Wprowadzenie do systemu operacyjnego ROS: symulator *turtlesim*[*]

Katarzyna Zadarnowska[†]
Laboratorium Robotyki
Wydział Elektroniki
Politechnika Wrocławska

1 Cel ćwiczenia

Celem ćwiczenia jest opanowanie podstaw programowania w systemie operacyjnym ROS.
W szczególności, studenci zaznajomią się z najważniejszymi poleceniami platformy programistycznej ROS, zrozumieją ideę przepływu informacji w systemie ROS, nauczą się tworzyć własne projekty oraz pisać, kompilować i uruchmiać proste węzły. Ćwiczenie stanowi wstęp do realizacji kolejnych ćwiczeń laboratoryjnych bazujących na systemie ROS [5, 6].

2 Wymagania wstępne

Przed przystąpieniem do realizacji ćwiczenia należy:

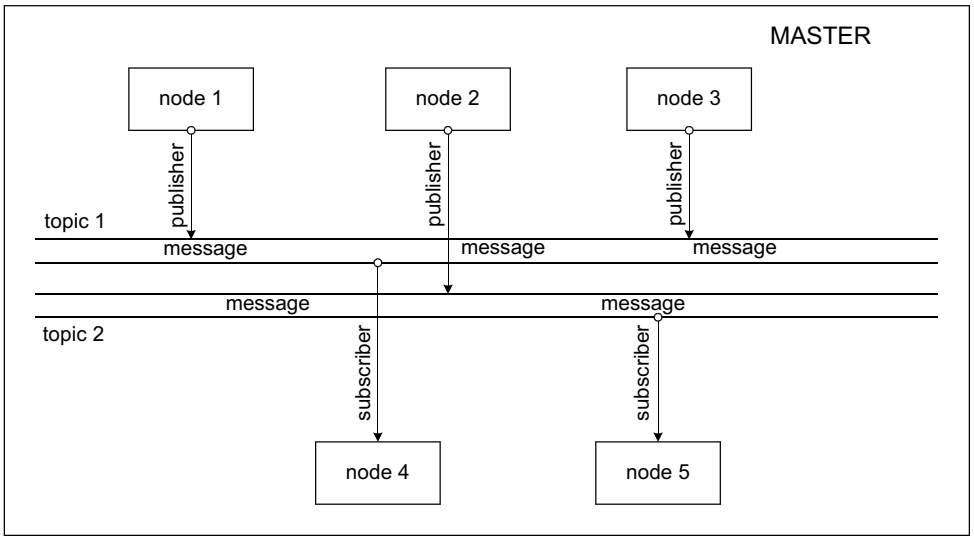
1. zapoznać się z Instrukcją bezpieczeństwa i higieny pracy w Laboratorium Robotów Autonomicznych L1.5 [1, 2],
2. znać podstawy programowania w językach C++,
3. zapoznać się z niniejszą instrukcją,
4. zapoznać się z zaproponowanymi w instrukcji rozdziałami pozycji [3] (rozdziały 3.2 - 3.4, 6.2 - 6.4, 8.3 - 8.4) i [4] (rozdział 2).

3 Wprowadzenie

ROS (Robot Operating System) to platforma programistyczna do tworzenia oprogramowania oraz sterowania robotów. Stanowi ona zbiór bibliotek pozwalających na sterowanie i symulację pracy robota. W szczególności, ROS zawiera podstawowe procesy systemowe obsługujące urządzenia sprzętowe robota, sterowanie niskopoziomowe, implementacje wykonywania typowych funkcji, komunikację międzywątkową oraz zarządzanie pakietami. Ideę środowiska ROS prezentuje rysunek 1.

[*]Ćwiczenie laboratoryjne przeznaczone do realizacji w ramach kursu Robotyka (2) – data ostatniej modyfikacji: 19 marca 2015

[†]Katedra Cybernetyki i Robotyki



Rysunek 1: Schemat projektu na platformie ROS

Oprogramowanie robotów tworzy się w postaci zbioru małych, działających współbieżnie i w dużej mierze niezależnych od siebie programów, nazywanych węzłami (ang. nodes). W celu ułatwienia procesu wymiany danych pomiędzy tymi węzłami w systemie ROS jest wykorzystywany węzeł pełniący funkcję węzła nadrzędnego (ROS Master), który dba o poprawność komunikacji zachodzącej między pozostałymi węzłami systemu. Węzeł ten uruchamia się wydając polecenie `roscore`. Węzły komunikują się między sobą poprzez wysyłanie wiadomości (ang. messages). Wiadomości zorganizowane są w tematach (ang. topic). Węzły, które chcą dzielić informacje (ang. publishers) publikują wiadomości w obrębie odpowiedniego tematu, natomiast węzły odbierające wiadomości, tzw. subskrybenci (ang. subscribers) subskrybują wiadomości. Węzeł nadrzędny powinien działać przez cały czas trwania pracy w systemie ROS, należy więc uruchmić go w oddzielnym terminalu (polecenie `roscore`) i pozostawić uruchomionym na czas pracy. Zatrzymanie węzła nadrzędnego następuje po wysyłaniu sygnał SIGINT (`Ctrl-C`).

