Informacje ogólne

- Zadania przedstawione w niniejszej instrukcji należy wykonać **samodzielnie** na systemie operacyjnym Ubuntu 18.04.2 (x64). Hasło użytkownika to `tello`.
- Wykonanie każdego zadania należy **udokumentować zrzutami ekranu obejmującymi cały ekran monitora.** Niezastosowanie się do powyższego będzie równoznaczne z pominięciem zadania przy ocenie.
- Po zakończeniu pracy należy usunąć pliki własne.

Zagadnienia

- środowisko ROS 2 Eloquent Elusor (ang. *Robot Operating System*),
- narzędzie symulacyjne Gazebo 9,
- symulator tello_ros,
- komunikacja z jednostką latającą z poziomu terminala systemowego oraz własnego Node'a.

Materiały pomocnicze

- https://docs.ros.org/en/eloquent/
- https://docs.python.org/3.6/
- http://gazebosim.org/tutorials
- https://github.com/clydemcqueen/tello_ros
- https://www.jetbrains.com/pycharm/download/#section=linux

Wprowadzenie

Usługi w środowisku ROS

Środowisko ROS oferuje cztery podstawowe struktury, które pozwalają na komunikację pomiędzy węzłami. Zostały one przedstawione w dokumencie „*Wprowadzenie do środowiska ROS (I)*". Na szczególną uwagę zasługują dwie najpopularniejsze z nich – *topic'ki* oraz *usługi*. Na potrzeby obsługi tych struktur, wraz ze środowiskiem ROS dostarczane są pakiety, zawierające podstawowe deklaracje oraz opisy. Mowa tu o pakietach: `std_msgs`[1] oraz `std_srvs`[2]. Pierwszy z nich zawiera opisy 30 typów wiadomości, które mogą zostać przesłane pomiędzy węzłami, drugi natomiast definiuje dodatkowo 3 standardowe opisy usług.

Nierzadko zdarza się, że realizacja założonego zadania wymaga utworzenia niestandardowych opisów. ROS pozwala na ich definiowanie, jednak proces ten należy do złożonych i wymaga od użytkownika modyfikacji plików roboczych. Niestandardowe definicje plików wykorzystywanych do komunikacji, odnaleźć można m.in. w bibliotece `tello_ros`. Tym samym, aby nawiązać z nią komunikację, należy wykorzystać te pliki w swoim węźle. Sposób redagowania plików został opisany w dokumentacji ROS'a[3]. Warto wspomnieć, że w ROS2 pliki `*.msg` oraz `*.srv` mogą być budowane tylko i wyłącznie w pakietach zdefiniowanych dla języka C++ – nie ma więc możliwości zdefiniowania tych plików wprost z poziomu projektu utworzonego dla języka Python.

[1] https://index.ros.org/p/std_msgs/#eloquent-assets
[2] https://index.ros.org/p/std_srvs/#eloquent-assets
[3] https://docs.ros.org/en/eloquent/index.html

Proces definiowania plików `*.msg` i/lub `*.srv` należy rozpocząć od utworzenia nowego pakietu, który będzie zawierał niestandardowe komunikaty. Pakiet tworzymy w istniejącej już przestrzeni roboczej (np. `~/ws_test/src`). Można tego dokonać za pomocą komendy:

```
ros2 pkg create --build-type ament_cmake tello_interface
```

która utworzy pakiet o nazwie *tello_interface* zdefiniowany dla języka C++. Następnie, w folderze `tello_interface` warto utworzyć podfoldery `msg` oraz `srv`, które będą przechowywały pliki o odpowiadających im rozszerzeniach. Ma to na celu uporządkowanie struktury danych w pakiecie.

