Robotyka Mobilna
Laboratorium

Narzędzia wykorzystywane w programowaniu robotów mobilnych

Opracowanie:
Łukasz Chechliński

Aktualizacja:
Daniel Koguciuk

Źródło:
www.ros.org

Wstęp

Celem niniejszego wstępu jest zapoznanie studentów z narzędziami wykorzystywanymi w dalszej części laboratorium, oraz ogólnie w robotyce mobilnej. Wprowadzenie to zostało wydzielone ze względu na trudności pojawiające się wśród studentów podczas realizacji ćwiczeń na robotach mobilnych. Aby uniknąć poświęcania większości czasu na rozwiązywanie problemów nie będących sednem robotyki wszelkie kwestie konfiguracyjne zostały przez prowadzących laboratoria maksymalnie uproszczone.

Niniejsza instrukcja opiera się w głównej mierze na materiałach ze strony *www.ros.org*. Przygotowanie studentów do laboratorium sprawdzone zostanie za pomocą testu, którego zaliczenie będzie konieczne do zaliczenia laboratorium. Przykładowe pytanie znajdują się na końcu instrukcji.

ROS – co to jest?

ROS (*ang. Robot Operating System*) jest elastycznym framework'iem do pisania oprogramowania dla robotów. Jest to zbiór narzędzi, bibliotek oraz sposobów programowania mający na celu uproszczenie zadania tworzenia złożonego i i solidnego oprogramowania działającego na różnych sprzętowych platformach robotycznych.

Dlaczego ROS jest nam w ogóle potrzebny? Ponieważ pisanie naprawdę solidnego i przenośnego oprogramowania dla robotów jest **trudne**. Z punktu widzenia robota problemy, które wydają się być trywialne dla ludzi wymagają do rozwiązania całego zestawu różnorodnych algorytmów. Rozwiązanie wszystkich tych problemów od zera przez pojedynczą osobę, laboratorium czy instytucję nie wydaje się być możliwe.

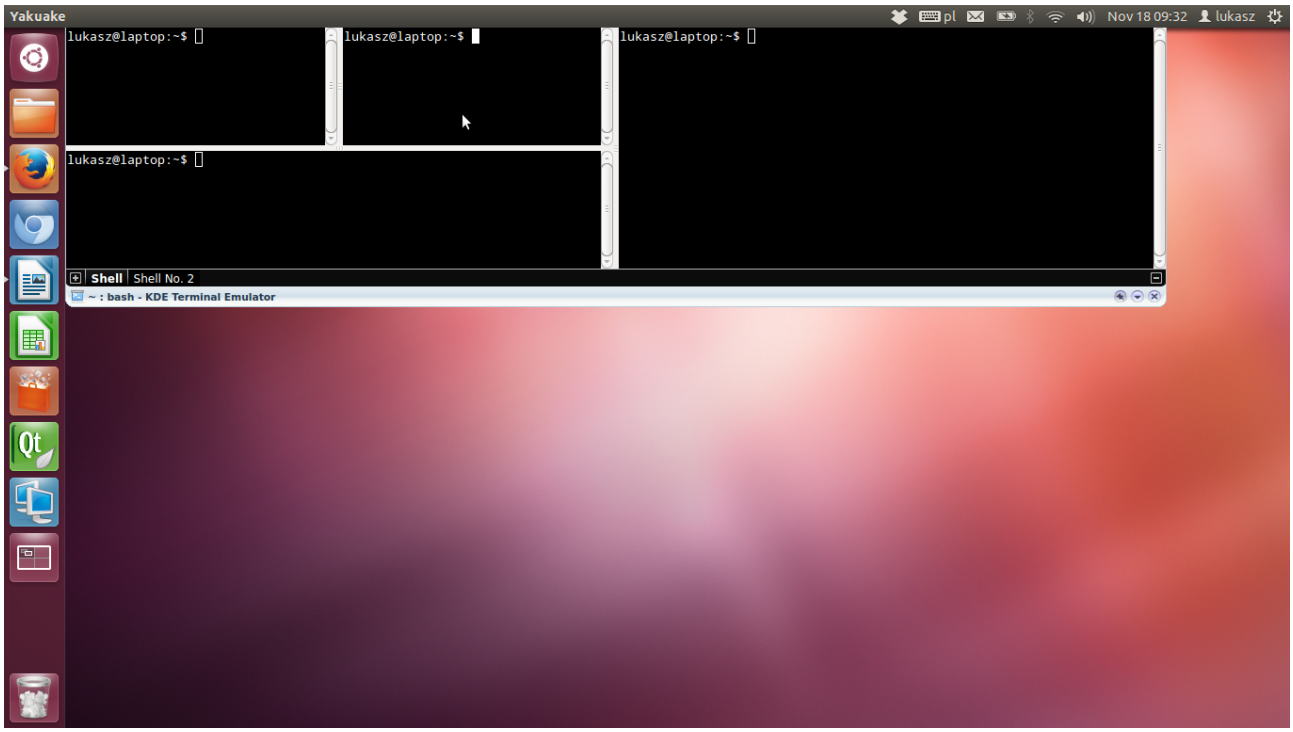
W związku z tym ROS został opracowany tak, aby możliwie ułatwiać współpracę w środowisku deweloperów oprogramowania dla robotów mobilnych. Dla przykładu, jedno laboratorium może specjalizować się w mapowaniu wnętrz budynków i opracować światowej klasy system budowania map. Inna grupa może składać się z ekspertów wykorzystujących te mapy do nawigacji, a jeszcze inna może opracować algorytm widzenia maszynowego dobrze radzący sobie z rozpoznawaniem małych obiektów w warunkach przysłaniania przez inne obiekty. ROS został opracowany właśnie z myślą o takich grupach, aby mogły współpracować i wzajemnie wykorzystywać swoje osiągnięcia.

