ARR 2016

Projekt przejściowy — Symulator laboratorium L1.5

24 stycznia 2017

Spis treści

1. Wstęp

Środowisko do symulowania działania robotów w warunkach laboratorium L1.5 został wykonany na zajęciach projektu przejściowego grupy ARR. Środowisko jest dostarczony w postaci kontenera dockera na którym znajduje się system operacyjny ubuntu 16.04 ros kinetic oraz gazebo 7.5.

$$x \tag{1.1}$$

Celem projektu było umożliwienie przygotowania się studentom do zajęć laboratoryjnych poprzez testowanie napisanych przez siebie programów w środowisku zbliżonym do udostępnianej przestrzeni podczas zajęć praktycznych. Dostarczony projekt umożliwia wybór laboratorium i robotów pionier oraz pozwala na dodawanie elementów sceny takich jak przeszkody. Z wykorzystaniem środowiska ros można sterować robotami. Symulator pozwala również na zmianę pozycji robota.

$$y \tag{1.2}$$

Osoba której zadanie ograniczać się będzie do wykonania ćwiczeń laboratoryjnych powinna zapoznać się z instrukcją z dodatku A. W celu ułatwienia rozwoju i modyfikacji projektu stworzony został dodatek B.

1.1. Motywacja

1.2. Cel projektu

2. Wykorzystane narzędzia

2.1. Zarządzanie projektem

2.1.1. Git

2.1.2. Trello

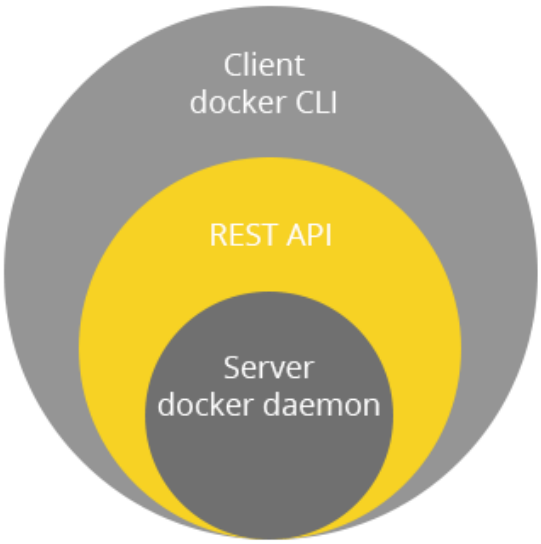
2.2. Docker

Docker to narzędzie szturmem zdobywające popularność na serwerach, szczególnie w środowiskach chmurowych, gdzie z powodzeniem wspiera lub czasem nawet zastępuje klasyczną wirtualizację oferowaną przez rozwiązania typu VMware lub XEN.
Docker działa tylko na jądrze Linux i pozwala uruchamiać tylko aplikacje przeznaczone dla Linuxa, ale dla wszystkich użytkowników Windows i Mac jest przygotowane narzędzie Docker Toolbox, które pozwala zainstalować Dockera w minimalnej maszynie wirtualnej pod kontrolą VirtualBox'a.
Zdecydowaną przewagą Dockera nad wirtualizacją jest możliwość uruchomienia aplikacji w wydzielonym kontenerze, ale bez konieczności emulowania całej warstwy sprzętowej i systemu operacyjnego. Docker uruchamia w kontenerze tylko i wyłącznie proces(y) aplikacji i nic więcej. Efektem jest większa efektywność wykorzystania zasobów sprzętowych, co przy rozproszonych aplikacjach instalowanych do tej pory na kilkunastu bądź kilkudziesięciu wirtualnych maszynach przynosi konkretne oszczędności.
Docker zastępuje wirtualizację przez stosowanie konteneryzacji. Konteneryzacja polega na tym, że umożliwia uruchomienie wskazanych procesów aplikacji w wydzielonych kontenerach, które z punktu widzenia aplikacji są odrębnymi instancjami środowiska uruchomieniowego. Każdy kontener posiada wydzielony obszar pamięci, odrębny interfejs sieciowy z własnym prywatnym adresem IP oraz wydzielony obszar na dysku, na którym znajduje się zainstalowany obraz systemu operacyjnego i wszystkich zależności / bibliotek potrzebnych do działania aplikacji.
Kontenery Dockera działają niezależnie od siebie i do chwili, w której świadomie wskażemy zależność pomiędzy nimi, nic o sobie nie wiedzą.
Dwie główne korzyści płynące z korzystania z Dockera to łatwość tworzenia środowisk deweloperskich oraz uproszczenie procesów dostarczania gotowych aplikacji na docelowe środowiska.
Zmorą każdego programisty jest tworzenie środowiska deweloperskiego na potrzeby każdego kolejnego projektu. Wiadomo, że każdy projekt będzie działał na innej bazie danych, innym kontenerze aplikacji z inną listą dodatkowych usług, które do czasu pojawienia się

