ΠΑΝΕΠΙΣΤΗΜΙΟ
ΔΥΤΙΚΗΣ ΜΑΚΕΔΟΝΙΑΣ
Τμήμα Μηχανικών
Πληροφορικής και τηλεπικοινωνιών

Διπλωματική Εργασία

ΛΕΙΤΟΥΡΓΙΚΟ ΣΥΣΤΗΜΑ ROS ΜΕ ΕΦΑΡΜΟΓΗ ΣΤΟ ΝΑΟ

Παναγιώτης Παναγιώτου

Επιβλέπων Καθηγητής: Δρ. Μηνάς Δασυγένης

Κοζάνη, Οκτώβριος 2015

Εξεταστική Επιτροπή:

Νικόλαος Φαχαντίδης

Μηνάς Δασυγένης

Θα ήθελα να ευχαριστήσω την οικογένειά μου για τη στήριξή της.

Επίσης, το Γιώργο Στεφανίδη, για τη βοήθειά του.

Περίληψη

Ο εξαιρετικά ενδιαφέρον και αναπτυσσόμενος τομέας της ρομποτικής έχει τις δικές του προκλήσεις και δυσκολίες. Το ROS είναι ένα εγχείρημα για τη διευκόλυνση της ανάπτυξης ρομποτικών εφαρμογών που έχει ιδιαίτερη επιτυχία. Στην παρούσα εργασία εξετάστηκε αυτό το σύστημα και αναπτύχθηκε μια εφαρμογή γραφικής διεπαφής χρήστη (GUI), για την επικοινωνία του χρήστη με το NAO, εξερευνώντας και χρησιμοποιώντας τις δυνατότητες του ROS.

Στο πρώτο κεφάλαιο πραγματοποιείται μια πρώτη γνωριμία με το NAO και εξηγείται η συμβολή του στον τομέα της ρομποτικής. Στη συνέχεια, στο δεύτερο κεφάλαιο αναλύεται το ROS και πιο συγκεκριμένα τα μοντέλα επικοινωνίας του και το σύστημα αρχείων του. Στο τρίτο κεφάλαιο παρουσιάζεται το ανθρωποειδές ρομπότ NAO και ο τρόπος με τον οποίο αυτό μπορεί να προγραμματιστεί. Στο τέταρτο κεφάλαιο παρουσιάζεται ο τρόπος ανάπτυξης της εφαρμογής, η οποία περιλαμβάνει την αποστολή εντολών στο ρομπότ από το χρήστη, αλλά και την οπτικοποίηση δεδομένων που προέρχονται από το NAO. Τέλος, στο πέμπτο κεφάλαιο παρουσιάζονται τα συμπεράσματα από την ενασχόληση με το ROS και οι μελλοντικές επεκτάσεις της εφαρμογής.

Abstract

Robotics is an extremely interesting and growing subject. Developers though, are facing many problems and challenges. ROS aims to ease the development of robotic applications and it is actually a very successful project, with a growing community using it.

In this thesis, on the one hand we investigate the way ROS works and the capabilities it offers to the user, while on the other hand we develop an application. It is a GUI for the facilitation of the communication between the user and the humanoid robot NAO.

In the first chapter, we introduce the idea of ROS and we answer the question “why should I use it”. In the second chapter, we get deeper into this platform, analyzing its communication models and its filesystem. In the third chapter, we present the basics of NAO and how it can be programmed. In the fourth chapter, we explain how the application was developed. Finally, we present our conclusions, after using ROS and propose future enhancements for the application.

Πίνακας περιεχομένων

Κεφάλαιο 1
Εισαγωγή

1.1 Προβλήματα και Προκλήσεις στο Χώρο της Ρομποτικής

Όσο ενδιαφέρον παρουσιάζει η ρομποτική, τόσες είναι και οι δυσκολίες, αλλά και οι προκλήσεις που παρουσιάζονται κατά την ανάπτυξη εφαρμογών στον τομέα της ρομποτικής. Αυτά προκύπτουν από τα εξής ιδιαίτερα χαρακτηριστικά:

- Από τη φύση των προβλημάτων, αφού στις περισσότερες περιπτώσεις, απαιτείται η εκτέλεση πολλών διαφορετικών εργασιών για μια και μόνο λειτουργία ενός ρομπότ. Για παράδειγμα, για το πιάσιμο ενός αντικειμένου, πρέπει να τρέχουν οδηγοί για τις κάμερες και τους χειριστές, αλλά ταυτόχρονα και ένας (δυνητικά μεγάλος) αριθμός ενδιάμεσων διεργασιών που εκτελούν λειτουργίες όπως η αναγνώριση αντικειμένων και η σχεδίαση των τροχιών της κίνησης. Ως αποτέλεσμα, συνήθως το απαιτούμενο εύρος γνώσεων και εξειδίκευσης απλώνεται πέρα από τις δυνατότητες οποιουδήποτε μεμονωμένου ερευνητή. Επίσης, οι εξαρτήσεις μεταξύ αυτών των συνιστωσών θα πρέπει να περιορίζονται στο λιγότερο δυνατό, ώστε

να επιτυγχάνεται καλύτερη συντήρηση, επεκτασιμότητα και επαναχρησιμοποίηση.

- Από το γεγονός ότι, μόνο και μόνο λόγω του μεγέθους των περισσότερων εφαρμογών, για την υλοποίησή τους πρέπει να συνεργαστούν πολλοί άνθρωποι.
- Από το μεγάλο αριθμό των διαφορετικών πανεπιστημιακών (και όχι μόνο) ιδρυμάτων που ασχολούνται με τη ρομποτική και το γεγονός ότι αυτά συνήθως εξειδικεύονται σε κάποιο συγκεκριμένο τομέα. Για παράδειγμα, κάποιο εργαστήριο μπορεί να έχει ειδικούς στην απεικόνιση εσωτερικών χώρων, κάποιο άλλο στην χρήση τέτοιων χαρτών για την πλοήγηση και ένα τρίτο να έχει ανακαλύψει μια προσέγγιση στον τομέα της μηχανικής όρασης (computer vision) που να κάνει καλή αναγνώριση μικρών αντικειμένων σε άτακτο περιβάλλον.
- Από το μεγάλο και συνεχώς αυξανόμενο αριθμό των ρομπότ, καθένα από τα οποία έχει διαφορετικό hardware, γεγονός που καθιστά την επαναχρησιμοποίηση κώδικα δύσκολη υπόθεση.
- Από την ανάγκη που υπάρχει πολλές φορές να είναι και η εκτέλεση των προγραμμάτων κατανεμημένη, .
- Από την ιδιαιτερότητα ότι οι δοκιμές που πρέπει να πραγματοποιηθούν για μια εφαρμογή είναι πολλές φορές δύσκολο να γίνουν, είτε λόγω επικινδυνότητας είτε γιατί τα ρομπότ δεν είναι πάντα διαθέσιμα.

Από τα παραπάνω, γίνεται εύκολα κατανοητή η ανάγκη για μια προτυποποίηση, προς την κατεύθυνση της διευκόλυνσης της εφαρμογής της μεθόδου του αρθρωτού προγραμματισμού (modular programming), που εκφράζει την οργάνωση του προγράμματος σε μικρά και ανεξάρτητα μέρη, τα οποία ονομάζονται ενότητες (modules). Αυτή η προσέγγιση είναι ιδιαίτερα επιθυμητή διότι κάνει πιο εύκολη την επαναχρησιμοποίηση κώδικα, αλλά και τη συνεργασία όσων συμμετέχουν στην ανάπτυξη μεγάλων ρομποτικών συστημάτων, λόγω της δυνατότητας αυτόνομης ανάπτυξης και ευκολότερης αποσφαλμάτωσης αυτών των ενοτήτων. Επίσης, με μια καλή τέτοιου είδους σχεδίαση διευκολύνεται ο διαχωρισμός του ελέγχου του υλικού του ρομπότ σε χαμηλό επίπεδο, από την επεξεργασία και τη λήψη αποφάσεων, που βρίσκονται σε ένα υψηλότερο επίπεδο, κάτι που επιτρέπει την προσωρινή αντικατάσταση των προγραμμάτων χαμηλού επιπέδου (και του υλικού που αντιστοιχεί σε αυτά) από έναν προσομοιωτή, προς όφελος των απαραίτητων δοκιμών.

Επίσης, θα ήταν πολλές φορές χρήσιμο, όταν εμφανίζεται μια καλή λύση να μπορεί να υιοθετηθεί και από άλλους, έτσι ώστε να μην χρειάζεται να «ξαναανακαλυφθεί ο τροχός», αλλά αντίθετα να δοθεί έμφαση στην έρευνα.

Κάπως έτσι σκέφτηκαν οι ομάδες στο Πανεπιστήμιο του Stanford στα μέσα της δεκαετίας του 2000, όταν ξεκίνησαν να φτιάχνουν κάποια πρότυπα για ευέλικτα, δυναμικά συστήματα λογισμικού, προσανατολισμένα για χρήση στη ρομποτική, τα οποία στη συνέχεια, το 2007, απέκτησαν τη μορφή του ROS (Robot Operating System), κάτω από την εποπτεία πλέον της Willow Garage, μέχρι το 2013, όταν η ευθύνη για την ανάπτυξη και τη συντήρηση του πυρήνα του ROS μεταφέρθηκε στην OSRF (Open Source Robotics Foundation). Περισσότερα ιστορικά στοιχεία, μπορείτε να βρείτε στο [1], ωστόσο αξίζει να σημειωθεί ότι από την αρχή συνέβαλαν στη ανάπτυξή του πολλοί ερευνητές από όλο τον κόσμο. Περισσότερα για αυτό θα αναφέρουμε στη συνέχεια.

1.2 Εισαγωγή στο ROS

Το ROS είναι μια πλατφόρμα λογισμικού ανοιχτού κώδικα (open source) που διευκολύνει την ανάπτυξη ρομποτικών εφαρμογών, απαντώντας στα προβλήματα και τις προκλήσεις που συναντούν οι ερευνητές που ασχολούνται με αυτό το αντικείμενο, τα οποία αναφέραμε πιο πάνω. Παρακάτω θα δούμε τους τρόπους με τους οποίους γίνεται αυτό.

Σε μια εφαρμογή σχεδιασμένη σε αυτήν την πλατφόρμα η λογική των εφαρμογών διαιρείται σε διεργασίες, οι οποίες ονομάζονται κόμβοι (nodes). Σε αυτούς πραγματοποιείται η επεξεργασία. Κατά το χρόνο εκτέλεσης, οι κόμβοι συνδυάζονται επικοινωνώντας μεταξύ τους. Ένα σημαντικό κομμάτι του ROS είναι η παροχή μιας υποδομής επικοινωνίας αυτών των αυτόνομων κόμβων, η οποία είναι «δικτυακά προσανατολισμένη» [2], που σημαίνει ότι δεν υπάρχει καμία διαφορά στο αν οι κόμβοι βρίσκονται στην ίδια ή είναι κατανεμημένοι σε διαφορετικές συσκευές.

Η βασική μέθοδος που χρησιμοποιείται είναι το μοντέλο εκδότη/συνδρομητή (publisher/subscriber), το οποίο προσφέρει χαμηλή σύζευξη μεταξύ των κόμβων, καθώς αυτοί στέλνουν μηνύματα, όχι απευθείας σε άλλους κόμβους, αλλά σε κοινούς χώρους ή «πίνακες» (boards), οι οποίοι ονομάζονται topics. Ωστόσο, στο ROS υποστηρίζονται και άλλοι τύποι επικοινωνίας μεταξύ των κόμβων, όπως οι υπηρεσίες (services) και τα actions. Όλοι οι διαφορετικοί τρόποι επικοινωνίας θα αναλυθούν στο επόμενο κεφάλαιο.

Στον πυρήνα της λειτουργικότητας του ROS, υπάρχει επίσης ένα μεγάλο εύρος εργαλείων σχετικά με την ανάπτυξη των εφαρμογών. Αυτά τα εργαλεία υποστηρίζουν την αυτοεξέταση, την αποσφαλμάτωση και την οπτικοποίηση της κατάστασης του συστήματος που αναπτύσσεται. Ο μηχανισμός εκδότη/συνδρομητή, με τη χαμηλή σύζευξη την οποία προσφέρει, επιτρέπει την επίβλεψη της πληροφορίας που μεταφέρεται μέσω του συστήματος. Τα εργαλεία

του ROS εκμεταλλεύονται αυτήν τη δυνατότητα αυτοεξέτασης μέσω μιας μεγάλης συλλογής από δυνατότητες, τόσο σε περιβάλλον κελύφους όσο και σε γραφικό περιβάλλον, που κάνουν εύκολη την κατανόηση και τη διόρθωση των σφαλμάτων, από τη μεριά του χρήστη. Τα πιο γνωστά εργαλεία γραφικού περιβάλλοντος είναι το rviz και το rqt. Για το δεύτερο, θα υπάρξει μεγαλύτερη ανάλυση στο Κεφάλαιο 4, καθώς είναι το εργαλείο στο οποίο βασίστηκε η ανάπτυξη μιας εφαρμογής που προσφέρει γραφική διεπαφή του χρήστη με το ρομπότ NAO.

Αυτή είναι η μια οπτική γωνία από την οποία μπορεί κανείς να δει το ROS. Η άλλη, είναι η ενθάρρυνση ενός συνεργατικού πλαισίου ανάπτυξης ρομποτικού λογισμικού. Το ROS χρησιμοποιείται, υποστηρίζεται και βελτιώνεται από μια μεγάλη κοινότητα. Η ιδέα είναι ότι κάθε οργανισμός (εταιρία ή Πανεπιστήμιο) που χρησιμοποιεί το ROS μπορεί να τοποθετήσει τον κώδικα στο δικό της αποθετήριο, να το κάνει δημόσια διαθέσιμο και να επωφεληθεί από οποιαδήποτε τεχνική ανάδραση και βελτιώσεις, όπως όλα τα προγράμματα ανοιχτού κώδικα. Ως αποτέλεσμα, προκύπτει ένα ομοσπονδοποιημένο σύστημα αποθετηρίων.

Για την υποστήριξη αυτής της ιδέας το ROS παρέχει ένα σύστημα πακέτων. Ένα πακέτο ROS είναι ένας κατάλογος ο οποίος περιέχει ένα αρχείο XML, στο οποίο περιγράφονται το πακέτο και οι εξαρτήσεις του. Οι μονάδες λογισμικού ομαδοποιούνται σε τέτοια πακέτα, με σκοπό τη διευκόλυνση της διανομής τους.

Μπορεί λοιπόν να βρει κανείς ένα μεγάλο αριθμό πακέτων λογισμικού σχετικού με τη ρομποτική, με οδηγούς (drivers) και αλγόριθμους που καλύπτουν τομείς όπως η κινηματική, η σχεδίαση κίνησης, η μηχανική εκμάθηση (machine learning), η επεξεργασία εικόνας και άλλους, υψηλότερου επιπέδου όπως η πλοήγηση (navigation), ο χειρισμός αντικειμένων (object manipulation) και η αναγνώριση αντικειμένων.

Ένα ακόμα σημαντικό στοιχείο είναι η έμφαση που δίνεται στην τεκμηρίωση. Στον ιστότοπο wiki.ros.org μπορεί κανείς να βρει και να πάρει πληροφορίες για τα χιλιάδες πακέτα που έχουν αναπτυχθεί από τους προγραμματιστές σε όλο τον κόσμο, αλλά και να γράψει εύκολα τη δική του σελίδα τεκμηρίωσης, χρησιμοποιώντας ένα εργαλείο αυτόματης δημιουργίας που παρέχεται. Επίσης, υπάρχει η πολύ χρήσιμη σελίδα answers.ros.org, την οποία διαχειρίζεται η ίδια η κοινότητα, με σκοπό την επίλυση ερωτήσεων των χρηστών, από την ίδια την κοινότητα.

Σύμφωνα με στοιχεία από το [3], υπάρχουν περισσότεροι από 3.300 χρήστες που συνέβαλαν με κάποια σελίδα στο wiki.ros.org, έτσι ώστε αυτό να έχει αυτήν την εποχή περισσότερες από 22.000 σελίδες, με 30 προσθήκες σελίδων την ημέρα. Στη σελίδα των ερωτήσεων-απαντήσεων έχουν ερωτηθεί 13000 ερωτήσεις, εκ των οποίων ένα 70% έχει βρει απάντηση. Το μέγεθος της κοινότητας του ROS, το

γεγονός ότι συνεχώς μεγαλώνει και το πόσο ενεργή είναι αυτή, «αποτελούν πιθανώς τον πιο σημαντικό παράγοντα στο ότι το ROS είναι μια από τις περισσότερο ολοκληρωμένες Πλατφόρμες Εφαρμογών Ρομποτικής (RSF)», σύμφωνα με το[4].

Αξίζει να σημειωθεί το καλό επίπεδο ολοκλήρωσης με άλλες υπάρχουσες τεχνολογίες. Σε αυτές συμπεριλαμβάνονται βιβλιοθήκες σχετικές με τη ρομποτική, όπως η βιβλιοθήκη μηχανικής όρασης OpenCV, που παρέχει χρήσιμους αλγόριθμους (σχετικούς για παράδειγμα με την παρακολούθηση αντικειμένων). Επίσης, παρέχεται ολοκλήρωση με τον προσομοιωτή Gazebo, με τη μέθοδο των plugin. Τέλος, υπάρχει και ένα καλό επίπεδο συμβατότητας με άλλα πλαίσια ανάπτυξης ρομποτικού λογισμικού, όπως το YARP, το OpenRTM, το OROCOS και το OpenRave.

Μια άλλη ιδιότητα του ROS, είναι ότι σχεδιάστηκε εξ αρχής για να υποστηρίζει τη χρήση διαφορετικών γλωσσών. Όπως αναφέρουν οι εμπνευστές του στο [15], «αντί να υπάρχει μια κεντρική υλοποίηση σε C, με την ύπαρξη διεπαφών για τη χρήση της μέσω άλλων γλωσσών, προτιμήθηκε το ROS να υλοποιηθεί ξεχωριστά σε αρκετές από τις γλώσσες που υποστηρίζονται, για να υπάρχει καλύτερη προσαρμογή στις ιδιαιτερότητες της κάθε γλώσσας». Οι βασικές γλώσσες είναι η C++ και η Python. Επίσης, μπορεί κανείς να προγραμματίσει χρησιμοποιώντας Java, LISP, Octave ή LabView. Σε ότι αφορά την υποστήριξη της δυνατότητας ανάπτυξης μιας εφαρμογής με συνδυασμό διαφορετικών γλωσσών (cross – language development), το ROS χρησιμοποιεί μια απλή γλώσσα προσδιορισμού διεπαφής (interface definition language - IDL) για την περιγραφή των μηνυμάτων που αποστέλλονται μεταξύ των modules. Αυτή η γλώσσα χρησιμοποιεί πολύ μικρά αρχεία κειμένου για την περιγραφή των πεδίων του κάθε μηνύματος και επιτρέπει τη σύνθεση μηνυμάτων. Οι γεννήτορες κώδικα για κάθε υποστηριζόμενη γλώσσα δημιουργούν τις αντίστοιχες υλοποιήσεις, που μοιάζουν με αντικείμενα και «σειριοποιούνται» (serialized) από το ROS, καθώς στέλνονται και λαμβάνονται τα μηνύματα. Αυτή η μέθοδος σώζει τον προγραμματιστή από πολύ χρόνο και λάθη.

Πριν ολοκληρώσουμε αυτήν την πρώτη εισαγωγή στο ROS, αξίζει να τονιστεί ότι, παρόλο που παρέχει υπηρεσίες που μοιάζουν με αυτές ενός λειτουργικού συστήματος, όπως η απόκρυψη υλικού και το πέρασμα μηνυμάτων μεταξύ των διεργασιών – κόμβων, δεν είναι ένα λειτουργικό σύστημα πραγματικού χρόνου.

1.3 Σκοπός της Εργασίας

Σκοπός της παρούσας διπλωματικής εργασίας είναι η μελέτη και η παρουσίαση των δυνατοτήτων, των εννοιών και των τρόπων χρήσης της πλατφόρμας ρομποτικού λογισμικού ROS. Παράλληλος στόχος είναι η κατασκευή

μιας εφαρμογής γραφικής διεπαφής χρήστη (GUI), στα πλαίσια αυτής της πλατφόρμας, μέσω της οποίας ο χρήστης θα μπορεί να στέλνει εντολές, αλλά και να λαμβάνει κάποιου είδους ανάδραση, από το ρομπότ NAO.

1.4 Παρουσίαση της Δομής της Εργασίας

Στο Κεφάλαιο 1 επιχειρήθηκε μια πρώτη γνωριμία με το ROS και δόθηκε μια απάντηση στην ερώτηση «γιατί αξίζει να χρησιμοποιηθεί». Στο Κεφάλαιο 2 γίνεται μια πιο λεπτομερής επεξήγηση των εννοιών του ROS και των τρόπων με τους οποίους αυτές υλοποιούνται και χρησιμοποιούνται. Στο Κεφάλαιο 3 γίνεται μια απαραίτητη παρουσίαση του NAO, από την οπτική γωνία του τρόπου με τον οποίο αυτό μπορεί να προγραμματιστεί. Αυτό χρειάζεται για να γίνει κατανοητός ο τρόπος με τον οποίο η εφαρμογή αποκτά πρόσβαση και επικοινωνεί με αυτό. Άλλωστε, το κομμάτι αυτό αποτελεί μέρος της διαδικασίας που ακολουθήθηκε για να αναπτυχθεί η εφαρμογή. Στο Κεφάλαιο 4, περιγράφονται οι στόχοι, η σχεδίαση της εφαρμογής και το πώς αυτή υλοποιήθηκε. Τέλος, στο Κεφάλαιο 5 διατυπώνονται τα τελικά συμπεράσματα σχετικά με τη χρήση του ROS και προτείνονται μελλοντικές επεκτάσεις και βελτιώσεις της εφαρμογής.

Κεφάλαιο 2
Παρουσίαση του ROS

2.1 Εγκατάσταση

Πριν από τα βήματα που πρέπει να ακολουθηθούν για την εγκατάσταση του ROS, ο χρήστης πρέπει να πάρει μια σημαντική απόφαση, καθώς τα πακέτα του πυρήνα του συστήματος που θα εγκατασταθούν διατίθενται σε εκδόσεις (versions), όπως συμβαίνει για παράδειγμα και με τη διάθεση των συστημάτων Linux (για παράδειγμα του Ubuntu). Αυτό γίνεται για να επιτυγχάνεται μια μεγαλύτερη σταθερότητα σε ότι αφορά τις εξαρτήσεις των πακέτων που αναπτύσσονται. Η πολιτική που ακολουθείται μέχρι στιγμής είναι να βγαίνει μια καινούρια έκδοση κάθε χρόνο. Αυτό συμβαίνει για να υπάρχει μια συσχέτιση με τη διάθεση των εκδόσεων των Ubuntu, από τη στιγμή που κάθε φορά η επίσημη πλατφόρμα για την οποία προορίζεται η εγκατάσταση του ROS είναι η πιο πρόσφατη έκδοση αυτού του λειτουργικού συστήματος.

Η πιο πρόσφατη μέχρι σήμερα έκδοση του ROS, της οποίας η διάθεση ξεκίνησε το Μάιο του 2015, έχει το όνομα Jade. Ωστόσο, για τις ανάγκες αυτής της διπλωματικής εργασίας επιλέχθηκε η έκδοση Indigo, ως η πιο πρόσφατη όταν ξεκινούσε η εκπόνησή της. Η διάθεσή του ROS Indigo ξεκίνησε τον Ιούλιο του 2014 και θα υποστηρίζεται μέχρι το 2019, σε αντίθεση με το Jade, για το οποίο η υποστήριξη θα είναι πιο βραχύβια, καθώς θα διαρκέσει μέχρι το 2017. Γενικώς,

σύμφωνα με την πολιτική του ROS για τις εκδόσεις, αυτές που βγαίνουν σε ζυγά έτη υποστηρίζονται για περισσότερο καιρό (για 5 χρόνια). Για το Indigo, η επίσημη πλατφόρμα στην οποία προορίζεται να λειτουργεί είναι η έκδοση Trusty των Ubuntu (14.04 LTS), ενώ η εγκατάσταση των πακέτων μπορεί να γίνει και στο Ubuntu Saucy (13.10).

Σε ότι αφορά την εγκατάσταση του Indigo, το πρώτο βήμα είναι να ορίσουμε ότι το σύστημά μας μπορεί να δέχεται λογισμικό από το αποθετήριο packages.ros.org. Αρχικά, πρέπει να βεβαιωθούμε ότι επιτρέπονται από τα αποθετήρια τα στοιχεία restricted, universe και multiverse. Αυτό μπορεί να γίνει σε γραφικό περιβάλλον από το «Software Sources», το οποίο μπορούμε να ανοίξουμε από το «Ubuntu Software Center» (και αφού εισάγουμε τον προσωπικό μας κωδικό). Οι επιλογές που μας ενδιαφέρουν βρίσκονται στην καρτέλα «Ubuntu Software». Ακολουθεί η δημιουργία (ως διαχειριστής) ενός αρχείου με όνομα /etc/apt/sources.list.d/ros-latest.list, το οποίο θα περιέχει μόνο μια γραμμή:

deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main

Η εντολή που πρέπει να δώσουμε στο τερματικό είναι η:

sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'

Σημειώνουμε ότι η εντολή sudo δηλώνει ότι εκτελούμε την επόμενη εντολή ως διαχειριστές και ότι η «$(lsb_release -sc)» δίνει την έκδοση των Ubuntu που χρησιμοποιείται.