Linux – przypomnienie podstaw

Systemem operacyjnym komputerów, na których zainstalowany jest ROS, jest najczęściej Ubuntu. Możliwe jest też skompilowanie ROSa na innych dystrybucjach Linuxa, ale nie jest to zalecane bez wyraźnej potrzeby. W tej części instrukcji nastąpi krótkie przypomnienie podstaw wykorzystania konsoli w systemach UNIXowych, które omawiane były na przedmiocie *Wprowadzenie do Technik Komputerowych*. Tę część instrukcji oparto na materiałach ze strony http://www.ee.surrey.ac.uk/Teaching/Unix/ .

Wykorzystywanie graficznego interfejsu użytkownika (*GUI*) oferowanego przez system Ubuntu nie stanowi większego problemu dla przeciętnego użytkownika komputerów. Do rozwijania oprogramowania robotów potrzebna jest jednak znajomość powłoki tekstowej. Dostęp do niej można uzyskać nie wyłączając całości powłoki graficznej, a jedynie uruchamiając odpowiedni terminal. W trakcie trwania laboratorium wykorzystywany będzie terminal ***yakuake***, który uruchamia się automatycznie po starcie systemu, pozostaje jednak schowany. Jego rozwinięcie (lub ponowne schowanie) następuje po naciśnięciu klawisza ***F12***.

yakuake wyświetlany jest na górze ekranu. Umożliwia obsługę wielu kart (pasek na dole) oraz podziały okna w pionie (***Ctrl+Shift+(***) oraz poziomie (***Ctrl+Shift+)***). Przykładowy podział przedstawia poniższy zrzut ekranu:



Do poruszania się w strukturze plików wykorzystywane są następujące komendy:

`ls`	Wyświetla listę plików i katalogów w obecnym folderze
`ls -a`	Wyświetla listę wszystkich (w tym ukrytych) plików i katalogów w obecnym folderze
`mkdir nowy_katalog`	Tworzy katalog o nazwie *nowy_katalog* w katalogu bieżącym
`cd`	Change directory – zmiana katalogu bieżącego. Przy braku argumentów oznacza powrót do katalogu głównego.
`cd ~`	Przejście do katalogu głównego

`cd ./Dokumenty`	Przejście do katalogu *Dokumenty* (o ile taki katalog znajduje się w katalogu bieżącym
`cd ..`	Przejście do katalogu nadrzędnego
`pwd`	Print working directory – wypisanie pełnej ścieżki bieżącego katalogu roboczego.

W terminalu można również uruchamiać programy posiadające interfejs graficzny. W ten sposób uruchamiana będzie również większość narzędzi wykorzystywanych przy pracy z robotami mobilnymi.

ROS – katalog roboczy

Programy, które będziemy pisać w ROSie znajdować się muszą w odpowiednio przygotowanym katalogu roboczym. Jego konfiguracja nie będzie elementem tego laboratorium. Katalog ten będzie nazywał się *rm_ws* i będzie znajdował się w katalogu domowym komputera. Możemy przejść do niego wpisując polecenie

```
roscd
```

Analogicznie jak polecenie `cd`, które przenosi nas do katalogu domowego Linuxa. W katalogu tym znajdują się trzy inne katalogi:

`src` – zawiera pliki źródłowe (kod C++ i inne pliki tekstowe) tworzonego oprogramowania

`build` – zawiera pliki generowane podczas kompilowania kodu

`devel` – zawiera pliki wykonywalne oraz inne „produkty końcowe” procesu kompilacji

Omawiany katalog zawiera tę część oprogramowania dostępną w systemie ROS, którą rozwijamy samodzielnie. Pozostałe programy znajdują się w katalogu instalacyjnym (czyli typowo */opt/ros/indigo*).

Poruszając się po katalogach z wykorzystaniem narzędzia *roscd* nie musimy wiedzieć, w jakim katalogu znajduje się dana paczka. Jeśli chcemy np. przejść w terminalu do paczki *roscpp*, to wpisujemy komendę:

```
roscd roscpp
```

Na przykładzie tego polecenia możemy też zobaczyć, jak działa w ROSie autouzupełnianie. Jeśli wpisując nazwę paczki wciśniemy klawisz TAB:

```
roscd roscpp_tut<<< teraz wciśnij klawisz TAB >>>
```

to nazwa paczki zostanie automatycznie dokończona:

```
roscd roscpp_tutorials/
```

Jeśli kilka paczek ma identyczny wpisany przez nas początek nazwy, to uzupełniana jest najpierw ich wspólna część nazwy, a po kolejnym wciśnięciu przycisku TAB wyświetlana jest lista paczek o danym początku nazwy. Mechanizm ten jest bardzo powszechny w ROSie (działa dla bardzo wielu poleceń nie tylko *roscd*), dlatego warto z niego korzystać – zwiększa to szybkość pracy, zwalnia z obowiązku pamiętania dokładnych nazw wszystkich używanych pakietów oraz zmniejsza ilość literówek.

