθυνο στην ουσία για την επιβολή των κανόνων κάθε παιχνιδιού. Με σκοπό την αποφυγή επιπλέον υπολογιστικού κόστους, τα δεδομένα ανάμεσα στο Engine και το υψηλότερο επίπεδο στην ιεραρχία της αρχιτεκτονικής, το World, συγχρονίζονται περιοδικά. Επιπλέον η επεξεργασία και η αποθήκευση των Events που έχουν δημιουργηθεί κατά την διάρκεια του παχνιδιού γίνεται τοπικά σε κάποια βάση δεδομένων.

Επίσης το Engine είναι ένας μηχανισμός ελέγχου που προσφέρει διαφορετικές υπηρεσίες για κάθε παιχνιδί και δίνει την δυνατότητα ανάπτυξης διαφορέτικων σεναρίων σε κάθε παιχνίδι. Η επικοινωνία ανάμεσα στα Stations και το Engine επιτυγχάνεται είτε μέσω ασυρμάτου δικτύου είτε μέσω Ethernet καλωδίου με την χρήση του Internet Protocol (IP).

3.2.4 World

Το υψηλότερο επίπεδο στην ιεραρχία της αρχιτεκτονικής, είναι το World που επιτρέπει την διαχείριση πολλαπλών παιχνιδιών. Το συγκεκριμένο επίπεδο περιλαμβάνει το World Portal, το οποίο είναι το κεντρικό σημείο διαχείρισης του συστήματος, προσφέρωντας αλληλεπίδραση με όλα τα παιχνίδια του πραγματικού κόσμου. Περιλαμβάνει επίσης την κεντρική βάση δεδομένων στην οποία αποθηκεύονται όλα τα δεδομένα που σχετίζονται με τα παιχνίδια όπως στατιστικά που αφορούν τους παίκτες και το ιστορικό παλιότερων παιχνιδιών.

Η επικοινωνία του World με το επίπεδο του Engine γίνεται μέσω της κεντρικής βάσης δεδομένων που υπάρχει στο World. Το Engine γράφει όλα τα δεδομένα απο τα παιχνίδια που εξελίσονται στην βάση δεδομένων και στην συνέχεια τα διαβάζει το World Portal. Αντιστρόφως από World Portal δημιουργούνται καινούρια παιχνίδια και νέοι παίκτες και τα δεδομένα εισάγονται στη βάση δεδομένων και στη συνέχεια μπορούν να χρησιμοποιηθούν από το Engine ανάλογα με το παιχνίδι που εκτελείται. Στο σχήμα 3.2 παρουσιάζεται το διάγραμμα συσχετίσεων.

Στη συνέχεια αναλύεται κάθε οντότητα του σχεσιακού διαγράμματος ξεχωριστά:

- **BattleEngine:** Κάθε στιγμιότυπο ενός παιχνιδιού έχει το δικό του BattleEngine. Το BattleEngine περιέχει πληροφορίες σχετικές με το παιχνίδι, όπως τοποθεσία, μέγιστο αριθμό παικτών, την κατάσταση της εξέλιξης του παιχνιδιού, την ώρα έναρξης και λήξης του παιχνιδίου καθώς επίσης και την IP διεύθυνση του μηχανήματος στο οποίο εκτελείται. Όλες αυτές οι πληροφορίες αρχικοποιούνται πριν την έναρξη του παιχνιδιού από το World Portal.

- **Station:** Σε κάθε BattleEngine έχουν οριστεί τα Stations τα οποία θα είναι

ενεργά κατά την διάρκεια του παιχνιδιού. Η οντότητα Station περιέχει το όνομα του Station, την IP διεύθυνση του μηχανήματος στο οποίο εκτελείται έτσι ώστε να μπορεί να επικοινωνήσει μαζί του το Engine, την τοποθεσία, τις ακριβείς συντεταγμένες κ.α. Κάθε Station δεν μπορεί να είναι ενεργό ταυτόχρονα σε παραπάνω απο ένα Engine.

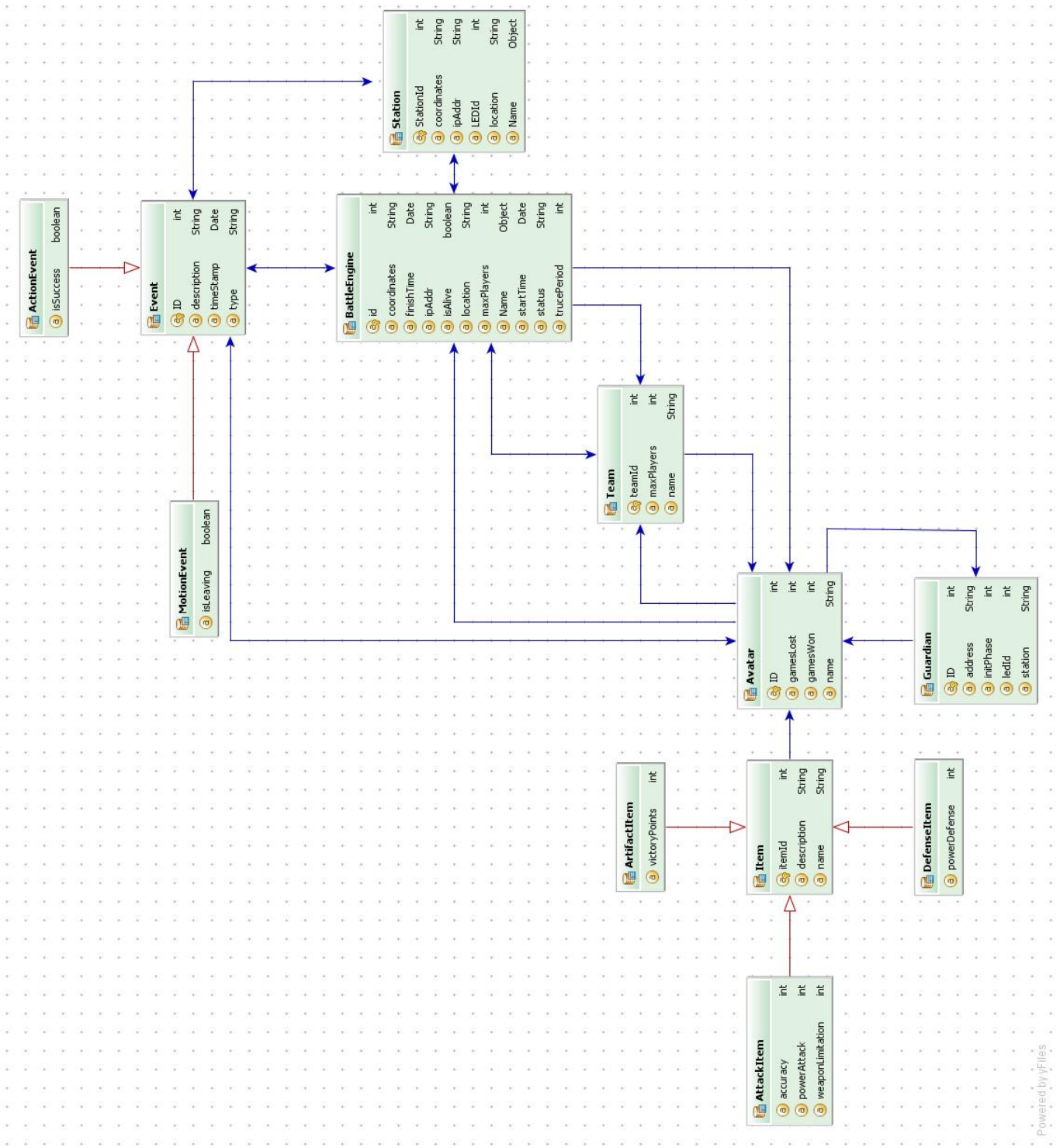
- **Avatar:** Με την εγγραφή ενός νέου παίκτη στο σύστημα δημιουργείται μια νέα εγγραφή στη βάση δεδομένων στον πίνακα Avatar. Η οντότητα Avatar περιέχει πληροφορίες σχετικές με τον παίκτη, είναι αυτή που του επιτρέπει να συμμετάσχει στα παιχνίδια και τον κάνει να ξεχωρίζει από τους άλλους παίκτες. Στον Avatar υπάρχουν και στατιστικά όσο αναφορά τα παιχνίδια που έχει χάσει ή κερδίσει.

- **Team:** Η οντότητα Team δίνει την δυνατότητα ομάδικων παιχνιδιών. Στο Team δηλώνονται ο μέγιστος αριθμός παικτών της ομάδας, το όνομα της καθώς και οι Avatars που ανήκουν στην ομάδα. Πρέπει να αναφερθεί πως οι ομάδες αφορούν στιγμιότυπα παιχνδιών και δεν είναι ίδιες για όλα τα παιχνίδια.

- **Event:** Κάθε κίνηση που σχετίζεται με το παιχνίδι αναπαρίσταται από ένα Event. Περιέχει την περιγραφή του Event, την χρονική στιγμή που έγινε και τον τύπο του. Event. Κάθε Event που δημιουργείται αντιστοιχεί σε ένα συγκεκριμένο Avatar. Υπάρχουν δύο τύποι Event, τα MotionEvent και τα ActionEvent. Τα MotionEvent αφορούν τις κινήσεις των παικτών στο χώρο ενώ τα ActionEvent τις ενέργειες του στο παιχνίδι.

- **Item:** Τα Items είναι τα αντικείμενα που μπορεί να βρει ένας παίκτης κατά την διάρκεια του παιχνιδιού και κάθε ένα από αυτά δίνει διαφορετικές ικανότητες στον παίκτη.

- **Guardian:** Η οντότητα Guardian αφορά την συσκευή που χρησιμοποιεί ο παίκτης κατά την διάρκεια ενός παιχνιδιού. Κάθε Avatar μπορεί να χρησιμοποιεί μόνο ένα Guardian σε κάθε παιχνίδι. Το ποιος Guardian αντιστοιχεί σε κάθε Avatar γίνεται μέσω του World Portal πριν την έναρξη του παιχνιδιού. Ο Guardian περιέχει την διεύθυνση της συσκευής (MAC address), πληροφορίες για το εάν εχει αρχικοποιηθεί η συσκευή (initPhase) και το Station στο οποίο μπορεί να βρίσκεται συνδεδεμένος ο Guardian.



Σχήμα 3.2: Σχεσιακό διάγραμμα.

Κεφάλαιο 4

Υλοποίηση

4.1 Εισαγωγή

Η αρχιτεκτονική που περιγράφηκε παραπάνω έχει υλοποιηθεί σε πραγματικό σύστημα. Οι τεχνολογίες που χρησιμοποιήθηκαν ανά επίπεδο είναι οι εξής :

- **Engine:** Το επίπεδο του Engine έχει υλοποιηθεί με Java 1.6, Standard Edition (Java SE). Η βάση δεδομένων είναι υλοποιημένη σε MySQL v5.0.51a ενώ για την επικοινωνία και την αλληλεπίδραση με την βάση δεδομένων χρησιμοποιήθηκε η Hibernate η οποία όπως αναφέρθηκε είναι μια object-relational mapping (ORM) βιβλιοθήκη για την Java. Η υλοποίηση του Engine εκτελείται είτε σε σταθερό υπολογιστή είτε σε Laptop.

- **Station:** Το επίπεδο του Station έχει υλοποιηθεί και αυτό με Java 1.6, Standard Edition (Java SE). Για την επικοινωνία με τα Guardians χρησιμοποιήθηκαν τα SunSPOT base station και η επικοινωνία γίνετε μέσω του IEEE 802.15.4 standard. Η επικοινωνία ανάμεσα στο Station και το Engine επιτυγχάνεται με την χρήση της Remote Method Invocation (RMI). Επίσης για την αποστολή των αντικειμένων Events ανάμεσα στα Station και Engine χρησιμοιποιείται το Java Serialization API. Το Station εκτελείται σε Alix Gateways.

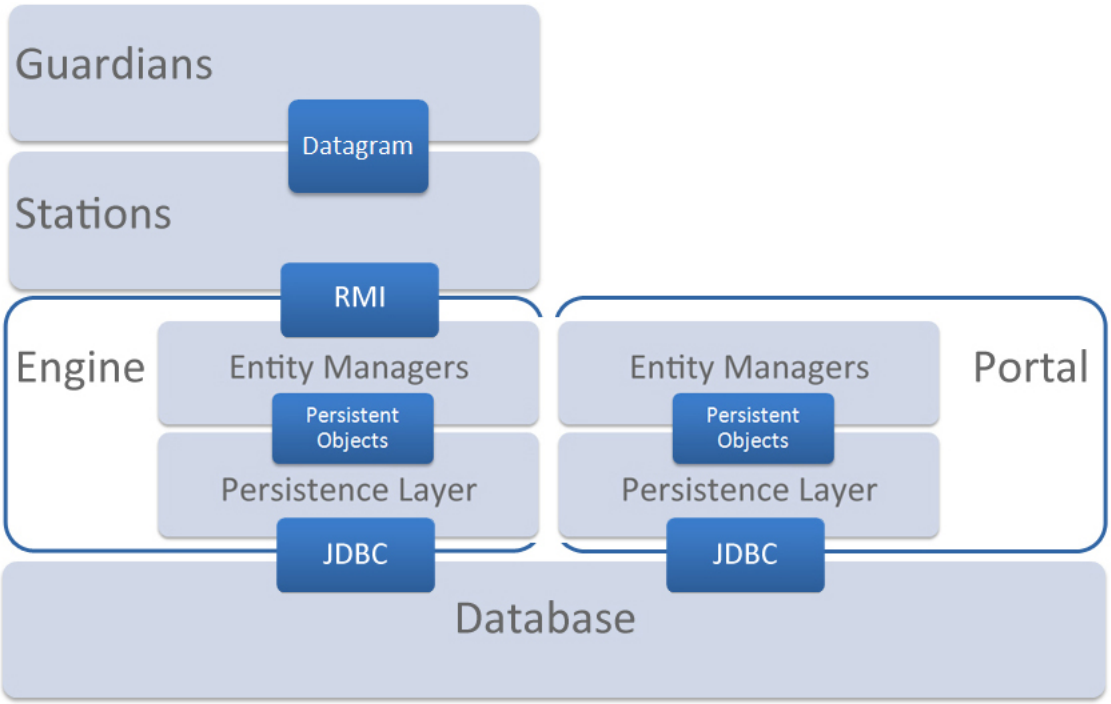
- **Guardian:** Οι συσκευές που χρησιμοποιήθηκαν στο συγκεκριμένο επίπεδο είναι τα SunSPOTs. Τα SunSPOTs κατασκευάζονται απο την Sun Microsystems. Πρόκειται για μία μικρή συσκευή που λειτουργεί με μπαταρία και προσφέρει μια πολύ απλή διεπαφή για την αλληλεπίδραση με τον χρήστη η οποία περιλαμβάνει 2 κουμπία και 8 LEDs. Οι αισθητήρες που περιλαμβάνει είναι αξελερόμετρο, αισθητήρας φωτός και θερμοκρασίας. Κάθε SunSPOT χρησιμοποιεί έναν 180MHz 32-bit ARM920T core processor με

512K PAM και 4M Flash. Τα SunSPOTs προγραμματίζονται σε Java Micro Edition (J2ME). Στην εικόνα 4.1 παρουσιάζεται ένα SunSPOT όπου μπορεί να καταλάβει κανείς και το μέγεθος της συσκευής.

Σχήμα 4.1: SunSPOT device.

- **World:** Στο επίπεδο του World έχει υλοποιηθεί το Portal το οποίο επιτρέπει την δημιουργία, την διαχείριση και την παρακολούθηση των παιχνιδιών. Το Portal έχει υλοποιηθεί σε Liferay 5.2 και εκτελείται σε έναν Tomcat 6.0 Server. Τα δεδομένα που απαιτούνται για την υλοποίηση και την εκτέλεση του Portal αποθηκεύονται σε μία HSQL βάση δεδομένων. Η επικοινωνία με την κεντρική βάση δεδομένων επιτυγχάνεται μέσω της Hibernate.

Όλα τα παραπάνω χαρακτηριστικά μαζί με τις τεχνολογίες υλοποιήσης και τους τρόπου επικοινωνίας ανάμεσα στα διαφορετικά επίπεδα, συνοψίζονται στην εικόνα 4.2.



Σχήμα 4.2: Αρχιτεκτονική συστήματος και επικοινωνία ανάμεσα σε όλα τα επίπεδα.

4.1.1 Cross-layer Object Persistence and Communication

Το κύριο χαρακτηριστικό του συστήματος είναι η δυνατότητα επικοινωνίας ανάμεσα στα διαφορετικά επίπεδα του συστήματος. Η επικοινωνία ανάμεσα στην κεντρική βάση δεδομένων και στο Engine γίνεται μέ την χρήση της Hibernate, ανάμεσα στο Station και στο Engine με την χρήση της RMI και ανάμεσα σε Guardian και Station με Radiograms. Τα Radiograms είναι τα μηνύματα επικοινωνίας μέσω 802.15.4 που χρησιμοποιούν τα SunSPOTs.

