ROS2. *First Steps*

Contents

compilat la: 28/04/2025, 11:07

Scopul lucrării

În aceste lucrare de laborator vă veți familiariza cu:

- utilizarea mediului de dezvoltare, simulare și operare a roboților (ROS2) prin configurarea unui *container* în Docker.
- elementele fundamentale ale ROS2 (noduri, topic-uri și servicii)
- aplicarea stategiilor de control clasice pentru un node de tip `turtlesim` pentru execuția automată a unor mișcări ale "țestoasei"

1 ROS - *Robot Operating System*

Roboții, oricât de avansați ar fi din punct de vedere mecanic/electric, au nevoie de partea software (în timp-real) pentru a deveni funcționali și eficienți. Fără o parte software care să proceseze datele de la senzori și să elaboreze și să transmită comenzi precise către efectori (elemente de execuție), roboții ar rămâne doar niște mecanisme inutile, incapabile să îndeplinească task-urile sau scopurile pentru care au fost construiție. Acest laborator introduce conceptul de *middleware* pentru programarea roboților, concentrându-se în special pe ROS2 (Robot Operating System 2), care va fi utilizat ca bază pentru discuțiile și aplicațiile din laboratoarele următoare.

Programarea unui robot presupune mult mai mult decât simpla scriere a unor comenzi. Dezvoltarea software-ului pentru robotică este o activitate extrem de complexă, având în vedere că roboții operează în medii reale, dinamice și adesea imprevizibile. Funcționarea cu succes a unui robot implică integrarea armonioasă cu o gamă variată de senzori, efectori și componente hardware. Încercarea de a programa toate aceste elemente de la zero este o misiune imposibilă, având în vedere complexitatea și diversitatea acestor task-uri, care, deseori, ar depăși rapid eforturile realizabile și ar produce rezultate inconsistente. Aici intervine *middleware*-ul, o componentă software ce se află între sistemul de operare și aplicațiile utilizatorului, simplificând procesul de dezvoltare.

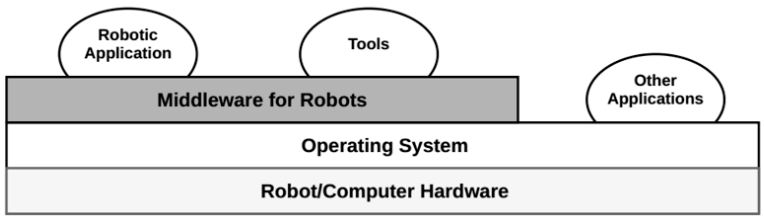


Figure 1: Ideea de *middleware* pentru sistemele robotice[**?**]

Middleware-ul, în special în domeniul roboticii, oferă o serie de instrumente esențiale, inclusiv biblioteci, drivere, instrumente de dezvoltare și metodologii, care

facilitează integrarea și monitorizarea aplicațiilor robotice. De-a lungul timpului, s-au dezvoltat mai multe sisteme middleware pentru programarea roboților, cum ar fi YARP[1], Carmen[2] și Player/Stage[3]. Cu toate acestea, niciunul nu a egalat succesul ROS (Robot Operating System)[4], care a devenit platforma predominantă în cercetarea și dezvoltarea roboticii în ultimii ani.

Acest laborator introduce ROS2, versiunea cea mai recentă a ROS, care se bazează pe fundamentele predecesorului său, aducând caracteristici îmbunătățite și o modularitate sporită, devenind astfel o alegere ideală pentru aplicațiile robotice moderne. Prin ROS2, dezvoltatorii beneficiază de o platformă flexibilă, modulară și susținută de o vastă comunitate, care continuă să se extindă pentru a răspunde provocărilor din robotica actuală.

Atenție!

ROS2 **NU** este un sistem de operare care înlocuiește Linux sau Windows, ci un *middleware* care crește capacitățile unui sistem de operare de a dezvolta aplicații robotice. Numărul 2 indică faptul că este a doua generație a acestui *middleware*.

2 Configurare unei distibuții ROS2 în Docker

Pentru instalarea ROS2, există mai multe variante, fiecare adaptată nevoilor și mediului de lucru al utilizatorului. Cele mai comune metode de instalare sunt:

- **Instalarea nativă prin pachete precompilate** - este, probabil, cea mai simplă pentru cei care lucrează pe un sistem de operare compatibil, precum Ubuntu, deoarece ROS2 este disponibil prin pachetele precompilate pentru majoritatea distribuțiilor Ubuntu și poate fi instalat rapid prin comenzi standard de pachet.
- **Instalarea prin compilarea codul sursă** - pentru cei care doresc o personalizare mai profundă sau care lucrează pe alte distribuții Linux, compilarea ROS2 din codul sursă oferă flexibilitate și control asupra componentelor instalate, însă necesită mai mult timp și atenție pentru rezolvarea dependențelor între pachete.
- **Utilizarea de containere Docker** - o metodă foarte populară și flexibilă pentru instalarea și utilizarea ROS2. Docker oferă o soluție izolată și portabilă, care permite rularea ROS2 într-un container, fără a afecta configurațiile de sistem ale gazdei. Acest lucru este ideal pentru dezvoltatorii care doresc să evite instalările multiple, să protejeze mediul de lucru sau să lucreze pe

[1] https://www.yarp.it/latest/
[2] https://carmen.sourceforge.net/intro.html
[3] https://playerstage.sourceforge.net/
[4] https://ros.org/

sisteme de operare care nu sunt compatibile nativ cu ROS2. Docker permite astfel configurarea rapidă a unui mediu de lucru standardizat, ușor de replicat între echipe sau între mașini diferite. În plus, containerele Docker pentru ROS2 sunt actualizate regulat și menținute de comunitatea ROS, astfel încât utilizatorii au acces la cea mai recentă versiune fără efort suplimentar. Prin Docker, se pot crea și distribui rapid containere standardizate, evitând problemele de compatibilitate și inconsistență între medii diferite. Acest lucru accelerează testarea și implementarea, facilitând dezvoltarea de software robust pentru robotică, într-un mod eficient și ușor de întreținut.

Info: Instalarea ROS2 direct pe Windows și macOS - nerecomandabil

Instalarea ROS2 pe Windows și macOS[a] în mod direct este posibilă, deși implică unele limitări și cerințe specifice, având în vedere că ROS2 a fost dezvoltat inițial pentru Linux. Totuși, comunitatea ROS a făcut progrese considerabile pentru a face ROS2 accesibil și pe alte platforme, oferind astfel mai multă flexibilitate dezvoltatorilor care lucrează în medii diverse.

[a] https://docs.ros.org/en/humble/Installation.html

Pe baza celor de mai sus, pentru laboratoarele de PATR vom opta pentru varianta de instalare prin Docker. Etapele de instalare sunt date pentru Windows, în principal, cu mențiuni unde e nevoie de particularizări pentru MacOS și Linux.