Στη συνέχεια, πριν προχωρήσουμε στην εγκατάσταση των πακέτων του ROS, προσθέτουμε το απαιτούμενο κλειδί πιστοποίησης αυτών των πακέτων στο εργαλείο apt, το οποίο είναι το σύστημα διαχείρισης πακέτων. Στη σελίδα των οδηγιών για την εγκατάσταση του indigo (http://wiki.ros.org/indigo/Installation/Ubuntu), δίνεται απ’ ευθείας το απαιτούμενο κλειδί και έτσι δεν χρειάζεται να ακολουθήσουμε τη συνηθισμένη διαδικασία, κατά την οποία χρειάζεται να το κατεβάσουμε πρώτα, με το εργαλείο wget. Αντίθετα, μπορούμε απλά να γράψουμε την εντολή:

sudo apt-key adv –keyserver hkp://pool.sks-keyservers.net –recv-key 0xB01FA116

Το επόμενο βήμα είναι να βεβαιωθούμε ότι το ευρετήριο των Debian πακέτων είναι ενημερωμένο, με την εντολή

sudo apt-get update

Για την εγκατάσταση του indigo στα Ubuntu 14.04.2 προαπαιτείται η εγκατάσταση κάποιων άλλων πακέτων:

sudo apt-get install xserver-xorg-dev-lts-utopic mesa-common-dev-lts-utopic libxatracker-dev-lts-utopic libopenvg1-mesa-dev-lts-utopic libgles2-mesa-dev-lts-utopic libgles1-mesa-dev-lts-utopic libgl1-mesa-dev-lts-utopic libgbm-dev-lts-utopic libegl1-mesa-dev-lts-utopic

Αν όμως χρησιμοποιείται η έκδοση 14.04, η παραπάνω εντολή πρέπει οπωσδήποτε να αντικατασταθεί από την:

sudo apt-get install libgl1-mesa-dev-lts-utopic

Ακολουθεί η εντολή με την οποία θα εγκαταστήσουμε τα πακέτα του ROS. Υπάρχουν τρεις πιθανές εντολές, ανάλογα με το είδος της εγκατάστασης που θέλουμε να κάνουμε. Οι διαφορετικές επιλογές προκύπτουν από το γεγονός ότι ο πυρήνας του ROS αποτελείται από ένα μεγάλο αριθμό βιβλιοθηκών και εργαλείων και μπορεί να χρειαστεί να μην τα εγκαταστήσουμε όλα, αν και αυτό δεν προτείνεται και θα πρέπει να γίνεται μόνο σε περιπτώσεις που δεν υπάρχει αρκετός διαθέσιμος χώρος (ολόκληρος ο πυρήνας του ROS απαιτεί μερικά GB χώρου). Για μια πλήρη εγκατάσταση, που περιλαμβάνει το ROS, τα εργαλεία γραφικού περιβάλλοντος rqt και rviz, βιβλιοθήκες γενικού περιεχομένου ρομποτικής (που δεν αφορούν δηλαδή κάποιο συγκεκριμένο ρομπότ), προσομοιωτές και πακέτα που έχουν σχέση με την πλοήγηση και την αντίληψη, δίνουμε την εντολή

sudo apt-get install ros-indigo-desktop-full

Για μια εγκατάσταση του ROS χωρίς τους προσομοιωτές και τα πακέτα για την πλοήγηση και την αντίληψη η εντολή είναι η:

sudo apt-get install ros-indigo-desktop

Για μια αρκετά «φτωχή» εγκατάσταση, που δεν θα περιλαμβάνει ούτε τα εργαλεία γραφικού περιβάλλοντος:

sudo apt-get install ros-indigo-ros-base

Σημειώνουμε ότι, στην (πιθανή) περίπτωση να θελήσουμε αργότερα να εγκαταστήσουμε ένα συγκεκριμένο πακέτο, υπάρχει η εντολή

sudo apt-get install ros-indigo-[όνομα πακέτου]

Η βασική διαδικασία έχει τελειώσει, όμως πριν χρησιμοποιήσουμε το ROS απαιτούνται κάποια επιπλέον βήματα. Πιο συγκεκριμένα:

Αρχικοποιούμε το εργαλείο rosdep, το οποίο εγκαθιστά τις εξαρτήσεις (system dependencies) που απαιτούνται κατά τη μεταγλώττιση πηγαίου κώδικα και είναι απαραίτητος και για το τρέξιμο κάποιων στοιχείων του πυρήνα του ROS:

```
sudo rosdep init

rosdep update
```

Το δεύτερο βήμα, έχει να κάνει με το ότι στην πορεία θα γίνει πολλές φορές χρήση κάποιων μεταβλητών περιβάλλοντος του ROS, άρα θα ήταν πολύ χρήσιμο να προστίθενται αυτόματα στο bash session, κάθε φορά που εκκινείται κάποιο κέλυφος (shell). Αυτό γίνεται με:

```
echo "source /opt/ros/indigo/setup.bash" >> ~/.bashrc

source ~/.bashrc
```

Ο ρόλος της ενσωματωμένης εντολής κελύφους «source» είναι να εκτελεί το περιεχόμενο του αρχείου που παίρνει ως όρισμα στο τρέχον κέλυφος. Η μεμονωμένη χρήση της εντολής «source /opt/ros/indigo/setup.bash» θα άλλαζε το περιβάλλον του τρέχοντος κελύφους, προσθέτοντας σε αυτό τις μεταβλητές περιβάλλοντος που έχουν οριστεί για το ROS, αλλά αυτό θα κρατούσε μόνο μέχρι το κλείσιμο αυτού του κελύφους.

Τέλος, είναι χρήσιμη η εγκατάσταση του εργαλείου γραμμής εντολών rosinstall, το οποίο επιτρέπει την ευκολότερη εγκατάσταση πολλών πακέτων με μόλις μια εντολή και διατίθεται ξεχωριστά από το ROS:

```
sudo apt-get install python-rosinstall
```

2.2 Ανάλυση του Επιπέδου Επικοινωνίας

Οι τρόποι με τους οποίους πραγματοποιείται η επικοινωνία μεταξύ των διεργασιών στο περιβάλλον του ROS είναι το σημαντικότερο θέμα το οποίο πρέπει κανείς να γνωρίζει, για να μπορέσει να προγραμματίσει σε αυτό το περιβάλλον. Σε αυτήν την ενότητα βλέπουμε ότι υπάρχουν τρεις βασικοί τρόποι επικοινωνίας, τα topics, οι υπηρεσίες και ο εξυπηρετητής παραμέτρων (parameter server), καθώς και ένας πιο περίπλοκος και εξειδικευμένος, τα actions. Πριν όμως από την ανάλυση αυτών και τα παραδείγματα της υλοποίησής τους με τη χρήση της βιβλιοθήκης rospy, με την οποία μπορούν να κληθούν οι λειτουργίες του ROS μέσα σε κώδικα γραμμένο σε Python, παρουσιάζονται τα βασικά στοιχεία από τα οποία αποτελείται ο «Υπολογιστικός Γράφος» (Computation Graph), που είναι η ονομασία που δίνεται στο [5] για να περιγραφεί η έννοια του δικτύου των διεργασιών που συνεργάζονται

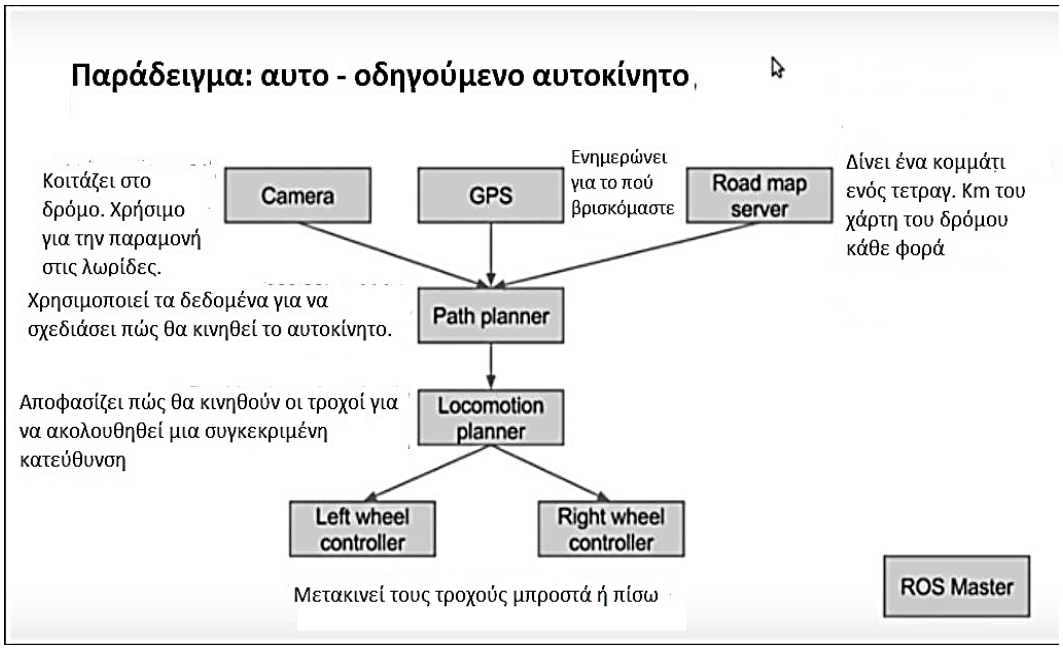
για την επεξεργασία των δεδομένων. Επίσης, παρουσιάζεται η χρήση της βιβλιοθήκης actionlib, για την επικοινωνία με actions.

2.2.1 Βασικές Έννοιες του Υπολογιστικού Γράφου

Οι βασικές έννοιες σε ένα δίκτυο επικοινωνίας στο ROS είναι οι εξής:

- Κόμβοι: Οι κόμβοι (nodes) είναι οι διεργασίες, στις οποίες πραγματοποιείται η επεξεργασία. Συνδυάζονται και επικοινωνούν μεταξύ τους, με τρόπο που θα μπορούσαμε να απεικονίσουμε με ένα γράφο (γι' αυτό και ονομάζονται έτσι) χρησιμοποιώντας τους μηχανισμούς επικοινωνίας που αναφέραμε παραπάνω και θα αναλύσουμε στη συνέχεια.

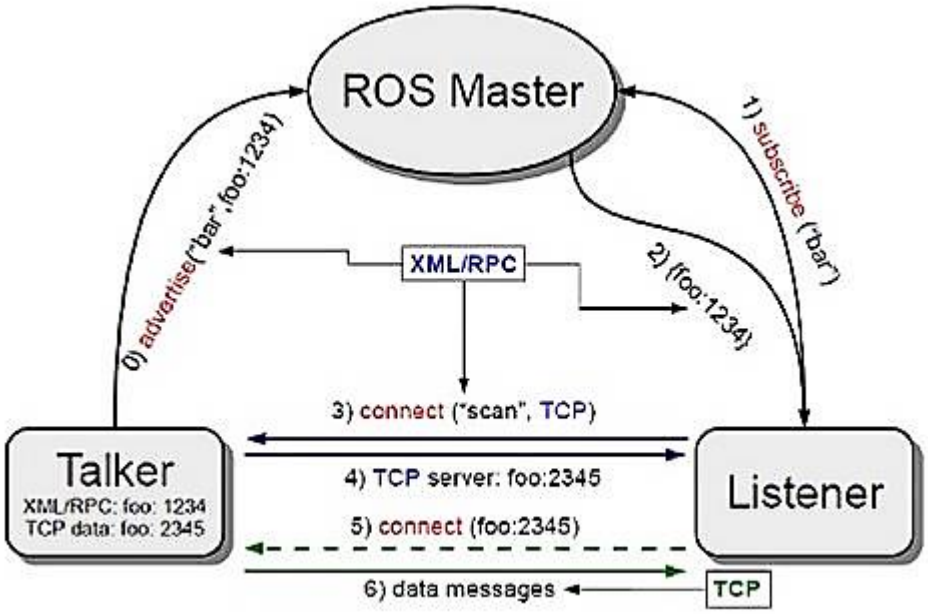
  Οι κόμβοι πρέπει να επιτελούν την ελάχιστη δυνατή λειτουργικότητα, έτσι ώστε ένα ολοκληρωμένο σύστημα να αποτελείται από πολλούς τέτοιους. Για παράδειγμα, αν θέλαμε να υλοποιήσουμε ένα σύστημα αυτόματου οδηγού, θα χρησιμοποιούσαμε τους κόμβους που φαίνονται στην Εικόνα 1 (κάθε ορθογώνιο εκφράζει έναν κόμβο). Θα υπήρχε λοιπόν ένας κόμβος που θα διάβαζε δεδομένα από την κάμερα (που χρειάζονται για να παραμείνει το αυτοκίνητο στη λωρίδα του), ένας άλλος που θα έπαιρνε δεδομένα από το GPS και θα μας έλεγε που είμαστε και ένας άλλος που θα έδινε ένα κομμάτι του χάρτη του δρόμου (για παράδειγμα ένα τετραγωνικό χιλιόμετρο) κατόπιν παραγγελίας. Ο κόμβος που λαμβάνει τα δεδομένα από την κάμερα, θα έστελνε στιγμιότυπα από αυτά στον κόμβο «Path planner», ο οποίος είναι υπεύθυνος για το σχεδιασμό της διαδρομής που θα ακολουθηθεί, αποκρινόμενο στα δεδομένα από τους αισθητήρες. Έτσι, δε χρειάζεται να «ανησυχεί» για το πώς θα πάρει τις εικόνες ή για το πού βρισκόμαστε, καθώς για αυτά είναι υπεύθυνοι οι άλλοι κόμβοι. Ο «Locomotion planner» είναι υπεύθυνος να αποφασίσει για το πώς θα κινηθούν οι τροχοί ώστε να ακολουθηθεί η τροχιά που έλαβε ως στόχο από τον «Path planner». Φυσικά, όσο γίνεται αυτό, ο «Path planner» επεξεργάζεται ήδη τον επόμενο στόχο που θα θέσει.



Εικόνα 1 - Παράδειγμα Τμηματικής (Στοιχειακής) Δομής Εφαρμογής

Ένας κόμβος εκκινείται με την εντολή rosrun:
rosrun <όνομα_πακέτου> <όνομα_εκτελέσιμου>

- Master: Είναι η οντότητα που καταχωρεί πληροφορίες σχετικές με το δίκτυο και τους κόμβους. Όταν εκκινείται ένας κόμβος εγγράφεται στο δίκτυο μέσω του master. Αυτό σημαίνει ότι ο master του δίνει μια ip διεύθυνση και τη θύρα στην οποία θα είναι διαθέσιμος, τις οποίες και καταχωρεί. Ο ρόλος του master είναι να παρέχει στους κόμβους την πληροφορία για το πού μπορούν να βρουν τον κόμβο που ζητούν. Αφού γίνει αυτό, τότε αυτοί επικοινωνούν μεταξύ τους, χωρίς πλέον τη διαμεσολάβηση του, έχουμε δηλαδή μια επικοινωνία σημείου προς σημείο (peer to peer). Όλες οι επικοινωνίες των υπόλοιπων κόμβων με το master πραγματοποιούνται με XML-RPC/TCP. Η διαδικασία φαίνεται σχηματικά στην Εικόνα 2.



Εικόνα 2 - Διαδικασία επικοινωνίας των κόμβων, με τη μεσολάβηση του master

Εκτός από την ονοματοδοσία και την εύρεση των κόμβων, ο master παρέχει επίσης τη λειτουργία του εξυπηρετητή παραμέτρων (Parameter Server), ο οποίος θα αναλυθεί στη συνέχεια.

Ο master εκκινείται με την εντολή roscore σε ένα τερματικό και αυτό είναι το πρώτο πράγμα που πρέπει να κάνει κάποιος κάθε φορά που θέλει να δουλέψει στο ROS.

- Μηνύματα: Ένα μήνυμα (message), είναι μια απλή δομή δεδομένων, που αποτελείται από πεδία, καθένα από τα οποία περιγράφεται από έναν τύπο. Υποστηρίζονται πολλοί τύποι όπως για παράδειγμα ακέραιοι (int), κινητής υποδιαστολής (floating point), Boolean και άλλοι, καθώς και συστοιχίες (arrays) αυτών. Επίσης είναι δυνατό να περιέχουν άλλες εμφωλευμένες δομές. Η δομή δεδομένων ενός μηνύματος προσδιορίζεται σε ένα απλό αρχείο κειμένου, με κατάληξη «.msg». Με όποιον από τους πιθανούς τρόπους και αν γίνεται η επικοινωνία μεταξύ των κόμβων (εκτός από τον εξυπηρετητή παραμέτρων), αυτή πραγματοποιείται με την αποστολή μηνυμάτων. Κατά τη μεταγλώττιση, τα αρχεία μηνυμάτων μεταφράζονται σε πηγαίο κώδικα, για παράδειγμα σε κλάσεις της Python ή της C++.

  Ένα πολύ απλό παράδειγμα μηνύματος, που περιέχει ένα μόνο πεδίο, είναι το String:

```
String str
```

Υπάρχει επίσης η έννοια του τύπου μηνύματος, που είναι ο τρόπος να αναφερόμαστε σε ένα είδος μηνύματος. Ένας τύπος αποτελείται από το όνομα του πακέτου στο οποίο περιέχεται το αρχείο μηνύματος, ακολουθούμενο από ένα σύμβολο «/» και το όνομα του αρχείου μηνύματος. Για παράδειγμα, αν υπάρχει το αρχείο μηνύματος std_msgs/msg/String.msg, όπου std_msgs είναι το όνομα του πακέτου και msg είναι το όνομα ενός υποφακέλου μέσα σε αυτό, ο αντίστοιχος τύπος μηνύματος είναι std_msgs/String.

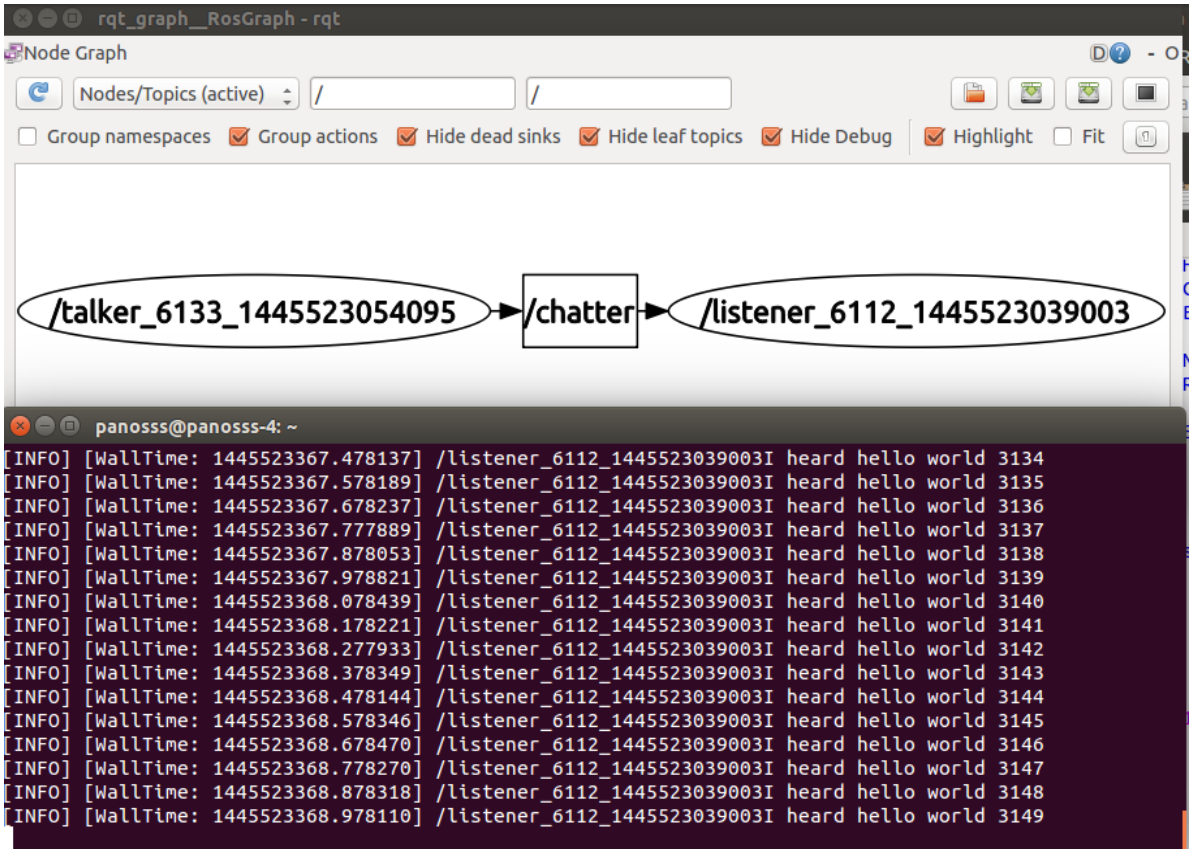
Στο ROS υπάρχουν έτοιμα, ήδη ορισμένα μηνύματα που διανέμονται στα πλαίσια της λήψης των βασικών πακέτων του, όπως για παράδειγμα αυτά που περιέχονται στον κατάλογο std_msgs. Η χρησιμότητά τους είναι ότι επιτυγχάνουν μια προτυποποίηση των διεπαφών της επικοινωνίας. Φυσικά, κάθε χρήστης του ROS μπορεί να δημιουργήσει και τις δικές του δομές δεδομένων, δηλαδή τα δικά του μηνύματα.

2.2.2 Μηχανισμοί επικοινωνίας

Στο ROS δεν υπάρχει μόνο ένας τρόπος επικοινωνίας των κόμβων. Ωστόσο, ο πιο συχνά χρησιμοποιούμενος είναι σίγουρα το μοντέλο εκδότη/συνδρομητή (publisher/subscriber). Πρόκειται για ένα μηχανισμό ο οποίος δεν αποτελεί καινοτομία αυτού του συστήματος, αλλά συναντιέται συχνά, ίσως με διαφορετική ορολογία. Η ιδέα είναι ότι υπάρχουν διεργασίες - παραγωγοί και διεργασίες - καταναλωτές δεδομένων, με σκοπό να διαχωριστεί η παραγωγή από την κατανάλωση της πληροφορίας. Οι παραγωγοί (που εδώ ονομάζονται εκδότες) «διαφημίζουν» στο master, ότι τα δεδομένα που πρόκειται να παράγουν θα αφορούν ένα συγκεκριμένο θέμα (topic). Άλλοι κόμβοι, που λειτουργούν ως καταναλωτές δεδομένων, «εγγράφονται» σε topics στα οποία θέλουν να «ακούν» για καινούρια δεδομένα. Από αυτήν την οπτική γωνία, θα μπορούσαμε να παρομοιάσουμε τα topics με πίνακες ανακοινώσεων. Όταν ένας publisher καλεί μια συνάρτηση publish, το μήνυμα σειριοποιείται από το σύστημα και λαμβάνεται από όποιον κόμβο είναι εγγεγραμμένος στο συγκεκριμένο topic. Σε κάθε topic, τα δεδομένα μπορούν να έχουν τη μορφή μόνο ενός συγκεκριμένου τύπου μηνύματος, καθορισμένου για αυτό, αλλιώς ο subscriber δεν μπορεί να τα δεχτεί.

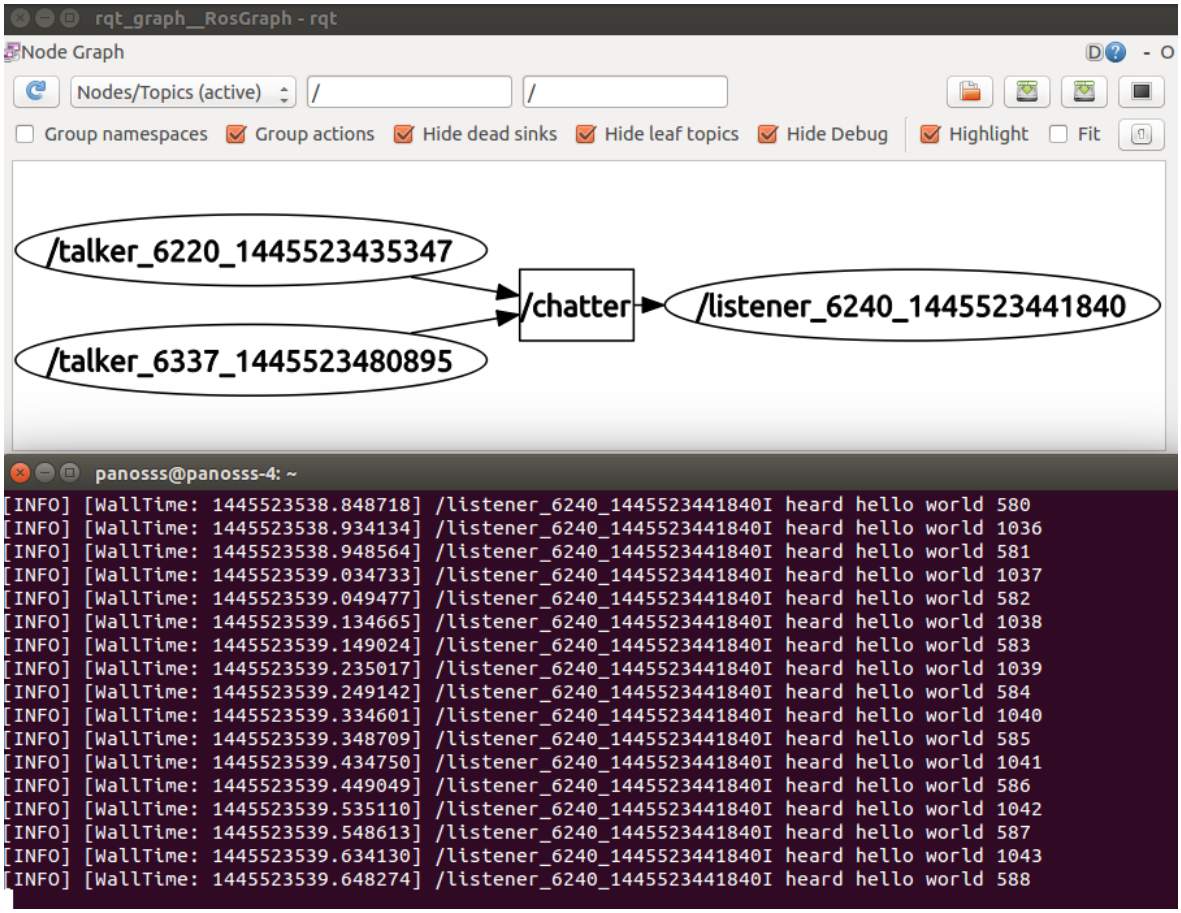
Όταν ένας κόμβος δηλώσει ότι θέλει να παρακολουθεί ένα συγκεκριμένο topic, ο master εκδίδει για αυτόν μια λίστα με τους κόμβους οι οποίοι δημοσιεύουν δεδομένα σε αυτό, η οποία όπως έχουμε ήδη πει περιέχει τις ip διευθύνσεις τους και τις θύρες τους. Η επικοινωνία τελικά πραγματοποιείται απευθείας μεταξύ των συνεργαζόμενων κόμβων (peer to peer) και χρησιμοποιείται το πρωτόκολλο TCP. Στην πραγματικότητα δηλαδή, τα topics μοιάζουν περισσότερο με διαύλους επικοινωνίας, οι οποίοι διακρίνονται μοναδικά από το όνομα τους.

Σε έναν τέτοιο δίαυλο, μπορούν να εκδίδουν δεδομένα πολλοί publishers και να τα λαμβάνουν πολλοί subscribers (στο εξής για αυτούς τους κόμβους θα χρησιμοποιείται η αγγλική ορολογία). Αυτό φαίνεται και στις παρακάτω εικόνες. Στην Εικόνα 3 έχουμε το παράδειγμα δύο κόμβων, του talker και του listener, οι οποίοι επικοινωνούν στο topic με όνομα chatter. Ο talker εκδίδει περιοδικά (για παράδειγμα 10 φορές το δευτερόλεπτο) ένα μήνυμα που περιέχει τη φράση «hello world», ακολουθούμενη από έναν ακέραιο, ο οποίος αυξάνει για κάθε επόμενο μήνυμα. Το τερματικό που απεικονίζεται, είναι αυτό στο οποίο τρέχει ο listener, που ακούει σε αυτό το topic.

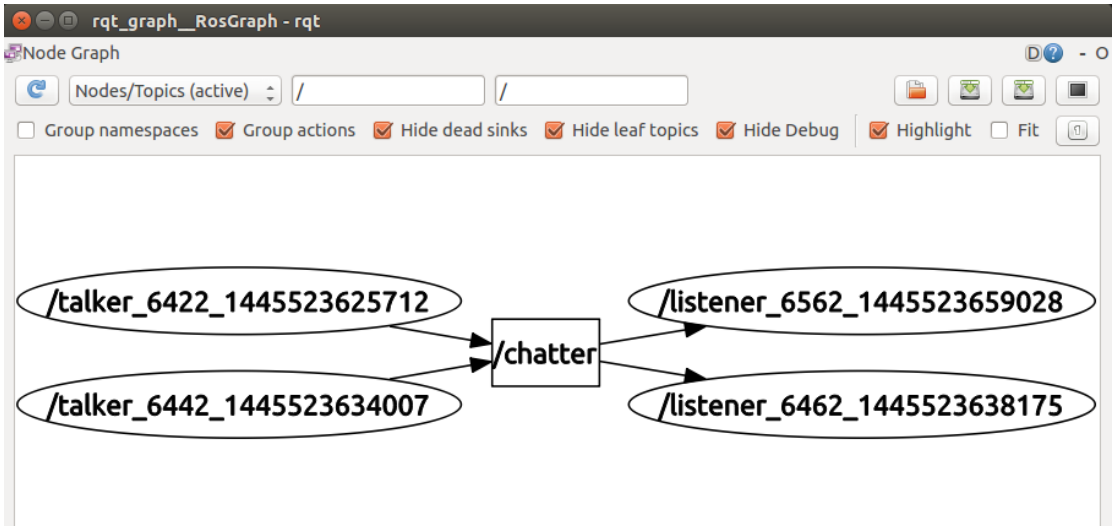


Εικόνα 3 - Απεικόνιση του γράφου επικοινωνίας στο εργαλείο rqt-graph

Στην Εικόνα 4 βλέπουμε την περίπτωση όπου στο ίδιο topic εκδίδει πλέον και ένας δεύτερος κόμβος, ο talker2, που κάνει το ίδιο με τον talker1. Παρατηρούμε ότι στα μηνύματα που ακούει ο listener οι ακέραιοι αριθμοί δεν αυξάνονται πάντα από μήνυμα σε μήνυμα και αυτό συμβαίνει γιατί τώρα λαμβάνει μηνύματα από δύο διαφορετικούς κόμβους, ο ένας εκ των οποίων ξεκίνησε πιο αργά από τον άλλον. Από αυτό το παράδειγμα συμπεραίνουμε ότι οι κόμβοι δεν ενδιαφέρονται και ούτε είναι ενήμεροι για το ποια είναι η πηγή των δεδομένων που λαμβάνουν (αντίστοιχα και ένας publisher δεν ενδιαφέρεται για το ποιοι κόμβοι τον ακούν, ούτε και για το αν τον ακούει κάποιος). Στην Εικόνα 5 βλέπουμε το διάγραμμα του συστήματος μετά την προσθήκη ενός ακόμα κόμβου που ακούει στο ίδιο topic.