3.1 Pakiety

Oprogramowanie systemu ROS stanowi zbiór pakietów. Pakiety z kolei są zbiorem plików realizujących określony cel. Pakiety posiadają swoją dokumentację (plik package.xml). Poniżej umieszczono listę przydatnych poleceń systemu ROS dotyczących pakietów:

- lista wszystkich dostępnych pakietów zdefiniowanych w systemie ROS
  `rospack list`
- znajdowanie katalogu zawierającego dany pakiet
  `rospack find` *`package-name`*
- przeglądanie zawartości katalogu danego pakietu
  `rosls` *`package-name`*
- przejście z bieżącego kaalogu do katalogu danego pakietu
  `roscd` *`package-name`*

3.2 Węzły

Węzły* to wykonywalne instancje programów systemu ROS. Wyróżniamy następujący zestaw poleceń dotyczących węzłów:

- tworzenie węzła (uruchamianie programu systemu ROS)
  `rosrun package-name executable-name`
- tworzenie węzła wraz z nadaniem mu nazwy
  `rosrun package-name executable-name __name:=node-name`
- wyświetlenie listy uruchomionych węzłów
  `rosnode list`
- uzyskiwanie informacji na temat węzła
  `rosnode info node-name`
- zatrzymanie pracy (zabicie) węzła
  `rosnode kill node-name`
- usunięcie z listy zatrzymanego (zabitego) węzła
  `rosnode cleanup`

3.3 Tematy i wiadomości

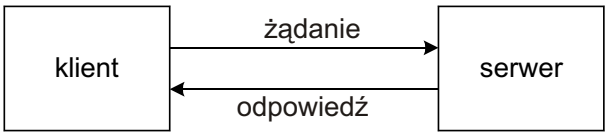
System ROS udostępnia następujący zbiór poleceń dotyczących tematów† i wiadomości przesyłanych w obrębie tematów:

- lista aktywnych tematów
  `rostopic list`
- wyświetlenie wiadomości przesyłanych aktualnie w obrębie danego tematu
  `rostopic echo topic-name`
- mierzenie częstotliwości publikowania wiadomości (wiadomość/sec)
  `rostopic hz topic-name`
- mierzenie przepustowości wiadomości (bajt/sec)
  `rostopic bw topic-name`
- uzyskiwanie informacji na temat danego tematu
  `rostopic info topic-name`
- uzyskiwanie szczegółowych informacji na temat przesyłanych wiadomości
  `rosmsg show message-type-name`
- publikowanie wiadomości
  `rostopic pub -r rate-in-hz topic-name message-type message-content`

3.4 Usługi

Usługi‡ (ang. services) oferują komunikację dwukierunkową: węzeł wysyła informację do innego węzła i oczekuje na odpowiedź. Rysunek 2 obrazuje przepływ informacji w ramach usług. Zatem węzeł–klient wysyła żądanie/prośbę (ang. request) do węzła–serwera, ten realizuje żą-

*http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes
†http://wiki.ros.org/ROS/Tutorials/UnderstandingTopics
‡http://wiki.ros.org/ROS/Tutorials/WritingServiceClient(c++)



Rysunek 2: Schemat komunikacji w obrębie klient–serwer

danie i odsyła odpowiedź (ang. response) do węzła–klienta. Usługi, podobnie jak wiadomości, posiadają konkretny typ danych, jednak podzielony jest on na dwie części: typ danych żądania i typ danych odpowiedzi.

Korzystanie z usług z linii poleceń wygląda następująco:

- lista aktywnych usług
  `rosservice list`
- lista usług oferowanych przez dany węzeł
  `rosnode info node-name`
- odnajdywanie węzła oferującego daną usługę
  `rosservice node service-name`
- odczytywanie typu danych konkretnej usługi
  `rosservice type service-name`
- szczegółowe informacje na temat typu danych usługi (osobno dla żądania i odpowiedzi)
  `rossrv show service-data-type-name`
- wywołanie usługi
  `rosservice call service-name request-content`
  przy czym należy pamiętać, że pole *request-content* powinno zawierać listę wartości uzupełniających poszczególne pola żądania

Programowe wywoływanie usług wygląda następująco:

Program – klient

Na początku należy dołączyć odpowiedni plik nagłowkowy definiujący typ danych żądania i odpowiedzi usługi

```
#include<package_name/type_name.h>
```

Następnie, po zainicjowaniu węzła (ros::init) i utworzeniu obiektu węzła (ros::NodeHandle), należy utworzyć obiekt klienta, który zajmie się wywołaniem usługi

```
ros::ServiceClient client = node_handle.serviceClient<service_type>
(service_name);
```

przy czym *node_handle* to nazwa obiektu definiującego węzeł, *service_type* to nazwa obiektu usługi zadeklarowanej wcześniej w pliku nagłówkowym, *service_name* to nazwa usługi wywoływanej. Obiekt klienta zwraca szczegółowe informacje na temat wywoływanej usługi.
W kolejnym kroku tworzony jest obiekt żądania, który wyśle dane do serwera i obiekt odpowiedzi klas

```
package_name::service_type::Request
package_name::service_type::Response
```