Kolejny krok to utworzenie właściwych plików, zawierających definicje komunikatów oraz opisy usług. Na potrzeby realizacji zadań zawartych w niniejszej instrukcji, utworzony zostanie tylko plik `TelloState.srv` (folder `srv`). Jego zawartość powinna wyglądać następująco:

```
bool request
---
string state
uint8 value
```

Znacznik „---” określa punkt rozdzielenia definicji komunikatu. Linijki występujące powyżej tego znacznika, oznaczają pola żądania wysyłanego przez klienta; linijki poniżej znacznika to pola przewidziane na odpowiedź serwera. Dokładny opis struktury oraz typów danych został umieszczony w dokumentacji[4].

Następny etap to edycja plików `package.xml` oraz `CMakeLists.txt`, znajdujących się w głównym katalogu pakietu. W pliku `package.xml`, za sekcją `export`, dopisać należy poniższe linijki kodu:

```
<build_depend>rosidl_default_generators</build_depend>
<exec_depend>rosidl_default_runtime</exec_depend>
<member_of_group>rosidl_interface_packages</member_of_group>
```

Są one niezbędne do poprawnego procesu budowy pakietu[5], zawierającego pliki `*.msg` oraz`*.srv`. Kolejny krok to edycja pliku `CMakeLists.txt`. Tutaj dopisać należy kod wskazujący na dodatkowe pakiety używane w procesie kompilacji oraz na pliki `*.msg` i `*.srv`, które mają zostać przetworzone.

```
find_package(ament_cmake REQUIRED)
find_package(rosidl_default_generators REQUIRED)
find_package(std_msgs REQUIRED)

rosidl_generate_interfaces(${PROJECT_NAME}
 ”srv/TelloState.srv”
 DEPENDENCIES std_msgs
)
```

Powyższy fragment kodu należy wpisać przed linijką:

```
ament_package()
```

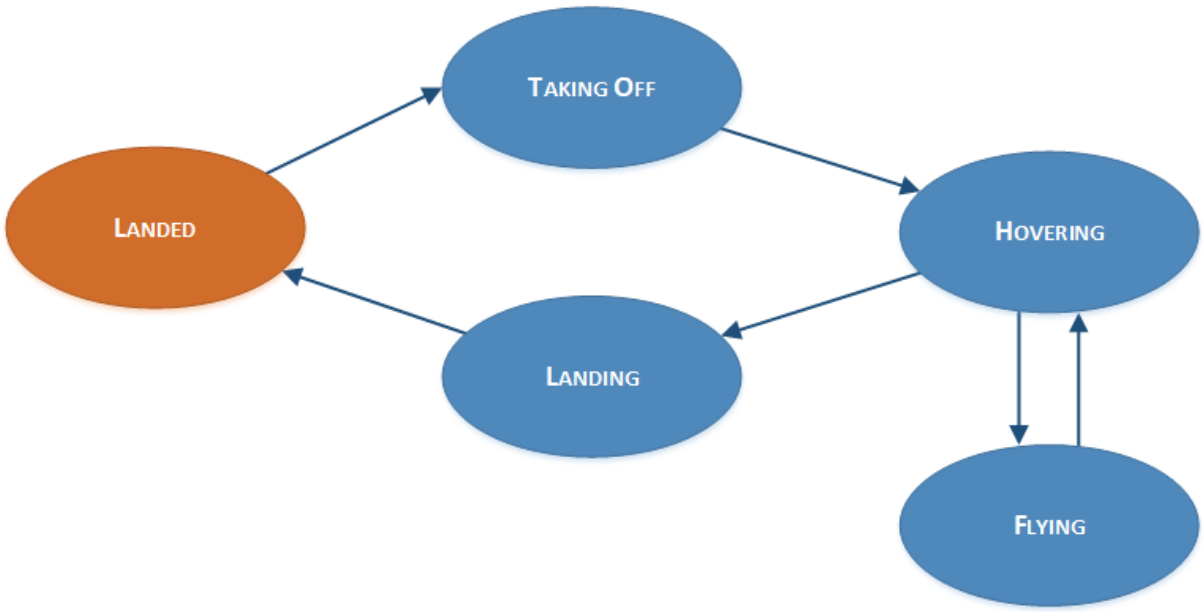
[4] https://docs.ros.org/en/eloquent/Concepts/About-ROS-Interfaces.html
[5] https://docs.ros.org/en/foxy/Concepts/About-Internal-Interfaces.html#the-rosidl-repository

Ostatni etap to przebudowa całej przestrzeni roboczej przy pomocy narzędzia `colcon`. Poprawność utworzenia pakietu zostanie zasygnalizowana w procesie kompilacji.