Η επιλογή διαφορετικών τεχνολογιών υλοποίησης σε κάθε επίπεδο δημιουργεί διάφορα θέματα συμβατότητας. Το πρόβλημα δημιουργείται λόγω των περιορισμένων λειτουργιών που προσφέρει η Java Micro Edition (J2ME) και το SunSPOT sdk σε σχέση με την Java Standard Edition. Μία απλή λύση θα ήταν η διαφορετική αναπαράσταση των οντοτήτων σε κάθε επίπεδο. Πράγμα που θα απαιτούσε την συνεχή μετατροπή των δεδομένων μεταξύ των διαφορετικών μοντέλων κάθε επιπέδου έχοντας ως αποτέλεσμα σημαντικη αύξηση του υπολογιστικού κόστους. Για παράδειγμα εάν η οντότητα Avatar οριζόταν διαφορετικά σε κάθε επίπεδο, τοτε θα απαιτούνταν συνεχείς μετατροπές του Avatar καθώς θα μεταφερόταν από την βάση δεδομένων στον αντίστοιχο Guardian. Για να αποφευχθεί το συγκεκριμένο υπολογιστικό κόστος αποφασίστηκε η χρήση ενός κοινού μοντέλου οντοτήτων για

όλα τα επίπεδα πράγμα που μειώνει της μετατροπές και αυξάνει την ταχύτητα της επικοινωνίας.

Στο κοινό μοντέλο οντοτήτων το μεγαλύτερο πρόβλημα είναι η υλοποίηση των επιπέδων σε διαφορετική έκδοση της Java. Συγκεκριμένα χρησιμοποιήθηκε η Java Standard Edition (J2SE) μαζί με την Hibernate για το Engine και το Station και η Java Micro Edition (J2ME) για τους Guardians οι οποίοι εκτελούνται στα SunSPOT.

Συγκεκριμένα το πρόβλημα με το κοινό μοντέλο οντοτήτων εμφανίζεται στην αλληλεπίδραση ανάμεσα στους Guardians και την βάση δεδομένων στην Hibernate. Στην Hibernate για να δηλωθούν οι την many-to-many συσχετίσεις απαιτείται η χρήση των κλάσεων `java.util.Set` και `java.util.HashSet` οι οποίες δεν είναι υλοποιημένες στην J2ME. Επιπρόσθετα η κλάση `HashSet` χρησιμοποιεί την `Serializable` η οποία δεν υποστηρίζεται πλήρως από την J2ME.

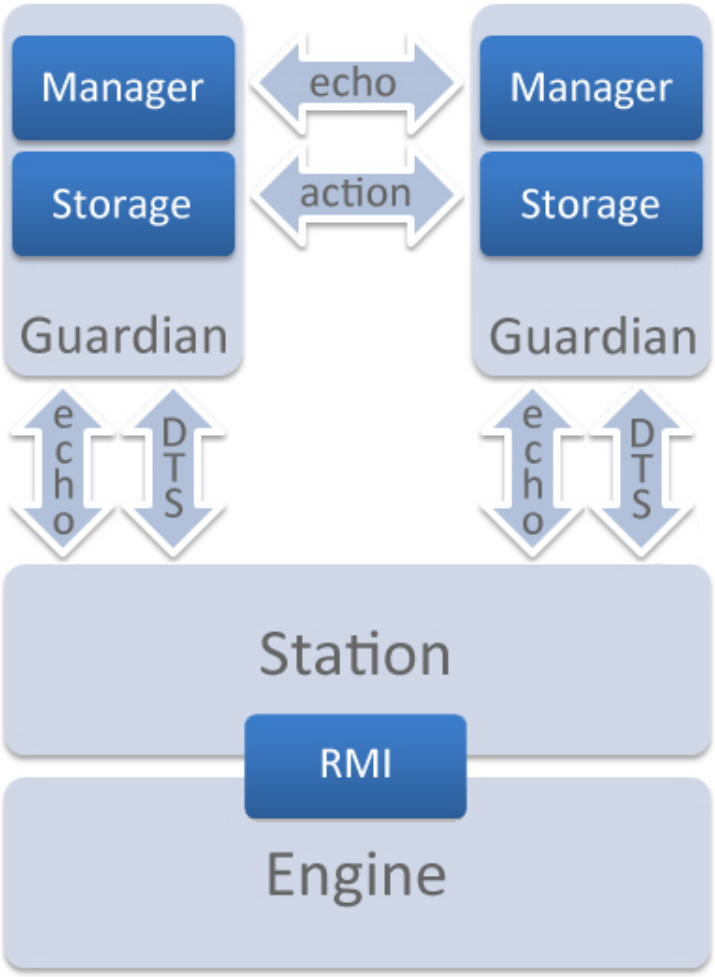
Για να ξεπεραστούν τα συγκεκριμένα προβλήματα διατηρώντας το υψηλότερο επίπεδο integration καθώς επίσης και την επαναχρησιμοποίηση του κώδικα ανάμεσα σε όλα τα επίπεδα, δημιουργήθηκαν κάποια ειδικά Ant scripts που κάτα το compilation αντικαθιστούν όλα τα μη υποστηριζόμενα αντικείμενα με υποστηριζόμενα αντικείμενα χαμηλώτερων επιπέδων. Συγκεκριμένα τα Java αντικείμενα τύπου java.util.Set αντικαθύστανται από τα java.Object σκοπεύοντας στον περιορισμό της κατανάλωσης της μνήμης καθώς η κλάση java.Object ανήκει στην κορυφή της ιεραρχίας κλάσεων της Java και τα αντικείμενα της απαιτούν την λιγότερη δυνατή μνήμη. Οι συναρτήσεις οι οποίες χρησιμοποιούσαν τα συγκεκριμένα αντικείμενα-μεταβλητές που αντικαταστάθηκαν δεν μπορούν πλέον να χρησιμοποιηθουν αφού το Ant script τις βάζει σε σχόλια (Comment out). Η συγκεκριμένη λύση δεν δημιουργεί κανέναν πρόβλημα στα χαμηλώτερα επίπεδα (π.χ Guardian Layer) καθώς οι Guardians έχουν ελάχιστη γνώση σε σχέση με το τι γίνεται στο επίπεδο του World και δεν απαιτείται να ξέρουν τις συσχετίσεις μεταξύ των διαφόρων αντικειμένων πόσο μάλλον των συσχετίσεων που αφορούν τους πίνακες της κεντρικής βάσης δεδομένων.

Στην συνέχεια της συγκεκριμένης ενότητας παρουσιάζεται ο τρόπος με τον οποιο επιτυγχάνεται η επικοινωνία ανάμεσα σε όλα τα επίπεδα.

- **Guardian:** Το συγκεκριμένο επίπεδο παρέχει κάποιες συγκεκριμένες υπηρεσίες και πρωτόκολλα που είναι απαραίτητα για την αναγνώριση και τη επικοινωνία με τις υπόλοιπες συσκευές καθώς επίσης και με την υποδομή. Συγκεκριμένα υπάρχουν τα:

  Echo Protocol: Το συγκεκριμένο πρωτόκολλο παρέχει λειτουργίες οι οποίες ειναι υπεύθυνες για την γρήγορη και αξιόπιστη εύρεση των γειτόνων μιας συσκευής.

  Action Protocol: Είναι υπεύθυνο για τον κατανεμημένο συντονισμό και επιτρέπει την αλληλεπίδραση ανάμεσα στους παίκτες.



Σχήμα 4.3: Επικοινωνία ανάμεσα σε όλα τα επίπεδα.

Delay Tolerant Service (DTS): Η συγκεκριμένη υπηρεσία επιτρέπει την επικοινωνία μεταξύ των Guardians και των Stations και την αποστολή των Events ακόμη και όταν δεν υπάρχει απευθείας επικοινωνία με τα Stations. Αποθηκεύει τα Events στο Storage των Guardians και μόλις κάποιος Guardian συνδεθεί σε κάποιο Station στέλνει όλα τα αποθηκευμένα Events ενημερώντας τα ανώτερα επίπεδα για την εξέλιξη του παιχνιδιού.

Τα πρωτόκολλα έχουν αναπτυχθεί σε Java Micro Edition (J2ME) και εκτελούνται στα SunSPOTs. Το SunSpot's sdk προσφέρει δύο ξεχωριστά πρωτόκολλα επικοινωνίας, το RadioStream και το Radiogram. Αποφασίστηκε και χρησιμοποιήθηκε το Radiogram λόγω του οτι προσφέρει αξιόπιστη επικοινωνία ανάμεσα στα SunSPOTs καθώς χρησιμοποιεί μεθόδους που προσφέρουν acknowledging και η αποστολή των δεδομένων επιτυγχάνεται μέσο data-

grams. Τα συγκεκριμένα πρωτόκολλα χρησιμοποιούνται επίσης και για την επικοινωνία των Guardians με τα Stations.

- **Station:**

  Echo Protocol: Πρόκειται για το ίδιο πρωτόκολλο το οποίο εκτελείται και στους Guardians και προσφέρει γρήγορο και αξιόπιστο εντοπισμό των Guardians οι οποίοι εισέρχονται στην εμβέλεια του Station.

  Delay Tolerant Service (DTS): Η συγκεκριμένη υπηρέσια είναι παρόμοια με τον DTS των Guardians με την μόνη διαφορά οτι είναι υλοποιημένη σε J2SE με την χρήση Blocking Queues προσφέρωντας περισσότερες λειτουργίες. Οι λειτουργικότητα και ο τρόπος υλοποίησης της υπηρεσίας αναλύονται στην συνέχεια του κεφαλαίου.

  RMI Interface: Τα Stations σε κάθε παιχνίδι παρέχουν ένα RMI Interface δίνοντας έτσι την δυνατότητα επικοινωνίας του Engine με αυτά. Υπάρχει ένα Abstract RMI Interface που προσφέρει τις βασικές μεθόδους που είναι απαραίτητες για την λειτουργία και την επικοινωνία του συστήματος και δίνει την δυνατότητα στον προγραμματιστή να υλοποιήσει το δικό του RMI Interface το οποίο κληρονομεί το Abstract εφόσον οι λειτουργίες τους Abstract δεν τον καλύπτουν.

- **Engine:**

  RMI Server, Interface: Η επικοινωνία ανάμεσα σε Engine και Station γίνεται μέσω RMI. Επομένως έχει υλοποιηθεί ένας RMI Server στον οποίο γίνεται register το Engine RMI Interface κατά την εκτέλεση του Engine. Μόλις εκτελεστεί το Station καλεί το Engine RMI Interface στέλνοντας μαζί και το δικό του RMI Interface με σκοπό να γίνει registered στο Engine. Για να ολοκληρωθεί η διαδικασία του registration το Engine αποθηκεύει το RMI Interface που έλαβε απο το Station έτσι ώστε να μπορεί να το καλέσει και να επικοινωνήσει μαζί του όποτε αυτο χρειαστεί. Υπάρχει δηλαδή αμφίδρομη επικοινωνία ανάμεσα στα registered Stations και στο Engine.

  Entity Managers: Η επικοινωνία και η αποθήκευση των δεδομένων στην κεντρική βάση δεδομένων επιτυγχάνεται με την χρήση της Hibernate. Για κάθε Entity του σχεσιακού διαγράμματος έχει υλοποιηθεί ο αντίστοιχος Entity Manager μέσο του οποίου μπορεί να γίνει ανάκτηση, εγγραφή και ανανέωση των δεδομένων στην βάση του αντίστοιχου Entity. Επομένως η επικοινωνία Engine και βάσης δεδομένων επιτυχγάνεται με την χρήση των Entity Managers.

- **World:**

Entity Managers: Το World επικοινωνεί μόνο με την κεντρική βάση δεδομένων. Ο τρόπος επικοινωνίας είναι ίδιος με αυτόν του Engine και της κεντρικής βάσης δεδομένων και επιτυγχάνεται μέσο των Entity Managers οι οποίοι αναφέρθηκαν προηγουμένος και ειναι κοινοί για το World και για το Engine.

Στην συνέχεια του κεφαλαίου υπαρχεί ξεχωριστή ενότητα για κάθε επίπεδο όπου αναλύεται και παρουσιάζεται ο τρόπος με τον οποίο υλοποιήθηκε κάθε επίπεδο της αρχιτεκτονικής του συστήματος.

4.2 Engine

Το Engine έχει υλοποιηθεί σε Java Standard Edition έκδοση 1.6. Αποτελείται από τους Entity Managers που είναι υπεύθυνοι για την επικοινωνία και την αλληλεπίδραση με την βάση δεδομένων καθώς επίσης και τον RMI Server οποίος χρησιμοποιείται για την επικοινωνία με τα κατώτερα επίπεδα της αρχιτεκτονικής του συστήματος. Στην συνέχεια αναλύονται τα δύο αυτά υποσυστήματα το καθένα ξεχωριστά :

- **Entity Managers:** Για κάθε Entity Class του σχεσιακού διαγράμματος υπάρχει ο αντίστοιχος Entity Manager. Έχει υλοποιηθεί ένας Abstract Manager ο οποιος προσφέρει τις βασικές Create, read, update and delete (CRUD) λειτουργίες του persistent storage. Συγκεκριμένα χρησιμοποιώντας τα Generics που προσφέρει η J2SE 1.6 έχουν υλοποιηθεί οι εξής συναρτήσεις :

  `public void add(final E entity)`: Η συγκεκριμένη συνάρτηση προσθέτει μια νέα εγγραφη στη βάση δεδομένων.

  `public void update(final E entity)` : Η update ανανεώνει την εγγραφή που αντιστοιχεί στο αντικείμενο που δεν σαν όρισμα στη βάση δεδομένων.

  `public void delete(final E entity, final int entityID)`: Δέχεται σαν όρισμα ένα αντικειμένο καθώς επίσης και το ID που έχει η αντίστοιχη εγγραφή στη βάση δεδομένων και το διαγράφει.

  `protected List<E> list(final E entity)`: Επιστρέφει μία λίστα με όλες τις εγγραφές της κλασής στην οποία ανήκει το αντικείμενο που δέχεται σαν όρισμα.

  `protected E getByID(final E entity, final int entityID)`: Η συγκεκριμένη συνάρτηση επιστρεφεί την εγγραφή που ανήκει στην κλάση που ανήκει το αντικείμενο entity και έχει ID = entityID.

Στην συνέχεια παρατίθεται ο κώδικας που υλοποιεί τον Abstract Manager.

Listing 4.1: AbstractManager.java

```
1  package db.managers;
2
3  import db.util.HibernateUtil;
4  import org.hibernate.Criteria;
5  import org.hibernate.Session;
6
7  import java.util.List;
8
9  /**
10  * This class implements an abstract manager which is responsible for all the basic
11  * CRUD(insert,select,update,delete) functions in the database. Other sub-classes
12  * inherit from this one. This means that every sub-class can use all the methods
13  * that are implemented here and consequently there is no need of defining the CRUD
14  * functions in every class. Templates are used here as a generic to all the class
15  * types that are defined inside the logic package.
16  *
17  * @param <E> Generic type of AbstractManager
18  */
19 public abstract class AbstractManager<E> { // NOPMD
20
21     /**
22      * adding a new entry into the database, according to the input object it receives.
23      *
24      * @param entity an Entity object that may be of every type of entity.
25      */
26     public void add(final E entity) {
27         final Session session = HibernateUtil.getInstance().getSession();
28         session.save(entity);
29     }
30
31     /**
32      * updating an entry into the database, according to the input object it receives.
33      *
34      * @param entity an Entity object that may be of every type of entity.
35      */
36     public void update(final E entity) {
37         final Session session = HibernateUtil.getInstance().getSession();
38         session.merge(entity);
39     }
40
41     /**
42      * deleting an entry into the database, according to the input object it receives.
43      *
44      * @param entity   an Entity object that may be of every type of entity.
45      * @param entityID the id of the Entity object.
46      */
47     public void delete(final E entity, final int entityID) {
48         final Session session = HibernateUtil.getInstance().getSession();
49         final Object entity2 = session.load(entity.getClass(), entityID);
50         session.delete(entity2);
51     }
52
53     /**
54      * listing all the entries from the database related to the input object it receives.
55      *
56      * @param entity an Entity object that may be of every type of entity.
57      * @return a list of all the records(objects) that exist inside the table related to
58      * the input Entity object.
```

```
59      */
60     @SuppressWarnings("unchecked")
61     protected List<E> list(final E entity) {
62         final Session session = HibernateUtil.getInstance().getSession();
63         final Criteria criteria = session.createCriteria(entity.getClass());
64         return (List<E>) criteria.list();
65
66     }
67
68     /**
69      * get the entry from the database that corresponds to the input id.
70      *
71      * @param entity   an Entity object that may be of every type of entity.
72      * @param entityID the id of the Entity object.
73      * @return an Entity object.
74      */
75     @SuppressWarnings("unchecked")
76     protected E getByID(final E entity, final int entityID) {
77         final Session session = HibernateUtil.getInstance().getSession();
78         return (E) session.get(entity.getClass(), entityID);
79     }
80
81 }
```

Κάθε Manager που αντιστοιχεί σε οποιδήποτε Entity Class του σχεσιακού διαγράμματος χρειάζεται τις συναρτήσεις που έχουν υλοποιηθεί από τον Abstract Manager. Επομένως αντί να υλοποιηθούν ξάνα σε κάθε Manager ξεχωριστά, κληρονομούν τον Abstract Manager και υλοποιούν μόνο περαιτέρω συναρτήσεις εφόσον αυτό κρίνεται απαραίτητο.