Tworzenie nowych paczek ROSa

Oprogramowanie w systemie ROS podzielone jest na paczki. Paczka może zawierać kilka powiązanych ze sobą programów, np. paczka do obsługi kamer może zawierać program do komunikacji z kamerami USB, drugi do komunikacji z kamerami podłączanymi przez Ethernet, trzeci do kalibracji kamer.

W ROSie od wersji Groovy wykorzystywany jest system budowania paczek *catkin*. Nazwa ta będzie pojawiać się często podczas wykorzystywania ROSa. Na laboratoriach nie będą omawiane starsze wersje paczek ani sposoby migracji do nowej wersji.

Paczka typu *catkin* musi spełniać kilka warunków:

- paczka musi zawierać zgodny z systemem *catkin* opis w postaci pliku *package.xml*
- paczka musi zawierać plik *CMakeLists.txt*, opisujący sposób budowania paczki z wykorzystaniem narzędzia *cmake*, oczywiście w zakresie zgodnym z systemem *catkin*.
- w każdym folderze może być tylko jedna paczka (nie mogą występować paczki zagnieżdżone jak również kilka paczek nie może współdzielić tego samego folderu).

Najprostsza możliwa paczka może wyglądać następująco:

```
my_package/
  CMakeLists.txt
  package.xml
```

Zalecanym sposobem pracy z paczkami *catkin* jest zapisywanie ich w katalogu roboczym systemu *catkin*. Prosty katalog roboczy może wyglądać następująco:

```
workspace_folder/        -- WORKSPACE
  src/                   -- SOURCE SPACE
    CMakeLists.txt       -- 'Toplevel' CMake file, provided by catkin
    package_1/
      CMakeLists.txt     -- CMakeLists.txt file for package_1
      package.xml        -- Package manifest for package_1
    ...
    package_n/
      CMakeLists.txt     -- CMakeLists.txt file for package_n
      package.xml        -- Package manifest for package_n
```

Tworzenie własnych paczek (w szczególności struktura plików *CMakeLists.txt* oraz *package.xml*) nie jest częścią laboratorium robotyki mobilnej. Na ćwiczeniach wykorzystywane będą gotowe szkielety paczek.

Aby skompilować napisane paczki należy przejść do katalogu roboczego i wykonać w nim polecenie budowania:

```
cd ~/catkin_ws
catkin_make
```

Aby korzystać z paczek zlokalizowanych we własnym katalogu roboczym, należy wykonać w terminalu polecenie:

```
source ~/catkin_ws/devel/setup.bash
```

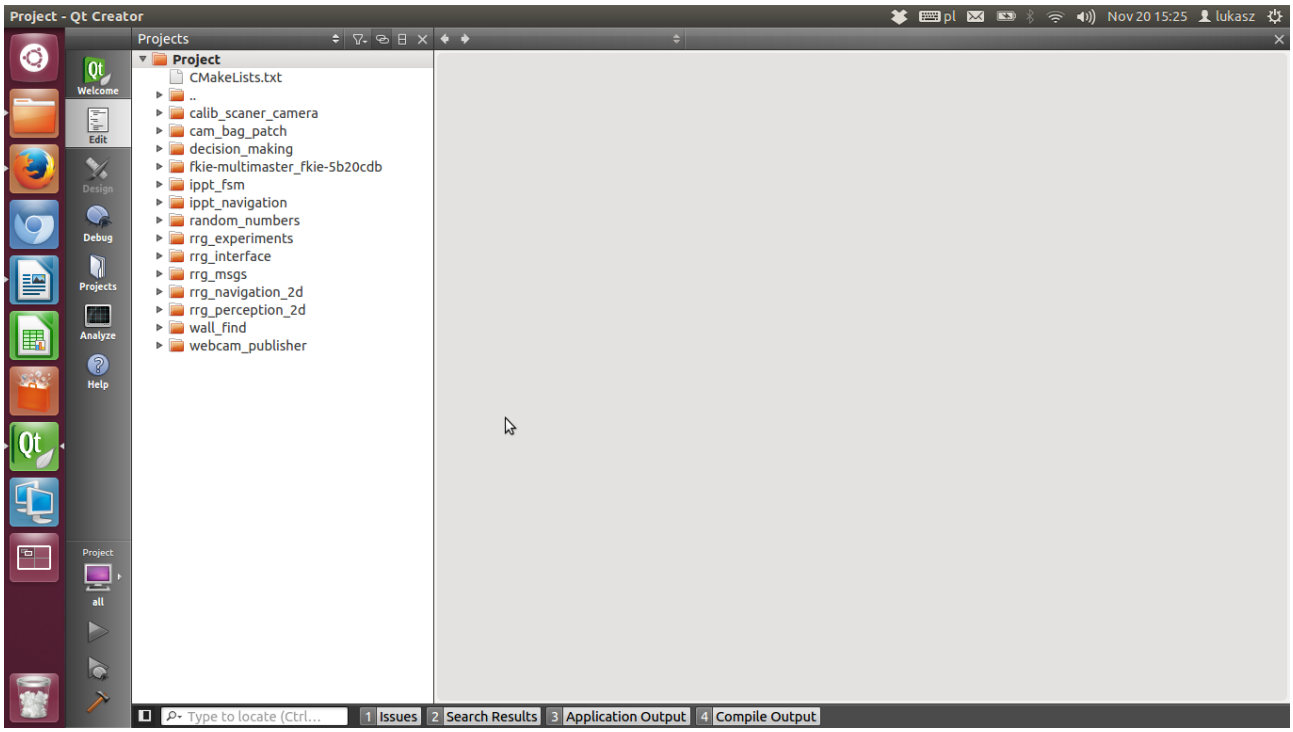
W trakcie laboratorium nie jest to jednak konieczne, ponieważ odpowiedni skrypt dba o jego automatyczne wykonywanie.