Εικόνα 4 - Επικοινωνία με topic 1xn

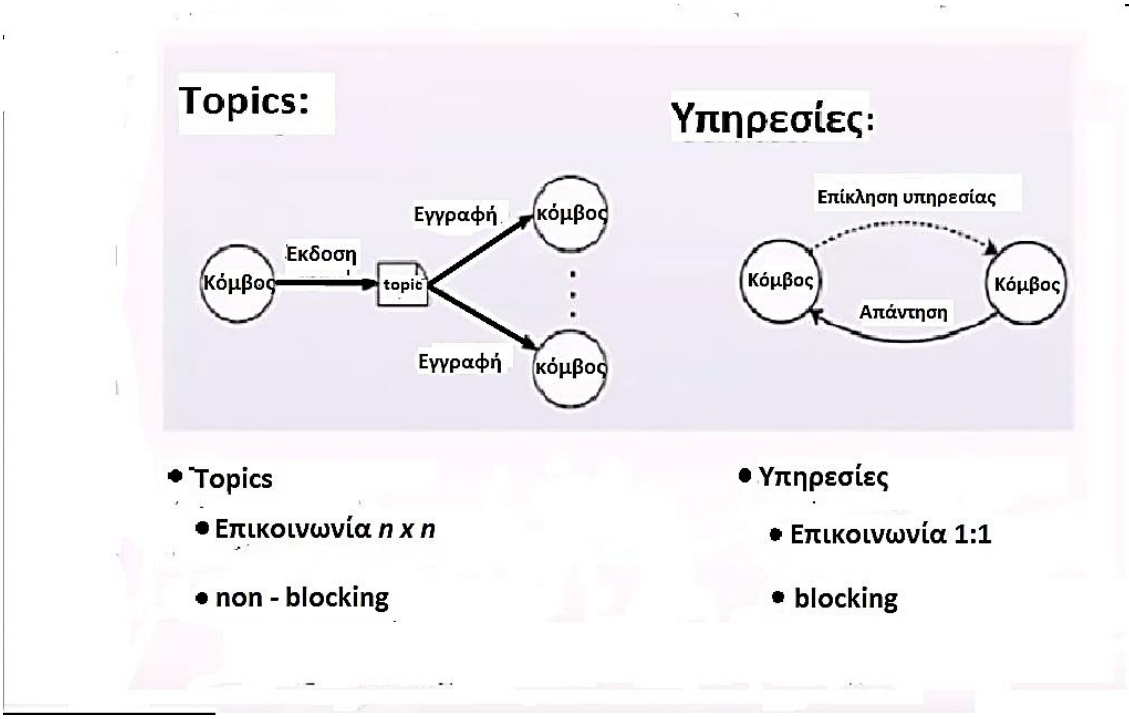


Εικόνα 5 - Επικοινωνία με topic, nxn

Διαγράμματα όπως αυτά που μπορείτε να δείτε στις παραπάνω εικόνες, με τα οποία διευκολύνεται η εποπτεία του συστήματος, είναι εύκολα προσβάσιμα

από το εργαλείο rqt-graph και, φυσικά, διαμορφώνονται δυναμικά καθώς αφαιρούμε ή προσθέτουμε κόμβους στο δίκτυο, αποτυπώνοντας πάντα αυτές τις αλλαγές. Αυτή η προσθαφαίρεση κόμβων από το δίκτυο, γίνεται χωρίς να επηρεάζεται το υπόλοιπο σύστημα, ακόμα και αν μιλάμε για κόμβους οι οποίοι είναι άμεσα συνδεδεμένοι, κάτι που είναι αποτέλεσμα του διαχωρισμού της παραγωγής από την κατανάλωση των δεδομένων που επιτυγχάνεται, ιδιότητα στην οποία αναφερθήκαμε νωρίτερα. Αυτή η ιδιότητα είναι πολύ χρήσιμη στην περίπτωση που θέλουμε να αποσφαλματώσουμε ή γενικότερα να αλλάξουμε κάτι σε έναν κόμβο: δε χρειάζεται να τερματίσουμε όλο το σύστημα, παρά μόνο αυτόν. Στη συνέχεια τον μεταγλωττίζουμε ξανά, τον ξανατρέχουμε και επανέρχονται όλες οι συνδέσεις του με τους άλλους κόμβους.

Η επικοινωνία με το μοντέλο publisher/subscriber είναι ευέλικτη, για τους λόγους που αναφέρθηκαν πιο πάνω, όμως υπάρχουν και περιπτώσεις για τις οποίες αυτή η μέθοδος δεν είναι η κατάλληλη. Αναφερόμαστε στις περιπτώσεις όπου η επικοινωνία θα πρέπει να είναι της μορφής αποστολής ενός αιτήματος (request) από την πλευρά ενός κόμβου, ακολουθούμενης από την αντίστοιχη απάντηση (reply) από την καλούμενη διεργασία – κόμβο. Εκεί δηλαδή όπου χρειάζεται η επικοινωνία να γίνεται και προς τις δύο κατευθύνσεις και, επιπλέον, να αφορά μόνο δύο κόμβους, χαρακτηριστικά τα οποία δεν υπάρχουν στο προαναφερόμενο μοντέλο. Για αυτές τις περιπτώσεις, υποστηρίζεται ένας άλλος τρόπος επικοινωνίας, το μοντέλο των υπηρεσιών (services). Οι διαφορές των δύο μοντέλων απεικονίζονται και σχηματικά στην Εικόνα 6.



Εικόνα 6 - Σύγκριση topics - υπηρεσιών

Μια κλήση μιας υπηρεσίας από έναν κόμβο (ο οποίος ονομάζεται πελάτης (client) της υπηρεσίας) μοιάζει με την κλήση μιας συνάρτησης, ή μιας απομακρυσμένης διεργασίας (remote procedure call - RPC). Η μορφή των δεδομένων που θα σταλούν μέσω μιας υπηρεσίας είναι συγκεκριμένη και ορίζεται με τη χρήση αρχείων με κατάληξη «.srv», κατά τον ίδιο τρόπο που η μορφή των δεδομένων που θα σταλούν μέσω ενός topic ορίζονται από τα αρχεία μηνυμάτων. Τα αρχεία υπηρεσιών λοιπόν μοιάζουν με τα αρχεία μηνυμάτων και, στην πραγματικότητα, αναλύονται στο ROS σε δύο μηνύματα, ένα που αφορά το κομμάτι της αίτησης και ένα άλλο για την απάντηση. Για να γίνει αυτό πιο συγκεκριμένο, παραθέτουμε ως παράδειγμα ένα αρχείο υπηρεσίας με όνομα AddTwoInts.srv.

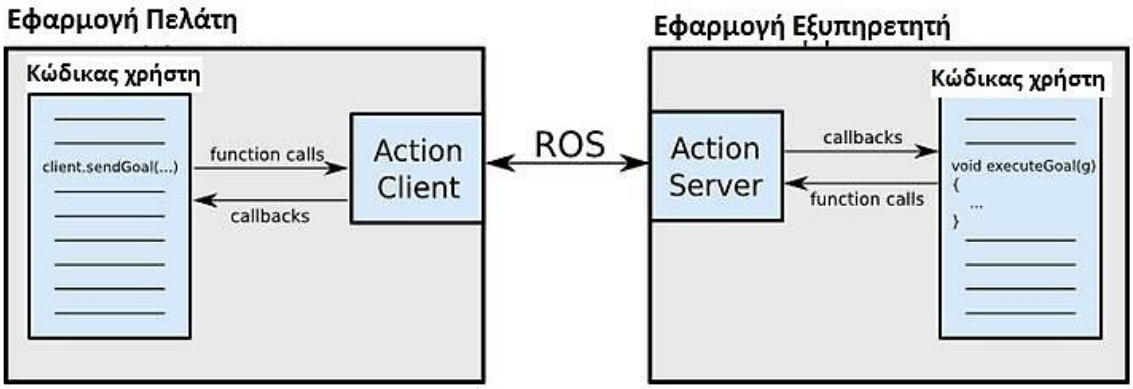
```
int64 a
int64 b
---
int64 sum
```

Παρατηρούμε ότι και πάλι έχουμε τον ορισμό κάποιων πεδίων (με ονόματα a, b, c), για κάθε ένα από τα οποία ορίζουμε και έναν τύπο (όλα τα πεδία έχουν τύπο int). Αυτήν τη φορά όμως, τα πεδία χωρίζονται σε δύο ομάδες, με τη χρήση των «---». Αυτά που βρίσκονται πάνω από τις παύλες ορίζουν το μήνυμα της αίτησης, ενώ αυτά που βρίσκονται από κάτω το μήνυμα της απάντησης. Αυτό το αρχείο όταν αναλυθεί από το σύστημα θα προκαλέσει τη δημιουργία δύο αρχείων με όνομα AddTwoIntsRequest.msg και AddTwoIntsResponse.msg αντίστοιχα. Αυτό το αρχείο υπηρεσίας λοιπόν ορίζει την αποστολή δύο ακεραίων στην υπηρεσία και την επιστροφή από αυτήν στον κόμβο που την κάλεσε ενός ακεραίου. Ένα τέτοιο αρχείο υπηρεσίας θα μπορούσε να χρησιμοποιηθεί για την πρόσθεση δύο αριθμών.

Αρκετά συχνά η εκτέλεση μιας υπηρεσίας που ζητάει κάποιος κόμβος – πελάτης απαιτεί χρόνο για την ολοκλήρωσή της. Για παράδειγμα, μπορεί να ζητηθεί η μετακίνηση της βάσης ενός ρομπότ σε μια θέση (με τις συντεταγμένες της θέσης να είναι το μήνυμα της αίτησης) ή ο εντοπισμός της λαβής μιας πόρτας. Σε τέτοιες περιπτώσεις, ίσως χρειάζεται αυτή η εργασία που καλείται να διεκπεραιώσει ο άλλος κόμβος να μπορεί να ακυρωθεί ή μπορεί ο πελάτης της υπηρεσίας να θέλει να λαμβάνει περιοδικά ανάδραση σχετικά με την εξέλιξη του αιτήματός του. Όταν θέλουμε να υπάρχει και αυτή η πρόσθετη λειτουργικότητα, μπορούμε να χρησιμοποιήσουμε έναν πιο εξελιγμένο τρόπο επικοινωνίας, που μοιάζει με τις υπηρεσίες αλλά είναι πιο σύνθετος. Πρόκειται για το πρωτόκολλο action. Τα εργαλεία για τη δημιουργία πελατών και εξυπηρετητών που επικοινωνούν

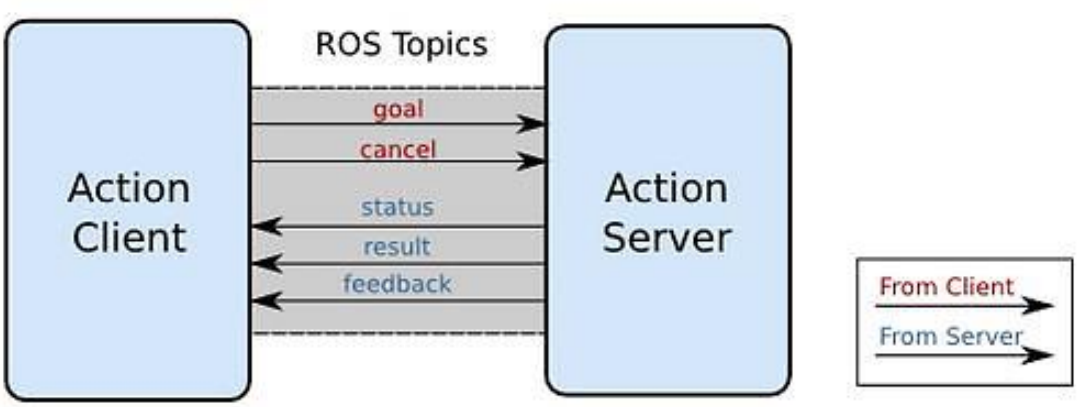
σύμφωνα με αυτό το πρωτόκολλο παρέχονται με το πακέτο actionlib, για το οποίο μπορείτε να διαβάσετε στο wiki.ros.org/actionlib.

Οι οντότητες που επικοινωνούν μέσω της υποδομής του ROS ονομάζονται Action Client (δημιουργείται στην εφαρμογή που θέλει να στέλνει τις αιτήσεις) και Action Server (στην πλευρά της εφαρμογής που προορίζεται να εκτελέσει τη ζητούμενη εργασία). Οι μέθοδοι αυτών των οντοτήτων παρέχουν σε αυτές τις εφαρμογές τις διεπαφές για την αίτηση στόχων (στην πλευρά του πελάτη) ή για την εκτέλεση στόχων (στην πλευρά της εφαρμογής του εξυπηρετητή). Αυτό φαίνεται σχηματικά στην Εικόνα 7.



Εικόνα 7 - Επικοινωνία πελάτη - εξυπηρετητή στο μοντέλο επικοινωνίας actions

Ο Action Client και ο Action Server επικοινωνούν με μηνύματα, η δομή των οποίων – όπως έχουμε πει – προσδιορίζεται από τα αρχεία μηνυμάτων, μέσω topics. Στην περίπτωση της επικοινωνίας με actions (η μετάφραση στα ελληνικά δεν αποτυπώνει ακριβώς το νόημα, οπότε θα χρησιμοποιούμε τον αγγλικό όρο), χρειάζεται να υπάρχουν πολλά τέτοια αρχεία, λόγω της φύσης αυτού του είδους επικοινωνίας. Βασικά, τα topics πάνω στα οποία γίνεται η επικοινωνία ενός action client και ενός action server, είναι πέντε και φαίνονται στην Εικόνα 8, οπότε εύκολα συμπεραίνει κάποιος ότι χρειάζεται να έχουν οριστεί πέντε τύποι μηνυμάτων.



Εικόνα 8 - Γενικό σχήμα διεπαφής στο μοντέλο actions

Όπως μπορούμε να δούμε στην παραπάνω εικόνα, πρέπει να υπάρχει ένα topic στο οποίο ο Action Client θα στέλνει το στόχο (goal). Επίσης, ένα topic στο οποίο ο Action Server θα ενημερώνει τον Action Client σχετικά με το αποτέλεσμα (result) της εργασίας που εκτελέστηκε στην εφαρμογή - εξυπηρετητή. Θα μπορούσε να είναι το αποτέλεσμα ενός υπολογισμού που του ζητήθηκε να εκτελέσει. Σε άλλες περιπτώσεις η αποστολή του αποτελέσματος δεν είναι τόσο σημαντική. Για παράδειγμα, αν το ζητούμενο είναι να κινηθεί το ρομπότ με στόχο να πάρει μια συγκεκριμένη στάση, το μήνυμα του αποτελέσματος θα περιείχε τις τελικές συντεταγμένες των αρθρώσεων του ρομπότ. Από την άλλη, το topic που περιγράφεται στην εικόνα ως feedback (στα ελληνικά ανάδραση), αφορά τα μηνύματα που πιθανώς να χρειάζεται να στέλνει ο Action Server στον Action Client για να τον ενημερώσει για την πορεία της εργασίας που εκτελείται στην εφαρμογή του. Για κάθε στόχο, είναι υποχρεωτικό να στέλνεται ακριβώς ένα (έστω και κενό) μήνυμα αποτελέσματος, που θα σημαίνει το τέλος της επεξεργασίας του συγκεκριμένου στόχου, ενώ η αποστολή δεδομένων ανάδρασης δεν είναι υποχρεωτική.

Για αυτά τα τρία topics, πρέπει να οριστούν τα αντίστοιχα μηνύματα που θα χρησιμοποιηθούν. Αυτό γίνεται με ένα αρχείο με κατάληξη «.action», το οποίο έχει πάντα μια συγκεκριμένη μορφή. Στο πρώτο πεδίο ορίζεται ο στόχος (δηλαδή το μήνυμα που θα χρησιμοποιηθεί για την περιγραφή του στόχου). Στη συνέχεια ορίζεται το αποτέλεσμα και στο τέλος η ανάδραση. Τα πεδία αυτά διαχωρίζονται από «---» Παρακάτω βλέπουμε πώς θα ήταν ένα αρχείο προσδιορισμού ενός action στο οποίο η εφαρμογή του εξυπηρετητή θα έπρεπε να υπολογίσει μια ακολουθία Fibonacci. Η ακολουθία Fibonacci ορίζεται από τον αναδρομικό τύπο $F_n = F_{n-1} + F_{n-2}$, με $F_0 = 0$ και $F_1 = 1$. Σε αυτό το παράδειγμα ο στόχος που θα σταλεί από τον Action Client θα ορίζει την τάξη της ακολουθίας, δηλαδή μέχρι ποιον όρο θα πρέπει υπολογίσει η εφαρμογή του εξυπηρετητή. Παρεμπιπτόντως, με αφορμή αυτό το παράδειγμα θα πρέπει να σημειώσουμε ότι συνήθως, τα δεδομένα που

αποστέλλονται ως στόχοι έχουν περισσότερο τη σημασία των ορισμάτων κατά την κλήση μιας συνάρτησης. Δεν έχουν δηλαδή τη μορφή εντολών, αλλά αντίθετα το τι θα γίνει όταν φτάσει κάποιο τέτοιο μήνυμα ορίζεται στην πλευρά της εφαρμογής του εξυπηρετητή. Αυτό βέβαια θα γίνει περισσότερο κατανοητό στην παράγραφο 2.5, όπου εξηγείται η χρήση της βιβλιοθήκης actionlib.

```
#ορισμός του στόχου
int32 order
---
#ορισμός του αποτελέσματος
int32[] sequence
---
#ορισμός της ανάδρασης
int32[] sequence
```

Ας συνεχίσουμε με την εξήγηση της λειτουργίας του παραπάνω αρχείου. Αρχικά ας το ονομάσουμε Fibonacci.action. Από αυτό, το σύστημα του ROS και πιο συγκεκριμένα το genaction.py θα δημιουργήσει εφτά αρχεία μηνυμάτων, με τα εξής ονόματα:

FibonacciGoal.msg

FibonacciResult.msg

FibonacciFeedback.msg

FibonacciAction.msg

FibonacciActionGoal.msg

FibonacciActionResult.msg

FibonacciActionFeedback.msg

Τα πρώτα τρία ορίζουν τη μορφή του στόχου, του αποτελέσματος και της ανάδρασης αντίστοιχα. Τα υπόλοιπα, είναι όπως τα προηγούμενα, όμως περιέχουν και ένα πρόσθετο πεδίο, το οποίο ονομάζεται Goal ID και συσχετίζει τα δεδομένα που μεταφέρονται με ένα συγκεκριμένο στόχο που βρίσκεται υπό επεξεργασία. Τα τελευταία είναι αυτά που στην πραγματικότητα χρησιμοποιούνται στην επικοινωνία μέσω των topic.

Στην Εικόνα 8, είδαμε ότι υπάρχουν πέντε topics, όμως το αρχείο ορισμού του action αναφέρεται μόνο σε τρία μηνύματα. Αυτό συμβαίνει γιατί τα μηνύματα που χρησιμοποιούνται στα άλλα δύο topics υπάρχουν ήδη στο ROS. Πιο συγκεκριμένα, στο topic που αφορά την αίτηση ακύρωσης από έναν Action Client σε έναν Action Server για ένα συγκεκριμένο στόχο (cancel topic) χρησιμοποιείται ο τύπος μηνύματος actionlib_msgs/GoalID, ο οποίος περιέχει δύο πεδία (timestamp και goalID). Ο τρόπος με τον οποίο θα συμπληρωθούν αυτά δείχνει ποιος/ποιοι στόχοι ζητείται να ακυρωθούν.

Από την άλλη, το status topic χρησιμοποιεί τον τύπο μηνύματος actionlib_msgs/GoalStatusArray και μέσω αυτού ο Action Client ενημερώνεται για την κατάσταση του στόχου για κάθε έναν από τους στόχους τους οποίους παρακολουθεί ο Action Server. Αυτό επιδέχεται λίγο περισσότερη ανάλυση: Όταν ληφθεί κάποιος στόχος από έναν Action Server, αυτός δημιουργεί μια μηχανή καταστάσεων (state machine) για να παρακολουθεί την κατάσταση του (αντιστοιχεί μια μηχανή καταστάσεων για κάθε στόχο). Οι βασικές καταστάσεις είναι:

- σε αναμονή (pending) – είναι η πρώτη κατάσταση από την οποία περνάει κάθε στόχος μόλις ληφθεί από τον action server
- ενεργός (active) – αν τη δεχθεί ο action server
- ανακλημένος (recalled) – μεταβαίνει σε αυτήν την τελική κατάσταση αν, ενώ βρίσκεται στην κατάσταση pending, ζητηθεί η ακύρωσή της από τον action client (και γίνει δεκτή η αίτηση ακύρωσης από την εφαρμογή του εξυπηρετητή)
- ακυρωμένος (preempted) – μεταβαίνει σε αυτήν την τελική κατάσταση αν ζητηθεί η ακύρωσή της από τον action client και γίνει δεκτή η αίτηση αυτή από την εφαρμογή του εξυπηρετητή, ενώ ο στόχος βρίσκεται στην κατάσταση active.
- επιτυχημένος (succeeded) – όταν ο στόχος έχει επιτευχθεί από την εφαρμογή του εξυπηρετητή.
- ματαιωμένος (aborted) – αν τερματιστεί από τον εξυπηρετητή χωρίς να ζητηθεί από τον πελάτη.
- απορριφθέντας (rejected) – αν απορριφθεί από τον εξυπηρετητή πριν ακόμα αυτός αρχίσει να τον επεξεργάζεται

Εξυπηρετητής Παραμέτρων

Κλείνοντας την παρουσίαση των τρόπων επικοινωνίας στο ROS, δεν θα μπορούσαμε να μην κάνουμε και μια αναφορά στον εξυπηρετητή παραμέτρων. Πρόκειται στην ουσία για ένα λεξικό, που είναι διαθέσιμο προς τους κόμβους του συστήματος. Αυτοί τον χρησιμοποιούν για να αποθηκεύουν και να βρίσκουν παραμέτρους κατά το χρόνο εκτέλεσής τους. Δεν είναι όμως σχεδιασμένος για υψηλή αποδοτικότητα,

γι’ αυτόν το λόγο χρησιμοποιείται κυρίως για αποθήκευση στατικών δεδομένων, όπως για παραμέτρους διαμόρφωσης. Είναι ορατός από όλους τους κόμβους και χρησιμοποιείται από τα εργαλεία του ROS, ώστε αυτά να μπορούν να επιβλέπουν την κατάσταση του συστήματος και να κάνουν αλλαγές όποτε χρειάζεται.

2.3 Το Σύστημα Αρχείων

Τα αρχεία στο ROS οργανώνονται, όπως έχει ήδη αναφερθεί, σε πακέτα. Ένα πακέτο μπορεί να περιέχει κόμβους, μια ανεξάρτητη του ROS βιβλιοθήκη, μια δομή δεδομένων, αρχεία διαμόρφωσης (configuration files), κάποιο κομμάτι λογισμικού που προέρχεται από άλλη πλατφόρμα και οτιδήποτε άλλο συνιστά λογικά μια χρήσιμη μονάδα. Τα πακέτα έχουν δύο ρόλους: Από τη μια πλευρά αποτελούν τον τρόπο με τον οποίο «τυλίγονται» οι διάφορες μονάδες λογισμικού με σκοπό τη διάθεσή τους στο δίκτυο του ROS. Από την άλλη, είναι και η πιο μικρή, δομική μονάδα «χτισίματος» σε αυτό. Το τελευταίο χρειάζεται περισσότερη ανάλυση.

Χτίσιμο (building) είναι η διαδικασία δημιουργίας «στόχων» (targets) από έναν πηγαίο κώδικα, οι οποίοι μπορούν να χρησιμοποιηθούν από έναν τελικό χρήστη. Αυτοί οι στόχοι μπορεί να είναι στη μορφή βιβλιοθηκών, εκτελέσιμων αρχείων, δημιουργούμενων αρχείων, εξαγόμενων διεπαφών (πχ C++ header files) ή οτιδήποτε άλλο δεν αποτελεί στατικό κώδικα. Για να χτίσει τους στόχους, το σύστημα χτισίματος (build system) χρειάζεται κάποιες πληροφορίες, όπως είναι οι τοποθεσίες των στοιχείων ενός toolchain (για παράδειγμα του compiler της C++), οι τοποθεσίες του πηγαίου κώδικα, οι εξαρτήσεις του κώδικα, εξωτερικές εξαρτήσεις, πού βρίσκονται αυτές, ποιοι στόχοι θα πρέπει να χτιστούν, που θα πρέπει να γίνει αυτό και πού θα πρέπει να εγκατασταθούν. Αυτά ορίζονται συνήθως σε κάποια αρχεία διαμόρφωσης τα οποία διαβάζονται από το σύστημα αρχείων.

Ένα από τα πιο γνωστά συστήματα χτισίματος είναι το CMake. Το επίσημο σύστημα χτισίματος του ROS ονομάζεται catkin και είναι βασισμένο σε αυτό. Για την ακρίβεια, συνδυάζει μακροεντολές του CMake και διαφόρα Python scripts, για να παρέχει κάποια πρόσθετη λειτουργικότητα σε σχέση με τη συνηθισμένη του CMake.

Γιατί όμως το ROS έχει το δικό του σύστημα χτισίματος; Για την ανάπτυξη μεμονωμένων προγραμμάτων, ένα συνηθισμένο σύστημα όπως το CMake είναι αρκετό. Παρ’ όλα αυτά, τέτοια εργαλεία είναι δύσκολο να χρησιμοποιηθούν από μόνα τους για μεγάλα, σύνθετα συστήματα στα οποία ενδεχομένως να χρησιμοποιείται και ετερογενής κώδικας, κυρίως λόγω του υψηλού αριθμού των εξαρτήσεων, της σύνθετης οργάνωσης του κώδικα και των κανόνων που θα μπορούσε να έχει κάποιος συγκεκριμένος στόχος, σε συνδυασμό με τη γενικότητα που διακρίνει αυτά τα εργαλεία. Το ROS είναι μια συλλογή από πακέτα, τα οποία διανέμονται ανεξάρτητα, όμως συνηθίζεται να έχουν εξαρτήσεις μεταξύ τους και

χρησιμοποιούν διαφορετικές γλώσσες προγραμματισμού, εργαλεία και πρότυπα οργάνωσης κώδικα. Εξαιτίας αυτού, η διαδικασία χτισίματος για ένα στόχο σε κάποιο πακέτο μπορεί να είναι εντελώς διαφορετική από τον τρόπο με τον οποίο χτίζεται κάποιος άλλος στόχος. Το catkin στοχεύει στη βελτίωση της ανάπτυξης μεγάλων ομάδων συσχετιζόμενων πακέτων με ένα σταθερό και συμβατικό τρόπο, κάνοντας το χτίσιμο και το τρέξιμο του κώδικα στο ROS ευκολότερο και απλούστερο και διευκολύνοντας την αποτελεσματική ανταλλαγή κώδικα βασισμένου στο ROS.

Με άλλα λόγια, τα πακέτα catkin, μπορούν να χτιστούν αυτόνομα, με τον τρόπο που αυτό συμβαίνει σε μια συνηθισμένη περίπτωση χρήσης του CMake, αλλά με το catkin παρέχεται και η έννοια των χώρων εργασίας (workspaces), που είναι ένας φάκελος στον οποίο ο χρήστης μπορεί να χτίσει πολλά αλληλεξαρτώμενα πακέτα μαζί, ταυτόχρονα. Ένας catkin χώρος εργασίας περιέχει μέχρι και τέσσερις διαφορετικούς χώρους, καθένας εκ των οποίων παίζει και ένα διαφορετικό ρόλο στη διαδικασία της ανάπτυξης λογισμικού.

