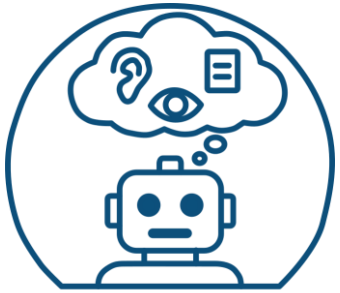
ROS2: Node, Topic, Service

운영체제의 실제

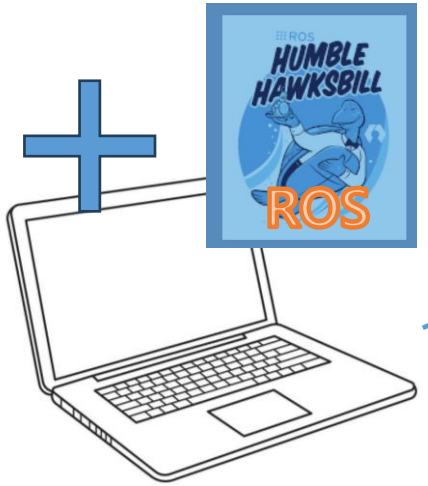
안인규 (Inkyu An)



How to Operate Robots

- **The main PC** is connected to **the robot** and **various sensors**, m ainly via USB
- It receives and processes data from multiple sensors through ROS on the main PC, and controls the robot accordingly.

Robot (Rainbow Robotics)



Main PC (e.g., Desktop, Laptop, Em bedded PC, …)

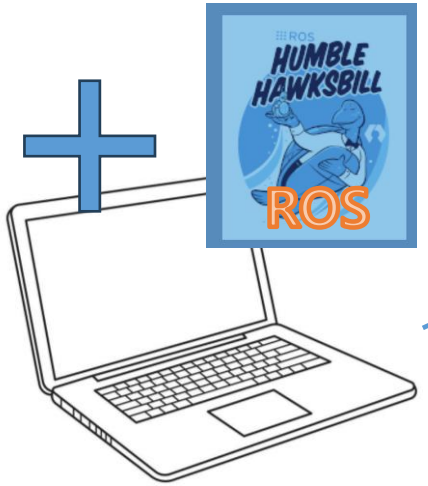
Depth Camera

Microphone

How to Operate Robots

- How do the PC obtain data from sensors?
- How do the PC control the robot?

Robot (Rainbow Robotics)



Main PC (e.g., Desktop, Laptop, Em bedded PC, …)

Depth Camera

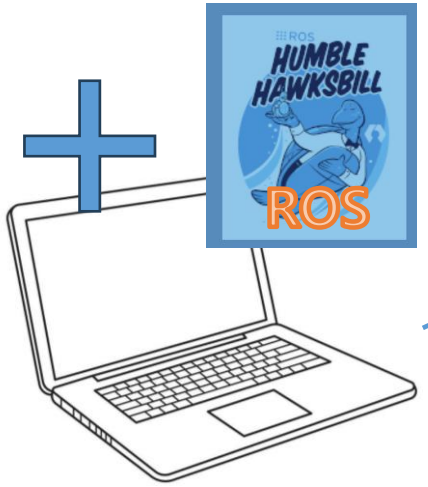
Microphone

How to Operate Robots

- How do the PC obtain data from sensors?
- How do the PC control the robot?

→ **Node, Topic, Service**

Robot (Rainbow Robotics)