Tello_ros – model komunikacji

Komunikacja pomiędzy biblioteką `tello_ros` a użytkownikiem odbywa się na dwóch płaszczyznach – komendy *start, lądowanie, stan baterii, lot* są zadawane za pośrednictwem usług, natomiast *pobranie obrazu kamery, dane odometryczne, status wykonania polecenia,* czy *lot* za pomocą *topic'ów*. Jak widać, *lot* może zostać zadany na dwa sposoby.

Istotna jest kwestia koordynacji przesyłanych poleceń. Niektóre z dronów posiadają wewnętrzną maszynę stanów, która pozwala na sterowanie jednostką latającą w zależności od następujących zdarzeń. Tello nie posiada takowej maszyny[6], a tym samym to po stronie programisty spoczywa obowiązek synchronizacji czynności wykonywanych przez drona. Uproszczony model ideowy maszyny stanowej został przedstawiony na rysunku 1.



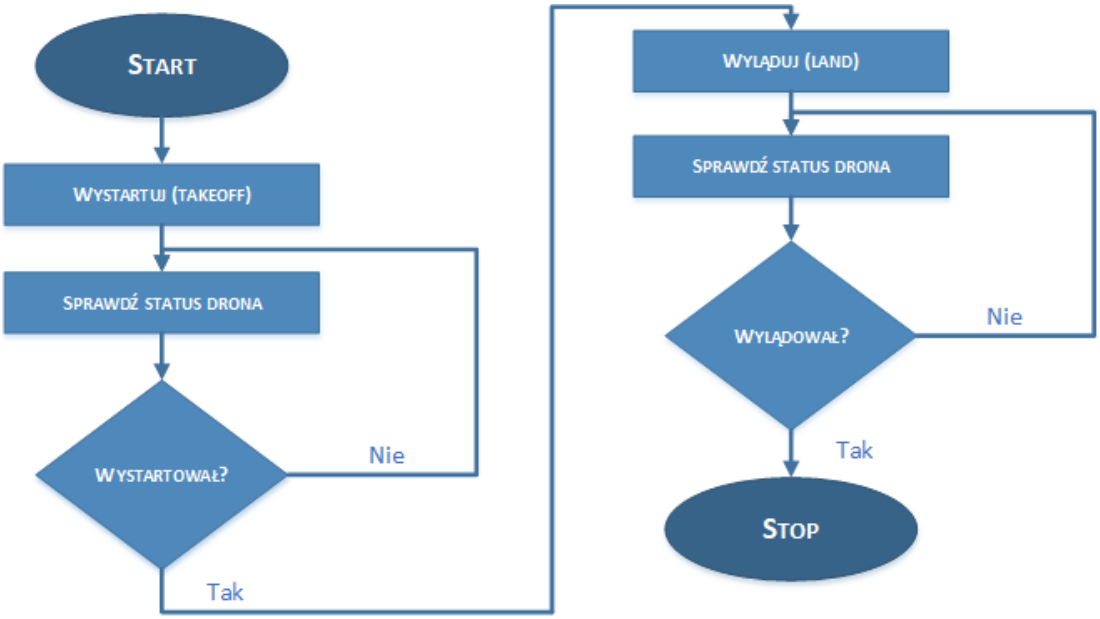
Rysunek 1 – stany wewnętrzne zakładanego modelu