narzędzi typu Vagrant instalowane były bezpośrednio na laptopie programisty. Utrzymanie kilku środowisk dla wielu projektów bywało niemożliwe. Vagrant w pewien sposób rozwiązuje ten problem przez tworzenie wirtualnego środowiska instalowanego za pomocą odpowiednich skryptów, ale to rozwiązanie nie rozwiązuje wszystkich problemów: nadal musimy napisać skrypty instalujące wszystkie zależności i potrzebny jest mocny sprzęt żeby udźwignąć pełne środowisko w trybie wirtualizacji. Kilka takich środowisk na laptopie może też skutecznie zająć całą przestrzeń dyskową.
Z pomocą przychodzi Docker, który pozwala zbudować środowisko deweloperskie bez wirtualizacji i bez większego wysiłku związanego z instalacją oprogramowania. Docker pozwala wykorzystywać gotowe obrazy zainstalowanych systemów, aplikacji i baz danych, które zostały wcześniej przygotowane i umieszczone w publicznym rejestrze. Rejestr jest dostępny za darmo i zawiera obrazy oficjalnie budowane przez opiekunów / twórców konkretnych rozwiązań: https://hub.docker.com/explore/. Dzięki temu, jeśli chcemy używać V-Rep, Gazebo czy ROS, jest duża szansa na to, że gotowy obraz będziemy mogli pobrać z repozytorium i nie tracić czasu na jego przygotowanie. Jeśli jednak nie znajdziemy tego czego szukamy, to zawsze możemy zbudować własny obraz bazując na jednym z bardziej generycznych zawierających tylko zainstalowany system operacyjny (Ubuntu, Fedora, itp.) lub zainstalowane środowisko uruchomieniowe (Java, Python czy ASP.NET).

2.2.1. Podstawowe elementy

Docker Engine to główna składowa i aplikacja typu klient-serwer, złożona z trzech elementów: klienta (CLI), serwera (daemon-a) oraz REST API.



Daemon Docker-a jest centralnym miejscem, z którego następuje zarządzanie kontenerami jak i obrazami. Dotyczy to zarówno ich pobierania, budowy czy uruchamiania. Polecenia do wykonywania tych czynności są przesyłane przez klienta z wykorzystaniem Docker REST API. Same obrazy są natomiast przechowywane w repozytorium obrazów, czy to publicznym czy prywatnym np. **Docker Hub**. W tym miejscu należy zaznaczyć, że poza wieloma oficjalnymi obrazami udostępniane są również nieoficjalne : budowane przez społeczność Docker-a.
W odróżnieniu od maszyn wirtualnych, kontenery wymagają dużo mniejszych zasobów do samego uruchomienia, a i sam czas ich uruchomienia jest znacząco niższy. Zostało to jednak uzyskane kosztem zmniejszenia izolacji pomiędzy kontenerem, a samym systemem