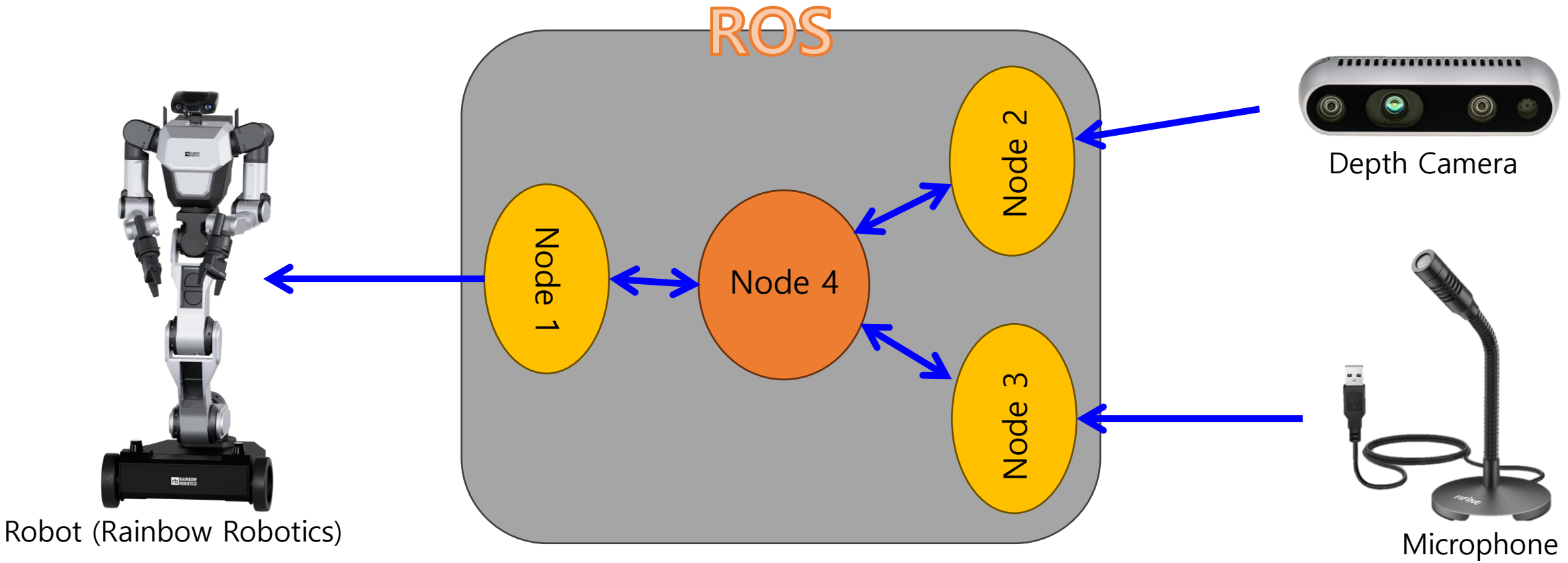
Main PC (e.g., Desktop, Laptop, Embedded PC, …)

Depth Camera

Microphone

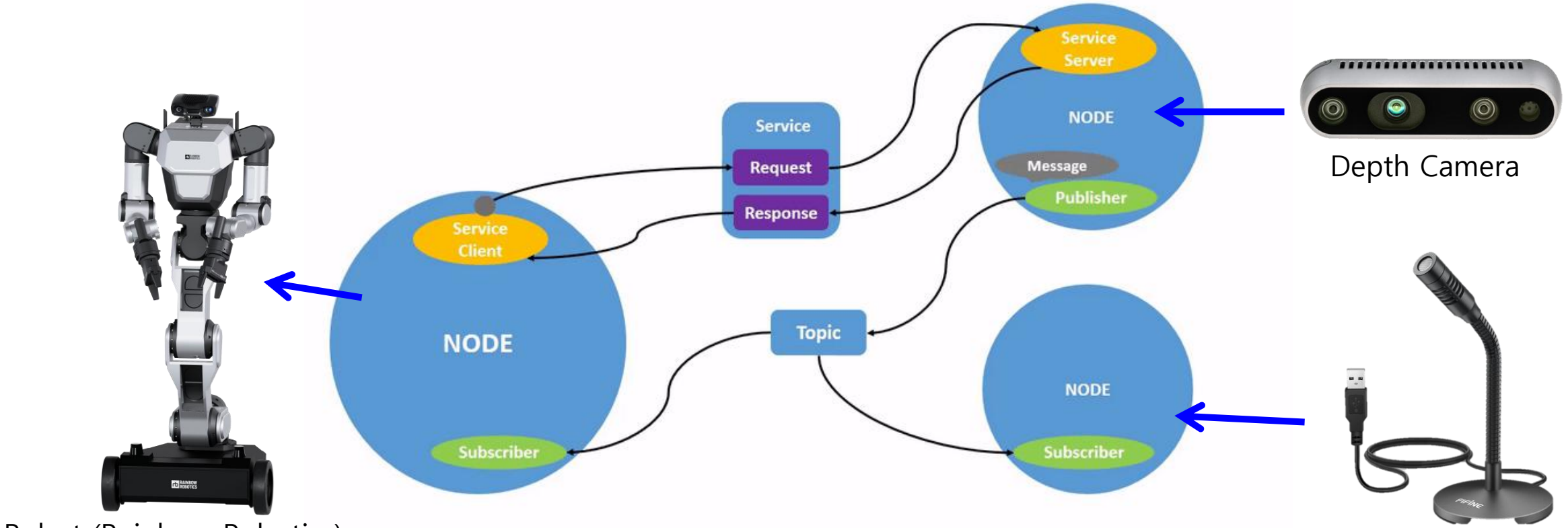
How to Operate Robots

- **Node**: sensors, actuator (로봇), algorithm 등을 실행하는 독립적인 실행 단위 (하나의 기능 단위를 담당하는 프로그램 = Process)
- **Topic, Service**: node간 데이터 통신 방식