Info: Aplicații necesare

- Instalare Docker Desktop (pentru Windows/Linux/MacOS): https://docs.docker.com/desktop/install/
- Pentru a utiliza interfața grafică a mașinii gazdă sub **Windows** este necesară instalarea și utilizarea aplicației *VcXsrv Windows X Server*: https://sourceforge.net/projects/vcxsrv/

Info: Pre-Setări necesare pentru Windows

De regulă, aceste setări sunt deja făcute "*by default*", dar totusi o verificare poate fi realizată:

- în meniul de start Windows la secțiunea 'Turn Windows features on or off', căsuța 'Windows Subsystem for Linux' trebuie bifată.
- Activare setări virtualizare VT-X (procesor Intel), respectiv AMD-V (procesor AMD). Fără această setare din BIOS fie nu se poate porni mașina virtuală fie va rula foarte lent, deoarece procesorul nu beneficiază de metode de accelerare specifice virtualizării[a]

[a]Pentru instrucțiuni detaliate: https://youtu.be/xqZnjS0h66E sau https://www.ninjaone.com/blog/how-to-enable-cpu-virtualization-in-your-computer-bios/

Pașii necesari instalării unui container Docker sub Windows pentru ROS2 versiunea ***Humble***:

1. Testați dacă Docker este instalat corect, deschizând un terminal ('Start Menu' → *Command Prompt* sau *Windows PowerShell (Admin)* și utilizând comanda: `docker` Această comandă trebuie să afișeze o listă a tuturor comenzilor disponibile pentru `docker`, altfel trebui refăcută instalarea și nu se poate trece la pasul următor.

2. Descărcarea imagini docker pentru ROS-Humble, există 2 variante:

   - dintr-un terminal, prin comanda:
     `docker pull osrf/ros:humble-desktop-full`
   - din Docker Desktop: se cauta în bara de sus *osrf/ros* si se selectează versiunea *humble-desktop-full*

   Pentru a verifica imaginile docker instalate se poate rula într-un terminal `docker images`, care afișează toate imaginile instalat (inclusiv cea cu tag-ul *humble-desktop-full*)

3. Se deschide aplicația **XLaunch** și se setează 'Display number' valoare 0 (v. Figura 2), restul setărilor rămân neschimbate (i.e., cele *default*). Se apăsă pe 'Next' pana la finalizarea setărilor.

   Atenție!

   Setarea aplicației **XLaunch** (**pasul 3**) este necesară la fiecare rularea/deschidere a unui container (**pasul 4**)

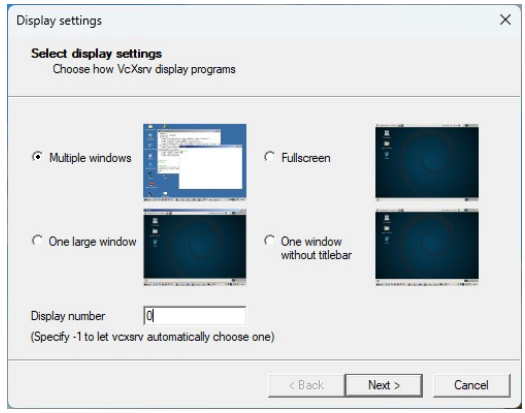


Figure 2: Setare **XLaunch** inainte de rularea container-ului docker.

4. Pentru a porni un container ROS2-humble pe baza imaginii descărcate la **pasul 2** și care să aibă acces la interfața grafică se utilizează pentru prima rulare, într-un terminal comanda:

```
docker run -e DISPLAY=host.docker.internal:0.0 -it osrf/ros:humble-desktop-full
```

5. Se poate continua rulare din terminalul Windows sau se poate folosi într-o manieră mai facilă VSCode[5].

Info: Instrucțiuni pentru VSCode

Pentru utilizarea container-ului în VSCode trebui instalate următoarele extensii specifice (din tab-ul Extensions al VSCode):

- **C/C++ Extension Pack** https://marketplace.visualstudio.com/items?itemName=ms-vscode.cpptools-extension-pack
- **Python Extension Pack** https://marketplace.visualstudio.com/items?itemName=donjayamanne.python-extension-pack
- **Dev Containers Extension** - pentru a permite deschiderea VSCode într-un container. După instalare va apărea în VSCode un tab în care apar toate containerele active și se selecteaza cel creat la **pasul 4**

6. La prima rulare a container-ului este recomandată rularea următoarele comenzi:

[5] https://code.visualstudio.com/

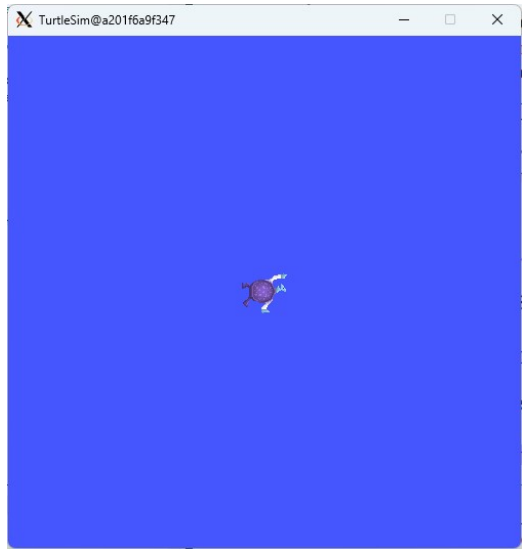


Figure 3: Simulare `turtlesim`

```
sudo apt-get update
source /opt/ros/humble/setup.sh
```

7. Pentru a testa că ROS2 este instalat corect putem rula un demo de tipul:

   ```
   ros2 run demo_nodes_cpp talker
   ```

   sau pentru a verifica și accesul la interfața grafică:

   ```
   ros2 run turtlesim turtlesim_node
   ```

8. Cea de-a doua comandă va deschide o fereastră ce conține un *turtle* ca în figura 4.

3 Concepte fundamentale ROS2

3.1 Context

ROS 2 se bazează pe conceptul de combinare a spațiilor de lucru (*workspaces*) utilizând terminal(linia de comandă, *shell*). Termenul spațiu de lucru (*workspace*) desemnează în ROS locația din sistem în care sunt dezvoltate de utilizator aplicații utilizând ROS2. Spațiul de lucru de bază al ROS 2 este denumit *underlay*, în timp ce spațiile de lucru locale ulterioare sunt numite *overlays*. De regulă, în procesul de dezvoltare cu ROS2, există mai multe spații de lucru active simultan.

Combinarea acestor spațiilor de lucru facilitează dezvoltarea folosind versiuni diferite ale ROS2 sau utilizarea unor grupuri distincte de pachete (*packages*). De asemenea, această metodă permite instalarea mai multor distribuții ROS2 (denumite, pe scurt *distros*,e.g., Dashing, Eloquent, Foxy sau Humble) pe aceeași mașină(sistem, de calcul) și comutarea între aceste distribuții.

Această funcționalitate este realizată prin adăugarea fișierelor sursă de configurare în spațiul de căutare la fiecare deschidere a unui nou terminal sau prin adăugarea acestor fișiere în script-ul de inițializare al terminalelor (e.g., `/.bashrc`). Fără această rulare a fișierelor de configurare, nu se vor putea accesa comenzile ROS2, și nici identifica sau utiliza pachetele specifice ROS2. Altfel spus, fără această preconfigurare nu se va putea rula ROS2.

Info

Pentru a parcurge următoarele secțiuni, asigurați-vă că dețineți/ aveți acces la o instalare corectă a ROS2-humble (v. **secțiunea 2**)

3.2 Configurare mediului de lucru

Pentru a putea accesa comenzile specifice ROS2, la deschiderea unui terminal trebui rulată comanda:

```
source /opt/ros/humble/setup.sh
```

Totuși, pentru a evita a repeta acest lucru la fiecare nou terminal se poate utiliza:

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

care adaugă respectiva comandă în script-ul ce rulează la deschiderea unui terminal.

Următorul pas constă în verificarea variabilelor specifice mediului de lucru ROS2, iar acest aspect se realizează prin comanda:

```
printenv | grep -i ROS
```

care ar trebui să returneze:

```
ROS_VERSION=2
ROS_PYTHON_VERSION=3
ROS_DISTRO=humble
```

Atenție!