Χώρος πηγής (source space): Περιέχει τον πηγαίο κώδικα των catkin πακέτων. Κάθε φάκελος σε αυτόν το χώρο περιέχει ένα ή περισσότερα catkin πακέτα. Ο χώρος αυτός πρέπει να παραμείνει ανεπηρέαστος από τη διαδικασία του χτισίματος ή της εγκατάστασης.

Χώρος χτισίματος (build space): Είναι ο χώρος στον οποίο καλείται η CMake να χτίσει τα πακέτα που υπάρχουν στο χώρο source. Η CMake και το catkin κρατάνε τις «κρυφές» (cache) πληροφορίες που επεξεργάζονται, καθώς και άλλα ενδιάμεσα αρχεία. Αυτός δεν είναι απαραίτητο να περιέχεται στο χώρο εργασίας ούτε να βρίσκεται έξω από το χώρο source, όμως αυτό είναι το προτεινόμενο.

Χώρος ανάπτυξης (devel space): Είναι ο χώρος στον οποίο τοποθετούνται οι στόχοι του χτισίματος πριν να εγκατασταθούν. Ο τρόπος με τον οποίο είναι οργανωμένοι οι στόχοι σε αυτόν είναι ο ίδιος με τη διάταξή που θα έχουν αφού εγκατασταθούν. Αυτό προσφέρει ένα χρήσιμο περιβάλλον δοκιμών και ανάπτυξης που δεν απαιτεί την επίκληση του βήματος της εγκατάστασης.

Χώρος install: Είναι ο χώρος στον οποίο εγκαθίστανται οι στόχοι μόλις χτιστούν. Δεν είναι απαραίτητο να βρίσκεται μέσα στο χώρο εργασίας.

Ο διαχωρισμός του χώρου ανάπτυξης από το χώρο εγκατάστασης είναι μια καινοτομία που έφερε το catkin σε σχέση με το σύστημα που χρησιμοποιούνταν αρχικά στο ROS, το οποίο ονομαζόταν rosbuild.

Η λειτουργία του catkin εξαρτάται από μεταβλητές περιβάλλοντος. Αυτές ορίζονται μέσω ενός αρχείου εγκατάστασης περιβάλλοντος (environment setup file), το οποίο ονομάζεται setup.*. Η κατάληξη ενός αρχείου εγκατάστασης

εξαρτάται από το κέλυφος (shell) το οποίο χρησιμοποιεί ο χρήστης, που μπορεί να είναι το bash, το zsh ή το sh.

Το ROS είναι μια πολύ μεγάλη συλλογή από πακέτα, πολλά από τα οποία προστίθενται και μεταβάλλονται συνέχεια. Λόγω της πολυπλοκότητάς του, συνήθως η εγκατάστασή του γίνεται χρησιμοποιώντας ήδη «χτισμένα» (prebuilt) δυαδικά πακέτα (όπως τα πακέτα deb στα Ubuntu). Αυτά εγκαθίστανται στο σύστημα, συνήθως στο /opt/ros/indigo (μιλώντας για την εγκατάσταση της έκδοσης indigo). Τέτοια ήταν και η εγκατάσταση της οποίας τα βήματα δείχτηκαν στην παράγραφο 2.1. Ο χρήστης έχει την επιλογή να χτίσει τα δικά του πακέτα σε αντιπαράθεση με μια συγκεκριμένη έκδοση ή να αναμίξει και να ταιριάξει κομμάτια από διάφορες εγκατεστημένες εκδόσεις, καθώς και δικές του εκδοχές (versions) ενός πακέτου. Για παράδειγμα, κάποιος θα μπορούσε να κάνει αναβαθμίσεις ενός πακέτου όπως το nodelet_core μέσα στο χώρο εργασίας του. Θα ήθελε να τροποποιήσει τον κώδικα, να το χτίσει και κάθε στόχος ο οποίος εξαρτάται από στόχους του nodelet_core (είτε κατά το build είτε κατά το runtime) να χρησιμοποιεί αυτήν την εκδοχή και όχι την εκδοχή που είναι εγκατεστημένη από το σύστημα. Για να γίνει αυτό, το ROS έχει ένα σύστημα επικαλύψεων (overlays), όπου το σύστημα χτισίματος μπορεί να διατρέξει πολλαπλές εγκαταστάσεις πακέτων για να βρει εξαρτήσεις.

Το ROS βασίζεται στην ιδέα του συνδυασμού χώρων εργασίας, με τη χρήση του περιβάλλοντος κελύφους. Κάθε επικάλυψη συνδέεται με ένα αρχείο εγκατάστασης. Όταν εκτελείται η source με όρισμα ένα catkin αρχείο εγκατάστασης, το αρχείο εγκατάστασης αντικαθιστά τις υπάρχουσες μεταβλητές περιβάλλοντος, αντί να τις επεκτείνει. Όταν κάποιος χτίζει κώδικα μέσα στο περιβάλλον εργασίας του (workspace), αρχεία setup δημιουργούνται μέσα στο χώρο devel. Όταν αυτά γίνονται source, κάθε αρχείο setup από άλλα περιβάλλοντα εργασίας που χρησιμοποιήθηκε κατά το χρόνο χτισίματος γίνεται επίσης αυτόματα source. Για παράδειγμα αν εκτελεστεί

```
source /opt/ros/indigo/setup.bash
```

και στη συνέχεια χτιστεί ένα περιβάλλον εργασίας, το να γίνει source το setup.bash αρχείο στο χώρο devel θα προκαλέσει και την ανάκτηση του /opt/ros/indigo/setup.bash. Παρομοίως, αν εγκατασταθεί ο χώρος εργασίας, ο χώρος εγκατάστασης θα περιέχει ένα αρχείο setup.bash το οποίο θα τοποθετήσει το χώρο εγκατάστασης πάνω στη βάση κάθε άλλου περιβάλλοντος εργασίας που χρησιμοποιήθηκε για να χτιστεί ο συγκεκριμένος χώρος εργασίας.

Η Δημιουργία ενός catkin Χώρου Εργασίας

Η δημιουργία ενός catkin χώρου εργασίας (αν τον ονομάσουμε catkin_ws όπως γίνεται στο σχετικό tutorial [5]) γίνεται ως εξής:

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/src
$ catkin_init_workspace
```

Ακόμα και αν ο χώρος εργασίας είναι κενός (αφού στην αρχή δεν υπάρχουν πακέτα στο φάκελο «src», παρά μόνο ένας σύνδεσμος CMakeLists.txt), μπορεί να εκτελεστεί η διαδικασία του χτισίματος, εφόσον θέσουμε τον catkin_ws ως τον τρέχον κατάλογο:

```
$ cd ~/catkin_ws/
$ catkin_make
```

Η εντολή catkin_make είναι ένα εργαλείο διευκόλυνσης του χειρισμού των catkin χώρων εργασίας. Κοιτώντας στον τρέχοντα κατάλογο αφού εκτελεστεί η παραπάνω εντολή, έχουν δημιουργηθεί δύο φάκελοι «build» και «devel». Μέσα στο φάκελο «devel» υπάρχουν αρκετά setup.*sh αρχεία, όπως αναμέναμε. Η εκτέλεση της εντολής source για οποιοδήποτε από αυτά θα «επικαλύψει» το περιβάλλον με το συγκεκριμένο χώρο εργασίας.

```
$ source devel/setup.bash
```

Μπορούμε να επιβεβαιώσουμε την επικάλυψη, ελέγχοντας αν η μεταβλητή περιβάλλοντος ROS_PACKAGE_PATH περιλαμβάνει τη διαδρομή στην οποία βρισκόμαστε.

```
$ echo $ROS_PACKAGE_PATH
/home/<user>/catkin_ws/src:/opt/ros/indigo/share:/opt/ros/indigo/stacks
```

Δημιουργία ενός catkin πακέτου

Ένα catkin πακέτο, πρέπει να περιέχει οπωσδήποτε δύο αρχεία, καθένα από τα οποία συνδέεται με έναν από τους δύο ρόλους των πακέτων. Πιο συγκεκριμένα, πρέπει να περιέχεται ένα αρχείο package.xml, στο οποίο ορίζονται ιδιότητες του πακέτου που χρειάζονται για τη δημοσίευσή του στην κοινότητα, καθώς και ένα αρχείο CMakeLists.txt, το οποίο είναι το αρχείο από το οποίο το σύστημα χτισίματος διαβάζει πληροφορίες που χρειάζονται σε αυτό για το χτίσιμο του κώδικα.

Οι ιδιότητες του πακέτου που ορίζονται στο package.xml έχουν να κάνουν με το όνομά του, τον αριθμό της συγκεκριμένης έκδοσης, τα ονόματα των συγγραφέων και των ανθρώπων που το συντηρούν και τις εξαρτήσεις του από άλλα catkin πακέτα. Αν οι εξαρτήσεις λείπουν, μπορεί να είναι εφικτό το χτίσιμο του πηγαίου κώδικα του πακέτου και το τρέξιμο δοκιμών στον υπολογιστή στο οποίο αυτό υπάρχει, όμως το πακέτο δεν θα δουλέψει σωστά όταν δημοσιευθεί στην κοινότητα του ROS. Όπως δηλώνει και η κατάληξη αυτού του αρχείου, για την περιγραφή των παραπάνω ιδιοτήτων χρησιμοποιείται γλώσσα xml και η πρώτη ετικέτα (tag)

συνήθως είναι αυτή στην οποία ορίζεται η έκδοση της xml που χρησιμοποιείται. Για παράδειγμα <?xml version=”1.0”?>. Η αρχή της περιγραφής του πακέτου δίνεται με το tag <package> (και αντίστοιχα το τέλος της με </package>). Άλλα tags που χρησιμοποιούνται είναι το <name>, το <version>, το <maintainer>, το <licence>, των οποίων η σημασία αντιστοιχεί στις ιδιότητες που αναφέρθηκαν, με τη σειρά με την οποία αναφέρθηκαν. Υπάρχει επίσης και το tag <description> στο οποίο δίνεται μια περιγραφή του πακέτου.

Για τη δήλωση των εξαρτήσεων του πακέτου χρησιμοποιούνται περισσότερα tags. Τα πιο σημαντικά είναι το <build_depend>, το <buildtool_depend> και το <run_depend>. Στο πρώτο δηλώνονται τα εργαλεία χτισίματος τα οποία χρειάζεται το πακέτο για να χτιστεί. Στις περισσότερες περιπτώσεις το μόνο εργαλείο που χρειάζεται είναι το catkin. Το δεύτερο ορίζει τα πακέτα που απαιτούνται κατά το χτίσιμο αυτού του πακέτου. Το τελευταίο δείχνει ποιες εξαρτήσεις αυτού του πακέτου (ας το αναφέρουμε ως πακέτο Α) χρειάζεται να υπάρχουν στο περιβάλλον εργασίας ενός τρίτου πακέτου (ας το πούμε πακέτο Γ) ώστε το πακέτο Γ να μπορεί να χρησιμοποιήσει το πακέτο Α. Αυτές διακρίνονται σε εκείνες που χρειάζονται για το τρέξιμο κώδικα που περιέχεται στο πακέτο Α και σε εκείνες που χρειάζονται για το χτίσιμο βιβλιοθηκών που βασίζονται στο πακέτο Α. Σε μια εντελώς πρόσφατη μορφή σύνταξης ενός αρχείου package.xml (η οποία, αν χρησιμοποιηθεί, πρέπει να δηλωθεί ως ιδιότητα στο tag package: <package format=”2”>), οι περιπτώσεις αυτές διαχωρίζονται, μπαίνοντας στα tags <build_export_depend> και <exec_depend> αντίστοιχα, τα οποία αντικαθιστούν το <run_depend>. Στις περισσότερες περιπτώσεις μια καταχώρηση στο <run_depend> θα είναι καταχώρηση και στο <build_depend>>. Να σημειωθεί ότι η παραδοσιακή μορφή συγγραφής ενός τέτοιου αρχείου μπορεί να χρησιμοποιηθεί ακόμα και ότι όλα τα tags τα οποία έχουν αναφερθεί παραμένουν τα ίδια στην καινούρια μορφή συγγραφής.

Σε ότι αφορά το αρχείο CMakeLists.txt, η δομή του είναι συγκεκριμένη και πρέπει να ακολουθείται αυστηρά (δηλαδή η σειρά των δηλώσεων είναι σημαντική). Η δήλωση της απαιτούμενης έκδοσης του CMake γίνεται με τη συνάρτηση cmake_minimum_required(). Ακολουθεί ο ορισμός του ονόματος του πακέτου, με τη συνάρτηση project(). Στη συνέχεια ορίζονται τα πακέτα που χρειάζονται για το χτίσιμο, με τη find_package(). Ακολουθούν οι δημιουργοί των μηνυμάτων, των υπηρεσιών και των actions, με τις add_message_files(), add_service_files() και add_action_files() αντίστοιχα. Η εκκίνηση της δημιουργίας ενός μηνύματος, μιας υπηρεσίας ή ενός action, δηλώνεται με την generate_messages(). Αν όμως χρησιμοποιούνται τέτοια αρχεία που είναι ήδη ορισμένα στο σύστημα (από κάποιο άλλο εγκατεστημένο πακέτο), αυτές οι συναρτήσεις δεν χρειάζονται. Στη συνέχεια χρησιμοποιείται η catkin_package(), η οποία είναι μια μακροεντολή του catkin που απαιτείται από το σύστημα χτισίματος για τη δημιουργία αρχείων διαμόρφωσης. Σε αυτήν ορίζονται αντικείμενα που πρόκειται να περαστούν σε εφαρμογές που

εξαρτώνται από αυτό το πακέτο. Ακολούθως, ορίζονται C++ βιβλιοθήκες και C++ εκτελέσιμα που πρέπει να χτιστούν (δηλαδή οι στόχοι), με τις add_library(), add_executable() και target_link_libraries(). Οι κανόνες εγκατάστασης ορίζονται με την install().

Για κώδικα γραμμένο σε Python, ο κανόνας εγκατάστασης είναι διαφορετικός, καθώς δεν χρησιμοποιούνται οι μέθοδοι add_library() και add_executable(), ώστε η CMake να ορίσει ποια αρχεία είναι στόχοι και τι είδους στόχοι είναι. Αντί για το συνηθισμένο τρόπο, στο CMakeLists.txt χρησιμοποιείται το παρακάτω:

catkin_install_python(PROGRAMS scripts/myscript

DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION})

Η δημιουργία ενός πακέτου προτείνεται να γίνεται σε ένα χώρο εργασίας, αν και τα πακέτα μπορούν να χτιστούν και από μόνα τους. Υπάρχει ένα εργαλείο, το catkin_create_pkg, το οποίο διευκολύνει τη δημιουργία του πακέτου. Εφόσον χρησιμοποιηθεί κάποιος χώρος εργασίας, μετακινούμαστε στο source space του (για παράδειγμα αν έχει το συνηθισμένο όνομα catkin_ws και έχουμε ονομάσει το source space «src» με την εντολή cd ~/catkin_ws/src). Στη συνέχεια, χρησιμοποιούμε αυτό το εργαλείο. Στο εγχειρίδιο [http://wiki.ros.org/ROS/Tutorials/CreatingPackage] δείχνεται η διαδικασία δημιουργίας ενός πακέτου με όνομα beginner_tutorials:

catkin_create_pkg beginner_tutorials std_msgs rospy roscpp

Το πρώτο όρισμα που πρέπει να δώσουμε, όταν χρησιμοποιούμε το catkin_create_pkg είναι το όνομα του πακέτου που θέλουμε να φτιάξουμε. Τα υπόλοιπα, ορίζουν τις εξαρτήσεις του. Με αυτήν την εντολή, δημιουργούνται αυτόματα τα αρχεία package.xml και CMakeLists.txt, τα οποία συμπληρώνονται μερικώς με τις εξαρτήσεις αυτού του πακέτου. Αυτό που πρέπει να κάνει ο χρήστης, είναι να συμπληρώσει τα tags του package.xml με τα στοιχεία που χρειάζονται, όταν τελικά αποφασίσει να διανείμει το πακέτο του στο δίκτυο. Σε ότι αφορά το CMakeLists.txt, αυτό γεμίζει με όλες τις πιθανές συναρτήσεις που μπορεί να χρησιμοποιηθούν, όμως οι περισσότερες είναι σχολιασμένες, ούτως ώστε να μην έχουν κάποια ισχύ αρχικά. Αυτό διευκολύνει κυρίως στο να τηρηθεί η σειρά με την οποία πρέπει να τοποθετούνται. Επίσης, υπάρχει περιγραφή της κάθε συνάρτησης και του τρόπου με τον οποίο αυτή πρέπει να χρησιμοποιηθεί, αν χρειαστεί.

Μετά από τη δημιουργία κάποιου πακέτου ή μετά από κάποια αλλαγή στα αρχεία του, πρέπει να εκτελείται η catkin_make και στη συνέχεια να γίνεται source το αρχείο setup.*, με τον τρόπο που δείχτηκε πριν.

2.4 Η χρήση της βιβλιοθήκης rospy

Μια βιβλιοθήκη – πελάτης του ROS (ROS client library) είναι μια συλλογή κώδικα που διευκολύνει τη δουλειά του προγραμματιστή. Η χρησιμότητά της είναι να κάνει τις έννοιες του ROS (τους κόμβους, τα μηνύματα, το μοντέλο publisher/subscriber, τις υπηρεσίες, τον εξυπηρετητή παραμέτρων) προσβάσιμες, ώστε να είναι εφικτή η ενσωμάτωσή τους στον κώδικα των εφαρμογών. Υπάρχουν τρεις βασικές τέτοιες βιβλιοθήκες και αρκετές ακόμα που βρίσκονται σε πειραματικό στάδιο. Οι βασικές βιβλιοθήκες είναι η roscpp, που είναι γραμμένη σε C++, η rospy, που είναι για την python, και η roslisp για τη LISP.

Παρακάτω, θα αναλύσουμε τέσσερα βασικά παραδείγματα χρήσης της rospy για τη δημιουργία κόμβων. Ο πρώτος κόμβος θα λειτουργεί ως publisher και θα στέλνει μηνύματα hello word, συνοδευόμενα κάθε φορά από την ώρα που αυτά εκδίδονται (για παράδειγμα hello world 10, hello world 11), ο δεύτερος ως subscriber και θα λαμβάνει αυτά τα μηνύματα, ο τρίτος θα παρέχει μια υπηρεσία (service node) και ο τέταρτος θα την καλεί (client).

Κόμβος Publisher:

```
 1 #!/usr/bin/env python
 2 import rospy
 3 from std_msgs.msg import String
 4
 5 def talker():
 6     pub = rospy.Publisher('chatter', String, queue_size=10)
 7     rospy.init_node('talker', anonymous=True)
 8     rate = rospy.Rate(10) # 10hz
 9     while not rospy.is_shutdown():
10         hello_str = "hello world %s" % rospy.get_time()
11         rospy.loginfo(hello_str)
12         pub.publish(hello_str)
13         rate.sleep()
14
15 if __name__ == '__main__':
16     try:
17         talker()
18     except rospy.ROSInterruptException:
19         pass
```

Ο παραπάνω κώδικας είναι από το tutorial [7]. Η πρώτη γραμμή χρησιμοποιείται αυτούσια για κάθε κόμβο ο οποίος είναι γραμμένος σε python, διότι με αυτήν δηλώνουμε ότι το συγκεκριμένο σενάριο (script) θα εκτελεστεί ως script της Python.

Στη συνέχεια, εισάγουμε τη rospy, με την εντολή import. Επίσης, σε αυτό το σημείο εισάγουμε και τους τύπους μηνυμάτων που θα χρησιμοποιήσουμε στην επικοινωνία. Στο συγκεκριμένο παράδειγμα, εισάγεται ο τύπος std_msgs/String.

Οι γραμμές 6 και 7 αφορούν το κομμάτι της διασύνδεσης με τους υπόλοιπους κόμβους του ROS. Στη γραμμή 6 δημιουργούμε μια υπόσταση (instance) της κλάσης Publisher (που περιλαμβάνεται στη rospy) και την ονομάζουμε pub. Η κλάση Publisher διαθέτει τη μέθοδο publish(), την οποία θα χρησιμοποιήσουμε στη συνέχεια για να δημοσιεύσουμε δεδομένα. Μια υπόσταση αυτής της κλάσης λαμβάνει κατά τη δημιουργία της ως πρώτο όρισμα το όνομα του topic στο οποίο θα δημοσιεύει ο κόμβος, ως δεύτερο τον τύπο μηνύματος που θα χρησιμοποιηθεί και ως τρίτο το μέγεθος της ουράς, ώστε αυτή να μη μεγαλώνει απεριόριστα σε περίπτωση που κανένας subscriber δεν δέχεται τα δεδομένα για κάποιο διάστημα. Στο παράδειγμα, ο κόμβος θα εκδίδει στο topic με όνομα «chatter», χρησιμοποιώντας τον τύπο μηνύματος String (ο οποίος μεταφράζεται από την Python σε μια κλάση std_msgs.msg.String). Ως μέγιστο μέγεθος της ουράς ορίζεται το 10.

Στη γραμμή 7 με την κλήση της μεθόδου init_node() της rospy, δημιουργούμε τον κόμβο για αυτήν τη διεργασία. Για κάθε διεργασία είναι δυνατή η ύπαρξη μόνο ενός κόμβου, οπότε μπορούμε να καλέσουμε αυτήν τη μέθοδο μόνο μια φορά μέσα σε ένα script. Η init_node() λαμβάνει ως όρισμα το όνομα με το οποίο θα είναι γνωστός ο κόμβος στο δίκτυο και είναι απαραίτητο να υπάρχει γιατί αλλιώς δεν μπορεί να γίνει η επικοινωνία του κόμβου με το master. Το όνομα κάθε κόμβου στο σύστημα πρέπει να είναι μοναδικό. Φυσιολογικά, αν κάποιος κόμβος με το ίδιο όνομα εισαχθεί στο σύστημα, ο παλαιότερος κόμβος θα τερματιστεί. Ωστόσο, υπάρχουν περιπτώσεις στις οποίες θα θέλαμε να υπάρχουν περισσότεροι κόμβοι που να εκτελούν την ίδια λειτουργία. Για τον λόγο αυτό, στις πρόσφατες εκδόσεις του ROS η init_node() μπορεί να λαμβάνει και ένα δεύτερο όρισμα, τη σημαία anonymous. Όταν δηλώνουμε ότι anonymous=True, σημαίνει ότι ο κόμβος που δημιουργήσαμε θα μετονομαστεί από το master, ώστε να μπορέσουμε να τρέχουμε ταυτόχρονα πολλούς ίδιους κόμβους.

Η δομή του βρόχου που υπάρχει στις γραμμές 9-13 είναι αρκετά συνηθισμένη για την αποστολή δεδομένων από έναν publisher, επειδή τις περισσότερες φορές δεν θέλουμε να στείλουμε απλώς μια φορά κάτι, αλλά κατ’ επανάληψη (αλλάζοντας συνήθως το περιεχόμενο της αποστολής). Αρχικά, στη γραμμή 10 δηλώνουμε ότι για την επανάληψη του βρόχου θα ελέγχουμε αν έχει ζητηθεί από το πρόγραμμα να τερματιστεί (με Ctrl-C ή οποιοδήποτε άλλο τρόπο). Όσο δεν συμβαίνει αυτό, ο βρόχος μπορεί να επαναλαμβάνεται. Στη γραμμή 13 καλούμε τη μέθοδο publish με την υπόσταση του Publisher που ήδη έχουμε δημιουργήσει, την οποία, όπως αναφέραμε, χρησιμοποιούμε για να στέλνουμε δεδομένα.

Η publish() μπορεί να χρησιμοποιηθεί με τρεις τρόπους. Ο πρώτος είναι να δημιουργηθεί πρώτα μια υπόσταση της κλάσης που δημιουργήθηκε από τον τύπο του μηνύματος και, στη συνέχεια, να περαστεί ως όρισμα στη μέθοδο. Για παράδειγμα:

```
msg = String()
msg.data = hello
```

Σημειώνουμε εδώ ότι το μήνυμα String (άρα και κάθε στιγμιότυπο της κλάσης String) αποτελείται από μόνο ένα πεδίο, με όνομα data, τύπου string. Στη συνέχεια μπορούμε να καλέσουμε την publish(msg).

Ο δεύτερος τρόπος είναι να αντιστοιχήσουμε κάθε πεδίο του μηνύματος σε μια παράμετρο της publish(). Δηλαδή στην πρώτη θέση μπαίνει η τιμή για το πρώτο πεδίο, στη δεύτερη για το δεύτερο κλπ. Αυτός είναι και ο τρόπος που χρησιμοποιείται στο παράδειγμα, αν και δεν φαίνεται πολύ ξεκάθαρα λόγω του ότι το συγκεκριμένο μήνυμα αποτελείται από μόνο ένα πεδίο.

Ο τρίτος τρόπος είναι να δημιουργήσουμε την υπόσταση κατά την κλήση της publish. Δηλαδή, αν θέλουμε να συμπληρώσουμε το πρώτο πεδίο με τη συμβολοσειρά foo και το δεύτερο με «bar»:

```
pub.publish(message_field_1='foo', message_field_2='bar')
```

Φυσικά, μπορούμε να συμπληρώσουμε κάποια από τα πεδία και για τα υπόλοιπα να αφήσουμε τις πρότυπες τιμές.

Εκτός από την publish(), μέσα στο βρόχο καλούμε και τη μέθοδο loginfo() της rospy, που παίρνει ως όρισμα μια συμβολοσειρά και την τυπώνει στην οθόνη. Επίσης, την εγγράφει στο εργαλείο rosout, που είναι ένα βοήθημα που διαθέτει το ROS για την αποσφαλμάτωση.

Εφόσον στο παράδειγμα δεν στέλνουμε απλώς μια φορά ένα μήνυμα αλλά έχουμε μια ροή δεδομένων, με τη χρήση του βρόχου, είναι λογικό να θέλουμε να δηλώσουμε και τη συχνότητα με την οποία θα στέλνονται αυτά. Χρησιμοποιούμε λοιπόν συνήθως ένα τέχνασμα, δημιουργώντας πριν το βρόχο (γραμμή 8 στο παράδειγμα) ένα αντικείμενο της κλάσης Rate για να χρησιμοποιήσουμε μέσα στο βρόχο (γραμμή 14) τη μέθοδο sleep(), η οποία δηλώνει ότι δε θα γίνει τίποτα για ένα συγκεκριμένο διάστημα. Το διάστημα αυτό το ορίζουμε κατά τη δημιουργία του αντικειμένου της Rate, όμως όχι με τη μορφή του χρόνου που πρέπει να περάσει για να εκτελεστεί η επόμενη εντολή, αλλά με τη συχνότητα που θέλουμε να επιτύχουμε, αφήνοντας στη μέθοδο τη δουλειά της μετάφρασης από την επιθυμητή συχνότητα στο απαιτούμενο διάστημα παύσης. Στο παράδειγμα δηλώνουμε ότι θέλουμε τα μηνύματα να στέλνονται με συχνότητα 10 Hz.

Στο τέλος του script, εκτελούμε το συνήθη έλεγχο της ειδικής μεταβλητής __name__ της Python, που χρησιμεύει για τη διάκριση των προγραμμάτων ως αυτόνομα εκτελούμενα ή όχι (γραμμές 15-19), με την πρόσθετη λειτουργία του χειρισμού της εξαίρεσης ROSInterruptException (γραμμές 18-19), που συμβαίνει όταν δοθεί κάποια εντολή κλεισίματος του κόμβου ενώ εκτελείται η sleep(). Σε αυτήν την περίπτωση, δεν πραγματοποιείται η εκτέλεση της συνάρτησης talker(), για να διασφαλίσουμε ότι δεν θα εκτελεστεί κάποιο κομμάτι κώδικα μετά τη sleep().

Σε αυτό το σημείο, πρέπει να αναφέρουμε ότι κάθε φορά που γράφουμε ένα αρχείο για τη δημιουργία ενός κόμβου στο ROS είναι απαραίτητο να το μετατρέψουμε σε εκτελέσιμο κώδικα, με την εντολή

chmod +x <όνομα αρχείου>

Για παράδειγμα, αν έχουμε ονομάσει το αρχείο του παραπάνω κόμβου «talker.py», θα γράψουμε στη γραμμή εντολών «chmod +x talker.py». Ακολουθεί η περιγραφή της δημιουργίας ενός subscriber στο topic chatter.

Κόμβος Subscriber:

```
 1 #!/usr/bin/env python
 2 import rospy
 3 from std_msgs.msg import String
 4
 5 def callback(data):
 6     rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)
 7
 8 def listener():
 9
10     # In ROS, nodes are uniquely named. If two nodes with the same
11     # node are launched, the previous one is kicked off. The
12     # anonymous=True flag means that rospy will choose a unique
13     # name for our 'listener' node so that multiple listeners can
14     # run simultaneously.
15     rospy.init_node('listener', anonymous=True)
16
17     rospy.Subscriber("chatter", String, callback)
18
19     # spin() simply keeps python from exiting until this node is stopped
20     rospy.spin()
21
22 if __name__ == '__main__':
23     listener()
```

Όπως βλέπουμε, αρχικά εισάγουμε και πάλι τη rospy και τους τύπους των μηνυμάτων που θα χρησιμοποιήσουμε, κάτι που γίνεται πάντα. Στη συνέχεια,

παρατηρούμε ότι εδώ έχουμε δύο συναρτήσεις αντί για μια που είχαμε στον publisher και επίσης είναι κανόνας για τους subscriber.

Η μια συνάρτηση ονομάζεται στο παράδειγμά μας listener() και είναι αυτή που καλείται πρώτη κατά την εκτέλεση του script. Η λειτουργία της είναι να εκκινήσει τον κόμβο, με την init_node() που είδαμε πριν. Επίσης, εδώ δημιουργείται το αντικείμενο Subscriber (γραμμή 17), με παραμέτρους το topic στο οποίο θα ακούει, τον τύπο μηνύματος που χρησιμοποιείται και τη συνάρτηση που εκτελείται κάθε φορά που φτάνει κάποιο μήνυμα. Στο παράδειγμα, την ονομάζουμε callback και η λειτουργία της φαίνεται στις γραμμές 5-6. Ως πρώτο όρισμα παίρνει το μήνυμα.

Η χρήση του ονόματος callback στο παράδειγμα δεν είναι τυχαία, αλλά αντίθετα υποδηλώνει μια έννοια. Γενικότερα, χρησιμοποιείται ως όρος στον προγραμματισμό για να περιγράψει μια συνάρτηση η οποία περνιέται ως όρισμα σε μια άλλη και εκτελείται μετά από κάποιο γεγονός. Ο έλεγχος μπορεί να περάσει στην callback είτε πριν επιστρέψει η βασική συνάρτηση, οπότε ονομάζεται σύγχρονη (synchronous ή blocking), είτε μετά (ασύγχρονη). Στην περίπτωση της ασύγχρονης υπάρχει η δυνατότητα να εκκινήσει η callback ένα νήμα στο οποίο αναθέτει τη λειτουργία της και επιστρέφει αμέσως. Αυτό συμβαίνει και στην υλοποίηση του subscriber στην Python. Στην υλοποίηση με την roscpp δεν συμβαίνει το ίδιο.

Για το τέλος αφήσαμε τη rospy.spin() (γραμμή 20) η οποία καλείται μέσα στη listener() και απλώς αποτρέπει την έξοδο της python μέχρι να διακοπεί ο συγκεκριμένος κόμβος.

Κόμβος υπηρεσίας

```
 1 #!/usr/bin/env python
 2
 3 from beginner_tutorials.srv import *
 4 import rospy
 5
 6 def handle_add_two_ints(req):
 7     print "Returning [%s + %s = %s]"%(req.a, req.b, (req.a + req.b))
 8     return AddTwoIntsResponse(req.a + req.b)
 9
10 def add_two_ints_server():
11     rospy.init_node('add_two_ints_server')
12     s = rospy.Service('add_two_ints', AddTwoInts, handle_add_two_ints)
13     print "Ready to add two ints."
14     rospy.spin()
15
16 if __name__ == "__main__":
```

```
 17    add_two_ints_server()
```

Στο παραπάνω παράδειγμα δημιουργούμε ένα κόμβο που παρέχει την υπηρεσία της πρόσθεσης δύο αριθμών, δεχόμενος αυτούς ως υπόσταση της κλάσης που προκύπτει από το μήνυμα αίτησης υπηρεσίας «AddTwoIntsRequest».

Πιο συγκεκριμένα, μετά τις εισαγωγές της rospy και του τύπου της υπηρεσίας που θα χρησιμοποιήσουμε, έχουμε δύο συναρτήσεις. Η add_two_ints_server() είναι αυτή στην οποία εκκινείται ο κόμβος με την init_node() και ορίζεται η υπηρεσία με τη δημιουργία μιας υπόστασης της κλάσης Service, η οποία παίρνει ως πρώτο όρισμα το όνομα της υπηρεσίας, ως δεύτερο τον τύπο της υπηρεσίας (στο παράδειγμα είναι ο AddTwoInts) και ως τρίτο τη συνάρτηση η οποία θα καλείται όταν φτάσει κάποια αίτηση.

Στο παράδειγμα, αυτή ονομάζεται handle_add_two_ints() (γραμμές 6-8), λαμβάνει ως όρισμα την υπόσταση του AddTwoIntsRequest και επιστρέφει υποστάσεις της AddTwoIntsResponse. Αυτά τα μηνύματα, δημιουργούνται από την υπηρεσία AddTwoInts, η οποία είναι η εξής:

Κόμβος - πελάτης της υπηρεσίας

```
 1 #!/usr/bin/env python
 2
 3 import sys
 4 import rospy
 5 from beginner_tutorials.srv import *
 6
 7 def add_two_ints_client(x, y):
 8    rospy.wait_for_service('add_two_ints')
 9    try:
10       add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
11       resp1 = add_two_ints(x, y)
12       return resp1.sum
13    except rospy.ServiceException, e:
14       print "Service call failed: %s"%e
15
16 def usage():
17    return "%s [x y]"%sys.argv[0]
18
19 if __name__ == "__main__":
20    if len(sys.argv) == 3:
21       x = int(sys.argv[1])
22       y = int(sys.argv[2])
23    else:
24       print usage()
25       sys.exit(1)
26    print "Requesting %s+%s"%(x, y)
27    print "%s + %s = %s"%(x, y, add_two_ints_client(x, y))
```

Στον παραπάνω κώδικα δημιουργούμε ένα πρόγραμμα το οποίο δέχεται από τη γραμμή εντολών δύο αριθμούς (γραμμές 19-22) και τους περνάει ως ορίσματα στη συνάρτηση add_two_ints_client(). Αυτό είναι το κομμάτι που μας ενδιαφέρει περισσότερο (γραμμές 7-4).

Στην add_two_ints_client(), αρχικά καλούμε τη rospy.wait_for_service() η οποία μπλοκάρει μέχρι να είναι διαθέσιμη η υπηρεσία της οποίας το όνομα δίνουμε ως όρισμα, στην περίπτωσή μας η «add_two_ints». Φυσικά, δεν έχουμε ξανασυναντήσει κάτι παρόμοιο στο μοντέλο publisher – subscriber, γιατί εκεί οι κόμβοι δεν είναι στενά συνδεδεμένοι μεταξύ τους, σε αντίθεση με το μοντέλο των υπηρεσιών.

Στη συνέχεια δημιουργούμε ένα διαμεσολαβητή (proxy) για την υπηρεσία που θέλουμε να καλέσουμε (γραμμή 10). Η κλάση αυτή ονομάζεται ServiceProxy και παίρνει ως πρώτο όρισμα το όνομα της υπηρεσίας και ως δεύτερο τον τύπο της υπηρεσίας. Το δεύτερο όρισμα δεν είναι υποχρεωτικό. Ουσιαστικά χρειάζεται για να δημιουργηθεί αυτόματα το αντικείμενο AddTwoIntsRequest. Καλούμε το διαμεσολαβητή που δημιουργήσαμε ως μια απλή συνάρτηση (γραμμή 11), με ορίσματα τα συμπληρωμένα πεδία του μηνύματος της αίτησης. Αν δεν δηλώναμε τον τύπο της υπηρεσίας κατά τη δημιουργία του ServiceProxy, η γραμμή 11 θα ήταν για παράδειγμα:

add_two_ints(AddTwoIntsRequest(1, 2))

δηλαδή ο τύπος του μηνύματος της αίτησης θα δηλωνόταν κατά την κλήση του. Η επιστρεφόμενη τιμή θα είναι μια υπόσταση της κλάσης AddTwoIntsResponse.

Αν αποτύχει η κλήση της υπηρεσίας, μπορεί να συμβεί μια εξαίρεση rospy.ServiceException, την οποία πρέπει να χειριστούμε με τη δομή try/except. Τέλος, σημειώνουμε ότι για έναν πελάτη μιας υπηρεσίας δεν χρειάζεται να χρησιμοποιήσουμε την init_node().

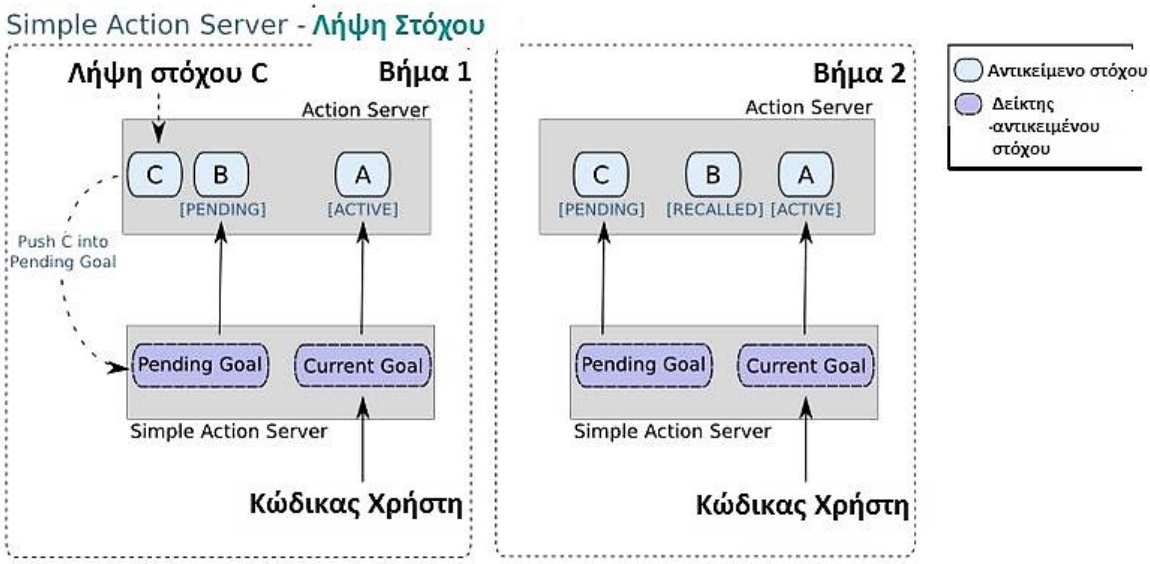
2.5 Η Βιβλιοθήκη actionlib

Το πακέτο actionlib παρέχει τη διεπαφή για την υλοποίηση της επικοινωνίας με actions. Πριν δούμε παραδείγματα δημιουργίας εφαρμογών που επικοινωνούν με αυτό το πρωτόκολλο, θα πρέπει να διευκρινιστούν κάποια σημαντικά στοιχεία. Οι βασικές λειτουργίες που μας ενδιαφέρουν κατά την υλοποίηση αυτού του μοντέλου, που είναι για παράδειγμα η αποστολή στόχων από τη μεριά της εφαρμογής – πελάτη και η διαχείρισή τους, καθώς και η παρακολούθησή των καταστάσεων στις οποίες βρίσκονται, για τις οποίες αναφερθήκαμε στην παράγραφο 2.3, πραγματοποιούνται με τη δημιουργία υποστάσεων των κλάσεων ActionClient και ActionServer αντίστοιχα, που διαθέτουν τις μεθόδους για την πραγματοποίηση αυτών των λειτουργιών. Επίσης, είναι σημαντικό ότι η εργασία την οποία καλείται να διεκπεραιώσει η εφαρμογή του εξυπηρετητή δεν πραγματοποιείται από το αντικείμενο ActionServer, αλλά αντίθετα η επεξεργασία

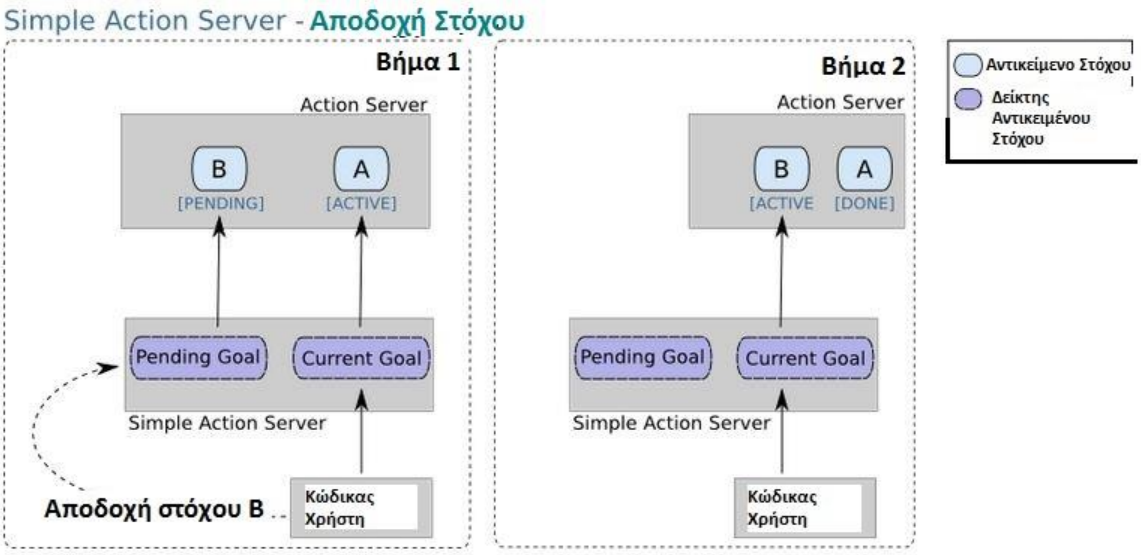
των στόχων γίνεται από τον χρήστη αυτού του αντικειμένου (δηλαδή από την εφαρμογή του εξυπηρετητή), με τη χρήση συνήθως μιας συνάρτησης callback, στην οποία περνάει ο έλεγχος μόλις φτάσει κάποιος στόχος (Ίσως μετά από αυτό γίνει καλύτερη η κατανόηση της Εικόνας 7).

Εκτός από αυτές τις βασικές λειτουργίες, συνήθως χρειαζόμαστε η διαχείριση των στόχων που φτάνουν στον εξυπηρετητή να είναι τέτοια ώστε μόνο ένας από αυτούς να είναι ενεργός κάθε στιγμή. Για το λόγο αυτό χρησιμοποιούμε τις κλάσεις SimpleActionServer και SimpleActionClient, αντί των ActionServer και ActionClient, καθώς έτσι παρέχεται μια πιο απλή πολιτική επεξεργασίας των στόχων.

Πιο αναλυτικά, η πολιτική που χρησιμοποιείται σε αυτήν την υλοποίηση ορίζει ότι κατά τη λήψη ενός στόχου, ο Simple Action Server το μετακινεί στη θυρίδα (slot) «σε αναμονή». Αν υπάρχει σε αυτήν κάποιος προηγούμενος στόχος, ακυρώνει τον παλιότερο. Η αποδοχή ενός στόχου (δηλαδή η μεταβολή της κατάστασής του σε «ενεργό»), συνεπάγεται την ακύρωση οποιουδήποτε παλιότερου ενεργού. Φυσικά, αυτό θα αποτυπωθεί και στην αλλαγή της κατάστασης του παλιότερου στόχου. Τα παραπάνω φαίνονται σχηματικά στις Εικόνες 9 και 10.



Εικόνα 9 - Η πολιτική που ακολουθείται από το Simple Action Server κατά τη λήψη ενός στόχου



Εικόνα 10 - Μόνο ένας στόχος μπορεί να είναι ενεργός κάθε φορά (Single Goal Policy)

Στη συνέχεια, θα δούμε πώς γίνεται η δημιουργία μιας εφαρμογής – εξυπηρετητή, η οποία υπολογίζει μια ακολουθία Fibonacci, δεχόμενη ως στόχο την τάξη της ακολουθίας. Το αρχείο προσδιορισμού του action έχει την παρακάτω μορφή:

```
int32 order
---
int32[] sequence
---
int32[] sequence
```

Ο παρακάτω κώδικας μπορεί να βρεθεί στο [8], ωστόσο με μια πλήρη εγκατάσταση του ROS, το πακέτο actionlib_tutorials βρίσκεται ήδη στο σύστημα.

```
 1 #! /usr/bin/env python
 2
 3 import roslib; roslib.load_manifest('actionlib_tutorials')
 4 import rospy
 5 import actionlib
 6 import actionlib_tutorials.msg
 7
 8 class FibonacciAction(object):
 9   # create messages that are used to publish feedback/result
10   _feedback = actionlib_tutorials.msg.FibonacciFeedback()
11   _result   = actionlib_tutorials.msg.FibonacciResult()
12
13   def __init__(self, name):
14     self._action_name = name
```

```
15            self._as = actionlib.SimpleActionServer(self._action_name,
actionlib_tutorials.msg.FibonacciAction, execute_cb=self.execute_cb, auto_start =
False)
16    self._as.start()
17
18  def execute_cb(self, goal):
19    # helper variable
20    success = True
21
22    # append the seeds for the fibonacci sequence
23    self._feedback.sequence = []
24    self._feedback.sequence.append(0)
25    self._feedback.sequence.append(1)
26
27    # publish info to the console for the user
28    rospy.loginfo('%s: Executing, creating fibonacci sequence of order %i
with seeds %i, %i' % (self._action_name, goal.order, self._feedback.sequence[0],
self._feedback.sequence[1]))
29
30    # start executing the action
31    for i in xrange(1, goal.order):
32      # check that preempt has not been requested by the client
33      if self._as.is_preempt_requested():
34        rospy.loginfo('%s: Preempted' % self._action_name)
35        self._as.set_preempted()
36        success = False
37        break
38        self._feedback.sequence.append(self._feedback.sequence[i] +
self._feedback.sequence[i-1])
39      # publish the feedback
40      self._as.publish_feedback(self._feedback)
41
42    if success:
43      self._result.sequence = self._feedback.sequence
44      rospy.loginfo('%s: Succeeded' % self._action_name)
45      self._as.set_succeeded(self._result)
46
47 if __name__ == '__main__':
48   rospy.init_node('fibonacci')
49   FibonacciAction(rospy.get_name())
50   rospy.spin()
```

Όπως βλέπουμε, αρχικά πρέπει πάντα να εισάγουμε την actionlib (φυσικά και τη rospy). Στη συνέχεια, εισάγουμε τα μηνύματα που θα χρησιμοποιηθούν στην επικοινωνία. Είναι αυτά που έχουν δημιουργηθεί από το αρχείο προσδιορισμού του action και στην περίπτωση του παραδείγματος βρίσκονται στον υποκατάλογο msg του actionlib_tutorials.

Στις γραμμές 15-16, δημιουργούμε ένα SimpleActionServer, στον οποίο δίνουμε ως πρώτο όρισμα ένα όνομα. Αυτό περιγράφει το χώρο ονομάτων (namespace) στον οποίο θα βρίσκονται τα topics που θα χρησιμοποιηθούν στην επικοινωνία. Το ίδιο όνομα πρέπει να δοθεί ως όρισμα και κατά τη δημιουργία του SimpleActionClient στην εφαρμογή – πελάτη. Ως δεύτερο όρισμα λαμβάνει τον τύπο του μηνύματος με κατάληξη «__Action». Σε αυτό ορίζονται τα μηνύματα που θα χρησιμοποιηθούν στην επικοινωνία. Ως τρίτο όρισμα εισάγουμε το όνομα της συνάρτησης callback. Αυτή θα εκκινεί ένα καινούριο νήμα κάθε φορά που φτάνει κάποιος καινούριος στόχος. Η τέταρτη παράμετρος είναι μια μεταβλητή, η auto_start, τύπου Boolean, η οποία πρέπει πάντα να παίρνει την τιμή False, κάτι που δηλώνει ότι ο ActionServer δεν θα αρχίσει να δημοσιεύει αμέσως μόλις δημιουργηθεί, για να αποφευχθούν συνθήκες ανταγωνισμού. Γι’ αυτό, καλούμε και τη μέθοδο start() στη συνέχεια.

Στις γραμμές 18-45 ορίζεται η callback, η οποία στο παράδειγμα ονομάζεται execute_cb(), με όρισμα το στόχο με τον οποίο καλείται κάθε φορά. Αρχικά, στις γραμμές 23-25 βάζουμε τους δύο πρώτους όρους της ακολουθίας στο μήνυμα FibonacciFeedback, καθώς αυτοί είναι πάντα οι ίδιοι (0 και 1). Στη συνέχεια, ξεκινάει να εκτελείται το action: σε ένα βρόχο, ο οποίος επαναλαμβάνεται τόσες φορές όσες λέει ο στόχος, προστίθεται κάθε φορά στη λίστα sequence του μηνύματος FibonacciFeedback ο επόμενος αριθμός της ακολουθίας, ο υπολογισμός του οποίου πραγματοποιείται στη γραμμή 38, και στη συνέχεια εκδίδεται η ανάδραση (feedback) με τη χρήση της publish_feedback(), που λαμβάνει ως όρισμα το συμπληρωμένο (μέχρι εκείνο το σημείο) FibonacciFeedback.

Αυτό συμβαίνει με την προϋπόθεση ότι δεν έχει ληφθεί από τον πελάτη κάποια αίτηση για ακύρωση, κάτι που ελέγχεται με την κλήση της μεθόδου is_preempt_requested(), η οποία επιστρέφει true αν έχει αιτηθεί ακύρωση και false στην αντίθετη περίπτωση (31-37). Αν επιστρέψει true, σταματάει η εκτέλεση του βρόχου.

Αν η εκτέλεση του action έχει ολοκληρωθεί με επιτυχία (ελέγχεται με τη μεταβλητή success), καλώ τη set_succeeded(), η οποία θέτει την κατάσταση του ενεργού στόχου σε «επιτυχημένος». Προαιρετικά, αυτή παίρνει ως πρώτο όρισμα ένα αποτέλεσμα για να στείλει πίσω και ως δεύτερο κάποιο μήνυμα κειμένου που θα σταλεί επίσης σε οποιοδήποτε πελάτη του στόχου.

Τέλος, έξω από την κλάση, εκκινείται ο κόμβος με την init_node() και αποτρέπεται η έξοδός του με τη spin(). Αυτές είναι και οι πρώτες μέθοδοι που θα κληθούν στη ροή του προγράμματος.

Ακολουθεί ο κώδικας της εφαρμογής – πελάτη για το action που περιγράφηκε προηγουμένως.