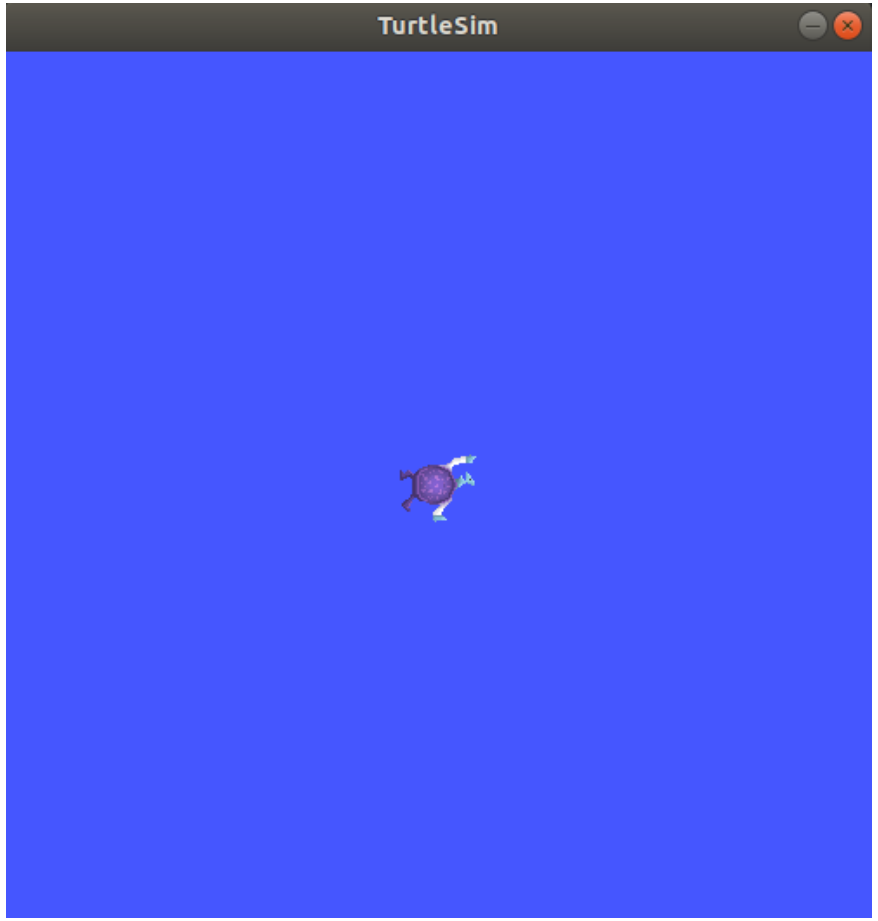


How to Operate Robots

- **Node**: sensors, actuator (로봇), algorithm 등을 실행하는 독립적인 실행 단위 (하나의 기능 단위를 담당하는 프로그램 = Process)
- **Topic, Service**: node간 데이터 통신 방식



Node – TurtleSim




```
$ ros2 pkg executables turtlesim
turtlesim draw_square
turtlesim mimic
turtlesim turtle_teleop_key
turtlesim turtlesim_node
```

- ros2 pkg executables <package 이름>
  - 특정 package <package 이름>의 실행 가능한 노드의 목록을 보여줌
- ros2 pkg –h

Nodes, Topic, Services

```
turtlesim/
├── CMakeLists.txt          # CMake 빌드 설정
├── package.xml             # 패키지 메타데이터 (package 이름, 버전, 설명, dependency(다른 package와) 등)
├── include/turtlesim/      # 헤더 파일 (C++)
├── src/
│   ├── turtle.cpp
│   ├── turtle_frame.cpp
│   └── turtlesim.cpp
├── launch/                 # 런치 파일
│   └── Multisim.launch.py
├── msg/                    # 메시지 타입 정의
│   ├── Color.msg
│   ├── Pose.msg
├── srv/                    # 서비스 타입 정의
│   ├── Kill.srv
│   ├── SetPen.srv
│   ├── ...
├── tutorials/
│   ├── draw_square.cpp
│   ├── mimic.cpp
│   ├── teleop_turtle_key.cpp
```

‘Turtlesim’ package를 통해 Nodes, Topic, Services를 실습

Nodes, Topic, Services

```
turtlesim/
├── CMakeLists.txt          # CMake 빌드 설정
├── package.xml             # 패키지 메타데이터
├── include/turtlesim/      # 헤더 파일 (C++)
├── src/
│   ├── turtle.cpp
│   ├── turtle_frame.cpp
│   └── turtlesim.cpp
├── launch/                 # 런치 파일
│   └── Multisim.launch.py
├── msg/                    # 메시지 타입 정의
│   ├── Color.msg
│   └── Pose.msg
├── srv/                    # 서비스 타입 정의
│   ├── Kill.srv
│   ├── SetPen.srv
│   └── ...
├── tutorials/
│   ├── draw_square.cpp
│   ├── mimic.cpp
│   └── teleop_turtle_key.cpp
```

C++ 또는 Python 을 통해 node가 정의 된 후,
CMakeList.txt 파일에서 실행 파일로 정의 되어야 함.