Στην συνέχεια παρουσιάζεται Manager που αντιστοιχεί στην Entity Class του BattleEngine.

Listing 4.2: BattleEngineManager.java

```
1  package db.managers;
2
3  import db.model.BattleEngine;
4  import db.util.HibernateUtil;
5  import org.hibernate.Query;
6  import org.hibernate.Session;
7
8  import java.util.List;
9
10
11 /**
12  * This class is responsible for all the basic CRUD functions in the database
13  * regarding BattleEngine objects. It inherits the AbstractManager class
14  * and inside also exist an other function:
15  * getByIp(String) Returns a battleEngine accorind to the input ip address.
16  */
17 public final class BattleEngineManager extends AbstractManager<BattleEngine> {
18     /**
19      * static instance(ourInstance) initialized as null.
20      */
21     private static BattleEngineManager ourInstance = null;
22
23     /**
```

```
24      * Private constructor suppresses generation of a (public) default constructor.
25      */
26     private BattleEngineManager() {
27         // Does nothing
28     }
29
30     /**
31      * BattleEngineManager is loaded on the first execution of
32      * BattleEngineManager.getInstance() or the first access to
33      * BattleEngineManager.ourInstance, not before.
34      *
35      * @return ourInstance
36      */
37
38     public static BattleEngineManager getInstance() {
39         synchronized (BattleEngineManager.class) {
40             if (ourInstance == null) {
41                 ourInstance = new BattleEngineManager();
42             }
43         }
44
45         return ourInstance;
46     }
47
48
49     /**
50      * get the battleEngine from the database that corresponds to the input id.
51      *
52      * @param entityID the id of the Entity object.
53      * @return a Battle Engine object.
54      */
55
56     public BattleEngine getByID(final int entityID) {
57         return super.getByID(new BattleEngine(), entityID);
58     }
59
60
61     /**
62      * deleting the input battleEngine from the database.
63      *
64      * @param battleEngine the object that we want to delete
65      */
66
67     public void delete(final BattleEngine battleEngine) {
68         super.delete(battleEngine, battleEngine.getId());
69     }
70
71     /**
72      * listing all the battleEngines from the database.
73      *
74      * @return a list of all the records(objects) that exist inside the
75      * table related to the input Entity object.
76      */
77     public List<BattleEngine> list() {
78         return super.list(new BattleEngine());
79     }
80
81     /**
82      * get the battleEngine from the database according to the ipnut ip address.
83      *
84      * @param battleIp the ip of the battlEngine
85      * @return a Battle Engine object.
```

```
86      */
87     public BattleEngine getByIp(final String battleIp) {
88         final Session session = HibernateUtil.getInstance().getSession();
89 
90         final String stringQuery
91                 = "from BattleEngine battleEngine where battleEngine.ipAddr = :battle";
92         final Query query = session.createQuery(stringQuery)
93                 .setParameter("battle", battleIp);
94 
95         return (BattleEngine) query.list().get(0);
96     }
97 
98     /**
99      * get the battleEngine from the database according to the ipnut ip address.
100     *
101     * @param battleIp the ip of the battlEngine
102     * @return a Battle Engine object.
103     */
104    public BattleEngine getAliveByIp(final String battleIp) {
105        final Session session = HibernateUtil.getInstance().getSession();
106 
107        final String stringQuery
108                = "from BattleEngine battleEngine where battleEngine.isAlive = 1 "
109                        + " AND battleEngine.ipAddr = :battle";
110        final Query query = session.createQuery(stringQuery)
111                .setParameter("battle", battleIp);
112 
113        return (BattleEngine) query.list().get(0);
114    }
115 
116    /**
117     * Change the status to Ready and set isAlive true to the battleEngine it receives.
118     *
119     * @param battleEngine The BattleEngine object
120     */
121    public void startGame(final BattleEngine battleEngine) {
122        battleEngine.setStatus("Ready");
123        battleEngine.setIsAlive(true);
124        BattleEngineManager.getInstance().update(battleEngine);
125 
126 
127    }
128 
129    /**
130     * Change the status and set isAlive false according to the battleEngine it
131     * receives and call the releaseUnusedGuardians from GuardianManager.
132     *
133     * @param battleEngine the battleEngine
134     */
135    public void endGame(final BattleEngine battleEngine) {
136        battleEngine.setIsAlive(false);
137        battleEngine.setStatus("Finished");
138        update(battleEngine);
139        //svGuardianManager.getInstance().releaseUnusedGuardians(battleEngine);
140    }
141 
142    /**
143     * Get the ID of the points of interest of the battle engine.
144     *
145     * @param battleID An int with the battle engine ID
146     * @return a list with the points of interest
147     */
```

```
148     @SuppressWarnings("unchecked")
149     public List<Integer> getPointsOfInterest(final int battleID) {
150         final Session session = HibernateUtil.getInstance().getSession();
151 
152         final String stringQuery = "SELECT STATION_ID FROM STATION "
153                 + " WHERE BATTLE_ID = :battleID  ";
154 
155         final Query query = session.createSQLQuery(stringQuery)
156                 .setParameter("battleID", battleID);
157 
158         return query.list();
159     }
160 
161 }
```

Κάθε Manager δοκιμάστηκε χρησιμοποιώντας έλεγχο μονάδας. Για αυτό το λόγο χρησιμοποιήθηκε το περιβάλλον εργασίας Junit. Στην συνέχεια παρατίθεται ο κώδικας υλοποιήσης που ελέγχει την σωστή λειτουργία του BattleEngineManager όπου παρουσιάζεται και ο τρόπος με τον οποίο μπορούν να χρησιμοποιηθούν οι Managers και οι συναρτήσεις που παρέχουν.

Ακολουθεί ο κώδικας του BattleEngineTester.java.

Listing 4.3: BattleEngineTester.java

```
1  package db.test.battleEngine;
2  
3  import db.managers.BattleEngineManager;
4  import db.model.BattleEngine;
5  import db.util.HibernateUtil;
6  import junit.framework.Assert;
7  import junit.framework.TestCase;
8  import org.hibernate.Transaction;
9  import util.Logger;
10 
11 import java.util.Iterator;
12 import java.util.List;
13 
14 /**
15  * This is a junit test for the BattleEngine class / table.
16  */
17 public class BattleEngineTester extends TestCase {
18 
19     /**
20      * Setup the Tester.
21      */
22     protected void setUp() {
23         // no setup is necessary
24     }
25 
26 
27     public void testListRecords() {
28         // Start new transaction
29         final Transaction tranx =
30                         HibernateUtil.getInstance().getSession().beginTransaction();
31 
32         final List<BattleEngine> list = BattleEngineManager.getInstance().list();
33         Iterator<BattleEngine> iter = list.iterator();
34         while (iter.hasNext()) {
```

```
35              BattleEngine battleEngine = iter.next();
36              Logger.getInstance().debug("BattlEngine name :"
37                  + battleEngine.getName() + " Battle Engine ID :" + battleEngine.getId());
38          }
39
40          // Commit Hibernate Transaction
41          tranx.commit();
42      }
43
44      /**
45       * Test Add functionality.
46       */
47      public void testAddRecord() {
48          // Start new transaction
49          final Transaction tranx =
50                          HibernateUtil.getInstance().getSession().beginTransaction();
51
52          // Construct new Engine
53          final BattleEngine battleEngine = new BattleEngine();
54          battleEngine.setIpAddr("127.0.0.1");
55          battleEngine.setIsAlive(false);
56
57          //Save the record
58          BattleEngineManager.getInstance().add(battleEngine);
59
60          // Check if record was added -- primary key should be assigned a value
61          Assert.assertTrue(battleEngine.getId() >= 0);
62
63          // Commit Hibernate Transaction
64          tranx.commit();
65      }
66
67      /**
68       * Test Get functionality.
69       */
70      public void testGetRecord() {
71          // Start new transaction
72          final Transaction tranx =
73                          HibernateUtil.getInstance().getSession().beginTransaction();
74
75          // Construct new MonkEngine
76          BattleEngine battleEngine;
77
78          // Try to get
79          battleEngine = BattleEngineManager.getInstance().getByID(1);
80
81          // Check if record was retrieved - primary key should be assigned the proper value
82          Assert.assertTrue(battleEngine.getId() == 1);
83
84          // Commit Hibernate Transaction
85          tranx.commit();
86      }
87
88      /**
89       * Test List functionality.
90       */
91      public void testList() {
92          // Start new transaction
93          final Transaction tranx =
94                          HibernateUtil.getInstance().getSession().beginTransaction();
95
96          // Try to save
```

```
97          final List<BattleEngine> list = BattleEngineManager.getInstance().list();
98
99          // Check if record was added -- primary key should be assigned a value
100         Assert.assertTrue(!list.isEmpty());
101
102         // Commit Hibernate Transaction
103         tranx.commit();
104     }
105
106     /**
107      * Test Delete functionality.
108      */
109     public void testDeleteRecord() {
110         // Start new transaction
111         final Transaction tranx =
112                         HibernateUtil.getInstance().getSession().beginTransaction();
113
114         // Construct new MonkEngine
115         final BattleEngine battleEngine = new BattleEngine();
116         battleEngine.setId(1);
117
118         // Delete record ID = 1
119         BattleEngineManager.getInstance().delete(battleEngine);
120
121
122         try {
123             // Try to load battleEngine ID = 1
124             BattleEngineManager.getInstance().getByID(1);
125         } catch (Exception e) {
126             if (!(e.getMessage().equals(
127                 "No row with the given identifier exists: [db.model.BattleEngine#1]"))) {
128                 Assert.fail("Failed to delete entity");
129             }
130         }
131
132         // Commit Hibernate Transaction
133         tranx.commit();
134     }
135
136 }
```

Αντίστοιχοι Managers έχουν υλοποιηθεί για όλες τις Entity Classes του μοντέλου της αρχιτεκτονικής.

- **RMI:** Η επικοινωνία του Engine με το Station επιτυγχάνεται με την χρήση της RMI. Η RMI για να λειτουργήσει χρειάζεται ένα Interface και το Implementation του Initerface. Έχει υλοποιηθεί το BaseEngineInterface το οποίο παρέχει την βασική συνάρτηση που μπορεί να κληθεί απομακρυσμένα απο το Station με σκοπό την αποστολή Event και είναι η addEvent(Event event). Εάν απαιτούνται και άλλες συναρτήσεις υλοποιείται καινούριο Interface το οποίο κληρονομεί το BaseEngineInterface και έχει δικό του καινούριο Implementation.

  Έχει υλοποιηθεί επίσης και ο GenericRMIServer ο οποίος καλείται κατά την εκκίνηση και στον οποίο δηλώνεται το ποιό RMI Interface απαιτείται να εκτε-

λεστεί. Στην συνέχεια ακολουθεί ενα παράδειγμα. Παρατίθεται ο κώδικας με τον οποίο υλοποιείται το SGEngineInterface και το SGEngineRMIIMpl. καθώς και ο τρόπος με τον οποίο χρησιμοποιείται ο GenericRMIServer.

Αρχικά καλείται ο GenericRMIServer και δηλώνεται το SGEngineRMIIMpl το οποίο είναι αυτό που θέλουμε να εκτελεστεί με τον εξής τρόπο:

```
1  // Invoke GenericRMIServer
2  GenericRMIServer.getInstance();
3
4  //Initialize Engine RMI implementation
5  try {
6          final SGEngineRMIImpl engineImpl = new SGEngineRMIImpl();
7          engineImpl.setEngineApp(this);
8
9          //Register RMI Interface
10         GenericRMIServer.getInstance().registerInterface(engineImpl.RMI_NAME, engineImpl);
11 } catch (RemoteException e) {
12             Logger.getInstance().debug("Unable to register RMI interface", e);
13 }
```

Ακολουθεί ο κώδικας του SGEngineInterface ο οποίος όπως φένεται κάνει extend το BaseEngineInterface.

Listing 4.4: SGEngineInterface.java

```
1  package engine.rmi;
2
3
4  import db.model.event.Event;
5  import station.rmi.StationInterface;
6
7  import java.rmi.RemoteException;
8
9  /**
10  * RMI Interface of Second Generation Engine used by the (Second Generation) Stations.
11  */
12 public interface SGEngineInterface extends BaseEngineInterface {
13
14     /**
15      * The name of the RMI class.
16      */
17     String RMI_NAME = "SGEngine";
18
19     /**
20      * This is the remote method which is called by the Stations to
21      * retrieve their initial information.
22      *
23      * @param stationIP the IP address of the Station
24      * @param stationIF the remote object implementing the StationInterface
25      * @throws RemoteException if RMI was unable to invoke method
26      */
27     void registerStation(String stationIP, StationInterface stationIF)
28                                                         throws RemoteException;
29
30
31 }
```

Ακολουθεί ο κώδικας του SGEngineRMIImpl όπου και υλοποιούνται οι συναρτήσεις του SGEngineInterface.

Listing 4.5: SGEngineRMIImpl.java

```
1  package engine.rmi;
2
3  import db.model.event.Event;
4  import engine.AbstractEngineApp;
5  import station.rmi.StationInterface;
6  import util.Logger;
7  import util.eventconsumer.EventConsumer;
8
9  import java.rmi.RemoteException;
10 import java.rmi.server.UnicastRemoteObject;
11
12 /**
13  * Implementation of Second Generation Engine RMI Interface.
14  */
15 public class SGEngineRMIImpl extends UnicastRemoteObject
16                                         implements SGEngineInterface {
17
18     /**
19      * Version for Serialization.
20      */
21     static final long serialVersionUID = 42L;
22
23     /**
24      * Instance of the Engine application.
25      */
26     private AbstractEngineApp engineApp;
27
28     /**
29      * Default Constructor.
30      *
31      * @throws RemoteException if RMI unable to connect
32      */
33     public SGEngineRMIImpl() throws RemoteException {
34         super();
35     }
36
37     /**
38      * Set a reference to the engine application object.
39      *
40      * @param engineAp the engine application object
41      */
42     public void setEngineApp(final AbstractEngineApp engineAp) {
43         this.engineApp = engineAp;
44     }
45
46     /**
47      * Get a reference from the engine application object.
48      *
49      * @return Engine application object.
50      */
51     public final AbstractEngineApp getEngineApp() {
52         return engineApp;
53     }
54
55     /**
56      * This is the remote method which is called by the Stations to retrieve
```

```
57          * their initial information.
58          *
59          * @param stationIP the IP address of the Station
60          * @param stationIF the remote object implementing the StationInterface
61          * @throws RemoteException if RMI was unable to invoke method
62          */
63         public void registerStation(final String stationIP,
64                 final StationInterface stationIF) throws RemoteException {
65             stationIF.completeRegistration();
66             engineApp.registerStationInterface(stationIP, stationIF);
67         }
68
69         /**
70          * This is the remote method which is called by the Stations to send
71          * new events to the Engine.
72          *
73          * @param event The new Event
74          * @throws RemoteException if RMI was unable to invoke method
75          */
76         public void addEvent(final Event event) throws RemoteException {
77             Logger.getInstance().debug("New Event received from Station: "
78                                                     + event.getDescription());
79             EventConsumer.getInstance().addEvent(event);
80         }
81
82     }
```

4.3 Station

- **Delay Tolerant Service (DTS):** Αποτελείται απο τον Receiver, Disributer, DTSManager, Receiver και από τον Processor. Ο Receiver του DTS λαμβάνει τα Events από τους Guardians με την μορφή Datagram και τα γράφει σε μια ουρά (Buffer). Στην συνέχεια ο Disributer διαβάζει τα Datagrams από τον Buffer και τα στέλνει στον αντίστοιχο Processor. Αυτός με την σειρά του αρχικά τα μετατρέπει σε Events και στην συνέχεια αφού τα επεξεργαστεί ανάλογα με την υλοποίηση που έχει γίνει αποφασίζει πως θα τα διαχειριστεί. Π.χ. Αν υπάρχει επικοινωνία με το Engine, του τα στέλνει είτε για περαιτέρω επεξεργασία είτε για αποθήκευση στην βάση δεδομένων. Η αρχιτεκτονική του DTS παρουσιάζεται στο σχημα 4.4.

  Στην συνέχεια αναλύεται καθένα από τα components από τα οποία αποτελείται ο DTS ξεχωριστά .

  DTSManager: Ο DTSManager είναι μία Singleton κλάση, δηλαλη δημιουργείται μόνο ένα instance της κάθε φορά. Στον DTSManager ορίζεται το EventPort στο οποίο ο Receiver θα λαμβάνει τα Events από τους Guardians. Επίσης αρχικοποιέι δύο Threads, αυτά των Receiver και Distributer. Κατά την αρχικοποίηση των Threads τους περναέι και σαν όρισμα την ουρά στην οποία θα γράφει ο Receiver και θα διαβάζει ο Distributer. Η ουρά

είναι μια BlockingQueue<Datagram> και είναι μια διασυνδεδεμένη λίστα η οποία είναι FIFO (first-in-first-out). Η BlockingQueue ανήκει στο πακέτο java.util.concurrent.BlockingQueue της J2SE 1.6. Μετά την ολοκλήρωση της αρχικοποίησης των Threads τα εκτελεί. Ο DTSManager ακολουθεί το Java Observer pattern και είναι Observable για αλλαγές από τους registerd Processors.