W ostatnim kroku należy uzupełnić pola obiektu żądania i wywołać usługę

```
bool success = service_client.call(request,response);
```

Powyższa metoda lokalizuje węzeł serwera, przesyła dane żądania, czeka na odpowiedź i zwraca dane odpowiedzi w odpowiednim obiekcie (`Response`). Metoda `call` zwraca wartość typu boolean mówiącą o tym, czy wywołanie usługi zakończyło się powodzeniem. Jeśli tak, mamy dostęp do danych odpowiedzi.

Program – serwer

Sposób tworzenia programu–serwera jest opisany w rozdz. 8.4 [3]. Przykładowe programy klienta i serwera wraz z ich szczegółowym opisem można odnaleźć w rozdz. 2 [4].

3.5 W razie problemów ☺

Jeśli ROS zachowuje się inaczej niż oczekujemy można skorzystać z polecenia

```
roswtf
```

które sprawdza poprawność wywołań (w tym zmienne środowiskowe, zainstalowane pliki, uruchomione węzły i inne).

4 Symulator TURTLESIM

W celu zapoznania się z koncepcją programowania w systemie ROS, wykorzystamy symulator *turtlesim*. Opisane poniżej czynności wygodnie jest wykonywać z poziomu osobnych powłok użytkownika (terminali), co pozwoli na łatwe obserwowanie komunikatów o stanie procesów i ewentualnych błędach[§]. W celu dodania do sesji basha odpowiednich zmiennych środowiskowych systemu ROS, należy zadbać, by w uruchamianych powłokach ustawiona była konfiguracja inicjowana skryptem /opt/ros/hydro/setup.bash[¶]. Postępuj według instrukcji umieszczonych poniżej. Upewnij się, że uruchomiony jest węzeł nadrzędny sprawujący kontrolę nad pozostałymi węzłami. Jeśli nie, wydaj polecenie `roscore`[‖].
Na początek spróbujmy uruchomić symulator *turtlesim* i przećwiczyć omawiane wcześniej podstawowe polecenia platformy ROS. W tym celu otwórz dwa terminale jednocześnie i wywołaj w nich kolejno polecenia:

```
rosrun turtlesim turtlesim_node
rosrun turtlesim turtle_teleop_key
```

[§] By zapobiec pojawieniu się na pulpicie zbyt wielu okien terminala można skorzystać z aplikacji pozwalających na uruchomienie wielu terminali w jednym oknie, takich jak *terminator*

[¶] Upewnij się czy linia `source /opt/ros/hydro/setup.bash` została dodana do pliku `~/.bashrc`. Jeśli nie, w każdym nowo otwieranym terminalu należy wydać polecenie `source /opt/ros/hydro/setup.bash`

[‖] Lokalne ustawienia konfiguracji platformy ROS mogą spowodować pojawienie się po uruchomieniu jądra ostrzeżenia „`WARNING: ROS_MASTER_URI [http://10.104.16.119:11311] host is not set to this machine auto-starting new master`". W takiej sytuacji, poniżej tego ostrzeżenia zostanie podany aktualny adres sieciowy uruchomionego jądra (np. `ROS_MASTER_URI=http://10.104.16.102:11311/`). Należy każdorazowo (w każdym nowo otwartym terminalu) ustawić wartość zmiennej środowiskowej `ROS_MASTER_URI` korzystając z aktualnego adresu sieciowego (np. `export ROS_MASTER_URI=http://10.104.16.102:11311/`). Powyższe polecenie można wpisać do pliku ~/.bashrc, wówczas wydawanie go przy każdym nowo otwieranym terminalu nie będzie konieczne. Należy jednak pamiętać, by na koniec zajęć usunąć je z pliku ~/.bashrc

Polecenie `rosrun` umożliwia uruchamianie węzłów. Przyjmuje on dwa parametry: nazwę pakietu oraz nazwę pliku wykonywalnego umieszczonego w pakiecie, inaczej nazwę węzła (`rosrun` *`package-name executable-name`*). W naszym przypadku, w ramach pakietu *turtlesim* utworzyliśmy dwa węzły. Jeden z nich jest instancją programu wykonywalnego *turtlesim_node* i jest on odpowiedzialny za utworzenie okna, a w nim symulatora robota (w kształcie żółwia), natomiast drugi jest instancją programu wykonywalnego *turtle_teleop_key*, który umożliwia sterowanie robotem za pomocą klawiszy strzałek. Drugi węzeł konwertuje naciśnięcie odpowiedniego klawisza strzałki na polecenie ruchu, a następnie polecenie to przesyła do węzła *turtlesim_node*. Spróbuj sterować robotem za pomocą strzałek (pamiętaj, że w celu wprowadzadzania instrukcji ruchu dla robota, powinieneś mieć aktywny terminal, w którym wywołałeś `rosrun turtlesim turtle_teleop_key`).