'Turtlesim' package를 통해 Nodes, Topic, Services를 실습

Nodes

<CMakeList.txt>

```
44  add_executable(turtlesim_node ----------> Turtlesim_node
45    src/turtlesim.cpp
46    src/turtle.cpp
47    src/turtle_frame.cpp
48    ${turtlesim_node_MOCS}
49  )
        ⋮
63  add_executable(turtle_teleop_key tutorials/teleop_turtle_key.cpp) ----------> Turtlesim_node
        ⋮
72  add_executable(draw_square tutorials/draw_square.cpp) ----------> draw_square
        ⋮
81  add_executable(mimic tutorials/mimic.cpp) ----------> mimic
```

Nodes

- The command ‘ros2 run’ launches an executable from a package
  - *ros2 run <package_name> <executable_name>*
  - e.g., ros2 *run turtlesim turtlesim_node*

- ‘ros2 node list’ will show you the names of all running nodes
  - *ros2 node list*

- Open another new terminal and start the teleop node with the c ommands:
  - *ros2 run turtlesim turtle_teleop_key*

Nodes - Remapping

- ‘Remmaping’ allows you to reassign default node properties, like node name, topic names, service names, etc., to custom values
- Let’s reassign the name of our ‘/turtlesim’ node
  - *ros2 run turtlesim turtlesim_node --ros_args --remap __node:=my_turtle*
  - *ros2 node list*

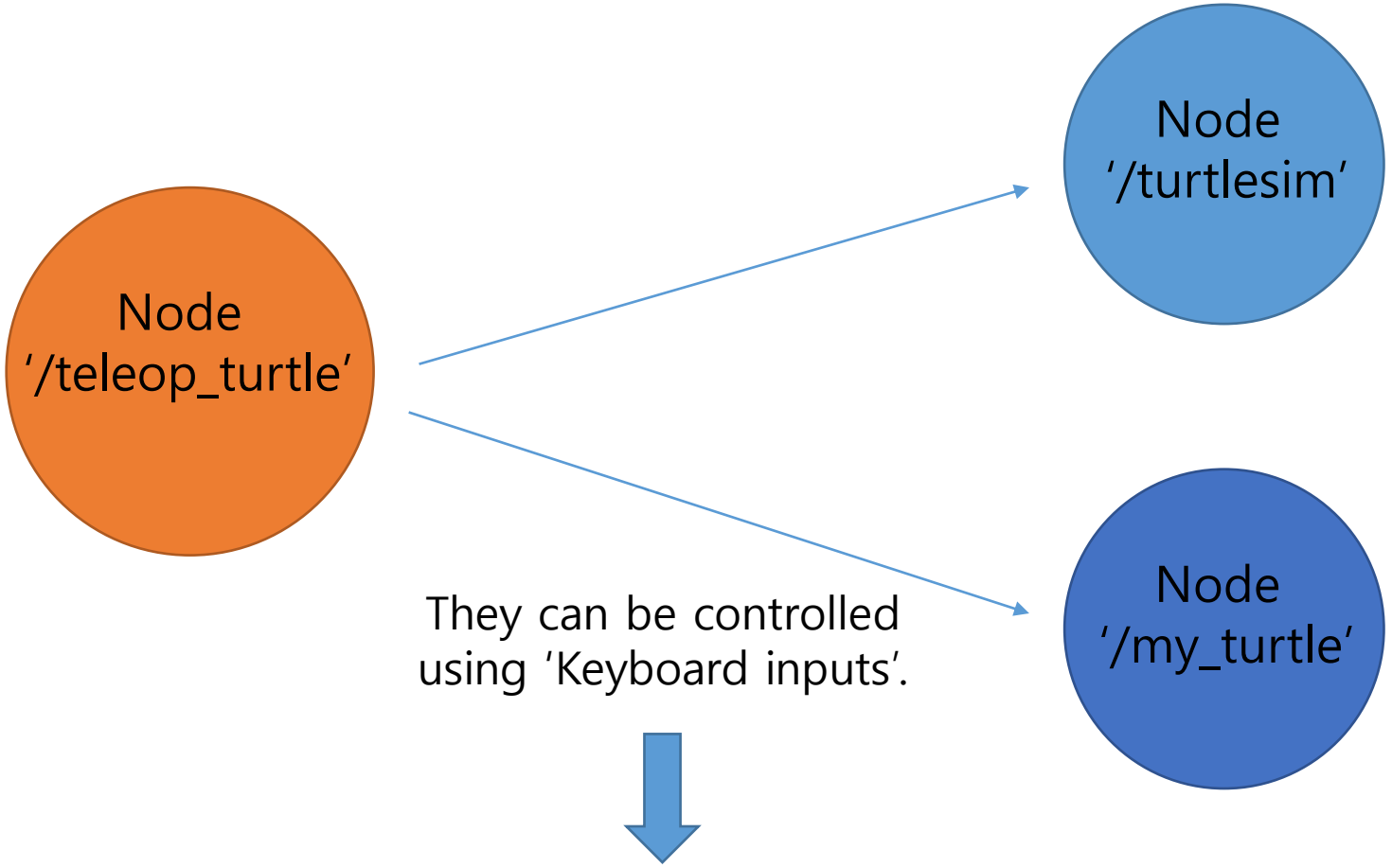
Nodes - Information

- Now that you know the names of your names, you can access more information about them with:
  - *ros2 node info <node_name>*
  - E.g., *ros2 node info /my_turtle*

```
$ ros2 node info /my_turtle
/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:
```