```
 1 #! /usr/bin/env python
 2
 3 import rospy
 4
 5 # Brings in the SimpleActionClient
 6 import actionlib
 7
 8 # Brings in the messages used by the fibonacci action, including the
 9 # goal message and the result message.
10 import actionlib_tutorials.msg
11
12 def fibonacci_client():
13     # Creates the SimpleActionClient, passing the type of the action
14     # (FibonacciAction) to the constructor.
15     client = actionlib.SimpleActionClient('fibonacci',
actionlib_tutorials.msg.FibonacciAction)
16
17     # Waits until the action server has started up and started
18     # listening for goals.
19     client.wait_for_server()
20
21     # Creates a goal to send to the action server.
22     goal = actionlib_tutorials.msg.FibonacciGoal(order=20)
23
24     # Sends the goal to the action server.
25     client.send_goal(goal)
26
27     # Waits for the server to finish performing the action.
28     client.wait_for_result()
29
30     # Prints out the result of executing the action
31     return client.get_result()  # A FibonacciResult
32
33 if __name__ == '__main__':
34     try:
35         rospy.init_node('fibonacci_client_py')
36         result = fibonacci_client()
37         print "Result:", ', '.join([str(n) for n in result.sequence])
38     except rospy.ROSInterruptException:
39         print "program interrupted before completion"
```

Μετά τις απαραίτητες εισαγωγές και αφού ξεκινήσουμε τον κόμβο με την init_node() (γραμμή 35), καλούμε τη συνάρτηση στην οποία θα δημιουργηθεί ο SimpleActionClient, θα στέλνει ένα μήνυμα στόχου στο SimpleActionServer και θα επιστρέφει με το αποτέλεσμα που έλαβε, το οποίο θα τυπώνεται στο τερματικό στο οποίο εκτελείται ο πελάτης.

Πιο συγκεκριμένα, η δημιουργία του SimpleActionClient πραγματοποιείται στη γραμμή 15 και στο αντικείμενο που δημιουργείται περνάμε ως ορίσματα, στην πρώτη θέση, το όνομα του action, το οποίο είναι, όπως αναφέρθηκε και στη δημιουργία του αντίστοιχου ActionServer, ο χώρος ονομάτων στον οποίο θα περιέχονται τα topics, ενώ στη δεύτερη τον τύπο μηνύματος που περιγράφει το action (FibonacciAction).

Ακολουθεί η κλήση των μεθόδων αυτού του αντικειμένου. Αρχικά, με τη wait_for_server() μπλοκάρεται η συνάρτηση, περιμένοντας την εκκίνηση του Action Server, αν αυτός δεν έχει ήδη εκκινηθεί. Στη συνέχεια δημιουργούμε το στόχο ως υπόσταση της κλάσης FibonacciGoal, περνώντας ως όρισμα τις τιμές για τα αντίστοιχα πεδία που αναφέρονται στο αντίστοιχο μήνυμα (στην προκειμένη περίπτωση έχει οριστεί ένα πεδίο, το order). Ο στόχος αποστέλλεται με τη send_goal(), η οποία λαμβάνει ως όρισμα το στόχο που δημιουργήσαμε προηγουμένως. Στη συνέχεια, γίνεται αναμονή της επεξεργασίας του στόχου από την πλευρά του εξυπηρετητή με τη wait_for_result(), μέχρι να εξαχθεί το αποτέλεσμα, το οποίο λαμβάνει ο SimpleActionClient με τη get_result().

Κεφάλαιο 3
Σύντομη Παρουσίαση του NAO

3.1 Γενικά Χαρακτηριστικά

Το NAO είναι ένα ανθρωποειδές ρομπότ το οποίο αναπτύχθηκε και εξακολουθεί να αναπτύσσεται από την εταιρία Aldebaran, με σκοπό την αλληλεπίδραση με τους ανθρώπους. Η προσπάθεια ξεκίνησε το 2004, το 2006 κατασκευάστηκε το πρώτο ρομπότ και το 2008 κυκλοφόρησε η πρώτη του έκδοση. Από τότε, ξεκίνησε να χρησιμοποιείται ευρέως από Πανεπιστήμια και ερευνητικά εργαστήρια, για ερευνητικούς και εκπαιδευτικούς σκοπούς, σε περισσότερες από 70 χώρες. Η έκδοση του ρομπότ που χρησιμοποιήθηκε για τις ανάγκες της εργασίας είναι η πέμπτη, με όνομα NAO Evolution και διατέθηκε πρώτη φορά το 2014.

Εικόνα 11 - Το NAO

Η έκδοση που χρησιμοποιήθηκε προσφέρει 25 βαθμούς ελευθερίας. Επίσης, διαθέτει δύο κάμερες υψηλής ανάλυσης (HD), τέσσερα μικρόφωνα, δύο πομπούς και δύο δέκτες υπερήχων, με τους οποίους υπολογίζει αποστάσεις, εννιά αισθητήρες αφής, οχτώ αισθητήρες πίεσης στα πόδια, καθώς και γυρόμετρο και αξελερόμετρο, με τα οποία μπορεί να έχει επίγνωση σε τι στάση βρίσκεται.

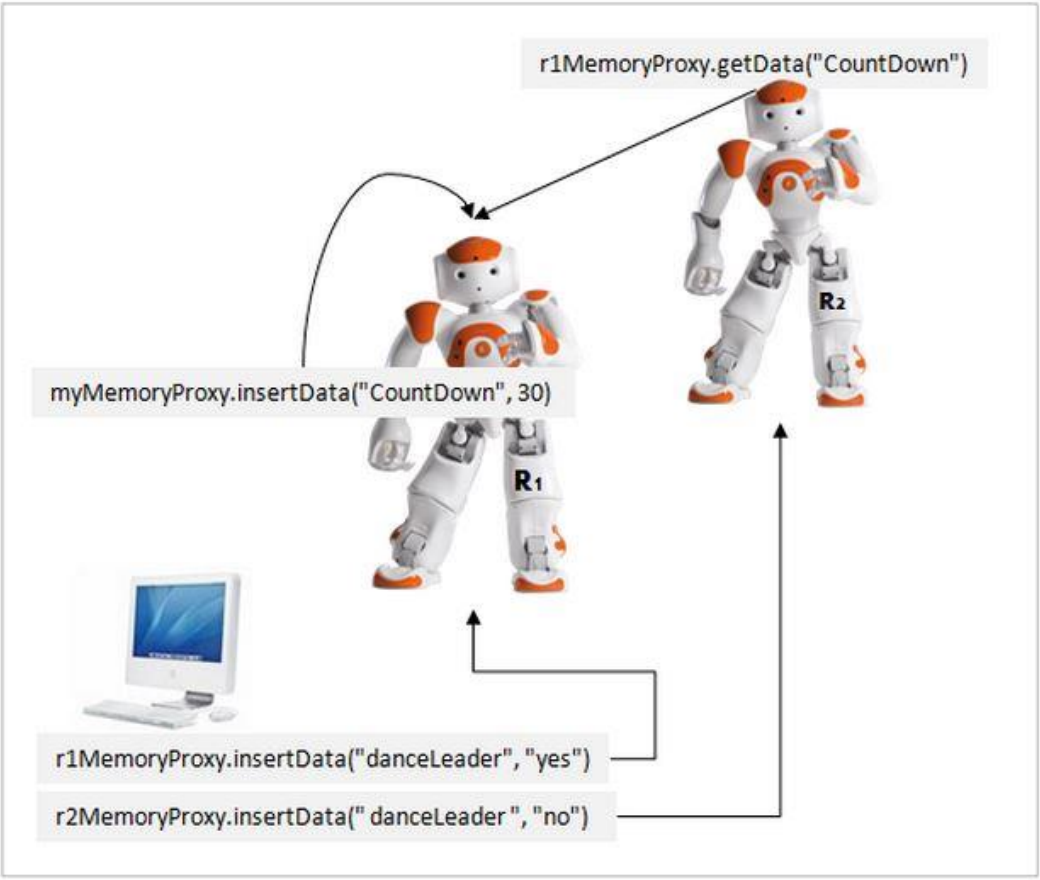
Αυτός ο συνδυασμός τεχνολογιών δίνει στο NAO την ικανότητα επίγνωσης του περιβάλλοντός του και επικοινωνίας με αυτό, μέσω της κατανόησης και της ερμηνείας των δεδομένων που λαμβάνονται από τους αισθητήρες του, από το λειτουργικό του σύστημα, το οποίο ονομάζεται NAOqi OS.

Οι προγραμματιστές, μπορούν να αναπτύξουν εφαρμογές, εκμεταλλευόμενοι όλες αυτές τις δυνατότητες ώστε να δημιουργήσουν σύνθετες συμπεριφορές, να αποκτήσουν πρόσβαση στα δεδομένα που προέρχονται από τους αισθητήρες και να ελέγξουν το ρομπότ, δίνοντάς του ζωή και εξελίσσοντας το. Υπάρχει επίσης ένα διαδικτυακό αποθετήριο εφαρμογών, από το οποίο το ρομπότ έχει τη δυνατότητα να κατεβάσει από μόνο του καινούριες συμπεριφορές [9].

3.2 Προγραμματισμός του NAO

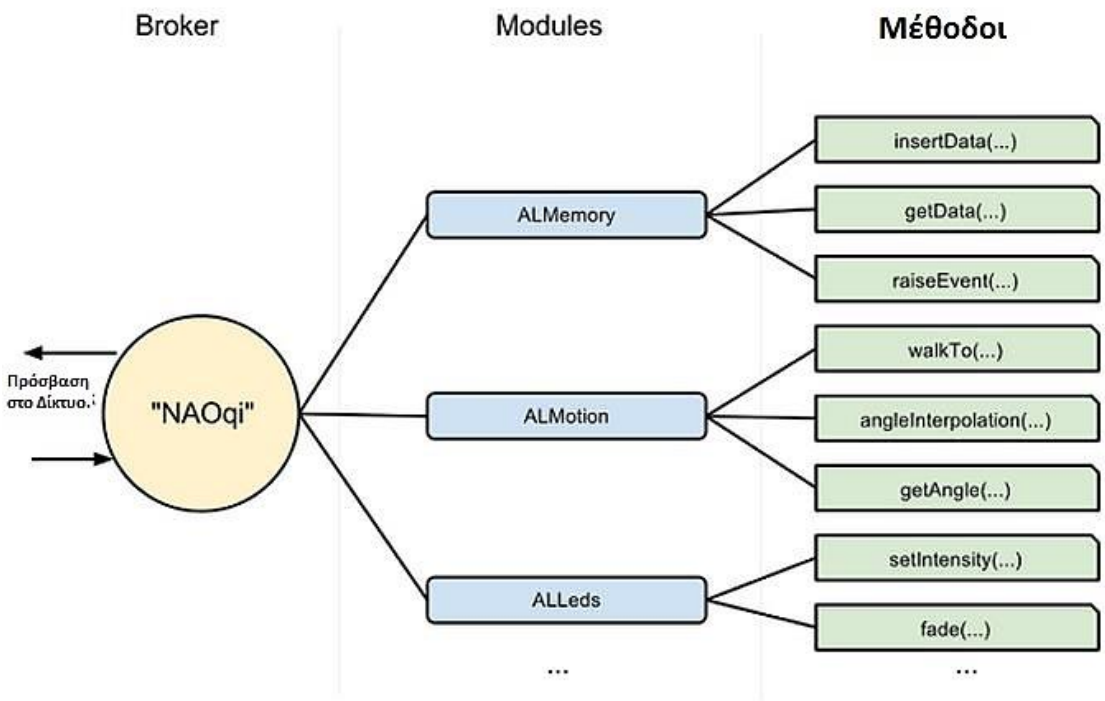
Το κυρίως λογισμικό που τρέχει στο ρομπότ και το ελέγχει είναι το NAOqi. Για τον προγραμματισμό του NAO, χρησιμοποιείται η πλατφόρμα NAOqi (NAOqi framework). Είναι δυνατή η ανάπτυξη κώδικα τόσο σε Python όσο και σε C++. Αυτός ο κώδικας μπορεί να τρέξει είτε απευθείας στο ρομπότ είτε σε έναν υπολογιστή.

Επίσης, μια εφαρμογή πραγματικού χρόνου μπορεί να αποτελείται από πολλές διεργασίες και μονάδες (modules) κατανεμημένες σε διαφορετικά ρομπότ και φυσικά σε υπολογιστές. Ένα εκτελέσιμο μπορεί να συνδεθεί σε ένα άλλο ρομπότ χρησιμοποιώντας την IP διεύθυνσή του και τη θύρα του και όλες οι μέθοδοι διεπαφής (API methods) από τα άλλα εκτελέσιμα που τρέχουν εκεί γίνονται διαθέσιμες σε αυτό, σαν να πρόκειται για τοπικές κλήσεις, καθώς το NAOqi κάνει αυτόματα την επιλογή μεταξύ μιας απευθείας κλήσης (LPC) ή μιας απομακρυσμένης κλήσης (RPC). Στην Εικόνα 12 σχηματοποιείται με ένα απλό παράδειγμα αυτή η δυνατότητα για ανάπτυξη κατανεμημένων εφαρμογών. Σε αυτό, ο συντονισμός των ρομπότ γίνεται από έναν υπολογιστή και αυτά επικοινωνούν μεταξύ τους, ανταλλάσσοντας πληροφορίες.



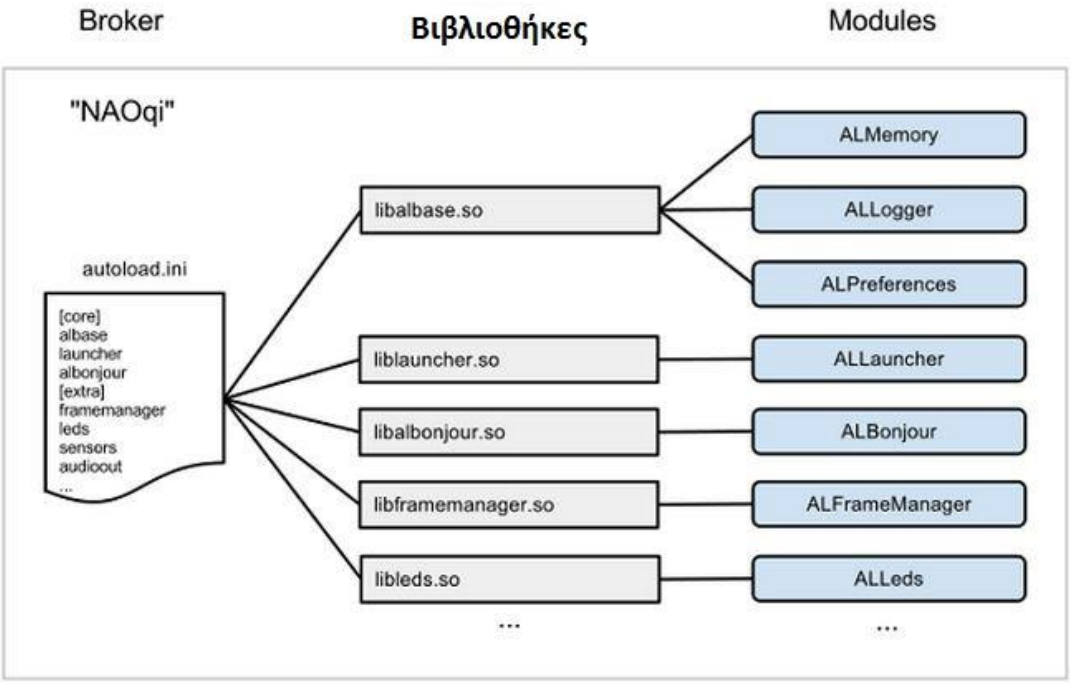
Εικόνα 12 - Παράδειγμα κατανεμημένης εφαρμογής με την πλατφόρμα NAOqi.

Ο μηχανισμός χάρις στον οποίο γίνεται αυτό εφικτό προσφέρεται από το πρόγραμμα NAOqi, το οποίο αναφέρθηκε παραπάνω. Το NAOqi είναι στην ουσία ένας «μεσάζων» (στο εξής θα αναφέρεται με τον αγγλικό όρο broker), που επιτρέπει στο ρομπότ να γνωρίζει όλες τις διαθέσιμες συναρτήσεις διεπαφής προγραμματισμού εφαρμογών (API). Αυτές περιέχονται σε μονάδες λογισμικού (modules) τα οποία οργανώνονται σε βιβλιοθήκες. Το NAOqi, που ξεκινάει μόλις το ρομπότ τεθεί σε λειτουργία, φορτώνει τις βιβλιοθήκες που χρειάζονται, οι οποίες ορίζονται σε ένα αρχείο με όνομα autoload.ini. Με αυτόν τον τρόπο τα διάφορα modules «διαφημίζουν» τις μεθόδους που διαθέτουν και μπορούν να έχουν πρόσβαση σε κάθε μέθοδο που διαφημίστηκε από τον τοπικό broker, ή από κάποιον άλλο broker στο δίκτυο. Αυτό απεικονίζεται στην Εικόνα 13.



Εικόνα 13 - Η κλήση των μεθόδων του NAOqi API γίνεται εφικτή με τον broker NAOqi

Στην Εικόνα 14 παρουσιάζεται ένα δέντρο που σχηματίζεται όταν φορτωθούν τα modules σε έναν broker, με τις μεθόδους να συνδέονται με τα modules και αυτά στον τοπικό broker, ο οποίος παρέχει και πρόσβαση στο δίκτυο.



Εικόνα 14 - Η σύνδεση των modules στον Broker

Ο τρόπος με τον οποίο καλείται μια μέθοδος ενός module, είναι με τη δημιουργία ενός proxy για αυτό, που συμπεριφέρεται όπως το module που αναπαριστά. Αν τα δύο modules βρίσκονται στον ίδιο broker, κατά τη δημιουργία του proxy χρειάζεται να περαστεί ως όρισμα μόνο το όνομα του module στο οποίο θέλουμε να συνδεθούμε. Αν βρίσκονται σε διαφορετικούς, πρέπει να περαστούν και η IP διεύθυνση και η θύρα του broker.

Στο NAOqi φορτώνονται αυτόματα modules τα οποία είναι πάντα διαθέσιμα και προσφέρουν τις διεπαφές που χρειάζονται για το χειρισμό του ρομπότ, κάνοντας διαθέσιμες λειτουργίες που έχουν σχέση με την κίνηση, την ομιλία, τη λήψη πληροφοριών από τους αισθητήρες, την αντίληψη, την αναγνώριση προσώπων και ομιλίας, την παρακολούθηση προσώπων και αντικειμένων, την αυτονομία του ρομπότ, την πρόσβαση σε μια απεικόνιση της μνήμης του και άλλες. Ωστόσο, ο προγραμματιστής μπορεί να δημιουργήσει το δικό του module, το οποίο είναι συνήθως μια κλάση, η οποία κληρονομεί από την ALModule. Το module αυτό μπορεί είτε να τρέξει ως εκτελέσιμο εκτός ρομπότ, για τη διευκόλυνση της χρήσης του και της αποσφαλμάτωσης του, είτε να μεταγλωττιστεί ως βιβλιοθήκη για αποκλειστική χρήση μέσα στο ρομπότ, για μεγαλύτερη αποδοτικότητα σε ότι αφορά την ταχύτητα, αλλά και για περιπτώσεις όπου χρειάζεται απευθείας πρόσβαση στη μνήμη του.

Έτσι λοιπόν, δύο modules μπορεί να είναι μεταξύ τους τοπικά (local), αν έχουν φορτωθεί στον ίδιο broker, ή απομακρυσμένα (remote). Στην πρώτη περίπτωση, όπως είναι φυσιολογικό, για την επικοινωνία τους χρειάζεται μόνο ένας broker. Στη δεύτερη, η επικοινωνία γίνεται μέσω του δικτύου και, όπως αναφέρθηκε και πριν, ο broker είναι υπεύθυνος για το κομμάτι αυτό. Η επικοινωνία μεταξύ δύο απομακρυσμένων modules μπορεί να είναι μονής ή διπλής κατεύθυνσης. Στην πρώτη περίπτωση δημιουργείται ένας proxy που θα συνδεθεί με τον broker στον οποίο είναι καταχωρημένο το άλλο module. Στη δεύτερη, δηλαδή στην αμοιβαία επικοινωνία, χρειάζεται πρώτα να συνδεθούν δύο broker μεταξύ τους, ένας στον οποίο είναι καταχωρημένο το ένα module και ένας στον οποίο έχει καταχωρηθεί το άλλο και, στη συνέχεια, ο proxy που έχει δημιουργηθεί στο ένα επικοινωνεί με τον broker του άλλου.

Προγραμματισμός με Python

Για την ανάπτυξη κώδικα με τη χρήση της Python, γίνεται χρήση του NAOqi Python API, το οποίο περιέχει τρία Python modules. Το πρώτο είναι η κλάση ALProxy, που επιτρέπει τη δημιουργία ενός proxy για κάποιο module. Μια υπόσταση αυτής της κλάσης μπορεί να δημιουργηθεί με δύο τρόπους, ανάλογα με το αν έχει δημιουργηθεί κάποιος broker από τη μεριά αυτής της διεργασίας ή όχι. Στην πρώτη περίπτωση, αυτό γίνεται ως εξής:

ALProxy(name, ip, port), όπου

- *name* είναι το όνομα του module στου οποίου τις μεθόδους θέλουμε να αποκτήσουμε πρόσβαση
- *ip* είναι η ip του broker στο οποίο τρέχει αυτό το module
- *port* είναι η θύρα αυτού του broker

Στη δεύτερη περίπτωση, ο δημιουργός είναι:

ALProxy(name)

όπου *name* είναι το όνομα του module.

Στην περίπτωση που είναι επιθυμητή η δημιουργία κάποιου module γραμμένου σε Python, χρησιμοποιούνται και δύο άλλες κλάσεις, η ALBroker και η ALModule. Πρώτα δημιουργείται μια υπόσταση της ALBroker, με την οποία δημιουργείται ένας broker:

ALBroker(name, ip, port, parent_ip, parent_port), όπου:

- *name* είναι το όνομα του broker.
- *ip* είναι η διεύθυνση IP στην οποία θα ακούει αυτός.
- *port* είναι η θύρα στην οποία αυτός θα ακούει. Αν τεθεί στην τιμή 0, ο broker θα βρει και θα χρησιμοποιήσει μια από τις ελεύθερες θύρες.
- *parent_ip* είναι η IP του broker με τον οποίο αυτός θα συνδεθεί.
- *parent_port* είναι η θύρα του άλλου broker.

Στη συνέχεια μπορεί να κατασκευαστεί το module, με το δημιουργό της κλάσης ALModule:

ALModule(name)

όπου name είναι το όνομα αυτού του module.

Απαραίτητη προϋπόθεση είναι να έχει εγκατασταθεί στον υπολογιστή το Python NAOqi SDK, το οποίο μπορεί να βρεθεί στη σελίδα community.aldebaran.com, εφόσον έχει γίνει πρώτα εγγραφή του χρήστη. Αυτό υπόκειται σε αναβαθμίσεις, όπως άλλωστε και το ίδιο το NAOqi και το λειτουργικό σύστημα του NAO, οπότε πρέπει να προσεχθεί να υπάρχει συμβατότητα ανάμεσα σε αυτά. Επίσης, πρέπει να υπάρχει συμβατότητα με την εγκατεστημένη έκδοση της Python. Για παράδειγμα, η έκδοση Python NAOqi SDK 2.1.3.3, η οποία χρησιμοποιήθηκε για αυτήν την εργασία είναι συμβατή με την Python 2.7.

Εφόσον γίνει η εγκατάσταση του SDK, θα πρέπει η διαδρομή της βιβλιοθήκης του NAOqi να προστεθεί στο PYTHONPATH, με την παρακάτω εντολή (αν για παράδειγμα το κατεβασμένο αρχείο έχει τοποθετηθεί σε ένα φάκελο naoqi, με το όνομα του αρχείου στην εντολή να αλλάζει φυσικά ανάλογα με την έκδοση):

```
$ export PYTHONPATH=~/naoqi/pynaoqi-python2.7-2.1.3.3-linux64:$PYTHONP
ATH
```

Για να αποθηκευτεί αυτή η επιλογή και να παραμείνει σε κάθε μελλοντική εκκίνηση κάποιου τερματικού, πρέπει να προστεθεί η PYTHONPATH στο αρχείο bashrc, με την εντολή:

```
$ echo 'export PYTHONPATH=~/naoqi/pynaoqi-python2.7-2.1.3-linux64:$PY
THONPATH' >> ~/.bashrc
```

Κεφάλαιο 4
Ανάλυση, Σχεδίαση και Υλοποίηση της Εφαρμογής

Σε αυτό το κεφάλαιο, θα παρουσιαστούν τα βήματα που ακολουθήθηκαν για τη δημιουργία της εφαρμογής. Πρόκειται για μια εφαρμογή που προσφέρει στο χρήστη ένα γραφικό περιβάλλον για τη διευκόλυνση της αλληλεπίδρασης με το ρομπότ NAO.

4.1 Στόχοι της Εφαρμογής

Ο στόχος ήταν η δημιουργία μιας εφαρμογής η οποία θα παρείχε στο χρήστη τις παρακάτω δυνατότητες:

- να στέλνει μια εντολή συγκεκριμένης κίνησης στο ρομπότ,
- να πληκτρολογεί και να στέλνει μια φράση, η οποία θα μετατρέπεται σε ομιλία από αυτό
- να μπορεί να παρακολουθήσει την πληροφορία των αισθητήρων αφής που αυτό διαθέτει.

4.2 Το Πλαίσιο Ανάπτυξης μιας Γραφικής Διεπαφής Χρήστη στο ROS

Η ανάπτυξη μιας εφαρμογής γραφικού περιβάλλοντος (GUI) στο ROS γίνεται εφικτή με τη χρήση της πλατφόρμας rqt, η οποία βασίζεται στην Qt. Η Qt είναι μια πλατφόρμα ανάπτυξης λογισμικού που χρησιμοποιείται κυρίως για εφαρμογές GUI και μπορεί να ενσωματωθεί σε πολλές γλώσσες, ανάμεσα στις οποίες βρίσκεται και η Python. Μάλιστα, αυτή είναι η προτεινόμενη γλώσσα για την ανάπτυξη εφαρμογών στην πλατφόρμα rqt, καθώς η πλειοψηφία των πακέτων που έχουν ήδη δημιουργηθεί είναι γραμμένα σε αυτήν. [10]. Για το λόγο αυτό, για την ανάπτυξη αυτής της εφαρμογής αποφασίστηκε να χρησιμοποιηθεί αυτή. Υπάρχουν δύο πάροχοι υλοποιήσεων της Qt για την Python, η PyQt και η PySide. Στο ROS indigo υποστηρίζεται η Qt 4.8. Υπάρχει το μεταπακέτο python_qt_binding, η οποία είναι ήδη εγκατεστημένη στο σύστημα αν έχει προηγηθεί πλήρης εγκατάσταση του ROS και περιέχει και τις δύο υλοποιήσεις της Qt στην Python, παρέχοντας δυνατότητα εναλλαγής της χρήσης τους.

Στο ROS ο πιο συνηθισμένος (και ο προτεινόμενος) τρόπος να τρέξουν τα παράθυρα που περιέχουν τα γραφικά στοιχεία (widgets) είναι ως plugins στο

πρόγραμμα rqt_gui, το οποίο είναι και αυτό ένα γραφικό στοιχείο που επιτρέπει την ενσωμάτωση σε αυτό πολλών παραθύρων ταυτόχρονα και την εύκολη διαρρύθμιση τους εκεί (μεγέθυνση, αλλαγή θέσης κλπ). Η λογική για τη δημιουργία ενός plugin παρέχεται στο μεταπακέτο qt_gui.

Για τη δημιουργία ενός πακέτου στο οποίο θα περιέχεται ένα rqt plugin, απαιτούνται περισσότερα βήματα από αυτά που χρειάζονται για τη δημιουργία ενός απλού πακέτου. Για την ακρίβεια, δεν αρκούν μόνο τα αρχεία CMakeLists.txt και package.xml, αλλά χρειάζονται και τέσσερα ακόμα αρχεία. Κάποια από αυτά προκύπτουν από το ότι πρόκειται για ένα πακέτο στο οποίο περιέχεται κώδικας της Python και άλλα λόγω του ότι η εφαρμογή είναι ένα plugin.

Για να μπορεί να βρεθεί το plugin, πρέπει να δημιουργηθεί ένα αρχείο plugin.xml. Το αρχείο που φτιάξαμε φαίνεται στην παρακάτω εικόνα.

plugin.xml

```
<library path="src">
  <class name="My Plugin for moving NAO" type="rqt_naomove.my_module.MyPlugin"
base_class_type="rqt_gui_py::Plugin">
    <description>
      A Python GUI plugin to create a user interface for moving NAO.
    </description>
    <qtgui>
      <!-- optional grouping...
      <group>
        <label>Group</label>
      </group>
      <group>
        <label>Subgroup</label>
      </group>
      -->
      <!--
      To onoma tou plugin
      Me afto to epilegw ston katalogo Plugins tou rqt
      -->
      <label>NAO Plugin</label>
      <icon type="theme">system-help</icon>
      <statustip>Great user interface to provide real value.</statustip>
    </qtgui>
  </class>
</library>
```

XML ▾ Tab Width: 8 ▾ Ln 19, Col 17 INS

Εικόνα 15 - Το αρχείο plugin.xml της εφαρμογής

Η ύπαρξη αυτού του αρχείου, χρειάζεται να αναφερθεί στο package.xml. Αυτό γίνεται προσθέτοντας τις παρακάτω γραμμές:

```
<export>
  <rqt_gui plugin="${prefix}/plugin.xml"/>
</export>
```

Πριν γίνει αναφορά στα υπόλοιπα αρχεία που χρειάζονται, θα πρέπει να ξεκαθαριστεί ότι η δομή ενός πακέτου χρειάζεται να ακολουθεί κάποια πρότυπα. Πιο συγκεκριμένα, στο πακέτο που δημιουργήθηκε, τα αρχεία CMakeLists, package.xml και plugin.xml βρίσκονται στο βασικό κατάλογο, κάτι που πρέπει να ισχύει πάντα. Στην περίπτωσή μας, το πακέτο ονομάζεται rqt_nao, οπότε αυτά βρίσκονται στη διαδρομή …/rqt_nao/. Το αρχείο όμως του plugin, που ονομάστηκε my_module, τοποθετήθηκε στη διαδρομή …/rqt_nao/src/rqt_nao/. Αυτή η πρακτική πρέπει να ακολουθείται πάντα, διότι ο φάκελος src σε ένα πακέτο είναι αυτός που εξάγεται αυτόματα στο PYTHONPATH.

Το my_module.py, συνοδεύεται και από ένα αρχείο __init__.py, το οποίο είναι απαραίτητο (από προσωπική εμπειρία) για την εκτέλεση ενός αρχείου python. Το __init__.py χρειάζεται για να σημαδέψει ένα φάκελο στο δίσκο ως πακέτο python.

Απαραίτητη είναι και η ύπαρξη ενός αρχείου setup.py. Αυτό έχει να κάνει και πάλι με την ύπαρξη κώδικα σε Python. Για τα python πακέτα, χρειάζεται να οριστεί από το χρήστη μια διαδικασία εγκατάστασης, η οποία προορίζεται να διαβαστεί από το catkin. Το αρχείο setup.py που δημιουργήθηκε για το πακέτο μας φαίνεται στην Εικόνα 16. Όταν υπάρχει ένα τέτοιο αρχείο, πρέπει να κληθεί στο CMakeLists.txt και η catkin_python_setup() (δεν λαμβάνει ορίσματα).

*setup.py ×

```
## ! DO NOT MANUALLY INVOKE THIS setup.py, USE CATKIN INSTEAD

from distutils.core import setup
from catkin_pkg.python_setup import generate_distutils_setup