operacyjnym : poprzez współdzielenie jądra systemu.
Obrazy są tak naprawdę szablonami w trybie read-only, z których kontenery są uruchamiane. Składają się one z wielu warstw (layer-ów), które, dzięki zastosowaniu ujednoliconego systemu plików (UFS), Docker łączy w jeden konkretny obraz. Podstawą każdego jest obraz bazowy, np. Ubuntu, na który nakładane są kolejne warstwy. Każda kolejna czynność (instrukcja) wykonywana na obrazie bazowym tworzy kolejną warstwę, np. wykonanie komendy czy utworzenie pliku/katalogu. Komplet instrukcji tworzących obraz jest przechowywany w pliku Dockerfile. Podczas żądania pobrania obrazu przez klienta, plik ten jest przetwarzany, czego wynikiem jest finalny obraz.
Poprzez zastosowanie warstw uzyskano bardzo niskie zużycie przestrzeni dyskowej, ponieważ podczas zmiany obrazu czy jego aktualizacji, budowana jest nowa warstwa, która zastępuje poprzednią (aktualizowaną). Pozostałe warstwy pozostają nienaruszone. Oznacza to, iż możliwe jest współdzielenie warstw tylko do odczytu pomiędzy kontenerami, czego efektem jest dużo niższe zużycie przestrzeni dyskowej, w porównaniu do standardowych VM.
Registry, czyli repozytorium obrazów, jest faktycznym miejscem przechowywania obrazów. Może być publiczne bądź prywatne, lokalne bądź zdalne. Najpopularniejszym repozytorium jest Docker Hub, który oferuje wiele dodatkowych funkcjonalności, np. możliwość utworzenia repozytorium prywatnego. Oczywiście istnieją inne platformy, które pozwalają na przechowywanie swoich obrazów, jak np. Quay.io, ale możliwe jest także utworzenie własnej biblioteki : czy to lokalnie na komputerze, na którym został zainstalowany Docker czy zdalnie, na jednej z innych maszyn, którymi zarządzamy.
Wspomniany wielokrotnie **kontener** to efekt ujednolicenia warstw tylko do odczytu oraz pojedynczej warstwy do odczytu i zapisu, dzięki której możliwe jest funkcjonowanie wymaganych zadań. Kontener wykonuje określone wcześniej zadanie, najczęściej jedno. Zawiera system operacyjny, pliki użytkownika, a także tzw. metadane, dodawane automatycznie podczas tworzenia bądź startu kontenera.
Kontener określany jest również jako środowisko wykonywalne Dockera. Może przyjmować jeden z pięciu stanów:
— created : utworzony, gotowy do uruchomienia,
— up : działający, wykonujący zadanie,
— exited : wyłączony, w trybie bezczynności po zakończeniu zadania,
— paused : wstrzymany,
— restarting : w trakcie ponownego uruchamiania.
Czas działania kontenera jest zależny od zadania, które wykonuje. Samo uruchomienie składa się z siedmiu kroków:
1. Pobrania wybranego obrazu, pod warunkiem, że nie został już pobrany wcześniej.
2. Utworzenia kontenera.
3. Załadowania systemu plików i utworzenia warstwy do odczytu i zapisu.
4. Zainicjowania sieci bądź mostka sieciowego.
5. Konfiguracji sieci (adresu IP).
6. Uruchomienia zadania.
7. Przechwytywania wyjścia i prowadzenia dziennika zdarzeń.

Możliwe jest łączenie (linkowanie) kontenerów, przez co zyskują one bezpośrednie połączenie ze sobą. Same zadania wykonywane przez kontener są tak samo wydajne, jakby były uruchamianie bezpośrednio w systemie gospodarza.