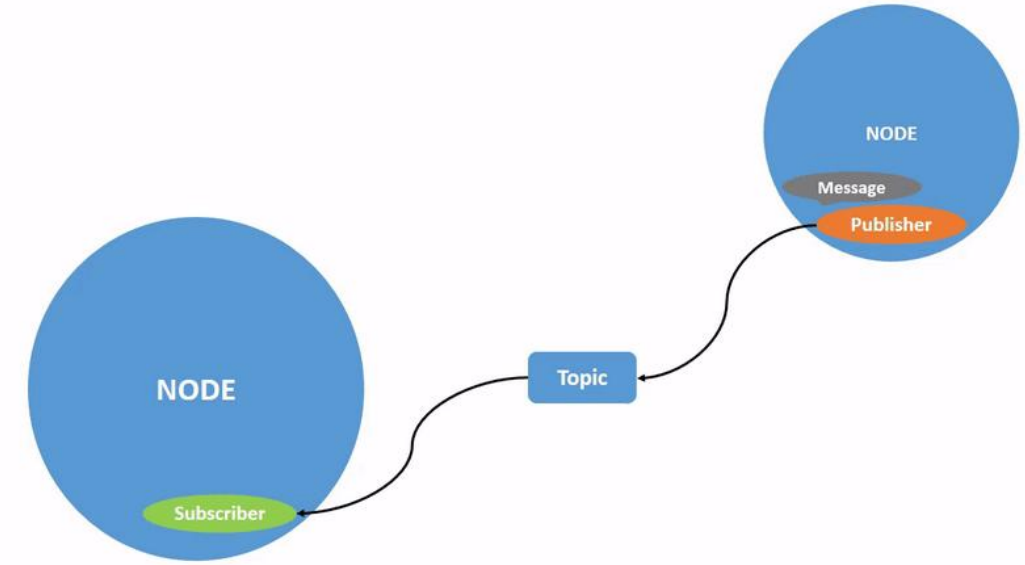
How to communicate between nodes

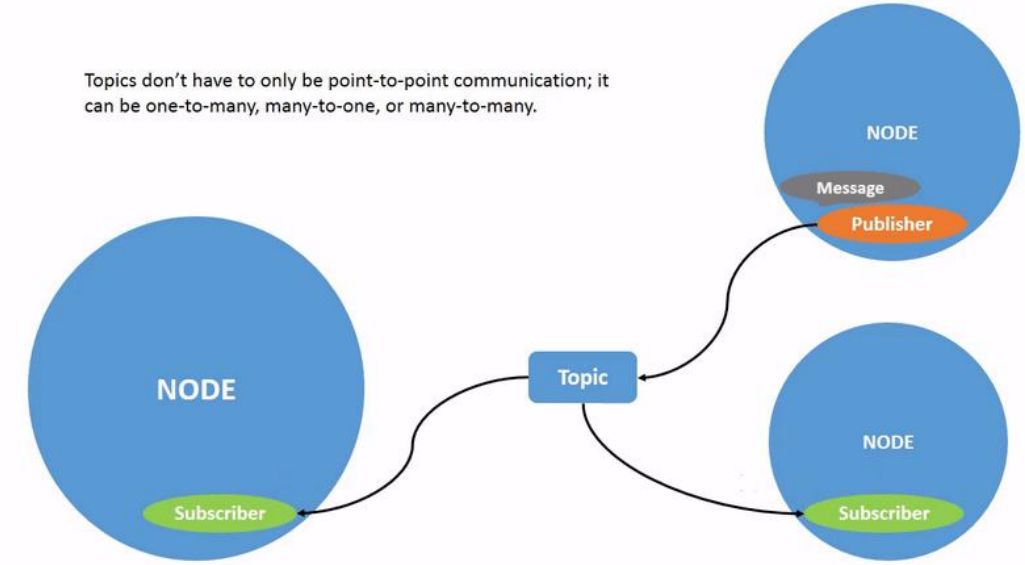


**How can I send 'Keyboard inputs'
to both nodes?**

Topics

- ROS2 breaks complex systems down input many modular nodes
- Topics are a vital element of the ROS graph that act as a bus for nodes to exchange messages
- A node may publish data to any number of topic and simultaneo usly have subscriptions to any number of topics