# fetch values from package.xml
d = generate_distutils_setup(
    packages=['rqt_nao'],
    package_dir={'': 'src'},
)

setup(**d)
```

Εικόνα 16 - Το αρχείο setup.py

Σε ότι αφορά το προγραμματιστικό κομμάτι, η διαδικασία της ανάπτυξης μιας γραφικής διεπαφής χρήστη με τη χρήση της Qt περιλαμβάνει την προσθήκη των γραφικών στοιχείων, τον ορισμό της διάταξής τους (layout) και τον ορισμό του

κώδικα που θα εκτελεστεί όταν συμβεί κάποιο γεγονός (και στη σύνδεση μεταξύ του γεγονότος και του αντίστοιχου κώδικα).

Οι κλάσεις της PyQt4 ταξινομούνται σε αρκετά modules, σημαντικότερο από τα οποία είναι το QtGui, στο οποίο περιέχονται οι κλάσεις των γραφικών στοιχείων και αρκετές ακόμα σχετικές με αυτές, που μας δίνουν τη δυνατότητα για παράδειγμα να εισάγουμε χρώματα ή άλλες ιδιότητες. Τα γραφικά στοιχεία μπορεί να είναι κουμπιά, παράθυρα, εργαλειοθήκες (toolbars), μπάρες κατάστασης (status bars), πλαίσια εισαγωγής κειμένου και πολλά άλλα. Ένα άλλο αρκετά σημαντικό module είναι το QtCore, στο οποίο υπάρχουν κλάσεις σχετικές με το χρόνο, τα αρχεία και τους φακέλους, τα νήματα και τις διεργασίες.

Υπάρχουν κάποια γραφικά στοιχεία τα οποία λειτουργούν ως «δοχεία» (containers) για την τοποθέτηση άλλων γραφικών στοιχείων μέσα σε αυτά. Ένα τέτοιο γραφικό στοιχείο είναι μια υπόσταση της κλάσης QWidget και τα στοιχεία που περιέχονται σε αυτό είναι τα παιδιά του. Αντίστοιχα, ένα τέτοιο αντικείμενο είναι ο πατέρας του. Ένα γραφικό στοιχείο το οποίο δεν έχει πατέρα (τιμή 0 στην παράμετρο parent κατά τη δημιουργία του) ονομάζεται παράθυρο (window). Όπως είναι λογικό, ένα παράθυρο μπορεί να περιέχει πολλά containers.

Ένας τρόπος να ορίσουμε το πώς θα τοποθετηθούν τα γραφικά στοιχεία μέσα σε ένα παράθυρο ή σε ένα container είναι με τον ορισμό της απόλυτης θέσης τους (absolute positioning) με την κλήση της μεθόδου move(int x, int y) στην οποία δηλώνουμε τις συντεταγμένες στις οποίες θα βρίσκεται το γραφικό στοιχείο, με αρχή την πάνω αριστερή γωνία του container και τις τιμές x και y να μεγαλώνουν από δεξιά προς τα αριστερά και από πάνω προς τα κάτω αντίστοιχα. Ωστόσο, με αυτόν τον τρόπο υπάρχουν σοβαροί περιορισμοί. Το μέγεθος και η τοποθέτηση ενός στοιχείου δεν αλλάζει αν αλλαχθεί το μέγεθος ενός παραθύρου, η αλλαγή μεγέθους της γραμματοσειράς σε κάποιο γραφικό στοιχείο ή σε όλη την εφαρμογή μπορεί να καταστρέψει τη διάταξη και επίσης, αν χρειαστεί να αλλαχθεί κάτι, η διάταξη πρέπει να ξαναγίνει από την αρχή. Για αυτούς τους λόγους στη συντριπτική πλειοψηφία των περιπτώσεων προτιμάται μια άλλη μέθοδος.

Ο δεύτερος τρόπος είναι η χρήση κάποιων κλάσεων διαχείρισης της διάταξης (layout management classes) στα πλαίσια ενός container. Οι πιο συχνά χρησιμοποιούμενες είναι η QVBoxLayout, η οποία ορίζει ότι τα γραφικά στοιχεία που προστίθενται θα τοποθετούνται σε κάθετη διάταξη, η QHBoxLayout, στην οποία τα γραφικά στοιχεία τοποθετούνται οριζόντια και η QGridLayout, για διάταξη των στοιχείων σε μορφή πλέγματος. Η χρήση κάποιας συγκεκριμένης διάταξης ορίζεται με τη μέθοδο QWidget.setLayout(self, QLayout), όπου στην παράμετρο QLayout τοποθετούμε μια υπόσταση μιας από τις παραπάνω κλάσεις διαχείρισης της διάταξης, ενώ η προσθήκη κάποιου γραφικού στοιχείου στη διάταξη γίνεται με τη μέθοδο addWidget(), με πρώτο όρισμα το γραφικό στοιχείο που θα προστεθεί

και πρόσθετα ορίσματα που διαφέρουν ανάλογα με τη διάταξη που χρησιμοποιείται. Οι κλάσεις διαχείρισης της διάταξης μπορούν να χρησιμοποιηθούν και συνδυαστικά, με τη χρήση της μεθόδου addLayout() που αυτές διαθέτουν.

Η δήλωση των γραφικών στοιχείων και η τοποθέτησή τους σε συγκεκριμένες διατάξεις μπορεί να γίνει ξεχωριστά από το κομμάτι του προγράμματος που έχει να κάνει με το χειρισμό των γεγονότων, σε ένα αρχείο με κατάληξη «.ui», με τη χρήση της γλώσσας xml. Το αρχείο αυτό μπορεί να φορτωθεί στο πρόγραμμα με τη μέθοδο loadUi(). Αυτή η επιλογή δεν προτιμήθηκε στην εφαρμογή που αναπτύχθηκε.

Σε ότι αφορά το χειρισμό των γεγονότων, ο τρόπος που γίνεται αυτός ονομάζεται στην Qt μηχανισμός «signal and slot». Κάποιο γεγονός, το οποίο τις περισσότερες φορές, αλλά όχι πάντα, προκαλείται από το χρήστη της εφαρμογής, προκαλεί τη δημιουργία ενός σήματος, το οποίο πρέπει να συνδεθεί με ένα slot (θυρίδα). Ένα slot δεν είναι τίποτα παραπάνω από μια μέθοδο που καλείται από αυτό το σήμα και αντιδρά με κάποιο τρόπο σε αυτό. Ο τρόπος με τον οποίο συνδέεται ένα σήμα με ένα slot άλλαξε στην έκδοση 4.5 της PyQt και γίνεται ως εξής: Έστω ότι έχουμε ορίσει ένα κουμπί της κλάσης QPushButton με όνομα button και θέλουμε να γίνει κάτι όταν αυτό πατηθεί. Υπάρχει ήδη ένα σήμα που αντιστοιχεί σε αυτό το γεγονός και είναι το σήμα clicked. Η παρακάτω γραμμή κώδικα συνδέει αυτό το σήμα με μια μέθοδο με όνομα buttonClicked(), η οποία θα οριστεί αργότερα για να δηλώσουμε τι θα γίνεται κάθε φορά που πατιέται το κουμπί με όνομα button:

```
button.clicked.connect(self.buttonClicked)
```

Στην PyQt χρειάζεται να δημιουργηθεί μια υπόσταση της κλάσης QApplication, της οποίας ο ρόλος είναι να καλεί τον κύριο βρόχο του προγράμματος στον οποίο γίνεται ο έλεγχος για την ύπαρξη κάποιου γεγονότος. Ωστόσο, στην υλοποίηση στο ROS δεν χρειάζεται να γίνει κάτι τέτοιο καθώς αυτό έχει ήδη ενσωματωθεί.

Συμπερασματικά, η ανάπτυξη της εφαρμογής χωρίστηκε νοητά σε τρία κομμάτια. Το πρώτο αφορά τα γραφικά στοιχεία τα οποία δίνουν στο χρήστη τη δυνατότητα να αλληλεπιδράσει με το ρομπότ, το δεύτερο τις ενέργειες που γίνονται όταν συμβεί κάποιο γεγονός και το τρίτο τη διάταξη και τη μορφοποίηση των γραφικών στοιχείων.

Σε ότι αφορά τις επιμέρους λειτουργίες, η πρώτη αφορά την αποστολή στο ρομπότ εντολής κίνησης, ενέργεια που θα προκαλείται με το πάτημα ενός κουμπιού από το χρήστη. Η δεύτερη την αποστολή εντολής ομιλίας στο ρομπότ, το περιεχόμενο της οποίας ορίζεται με είσοδο κειμένου από το χρήστη σε μια υποδοχή

στην εφαρμογή. Η τρίτη λειτουργία δεν εκκινείται από το χρήστη αλλά με την έναρξη της εφαρμογής. Αυτήν τη φορά δεν πρόκειται για αποστολή κάποιας εντολής από το χρήστη προς το ρομπότ, αλλά για οπτικοποίηση δεδομένων που προέρχονται από αυτό και πιο συγκεκριμένα του γεγονότος αλλαγής της τιμής των μεταβλητών που διατηρούν την κατάσταση των αισθητήρων αφής που βρίσκονται στο κεφάλι και στα πόδια του από «touched» (όταν αυτοί αναγνωρίζουν κάποια επαφή) σε «untouched», όταν δεν υφίσταται επαφή σε αυτά τα σημεία.

4.3 Σχεδίαση της Εφαρμογής και Επικοινωνία με το Υπάρχον Λογισμικό

Η αρχιτεκτονική που ακολουθήθηκε κατά την ανάπτυξη της εφαρμογής ακολουθεί τα πρότυπα του ROS, κάτι που προσφέρει όλα τα πλεονεκτήματα που αναλύθηκαν στα προηγούμενα κεφάλαια. Αυτό σημαίνει ότι η απευθείας επικοινωνία με το λογισμικό του ρομπότ, δηλαδή με τις μεθόδους των modules που φορτώνονται σε αυτό κατά την εκκίνησή του και το ελέγχουν, δεν αποτελεί κομμάτι του plugin της rqt. Αντίθετα, χρησιμοποιήθηκαν ενδιάμεσες μονάδες λογισμικού, οι οποίες αναλαμβάνουν την επικοινωνία με αυτό, με τη χρήση των κλάσεων ALProxy, ALModule και ALBroker, των οποίων η λειτουργία εξηγήθηκε στο Κεφάλαιο 3. Υπάρχουν κάποια έτοιμα τέτοια πακέτα, που περιέχουν κώδικα στον οποίο ενσωματώνονται τα απαιτούμενα κομμάτια του NaoQi API ώστε να διευκολυνθεί η χρήση του στο ROS. Αυτά είναι τα naoqi_bridge, nao_robot και nao_extras. Για τη λειτουργία αυτών, χρειάστηκε να προηγηθεί η εγκατάσταση του NAOqi SDK στον υπολογιστή, όπως αυτή περιγράφηκε στην Παράγραφο 3.2. Η διαδικασία της εγκατάστασης αυτών των πακέτων, αναφέρεται στο Παράρτημα 1.

Το μεταπακέτο (metapacket: απλώς μια συλλογή από λογικά συνδεδεμένα πακέτα) naoqi_bridge περιέχει πακέτα που δημιουργούν μια «γέφυρα» με το NAOqi. Εκεί υπάρχει το naoqi_msgs στο οποίο περιέχονται τα μηνύματα που χρησιμοποιούνται. Στο naoqi_pose υπάρχουν κόμβοι που παρέχουν πρόσβαση στις μεθόδους κίνησης του NAO. Επίσης, υπάρχουν τα naoqi_driver και naoqi_driver_py. Στο naoqi_driver_py περιέχεται μεταξύ άλλων το naoqi_node.py στο οποίο δημιουργείται μια κλάση με όνομα NaoqiNode η οποία κληρονομείται από αρκετές άλλες, παρέχοντας μια βάση για τη σύνδεση με το NAOqi με τη δημιουργία proxy για modules του NAO (με τη χρήση της μεθόδου ALProxy) και για το χειρισμό του τερματισμού του ROS.

Το μεταπακέτο nao_robot περιέχει τα πακέτα nao_description, nao_bringup και nao_apps. Στο nao_description υπάρχουν αρχεία περιγραφής και αναπαράστασης του NAO που μπορούν να χρησιμοποιηθούν σε εργαλεία προσομοίωσης όπως το rviz και το Gazebo. Στο nao_bringup υπάρχουν δύο αρχεία «launch» με τα οποία είναι εφικτή η ταυτόχρονη εκκίνηση των περισσότερων από

τους κόμβους που περιέχονται στα σχετικά με το NAO πακέτα, ώστε να μη χρειάζεται να εκκινείται ένα διαφορετικό terminal για κάθε κόμβο.

4.3.1 Λειτουργία Αποστολής Εντολής Κίνησης

Ο κόμβος pose_controller που βρίσκεται στο πακέτο naoqi_pose παρέχει πρόσβαση στα βασικότερα modules από αυτά που είναι υπεύθυνα για την κίνησή του NAO. Αυτά είναι το ALMotion και το ALRobotPosture. Η κλάση PoseController κληρονομεί από τη NaoqiNode και δημιουργεί proxy σε αυτά τα modules. Οι μέθοδοι που αυτά διαθέτουν είναι αρκετές και ο pose_controller προσφέρει πρόσβαση σε πολλές από αυτές, άρα υπάρχουν και πολλές επιλογές.

Γενικότερα, η ALMotion περιέχει μεθόδους που έχουν να κάνουν με τον έλεγχο της κίνησης του ρομπότ έπειτα από εντολή από το χρήστη, αλλά και με αυτόνομη συμπεριφορά που χρειάζεται για την αποφυγή εμποδίων, τη διαχείριση πτώσης και άλλα. Στα πλαίσια της εφαρμογής πιο ενδιαφέρουσες είναι οι πρώτες και μπορούν να κατηγοριοποιηθούν παραπάνω, σε ότι αφορά τη λειτουργικότητά τους, σε αυτές που έχουν να κάνουν με τον έλεγχο των αρθρώσεων, με το περπάτημα και με την αντίστροφη κινηματική. Επίσης, υπάρχουν μέθοδοι με τις οποίες ρυθμίζεται η «σκληρότητα» των αρθρώσεων, δηλαδή το πόσο εύκαμπτες είναι αυτές, που είναι μια σημαντική παράμετρος διότι μια εντελώς «ελεύθερη» άρθρωση δεν μπορεί να ελεγχθεί από τον ελεγκτή της.

Η angleInterpolation(names, angleLists, timeLists, isAbsolute) μετακινεί μια ή περισσότερες αρθρώσεις προς μια γωνία που αποτελεί το στόχο ή μεταξύ μιας λίστας από γωνίες που καθορίζουν μια τροχιά. Names είναι τα ονόματα των αρθρώσεων, angleLists είναι μια λίστα με τις γωνίες που πρόκειται να ακολουθήσει ένας σύνδεσμος ή μια λίστα λιστών, αν πρόκειται για το χειρισμό περισσότερων του ενός συνδέσμων. TimeLists είναι μια λίστα ή μια λίστα λιστών η οποία περιέχει τους αντίστοιχους χρόνους στους οποίους πρέπει οι γωνίες των συνδέσμων να έχουν τις τιμές που έχουν οριστεί. Με τη Boolean isAbsolute καθορίζεται αν οι τιμές των γωνιών είναι απόλυτες ή σχετικές με την τρέχουσα γωνία. Οι τιμές των γωνιών δίνονται σε rad, όπως και σε όλες τις υπόλοιπες μεθόδους.

Η angleInterpolationWithSpeed(names, targetAngles, maxSpeedFraction) μετακινεί μια ή περισσότερες αρθρώσεις προς μια γωνία – στόχο. Σε αυτήν την περίπτωση, επιτρέπεται ο ορισμός μόνο μιας γωνίας – στόχου και καμίας άλλης παρεμβάλλουσας τιμής. Οι αρθρώσεις καθορίζονται στη names και οι γωνίες – στόχοι στην targetAngles. Η ταχύτητα της κίνησης καθορίζεται με τη maxSpeedFraction και είναι ένα κλάσμα της μέγιστης δυνατής.

Μια παρόμοια με την angleInterpolationWithSpeed μέθοδος είναι η setAngles(names, angles, fractionMaxSpeed), με τη διαφορά ότι το νήμα στο οποίο αυτή εκτελείται μπορεί να ανακληθεί (non-blocking).

Για το χειρισμό της ελαστικότητας των αρθρώσεων υπάρχει οι stiffnessInterpolation(names, stiffnessLists, timeLists), που έχει ανάλογη λειτουργία με την angleInterpolation(), μόνο που στη θέση των γωνιών μπαίνουν οι τιμές των ελαστικοτήτων. Επίσης, υπάρχει η setStiffnesses(names, stiffnesses), στην οποία δεν χρησιμοποιείται κάποια παράμετρος χρόνου και είναι non-blocking. Για να ρυθμιστεί η ελαστικότητα όλων των αρθρώσεων του ρομπότ χρησιμοποιούνται και οι wakeup() και rest().

Η ALRobotPosture παρέχει την goToPosture(string postureName, float speed) η οποία μετακινεί το ρομπότ σε μια προκαθορισμένη στάση. Το NAO προσδιορίζει την τρέχουσα θέση του, υπολογίζει μια διαδρομή από αυτήν προς τη στάση – στόχο και εκτελεί την κίνηση.

Για κάθε μια από τις παραπάνω μεθόδους παρέχεται πρόσβαση από τον κόμβο pose_controller. Κάθε μια από αυτές καλείται όταν φτάσει σε αυτόν τον κόμβο κάποιο μήνυμα ή κάποια αίτηση από μια υπηρεσία, καθώς αυτός λειτουργεί ταυτόχρονα ως παροχέας υπηρεσιών, ως action server αλλά και ως subscriber, αφού συνολικά εκκινούνται και λειτουργούν 15 τέτοια αντικείμενα.

Για τις ανάγκες της εφαρμογής, και πιο συγκεκριμένα για την πρώτη λειτουργία, επιλέχθηκε να χρησιμοποιηθεί η μέθοδος angleInterpolationWithSpeed(). Αυτή καλείται μόλις φτάσει στον pose_controller κάποιο μήνυμα στο joint_angles_action. Πρόκειται, όπως προδίδει και το όνομα, για ένα action, για το οποίο ο Action Server δημιουργείται ως εξής:

self.jointAnglesServer = actionlib.SimpleActionServer(“joint_angles_action”, JointAnglesWithSpeedAction, execute_cb=self.executeJointAnglesWithSpeedAction, auto_start=False)

Από τη δημιουργία αυτού του αντικειμένου, συμπεραίνουμε ότι ο στόχος πρέπει να είναι ένα μήνυμα JointAnglesWithSpeedGoal. Γενικά, όλα τα μηνύματα που χρησιμοποιούνται για αυτό το action βρίσκονται στον κατάλογο naoqi_bridge_msgs (στο opt/ros/indigo/share) και δημιουργούνται από το αρχείο JointAnglesWithSpeed.action, που είναι ένα προκαθορισμένο αρχείο περιγραφής action και φαίνεται παρακάτω:

```
# goal: ένα καταχωρημένο όνομα στάσης που αποτελεί το στόχο
naoqi_bridge_msgs/JointAnglesWithSpeed joint_angles
---
# αποτέλεσμα είναι η στάση η οποία τελικά επιτεύχθηκε
sensor_msgs/JointState goal_position
---
# δεν έχει οριστεί μέχρι στιγμής feedback
```

Όπως βλέπουμε, ο τύπος του στόχου είναι ένα μήνυμα JointAnglesWithSpeed.msg:

```
Header header
string[] joint_names
float32[] joint_angles
float32 speed
uint8 relative
```

Στην τεκμηρίωσή του αναφέρεται ότι στη λίστα joint_names τοποθετούνται τα ονόματα των συνδέσμων, όπως αυτά ορίζονται στην τεκμηρίωση του NAOqi [11]. Στη joint_angles τοποθετούνται οι τιμές των γωνιών που αποτελούν τους στόχους. Οι δύο λίστες, όπως είναι φυσιολογικό, πρέπει να έχουν το ίδιο μέγεθος, εκτός από την περίπτωση που χρησιμοποιείται μια λέξη – κλειδί στη joint_names, για παράδειγμα η «Body», με την οποία αναφερόμαστε σε όλους τους συνδέσμους ταυτόχρονα. Στη μεταβλητή speed τοποθετείται η επιθυμητή ταχύτητα με την οποία θα γίνει η κίνηση, η οποία είναι τύπου float32 και παίρνει τιμές από 0 ως 1, κάτι λογικό αν θυμηθεί κανείς ότι αυτή θα χρησιμοποιηθεί κατά την κλήση της μεθόδου angleInterpolationWithSpeed από την πλευρά του pose_controller. Η relative δεν θα συμπληρωθεί στα πλαίσια αυτής της επικοινωνίας και χρησιμοποιείται σε άλλες περιπτώσεις.

Η κίνηση που επιλέχθηκε να αποστέλλεται με το πάτημα του κουμπιού «» είναι το σήκωμα των χεριών του NAO στο ύψος των ώμων. Στο plugin, ο στόχος δημιουργήθηκε ως εξής:

angle_goal = naoqi_bridge_msgs.msg.JointAnglesWithSpeedGoal()

angle_goal.joint_angles.relative = 0

angle_goal.joint_angles.joint_names = [“LShoulderPitch”,”RShoulderPitch”]

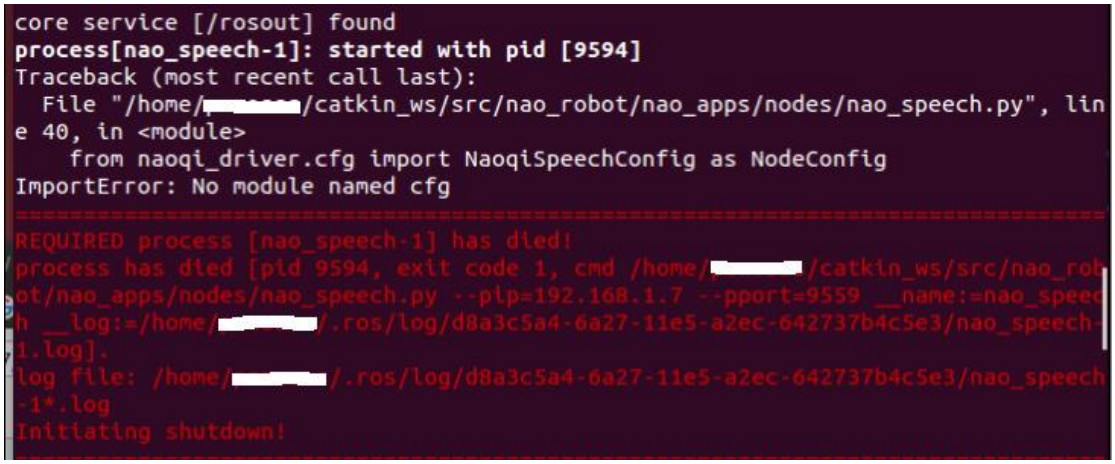
angle_goal.joint_angles.joint_angles = [-1.5, -1.5]

angle_goal.joint_angles.speed = 0.2

Τα όρια των τιμών που μπορούν να πάρουν οι γωνίες κάθε συνδέσμου (σε rad αλλά και σε μοίρες), καθώς και κάποιες ενδείξεις για την ερμηνεία τους, βρίσκονται στο [12], αν και η απόφαση για το ποιες τιμές θα χρησιμοποιηθούν, ανάλογα με την επιθυμητή κίνηση, είναι πολλές φορές και θέμα δοκιμών.

4.3.2 Λειτουργία Αποστολής Κειμένου για Ομιλία

Για τη λειτουργία της αποστολής κειμένου για να μετατραπεί σε ομιλία από το ρομπότ, αρχικά επιχειρήθηκε να χρησιμοποιηθεί ο κόμβος nao_speech.py του nao_robot, χωρίς όμως επιτυχία. Πρόκειται για έναν κόμβο ο οποίος δημιουργήθηκε το 2012, έχει δοκιμαστεί στο NAOqi 1.12 και προοριζόταν να παρέχει στο ROS τις διεπαφές για τις λειτουργίες αναγνώρισης ομιλίας και μετατροπής κειμένου σε ομιλία, η οποία μας ενδιέφερε. Ωστόσο, χρησιμοποιεί ένα αρχείο NaoqiSpeechConfig, το οποίο θα έπρεπε να βρίσκεται στη διαδρομή naoqi_driver/cfg/, αλλά δεν υπάρχει και επίσης δεν βρίσκεται πουθενά στο σύστημα, καθώς έχει αλλάξει. Συγκεκριμένα, κατά την εκκίνηση του συγκεκριμένου κόμβου εμφανίστηκε το μήνυμα:



Εικόνα 17 - Το μήνυμα σφάλματος κατά το τρέξιμο του nao_speech.py

Για το λόγο αυτό, δημιουργήθηκε ένας νέος κόμβος, ο οποίος προσφέρει μόνο τη διεπαφή για τη λειτουργία μετατροπής κειμένου σε ομιλία. Ονομάστηκε «speech2.py» και περιέχει μια κλάση με όνομα NaoSpeech η οποία κληρονομεί από τη NaoqiNode και χρησιμοποιεί τη μέθοδό της, get_proxy(), για να αποκτήσει πρόσβαση στις μεθόδους της ALTextToSpeech, και πιο συγκεκριμένα στη say:

self.tts = self.get_proxy(“ALTextToSpeech”)

Στη συνέχεια, καλείται η say(const std::string& stringToSay), με την οποία το ρομπότ λέει τη λέξη ή φράση που αυτή δέχεται ως παράμετρο. Η παράμετρος είναι κωδικοποιημένη σε UTF-8.

Στο speech2.py το όρισμα της say είναι ένα μήνυμα από το plugin, που είναι τύπου String (το οποίο είναι ένας προκαθορισμένος τύπος μηνύματος και ξανααναφέρθηκε στο παράδειγμα της δημιουργίας ενός κόμβου Publisher στην παράγραφο 2.4) και έχει οριστεί ότι θα το λαμβάνει στο topic “speech”, με τον παρακάτω δημιουργό του subscriber:

self.sub = rospy.Subscriber(“speech”, String, self.say)

Στο plugin, ο publisher που θα στέλνει αυτά τα μηνύματα είναι ο:

self._speechpub = rospy.Publisher(‘speech’, String, queue_size=10)

4.3.3 Λειτουργία Παρακολούθησης των Αισθητήρων Αφής

Σε ότι αφορά την παρακολούθηση της κατάστασης των αισθητήρων αφής, χρησιμοποιήθηκε ως ενδιάμεσος κόμβος ο nao_tactile.py, που περιέχεται στο πακέτο nao_apps του nao_robot. Σε αυτόν υπάρχουν δύο publishers. Ο πρώτος δημοσιεύει στο topic tactile_touch μηνύματα τύπου TactileTouch. Πρόκειται για ένα μήνυμα που βρίσκεται στο naoqi_bridge_msgs και το περιεχόμενό του φαίνεται παρακάτω:

```
uint8 button
uint8 state

uint8 buttonFront=1
uint8 buttonMiddle=2
uint8 buttonRear=3

uint8 stateReleased=0
uint8 statePressed=1
```

Παρατηρούμε ότι αυτό το μήνυμα περιέχει δύο πεδία με μεταβλητές τιμές και πέντε σταθερές. Το πεδίο button παίρνει την τιμή 1 όταν συμβεί αλλαγή στην κατάσταση του μπροστινού αισθητήρα αφής του κεφαλιού στο module ALMemory, που προσομοιώνει τη μνήμη του NAO. Η αλλαγή μπορεί να είναι η μετάβαση του αισθητήρα από ελεύθερο σε πιεσμένο ή το αντίστροφο. Στην πρώτη περίπτωση η μεταβλητή state παίρνει την τιμή 1, ενώ στη δεύτερη γίνεται 0. Σε κάθε τέτοια μετάβαση αποστέλλεται στο plugin ένα τέτοιο μήνυμα. Αντίστοιχα, η μεταβλητή button παίρνει την τιμή 2 όταν υπάρχει αλλαγή στην κατάσταση του μεσαίου αισθητήρα αφής κεφαλιού και 3 για αλλαγή στον πίσω αισθητήρα.

Ο δεύτερος publisher δημοσιεύει στο «bumper», μηνύματα τύπου Bumper, το οποίο επίσης βρίσκεται στο naoqi_bridge_msgs. Η μορφή του είναι η εξής:

```
uint8 bumper
uint8 state

uint8 right=0
uint8 left=1

uint8 stateReleased=0
uint8 statePressed=1
```

Ο τρόπος με τον οποίο συμπληρώνεται αυτό το μήνυμα είναι παρόμοιος με το TactileTouch. Η τιμή 0 δίνεται στη μεταβλητή bumper όταν συμβαίνει αλλαγή στον αισθητήρα του δεξιού ποδιού και η τιμή 1 αντιστοιχεί στο αριστερό πόδι. Η

state είναι η μεταβλητή που δείχνει τον τύπο της μετάβασης και παίρνει τιμή 1 όταν αυτό πιεστεί και 0 όταν αφεθεί. Επομένως, στο plugin, δημιουργήθηκαν δύο subscribers για τη λήψη αυτών των μηνυμάτων.

4.4 Επικοινωνία με το Χρήστη

Η ενεργοποίηση της αποστολής του μηνύματος της κίνησης προς τον pose_controller γίνεται με το πάτημα ενός κουμπιού με όνομα «Hands up!». Ο κατάλληλος τρόπος για την κατασκευή ενός τέτοιου κουμπιού είναι με την κλάση QPushButton. Η δημιουργία ενός αντικειμένου αυτής της κλάσης, με όνομα «_push_button» γίνεται με:

self._push_button = QPushButton(“Hands up!”)

Το σήμα που δημιουργείται με το πάτημα του κουμπιού συνδέεται με μια συνάρτηση με όνομα _button1_clicked, η οποία εκτελεί τη δημιουργία και την αποστολή του στόχου. Αυτό γίνεται με:

self._push_button.clicked.connect(self._button_clicked)

Στην _button_clicked(), εκτελείται η δημιουργία του στόχου, όπως δείχτηκε στην παράγραφο 4.3.1 και αποστέλλεται (με τη send_goal()).

Σε ότι αφορά τη λειτουργία της ομιλίας, Η εισαγωγή του κειμένου που θα σταλεί ως μήνυμα στο topic «speech» γίνεται από το χρήστη σε ένα γραφικό στοιχείο της κλάσης QLineEdit.

Ένα μήνυμα αποστέλλεται μόλις ο χρήστης πατήσει “Enter”, βρισκόμενος στη γραμμή εισαγωγής. Τότε προκαλείται ένα σήμα returnPressed, το οποίο συνδέεται στη συνάρτηση στην οποία διαμορφώνεται και αποστέλλεται το μήνυμα:

request = self._le.text()

self._label1.setText(request)

self._speechpub.publish(request)

Η μέθοδος text() επιστρέφει το κείμενο που εισήγαγε ο χρήστης στο αντικείμενο QLineEdit.

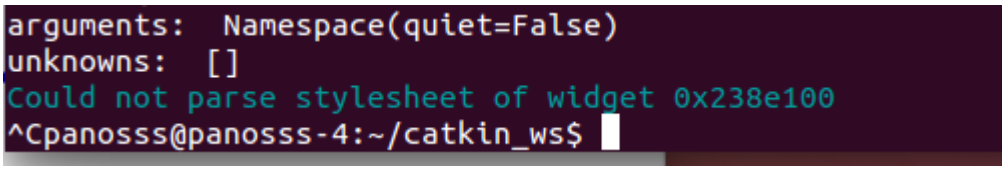
Για την παρακολούθηση της κατάστασης των αισθητήρων, δημιουργήθηκαν πέντε αντικείμενα της κλάσης QFrame (εισάγεται από τη μονάδα QtGui). Ένα τέτοιο αντικείμενο συνήθως έχει το ρόλο του πλαισίου για την τοποθέτηση άλλων γραφικών στοιχείων, όμως στη συγκεκριμένη περίπτωση χρησιμοποιήθηκε για να απεικονιστούν πέντε τετράγωνα, των οποίων το κανονικό χρώμα έχει οριστεί σε μπλε. Κάθε τετράγωνο αντιστοιχεί σε έναν από τους αισθητήρες που αναφέρθηκαν

παραπάνω. Μόλις η κατάσταση κάποιου από αυτούς αλλαχθεί σε «πιεσμένος» (και για όσο αυτό διαρκέσει) το χρώμα του αντίστοιχου τετραγώνου γίνεται κόκκινο.

4.5 Πρόβλημα στον Ορισμό Χρωμάτων Γραφικού Στοιχείου και η Λύση του

Συνήθως για να γίνει ο ορισμός του χρώματος σε κάποιο γραφικό στοιχείο, είτε πρόκειται για χρώμα γεμίσματος (background) είτε για χρώμα γραμματοσειράς, χρησιμοποιούνται stylesheets για τη μορφοποίηση του συγκεκριμένου γραφικού στοιχείου ή μιας ομάδας τέτοιων, σε συνδυασμό με την κλάση QColor(QtGui) με την οποία συνθέτουμε ένα χρώμα από τις τιμές που δίνουμε στις συνιστώσες κόκκινο, πράσινο και μπλε (από 0 ως 255), το οποίο χρησιμοποιούμε στο stylesheet.

Ωστόσο, η χρησιμοποίηση stylesheet απέτυχε:



Εικόνα 18 - Αποτυχία χρήσης stylesheet για τη μορφοποίηση

Αυτό φαίνεται ότι συνέβη, μετά και από μια επίσκεψη στο [13], λόγω του ότι η χρήση των stylesheets είναι γενικότερα προβληματική στα πλαίσια του rqt.

Προτιμήθηκε γι' αυτό μια εναλλακτική λύση, που είναι η χρήση της κλάσης QPalette (QtGui) και της μεθόδου της, setColor, η οποία θέτει κάποιο χρώμα για το φόντο ή για τη γραμματοσειρά ή για πιο εξειδικευμένους ρόλους (τους οποίους μπορεί να δει κάποιος στο [14]) και μπορεί να χρησιμοποιηθεί από ένα ή περισσότερα γραφικά στοιχεία για να χρωματιστούν:

setColor(self, ColorGroup acg, ColorRole acr, QColor acolor)

Η παράμετρος acg δεν είναι υποχρεωτική και χρησιμοποιείται αν είναι επιθυμητός ο ορισμός διαφορετικών ομάδων χρωμάτων για τις διαφορετικές καταστάσεις στις οποίες μπορεί να βρίσκεται ένα γραφικό στοιχείο, που είναι «ενεργό» (Active), «ανενεργό» (Inactive) και «απενεργοποιημένο» (Disabled). Στην πρώτη κατάσταση βρίσκεται ένα παράθυρο όταν βρίσκεται υπό τον έλεγχο του πληκτρολογίου, στη δεύτερη όταν δεν συμβαίνει αυτό, ενώ η τρίτη είναι μια ειδική κατάσταση στην οποία μπορούν να βρεθούν κάποια γραφικά στοιχεία (όχι παράθυρα).

Η παράμετρος acr αναφέρεται στο ρόλο για τον οποίο θα χρησιμοποιηθούν τα χρώματα. Αυτοί οι ρόλοι είναι τύποι της QPalette, οι πιο συνηθισμένοι από τους

οποίους είναι ο τύπος Window, που αναφέρεται στο χρώμα φόντου (παρασκηνίου) και ο WindowText, που αναφέρεται στο γενικό χρώμα κειμένου.

Η παράμετρος acolor χρησιμοποιείται για το χρώμα που θέλουμε να ορίσουμε και ο τύπος της ορίζεται από την κλάση QColor.

Εφόσον δημιουργηθεί μια υπόσταση της QPalette και οριστούν τα χρώματά της, η χρήση της από ένα γραφικό στοιχείο γίνεται με τη μέθοδο setPalette της QApplication:

setPalette(QPalette palette, str className = None),

όπου palette είναι το όνομα του αντικειμένου της QPalette το οποίο θέλουμε να χρησιμοποιήσουμε.

Αξίζει να σημειωθεί ότι για να έχει αποτέλεσμα η χρήση της QPalette, πρέπει να κληθεί από το αντικείμενο η μέθοδος setAutoFillBackground(self, bool enabled), με την παράμετρο enabled να παίρνει την τιμή True. Η μέθοδος αυτή ανήκει στην QWidget (από την οποία κληρονομούν όλες).

4.6 Άλλα Γραφικά Στοιχειά που Χρησιμοποιήθηκαν

Εκτός από τις λειτουργίες που αναφέρθηκαν παραπάνω, χρησιμοποιήθηκαν κάποια γραφικά στοιχεία για την ενημέρωση του χρήστη σχετικά με το πρόγραμμα. Ένα κείμενο που προορίζεται για να αναγνωστεί από το χρήστη μπαίνει σε ένα γραφικό στοιχείο QLabel, το οποίο, κατά τη δημιουργία του, λαμβάνει ως όρισμα το κείμενο που θα εμφανιστεί. Χρησιμοποιήθηκε ένα τέτοιο αντικείμενο στο πάνω μέρος του παραθύρου, που ενημερώνει το χρήστη για τις δυνατότητες που παρέχονται στο πρόγραμμα, και ένα για κάθε γραφικό στοιχείο, που διευκρινίζει τη χρήση του. Επίσης, προστέθηκε άλλο ένα στο τέλος του προγράμματος που χρησιμοποιείται για την ενημέρωση του χρήστη για την τρέχουσα κατάσταση του προγράμματος.

4.7 Διάταξη των Γραφικών Στοιχείων

Για τη διάταξη των γραφικών στοιχείων, ως βασική επιλέχθηκε η κατακόρυφη, με τη χρήση της QVBoxLayout, με τη λογική ότι κάθε ξεχωριστή λειτουργία θα έπρεπε να βρίσκεται κάτω από την προηγούμενη. Αυτή όμως δεν είναι η μοναδική που χρησιμοποιήθηκε, καθώς για τις δύο πρώτες λειτουργίες προστέθηκε η οριζόντια διάταξη (QHBoxLayout). Ως πρώτο στοιχείο τοποθετήθηκε η περιγραφή του κυρίως γραφικού στοιχείου και δίπλα του τοποθετήθηκε αυτό. Για την τρίτη λειτουργία, χρησιμοποιήθηκαν συνδυαστικά η οριζόντια διάταξη και η διάταξη σε πλέγμα (QGridLayout). Με τη διάταξη πλέγματος επιτεύχθηκε η σωστή

στοίχιση των πλαισίων απεικόνισης της κατάστασης των αισθητήρων με τις λεζάντες τους (τα στοιχεία QLabel), με τα οποία περιγράφεται σε ποιον αισθητήρα αναφέρεται κάθε ένα από αυτά. Χρησιμοποιήθηκαν δύο τέτοια πλέγματα, ένα για τους αισθητήρες κεφαλιού και ένα για αυτούς που βρίσκονται στα πόδια.

Η τοποθέτηση των στοιχείων σε μια διάταξη QGridLayout, γίνεται όπως και για τις άλλες διατάξεις, με τη διαφορά ότι η μέθοδος addWidget() αυτής της κλάσης χρησιμοποιεί άλλα ορίσματα:

QGridLayout.addWidget(self, QWidget, int row, int column, Qt.Alignment alignment = 0)

Τα ορίσματα row και column δείχνουν τον αριθμό της σειράς και της στήλης αντίστοιχα στις οποίες θα τοποθετηθεί το γραφικό στοιχείο.

Η σωστή τοποθέτηση των λεζαντών πάνω από κάθε πλαίσιο απεικόνισης της κατάστασης των αισθητήρων, με τη διατήρηση μιας σταθερής και κυρίως μικρής απόστασης μεταξύ τους, δεν ήταν τόσο εύκολη υπόθεση και χρειάστηκε αρκετές δοκιμές. Επιτεύχθηκε με τον ορισμό σταθερού μήκους για τη διάταξη (με την κλήση της setFixedHeight()), σταθερού μεγέθους για τα πλαίσια απεικόνισης των αισθητήρων (με τη setFixedSize()), αλλά κυρίως με τη μέθοδο setAlignment() που κλήθηκε από κάθε ένα από τα αντικείμενα QLabel που χρησιμοποιήθηκαν για την τοποθέτηση των λεζαντών. Αυτή καθορίζει τη στοίχιση του κειμένου μέσα σε ένα QLabel:

setAlignment(self, Qt.Alignment),

όπου Qt.Alignment είναι ένα αντικείμενο που καθορίζει την οριζόντια στοίχιση, την κάθετη ή και το συνδυασμό τους. Για οριζόντια στοίχιση υπάρχουν οι τιμές Qt.AlignLeft, Qt.AlignRight, Qt.AlignCenter και Qt.AlignJustify και για κάθετη οι Q.AlignTop, Qt.AlignBottom, Qt.AlignVCenter.

Για τις λεζάντες χρησιμοποιήθηκε κεντρική οριζόντια στοίχιση και τοποθέτηση του κειμένου στο κάτω μέρος του QLabel (Qt.AlignCenter | Qt.AlignBottom). Το τελευταίο είναι απαραίτητο για τη μείωση της απόστασης κάθε λεζάντας από το αντίστοιχό QFrame.

4.8 Εκτέλεση και Αποτελέσματα της Εφαρμογής

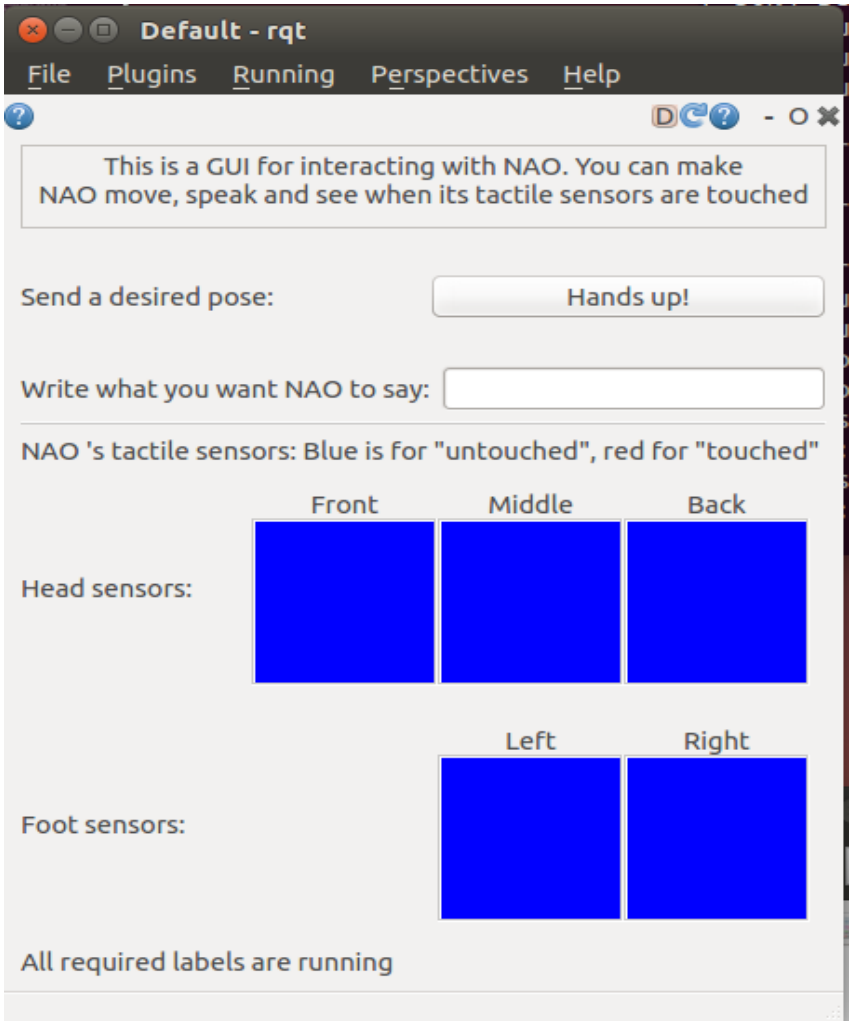
Για την εκτέλεση του plugin, εφόσον έχουμε τοποθετήσει το πακέτο στο οποίο περιέχεται η εφαρμογή (με όνομα rqt_nao) σε κάποιο χώρο εργασίας, ανοίγουμε το rqt. Το rqt είναι ειδική περίπτωση και δεν ανοίγει με το εργαλείο rosrun. Το ανοίγουμε απλώς με «rqt».

Επίσης, πρέπει να τρέχουν παράλληλα οι υπόλοιποι κόμβοι. Αυτό δεν γίνεται με την εντολή rosrun, διότι πρέπει να περαστούν ως ορίσματα σε αυτούς η IP του NAO (δίνεται από αυτό) και η roscore_ip, η οποία είναι η broadcast address. Αυτό γίνεται με:

roslaunch <όνομα πακέτου> <όνομα launch file> nao_ip:=<…> roscore_ip:=<…>,

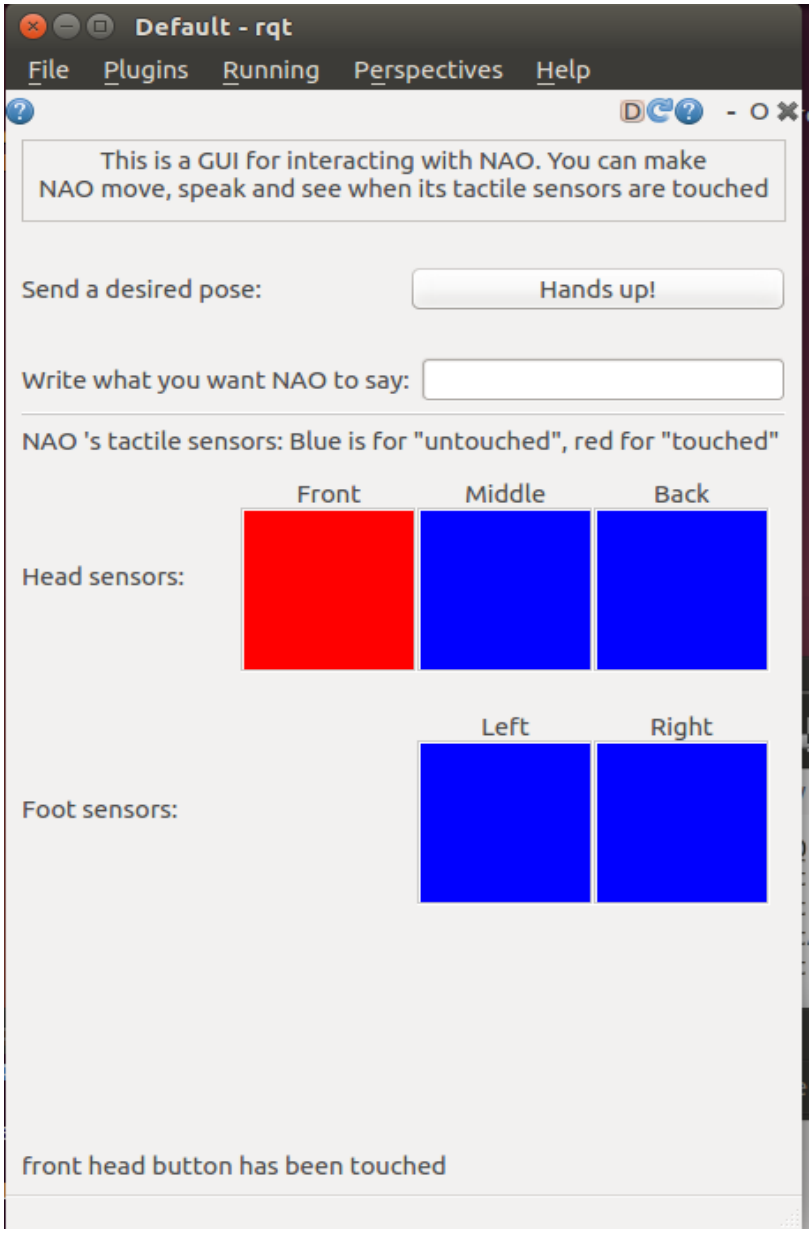
για κάθε έναν από τους κόμβους αυτούς.

Στο μενού Plugin, μπορεί κανείς να βρει όλα τα plugin που είναι διαθέσιμα. Ανάμεσά τους και το «my_module». Όμως, δεν θα το βρει με το όνομα αυτό, αλλά ως «NAO Plugin»: Είναι το όνομα που δώσαμε στην ετικέτα <label> στο αρχείο plugin.xml. Εφόσον το ανοίξουμε εμφανίζεται η εφαρμογή ενσωματωμένη στο rqt, όπως φαίνεται στην Εικόνα 19.



Εικόνα 19 - Το NAO Plugin

Παρακάτω, βλέπουμε την περίπτωση όπου ο χρήστης κρατάει πατημένο τον αισθητήρα στο μπροστινό μέρος του κεφαλιού του ρομπότ. Το χρώμα του αντίστοιχου πλαισίου γίνεται κόκκινο, ενώ η ενέργεια περιγράφεται και λεκτικά στο κάτω τμήμα της εφαρμογής.



Εικόνα 20 - Χρήση της λειτουργίας παρακολούθησης των αισθητήρων

Κεφάλαιο 5
Συμπεράσματα και Μελλοντικές Επεκτάσεις

5.1 Συμπεράσματα

Ολοκληρώνοντας τη συγγραφή αυτού του κειμένου, θα ήθελα να μοιραστώ με τον αναγνώστη τα συμπεράσματά μου από την ενασχόληση με το ROS και την εμπειρία της χρήσης αυτής της πλατφόρμας για την υλοποίηση της εφαρμογής.

Αρχικά θα ήθελα να πω ότι η ενασχόλησή μου με το ROS ήταν πολύ εποικοδομητική και νοιώθω ότι αποκόμισα οφέλη και βελτιώθηκα μέσα από αυτή. Το ROS δίνει μια άλλη διάσταση στην ανάπτυξη εφαρμογών. Το βάρος πέφτει στην ενσωμάτωση κώδικα και όχι στο αλγοριθμικό κομμάτι. Αυτό προσωπικά συνεπαγόταν την ανάγκη για την εκμάθηση ενός καινούριου τρόπου σκέψης, κάτι αρκετά δύσκολο στην αρχή, αλλά τελικά πολύ ενδιαφέρον.

Η φιλοσοφία της ενσωμάτωσης κώδικα, καθώς και η υλοποίηση κώδικα μέσα σε συγκεκριμένα πλαίσια, όπως αυτά που προσφέρει για παράδειγμα το rqt, προσφέρει το σημαντικό πλεονέκτημα ότι δεν χρειάζεται «να ξαναανακαλυφθεί ο τροχός». Ωστόσο, το κόστος της δεν είναι αμελητέο, καθώς απαιτείται η ανάγνωση εγγράφων τεκμηρίωσης και κώδικα άλλων ανθρώπων. Φυσικά, με την απόκτηση εμπειρίας αυτό είναι κάτι που γίνεται πιο εύκολο στην πορεία. Στην αρχή όμως, απαιτείται σίγουρα ένα διάστημα προσαρμογής.

Σε ότι αφορά τα πρακτικά προβλήματα που αντιμετωπίστηκαν, κάποια από αυτά είναι αναπόφευκτα και προέρχονται από τη φύση του ROS: Αυτή η πλατφόρμα είναι σχεδιασμένη να χρησιμοποιεί και να συνεργάζεται με άλλες βιβλιοθήκες ή άλλες πλατφόρμες και αυτό φυσικά είναι κάτι παραπάνω από επιθυμητό, όμως στη συνηθισμένη (και επίσης επιθυμητή) πρακτική αυτές οι συνεργαζόμενες μονάδες να αναβαθμίζονται, προκύπτουν κάποια θέματα συμβατότητας, παρόλο που στις περισσότερες περιπτώσεις οι εξελίξεις παρακολουθούνται και αποτυπώνονται σε αλλαγές στα πακέτα του ROS. Για παράδειγμα, ο κόμβος pose_controller.py, που χρησιμοποιήθηκε για τη λειτουργία της αποστολής κίνησης στο NAO, χρειάστηκε να αλλάξει σε κάποια σημεία, λόγω αλλαγών σε πιο πρόσφατες εκδόσεις του NAOqi. Γενικά, μια λύση σε αυτό είναι η χρήση παλαιότερων εκδόσεων όπου αυτό είναι εφικτό, μέχρι να υπάρχει πιο σωστή ενσωμάτωση στο ROS.

Ένα παρόμοιο πρόβλημα που ίσως προκύψει σε κάποιον προέρχεται ως παρενέργεια από την πολύ δραστήρια κοινότητα του ROS. Η ανανέωση κάποιων πακέτων ή στοιχείων αυτών μπορεί να προκαλέσει προβλήματα (προσωρινής) συμβατότητας με άλλα πακέτα ή στοιχεία. Επίσης, είναι αδύνατο οι αλλαγές να αποτυπώνονται πάντα γρήγορα και αποτελεσματικά στις σελίδες τεκμηρίωσης. Ωστόσο, σε γενικές γραμμές το επίπεδο της τεκμηρίωσης είναι καλό (και σε κάποιες πολύ καλό). Τέλος, η ύπαρξη συχνά πολλών πακέτων και εφαρμογών για τον ίδιο ή για παρεμφερείς σκοπούς, προκαλεί κάποιες φορές σύγχυση.

Σε ότι αφορά τα συνοδευτικά αρχεία που χρειάστηκε να υπάρχουν στο πακέτο που δημιουργήθηκε, το θέμα δεν καλύπτεται επαρκώς στις σελίδες τεκμηρίωσης, με την έννοια ότι οι πληροφορίες που απαιτήθηκαν δεν υπάρχουν συγκεντρωμένα κάπου. Βέβαια αυτό είναι και θέμα πολυπλοκότητας της συγκεκριμένης περίπτωσης. Ωστόσο, σε αυτές τις περιπτώσεις, αλλά και γενικότερα, η σελίδα των ερωτήσεων (ros.answers.org) ήταν πολύ βοηθητική.

5.2 Μελλοντικές Επεκτάσεις

Οι πολλές δυνατότητες του NAO, καθώς και η υπάρχουσα υποδομή επικοινωνίας με αυτό στο ROS, προσφέρουν πολλές δυνατότητες για βελτίωση της εφαρμογής. Σε αυτό, βοηθάει και το γεγονός ότι η επέκταση μιας εφαρμογής στην οποία χρησιμοποιήθηκε η Qt είναι εύκολη.

Σε πρώτη φάση, σχεδιάζεται να δημιουργηθούν περισσότερες κινήσεις. Επίσης, θα ήταν ενδιαφέρουσα η προσθήκη γραφικού στοιχείου που θα επιτρέπει την παρακολούθηση δεδομένων από τις κάμερες του ρομπότ.

Βιβλιογραφία

Βιβλιογραφια

[1] http://www.ros.org/history/

[2] https://www.youtube.com/watch?v=ueAByx7zQrg (σελ.7)

[3] www.ros.org

[4] Pablo Iñigo-Blasco, Fernando Diaz-del-Rio, Ma Carmen Romero-Ternero, Daniel Cagigas-Muñiz , Saturnino Vicente-Diaz, «Robotics software frameworks for multi-agent robotic systems development», Robotics and Autonomous Systems, Volume 60, Issue 6, June 2012, Pages 803–821

[5] http://wiki.ros.org/ROS/Concepts

[6] http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment

[7] http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29

[8] https://github.com/ros/common_tutorials/tree/hydro-devel/actionlib_tutorials

[9] https://www.aldebaran.com/en/humanoid-robot/nao-robot-working

[10] http://wiki.ros.org/rqt/Tutorials/Create%20your%20new%20rqt%20plugin

[11] doc.aldebaran.com/2-1/

[12] doc.aldebaran.com/2.1/family/robots/joints_robot.html

[13] answers.ros.org

[14] pyqt.sourceforge.net/Docs/PyQt4/qpalette.html

[15] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, Andrew Ng,“ROS: an open-source Robot Operating System”, p 2

Παράρτημα 1

Σε αυτό το παράρτημα, δείχνονται τα βήματα της εγκατάστασης των πακέτων του ROS που παρέχουν τη διασύνδεση με το NAOqi.

Αρχικά, χρειάζεται να γίνει εγκατάσταση κάποιων άλλα πακέτα του ROS, ως προαπαιτούμενα, με την παρακάτω εντολή:

sudo apt-get install ros-indigo-driver-base ros-indigo-move-base-msgs ros-indigo-octomap ros-indigo-octomap-msgs ros-indigo-humanoid-msgs ros-indigo-humanoid-nav-msgs ros-indigo-camera-info-manager ros-indigo-camera-info-manager-py

Στη συνέχεια πραγματοποιείται η εγκατάσταση των πακέτων. Στον υποκατάλογο src ενός χώρου εργασίας (workspace) δημιουργείται ένα κενό αρχείο με κατάληξη .rosinstall στο οποίο προστέθηκαν οι παρακάτω γραμμές:

- git: {local-name: naoqi_bridge, uri: "https://github.com/ros-naoqi/naoqi_bridge.git", version: master}

- git: {local-name: nao_robot, uri: "https://github.com/ros-naoqi/nao_robot.git", version: master}

- git: {local-name: nao_extras, uri: "https://github.com/ros-naoqi/nao_extras.git", version: master}

- setup-file: {local-name: ../devel/setup.bash }

Έπειτα, κατεβάζονται τα παραπάνω πακέτα από το github, με την εντολή:

$ wstool update

Τέλος, ο χώρος εργασίας χτίζεται και εγκαθίσταται, με τις εντολές:

$ cd ..

$ catkin_make

$ source devel/setup.bash