Receiver: Ο σκοπός του Receiver είναι η λήψη των Datagrams από τους Guardians και το όσο το δυνατόν γρηγορότερο κλείσιμο της σύνδεσης με τον Guardian έτσι ώστε να επιτευχθεί μεγάλος ρυθμός λήψης Datagrams. Για αυτό το λόγο ο Receiver πρέπει να έχει μόνο της απολύτως απαραίτητες λειτουργίες. Επομένως ο Receiver ασχολείται αποκλειστικά με την λήψη των Datagrams και μόλις γράψει το Datagram στην ουρά κλείνει την σύνδεση κρατώντας την έτσι ανοιχτή για τον λιγότερο δυνατό χρόνο.

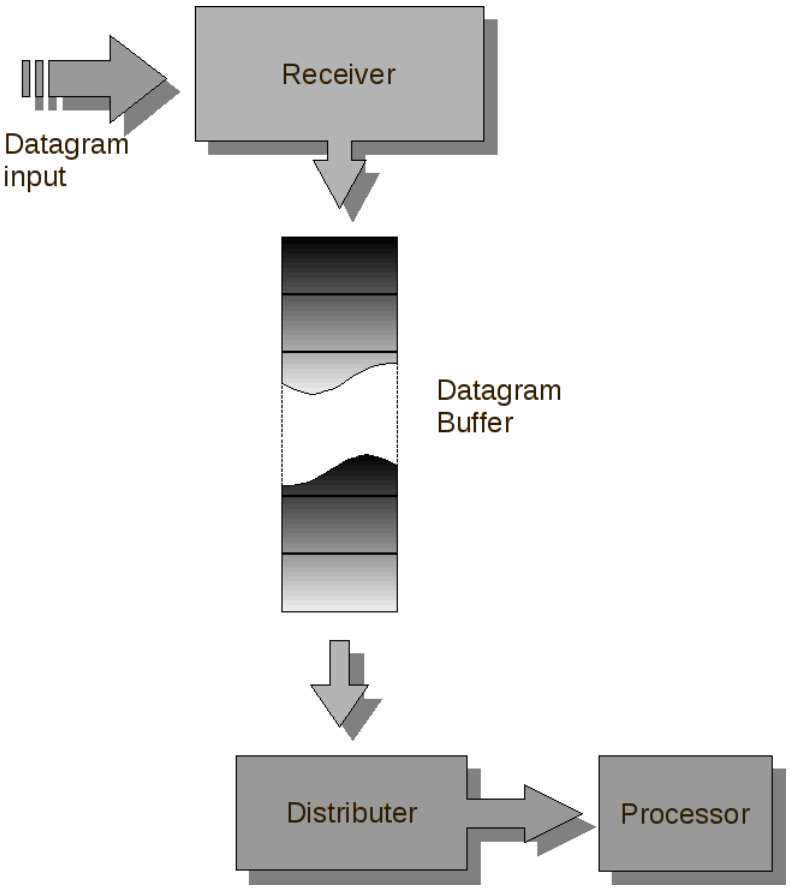
Distributer: Ο Distributer διαβάζει από την ουρά Datagram το οποίο εγγράφεται. Εάν ο Distributer επεξεργαζόταν τα Datagrams που διάβαζε απο τη ουρά, σε περίπτωση όπου έστελναν πολλοί Guardians ταυτόχρονα θα απαιτούσε μεγαλύτερο χρόνο από τον χρόνο εγγραφής των Events από τον Receiver στην ουρά με αποτέλεσμα κάποια στιγμή να γεμίσει η ουρά και έτσι θα υπήρχε απώλεια κάποιων Events καθώς δεν θα μπορούσαν να αποθηκευτούν στην ουρά. Για την αποφυγή του συγκεκριμένου φαινομένου κάθε φορά που φτάνει ενα Datagram στον Distributer καλείται η `DTSManager.getInstance().setChanged()` και στην συνέχεια η `DTSManager.getInstance().setChanged()` έτσι ώστε να φτάσει το νέο Datagram στους Observers δηλαδή στους registerd Processors. Επομένως η επεξεργασία των Datagrams αφήνεται στον Processor.

Processor: Ο Processor είναι υπεύθυνος για την μετατροπή των Datagrams σε Events. Ο Processor ακολουθεί και αυτός το Java Observer pattern και συγκεκριμένα είναι Observer του DTSManager. Λόγω του ότι το SunSPOT sdk δεν υποστηρίζει Serialization η διαδικασία μετατροπης του Datagram σε Event είναι η ακόλουθη:
Το Datagram μετατρέπεται σε Byte Array και στην συνέχεια σε αντικείμενο της κλάσης Event με την χρήση του Reflection της Java. Η συγκεκριμένη διαδικασία λόγο της χρήση του Reflection είναι αρκετά αργή γιαυτό θεωρείται απαραίτητη η χρήση του Processor και γιαυτό το λόγο δεν γίνεται μέσα στο Distributer. Ακολουθεί ο κώδικας που υλοποιεί την παραπάνω μετατροπή:

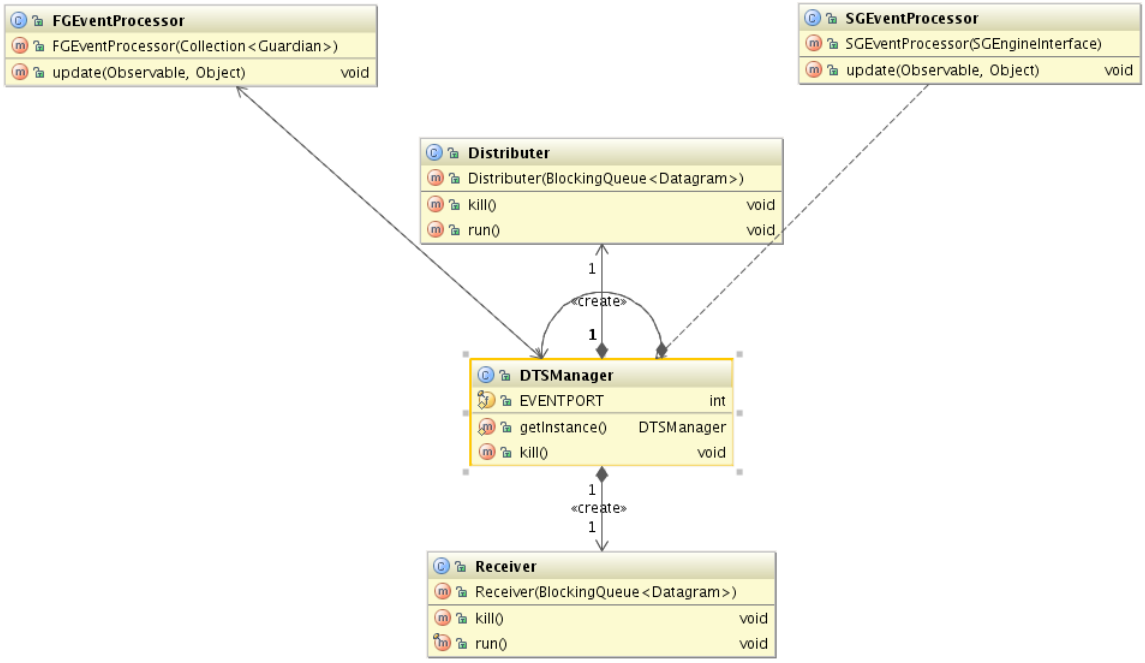
```
1 final Datagram datagram = (Datagram) arg;
2
3 //Event's classType
4 final String classType = datagram.readUTF();
5
6 //Locate Class based on class type name
7 final Class eventClass = Class.forName(classType);
```

```
 8
 9  //Get Default Constructor
10  @SuppressWarnings("unchecked")
11  final Constructor eventConstructor = eventClass.getConstructor(new Class[]{});
12
13  //Construct new Instance for this event type
14  final Event actEvent = (Event) eventConstructor.newInstance(new Object[]{});
15
16  //Extract Byte array length
17  final int length = datagram.readInt();
18
19  //Extract byte array
20  final byte[] eventArray = new byte[length];
21  datagram.readFully(eventArray, 0, length);
22
23  //Convert byte array to event object
24  actEvent.fromByteArray(eventArray);
```

Στο σχήμα 4.5 παρουσιάζεται το Class Diagram του DTS. Υπάρχει δυνατότητα να εκτελούνται περισσότεροι από ένας Distributer ταυτοχρόνα με σκοπό την αύξηση του ρυθμού διανομής των Datagrams στον Processor. Επίσης λόγο του Java Observer pattern σύμφωνα με το οποίο έχει υλοποιηθεί η επικοινωνία ανάμεσα στον DTSManager και στον Processor υπάρχει δυνατότητα να εκτελουνταί περισσότεροι από έναν Processor οι οποίοι θα δέχονται τα ίδια Datagrams αλλά θα τα επεξεργάζονται και θα τα χρησιμοποιούν με διαφορετικό τρόπο. Επίσης ο σχεδιασμός του DTS δίνει την δυνατοτήτα στον προγραμματιστή να υλοποιήσει εύκολα τον δικό του Processor που θα λειτουργεί όπως επιθυμεί χωρίς να χρειάζεται να γνωρίζει την συνολική λειτουργία του DTS.



Σχήμα 4.4: DTS Architecture.

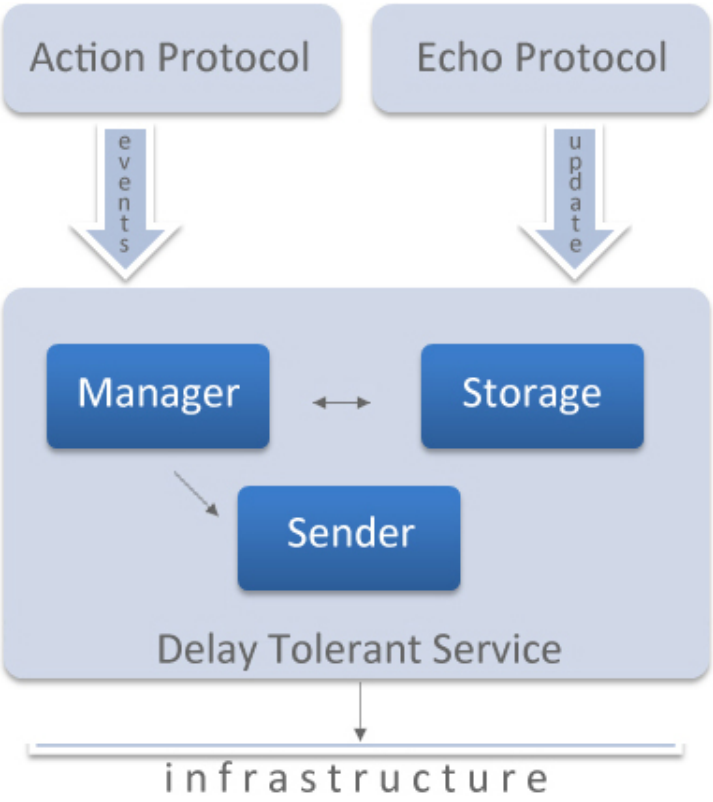


Σχήμα 4.5: DTS Class Diagram.

4.4 Guardian

- **Delay Tolerant Service (DTS):** Ο (DTS) στον Guardian αποτελείται από τρία διαφορετικά components. Τον Manager, τον Sender και το Storage. Μόλις δημιουργηθεί κάποιο καινούριο Event λόγο του Action Protocol ο Manager ελέγχει μέσο του EchoProtocol εάν ο Guardian είναι συνδεδεμένος σε κάποιο Station. Εάν είναι συνδεδεμένος ο Sender αναλαμβάνει να στείλει το Event στο Station. Στην περίπτωση που δεν είναι συνδεδεμένος το Storage Service αναλαμβάνει να αποθηκεύσει το Event στην Flash Memory του SunSPOT. Για την αποθήκευση των δεδομένων στην Flash Memory έχει υλοποιηθέι το Storage Service το οποίο με την χρήση της κλάσης javax.microedition.rms.RecordStore της J2ME αποθηκεύει το Event σε μορφή Byte Array.
  Μόλις έρθει ειδοποίηση απο το Echo Protocol ότι ο Guardian έχει συνδεθεί σε κάποιο Station ο Manager καλεί το Storage Service και ελέγχει εάν υπάρχουν αποθηκευμένα Event στην Flash Memory. Εφόσον υπαρχεί το Storage Service τα ανακτά και ο Sender αναλαμβάνει να τα στείλει στο Station.
  Η αρχιτεκτονική του DTS παρουσιάζεται στο σχήμα 4.6.

  Στην εικόνα 4.7 εμφανίζονται οι λειτουργίες που παρέχονται από το Storage Service. Κάθε Entity βάση του οποίου αποθηκεύεται στην Flash Memory. Ε-

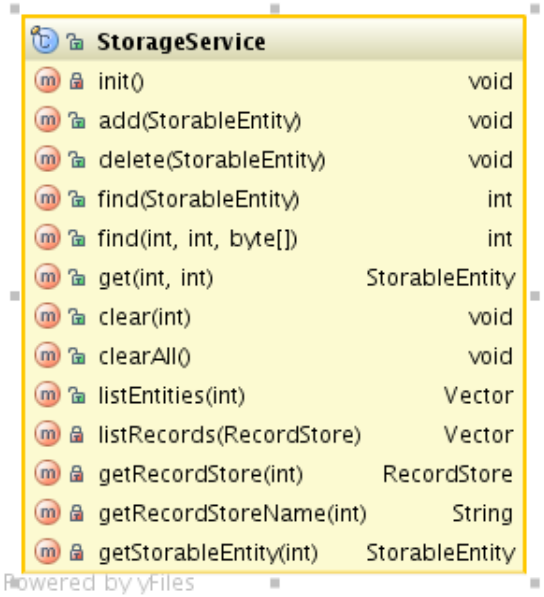


Σχήμα 4.6: DTS Architecture.

πίσης σε κάθε Entity Class έχουν υλοποιηθεί οι συναρτησείς `toByteArray()` που μετατρεπουν το αντικείμενο σε Byte Array καθώς επίσης και η `fromByteArray(final byte[] entityBuffer)` η οποία μετατρέπει το Byte Array που δέχεται ως όρισμα σε αντικείμενο. Στην συνέχεια ακολουθει ο κώδικας με τον οποίο υλοποίουνται οι δύο αυτές συναρτήσεις για την κλάση ActionEvent.

```
1     /**
2      * Converts the entity to byte array.
3      *
4      * @return the byte array of the entity
5      */
6     public byte[] toByteArray() {
7
8         byte[] data = null;
9
10        try {
11            // Create byte array
12            final ByteArrayOutputStream bout = new ByteArrayOutputStream();
13            final DataOutputStream dout = new DataOutputStream(bout);
14
15            dout.writeInt(getID());
16            dout.writeInt(getGuardianID());
```

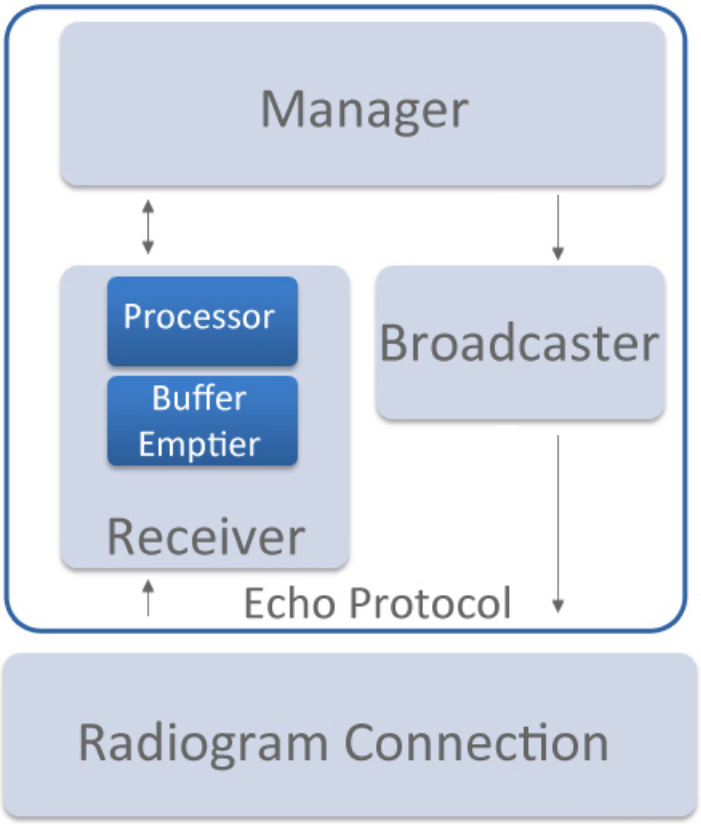
```
17              dout.writeInt(getGuardianCounter());
18              dout.writeUTF(getType());
19              dout.writeUTF(getDescription());
20              if (getBattleEngine() == null) {
21                  dout.writeInt(-1);
22              } else {
23                  dout.writeInt(getBattleEngine().getId());
24              }
25              dout.writeLong(getTimeStamp().getTime());
26
27              dout.writeBoolean(getIsSuccess());
28              if (getStation() == null) {
29                  dout.writeInt(-1);
30              } else {
31                  dout.writeInt(getStation().getStationId());
32              }
33
34              dout.flush();
35              dout.close();
36
37              data = bout.toByteArray();
38
39          } catch (Exception e) {
40              Logger.getInstance().debug(e);
41          }
42
43          // Return byte array
44          return data;
45      }
46
47      /**
48       * Initializes the entity by reading its byte array.
49       *
50       * @param entityBuffer - the entity's buffer
51       */
52      public void fromByteArray(final byte[] entityBuffer) {
53
54          try {
55              // Read entity's fields from byte array
56              final ByteArrayInputStream bin = new ByteArrayInputStream(entityBuffer);
57              final DataInputStream din = new DataInputStream(bin);
58
59              setID(din.readInt());
60              setGuardianID(din.readInt());
61              setGuardianCounter(din.readInt());
62              setType(din.readUTF());
63              setDescription(din.readUTF());
64              setBattleEngine(new BattleEngine());
65              getBattleEngine().setId(din.readInt());
66              setTimeStamp(new Date(din.readLong()));
67
68              setIsSuccess(din.readBoolean());
69              setStation(new Station());
70              getStation().setStationId(din.readInt());
71
72              din.close();
73              bin.close();
74
75          } catch (Exception e) {
76              Logger.getInstance().debug(e);
77          }
78      }
```



Σχήμα 4.7: Λειτουργίες που παρέχει το Storage Service .