Pozostawiając uruchomione oba węzły przetestuj polecenia realizujące następujące czynności:

- wyświetlenie listy uruchomionych węzłów
- uzyskanie informacji na temat węzła *turtlesim*
- wyświetlenie listy aktywnych tematów
- wyświetlenie wiadomości przesyłanych w obrębie tematu *turtle1/cmd_vel* i *turtle1/pose*
- uzyskiwanie szczegółowych informacji na temat wiadomości *geometry_msgs/Twist* i *geometry_msgs/Pose*
- sprawdzenie listy oferowanych usług dla węzła *turtlesim*
- sprawdzenie typu danych usługi *spawn*
- wywołanie usługi *spawn* (proszę pamiętać o parametrach wywołania)

W kolejnym kroku, spróbujmy utowrzyć własny pakiet. W tym celu:

1. Utwórz swój własny obszar/katalog roboczy (ang. workspace), w którym będziesz przechowywać nowo utworzone pakiety/projekty[**]

   ```
   mkdir catkin_ws
   ```

   Wejdź do tak utworzonego katalogu.

2. Następnie utwórz katalog *src*, w którym będziesz przechowywać kody źródłowe swoich pakietów.

   ```
   mkdir src
   ```

3. W katalogu *src* utwórz swój projekt

   `catkin_create_pkg` *`package_name`*

   Wywołanie powyższego polecenia spowoduje, że zostanie utworzony katalog o nazwie takiej jak nazwa pakietu, a w nim dwa pliki konfiguracyjne: *package.xml* stanowiący dokumentację pakietu i *CMakeLists.txt*. *CMakeLists* jest skryptem dla wieloplatformowego systemu budowania *CMake*. Zawiera on listę instrukcji budowania mówiących o tym jakie programy wykonywalne będą tworzone, jakie pliki źródłowe będą wykorzystane do ich utworzenia, gdzie poszukiwać dołączanych plików/bibliotek itp.

[**] http://wiki.ros.org/ROS/Tutorials/CreatingPackage

4. W katalogu projektu utwórz jeszcze raz katalog *src*, w którym bedziesz przechowywać pliki źródłowe projektu

   ```
   mkdir src
   ```

   Teraz możesz już spróbować pisać własne programy w systemie ROS (obowiązujący język programowania to *C++*). W dalszym kroku spróbujemy utworzyć węzeł, który będzie publikował losowe prędkości. Prędkości będą odbierane przez zaimplementowanego w pakiecie *turtlesim* robota - żółwia, który odzwierciedla zachowanie się znanego robota typu monocykl. Nietrudno się domyślić, że sterowaniami robota są: prędkość zmiany orientacji robota oraz prędkość postępowa robota.
   Zadanie, jako przykładowe, zostało szczegółowo opisane w [3] (podrodział 3.3). Sposób tworzenia, kompilacji oraz uruchamiania nowego węzła jest opisany w podrozdziale 3.2 pozycji [3]. W celu realizacji zadania:

5. Do katalogu *src* w katalogu projektu (patrz punkt 4 powyżej) skopiuj plik *pubvel.cpp* z rysunku (3) (znajdziesz go w /opt/ROS_Lab1_5/materialy/). Zapoznaj się z jego treścią.

   Najważniejsze informacje dotyczące programu *pubvel* (rys. 3):

   Program *pubvel* tworzy węzeł, który generuje oraz publikuje losowe prędkości. Prędkości są typu `Twist`. Jako że typ `Twist` zdefiniowano w pakiecie `geometry_msgs`, konieczne jest pojawienie się w programie dyrektywy

   ```
   #include<geometry_msgs/Twist.h>
   ```

   Dalej, program tworzy obiekt publikujący `pub` klasy `ros::Publisher`.

   ```
   ros::Publisher pub = nh.advertise<geometry_msgs::Twist>
   ("turtle1/cmd_vel", 1000);
   ```

   Tutaj ustalany jest typ wiadomości (`geometry_msgs/Twist`) publikowanych przez obiekt publikujący, nazwa tematu (`turtle1/cmd_vel`), w obrębie którego będą publikowane wiadomości oraz rozmiar kolejki wiadomości (1000). Temat `turtle1/cmd_vel` zdefiniowano w pakiecie *turtlesim*, domyślamy się więc, że prędkości będą subskrybowane przez węzeł definiujący robota–żółwia (`turtlesim_node`). Typ wiadomości `geometry_msgs/Twist` składa się z dwóch pól opisujących prędkość liniową (m/sec) oraz prędkość kątową (rad/sec) robota

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

   Każda z prędkości jest więc wektorem (x,y,z) (elementy typu *double*) opisującym jej poszczególne współrzędne. Program *pubvel* przypisuje losowe wartości poszczególnym składowym prędkości: `msg.linear.x` (prędkość postępowa robota wzdłuż osi x) oraz `msg.angular.z` (prędkość zmiany orientacji robota wokół osi z). Pozostałe składowe przyjmują domyślne (zerowe) wartości. Dalej wywoływana jest metoda `publish` obiektu publikującego