Taki model został zaimplementowany i dołączony do instrukcji w formie pliku `controller.py`. Przeanalizujmy teraz przykładowy model ideowy dla przypadku startu oraz lądowania drona Tello (rysunek 2). Jak widać, po uruchomieniu skryptu i wysłaniu komendy `takeoff`, dron rozpoczyna wznoszenie. Proces ten może trwać różny czas, zależnie od próby, dlatego nie można tutaj wykorzystać licznika. Jedyną opcją jest odpytywanie drona o stan wykonania żądania (*topic* `tello_response`) tak długo, aż otrzymany zostanie komunikatu postaci:

```
rc: 1
str: ok
```

[6] Maszyna stanów jest zaimplementowana w ramach biblioteki `tello_ros`, jednak tylko na użytek wewnętrzny (https://github.com/clydemcqueen/tello_ros/blob/master/tello_gazebo/src/tello_plugin.cpp).

Powyższa informacja wskazuje, że operacja została poprawnie wykonana i można uznać za poprawny start jednostki latającej. Analogicznie wygląda sytuacja dla przypadku lądowania. Co istotne, komunikaty te są zwracane tylko w przypadku wysłania poleceń `takeoff` i `land` – w przypadku komendy `rc`, odpowiedzialnej za ruch, żaden komunikat nie zostanie zwrócony.



Rysunek 2 – schemat blokowy dla przypadku startu oraz lądowania drona Tello

Realizacja skryptu umożliwiającego sterowanie jednostką latającą powinna rozpocząć się od utworzenia nowego pakietu. Tym razem będzie to `tello_controller`.

```
ros2 pkg create --build-type ament_python tello_controller --dependencies rclpy
tello_interface
```

W odróżnieniu od poleceń odnoszących się do poprzednich pakietów, tutaj jako argument wskazano zależności, które powinny zostać dołączone do pliku `package.xml`. Kolejno, w folderze `tello_controller/tello_controller` utworzyć należy plik `controller.py` i przekopiować do niego zawartość udostępnionego pliku, o tej samej nazwie. W przypadku skopiowania całego pliku (a nie tylko jego zawartości), koniecznym może się okazać nadanie niezbędnych uprawnień do pliku (polecenie `chmod`).

Przejdźmy teraz do struktury pliku `controller.py`. Istotna z naszego punktu widzenia jest klasa `ControllerNode`, która odpowiada za utworzenie i zarządzanie węzłem. W ramach tej klasy, zagnieżdżona została klasa `TelloState`, która zawiera „opisy" stanów, jakie może przyjąć jednostka latająca. Są to wartości zgodne z tymi, przedstawionymi na rysunku 1. Zmienne `state` oraz `next_state` wskazują na aktualny i przyszły stan drona. `Action_done` to flaga, określająca czy zlecona misja została już zrealizowana.

W ramach klasy `ControllerNode` zdefiniowano następujące metody:

- `__init__` – konstruktor klasy; tworzy instancje subskrybentów, klientów oraz serwerów usług;
- `state_callback` – generuje odpowiedź na zapytanie o stan wewnętrzny drona;
- `main_callback` – główne wywołanie węzła; dokonuje podstawowej konfiguracji i przekazuje sterowanie do kontrolera;

- `controller` – główny kontroler węzła; zarządza wykonaniem akcji w zależności od obecnego i/lub przyszłego stanu drona;
- `tello_response_callback` – obsługuje odpowiedzi drona z *topic'u* `tello_response`;
- `taking_off_func` – odpowiada za start drona;
- `flying_func` – odpowiada za zlecenie misji do wykonania;
- `mission_func` – zawiera opis misji do wykonania;
- `landing_func` – odpowiada za lądowanie drona.

W skrypcie `controller.py` zrealizowano w pełni obsługę startu drona. Należy pamiętać, aby w przypadku ponowienia próby, wylądować dronem, a następnie zrestartować węzeł (`controller.py`). Jest to krytyczne z uwagi na brak wewnętrznej maszyny stanu w dronie. Przed uruchomieniem i przetestowaniem węzła należy ponadto odpowiednio zmodyfikować plik `setup.py` (patrz: *Wprowadzenie do środowiska ROS (I)*) oraz przebudować przestrzeń roboczą.