Warto zasygnalizować, że paczki mogą wymagać do uruchomienia lub kompilacji innych paczek *catkin*. Niespełnienie zależności będzie skutkowało wyświetleniem odpowiednich komunikatów. Polecenie *catkin_make* umożliwia budowanie programów z poziomu wiersza poleceń. Jest ono niezbędne przy pierwszej kompilacji katalogu roboczego, jednak przy dalszej pracy warto wykorzystywać środowisko deweloperskie zapewniające wygodna pracę z kodem. W naszym wypadku będzie to *Qt Creator*.

Przy użyciu *Qt Creator'* otwieramy jednocześnie wszystkie paczki znajdujące się w katalogu roboczym. Aby kompilacja zakończyła się sukcesem *Qt Creator* musi zostać uruchomiony za pomocą komendy w terminalu (przy uruchomieniu przez kliknięcie w ikonkę nie będą ustawione odpowiednie zmienne środowiskowe ROSa i kompilator nie znajdzie zależności). W tym celu należy przejść do podkatalogu *src* w katalogu roboczym i otworzyć główny plik *CMakeLists.txt* katalogu roboczego:

```
cd ~/catkin_ws/src
qtcreator CMakeLists.txt
```

Jeśli jest to pierwsze otwarcie tego pliku *CMakeLists.txt*, to *Qt Creator* zapyta o ścieżkę katalogu budowania. Należy ją koniecznie zmienić z domyślnej na ***catkin_ws/build***. Po uruchomieniu w lewej części okna będzie znajdowało się drzewo zawierające wszystkie rozwijane paczki:



ROS Node

W paczce ROSa może znajdować się kilka programów. Nazywane są one node-ami. W języku angielskim słowo ***node*** oznacza węzeł lub wierzchołek sieci, co dobrze oddaje naturę współpracy tych programów wewnątrz ROSa – każdy z nich stanowi węzeł powiązany kilkoma nitkami z innymi węzłami sieci. Jednak tłumaczenie tego terminu może wprowadzać niepotrzebny bałagan, w dalszej części instrukcji wykorzystywane więc będzie angielskie słowo node.
Zapoznajmy się więc z podstawowymi elementami tej sieci. Są to:

- **node**'y – pliki wykonywalne wykorzystujące ROSa do komunikacji z innymi node'ami,
- **wiadomości (*messages*)** – struktury danych wykorzystywane podczas wysyłania/subskrybowania informacji,
- **tematy (*topics*)** – kanały komunikacji. Każdy node może wysyłać wiadomości na wybrany kanał komunikacyjny lub pobierać je z niego,
- ***master*** – proces główny, pozwalający node'om na wzajemną komunikacje poprzez zarządzanie nazwami,
- ***rosout*** – wyjście ROSa, odpowiednik wykorzystywanego w C++ std::out oraz std::cerr,
- **Rdzeń ROSa *(roscore)*** – *Master* + *rosout* + (niewykorzystywany w czasie laboratorium) serwer parametrów

Node jest niczym więcej niż plikiem wykonywalnym (programem) znajdującym się w paczce ROSa. Node'y ROSa mogą wykorzystywać bibliotekę kliencką ROSa aby komunikować się z innymi node'ami. Node'y mogą publikować informacje na topic'ach lub subskrybować je. Node'y mogą również udostępniać lub wykorzystywać serwisy, jednak temat ten nie będzie poruszany na laboratorium.

Biblioteki klienckie ROSa umożliwiają wzajemna komunikacje pomiędzy node'ami napisanymi w różnych językach programowania. Dostępne są:

- roscpp – biblioteka w języku C++
- rospy – biblioteka w języku Python (nieużywana w trakcie laboratorium)

Rdzeń ROSa jest pierwszą rzeczą, jaką należy uruchomić przy pracy z ROSem. Służy do tego polecenie:

```
roscore
```

Po jego uruchomieniu w konsoli zostanie wyświetlony tekst zbliżony do poniższego:

```
...   logging   to   ~/.ros/log/9cf88ce4-b14d-11df-8a75-00251148e8cf/roslaunch-
machine_name-13039.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://machine_name:33919/
ros_comm version 1.4.7

SUMMARY
========

PARAMETERS
 * /rosversion
 * /rosdistro
```

```
NODES

auto-starting new master
process[master]: started with pid [13054]
ROS_MASTER_URI=http://machine_name:11311/

setting /run_id to 9cf88ce4-b14d-11df-8a75-00251148e8cf
process[rosout-1]: started with pid [13067]
started core service [/rosout]
```