   ```
   pub.publish(msg)
   ```

```
// this program randomly-generated velocity messages for turtlesim
#include <ros/ros.h>
#include <geometry_msgs/Twist.h>
#include <stdlib.h>

int main(int argc, char ** argv) {
	// init ROS
	ros::init(argc, argv, "publish_velocity");
	ros::NodeHandle nh;

	// create a publisher object
	ros::Publisher pub =
	nh.advertise<geometry_msgs::Twist>("turtle1/cmd_vel", 1000);

	// seed random number generator
	srand(time(0));

	// loop at 2Hz until the node is shutdown
	ros::Rate rate(2);
	while(ros::ok) {

		//  create and fill in the message
		geometry_msgs::Twist msg;
		msg.linear.x = double(rand())/double(RAND_MAX);
		msg.angular.z = 2 * double(rand())/double(RAND_MAX) - 1;

		// publish the message
		pub.publish(msg);

		// send a message to rosout
		ROS_INFO_STREAM("Sending random velocity: "
		<< " linear=" << msg.linear.x
		<< " angular=" << msg.angular.z);

		// wait until its time for another iteration
		rate.sleep();
	}
}
```

Rysunek 3: Listing programu `pubvel.cpp` publikującego losowe prędkości dla robota zaimplementowanego w pakiecie *turtlesim*

Metoda dodaje wiadomość do kolejki wiadomości, skąd wiadomość możliwie najszybciej zostaje wysyłana do wszystkich subskrybentów odpowiadających danemu tematowi. Polecenie `ROS_INFO_STREAM` wysyła komunikaty na wyjście. Program powtarza kroki publikowania wiadomości w pętli while. Warunek pętli `ros::ok()` zwraca *true*, dopóki węzeł istnieje i wykonuje pracę. *False* uzyskamy gdy np. wykonamy instrukcję `rosnode kill` lub gdy do programu wyślemy sygnał `Ctrl-C`. Należy pamiętać, że `Ctrl-C` powoduje, że warunek w pętli while przyjmuje wartość *false*, natomiast nie przerywa programu. Działanie programu można przerwać poprzez wysłanie sygnału `Ctrl-Z` (ten sam efekt można uzyskać poprzez wywołanie w programie `ros::shutdown()`). Wreszcie, ważnym elementem systemu `ROS` jest kontrolowanie częstotliwości publikowania wiadomości. Częstotliwość tę ustala obiekt `ros::Rate`. Parametrem jego konstruktora jest częstotliwość podawana w hercach (Hz). I tak,

```
ros::Rate rate(2)
```

tworzy obiekt `rate`, który reguluje pracą pętli dopuszczając w tym przypadku na wykonywanie instrukcji pętli z częstotliwością dwa razy na sekundę. Pod koniec pętli wykonywana jest metoda `sleep` obiektu `rate`

```
rate.sleep()
```

która nie pozwala wykonywać się pętli szybciej niż to określono wcześniej za pomocą wyspecyfikowania częstotliwości.

6. W katalogu projektu edytuj pliki *package.xml* oraz *CMakeLists.txt*. W szczególności, w domyślnej wersji pliku *CMakeLsts.txt* zawarta jest linia

   ```
   find_package(catkin REQUIRED)
   ```

   w której, w sekcji `COMPONENTS`, określa się zależności od innych pakietów. W naszym przypadku linia powinna wyglądać następująco

   ```
   find_package(catkin REQUIRED COMPONENTS roscpp geometry_msgs)
   ```

   Ponadto, należy dodać linię określającą uruchamiany węzeł oraz listę plików źródłowych, na podstawie których tworzony jest dany węzeł, a zatem (jeśli pakiet zawiera więcej węzłów, dla każdego węzła dodaje się osobną linię)

   ```
   add_executable(pubvel src/pubvel.cpp)
   ```

   I wreszcie linia mówiąca o użyciu odpowiednich bibliotek podczas konsolidacji węzłów (dla każdego węzła osobna linia)

   ```
   target_link_libraries(pubvel ${catkin_LIBRARIES})
   ```

   Zawartość przykładowych plików została zaprezentowana oraz szczegółowo opisana w podrozdziale 3.2 pozycji [3] (listingi 3.1 oraz 3.3). W podrozdziale 3.3 zasygnalizowano jakich zmian należy dokonać w plikach zaprezentowanych w listingach 3.1 i 3.3, by odpowiadały naszemu projektowi.

7. Przejdź do katalogu obszaru roboczego. Skompiluj projekt poleceniem

   ```
   catkin_make
   ```

   Powyższe polecenie buduje wszystkie projekty znajdujące się w naszym obszarze roboczym, stąd, by móc wywołać polecenie, należy znajdować się w katalogu obszaru roboczego. Wywołane polecenie utworzy m.in. katalogi *devel* i *bulid*.