Dacă variabilele nu sunt configurate corect, i.e. nu coincid cu valorile de mai sus, reveniți la instalarea pachetelor ROS 2 din **Laborator 6** sau *ask the ROS community*[a].

[a] https://robotics.stackexchange.com/

Info

Dacă întâmpinați vreodată dificultăți în localizarea sau utilizarea pachetelor în ROS2, primul pas ar trebui să fie verificarea variabilelor descrse mai sus și asigurarea că acestea sunt configurate pentru versiunea și distribuția pe care o utilizați.

3.3 Utilizarea `turtlesim`, `ros2` și `rqt`

`Turtlesim` este un simulator simplu, conceput pentru a facilita învățarea ROS2. Acesta ilustrează funcționalitățile de bază, oferind o imagine generală asupra activităților care vor putea fi desfășurate ulterior cu un robot real sau simulat (de regulă, în Gazebo).
Instrumentul/comanda **`ros2`** reprezintă modul principal prin care utilizatorii gestionează, inspectează și interacționează cu un sistem de tip ROS. Acesta oferă o gamă variată de comenzi ce acoperă diferite aspecte ale sistemului și funcționării acestuia. De exemplu, poate fi utilizat pentru a porni un nod, a seta un parametru, a asculta un topic etc. Comanda **`ros2`** face parte din pachetul de bază a ROS2 (i.e. este inclusă în `/opt/ros/humble/setup.bash`).
`rqt` este o interfață grafică (**GUI**) pentru ROS2. Toate operațiunile disponibile în rqt pot fi realizate și din linia de comandă, însă **`rqt`** oferă o metodă mai accesibilă de manipulare a elementelor ROS2.

Pentru a ne asigura ca cele 3 instrumente funcționeaza corect verificați urmatoarele comenzi:

1. Deschideți un terminal în container-ul generat la **Laboratorul 6**
2. Verificați existența pachetelor `turtlesim` prin comanda:

   ```
   ros2 pkg executables turtlesim
   ```

   care ar trebui să returneze:

   ```
   turtlesim draw_square
   turtlesim mimic
   turtlesim turtle_teleop_key
   turtlesim turtlesim_node
   ```

   Info

   Instalarea pachetelor specifice `turtlesim` se face prin:

   ```
   sudo apt update
   sudo apt install ros-humble-turtlesim
   ```

3. Porniți o simulare prin comanda: `ros2 run turtlesim turtlesim_node`

Se va deschide o fereastră ca in figura 4, iar în terminal veți observa numele (`turtle1`) și coordonatele obiectului de tip *turtle*:

```
[INFO] [turtlesim]: Starting turtlesim with node name /turtlesim
[INFO] [turtlesim]: Spawning turtle [turtle1] at x=[5.544445], y
    =[5.544445], theta=[0.000000]
```

4. Deschideți un nou terminal și rulați următoarea comandă:

   ```
   ros2 run turtlesim turtle_teleop_key
   ```

   În acest moment, ar trebui să aveți deschise trei ferestre: un terminal în care rulează `turtlesim_node`, un terminal în care rulează `turtle_teleop_key` și fereastra `turtlesim`. Aranjați aceste ferestre astfel încât să puteți vedea fereastra `turtlesim`, dar și să aveți activ terminalul în care rulează texttt-turtle_teleop_key, pentru a putea controla mișcarea obiectului din fereastra `turtlesim` prin utilizarea tastaturii "Țestoasa" se va deplasa pe ecran, utilizând "stiloul" atașat pentru a desena traseul parcurs până la momentul respectiv.

5. Verificați dacă `rqt` este instalat, rulând într-un nou terminal comanda: `rqt`

   Info

   Instalare `rqt` se realizeaza prin comenzile:

   ```
   sudo apt update
   sudo apt install ' nros-humble-rqt*'
   ```

6. Când rulați `rqt` pentru prima dată, fereastra va fi goală. Trebuie să selectați din bara de meniu opțiunea **Plugins** → **Services** → **Service Caller**

   Info

   Este posibil ca `rqt` să necesite ceva timp pentru a localiza toate plugin-urile. Dacă accesați **Plugins**, dar nu vedeți opțiunea **Services** sau alte opțiuni, închideți rqt și introduceți comanda `rqt --force-discover` în terminal.

7. Utilizați butonul de **Refresh** aflat în stânga listei de tip *dropdown* din dreptul **Service** pentru a vă asigura că toate serviciile asociate nodului `turtlesim` sunt disponibile.

8. Faceți clic pe lista **Service** pentru a vizualiza serviciile asociate cu `turtlesim` și selectați serviciul `/spawn`.

9. Acest `/spawn` va genera un nou obiect de tip `turtle` în aceeași fereastră (deja deschisă). Setați parametrii din Figura 6 cu ce valori vreți, după care butonul **Call**

   Atenție!

   Dacă încercați să generați o nouă "țestoasă" cu același nume ca "țestoasa" existentă, cum ar fi numele implicit turtle1, veți primi un mesaj de eroare în terminalul în care rulează `turtlesim_node`: `[ERROR] [turtlesim]: A turtle named [turtle1] already exists`

10. Dacă **Call**-ul a fost realizat cu succes, o nouă "țestoasă" (cu un design aleatoriu) va apărea la coordonatele introduse.

11. Dacă actualizați lista de servicii în `rqt` folosind butonul de **Refresh**, veți observa că acum există servicii asociate cu noua "țestoasă", de forma `/turtle2/...`, pe lângă cele pentru `/turtle1/...`

12. Selectați serviciul `/turtle1/set_pen`, schimbați valorile **`r, g, b`** și/sau **`width`** și apăsăți pe **Call**. Ce observați când dați o nouă comandă către `/turtle1/`?

13. Pentru a mișca și cea de a doua "țestoasă" va fi necesar un nou terminal în care să rulați comanda:

    ```
    ros2 run turtlesim turtle_teleop_key
    ```

    Ce observați? Se mișcă și cea de a doua "țestoasă"? **Nu**. Pentru aceasta va fi necesara adăugarea unor argumente comenzii anterioare:

    ```
    ros2 run turtlesim turtle_teleop_key --ros-args --remap turtle1/cmd_vel:=turtle2/cmd_vel
    ```

    care *remapeaza* comenzile de viteză pe cel de al doilea obiect.

14. *Clean up*: Închideți toate terminalele deschise, nu înainte de a opri execuția prin `Ctrl+C`.