Po uruchomieniu *roscore'a* możemy przyjrzeć się prostym narzędziom do analizy stanu systemu ROS. Aby np. zobaczyć listę uruchomionych node'ów należy w terminalu wpisać polecenie:

```
rosnode list
```

W tym wypadku wyjście będzie wyglądało następująco:

```
/rosout
```

Widać, że uruchomiony jest tylko jeden node: *rosout*. Aby uzyskać więcej informacji o uruchomionym procesie, należy wykorzystać polecenie *rosnode info*. Na przykład:

```
rosnode info /rosout
```

Wyświetli ono podstawowe informacje, które mogą być przydatne podczas debugowania, takie jak subskrybowane oraz publikowane *topic'i*. Przykładowe wyjście wygląda następująco:

```
--------------------------------------------------------------------------------
Node [/rosout]
Publications:
 * /rosout_agg [rosgraph_msgs/Log]

Subscriptions:
 * /rosout [unknown type]

Services:
 * /rosout/set_logger_level
 * /rosout/get_loggers

contacting node http://machine_name:54614/ ...
Pid: 5092
```

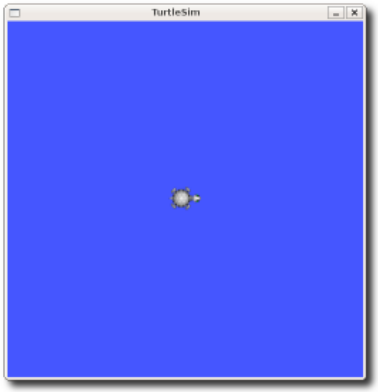
Sam */rosout* to jednak za mało, aby robot działał. Przejdźmy więc do uruchamiania kolejnych node'ów. Służy do tego polecenie *rosrun*. Pozwala ono na uruchomienie node'a znajdującego się w danej paczce bez dokładnej wiedzy o jej lokalizacji. Ma ono następującą składnię:

```
rosrun [nazwa_paczki] [nazwa_node'a]
```

Przy wpisywaniu obu nazw można skorzystać z autouzupełniania (klawisz TAB). Przykładowo uruchomimy teraz ćwiczeniowy robotyczny symulator żółwia, *TurtleSim*. Wpisujemy więc polecenie:

```
rosrun turtlesim turtlesim_node
```

Wyświetli się wtedy okno symulatora:



Jeśli teraz sprawdzimy listę node'ów poleceniem:

```
rosnode list
```

To zobaczymy na liście uruchomiony program:

```
/rosout
/turtlesim
```

ROS Topic

Topic'i są kanałami, po których komunikują się ze sobą node'y. Aby zrozumieć ich działanie uruchomimy drugi program, sterujący żółwiem widocznym w naszym symulatorze. W tym celu wpisujemy komendę:

```
rosrun turtlesim turtle_teleop_key
```

```
[ INFO] 1254264546.878445000: Started node [/teleop_turtle], pid [5528], bound
on [aqy], xmlrpc port [43918], tcpros port [55936], logging to
[~/ros/ros/log/teleop_turtle_5528.log], using [real] time
Reading from keyboard
---------------------------
Use arrow keys to move the turtle.
```

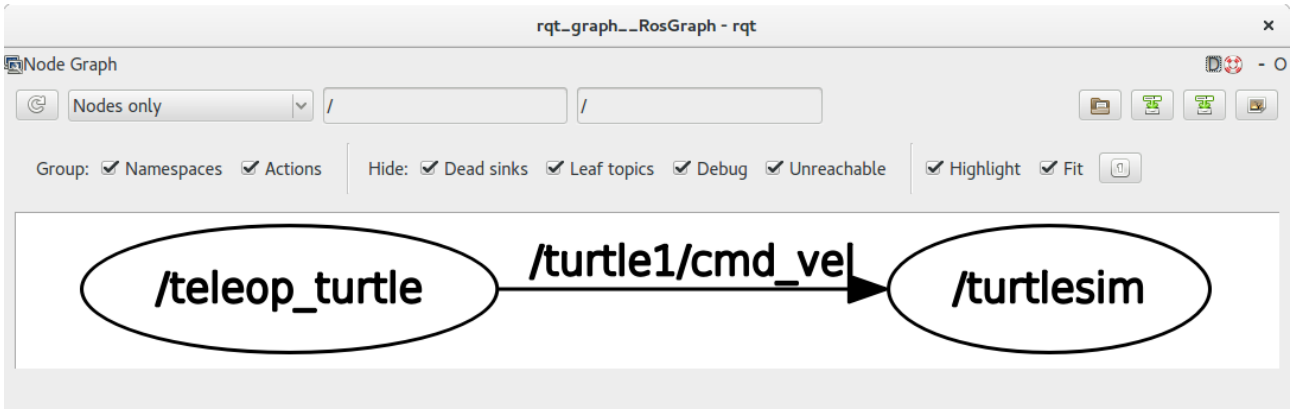
Teraz możemy sterować żółwiem za pomocą strzałek na klawiaturze (jeśli żółw się nie rusza, to należy się upewnić, że okno terminalu z programem *turtle_teleop_key* jest zaznaczone).

Widzimy więc, że komunikacja pomiędzy programami działa. Teraz przyjrzymy się dokładniej jej mechanizmom.