- **EchoProtocol:**

4.5 World



Σχήμα 4.8: EchoProtocol Architecture.

Κεφάλαιο 5

Evaluation

5.1 Εισαγωγή

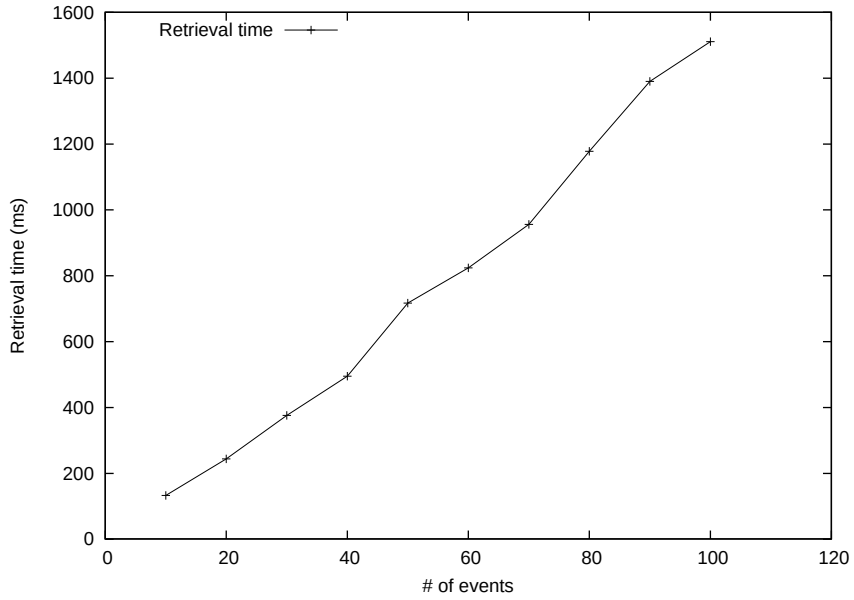
Στον παρών κεφάλαιο γίνεται μια προσπάθεια να προσδιοριστεί η απόδοση του συστήματος. Εστιάζει κυρίως στην απόδοση των πρωτοκόλλων που χρησιμοιούνται στους αισθητήρες καθώς επίσης και στην επικοινωνία ανάμεσα σε όλα τα επίπεδα του συστήματος. Όπως αναφέρθηκε και στο προηγούμενο κεφάλαιο οι αισθητήρες που χρησιμοποιήθηκαν στην πλατφόρμα είναι τα Sun's SPOTs. Τα Sun's SPOT βρίσκονται στο χαμηλοτέρο επίπεδο της πλατφόρμας και είναι οι συσκεύες που χρησιμοποιούν οι παίκτες για το παιχνίδι. Είναι μικρές συσκεύες που έχουν μπαταρία, προσφέρουν μια πολύ απλή διεπαφή στον χρήστη (2 κουμπιά και 8 LEDs) καθώς επίσης και κάποιους αισθητήρες (accelerometer, θερμοκρασία και φως). Τα Sun's SPOTs προγραμματίζονται σε Java Micro Edition (J2ME). Κατά την διάρκεια των μετρήσεων χρησιμοποιήθηκαν φορητοί και σταθεροί υπολογιστές μαζί με μερικά Alix Gateways έτσι ώστε να δημιουργηθεί η δομή της πλατφόρμας. Το μικρό μέγεθος των Alix επιτρέπει την χρήση τους ως Game Stations ή Engines. Το λειτουργικό σύστημα που χρησιμοποιήθηκε ήταν η ελαφριά Xubuntu Linux διανομή. Η επικοινωνία μεταξύ των Stations και των Sun's SPOTs επιτυγχάνεται μέσω του 802.15.4 ενώ μεταξύ των Stations και μεταξύ των Stations και του Engine μέσω του 802.11b. Η επικοινωνία του Engine και της βάσης δεδομένων MySQL v5.0.51a γίνεται πάνω από 100 Mbps LAN Ethernet.

5.2 Πρωτόκολλα στο επίπεδο των Guardian

5.2.1 Delay Tolerant Service

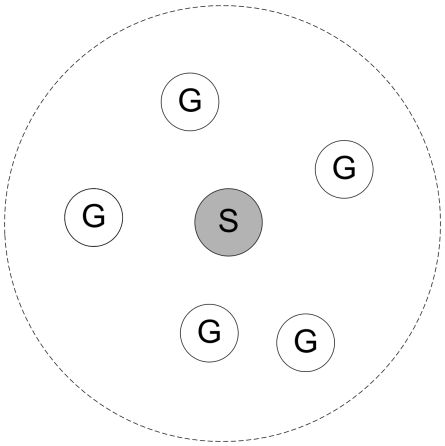
Ένα σημαντικό θέμα όσο αναφορά το Delay Tolerant Service Evaluation είναι ο χρόνος που απαιτείται από έναν Guardian όταν πλησιάζει ένα Station, για να ανα-

κτήσει τα Events που είναι αποθηκευμένα στη flash memory και να τα στείλει στο Station. Είναι γεγονός πως όση περισσότερη ώρα μένει ένας Guardian αποσυνδεδεμένος τόσα περισσότερα Events δημιουργούνται και αποθηκεύονται στην flash memory και επομένος απαιτείται περισσότερος χρόνος για την αποστολή τους στο Station. Στο σχήμα 5.1 παρουσιάζεται ο χρόνος που απαιτείται για να ανακτηθούν και να αποθηκευτούν προσωρινά σε κάποιον buffer. Στην συγκεκριμένη υλοποίηση όλα τα αποθηκευμένα Events ανακτώνται μονομίας έτσι ώστε να μειώθει το κόστος της συνεχόμενης ανάγνωσης απο την flash memory.

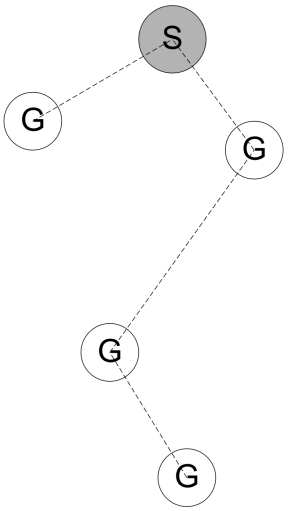


Σχήμα 5.1: Χρόνος ανάκτησης αποθηκευμένων Events απο την flash memory.

Για να μειωθεί ο χρόνος αποστολής των Events θα μπορούσε να αυξηθεί ο αριθμός των hops που θα φτάνουν τα beacons του Station. Με αυτό τον τρόπο το Station θα καλύπτει μεγαλύτερη περιοχή και οι Guardians θα στέλνουν τα Events απευθείας στο Station χωρίς να χρειάζεται να τα αποθηκεύσουν και έτσι δεν θα χρησιμοποιείται το Delay Tolerant Service. Όμως αυτό αυξάνει την πυκνότητα του δικτύου στην γειτονιά του Station. Αυτό οδηγεί σε αυξημένο υπολογιστικό φόρτο καθώς καθε κόμβος του δικτύου θα επεξεργάζεται και θα προωθεί περισσότερες απο μία φορές τα beacons του Station. Συγκεκριμένα σε ένα πλήρως συνδεδεμένο δίκτυο που αποτελείται από n κόμβους μέσα στην εμβέλεια του Station, κάθε κόμβος λαμβάνει $(n-1)^{max\ hop\ count-1}$ Station beacons. Επομένως η αποστολή των beacons σε παραπάνω από ένα hop θα πρέπει να γίνετε μόνο σε περιπτώσεις που υπάρχει πραγματικά ανάγκη. Στην συνέχεια παρουσιάζονται δύο διαφορετικά σεναρία χρήσης, το 5.2 και 5.3.



Σχήμα 5.2: Αυξημένη πυκνότητα του δικτύου. Περιορισμένο Station's beacons max hop count



Σχήμα 5.3: Μειωμένη πυκνότητα του δικτύου. Αυξημένο Station's beacons max hop count οδηγεί σε ενα πλήρως συνδεδεμένο δίκτυο

5.2.2 Echo Protocol

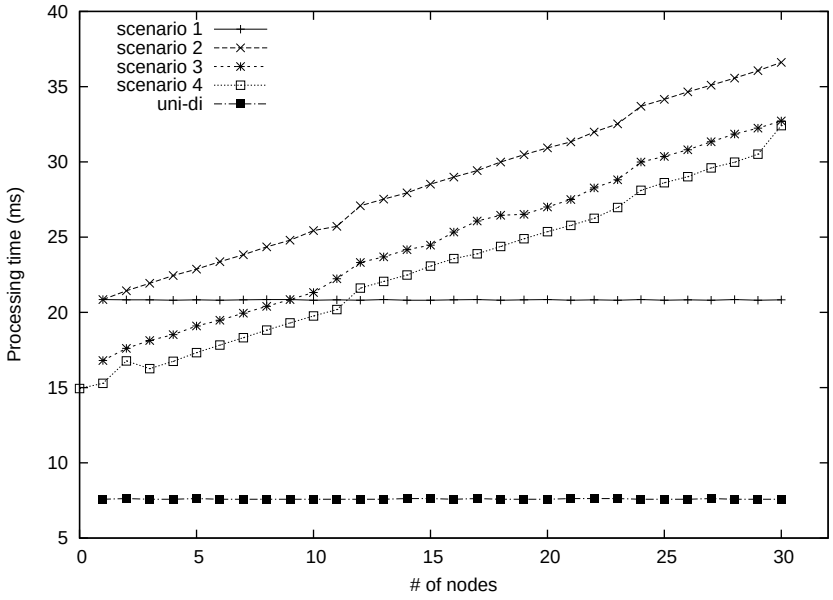
Αρχικά η αξιολόγηση του Echo Protocol αφορά τον χρόνο που απαιτείται από ένα κόμβο για να ανανεώσει την λίστα με τους γείτονες του χρησιμοποιώντας τον μηχανισμό του Echo Protocol. Συγκεκριμένα εξετάζεται η σχέση ανάμεσα στην πυκνότητα του δικτύου και στον χρόνο επεξεργασίας των beacons. Είναι σημαντικό για την απόδοση των παιχνιδιών να μπορεί η συσκευή ενός παίκτη να αντιληφθεί γρήγορα τις αλλαγές του δικτύου δίνοντας του την δυνατοτητα να αντιδράσει εγκαίρως.

Για την αξιολόγηση του Echo Protocol χρησιμοποιούνται δύο κόμβοι, ο Receiver και ο Broadcaster και οι μετρήσεις αφορούν τον χρόνο που απαιτείται από τον Receiver για να επεξεργαστεί τα beacons του Broadcaster. Ας υποθέσουμε οτι NR είναι ο αριθμός των γειτόνων του Receiver και NB ο αριθμός των γειτόνων του Broadcaster και t0 ο χρόνος που απαιτείται για να επεξεργαστεί ο Receiver ένα beacon και tnet ο χρόνος που απαιτείται για να επεξεργαστεί ο Receiver όλα τα beacons έλαβε. Στο σχήμα 5.4 απεικονίζονται οι χρόνοι που χρειάστηκε ο Receiver για την επεξεργασία ενός beacon σε διάφορα σενάρια.

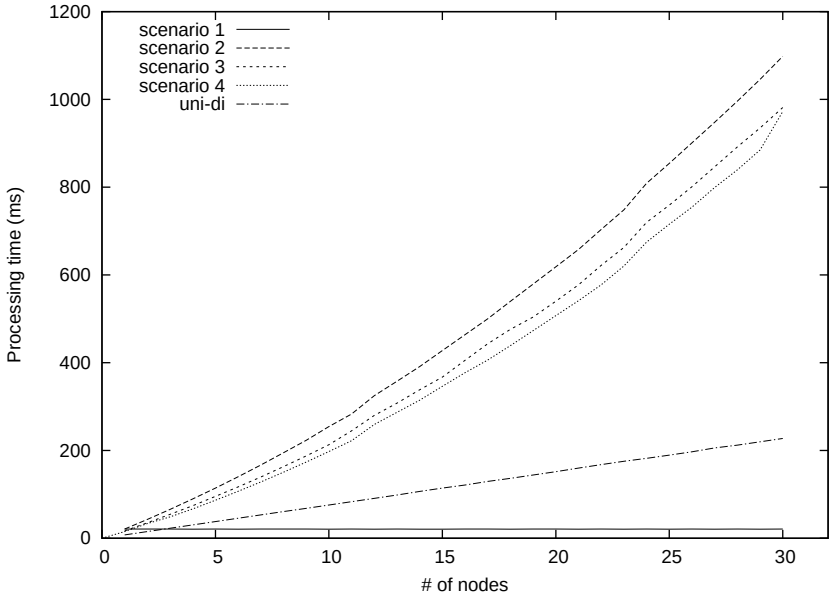
Στο σενάριο 1, η γειτονιά του Broadcaster παραμένει σταθερή (NB = 1) ενώ του Receiver αυξάνεται (NR =1;2;3; :::; 30). Αυτό έχει ως αποτέσμα ο χρόνος που απαιτείται για να επεργαστεί ο Receiver το beacon του Broadcaster να παραμένει σταθερός. Όμως όσο αυξάνονται οι γείτονες του Receiver ο χρόνος που απαιτείται για να επεξεργαστεί ολα τα beacons των γειτόνων αυξάνεται.

Στο σενάριο 2, ο Receiver και Broadcaster είναι κόμβοι ενός πλήρως συνδεδεμένου δικτύου με αυξανόμενο αριθμό γειτόνων (NB = NR = 1;2;3:::;30). Σε αυτήν την περίπτωση ο χρόνος t0 αυξάνεται καθώς τα δεδομένα που πρέπει να ανακτηθούν από το beacon είναι περισσότερα καθώς περιέχουν περισσότερες MAC addresses.

Στο σενάριο 3, υποθέτουμε ότι όλοι οι κόμβοι έχουν την ίδια εμβέλεια αποστολής μηνυμάτων επομένων όλες οι συσκευές μπορούν να επικοινωνήσουν μεταξύ τους (bi-directional links). Στα σενάρια 1 και 3 υποθέσαμε οτι δεν είχαν όλοι οι κόμβοι την ίδια εμβέλεια αποστολής μηνυμάτων επομένως δεν υπήρχαν bi-directional links. Η δυνατότητα αναγνώρισης των bi-directional links και οι μετρήσεις οδηγούν στο συμπέρασμα ότι η απόδοση του πρωτοκόλλου μειώνεται όσο αυξάνεται η πυκνότητα του δικτύου κυρίως όταν υπάρχουν bi-directional links. Συγκεκριμένα, σε όλες τις περιπτώσεις που υπάρχουν bi-directional links ο χρόνος tnet αυξάνεται εκθετικά σε σχέση με τον αριθμό των κόμβων του δικτύου. Στο σχήμα 5.5 παρουσιάζεται ο χρόνος tnet για κάθε ένα από τα σενάρια.



Σχήμα 5.4: Ο χρονος t_O σε κάθε σενάριο.



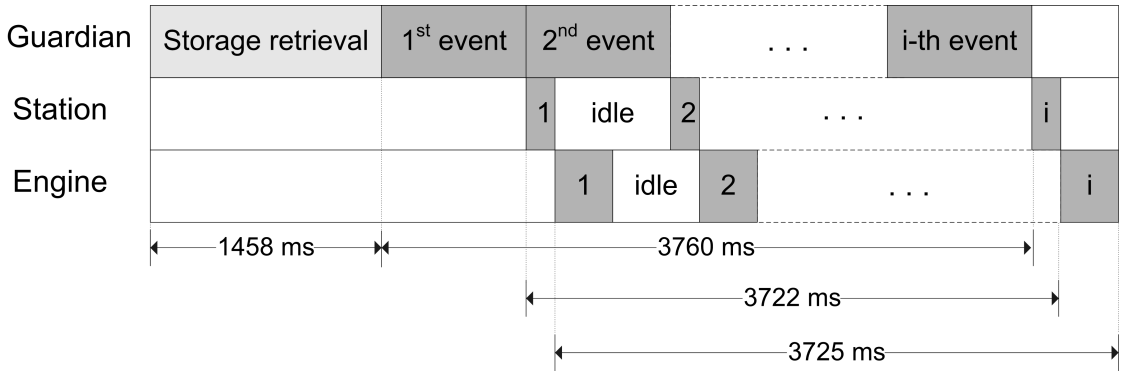
Σχήμα 5.5: Ο χρονος t_{net} σε κάθε σενάριο.