8. W katalogu *devel* zostanie utworzony skrypt *setup.bash*, który należy uruchomić poleceniem

   `source devel/setup.bash`

   Omawiany skrypt ustawia zmienne środowiskowe, które umożliwią systemowi ROS m.in. zlokalizowanie pakietów oraz programów wykonywalnych. Polecenie należy wywołać w każdym nowo otwieranym terminalu.

9. Uruchom symulator robota

   `rosrun turtlesim turtlesim_node`

10. Uruchom nowo utworzony węzeł poleceniem

    `rosrun nazwa_projektu nazwa_wezla`

    (np. `rosrun projekt pubvel`).

11. Zaobserwuj jak zachowuje się robot w odpowiedzi na zadawane przez węzeł *pubvel* prędkości losowe.

12. W podrozdziale 3.4 pozycji [3] opisano jak tworzy się węzły subskrybujące.

13. W systemie ROS konieczność uruchamiania dwóch lub większej liczby węzłów w tym samym czasie stanowi podstawę do sporządzenia pliku uruchamiającego (ang. launch file). Plik uruchamiający (o nazwie z rozszerzeniem *.launch*) znajduje się w katalogu danego pakietu i stanowi specyfikację (w formacie *XML*) węzłów, które mają być uruchomione w tym samym czasie. Odpowiednio zdefiniowany plik wywołuje się poleceniem

    `roslaunch package-name launch-file-name`

    Polecenie `roslaunch` sprawdza czy węzeł nadrzędny jest uruchomiony i jeśli nie uruchamia go automatycznie. Wysłanie sygnału SIGINT (`Ctrl-C`) kończy sesję uruchomioną poleceniem `roslaunch` (w szczególności kończy pracę uruchomionych węzłów).

    Szczegółowe informacje dotyczące składni/formatu pliku uruchamiającego znajdziesz w rozdziale 6.2 pozycji [3]. Przykładowy plik uruchamiający umieszczono na rysunku 4. Plik uruchamia węzeł *turtle_node* implementujący robota, węzeł *turtle_teleop_key* pozwalający na sterowanie robotem za pomocą klawiszy strzałek oraz utworzony przez nas węzeł publikujący prędkości losowe (*pubvel*). Umieść omawiany plik uruchamiający w katalogu z nazwą utworzonego projektu (plik znajdziesz w /opt/ROS_Lab1_5/materialy), edytuj go zmieniając nazwę pakietu na właściwą (pole: `pkg="package-name"`) i uruchom za pomocą polecenia `roslaunch package-name launch-file-name`. Sprawdź jak działają poszczególne węzły. Spróbuj zamykać kolejne węzły. Co zaobserwowałeś? Zinterpretuj znaczenie poleceń: respawn="true" oraz required="true".

    Komunikacja w obrębie pojedynczego tematu odbywa się na zasadzie: "wielu–do–wielu" (ang. "many–to–many"). Jako że pojedynczy temat jest dzielony jednocześnie przez wiele węzłów publikujących i subskrybujących, to bez względu na to, który węzeł publikuje, wiadomości odbierane są przez wszystkich subskrybentów danego tematu. Zaobserwuj działanie komunikacji "wielu–do–wielu". Próbuj na przemian sterować robotem za pomocą węzła *pubvel* oraz węzła (*turtle_teleop_key*).

    Dzięki temu, że poszczególne węzły działają niezależnie od siebie, a jedynym ich łącznikiem są tematy i wiadomości, potrafimy zorganizować komunikację "jeden–do–jednego" (ang. “one–to–one“). Jednym ze sposobów jest wykorzystanie idei przestrzeni nazw (ang.

```
<launch>
 <node
      pkg="turtlesim"
      type="turtlesim_node"
      name="turtlesim"
      respawn="true"
 />
<node
      pkg="turtlesim"
      type="turtle_teleop_key"
      name="teleop_key"
      required="true"
      launch-prefix="xterm -e"
/>
<node
      pkg="pakiet"
      type="pubvel"
      name="pubvel"
      launch-prefix="xterm -e"
/>
</launch>
```

Rysunek 4: Listing pliku uruchamiającego `przyklad.launch`

namespace) (patrz rozdział 6.3 [3]). W tym przypadku, w obrębie pliku uruchamiającego przypisuje się poszczególnym węzłom atrybuty definiujące ich domyślną przestrzeń nazw, w obrębie której będą uruchamiane poszczególe węzły (`ns="namespace"`). Przypisanie w pliku uruchamiającym poszczególnym węzłom atrybutu `ns` oznacza, że w ramach jednego tematu komunikować się będą jedynie węzły dzielące ten sam atrybut definiujący ich domyślną przestrzeń nazw.