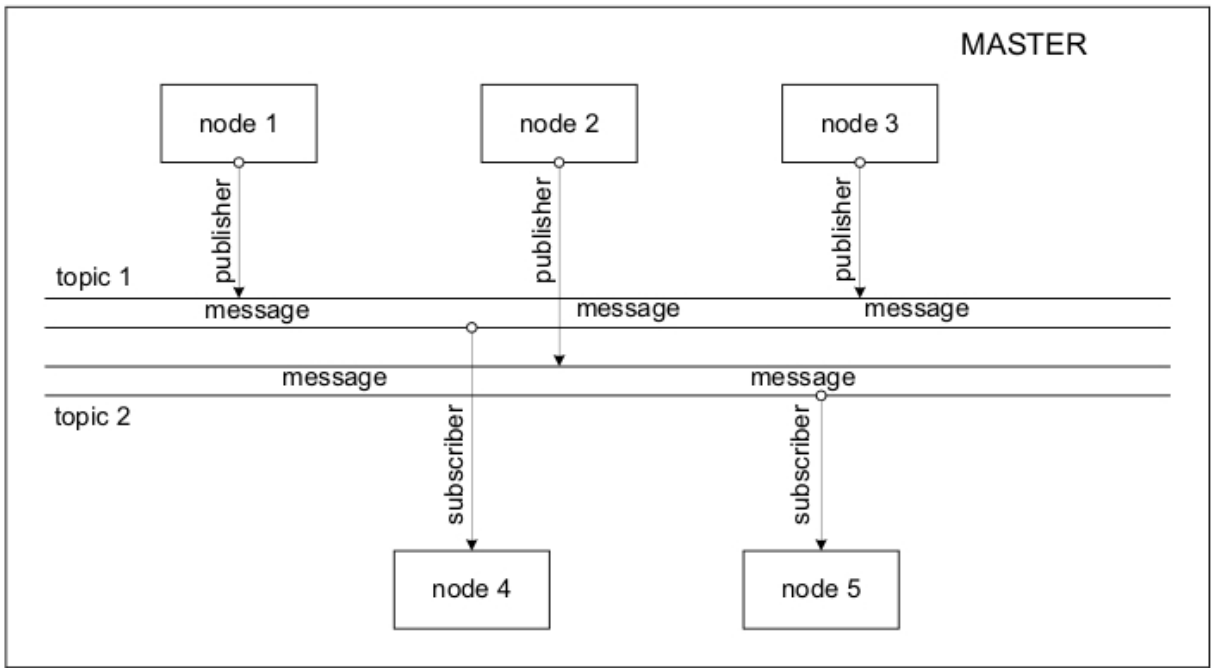
2.3. Gazebo

2.4. ROS

ROS jest meta-systemem operacyjnym rozwijanym w ramach ruchu wolnego oprogramowania. Zawiera: podstawowe procesy systemowe obsługujące urządzenia sprzętowe robota, sterowanie niskopoziomowe, implementacje wykonywania typowych funkcji, komunikację międzywątkową oraz zarządzanie pakietami. Oprócz tego dostarcza użytkownikowi narzędzia i biblioteki pozwalające na tworzenie i uruchomianie programu równocześnie na wielu komputerach. Jest platformą programistyczną przeznaczoną do tworzenia oprogramowania sterującego robotami. ROS jest szczegółowo opisany na stronie internetowej projektu [1].

2.4.1. Struktura

Dzięki swojej strukturze (zbiór narzędzi, bibliotek i konwencji) ROS znacząco upraszcza proces modelowania złożonych zachowań robota jednocześnie zapewniając łatwość przenośności kodu pomiędzy różnymi platformami robotycznymi. Zasada działania projektu w ROS polega na komunikacji *peer-to-peer* luźno połączonych ze sobą procesów (mogą być uruchomione na różnych maszynach) za pomocą infrastruktury komunikacyjnej zapewnianej przez platformę. Idea struktury środowiska przedstawiona jest na rysunku 2.1.



Rysunek 2.1. Schemat projektu na platformie ROS [4]

Pakiety (*packages*)

Pakiety są główną jednostką służącą do organizacji oprogramowania. Mogą zawierać: węzły, biblioteki zależne od ROSa, zbiory danych, pliki konfiguracyjne lub inne elementy użytecznie powiązane ze sobą. Pakiety są najmniejszą jednostką budulcową systemu

ROS, czyli najmniejszą rzeczą, którą można zbudować i udostępnić. Każdy z nich zawiera dokumentację (*manifest*) opisaną w pliku `package.xml`.