5.2.3 Cross-layer Evaluation

Στο παρών κεφάλαιο αναλυέται ο χρόνος που απαιτείται για να φτάσουν τα Events που δημιουργούνται στις συσκευές των παικτών για να περάσουν από όλα

τα επίπεδα της αρχιτεκτονίκης και να αποθηκευτούν τελικά στην κεντρική βάση δεδομένων. Συγκεκριμένα, αρχικά θεωρούμε πως ο Guardian είναι συνδεδεμένος σε κάποιο Station και ορίζουμε ως timeG τον χρόνο επεξεργασίας ενός Event στο επίπεδο του Guardian ενώ ορίζουμε ως timeS και timeE τον χρόνο που παραμένει το Event στο επίπεδο του Station και του Engine αντίστοιχα.

Στα πειράματα που έγιναν ο Guardian έστελνε 100 Events κάθε φορά. Ο συνολικός χρόνος που χρειάζεται 1 Event για φτάσει από τον Guardian στο Engine και να εγγραφεί στη βάση δεδομένων είναι περίπου 46 ms. Συγκεκριμένα η διάρκεια που παρέμεινε Event σε κάθε επίπεδο είναι: timeG = 38ms, timeS=2.5ms και timeE=5.5ms. Φένεται καθαρά ότι το Event το μεγαλύτερο χρονικό διάστημα παραμένει στο επίπεδο του Guardian και αυτός είναι της τάξης του 83%. Αλλά περαιτέτω μελέτη οδήγησε στο συμπεράσμα ότι ο μεγάλος αυτός χρόνος οφείλεται στον τρόπο λειτουργίας των συναρτήσεων του SunSPOT οι οποίες καταλώνουν μεγάλο χρονικό διάστημα στην προσπάθεια τους να δημιουργήσουν μια νέα σύνδεση με το Base Station με σκοπό να στείλουν δεδομένα. Δεν οφείλεται δηλαδή στην αρχιτεκτονική του συστήματος αλλά ούτε και στον τρόπο με τον οποίο υλοιήθηκε η πλατφόρμα. Οι συναρτήσεις αυτές δημιουργούν ιδιαίτερο πρόβλημα καθώς απαιτούν το 83% του timeG. Ο χρόνος που χρειάζεται ο Guardian για να διαβάσει από το Storage τα Events είναι σταθερός και μελετήθηκε στην ενότητα που αφορά την αξιολόγηση του Delay Tolerant Service, οπότε δεν συμπεριλαμβάνεται στο timeG.



Σχήμα 5.6: Το pipeline effect ανάμεσα στα τρία επίπεδα, Guardian, Station και Engine. Ένας Guardian στέλνει i Events. Ο χρόνος που απαιτείται μετρήθηκε με την αποστολή 100 Events (i = 100).

Αξιολογήθηκε επίσης η περίπτωση που πολλοι Guardians είναι γύρω από ένα Station και στέλνουν τα Events που έχουν δημιουργήσει. Σε αυτήν την περίπτωση μετρήθηκε ο χρόνος που απαιτείται έτσι ώστε όλα τα Events να φτάσουν στην βάση δεδομένων σε σχέση με τους Guardians που στέλνουν Events. Ο αριθμός των αποθηκευμένων Events στο Storage των Guardians παραμένει σταθερός και ειναι ίσος με 100 ενώ αυξάνεται ο αριθμός των Guardians που στέλνουν Events.

Συγκεκρίμενα αξιολογείται ο ρυθμός λήψης Events (ms/event) στο επίπεδο του Station (reception rate) και ο ρυθμός επεξεργασίας των Events (ms/event) στο επίπεδο του Engine (processing rate). Ο Station reception rate δείχνει πόσο χρόνο θέλει ένα Event για να φτάσει από ένα Guardian στο Station και να επεξεργαστεί πριν αποσταλεί στο Engine ενώ ο Engine processing rate το πόσο χρόνο χρειάζεται ένα Event για να επεξεργαστεί απο το Engine και να αποθηκευτεί στην βάση δεδομένων.

Αρχικά το πείραμα πραγματοποιήθηκε χρησιμοποιώντας έναν μόνο Guardian. Στην συγκεκριμένη περίπτωση ο συνολικός ρυθμός αποστολής από τον Guardian ήταν 37.6 ms/event ενώ ο Station reception rate ήταν 2.4 ms/event και ο Engine processing rate ήταν 5.3 ms/event. Είναι προφανές ότι ο χρόνος λήψης και επεξεργασίας των Events στα επίπεδα του Station και του Engine είναι πολύ μικρότερος σε σχέση με τον χρόνο που απαιτείται για να αποσταλούν τα Events από τον Guardian. Για αυτό το λόγο το Station και το Engine παραμένουν ανενεργά για 35.2 ms και 32.3 ms ανά Event αντίστοιχα. Επομένως το η συνολική καθυστέρηση του συστήματος οφείλεται στον Guardian όπου και εμφανίζετε bottleneck. Αυτό προκαλείται λόγω των περιορισμών και του προβλήματος που αναφέρθηκε προηγουμένος των SunSPOTs.

Επίσης πρέπει να αναφερθεί ότι η υλοποίηση του συστήματος επιτρέπει την ασύγχρονη επεξεργασία των Events. Τα αποτελέσματα των μετρήσεων παρουσιάζονται στο σχήμα 5.6. Παρατηρώντας το σχήμα μπορεί κανείς να παρομοιάσει τον τρόπο με τον οποίο λαμβάνονται και επεξεργάζονται τα Events με το φαινόμενο του pipeline. Λόγω του συγκεκριμένου φαινομένου ο συνολικός χρόνος που απαιτείται για να φτάσουν και τα 100 Events στην βάση δεδομένων είναι ίσος με totalG + timeS + timeE, όπου totalG ο συνολικός χρόνος για να αποσταλούν τα 100 Events από τον Guardian χωρίς να προμετράτε ο Storage retrieval χρόνος.

Στην συνέχεια αυξάνουμε τον αριθμό των Guardians επαλαμβάνοντας την ίδια πειραματική διαδικασία με πριν. Αυτό είναι ενα ρεαλιστικό σενάριο όπου περισσότεροι από έναν Guardian είναι συνδεδεμένοι σε ένα Station ταυτόχρονα και προσπαθούν να στείλουν τα Events που είναι αποθηκευμένα στο Storage.

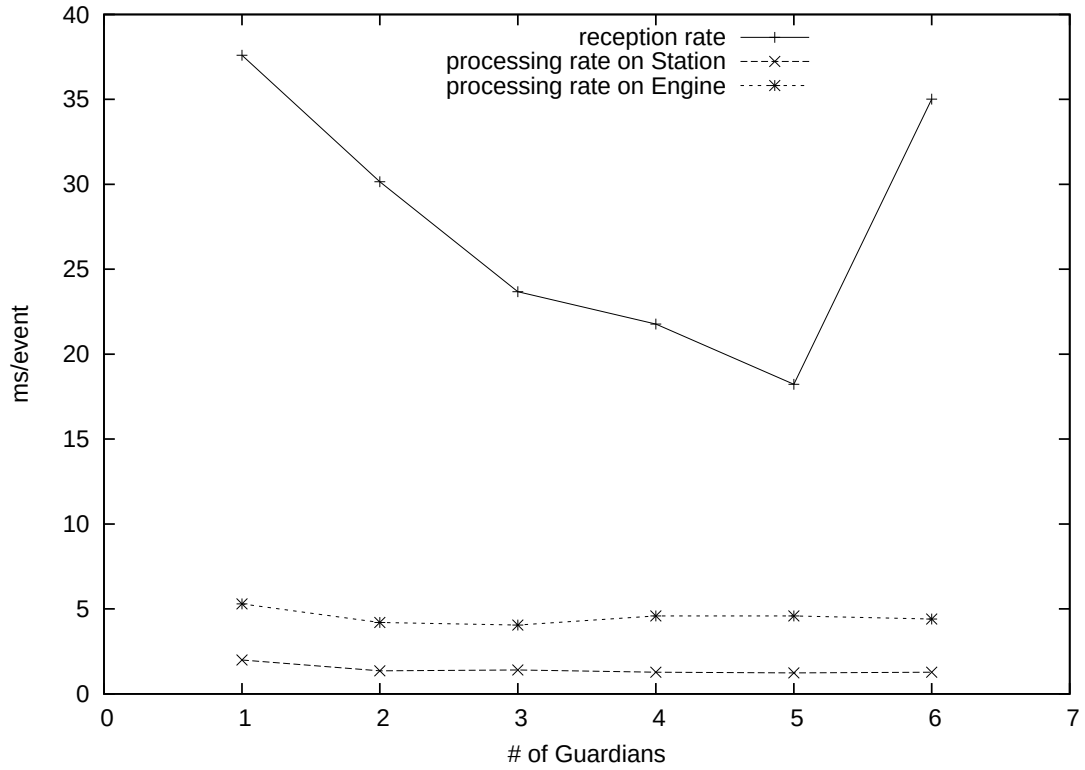
Προσπαθήσαμε να μειώσουμε τον event reception rate όσο περισσότερο γινόταν, πετυχαίνωντας ρυθμό μικρότερο απο τον Station Event processing rate. Αυτό θα είχε ως αποτέλεσμα τα Events να φτάνουν πιο γρήγορα από ότι μπορούν να επεξεργαστούν στο Station αναγκάζοντας τα έτσι να μπουν πρώτα στις ουρές που υπάρχουν στο Station πριν επεξεργαστούν. Τα αποτελέσματα συνοψίζονται στον πίνακα 5.1.

Παρατηρούμαι στο σχήμα 5.7 ότι όσο αυξάνεται ο αριθμός των Guardians, ο event reception rate μειώνεται. Παρόλα αυτά η συμπεριφορά του συστήματος αλλάζει όταν ενας 6ος Guardian εισέρχεται στο δίκτυο και ξεκιναέι να στέλνει Events. Ο event reception rate αυξάνεται καθώς η αύξηση της πυκνότητας του δικτύου (6

	Guardian	Station		Engine		
Nodes	Total	Total	Proc	Total	Proc	Overall
1	3760	3722	242	3725	531	3786
2	6031	5993	272	5996	840	6036
3	7106	7068	425	7070	1216	7111
4	8709	8669	511	8674	1839	8715
5	9115	9074	620	9079	2295	9121
6	21009	20969	768	20973	2648	21015

Πίνακας 5.1: Κατανομή του χρόνου σε milliseconds ανάμεσα στα διαφορετικά επίπεδα. Λόγω του bottleneck effect ο συνολικός χρόνος είναι σχεδόν ίσος με τον Total Guardian χρόνο. Ως Proc ορίζουμε τον χρόνο επεξεργασίας των Events σε κάθε επίπεδο.

Guardian + 1 Station) και η ποσότητα των Events που αποστέλονται (600) απαιτεί ένα μεγάλο ποσοστό των ήδη περιορισμένων πόρων του SunSPOT. Επίσης οι συγκρούσεις των πακέτων λόγων του 802.15.4 οδήγουν στην επαναποστολή τους οδηγώντας σε ένα υψηλότερο event reception rate. Τελικά οπώς φαίνεται εάν αυξηθεί ο αριθμός των Guardians σε περισσότερους απο 6 ο event reception rate αντί να μειώνεται αυξάνεται και το δίκτυο πλέον είναι ασταθές.



Σχήμα 5.7: Event reception και processing rate σε κάθε επίπεδο.

Κεφάλαιο 6

Tiny Web Services

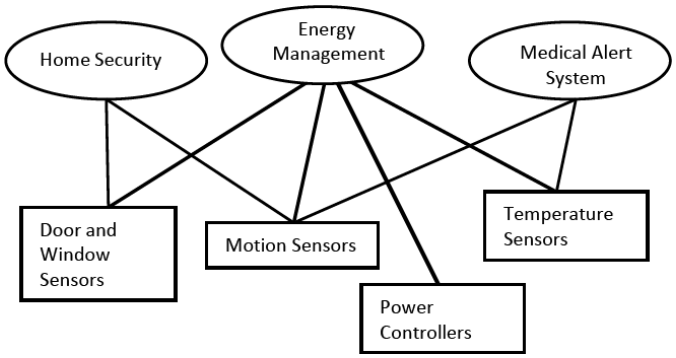
6.1 Εισαγωγή

Στο συνέδριο Sensys 2008 δημοσιεύτηκε το άρθρο με τίτλο Tiny Web Services: Design and Implementation of Interoperable and Evolvable Sensor Networks από τους Nissanka B. Priyantha, Aman Kansal, Michel Goraczko, and Feng Zhao της Microsoft Research, το οποίο είναι μια διαφορέτικη προσέγγιση του προβλήματος που προσπαθήσαμε να λύσουμε και ανάλυσαμε στα προηγούμενα κεφάλαια. Σύμφωνα με την συγκεκριμένη προσέγγιση σε κάθε επίπεδο της αρχιτεκτονικής εκτελούνται Web Services μέσω των οποίων παρέχεται η απαραίτητη λειτουργικότητα κάθε επιπέδου.

Ο σκοπός της δημιουργίας ενός συστήματος σύμφωνα με την προσέγγιση των Web Services είναι η δημιουργία ενός νέου επαναστατικού δικτύου αισθητήρων όπου επιπρόσθετοι κόμβοι θα μπόρουν να εισέρχονται στο δίκτυο μέτα από την αρχική τοποθέτηση - εγκατάσταση. Η λειτουργικότητα και τα δεδομένα κάθε κόμβου θα παρέχονται σε δομημένο τρόπο έτσι ώστε να μπορον να χρησιμοποιηθούν απο πολλαπλές εφαρμογές. Το αποτέλεσμα θα είναι η δημιουργία ενός συστήματος όπου θα εκτελούντε διάφορες εφαρμόγες χρησιμοποιώντας την ίδια υποδόμη και τους ίδιους αισθητήρες του δικτύου όπως φαίνεται στην εικόνα 6.1.

Η κύρια πρόκληση ενός συστήματος αισθητήρων με περιορισμένους υπολογιστικούς πόρους που χρησιμοποιεί web services είναι η επιπρόσθετη κατανάλωση ενέργειας καθώς επίσης και οι αυξημένες απαιτήσεις σε εύρος ζώνης λόγω του δομημένου τρόπου που παρέχονται τα δεδομένα λόγω της χρήσης των web services.

Οι αισθήρες που χρησιμοποιήθηκαν για την ανάπτυξη της πλατφόρμας όπου ελέγχθηκε και η σωστή λειτουργία του συστήματος, αποτελούνται από 48k ROM, 10k RAM και η επικοινωνία γίνετε μέσω του 802.15.4 πρωτοκόλλου. Οι μετρήσεις και τα αποτελέσματα από τις δοκιμές που έγιναν παρουσιάζονται αναλυτικά στη συνέχεια.

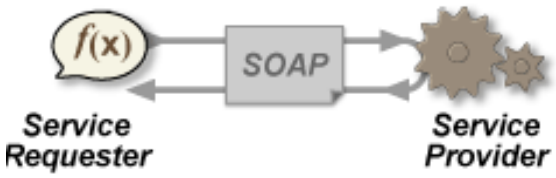


Σχήμα 6.1: Ένα παράδειγμα ενός συστημάτος οπου εκτελούνται πολλαπλές εφαρμογές πάνω στην ιδια υποδομή

6.2 Web Services σε δίκτυα αισθητήρων

Η ανάλυση των δυνατοτήτων των Web Services καθώς επίσης και τα πλεονεκτήματα που προσφέρουν αποδεικνύουν την σχέση τους με τα δίκτυα αισθητήρων και τους λόγους για τους οποίους μπορούν να χρησιμοποιηθούν στα συγκεκριμένα δίκτυα.

Τα Web Services προσφέρουν έναν μηχανισμό για κατανεμημένες εφαρμογες που επιτρέπει την κλήση μεθόδων σε απομακρυσμένες συσκευές. Ένα τυπικό σενάριο χρήσης ειναι το εξής : Κάποια λειτουργία ενός server παρέχεται ως μέθοδος η οποία μπορεί να κληθεί από μια απομακρυσμένη συσκευή. Για παράδειγμα, υπαρχεί ένας server ο οποίος ενημερώνει για τον καιρό μέσω της *Float temperature = GetTemperature(string Location)*. Οποιαδήποτε εφάρμογη έχει πρόσβαση στο δίκτυο που βρίσκεται ο συγκεκριμένος server μπορεί να καλέσει την συγκεκριμένη μέθοδο, δίνοντας σαν όρισμα την περιοχή για την οποία ενδιαφέρεται, και θα ενημερωθεί για την τιμή της συγκεκριμένης τοποθεσίας. Ο τρόπος με τον οποίο λειτουργούν τα Web Services παρουσιάζεται στο σχήμα 6.2.



Σχήμα 6.2: Λειτουργία των Web Services

Το πρωτοκολλο των Web Services δημιουργεί εσωτερικά και ανταλλάσει τα κατάλληλα μηνύματα που απαιτούνται για την κλήση κάποιας απομακρυσμένης μεθόδου και στην συνέχεια παρέχει το αντικείμενο με τα αποτελέσματα στην εφαρμογή

η οποία την κάλεσε. Ο προγραμματιστής δεν χρειάζεται να ανησυχεί για το τι πρωτόκολλα επικοινωνίας πρέπει να χρησιμοποιήσει, για το τύπο των πακέτων που πρέπει να στείλει ουτε για υπόλοιπες λεπτομέριες του δικτύου.