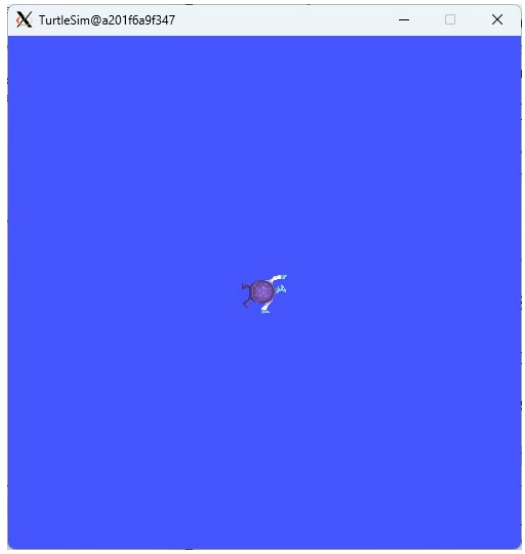


Figure 4: Simulare `turtlesim`

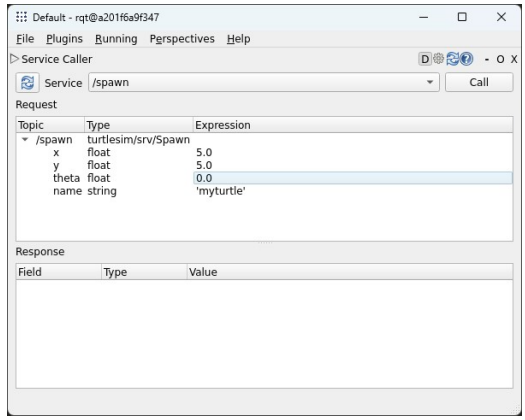


Figure 5: `rqt` Service Caller `/spawn`

Figure 6: Simulare `turtlesim` după apelul *service*-ului `/spawn`

3.4 Graful computațional

În cele ce urmează vom prezenta conceptele fundamental utilizate în cadrul ROS(2): Noduri(*Nodes*), Topic-uri (*Topics*), Servicii(*Services*), Parametri(*Parameters*) și Acțiuni(*Actions*). Toate acestea formează ceea ce in comunitatea ROS se numește "***ROS(2) graph***" sau "***Computational graph***" . Indiferent de denumirea folosită, acest graf, ca orice graf, este compus din vârfuri și arce. Dacă arcele modelează legăturile de comunicație (uni-/bidirecționale) între elementele fundamentale din ROS 2, nodurile sunt chiar aceste elementele fundamentale care compun o aplicație ROS(2). În plus, particularitatea grafului ROS(2) este că are o componentă de monitorizare , i.e., noduri inserate în acest graf al căror rol este strict monitorizarea comunicatiei și legăturilor între acele elemente fundamentale.

3.5 Nodurile în ROS2

Se recomandă ca fiecare nod din graful ROS(2) să fie responsabil pentru un scop precis, unic și modular, de exemplu, controlul motoarelor roților sau citirea datelor de la senzor. Fiecare nod poate trimite și primi date de la alte noduri prin intermediul topic-urilor (topics), serviciilor (*services*), acțiunilor (*actions*) sau parametrilor.
Un sistem robotic complet este alcătuit din multe noduri care lucrează împreună. În ROS 2, un singur executabil (program C++, program Python etc.) poate conține unul sau mai multe noduri. Pentru a rula un astfel de executabil (e.g., cum am făcut mai sus cu `turtlesim_node` sau cu `turtle_teleop_key` din pachetul `turtlesim`) utilizăm comanda:

```
ros2 run <package_name> <executable_name>
```

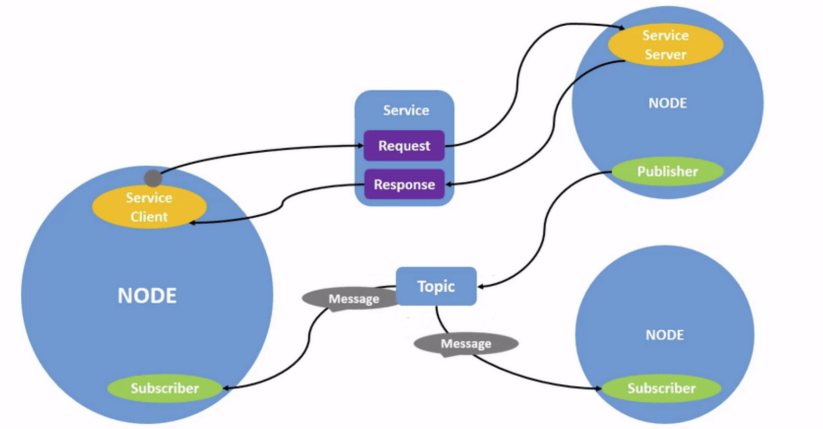


Figure 7: Exemplu de graf ROS2

Această comandă nu furnizează informații despre nodurile create prin rularea executabilului. În acest scop există altă comandă:

```
ros2 node list
```

care va afișa numele tuturor nodurilor aflate în execuție. Această comandă este foarte utilă atunci când se dorește interacțiunea cu un anumit nod sau există mai multe noduri care sunt rulate în/de sistem, noduri care trebuie monitorizate.

Pentru a observa cum sunt utilizate nodurile în ROS2 și comenzile de monitorizare a lor, încercați rularea următorilor pași:

1. Deschideți un terminal și rulați `ros2 run turtlesim turtlesim_node`
2. Deschideți un nou terminal și rulați `ros2 node list`. Aceasta comandă ar trebui să returneze:

```
1
        /turtlesim
2
```

3. Deschideți încă un nou terminal și rulați `ros2 run turtlesim turtle_teleop_key`
4. Întoarceți-vă la terminalul de la 2 și re-rulați `ros2 node list`. Acum, comanda ar trebui să returneze:

```
1
        /turtlesim
        /teleop_turtle
3
```

5. Folosind remaparea (i.e., parametri `--ros-args` la comanda `ros2 run`), rulați dintr-un al 4-lea terminal:

```
ros2 run turtlesim turtlesim_node --ros-args --remap __node:=my_turtle
```

Această comandă va deschide o nou fereastră de tip `turtlesim` și asocia un numele `my_turtle` cu nodul asociat.

6. Întoarceți-vă la terminalul de la 2 și re-rulați `ros2 node list`. Ce observați?

7. În același terminal rulati `ros2 node info /my_turtle`. Aceasta comandă va afișa toate informațiile disponibile despre nodul `my_turtle`, i.e., listele de *subscribers*, *publishers*, servicii și acțiuni, pe scurt, conexiunile care interacționează cu nodul în cadrul grafului ROS2.

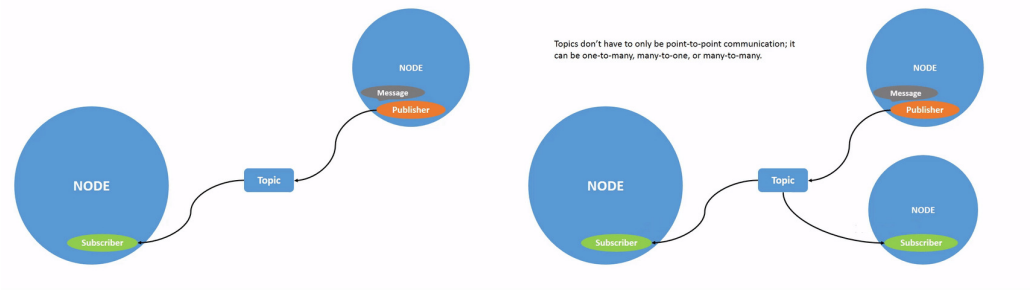
```
/my_turtle
  Subscribers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /turtle1/cmd_vel: geometry_msgs/msg/Twist
  Publishers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /rosout: rcl_interfaces/msg/Log
    /turtle1/color_sensor: turtlesim/msg/Color
    /turtle1/pose: turtlesim/msg/Pose
  Service Servers:
    /clear: std_srvs/srv/Empty
    /kill: turtlesim/srv/Kill
    /my_turtle/describe_parameters: rcl_interfaces/srv/
    DescribeParameters
    /my_turtle/get_parameter_types: rcl_interfaces/srv/
    GetParameterTypes
    /my_turtle/get_parameters: rcl_interfaces/srv/GetParameters
    /my_turtle/list_parameters: rcl_interfaces/srv/ListParameters
    /my_turtle/set_parameters: rcl_interfaces/srv/SetParameters
    /my_turtle/set_parameters_atomically: rcl_interfaces/srv/
    SetParametersAtomically
    /reset: std_srvs/srv/Empty
    /spawn: turtlesim/srv/Spawn
    /turtle1/set_pen: turtlesim/srv/SetPen
    /turtle1/teleport_absolute: turtlesim/srv/TeleportAbsolute
    /turtle1/teleport_relative: turtlesim/srv/TeleportRelative
  Service Clients:
  Action Servers:
    /turtle1/rotate_absolute: turtlesim/action/RotateAbsolute
  Action Clients:

```

8. Încercati aceeași comandă și pentru nodul `/teleop_turtle`. Care sunt conexiunile cu nodul `my_turtle`?