Node'y *turtlesim_node* oraz *turtle_teleop_key* komunikują się ze sobą poprzez topic ROSa. Node *turtle_teleop_key* publikuje komendy sterowania wydawane przez użytkownika, podczas gdy *turtlesim_node* subskrybuje je. Aby wygodnie przyjrzeć się sieci połączeń wykorzystamy narzędzie *rqt_graph*. Wizualizuje ono aktualną sieć połączeń pomiędzy node'ami. Uruchamiamy je poleceniem:

```
rosrun rqt_graph rqt_graph
```

Zobaczymy następujący widok:



Widzimy wyraźnie, że komunikacja odbywa się poprzez topic */turtle1/cmd_vel*. Node */teleop_turtle* publikuje wiadomości, a node */turtlesim* je odbiera, co pokazuje strzałka. Znamy więc już kanał i kierunek komunikacji, ciągle nie wiemy jednak jaka jest treść komunikatów. Aby się temu przyjrzeć wykorzystamy narzędzie *rostopic*.

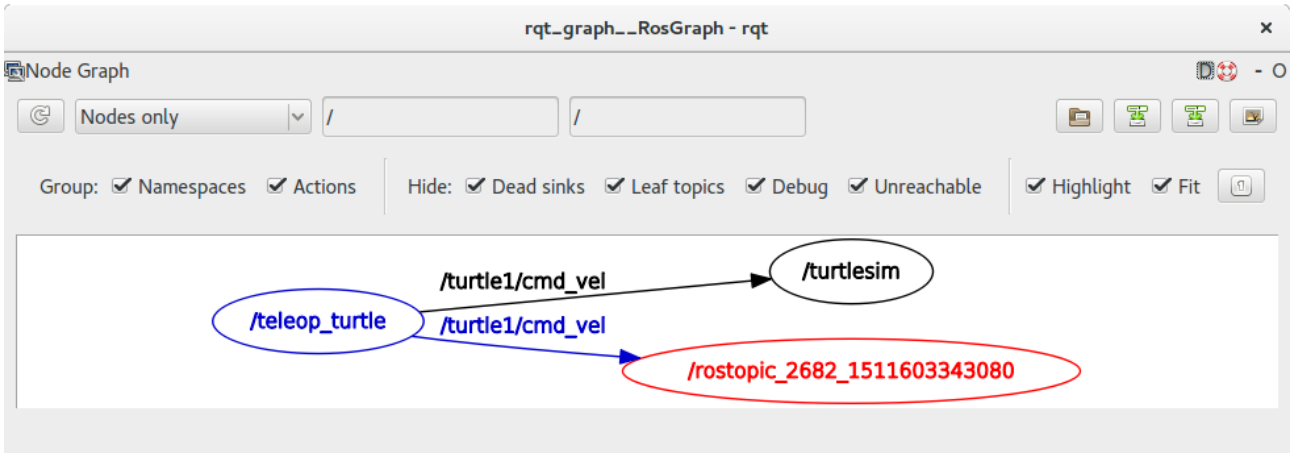
Narzędzie to ma kilka wariantów użycia. Zacznijmy od najprostszego, czyli od odsłuchania wiadomości publikowanych na topicu. W tym celu w terminalu wpisujemy polecenie:

```
rostopic echo /turtle1/cmd_vel
```

Po wpisaniu tego polecenia najprawdopodobniej nie wyświetla się żadne informacje, ponieważ nie sterujemy obecnie naszym żółwiem i sterowania nie są publikowane. Wystarczy jednak poruszyć żółwiem, aby wyświetlone zostały informacje zbliżone do poniższych:

```
linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
```

Spójrzmy teraz ponownie na widok połączeń w *rqt_graph*. Po wciśnięciu przycisku odświeżenia będzie on wyglądał następująco:



Widać, ze proces *rostopic echo*, zaznaczony tutaj na czerwono, również zasubskrybował topic */turtle1/cmd_vel*.

Narzędzie *rostopic* może służyć również do wyświetlenia listy wszystkich topiców aktywnych w systemie. Aby ja wyświetlić wpisujemy polecenie:

```
rostopic list
```

Komunikacja na topicach odbywa się poprzez wysyłanie wiadomości pomiędzy node'ami. Aby komunikacja mogła dojść do skutku node publikujący i subskrybujący muszą wykorzystywać ten sam typ wiadomości. Oznacza to, że typ topicu definiowany jest przez typ wiadomości na nim publikowanych. Może on być określony za pomocą polecenie *rostopic type* (lub bardziej dokładnego *rostopic info*). W naszym przykładzie możemy wpisać:

```
rostopic type /turtle1/cmd_vel
```

W efekcie otrzymamy typ wiadomości:

```
geometry_msgs/Twist
```

Jeszcze nie znamy tego typu wiadomości, więc chcielibyśmy poznać nie tylko jego nazwę, ale i strukturę, W tym celu wpisujemy:

```
rosmsg show geometry_msgs/Twist
```