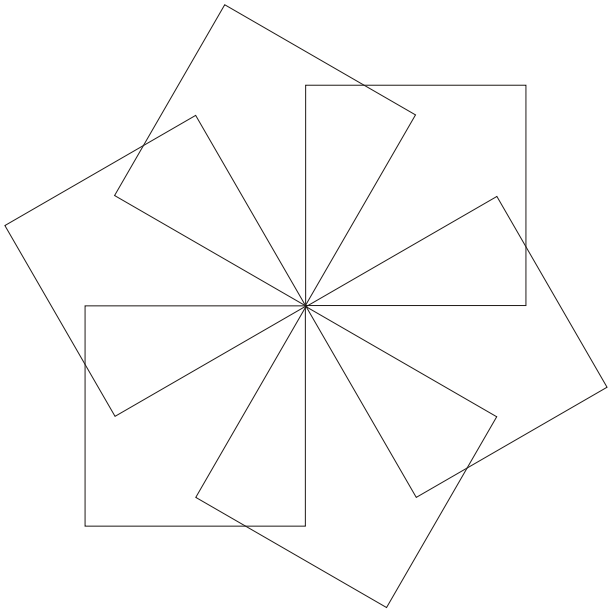
Innym sposobem zapewnienia komunikacji "jeden–do–jednego"jest wykorzystanie idei usług (patrz rozdział 3.4).

5 Zadania do wykonania

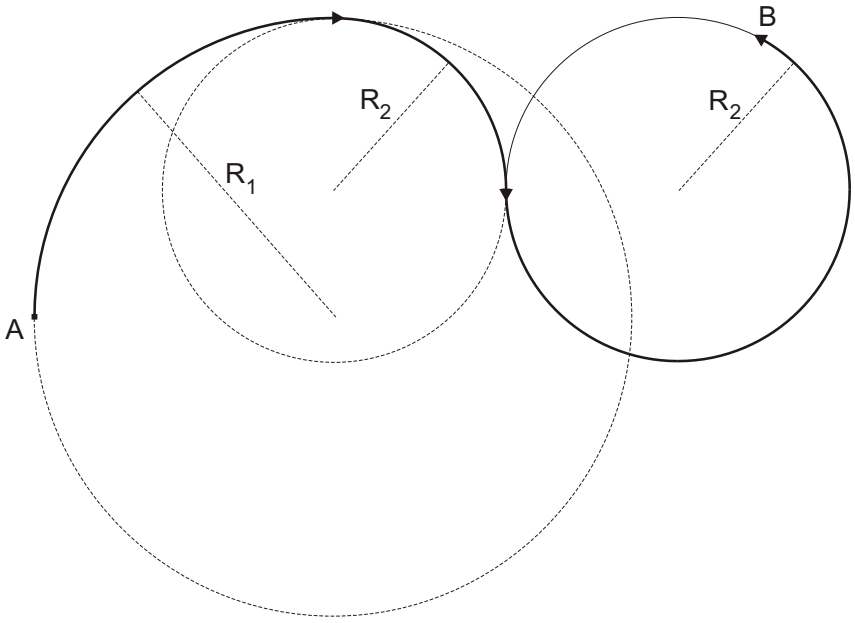
1. Na przykładzie węzła *pubvel* utwórz węzeł publikujący prędkości pozwalające zrealizować zadaną na rys. 5 ścieżkę robota zdefiniowanego w pakiecie *turtlesim*. Ścieżka jest cykliczna i składa się z szeregu wzjemnie zorientowanych kwadratów. Niech długość boku kwadratu oraz kąt jego obrotu będą zadawane przez użytkownika zaraz po uruchomieniu węzła. Wskazówka: do obliczenia czasu potrzebnego na realizację zadanego odcinka drogi wykorzystaj informacje dotyczące częstotliwości publikowania wiadomości (`ros::Rate`).

2. Utwórz węzeł, który opublikuje prędkości przeprowadzające robota z punktu *A* do punktu *B* wzdłuż ścieżki zadanej na rys. 6. Ścieżka składa się z łuków o zadanych promieniach krzywizny.

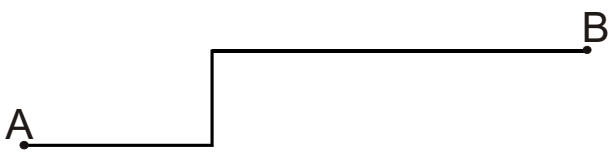
3. W ramach węzła publikującego wygeneruj funkcje sterujące $u = (v, \omega)$ przeprowadzające robota z punktu A do punktu B wzdłuż ścieżki zadanej na rys. 7. Znajdź sterowania $\hat{u} = (\hat{v}, \hat{\omega})$ przeprowdzającego robota z punktu B do punktu A.



Rysunek 5: Ścieżka 1



Rysunek 6: Ścieżka 2



Rysunek 7: Ścieżka 3

4. Dla każdego z poprzednich zadań sporządź plik uruchamiający. Informacje dotyczące składni/formatu pliku uruchamiającego znajdziesz w rozdziale 6.2 pozycji [3]. Niech węzły publikujące prędkości uruchamiają się w odzielnych terminalach (skorzystaj z polecenia `launch-prefix="command-prefix"`). Dodatkowo, niech węzły publikujące będą węzłami wymagalnymi (jeśli praca wymagalnego węzła zostanie z jakiegoś powodu przerwana, to plik uruchamiający przerwie pracę pozostałych aktywnych węzłów i zakończy swoje działanie). Wreszcie, zapewnij, by węzeł symulujący pracę robota (*turtlesim_node*) był, w razie przerwania jego pracy, uruchamiany na nowo.