3.6 Topic-urile în ROS2

ROS 2 descompune sistemele complexe în multiple noduri modulare. Topic-urile (*topics*) reprezintă un element fundamental al grafului ROS, funcționând ca un fel de magistrale de comunicație prin care nodurile pot face schimb de mesaje. Topic-



(a) Topic 1-la-1 (b) Topic 1-la-2

Figure 8: Topic-uri ROS2

urile reprezintă una dintre principalele modalități prin care datele sunt transferate între noduri și, implicit, între diferitele părți ale sistemului.

După cum se poate observa și în figura 8, un nod poate "publica" date către oricâte topic-uri și, simultan, poate avea "abonamente" la oricâte topic-uri.

Pentru a înțelege modul de interacționare cu și prin topic-uri, încercați utilizarea următoarelor comenzi:

1. Deschideti 2 terminale (dacă nu sunt deja deschise). Într-unul rulați `ros2 run turtlesim turtlesim_node`, iar în celălalt, `ros2 run turtlesim turtle_teleop_key`

2. Deschideti un nou terminal și utilizați comanda `rqt_graph`. Sau utilizați `rqt`, și apoi selectați **Plugins** → **Introspection** → **Node Graph.** Ce observați?

3. Deschideți un nou terminal și rulați comanda `ros2 topic list`. Această comandă ar trebui să returneze:

```
/parameter_events
/rosout
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
```

4. În același terminal și rulați comanda `ros2 topic list - t`. Această comandă ar trebui să returneze:

```
/parameter_events [rcl_interfaces/msg/ParameterEvent]
/rosout [rcl_interfaces/msg/Log]
/turtle1/cmd_vel [geometry_msgs/msg/Twist]
/turtle1/color_sensor [turtlesim/msg/Color]
/turtle1/pose [turtlesim/msg/Pose]
```

Pe lângă topic-uri sunt afișate și atributele acestora, atribute (`types`) care arată structura mesajelor între noduri și modul în care informația transmisă este referențiată de acele noduri. Toate aceste atribute pot fi vizualizate și în `rqt_graph` prin deselectarea casuțelor de lângă **Hide**.

5. Pentru a vedea datele care sunt transmise printr-un topic se utilizează `ros2 topic echo <topic_name>`. Încercați comanda pentru topic-ul `/turtle1/cmd_vel`.

6. Reveniți la terminalul în care rulează `turtle_teleop_key` și utilizați tastatura pentru a deplasa "țestoasa". Urmăriți, în același timp, terminalul unde rulează comanda `echo`, și veți observa că datele de poziție sunt publicate pentru fiecare mișcare pe care o efectuați.

7. Într-un nou terminal rulați pe rând comenzile:
   a) `ros2 topic info /turtle1/cmd_vel` . Ce observați? Ce tip de comunicație are topic-ul `cmd_vel` ?
   b) `ros2 interface show geometry_msgs/msg/Twist`. Această comanda afișează structura mesajelor trimise prin respectivul topic. Ce remarcați în raport cu ce returnează echo `ros2 topic echo`?

8. Pe baza structurii/formatului mesajelor dintr-un topic putem "publica" measje pe acel topic direct din linia de comandă prin:

```
ros2 topic pub <topic_name> <msg_type> '<args>'
```

Argumentul `<args>` reprezintă datele efective care sunt transmise topic-ului, structurate conform formatului afișat în urma rulării comenzii de la punctul anterior `ros2 interface show <msg_type>`

"Țestoasa" (și, în mod obișnuit, roboții reali pe care aceasta îi emulează) necesită un flux constant de comenzi pentru a funcționa în mod continuu. Astfel, pentru a pune "țestoasă" în mișcare și pentru a o menține în mișcare, puteți utiliza următoarea comandă. Este important de reținut că aceste argumente trebuie introdus utilizând sintaxa YAML. Astfel, comanda completă ar fi:

```
ros2 topic pub /turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x:
2.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 1.8}}"
```

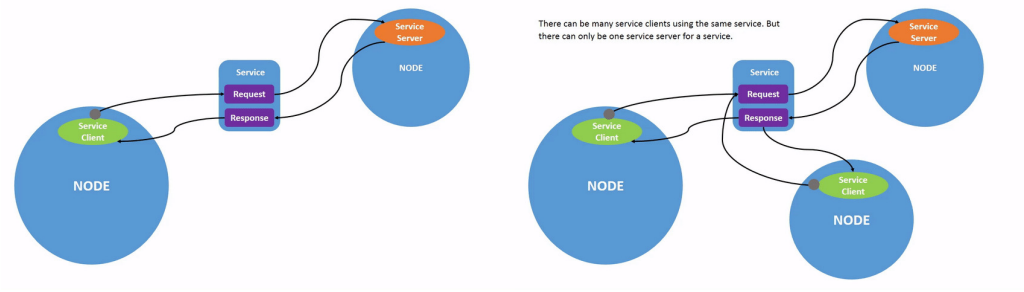
Fără opțiuni în linia de comandă, comanda `ros2 topic pub` "publică" mesajul într-un flux constant la o frecvență de 1 Hz. Pentru a "publica" mesajul o singură dată, trebuie adăugată opțiunea `--once`:

```
ros2 topic pub --once -w 2 /turtle1/cmd_vel geometry_msgs/msg/Twist "{linear:
{x: 2.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 1.8}}"
```

Opțiunea `-w 2` este opțională, înseamnă "wait for two matching subscriptions" și este necesară deoarece avem și terminalul cu `echo` care e atașat acestui topic.

Opțional, puteți verifica modificările și `rqt_graph`

9. Alte comenzi utile, pe care le puteți încerca și observa efectele:
   a) `ros2 topic hz /turtle1/pose`
   b) `ros2 topic bw /turtle1/pose`
   c) `ros2 topic find <topic_type>` sau `ros2 topic find geometry_msgs/msg/Twist`
10. **To Do [opțional]** Repetați pași de mai sus (începand cu pasul **6**) pentru `/turtle1/pose`,
11. *Clean up*: Închideți toate terminalele deschise, nu înainte de a opri execuția prin `Ctrl+C`.



(a) Serviciu 1-la-1 (b) Serviciu 1-la-2

Figure 9: Servicii ROS2 - paradigma *call-and-response*

3.7 Servicii în ROS2

Serviciile (*services*) reprezintă o altă metodă de comunicare între noduri în graful ROS. Acestea se bazează pe un model de tip *call-and-response*, spre deosebire de modelul *publisher-subscriber*/client-server utilizat de topic-uri. În timp ce topic-urile permit nodurilor să se aboneze la fluxuri de date și să primească actualizări continue, serviciile furnizează date doar atunci când sunt apelate în mod explicit de un client.