Topics – Turtlesim

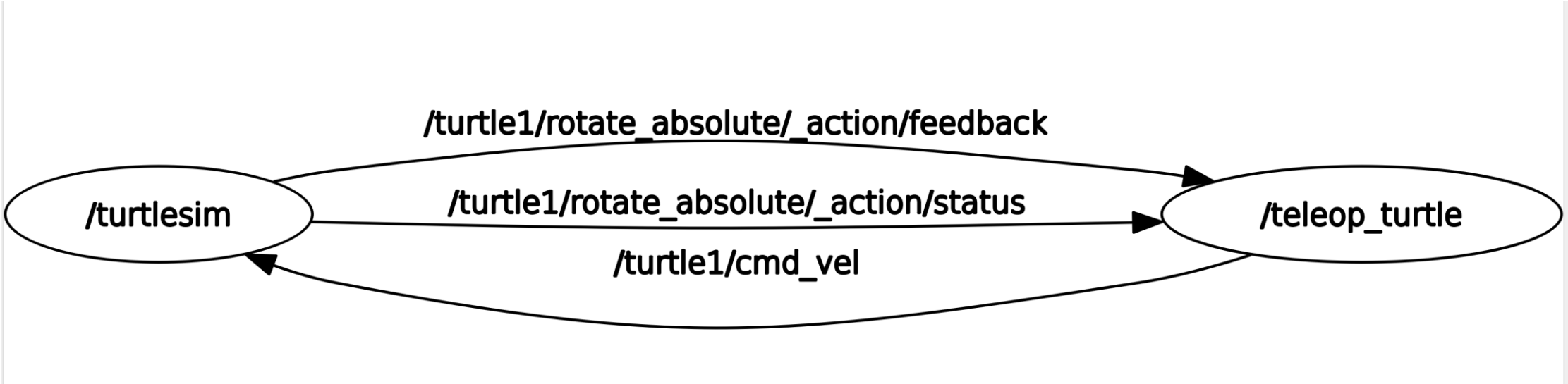
- Open a new terminal and run:
  - ros2 run turtlesim turtlesim_node
- Open another terminal and run:
  - ros2 run turtlesim turtle_teleop_key

- We will use 'rqt_graph' to visualize the changing nodes and topics

  - 'rqt_graph'는 RQT (ROS Qt) 프레임워크 기반의 플러그인 중 하나로, ROS2 그래프 구조를 GUI 형태로 확인할 수 있음
  - 즉, Node들이 Topic, Service, Action 을 통해 메시지를 주고받으며 통신하는데, 이를 그림으로 표현해 줌

Topics – Turtlesim

- To run ‘qrt_graph’, open a new terminal and enter the command:
  - *ros2 run qrt_graph rqt_graph*



Topics – Turtlesim

- Running the ‘ros2 topic list’ command in a new terminal will retur n a list of all the topics currently active in the system:
  - *ros2 topic list*

```
$ ros2 topic list
/parameter_events
/rosout
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
```

  - *ros2 topic list –t* will return the same list of topics, this time with the topic type appended in brackets:

```
$ ros2 topic list -t
/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]
```

Topics – Turtlesim

- To see the data being published on a topic, use:
  - *ros2 topic echo <topic_name>*
  - E.g., *ros2 topic echo /turtle1/cmd_vel* → **Who published this topic?**

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

Topics – Turtlesim

- Topics don’t have to only be one-to-one communication; they ca n be one-to-many, many-to-one, or many-to-many.
- Another way to look at this is running
  - *ros2 topic info /turtle1/cmd_vel*

```
$ ros2 topic info /turtle1/cmd_vel
Type: geometry_msgs/msg/Twist
Publisher count: 1
Subscription count: 2
```

Who subscribed this topic?

Topics – Turtlesim

- Nodes send data over topics using message
  → Publishers and subscribers must send and receive the same ty pe of message to communicate

- The topic types we saw earlier after running 'ros2 topic list –t' let us know what message type is used on each topic
- Recall that the 'cmd_vel' topic has the type:
  - *ros2 topic info /turtle1/cmd_vel*

```
$ ros2 topic info /turtle1/cmd_vel
Type: geometry_msgs/msg/Twist
Publisher count: 1
Subscription count: 2
```

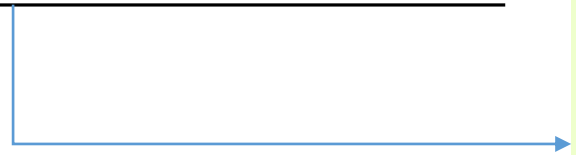
'Geometry_msgs/msg/Twist'
→ This means that in the package 'geometry_msgs' there is a msg called Twist

Topics – Turtlesim

- Now we can run ‘ros2 interface show <msg_type>’ on this type to learn its details
  - *ros2 interface show geometry_msgs/msg/Twist*

```
# This expresses velocity in free space broken into its linear and angular parts.
    Vector3  linear
            float64 x
            float64 y
            float64 z
    Vector3  angular
            float64 x
            float64 y
            float64 z
```

  - ‘/turtlesim’ node is expecting a message with two vectors, ‘linear’ and ‘angular’, of three elements each



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

Topics – Turtlesim

- Now that you have the message structure, you can publish data to a topic directly from the command line using:
  - *ros2 topic pub <topic_name> <msg_type> '<args>'*
  - The '<args>' argment is the actual data you'll pass to the topic
  - Make the robot turn right!