5. Sporządź plik uruchamiający jednocześnie dwa pierwsze zadania. Zaobserwuj jak wygląda praca węzłów w ramach komunikacja "wielu–do–wielu". W celu zapewnienia poprawnej komunikacji poszczególych węzłów ("jeden–do–jednego") skorzystaj z idei przestrzeni nazw (rozdz. 6.3 [3]).

6. Przygotuj plik uruchamiający, który:

   - uruchomi w odzielnym terminalu napisany w pierwszym zadaniu węzeł publikujący prędkości realizujące śledzenie ścieżki z Rys. 5
   - uruchomi dwie instancje symulatora robota (*turtlesim_node*)
   - jeden z robotów będzie odpowiadał na prędkości publikowane przez węzeł, natomiast drugi robot będzie podążał za pierwszym robotem. Wskazówka: Wykorzystaj węzeł *mimic*[††,‡‡] oraz pojęcie mapowania (patrz rozdz. 6.4 [3]).

7. Zrealizuj następujące zadanie: w oknie symulatora *turtlesim* znajdują się dwa roboty. Jeden z nich porusza się z pewnymi prędkościami, natomiast drugi realizuje ruch dla tych samych prędkości, ale ze zmienionymi znakami. Wskazówka: węzeł publikujący prędkości dla drugiego robota powinien subskrybować prędkości publikowane dla pierwszego robota, zmieniać ich znaki na przeciwne i w nowej postaci publikować prędkości dla drugiego robota. Wykorzystaj usługę */spawn* pakietu *turtlesim* w celu umieszczenia nowego robota w oknie symulatora (patrz rozdz. 8.3 [3]). Spróbuj wykorzystać usługę */clear* pakietu *turtlesim*, aby na początku wyczyścić okno symulatora. Podobnie, spróbuj wykorzystać usługę */teleport_relative* lub */teleport_absolute* pakietu *turtlesim*, aby umieścić robota we właściwej pozycji startowej.

8. Zapoznaj się z informacjami na temat tworzenia programu–serwera (rozdz. 8.4 [3]). Napisz program–serwer oferujący usługę zamiany znaków prędkości. Wykorzystaj go do realizacji poprzedniego zadania.

6 Sprawozdanie

Sprawozdanie z przebiegu ćwiczenia powinno zawierać:

- Imię i nazwisko autora, numer i termin grupy, skład grupy, temat ćwiczenia, datę wykonania ćwiczenia.
- Cel ćwiczenia.

[††] http://wiki.ros.org/turtlesim

[‡‡] http://wiki.ros.org/ROS/Tutorials/UsingRxconsoleRoslaunch
http://wiki.ros.org/ROS/Tutorials/UnderstandingServicesParams
http://wiki.ros.org/ROS/Tutorials/WritingServiceClient(C++)

- Opis przebiegu i efektu wykonania realizowanych zadań.
- Wnioski końcowe.

Literatura

[1] Mariusz Janiak. *Instrukcja bezpieczeństwa i higieny pracy w Laboratorium Robotów Autonomicznych L1.5*. Katedra Cybernetyki i Robotyki, Politechnika Wrocławska, Instrukcja Laboratorium Robotów Autonomicznych L1.5, 2015.

[2] Mariusz Janiak, Aleksandra Grzelak. *Instrukcja bezpieczeństwa i higieny pracy przy obsłudze robotów Pionneer P3-DX*. Katedra Cybernetyki i Robotyki, Politechnika Wrocławska, Instrukcja Laboratorium Robotów Autonomicznych L1.5, 2015.

[3] J. M. O'Kane. *A Gentle Introduction to ROS*. Independently published, 2013. Available at `http://www.cse.sc.edu/~jokane/agitr/`.

[4] A. Martinez i E. Fernandez. *Learning ROS for Robotics Programming. A practical, instructive, and comprehensive guide to introduce yourself to ROS, the top - notch, leading robotics framework*. 2013. Available at `http://it-ebooks.info/book/3183/`.

[5] Joanna Ratajczak. *Weryfikacja własności ruchowych układów nieholonomicznych na przykładzie robota mobilnego Pioneer 3DX*. Katedra Cybernetyki i Robotyki, Politechnika Wrocławska, Instrukcja Laboratorium Robotów Autonomicznych L1.5, 2015.

[6] Robert Muszyński. *Zastosowanie czujnika głębi Kinect do interakcji z robotem mobilnym z wykorzystaniem platformy ROS*. Katedra Cybernetyki i Robotyki, Politechnika Wrocławska, Instrukcja Laboratorium Robotów Autonomicznych L1.5, 2015.