1. Deschideti 2 terminale (dacă nu sunt deja deschise). Într-unul rulați `ros2 run turtlesim turtlesim_node`, iar în celălalt, `ros2 run turtlesim turtle_teleop_key`

2. Deschideți un nou terminal și rulați comanda `ros2 service list`. Această comandă ar trebui să returneze:

```
/clear
/kill
/reset
/spawn
/teleop_turtle/describe_parameters
/teleop_turtle/get_parameter_types
/teleop_turtle/get_parameters
/teleop_turtle/list_parameters
/teleop_turtle/set_parameters
/teleop_turtle/set_parameters_atomically
/turtle1/set_pen
/turtle1/teleport_absolute
/turtle1/teleport_relative
/turtlesim/describe_parameters
/turtlesim/get_parameter_types
/turtlesim/get_parameters
/turtlesim/list_parameters
/turtlesim/set_parameters
/turtlesim/set_parameters_atomically
```

De remarcat faptul că ambele noduri (`/teleop_turtle` și `/turtle1`) au aceleași **6** servicii, ce conțin `_parameters_` în nume. De altfel, aproape fiecare nod din ROS 2 dispune de aceste servicii de infrastructură pe care se bazează parametrii (mai pe larg în următoarea secțiune, acum vom omite acest 6 servicii)

În cele ce urmează ne concentrăm pe serviciile specifice `turtlesim`: `/clear`, `/kill`, `/reset`, `/spawn`, `/turtle1/set_pen`, `/turtle1/teleport_absolute`, și `/turtle1/teleport_relative`. Am interacționat cu unele dintre acestea utilizând `rqt` în prima secțiune a acestui laborator.

3. Serviciile au tipuri (*types*) care descriu modul în care sunt structurate datele de solicitare și răspuns ale unui serviciu. Tipurile de servicii sunt definite într-un mod similar cu tipurile de topic-uri (*topic types*), cu excepția faptului că tipurile de servicii au două părți: un tip (de structura) pentru solicitare și un altul pentru răspuns.

   Pentru a afla tipul unui serviciu, utilizați comanda: `ros2 service type <service_name>`. Rulați, de exemplu, într-un terminal `ros2 service type /clear`.

   De asemenea se poate utiliza `ros2 service list -t` pentru a identifica atributele/ tipurile pentru toate serviciile:

```
/clear [std_srvs/srv/Empty]
/kill [turtlesim/srv/Kill]
/reset [std_srvs/srv/Empty]
/spawn [turtlesim/srv/Spawn]
...
/turtle1/set_pen [turtlesim/srv/SetPen]
/turtle1/teleport_absolute [turtlesim/srv/TeleportAbsolute]
/turtle1/teleport_relative [turtlesim/srv/TeleportRelative]
...
```

4. Ca și pentru topic-uri se pot utiliza comenzi ca:

   a) `ros2 service find <type_name>`

   b) `ros2 interface show <type_name>`

   Încercați aceste comenzi pentru unul din tipurile/atributele returnate la pasul anterior.

5. Un serviciu poate fi apelat și direct din linia de comanda (nu doar din `rqt`, ca în prima secțiune). Pentru aceasta folosim:

   ```
   ros2 service call <service_name> <service_type> <arguments>
   ```

   Partea `<arguments>` este opțională, de exemplu, rulați `ros2 service call /clear std_srvs/srv/Empty`. Ce efect are această comandă?

6. Acum, să generăm o nouă "țestoasă" apelând serviciul `/spawn` și setând argumentele într-un mod adecvat. Ca și la topic-uri, introducerea argumentelor într-un apel de serviciu din linia de comandă trebuie să folosească sintaxa YAML.

   Comanda pentru a apela serviciul `/spawn` ar fi următoarea:

   ```
   ros2 service call /spawn turtlesim/srv/Spawn "{x: 2.0, y: 3.0, theta:
   0.0, name: 'turtle2'}"
   ```

   unde `<arguments>` date sunt coordonatele `x` și `y`, unghiul de orientare `theta`, și numele "țestoasei"

7. *Clean up*: Închideți toate terminalele deschise, nu înainte de a opri execuția prin `Ctrl+C`.

Atenție!

În general, **nu este recomandat** să folosiți un serviciu pentru apeluri continue; topic-urile sau chiar acțiunile (actions) sunt mai potrivite pentru astfel de cazuri.
Serviciile sunt destinate interacțiunilor de tip solicitare-răspuns care au loc o singură dată, în timp ce topic-urile și acțiunile sunt mai adecvate pentru comunicația continuă sau pe termen lung.

3.8 Parametri în ROS2

Un parametru este o valoare de configurare a unui nod. Puteți considera parametrii ca fiind setările nodului. Un nod poate stoca parametri sub formă de întregi, numere reale, booleene, șiruri de caractere și liste. În ROS 2, fiecare nod își gestionează propriii parametri.

Pentru a înțelege modul de gestionare și utilizarea a parametrilor, încercați utilizarea următoarelor comenzi:

1. Deschideti 2 terminale (dacă nu sunt deja deschise). Într-unul rulați `ros2 run turtlesim turtlesim_node`, iar în celălalt,
   `ros2 run turtlesim turtle_teleop_key`

2. Pentru a vizualiza parametrii care aparțin nodurilor activate la pasul 1, deschideți un nou terminal și introduceți comanda: `ros2 param list`. Aceasta va afișa în terminal:

```
/teleop_turtle:
  qos_overrides./parameter_events.publisher.depth
  qos_overrides./parameter_events.publisher.durability
  qos_overrides./parameter_events.publisher.history
  qos_overrides./parameter_events.publisher.reliability
  scale_angular
  scale_linear
  use_sim_time
/turtlesim:
  background_b
  background_g
  background_r
  qos_overrides./parameter_events.publisher.depth
  qos_overrides./parameter_events.publisher.durability
  qos_overrides./parameter_events.publisher.history
  qos_overrides./parameter_events.publisher.reliability
  use_sim_time
```

   Fiecare nod are parametrul `use_sim_time`, care nu este specific doar pentru `turtlesim`. Pe baza denumirii parametrilor, se observă că parametrii de la `/turtlesim` determină culoarea de fundal a ferestrei **turtlesim** utilizând valori de culoare RGB.

   Pentru a determina tipul unui parametru, se utilizează comanda `ros2 param get`. De exemplu:

   ```
   ros2 param get /turtlesim background_g
   ```

   Rulați aceeași comanda și pentru ceilalți parametri.

3. Pentru a seta paramaterii utilizăm:

   `ros2 param set <node_name> <parameter_name> <value>`.

Setați valoare pentru `background_r` la o altă valoare. Care este efectul?

4. Pentru a vizualiza toate valorile parametrilor curenți ale unui nod se utilizează comanda: `ros2 param dump <node_name>`. Comanda va afișa valorile parametrilor în `stdout` (ieșirea standard -terminalul) în mod implicit, dar există posibilitatea, de asemenea, de a redirecționa valorile parametrilor într-un fișier. De exemplu, pentru a salva configurația curentă a parametrilor nodului `/turtlesim` într-un fișier `turtlesim.yaml`, introduceți comanda:
   `ros2 param dump /turtlesim > turtlesim.yaml`

   Se va genera astfel un fișier nou `turtlesim.yaml` în directorul curent, în care rulează terminalul. Dacă deschideți acest fișier, veți vedea următorul conținut:

```
/turtlesim:
  ros__parameters:
    background_b: 255
    background_g: 86
    background_r: 150
    qos_overrides:
      /parameter_events:
        publisher:
          depth: 1000
          durability: volatile
          history: keep_last
          reliability: reliable
    use_sim_time: false
```

5. Puteți încărca parametrii dintr-un fișier într-un nod care rulează deja utilizând comanda:

   `ros2 param load <node_name> <parameter_file>`

   De exemplu, pentru a încărca fișierul `turtlesim.yaml` generat cu comanda `ros2 param dump` în parametrii nodului `/turtlesim`, introduceți comanda:

   `ros2 param load /turtlesim turtlesim.yaml`

   Modificați fișierul și reîncărcați. La o încărcare reușită, mesajul din terminal este:

```
Set parameter background_b successful
Set parameter background_g successful
Set parameter background_r successful
Set parameter qos_overrides./parameter_events.publisher.depth
    failed: parameter 'qos_overrides./parameter_events.publisher.
    depth' cannot be set because it is read-only
```

```
Set parameter qos_overrides./parameter_events.publisher.durability
    failed: parameter 'qos_overrides./parameter_events.publisher.
    durability' cannot be set because it is read-only
Set parameter qos_overrides./parameter_events.publisher.history
    failed: parameter 'qos_overrides./parameter_events.publisher.
    history' cannot be set because it is read-only
Set parameter qos_overrides./parameter_events.publisher.
    reliability failed: parameter 'qos_overrides./parameter_events
    .publisher.reliability' cannot be set because it is read-only
Set parameter use_sim_time successful
```

Info

Parametrii *read-only* pot fi modificați doar la pornirea nodului și nu ulterior, de aceea apar unele avertismente pentru parametrii `"qos_overrides"`. Acești parametri sunt setați doar la inițializarea nodului și nu pot fi modificați dinamic după ce nodul este deja în execuție, ceea ce explică de ce nu se permit modificări ulterioare la aceștia.

6. Pentru a porni același nod utilizând valorile de parametri salvate, utilizați comanda:

```
ros2 run <package_name> <executable_name> --ros-args --params-file <file_name>
```

Practic, aceeași comandă care se folosește pentru a porni `turtlesim`, cu adăugarea flag-urilor `--ros-args` și `--params-file`, urmate de fișierul ce conține valorile parametrilor cu tructura YAML.

Opriți nodul `turtlesim` care rulează în prezent și încercați să-l reîncărcați cu parametrii salvați, utilizând:

```
ros2 run turtlesim turtlesim_node --ros-args --params-file turtlesim.yaml
```

Info

Atunci când un fișier de parametri este folosit la pornirea nodului, toți parametrii, inclusiv cei *read-only*, vor fi actualizați.

7. *Clean up*: Închideți toate terminalele deschise, nu înainte de a opri execuția prin `Ctrl+C`.

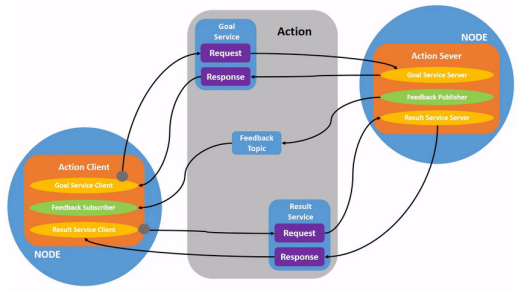


Figure 10: Action în ROS2

3.9 Acțiuni (actions) în ROS2

Acțiunile (*Actions*) sunt un tip de comunicare în ROS 2 și sunt destinate task-urilor de lungă durată. Acestea constau în trei părți: un scop (*goal*), feedback și rezultatul (*result*) (v. Figura 10). Acțiunile sunt construite pe baza topic-urilor și serviciilor. Funcționalitatea lor este similară cu serviciile, cu excepția faptului că acțiunile pot fi anulate. De asemenea, ele oferă feedback constant, spre deosebire de servicii care returnează un singur răspuns.

Acțiunile folosesc un model client-server, similar modelului *publisher-subscriber* (descris în secțiunea despre topic-uri). Un nod "client-acțiune" trimite un scop (*goal*) unui nod "server-acțiune", care recunoaște scopul și returnează un **flux** de feedback și un rezultat.

1. Deschideti 2 terminale (dacă nu sunt deja deschise). Într-unul rulați `ros2 run turtlesim turtlesim_node`, iar în celălalt,
   `ros2 run turtlesim turtle_teleop_key`

2. Când nodul `/teleop_turtle`, în terminal va apărea următorul mesaj:

```
Use arrow keys to move the turtle.
Use G|B|V|C|D|E|R|T keys to rotate to absolute orientations. 'F'
    to cancel a rotation.
```

   Dacă prima linie se referă la topic-ul `cmd_vel`, cea de a doua linie se referă la o acțiune. Observați că tastele **G|B|V|C|D|E|R|T** formează un „cadru" în jurul tastei **F** pe o tastatură QWERTY. Poziția fiecărei taste în jurul tastei **F** corespunde unei orientări în `turtlesim`. De exemplu, tasta **E** va roti "țestoasa" spre colțul din stânga sus.

3. Acum concentrați-vă pe terminalul în care rulează nodul `/turtlesim`. De fiecare dată când apăsați una dintre aceste taste, trimiteți un scop (*goal*) către un server de acțiuni din cadrul nodului `/turtlesim`. Scopul este ca "țestoasă" să se rotească într-o anumită direcție. După finalizează rotația, un mesaj care indică rezultatul ar trebui să apară:

```
[INFO] [turtlesim]: Rotation goal completed successfully
```

Tasta **F** permite anularea unui scop în timpul execuției. Încercați să apăsați tasta **C**, apoi tasta **F** înainte ca broasca "țestoasa" să finalizeze rotația. În terminalul unde rulează nodul `/turtlesim`, veți vedea mesajul:

```
[INFO] [turtlesim]: Rotation goal canceled
```

4. Pe lângă anularea de către client (comenzile date din/prin `teleop`), scopurile pot fi anulate și de către server (nodul `/turtlesim`). Când serverul decide să oprească procesarea unui scop, se spune că acesta îl anulează.

   Încercați să apăsați tasta **D**, apoi tasta **G** înainte ca prima rotație să fie completă. În terminalul unde rulează nodul `/turtlesim`, veți vedea mesajul:

```
[WARN] [turtlesim]: Rotation goal received before a previous goal
    finished. Aborting previous goal
```

   Acest server de acțiuni a ales să anuleze primul scop, deoarece a primit unul nou. Cu toate acestea, serverul ar fi putut alege și altceva, cum ar fi: **1.** Să respingă noul scop. **2.** Să execute al doilea scop după ce primul s-a finalizat.

   Info: Important

   Nu presupuneți că orice server de acțiuni va alege automat să anuleze scopul curent atunci când primește unul nou. Acest comportament este specific acestui server de acțiuni.

5. Pentru a vedea acțiunile pe care un nod le oferă, cum este cazul `/turtlesim`, deschideți un terminal nou și rulați comanda: `ros2 node info /turtlesim`. Această comandă va afișa o listă ce conține și serverele/clienți de acțiuni pentru nodul `/turtlesim`. Rezultatul ar trebui să fie similar cu:

```
/turtlesim
  Subscribers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /turtle1/cmd_vel: geometry_msgs/msg/Twist
  Publishers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /rosout: rcl_interfaces/msg/Log
    /turtle1/color_sensor: turtlesim/msg/Color
    /turtle1/pose: turtlesim/msg/Pose
  Service Servers:
    /clear: std_srvs/srv/Empty
    /kill: turtlesim/srv/Kill
    /reset: std_srvs/srv/Empty
    /spawn: turtlesim/srv/Spawn
```

```
    /turtle1/set_pen: turtlesim/srv/SetPen
    /turtle1/teleport_absolute: turtlesim/srv/TeleportAbsolute
    /turtle1/teleport_relative: turtlesim/srv/TeleportRelative
    /turtlesim/describe_parameters: rcl_interfaces/srv/
    DescribeParameters
    /turtlesim/get_parameter_types: rcl_interfaces/srv/
    GetParameterTypes
    /turtlesim/get_parameters: rcl_interfaces/srv/GetParameters
    /turtlesim/list_parameters: rcl_interfaces/srv/ListParameters
    /turtlesim/set_parameters: rcl_interfaces/srv/SetParameters
    /turtlesim/set_parameters_atomically: rcl_interfaces/srv/
    SetParametersAtomically
  Service Clients:

  Action Servers:
    /turtle1/rotate_absolute: turtlesim/action/RotateAbsolute
  Action Clients:
```

Se poate observa că în secțiunea `Actions Servers` apare acțiunea `/turtle1/rotate_absolute`. Acest lucru înseamnă că nodul `/turtlesim` poate răspunde și oferi feedback pentru acțiunea `/turtle1/rotate_absolute`.