I otrzymujemy strukturę wiadomości:

```
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

Jak widać typowa komenda prędkości w ROSie składa się z prędkości liniowej i kątowej, z których każda wyrażona jest poprzez jej składowe 3D. Oczywiście większość robotów mobilnych wykorzystuje tylko niektóre z nich. W trakcie tego laboratorium będą to składowa x prędkości liniowej oraz składowa z prędkości kątowej. Z powyższą wiedzą możemy przejść do napisania korzystających z ROSa programów w języku C + +.

Własny node publikujący i subskrybujący wiadomości

W tej części instrukcji omówimy napisanie kodu programu w języku C++ umożliwiającego publikowanie prostej wiadomości na utworzonym przez nas topicu. Założymy przy tym, że szkielet paczki został już utworzony i skupimy się jedynie na kodzie źródłowym.

Edytowana przez nas paczka ćwiczeniowa nazywać się będzie *beginner_tutorials* i będzie znajdować się w katalogu *~/catkin_ws/src/beginner_tutorials*. Jej struktura jest następująca:

```
beginner_tutorials/
  CMakeLists.txt
  package.xml
  src/
    talker.cpp
    listener.cpp
```

Plik *talker.cpp* zawiera kod node'a publikującego wiadomości. Jest on następujący:

```
#include "ros/ros.h"
#include "std_msgs/String.h"

#include <sstream>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "talker");
  ros::NodeHandle n;
  ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
  ros::Rate loop_rate(10);
  int count = 0;
```

```
  while (ros::ok())
  {
    std_msgs::String msg;

    std::stringstream ss;
    ss << "hello world " << count;
    msg.data = ss.str();
    ROS_INFO("%s", msg.data.c_str());

    chatter_pub.publish(msg);

    ros::spinOnce();

    loop_rate.sleep();
    ++count;
  }
  return 0;
}
```

Przejdźmy do wyjaśnienia kolejnych fragmentów tego kodu.

```
#include "ros/ros.h"
```

Jest to dyrektywa dołączająca podstawowy plik nagłówkowy ROSa, zawierający większość funkcji ogólnego przeznaczenia.

```
#include "std_msgs/String.h"
```

Dołącza plik nagłówkowy wiadomości ROSowej typu *String*, znajdującej się w paczce wiadomości standardowych *std_msgs*.

```
  ros::init(argc, argv, "talker");
```

W tej linii inicjalizujemy komunikację naszego node'a z ROSem. Nadajemy mu przy tym nazwę *talker*, która musi być unikalna w całym systemie robota.

```
  ros::NodeHandle n;
```

Tworzy uchwyt do node'a procesu. Za jego pomocą przypisujemy obiekty subskrybujące i publikujące do topiców.

```
  ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
```

W tej linii informujemy ROSa, że chcemy publikować wiadomości typu *<std_msgs::String>* na topicu */chatter*. Dzięki temu wszystkie node'y nasłuchujące topicu */chatter* będą nasłuchiwać również informacji publikowanych przez nasz node. Drugi argument to rozmiar kolejki wiadomości. W tym wypadku mówi on, że jeśli będziemy publikować wiadomości zbyt szybko, to w buforze zostanie zgromadzonych do 1000 wiadomości, nim zaczną być one pomijane – warto więc pamiętać o ustawieniu odpowiedniego rozmiaru bufora. Funkcja *advertise* zwraca obiekt typu *ros::Publisher*, który ma dwa zadania:

- umożliwia publikowanie wiadomości poprzez funkcję *publish*
- automatycznie informuje o końcu nadawania po wykonaniu jego destruktora.

```
  ros::Rate loop_rate(10);
```

Obiekt typu *ros::Rate* pozwala na określenie częstotliwości, z jaką wykonywać będzie się pętla główna naszego programu. Mierzy on czas jaki upłynął od zakończenia poprzedniego wywołania funkcji *Rate::sleep*, a następnie zatrzymuje wykonywanie programu na obliczony na tej podstawie czas. W tym wypadku chcemy, aby program działał z częstotliwością 10 Hz.

```
  while (ros::ok())
  {
```

Domyślnie *roscpp* umożliwia przerywanie działania programu poprzez kombinacje klawiszy *Ctrl+C*. Skutkuje to właśnie zwróceniem wartości *false* przez funkcję *ros::ok()*, co przerywa wykonywanie pętli głównej.

```
    std_msgs::String msg;

    std::stringstream ss;
    ss << "hello world " << count;
    msg.data = ss.str();
```

W tym fragmencie kodu deklarujemy zmienną typu *std_msgs::String*, a następnie wypełniamy jej pola danymi. W tym wypadku wiadomość ma tylko jedno pole, mianowicie *data* typu *string*. W dalszej części laboratorium będziemy wykorzystywać bardziej złożone wiadomości.

```
    ROS_INFO("%s", msg.data.c_str());
```

ROS_INFO i inne funkcje z tej dziedziny (*ROS_WARN*, *ROS_ERROR*) są odpowiednikami *std::cout*. W node'ach można wykorzystywać standardowe wyjście *std::cout*, ale jeśli interesuje nas np. dokładny czas wyświetlenia komunikatu czy też wypisywanie błędów kolorem czerwonym, a ostrzeżeń żółtym, to wygodniej jest użyć wersji ROSowej.

```
    chatter_pub.publish(msg);
```

W tej linii wykonuje się wysłanie wiadomości do wszystkich node'ów połączonych z topiciem na którym publikujemy.

```
    ros::spinOnce();
```

Wywołanie tej funkcji nie jest w wypadku tak prostego programu konieczne, gdyż odpowiada ona za wywołanie funkcji obsługujących wiadomości przychodzące, a powyższy program żadnych wiadomości przychodzących nie subskrybuje. Można jednak dodać ja do kodu na wypadek, gdybyśmy chcieli w przyszłości rozwijać nasz program oraz dla zachowania dobrych praktyk programowania w ROSie. Jeśli funkcja ta nic nie robi, to nie stanowi mierzalnego narzutu obliczeniowego.