Topics – Turtlesim

- *ros2 topic pub <topic_name> <msg_type> '<args>'*
  - The '<args>' argment is the actual data you'll pass to the topic
  - It is important to note that this argument needs to be input in YAML syntax

```
linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 1.8
```

{linear: {x: 2.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 1.8}}

  - With no command-line options, 'ros2 topic pub' publishes the command in a steady stream at 1 Hz

Topics – Turtlesim

- With no command-line options, ‘ros2 topic pub’ publishes the command in a steady stream at 1 Hz → 만약 한번만 publish 하고 싶으면?
  - *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}}”*
  - --once: an optional argument meaning “publish one message then exit”.
  - -w 2: an optional argument meaning “wait for two matching subscriptions”.

Topics – Turtlesim

- ros2 topic hz
  - you can also view the rate at which data is published using:
  - *ros2 topic hz /turtle1/pose* → **What happened? Check 'rqt_graph'**

- ros2 topic bw
  - The bandwidth used by a topic can be viewed using:
  - *ros2 topic bw /turtle1/pose*

```
$ ros2 topic bw /turtle1/pose
Subscribed to [/turtle1/pose]
1.51 KB/s from 62 messages
    Message size mean: 0.02 KB min: 0.02 KB max: 0.02 KB
```

Topics – Turtlesim

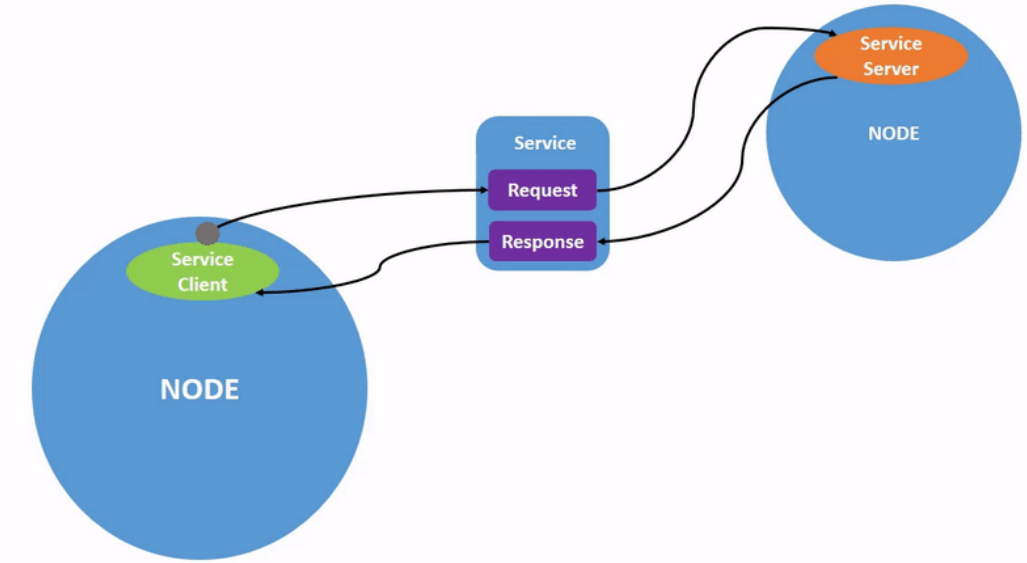
- ros2 topic find
  - To find a list of available topics of a given type use:
  - *ros2 topic find <topic_type>*