Węzły (*nodes*)

Węzły są wykonywalnymi instancjami programów środowiska ROS. System sterowania robotem zazwyczaj składa się z wielu węzłów. Przykładowo dla jednego robota mobilnego można wyróżnić kilka węzłów odpowiadających za różne funkcjonalności: skaner laserowy, silniki, lokalizację, planowanie ruchu itd. Wyróżnia się również węzeł nadrzędny `master`, który odpowiada za poprawność komunikacji między wszystkimi węzłami systemu.

Tematy (*topics*) i wiadomości (*messages*)

Węzły porozumiewają się ze sobą poprzez przesyłanie wiadomości. Wiadomość jest prostą strukturą danych, w której znajdować się mogą pola różnych typów prostych oraz tablic typów prostych. Co istotne, struktura może mieć zagnieżdżoną budowę, czyli składać się z dowolnej liczby zagnieżdżonych struktur i tablic. Wiadomości są przesyłane poprzez system transportowy zorganizowany na zasadzie nadawca/odbiorca (*publisher/subscriber*).

Węzeł wysyła wiadomość poprzez opublikowanie jej w danym temacie. Temat jest nazwą służącą do identyfikacji zawartości wiadomości. Węzeł odbierający śledzi wybrane tematy i otrzymuje wiadomości wysyłane do nich. Może być wiele równoczesnych nadawców i odbiorców dla jednego tematu oraz jeden węzeł może publikować i subskrybować wiele tematów. Poszczególne węzły uczestniczące w komunikacji nie są świadome istnienia innych. Odpowiada to idei rozłączenia produkcji danych od ich konsumpcji.

Usługi (*services*)

Usługi udostępniają inny mechanizm komunikacji pomiędzy węzłami. Usługa jest zdefiniowana poprzez parę struktur wiadomości: jedną dla żądania, drugą dla odpowiedzi. W przeciwieństwie do prostego przesyłania wiadomości, które przekazywane są w jednym kierunku na zasadzie wiele-do-wielu, usługi pozwalają na interakcję żądanie/odpowiedź. Takie podejście jest często wymagane w rozproszonym systemie. Węzeł-serwer oferuje usługę o danej nazwie, natomiast węzeł-klient korzysta z niej poprzez zgłoszenie wiadomości żądania, a następnie oczekiwanie na odpowiedź.

Worki (*bags*)

Worki są formatem zapisu i odtwarzania informacji zawartych w wiadomościach. Jest to istotny mechanizm służący do przechowywania danych, które mogą być trudne do zebrania np. danych sensorycznych. Umożliwiają również testowanie programów dla jednolitego, uprzednio zdefiniowanego, zestawu danych.

2.5. Qt

2.6. Inventor

3. Opis systemu

Symulator robotów jest doskonałym narzędziem dla każdej osoby zajmującej się robotyką. Pozwala szybko przetestować różne algorytmy i konstrukcje oraz skomplikowane systemy realizujące niecodzienne scenariusze. Jednym z takich narzędzi jest darmowy program Gazebo, przeznaczony do tworzenia dokładnych i efektywnych symulacji robotów działających w złożonych środowiskach. Posiada zaawansowany silnik fizyki, wysokiej jakości grafikę oraz wygodne i programowalne interfejsy.