6. Rulati acum pentru `/teleop_turtle` comanda de tip `info`. Unde apare `/turtle1/rotate_absolute`?

7. În același terminal rulați: `ros2 action list`. Ce returneaza această comandă? Dar `ros2 action list -t`?

8. Această singură acțiune activă poate fi inspectată mai departe folosind: `ros2 action info /turtle1/rotate_absolute`, care returnează:

```
Action: /turtle1/rotate_absolute
Action clients: 1
    /teleop_turtle
Action servers: 1
    /turtlesim
```

9. Similar cu topic-urile, rulați următoarele comenzi:

   a) `ros2 interface show turtlesim/action/RotateAbsolute`

   b) `ros2 action send_goal /turtle1/rotate_absolute turtlesim/action/RotateAbsolute "{theta: 1.57}"`

   c) `ros2 action send_goal /turtle1/rotate_absolute turtlesim/action/RotateAbsolute "{theta: -1.57}" --feedback`

10. *Clean up*: Închideți toate terminalele deschise, nu înainte de a opri execuția prin `Ctrl+C`.

4 ROS2 - Workspace

În contextul ROS2, workspace-ul se constituie într-un element fundamental al ecosistemului, menținând, generând un mediu de lucru structurat și organizat. Acesta nu este doar locul unde sunt stocate și gestionate pachetele software ROS2, ci și un ansamblu de instrumente și procese integrate, ce permit funcționarea eficientă a grafului computațional al ROS2.

Din perspectivă statică, workspace-ul ROS2 găzduiește întregul mecanism software necesar, incluzând executabile, biblioteci, definiții de mesaje comune, precum și alte componente esențiale. Dinamic, acesta facilitează execuția grafului ROS2 prin intermediul instrumentelor și proceselor specifice.

Workspace-ul este organizat pe două niveluri distincte: ***underlay*** și ***overlay***. *Underlay*-ul conține software-ul de bază ROS2, oferind fundamentul pentru dezvoltările ulterioare., în timp ce ***overlay***-ul, permite utilizatorului să dezvolte propriile pachete și aplicații, construind deasupra infrastructurii existente în underlay. Această structură ierarhică oferă flexibilitate și modularitate în dezvoltarea de aplicații ROS2.

```
# structura de fisiere a WS
build install log src
# source the overlay
cd ~/ros2_ws
source install/local_setup.bash
```

Procedura standard în definirea unui workspace:

- Clonare un repository (de obicei)

```
git clone https://github.com/ros/ros_tutorials.git -b humble
```

- Verificare/rezolvare dependențe

```
cd ..
rosdep install -i --from-path src --rosdistro humble -y

```

- Build pentu workspace

```
colcon build

```

5 ROS2 - *Packages* (pachete)

În cadrul ROS2, pachetul(*package*) se distinge ca unitatea fundamentală de organizare a codului, facilitând instalarea, distribuirea și reutilizarea eficientă a componentelor software. Prin utilizarea sistemului de *build* `ament` și a instrumentului `build colcon`, pachetele ROS2 pot fi dezvoltate prin intermediul limbajului C++ (utilizând CMake) sau Python, oferind flexibilitate în procesul de creare și integrare a modulelor software.

5.1 Sarcini de lucru

Exercise 1. Creați un pachet simplu urmând tutorialul din următorul link.

Exercise 2. Urmați tutorialul pentru codarea unui exemplu simplu de *publisher - subscriber* fie cod C++, fie cod Python

Exercise 3 (Opțional). Urmați tutorialul pentru codarea unui exemplu simplu de tip service-and-client.

6 Algoritmi simpli de control pentru `turtlesim`

6.1 Mișcarea într-o linie dreaptă folosind `turtlesim`

Pentru a începe, trebuie să pornim nodul **`turtlesim`**, care creează o fereastră grafică cu o țestoasă. Într-un terminal, rulează următoarea comandă:

```
ros2 run turtlesim turtlesim_node
```

Într-un alt terminal vom rula comenzile următoare pentru a crea un pachet în workspace-ul creat la laboratorul precedent:

```
cd ~/ros2_ws/src
ros2 pkg create turtlesim_controller --build-type ament_python
```

Aceste comenzi are trebui să genereze următoarea structură de fișiere și foldere:

```
turtlesim_controller/
 package.xml
 setup.py
 resource/
      turtlesim_controller
 turtlesim_controller/
      __init__.py
      straight_line.py
```

Exceptând fisierul **`straight_line.py`** care trebuie creat separat și în care se copiaza conținutul fisierului cu același nume.

De asemenea, fișierul **`setup.py`** trebuie să aibă următorul conținut (**Important!** conținutul linie 20):

```
from setuptools import setup

package_name = 'turtlesim_controller'
setup(
    name=package_name,
    version='0.0.0',
    packages=[package_name],
    data_files=[
        ('share/ament_index/resource_index/packages', ['resource/' +
    package_name]),
        ('share/' + package_name, ['package.xml']),
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='Your Name',
    maintainer_email='your_email@example.com',
    description='A ROS2 package to control Turtlesim',
    license='Apache License 2.0',
    entry_points={
        'console_scripts': [
            'straight_line = turtlesim_controller.straight_line:main',
        ],
    },
)
```

Pasul următor este construirea executabilelor aferente pachetului, utilizând comenzile:

```
cd ~/ros2_ws
colcon build --packages-select turtlesim_controller
```

și comanda pentru a face *source*:

```
source install/setup.bash
```

Se rulează nodul:

```
ros2 run turtlesim_controller straight_line
```

6.2 Sarcini de lucru

Exercise 4. Analizați codul din `straight_line.py`. Rulați pentru diferite exemple. Urmați aceeași pași și pentru codul din `rotate.py`.

Exercise 5. Similar, codul din `go_to_goal.py` implementează un compensator proporțional pentru a deplasa țestoasa într-o anumită. Analizați codul și identificați parametri și legile de control. Rulați pentru diverse set-uri de valori. Ce observați? Folosind codul propus, realizați următoarele:

1. mișcarea țestoasei într-un cerc cu viteză unghiulară constantă
2. mișcarea țestoasei într-un pătrat cu latura de 2x2 unități, utilizând o viteză liniară de 0,2 m/s și o viteză unghiulară de 0,2 rad/s (*"bucla deschisă"*)
3. (**BONUS 3p**) mișcarea țestoasei într-un pătrat cu latura de 3x3 unități, utilizând o lege de comandă a vitezei. Se predefinesc în program coordonatele punctelor pătratului. De exemplu, țestoasa să se deplaseze succesiv între punctele $(5,5) \to (8,5) \to (8,8) \to (5,8) \to (5,5)$ (*"bucla închisă"*)
4. (**BONUS 5p**) mișcarea țestoasei astfel încât aceasta să urmărească o traiectorie curbilinie pre-calculată, utilizând o lege/strategie de comandă la alegere. Evaluați performanța de urmărire.