```
    loop_rate.sleep();
```

Funkcje te wywołujemy, aby wstrzymać proces na czas potrzebny do uzyskania zadanej przez nas częstości działania, czyli 10 Hz.

A więc w skrócie działanie programu składa się z następujących etapów:

- inicjalizacja ROSa,
- poinformowanie *mastera*, że zamierzamy publikować wiadomości typu *std_msgs::String* na topicu */chatter*,
- publikowanie wiadomości na topicu */chatter* z częstotliwością 10 razy na sekundę.

Teraz pora napisać node, który będzie te wiadomości odbierał. Jego kod znajduje się w pliku *src/listener.cpp* i ma następującą treść:

```
#include "ros/ros.h"
#include "std_msgs/String.h"

void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
  ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "listener");

  ros::NodeHandle n;

  ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);

  ros::spin();

  return 0;
}
```

Omówmy teraz kolejne fragmenty tego kodu, omijając te wyjaśnione wcześniej.

```
void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
  ROS_INFO("I heard: [%s]", msg->data.c_str());
}
```

Jest to funkcja, która będzie wywołana gdy nowa wiadomość zostanie opublikowana na topicu */chatter*.

```
  ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
```

W tej linii informujemy ROSa, że chcemy subskrybować wiadomości pojawiające się na topicu */chatter*. Drugi argument to rozmiar bufora wiadomości – w tym wypadku, jeśli będziemy przetwarzać wiadomości zbyt wolno, to w buforze zostanie zakolejkowanych do 1000 wiadomości nim zostaną one stracone. Podany tutaj rozmiar bufora nie musi być identyczny jak ten, który podaliśmy pisząc node publikujący. Trzeci argument to nazwa funkcji, któr ma być wywoływana po otrzymaniu nowej wiadomości.

Funkcja *subscribe* zwraca obiekt typu *ros::Subscriber*, który wprawdzie nie jest jawnie wykorzystywany w żadnym innym miejscu kodu, ale nie może zostać zniszczony – wywołanie jego destruktora spowoduje zakończenie subskrybowania danych.

```
  ros::spin();
```

ros::spin() wchodzi w pętlę wywołującą funkcje obsługi przychodzących wiadomości tak szybko jak to możliwe. Nie należy jednak obawiać się jej stosowania, ponieważ nie obciąży ona CPU znacząco, jeśli ilość przychodzących wiadomości jest mała. Funkcja zakończy działanie, gdy *ros::ok()* zwróci wartość *false* (czyli np. po wciśnięciu kombinacji *Ctrl+C*).

Podsumujmy więc kolejne kroki działania programu:

- inicjalizacja ROSa,
- zasubskrybowanie topicu */chatter*,
- oczekiwanie na nadejście wiadomości,
- gdy wiadomość przyjdzie wywoływana jest funkcja *chatterCallback()*.

Przykładowe pytania na kartkówce

1. Jakim skrótem klawiszowym wyświetla się i chowa terminal yakuake?
2. Jakimi skrótami klawiszowymi dzieli się okno terminalu yakuake w pionie i w poziomie?
3. Jaką komendą wyświetla się listę plików i katalogów w katalogu bieżącym w systemie Linux?
4. Jaką komendą przechodzi się w systemie Linux do katalogu nadrzędnego?
5. Jaką komendą przechodzi się w systemie Linux do katalogu podrzędnego o danej nazwie?
6. Jaką komendą przechodzi się w systemie Linux do katalogu domowego?
7. Jaką komendą przechodzi się w systemie ROS do katalogu paczki o danej nazwie?
8. Jakie pliki muszą wchodzić w skład paczki typu catkin w systemie ROS?
9. Jak z poziomu terminala zbudować paczki znajdujące się w katalogu roboczym ~/catkin_ws?
10. Jak poprawnie otworzyć wszystkie paczki z katalogu roboczego ~/catkin_ws w programie Qt Creator?
11. Jak uruchomić rdzeń systemu ROS?
12. Jak wyświetlić w terminalu listę aktualnie uruchomionych node'ów?
13. Jak wyświetlić w terminalu szczegółowe informacje o danym nod'zie?
14. Jak uruchomić w terminalu node, znając jego nazwę oraz nazwę paczki w której się on znajduje?
15. Jak uruchomić narzędzie *rqt_graph*?
16. Jak w terminalu wyświetlić treść wiadomości publikowanych na topicu */nazwa_topicu*?
17. Jak wyświetlić w terminalu listę aktywnych topiców?
18. Jak sprawdzić w terminalu typ wiadomości publikowanych na topicu */nazwa_topicu*?
19. Jak wyświetlic w terminalu strukturę wiadomości typu *Twist* znajdującej się w paczce *geometry_msgs*?
20. Jaką dyrektywą dołączyć do pliku *.cpp plik nagłówkowy ROSa, zawierający większość funkcji ogólnego przeznaczenia.?
21. Jaką dyrektywą dołączyć do pliku *.cpp plik nagłówkowy zawierający strukturę wiadomości typu *std_msgs/String*?
22. Jak w języku C++ zadeklarować uchwyt do node'a?