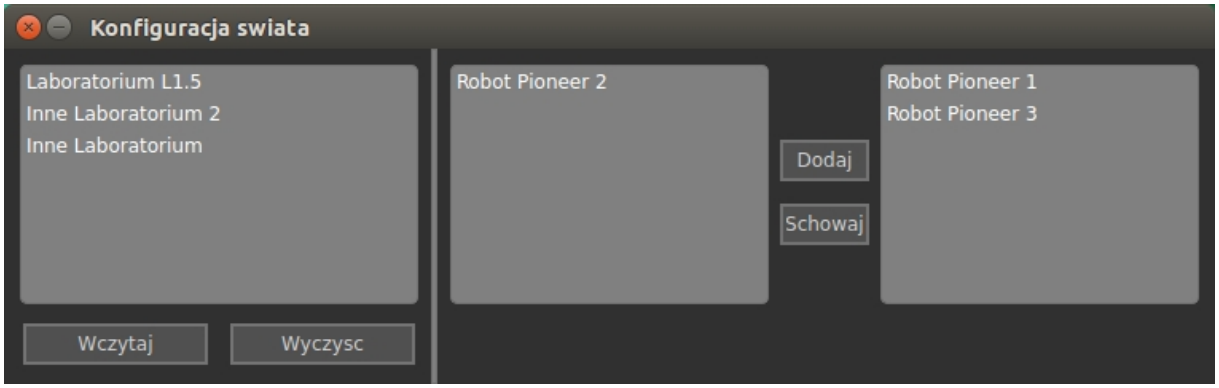
To powyżej to próba tłumaczenia opisu poniżej ze strony Gazebo :D

Robot simulation is an essential tool in every roboticist's toolbox. A well-designed simulator makes it possible to rapidly test algorithms, design robots, perform regression testing, and train AI system using realistic scenarios. Gazebo offers the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments. At your fingertips is a robust physics engine, high-quality graphics, and convenient programmatic and graphical interfaces. Best of all, Gazebo is free with a vibrant community.

3.1. Okienka

Funkcjonalności + Implementacja

3.1.1. Konfiguracja świata



Rysunek 3.1. Okno konfiguracji świata

3.1.2. Zarządzanie robotami

Okienko `Zarządzanie robotami` umożliwia ustawianie pozycji oraz orientacji wybranego robota na scenie. Zakładki umożliwiają wybór *Pioneera*, którego pozycję chcemy

zmienić. Aktywne są tylko zakładki skojarzone z robotami dodanymi aktualnie do świata. W polach odpowiedzialnych za pozycję i orientację robota kolor informuje o poprawności wprowadzonych danych – zielony oznacza poprawnie wypełnione pola, natomiast czerwony błędnie.

Rysunek 3.2. Okno zarządzania robotami

Implementacja

Klasa odpowiadająca za całe okienko to `RobotManagementWindow`. Dziedziczy ona po klasie `QDialog`. Jej kluczowe metody to:

```
public slots:
  void onAddNewRobot(int id);
  void onHideRobot(int id);
```

Odpowiadają one za to co dzieje się w oknie odpowiednio w momencie dodania robota i schowania go. Mianowicie, w momencie otrzymania odpowiednich sygnałów aktywowana bądź dezaktywowana jest stosowna zakładka.

Klasa odpowiadająca za wygląd zakładki to `RobotManagementTab`. Dziedziczy ona po klasie `QWidget`. Jej kluczowe metody to:

```
void receivedMsg(const boost::shared_ptr<const gazebo::msgs::Int> &msg);
private slots:
  void on_pushButtonUstaw_clicked();
  void on_pushButtonReset_clicked();
```

3.1.3. Wyniki symulacji

3.2. Pluginy

3.2.1. GUI

3.2.2. World

3.3. ROS + catkin

3.4. Modele

4. Testy

4.1. Przygotowanie

4.2. Obserwacje

5. Wnioski

5.1. Serwer

Bibliografia

[1] *Gazebo – strona internetowa projektu*. http://www.gazebosim.org/.

[2] *ROS – dokumentacja*. http://wiki.ros.org/.

[3] *The Robot Operating System (ROS)* – strona internetowa projektu. http://www.ros.org/.

[4] K. Zadarnowska. Wprowadzenie do systemu operacyjnego ROS: symulator turtlesim. *Instrukcja laboratoryjna*.