Na potrzeby testów, przydatne będą polecenia terminala, związane z komunikacja z usługami. Stan drona można sprawdzić przy pomocy polecenia:

```
ros2 service call /iisrl/tello_state tello_interface/srv/TelloState request:\
true
```

Lądowanie może się odbyć przy pomocy:

```
ros2 service call /drone1/tello_action tello_msgs/srv/TelloAction cmd:\ \'land\
'\
```

Uwagi

- Symulowany dron jest wyposażony w baterię, która również podlega symulacji. W przypadku, gdy zostanie ona rozładowana, niemożliwa będzie dalsza komunikacja z jednostką latającą. Należy wtedy ponownie uruchomić symulator Gazebo.
- Podczas uruchamiania implementowanego węzła lub podczas próby komunikacji z usługą `tello_action`, może pojawić się komunikat błędu dotyczący braku modułu `tello_msgs`. Należy wtedy wykonać polecenie `source ~/tello_ros_ws/install/setup.bash` i ponownie podjąć próbę uruchomienia węzła i/lub komunikacji z usługą.

Zadania szczegółowe

1. Korzystając z przestrzeni roboczej z poprzednich zajęć, utwórz pakiet `tello_interface`, zawierający definicję usługi stanu drona. Zwracana informacja powinna zostać zaprezentowana pod postacią wartości liczbowej oraz tekstu (dwa pola).
2. Utwórz pakiet `tello_controller`, który będzie korzystał z pakietu `tello_interface`. Zdefiniuj skrypt `controller.py` i wypełnij go zawartością pliku, dostępnego w ramach kursu w systemie eKursy.
3. Skonfiguruj odpowiednio pakiet `tello_controller` i uruchom węzeł. Pamiętaj o uprzednim uruchomieniu symulatora.
4. Za pomocą terminala odpytaj węzeł o stan drona. Jaką odpowiedź udało się otrzymać? Na co ona wskazuje?
5. Opublikuj *topic* `/iisrl/tello_controller`, obserwując jednocześnie postęp symulacji oraz odpytując węzeł o stan drona. Co udało się zaobserwować?
6. Zakończ pracę węzła i wyląduj dronem (za pośrednictwem komendy z terminala).
7. Przejdź do pliku `controller.py` i przeanalizuj strukturę metody `taking_off_func`.
8. Uzupełnij ciało metody `landing_func` i przetestuj działanie symulacji. Czy teraz dron poprawnie wystartował i wylądował?
9. Przejdź do metody `main_callback` i zmień wartość `self.action_done` na `False`.
10. Zmodyfikuj ciało metod `flying_func` oraz `mission_func` w taki sposób, aby dron zawisł w powietrzu na 5 sekund[7]. Przetestuj poprawność działania symulacji.
11. Zmodyfikuj ciało metod `flying_func` oraz `mission_func` w taki sposób, aby dron przez 3 sekundy poruszał się wzdłuż osi X (wartość zadana prędkości: `0.1`). Przetestuj poprawność działania symulacji i przeanalizuj zachowanie drona.
12. Przygotuj kod, który pozwoli na lot po trasie, której kształt będzie przypominać kwadrat (czas lotu wzdłuż boku to 2 sekundy). Przetestuj poprawność działania symulacji i przeanalizuj zachowanie drona.
13. Przeanalizuj układ odniesienia drona w stosunku do układu Gazebo. Czy są takie same? Czy układ osi współrzędnych odpowiada sobie nawzajem? Sporządź stosowny rysunek.
14. Brak wbudowanej maszyny stanu powoduje, że w przypadku konieczności ponownego uruchomienia węzła, przyjmujemy on stan domyślny, który nie zawsze oddaje stan właściwy dla jednostki latającej. Zaproponuj oraz zaimplementuj rozwiązanie, które pozwoli na dokonanie korekty stanu (może to mieć wpływ na aktualnie wykonywane zadanie).

[7] Zapoznaj się z metodą `create_timer` (https://docs.ros2.org/crystal/api/rclpy/api/node.html).