Η χρήση των Web Services απαιτεί τον σωστό ορισμό δυο αρχείων: ports και bindings. Τα ports ορίζουν τις λειτουργίες που προσφέρουν τα Web Services που τρέχουν σε μια συσκευή σε επίπεδο εφαρμογής και ορίζει επίσης και τις μεθόδους που μπορούν να κληθούν για να χρησιμοποιηθούν οι συγκεκριμένες λειτουργίες. Τα bindings καθορίζουν ποια πρωτόκολλα δικτύου μπορούν να χρησιμοποιηθούν. Για παράδειγμα αν κάποιος θέλει τα δεδομένα και οι παράμετροι να ανταλλάσσονται με SOAP μηνύματα θα πρέπει να ορίσει στα bindings οτι η επικοινωνία θα γίνει με χρήση SOAP.

Τα ports καθώς επίσης και τα bindings ορίζονται με την χρήση της web service description language (WSDL). Οι λειτουργίες που προσφέρει κάποιο συγκεκριμένο Web Service είναι γνωστό και ως WSDL περιγραφή. Εάν η λειτουργικότητα ενός αισθητήρα προσφέρεται μέσω μίας WSDL περιγραφής, τότε είναι εύκολο να προσπελαστεί χρησιμοποιώντας της μεθόδους που ειναι ορισμένες στην περιγραφή.

Το μεγαλύτερο πλεονέκτημα της χρήσης Web Services σε δίκτυα αισθητήρων είναι η δυνατότητα που προσφέρει σε πολλές εφαρμογές να τρέχουν ταυτόχρονα στο ίδιο δίκτυο χρησιμοποιώντας κοινούς αισθητήρες με απλό και εύκολο τρόπο. Βελτιώνει την λειτουργικότητα των εφάρμογων και επιτρέπει την υλοποίηση νέων χωρίς την ανάγκη εγκατάστασης νέων αισθητήρων. Ενα χαρακτηριστικό παράδειγμα είναι αυτό που δείχνει το σχήμα 6.1 όπου οι αισθητήρες που χρησιμοποιούνται για την εφαρμογή που είναι υπεύθυνοι για την ασφάλεια χρησιμοποιούνται και από την εφαρμογή που είναι υπεύθυνη για την εξοικονόμηση ενέργειας.

Ένα δεύτερο πλεονέκτημα είναι η βελτίωση της προγραμματιστικότητας του συστήματος. Η WSDL περιγραφή των δυνατοτήτων των αισθητήρων δίνει την δυνατότητα σε προγράμματα ανάπτυξης λογισμικού να δημιουργήσουν αυτόματα και γρήγορα κώδικα για την επικοινωνία και την χρήση των αισθητήρων. Ο προγραμματιστής βλέπει μόνο ένα αντικείμενο με τις κατάλληλες μεθόδους και μπορεί να το χρησιμοποιήσει. Το πρωτόκολλο των Web Services είναι υπεύθυνο για την δημιουργία των κατάλληλων μηνυμάτων που θα αποσταλούν, είναι υπεύθυνο επίσης για την αποστολή των αιτήσεων και την λήψη της απάντησης και παρέχει στην εφαρμογή μόνο το τελικό αποτέλεσμα. Έτσι ο προγραμματιστής δεν χρειάζεται να αναπτύξει δικά του πρωτόκολλα επικοινωνίας για τους αισθητήρες.

Επίσης τα Web Services δίνουν την δυνατότητα επικοινωνίας με τους αισθητήρες σε οποιαδήποτε εφαρμογή είναι συνδεδεμένη με το Internet δημιουργώντας έτσι νέα συστήματα με τεράστιες δυνατότητες. Η απευθείας εκτέλεση των Web Services στους αισθητήρες αποτρέπει την ανάγκη χρήσης πολλαπλών προκαθορισμένων πυλών και την δημιουργία ειδικών μηνυμάτων για την επικοινωνία.

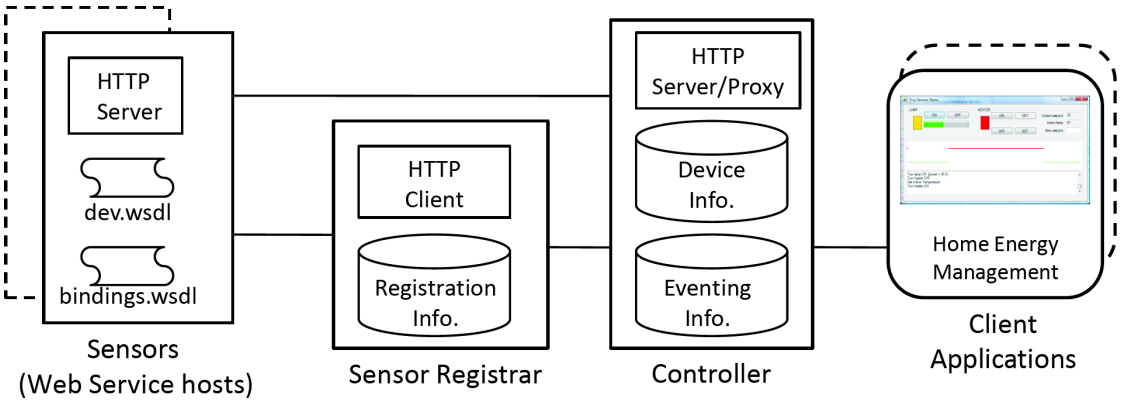
Επιπρόσθετα η χρήση του Internet Protocol σε επίπεδο δικτύου προσφέρει

πλεοεκτήματα όπως η χρήση DHCP για την δυναμική ανάθεση διευθύνσεων στους αισθητήρες. Το προτινόμενο πρωτόκολλο επικοινωνίας που μπορεί να χρησιμοποιηθεί σε δίκτυα αισθητήρων ειναι το 6lowpan που επιτρέπει την χρήση του Internet Protocol χωρίς να προσθέτει επιπρόσθετες επιβαρύνσεις στους αισθητήρες και προσφέρει και λύνει και το πρόβλημα της δρομολόγησης των πακέτων. Φυσικά εφόσον έχουν οριστεί οι WSDL περιγραφές μπορεί να χρησιμοποιηθεί οποιοδήποτε πρωτόκολλο επικοινωνίας καθώς αυτο δεν επηρεάζει την λειτουργία των Web Services.

Τα πλεονεκτήματα λοιπόν που προσφέρουν τα Web Services είναι αρκετά αλλά πρέπει πρώτα να μελετηθούν οι απαιτήσεις που έχουν για να εκτελεστούν σε αισθητήρες με περιορισμένους υπολογιστικούς πόρους, το πόσο τους επιβαρύνουν και αν είναι αποδεκτή η επιπρόσθετη κατανάλωση υπολογιστικών πόρων σε σχέση με τα πλεονεκτήματα που προσφέρουν.

6.3 Περιγραφή Συστήματος

Για την υλοποίηση των web services χρησιμοποιήθηκαν αισθητήρές που χρησιμοποιούν MSP430 επεξεργαστές και 802.15.4 radio. Ένας κεντρικός ελεγκτής controller αναπτύχθηκε σε έναν υπολογιστή με 802.15.4 radio. Στην 6.3 φαίνεται ενα διάγραμα με την υλοποίηση του συστήματος.



Σχήμα 6.3: Περιγραφή του Συστήματος

Το σύστημα αποτελείται απο τέσσερα διαφορετικά συστατικά μέρη (components). Αυτά είναι οι αισθητήρες (sensors) , ένα που είναι υπεύθυνο για την εγγραφή νέων αισθητήρων στο δίκτυο (sensor registar), ο ελεγκτής (controller) και η εφαρμογή (client application). Καθένα απο αυτά αναλύεται στην συνέχεια :

- **Sensors :** Κάθε αισθητήρας περιέχει δύο WSDL αρχεία που περιγράφουν τις θύρες (ports) και τη σύνδεση (binding). Το dev.wsdl ορίζει τις συναρτήσεις οι οποίες παρέχονται από κάθε συσκευή και το bindings.wsdl ορίζει τον

τρόπο με τον οποίο μπορούν να κληθούν και ενθυλακωθούν οι συναρτήσεις. Οι πληροφορίες που περιέχονται στο dev.wsdl περιγράφουν τα ονόματα των συναρτήσεων, τα ονόματα των παραμέτρων, τον τύπο δεδομένων της κάθε παραμέτρου καθώς επίσης και τον τύπο των επιστρεφόμενων τιμών. Το bindings.wsdl ενσωματώνει το dev.wsdl δημιουργώντας έτσι το bindings.wsdl το οποίο περιέχει την πλήρη WSDL περιγραφή του κάθε αισθητήρα. Οι αισθητήρες που δημιουργούν events περιέχουν στο bindings.wsdl τρεις ακόμη συναρτήσεις και ένα διαφορετικό τύπο Port. Οι δύο συναρτήσεις είναι η *void eventOn(string eventName)* και η *eventOff(string eventName)* που χρησιμοποιούνται για να ενεργοποιήσουν και να απενεργοποιήσουν αντίστοιχα κάποια συγκεκριμένα events. Η τρίτη συνάρτηση είναι η void setHandlerURL(string url) η οποία ορίζει σε ποιο URL θα στέλνονται τα events. Αυτές οι τρεις συναρτήσεις είναι προσβάσιμες μόνο από τον Controller ενώ οι συναρτήσεις που είναι ορισμένες στο dev.wsdl είναι προσβάσιμες από την εφαρμογή. Το bindings.wsdl χρησιμοποιει HTTP bindings με URL κωδικοποίηση για την κλήση των μεθόδων και XML μηνύματα για την επιστροφή των αποτελεσμάτων από την κλήση τους. Οι αισθητήρες τρέχουν έναν HTTP server για να μπορούν να χρησιμοποιήσουν τα WSDL αρχεία. Η απομακρυσμένη κλήση των μεθόδων που παρέχουν οι αισθητήρες επιτυγχάνεται μέσω του HTTP server.

Στον πίνακα 6.4 φένεται η κατανάλωση της μνήμης του αισθητήρα. Στο κέλι Const φένονται τα bytes που είναι μόνιμα αποθηκευμένα στην ROM. Από τον πίνακα 6.4 μπορεί να παρατηρήσει κανείς πως η συγκεκριμένη υλοποίηση μπορεί εύκολα να εγκατασταθεί και να λειτουργήσει σε αδύναμους επεξεργαστές όπως είναι ο MSP430. Τα αρχεία dev.wsdl και bindings.wsdl είναι τα μόνα που είναι σχετικά μεγάλα αλλά αυτό δεν δημιουργέι μεγάλο πρόβλημα καθώς προσπέλαση σε αυτά γίνεται σπάνια από τον Controller οπώς θα αναλυθεί στην συνέχεια.

Module	Code (bytes)	Data (bytes)	Const (bytes)
Libraries	804	2	76
Hardware control	408	78	12
Radio Driver	4282	404	14
TCP/IP	2964	332	4
Web server+ XML parser	2380	54	4864
Total	**10838**	**870**	**4970**

Σχήμα 6.4: Χρήση Μνήμης

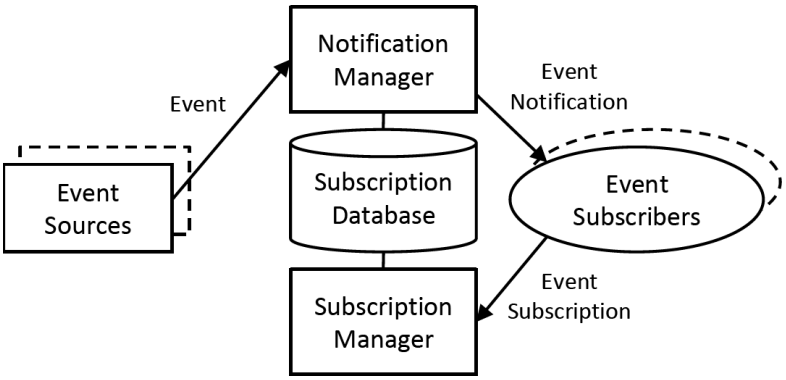
- **Sensor registar :** Είναι υπεύθυνος για την εγγραφή νέων συσκευών στο σύστημα. Στην συγκεκριμένη υλοποιήση, ο χρήστης δίνει στο συσκευή μια προκαθορισμένη IP διεύθυνση και ένα όνομα στην κάθε συσκευή μέσω ενός GUI. Όταν ένας νέος αισθητήτας συνδεθεί στο σύστημα το Sensor registar συνδέεται στην στον συγκεκριμένο αισθητήρα και κατεβάζει τα δύο WSDL αρχεία του. Στη συνέχεια συνδυάζει τα δύο WSDL αρχεία, την IP διεύθυνση και το όνομα που δόθηκε στον αισθητήρα και τα αποθηκεύει όλα σε ένα απλό XML αρχείο το οποίο το ανεβάζει στον Controller. Παρόλο που στη συγκεκριμένη υλοποίηση ο Sensor registar εκτελείται σε κάποιον ηλεκτρονικό υπολογιστή, υπάρχει υποστηρίξη απο το σύστημα ώστε ο Sensor registar και ο Controller να εκτελούνται σε κάποιο 802.15.4 access point.

- **Controller :** Ο Controller λειτουργεί σαν μια οντότητα η οποία ενώνει το δίκτυο των αισθητήρων με την Client εφαρμογή η οποία θέλει να χρησιμοποιήσει τους αισθητήρες. Ο Controller λειτουργεί επίσης και ως ένας ο HTTP server για την εγγράφη νέων αισθητήρων στο σύστημα και για την προσπέλαση των ήδη εγγεγραμμένων συσκευών. Όταν ο Client προσπελαύνει την διεύθυνση *http://Controller/index.htm*, ο Controller δυναμικά δημιουργεί μια HTML σελίδα όπου υπάρχει μια λίστα με όλους τους εγγεγραμένους αισθητήρες. Επίσης κάθε αισθητήρας περιέχει έναν σύνδεμο στην WSDL περιγράφη του, ο οποίος είναι της μορφής *http://Controller/wsdl/<sensorName>.wsdl*. Όταν ο WSDL σύνδεσμος προσπελαύνετε, ο Controller χρησιμοποιεί τα αποθηκευμένα WSDL αρχεία του αισθητήρα και δυναμικά δημιουργεί ένα νεο WSDL αρχείο με SOAP bindings στην θέση των HTTP bindings του αισθητήρα στο bindings.wsdl. Τα SOAP bindings προσφέρουν πρόσθετες λειτουργίες στη σύνδεση μεταξύ Controller και client applications. Για παράδειγμα, η client application μπορέι να εκτελείται σε μια συσκεύη που είναι συνδεδεμένη στο σπίτι μέσω Internet και οι λειτουργίες ασφαλείας που προσφέρει το SOAP θα μπορούσαν να χρησιμοποιηθούν για να ελέγξουν την πρόσβαση στη συγκεκριμένη συσκευή καθώς επίσης και για την πιστοποίηση των χρηστών που την χρησιμοποιούν.

  Όταν η client εφαρμόγη καλεί μια μέθοδο χρησιμοποιώντας SOAP, ο Controller δέχεται ένα SOAP μήνυμα, εξάγει το όνομα της μεθοδου που έχει κληθεί που και τις παραμετρους της και τα στέλνει στον αισθητήρα με τη χρήση HTTP bindings. Αν η κλήση της μεθόδου είναι επιτυχής, ο Controller ενθυλακώνει το XML αρχείο που επιστρέφει ο αισθητήρας μέσα σε ένα SOAP μήνυμα και το επιστρέφει στην client εφαρμόγη. Σε περίπτωση που η κλήση της μεθόδου δεν είναι επιτυχής, ο Controller επιστρέφει ένα SOAP μήνυμα λάθους.

  Στον Controller έχουν υλοποιηθεί επίσης όπως φενεται και στο σχήμα 6.5,

ο event subscription manager, το subscription database και ο notification manager που χρειάζεται για τα events και αποτελούν το WS-Eventing. Το WS-Eventing αποτελείται απο πέντε οντότητες : Τα Event sources που δημιουργούν events, τους Event subscribers που ενδιαφέρονται για συγκεκριμένα events, τον subscription manager που αποδέχεται και διαχειρίζεται της αιτήσεις των Event subscribers, την subscription database που αποθηκέυει όλους τους ενεργούς Event subscribers και τον notification manager που δέχεται τα events που έχουν δημιουργηθεί και τα στέλνει σε όλους τους ενεργούς Event subscribers που βρίσκονται στην βάση δεδομένων.



Σχήμα 6.5: Αρχιτεκτονική των Web Services στον Controller

Όταν μια εφάρμογη κάνει εγγραφή για παρακολούθηση ενός συγκεκριμένου event ενός αισθητήρα, ο Controller ανανεώνει την subscription database του, και ο ίδιος καλέι την μέθοδο εγγραφής του συγκεκριμένου event στον αισθητήρα στον οποίο υπάρχει. Όποτε ο αισθητήρας δημιουργήσει ένα event, ειδοποιείται ο Controller, ο οποίος είναι στην ουσία και ο notification manager, και αυτός με τη σειρά του διανέμει το event σε όλες τις client εφαρμογες που ειναι εγεγραμμενοι για την παρακολούθηση του συγκεκριμένου event.