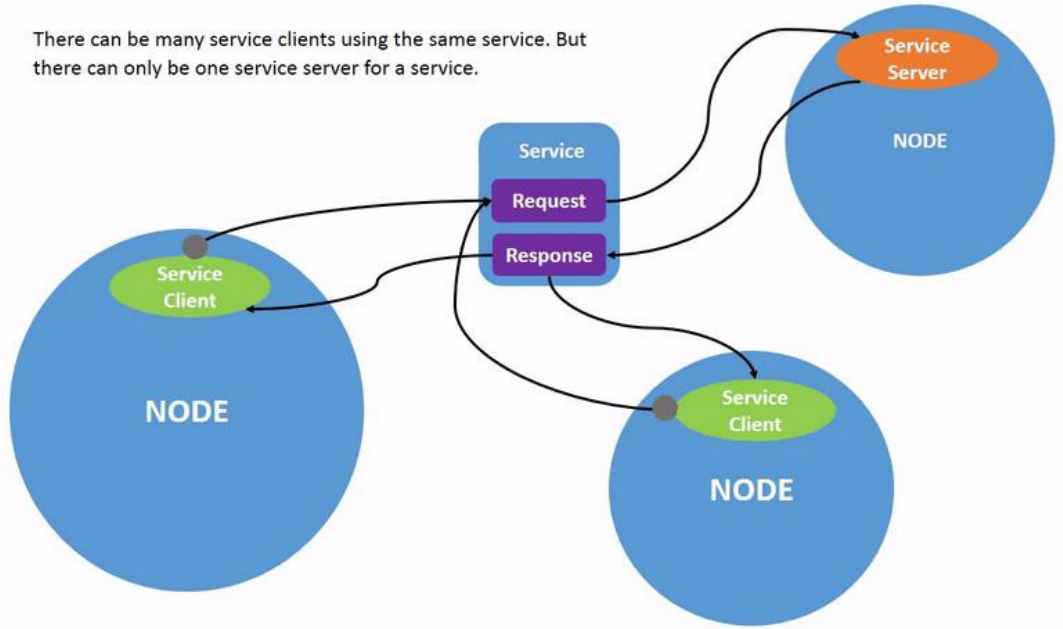
  - Recall that the 'cmd_vel' topic has the type: `geometry_msgs/msg/Twist`

  - E.g., *ros2 topic find geometry_msgs/msg/Twist*

Service

- Services are another method of communication for nodes in the ROS graph
- Services are based on a call-and-response model versus the publisher-subscriber model of topics
- While **topics** allow nodes to subscribe to data streams and get continual updates, **services** only provide data when they are specifically called by a client





Service – Turtlesim

- Open a new terminal and run:
  - ros2 run turtlesim turtlesim_node
- Open another terminal and run:
  - ros2 run turtlesim turtle_teleop_key
- Run the ‘ros2 service list’ command in a new terminal

```
$ ros2 service list
/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
```

Service – Turtlesim

- Services have types that describe how the request and response data of a service is structured.
- Service types are defined similarly to topic types, except service types have two parts: one message for **the request** and **another for the response**.
- ros2 service type <service_name>
  - Let's take a look at turtlesim's '/clear' service
  - *ros2 service type /clear*

```
$ ros2 service type /clear
std_srvs/srv/Empty
```

  - The 'Empty' type means the services call sends no data when making a request and receives no data when receiving a response (즉, 데이터를 주고받지 않고 단순히 'Trigger (동작 실행 신호)' 역할만 하고 싶을 때 사용)

Service – Turtlesim

- To see the types of all the active services at the same time, you can append the '--show-type' option, abbreviated as '-t', to the 'list' command
  - *ros2 service list -t*

```
$ ros2 service list -t
/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]
...
```

Service – Turtlesim

- If you want to find all the services of a specific type, you can use the command:
  - *ros2 service find <type_name>*
  - E.g., *ros2 service find std_srvs/srv/Empty*

```
$ ros2 service find std_srvs/srv/Empty
/clear
/reset
```

- You need to know the structure of the input arguments
  - *ros2 interface show <type_name>*
  - E.g., *ros2 interface show std_srvs/srv/Empty*

```
$ ros2 interface show std_srvs/srv/Empty
---
```

The --- separates the request structure (above) from the response structure (below). But, as you learned earlier, the Empty type doesn't send or receive any data. So, naturally, its structure is blank.

Service – Turtlesim

- You need to know the structure of the input arguments
  - ros2 interface show <type_name>
  - E.g., *ros2 interface show turtlesim/srv/Spawn*

```
$ ros2 interface show turtlesim/srv/Spawn
float32 x
float32 y
float32 theta
string name # Optional.  A unique name will be created and returned if this is empty
---
string name
```

The request structure (above)

The response structure (below)

- **Request (요청)**
  - x: 생성할 거북이의 x 좌표
  - y: 생성할 거북이의 y 좌표
  - theta: 생성할 거북이의 방향(라디안 단위, 0 = 오른쪽)
  - name: 새 거북이의 이름(빈 문자열이면 자동으로 turtle2, turtle3 … 식으로 생성됨)
- **Response (응답)**
  - name: 생성된 거북이의 이름

Service – Turtlesim

- Now that you know what a service type is, how to find a service's type, and how to find the structure of that type's arguments, you can call a service using:
  - *ros2 service call <service_name> <service_type> <arguments>*
  - E.g., *ros2 service call /clear std_srvs/srv/Empty*
  - E.g., *ros2 service call /spawn turtlesim/srv/Spawn "{x: 2, y: 2, theta: 0.2, name: ''}"*

Service – Turtlesim

- HW1: How can we control the new turtle?
  - 새로운 turtle이 그 자리에서 계속해서 원을 그리며 움직이게 하기 위해서는 어떻게 해야 할까?