Ο αισθητήρας έχει επίσης τη δυνατότητα να μπαίνει σε κατάσταση χαμηλής κατανάλωσης ενέργειας με το radio απενεργοποιημένο μετά από ένα συγκεκριμένο αριθμό αιτήσεων που δέχεται, καθώς κάποιες εφαρμογές το συγκεκριμένο χρονικό διάστημα θα πρέπει να επεξεργαστούν τα δεδομένα που δημιουργήθηκαν από τον αισθητήρα. Ο αισθητήρας ανοίγει το radio μόνο για να στείλει τα νέα events όταν αυτά δημιουργηθούν. Εάν περισσότερες από μία εφαρμογές θέλουν να καλέσουν μία μέθοδο ενός αισθητήρα, ο Controller θα πρέπει να περιμένει μέχρι ο αισθητήρας να ενεργοποιήσει το radio του για να του στείλει την αίτηση. Η κλήση των μεθόδων των αισθητήρων και οι απαντήσεις στις συγκεκριμένες αιτήσεις, οταν ο αισθητήρας βρίσκεται σε

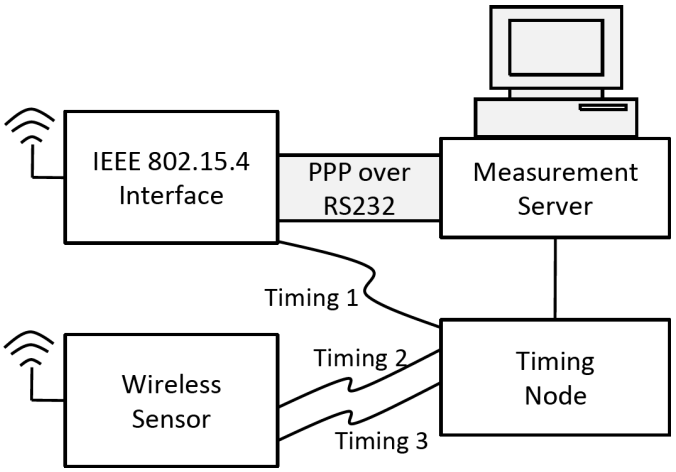
κατάσταση χαμηλής κατανάλωσης, γινεται ασύγχρονα.

- **Client application :** Η Client εφαρμογή είναι αυτή που χρησιμοποιεί ο τελικός χρήστης για να επικοινωνήσει με το δίκτυο των αισθητήρων μέσω της διεπαφης των web services. Η εφαρμογή καλεί την διεύθυνση του αισθητήρα στην οποία παρέχονται όλες οι μέθοδοι που υποστηρίζει, μέσω του WSDL αρχείου που δημιουργείται απο τον Controller, με χρήση SOAP μηνυμάτων. Υπάρχουν αρκέτα προγραμμάτα ανάπτυξης λογισμικού, όπως το Visual Studio και το NetBeans IDE ,που μπορούν να διαβάζουν τα WSDL αρχεία του Controller και να δημιουργήσουν αυτόματα αντικείμενα υψηλού επιπέδου γλωσσών προγραμματισμου, οπως για παράδειγμα αντικείμενα Java. Κάθε ένα από αυτά αναπαριστά τα web services που προσφέρει κάθε αισθητήρας. Ο προγραμματιστής το μόνο που έχει να κάνει είναι να καλέσει της μεθόδους του αντικειμένου για να χρησιμοποιήσει τον αισθητήρα. Το μόνο που χρειάζεται να ξέρει είναι σε ποιά λειτουργία αντιστοιχεί κάθε μεθόδος. Δεν χρειάζεται να ασχοληθεί καθόλου με τον τρόπο που πρέπει να αποσταλούν τα δεδομένα ή με το τι δεδομένα επιστρέφει κάθε συνάρτηση γιατί αυτό γίνεται αυτόματα κατά την δημιουργία του αντικειμένου από το πρόγραμμα ανάπτυξης λογισμικού όταν διαβάζει το WSDL αρχείο.

6.4 Απόδοση, Μετρήσεις

Η πειραματίκη διαδικασία για την σωστή μέτρηση του επιπλεόν κόστους απαιτεί την γνώση του αυξημένου μεγέθους των μηνυμάτων. το χρόνο επεξεργασίας του και τον αυξημένο αριθμό πακετων που ανταλλάσονται λόγω της αύξησης του μεγέθους όταν γίνεται χρήση των web services στους αισθήτηρες. Το σύστημα που εκτελέστηκε το πείραμα εμφανίζεται στο σχημα 6.6 και η επικοινωνία με τους αισθητήρες έγινε μεσω του 802.15.4 πρωτοκόλλου.

Έχει στηθεί ένας server που έχει και 802.15.4. Ο ασύρματος αισθητήρας είναι αποτελείται από έναν MSP430F1611 επεξεργαστή που τρέχει στα 6MHz και ένα CC2420 radio. Στον αισθητήρα εκτελείται το uIP TCP/IP μαζί με την υλοποιήση των web services που αναλύθηκε παραπάνω. Χρησιμοιήθηκε επίσης ένας κόμβος για να παίρνει μετρήσεις και αποτελείται απο ένα MSP430, είναι συνδεδεμένος μέσω USB στο server για να παίρνει διάφορα timing events από τον αισθητήρα και από τον server. Οι χρόνοι που αποθηκεύονται είναι οι εξής : Timing1 είναι ο χρόνος από την εκκίνηση αποστολής ενός πακέτου απο τον server μέχρι την ολοκλήρωση της αποστόλης, Timing2 είναι ο χρόνος από την εκκίνηση λήψης ενός πακέτου από τον αιθητήρα μέχρι και την επεξεργασία του, Timing3 είναι ο χρόνος από την εκκίνηση αποστολής ενός πακέτου απο τον αισθητήρα μέχρι την ολοκλήρωση της αποστόλης. Η διαφορά του Timing1 από το Timing2 δίνει τον



Σχήμα 6.6: Συστήμα πειραματικής διαδικασίας

χρόνο που χρειάστηκε ο αισθητήρας για να αποδικοποιήσει το πακέτο και να το επεξεργαστεί. Για να είναι ακριβείς οι μετρήσεις χρησιμοποιήθηκε ένα ρολοί που τρέχει στα 6MHz. Για τις μετρήσεις στον server χρησιμοιποιήθηκε το theWireshark packet sniffer. Μια απλή εκτέλεση του πειράματος δίνει τα αποτελέσματα που παρουσιάζονται στο σχήμα 6.7.

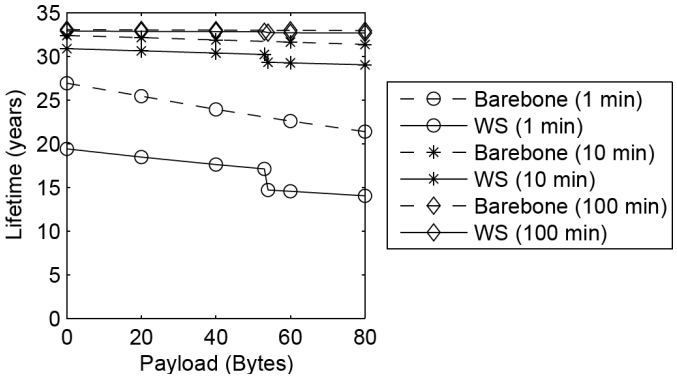
Time (ms)	Event	TCP Action
0.00	Server Tx start	TCP data
6.19	Server Tx done	(74 byte request)
9.68	Sensor Rx done	
10.67	Packet processed	
10.68	Sensor Tx start	TCP ACK
11.71	Sensor Tx done	
29.29	Server Tx start	TCP data
33.35	Server Tx done	(27 byte request)
35.53	Sensor Rx done	
36.35	Packet processed	
36.36	Sensor Tx start	TCP data
37.78	Sensor Tx done	(37 byte reply)

Σχήμα 6.7: Χρόνοι επικοινωνίας και επεξεργασίας των μηνυμάτων

Από τα αποτελέσματα του πειράματος καταβαίνει κανείς οτι ο επιπρόσθετος χρόνος για την επεξεργασία των πακέτων στον αισθητήτα λόγω των web services και λόγω του TCP/IP stack είναι πολύ μικρός. Συγκεκριμένα για το πρώτο πακέτο ο χρόνος επεξεργασίας είναι 10.67 - 9.68 = 0.99ms ενώ για το δεύτερο είναι 36.35 - 35.53 = 0.82ms

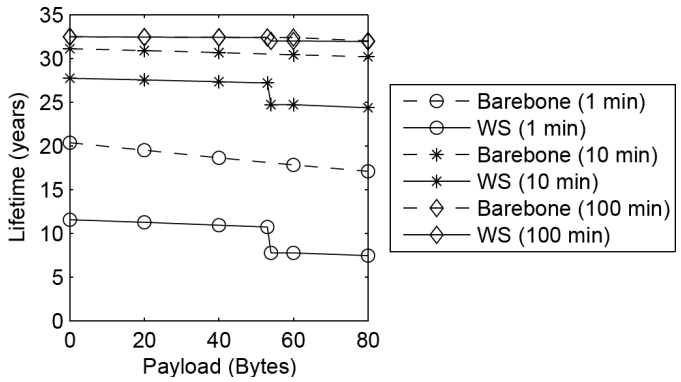
Η χρήση του TCP/IP και τα XML δεδομένα που απαιτούν τα web services αυξάνουν το μέγεθος των μηνυμάτων και αυτό έχει ως συνέπεια την αύξηση της κατανάλωσης ενέργειας στους αισθητήρες. Η αύξηση της κατανάλωσης ενέργειας μειώνει την διάρκεια ζωής της μπατάρίας. Η χρήση των web services μαζί με το TCP/IP προσθέτει σε κάθε μήνυμα κεφαλίδες ενθυλάκωσης οι οποίες για τα μηνύματα αιτήσεων είναι 21 bytes ενώ για τα μηνύματα απάντησης είναι 37 bytes όταν χρησιμοποιείται το HTTP 1.1. Όταν αποστέλετε ένα μήνυμα μεγάλυτερο των 57 bytes τότε απαιτούνται δύο πακέτα λόγω των TCP/IP και HTTP headers.

Ο επεργαστής των αισθητήρων που χρησιμοποιήθηκαν καταναλώνει 3 mA όταν είναι ενεργός και 2 μA όταν είναι σε κατάσταση sleep, ενώ το CC2420 radio καταναλώνει 18.8 mA όταν λαμβάνει δεδομένα, 17.4 mA όταν στέλνει και 0.02 μA οταν είναι σε κατάσταση sleep. Οί χρόνοι μετρήθηκαν στο σύστημα που ανάλυθηκε και χρησιμοποιήθηκε και παραπάνω και οι μπαταρίες που χρησιμοποιήθηκαν ήταν δύο AA των 2200 mAh.



Σχήμα 6.8: Κλειστό Radio κατά την μεταφορά

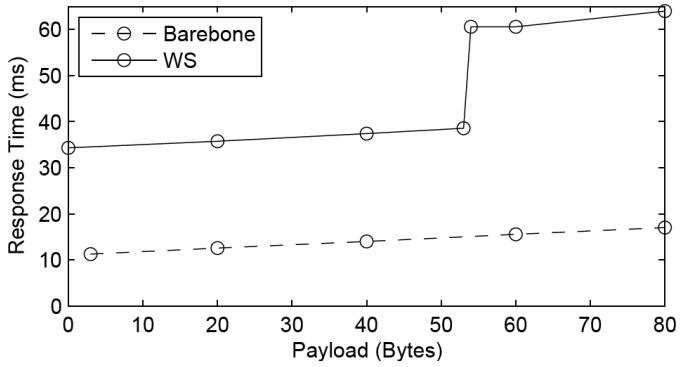
Η μπαταρία χάνει ετησίως το 3% από την χωρητικότητα της. Εάν ο χρόνος t εκφράζεται σε χρόνια, η χωρητικότητα της μπαταρίας δίνεται από τον τύπο $c(t) = c_0^{-t*ln(0.97)}$ όπου το c_0 είναι η αρχική χωρητικότητα της μπαταρίας. Η μπαταρία θεωρείται νεκρή όταν η τάση της πέσει κάτω απο 1.05 V που αντιστοιχεί στο $c(tdead) = 400mAH$. Το σχήμα 6.8 δείχνει την διάρκεια ζωής της μπαταρίας όταν το radio παραμένει κλειστό ανάμεσα στα κένα που δημιουργούνται αναμέσα στην αποστολή των TCP πακέτων και το σχήμα 6.9 δείχνει την διάρκεια ζωής της μπαταρίας όταν το radio παραμένει ανοιχτό καθόλη την διάρκεια της TCP μεταφοράς. Υπάρχουν τρία σενάρια μετρήσεων. Στο πρώτο έχουμε ανταλλαγή μηνυμάτων κάθε 1 λεπτό, στο δεύτερο κάθε 10 και στο τρίτο καθε 100. Στις γράφικες γίνεται σύγκριση της διάρκειας της μπαταρίας ανάμεσα σε αισθητήρες που τρέχουν τα συγκεκριμένα web services και σε αισθητήρες που στέλνουν τα μηνύματα απευθείας (barebone).



Σχήμα 6.9: Ανοιχτό Radio καθόλη την διάρκεια της μεταφοράς

Παρατήρειται ότι το κόστος λόγω του διαφορετικού μεγέθους των μηνημάτων είναι σχετικά μικρό όσο αναφορά την κατανάλωση της μπαταρίας όταν το μήνυμα χωράει σε ένα πακέτο. Όταν ένα μήνυμα στέλνεται κάθε 10 λέπτα και το μέγεθος του ειναι 40 bytes η διάρκεια ζωής της μπαταρίας μειώνεται κατα 4.74% και με το radio ανοιχτό καθόλη την διάρκεια της μεταφοράς μειώνεται κατά 10.85%. Η μείωση αυτή γίνεται ιδιαίτερα αισθητή όταν ο ρυθμός αποστολης μηνυμάτων είναι πιο συχνός, π.χ. ένα μήνυμα ανά λεπτό, αλλά τέτοιοι υψηλοί ρυθμοί μάλλον δεν χρειάζονται σε ένα τέτοιοι δίκτυο.

Συνοψίζοντας, παρατηρείται ότι η αύξηση του κόστους λόγω της χρήσης των web services δεν είναι σημαντικά μεγαλύτερη σε σχέση με ένα barebone πρωτόκολλο. Βέβαια στις μετρήσεις που αναπαριστώνται στα παραπάνω σχήματα φένεται καθαρά ότι ένα barebone πρωτόκολλο υπερτερεί.



Σχήμα 6.10: Χρόνοι απόκρισης συστήματος

Σε δίκτυα αισθητήρων όπου υπάρχει αλληλεπίδραση με ανθρώπους ο χρόνος απόκρισης του συστήματος είναι πολυ σημαντικός. Για παράδειγμα, ο χρόνος που θα χρειάζετε μια εφαρμόγη που τρέχει σε μία κινητή συσκεύη για να ανάψει μια

λάμπα στο σπίτι θα πρέπει να είναι πολύ μικρος.

Ενώ στα περισσότερα δίκτυα το επιπλέον κόστος λόγω των web services είναι αποδεκτό, στα δίκτυα αισθητήρων που χρησιμοποιούν 802.15.4 το μέγεθος των πακέτων αυξάνεται σημαντικά επηρεάζοντας το χρόνο απόκρισης του συστήματος. Ο χρόνος απόκρισης του συστήματος εξαρτάται από τον χρόνο που χρειάζεται για να ολοκληρωθεί η κλήση μιας μεθόδου ενός web service.

Μετρήθηκαν οι χρόνοι για διάφορα μεγέθη αιτήσεων, χρησιμοποιώντας το προηγούμενο σύστημα μετρήσεων και τα αποτελέσματα παρουσιάζονται στο σχήμα 6.10. Σημειώνεται οτι χρόνος απόκρισης δεν επηρεάζεται σημαντικά όταν η αίτηση κλήσης μίας μεθόδου έχει χωράει σε ενα απλό IP πακέτο. Για μηνύματα των 40 bytes με μέσο ρυθμό αποστολής ένα μήνυμα ανά 10 λεπτά η καθυστέρηση που προκύπτει είναι περίπου 23.09 ms. Η καθυστέρηση είναι και πάλι αρκέτα μικρή ακόμη και όταν το μήνυμα σπάει σε δύο πακέτα. Υπάρχει βέβαια τρόπος να μειωθεί και άλλο ακόμα και όταν το μέγεθος των μηνυμάτων απαιτεί πάνω από δύο πακέτα. Η καθυστέρηση οφείλεται κυρίως στο TCP/IP stack που παρέχεται από το uIP που χρησιμοποιήθηκε στους αισθήτηρες. Ο λόγος είναι ότι το uIP για κάθε πακέτο που στέλνεται επιστρέφει πίσω ένα acknowledgement. Απενεργοποιώντας τα acknowledgements μπορεί να μειωθεί ακόμη περισσότερο ο χρόνος απόκρισης.

Κεφάλαιο 7

Συμπεράσματα

Βιβλιογραφία