Asssignment - Pick-and-place (15%)

MTRN4231 - UNSW School of Mechanical and Manufacturing Engineering

Introduction

This assignment is derived from a pick-and-place operation. In which a camera identifies items and reports them to an operator. The operator then sends pick or place commands to move the item to and from the inventory. For simplicity, a simulated item identification message is used as well as a simulated arm service. To complete this assignment you will be tasked with creating 5 nodes separated into 3 packages. You may complete your nodes in Python and/or C++. Your solution will be auto-marked based on the input and output interfaces specified in the assignment specification.

It is recommended you develop packages and nodes in the order in this assignment. First, develop and test the inventory node. If you verify it is correct only then move to the next node. The beauty of ROS is that it is very easy to subdivide tasks into nodes and packages. As a result, it scales efficiently and allows for easy integration. This assignment is laid out to mimic and show you a process that can be used to design an effective architecture with ROS2. Here is an overview:

- Determine a set of project goals and outcomes.
- Generate a program flow chart/decision tree.
- Separate relevant sections into packages and further nodes.
- Design the node interactions.
- Decide on a set of common interfaces.
- Design the specific input and output aspects for each node.

Workspace structure

Below is a high-level overview of the nodes and packages which will be included in your workspace. Your task is to write the brain, inventory and perception packages. Additionally, you will need to complete the semi-complete interfaces package.

```
arm *
└── ArmService *
brain
├── brain
└── brain_transformations
interface_verification *
├── brain_verification *
├── inventory_verification *
└── perception_verification *
interfaces **
inventory
├── inventory_manager
└── inventory_transformations
perception
└── perception
```

* - indicates provided completed code, ** - indicates semi-complete code that you will need to finish.

Running your code - individual packages

Each package must have its own launch file which launches all the nodes contained in the package. Your launch file must be in the format:

```
ros2 launch {package_name} {package_name}_launch.py
```

For instance, to launch the inventory package you would run the command:

```
ros2 launch inventory inventory_launch.py
```

and it would launch the **inventory_transformations** and **inventory_manager** nodes.

Running your code - complete solution

Place a launch file in your brain package that launches all other launch files. Your entire solution should launch using the command:

```
ros2 launch brain system_launch.py
```

Project goals

- Operator requests item pick and place operations.
- Inventory system is used to store items.
- Relevant package status messages are published to the operator.
- System is visualised in RVIZ.

Project decision tree

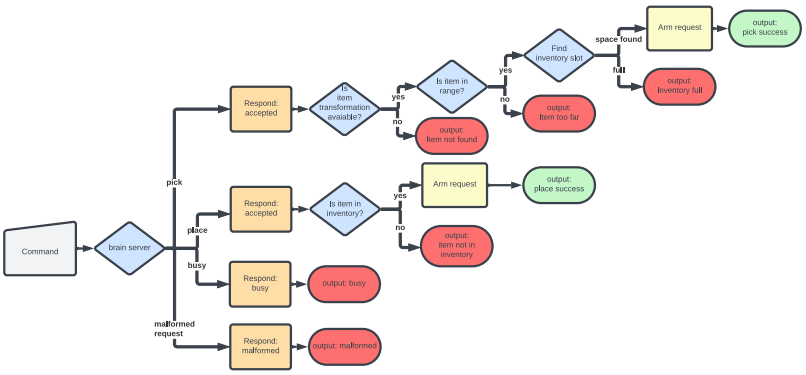


Figure 1: System flowchart from the operators perspective.

High-level node overview

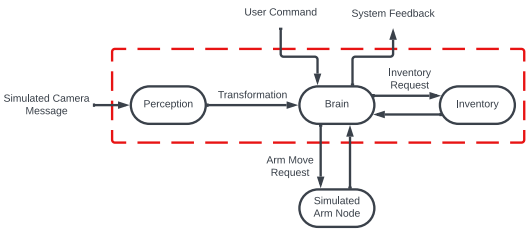


Figure 2: System architecture overview with important node relations. The packages within the red box are what you implement in this assignment.

Task 1 - Interfaces - 1 Mark

You must implement and use the interface descriptions outlined below. Two interfaces **ArmMovement.srv** and **Camera.msg** have already been provided, you must use these and must not edit their content.

BrainStatus.msg

This message interface is used to send status updates from the brain node to the operator.

```
string status
```

InventoryStatus.msg

This message interface is used to send status updates from the brain node to the operator.

```
int32 slot_1
int32 slot_2
int32 slot_3
```

Command.srv

This message type is used by the operator to send a pick or place request to the system.

```
string command
int32 item_id
---
string accept
```

Inventory.srv

This message type is used by the brain to request a state change or information from the inventory server.

```
string command
int32 item_id
---
int32 response
```

Task 2 - Package: inventory - Node: inventory_transformations - 1 Mark

The **inventory_transformations** node serves as a static transformation broadcaster that publishes the locations of the three inventory slots relative to a **base_link**.

Parent Frame	Child Frame	Transform (Translation, RPY)
base_link	inventory_slot_1	(0.25, 0.25, 0.0), (0.0, 0.0, $\frac{\pi}{4}$)
	inventory_slot_2	(0.25, 0.0, 0.0), (0.0, 0.0, 0.0)
	inventory_slot_3	(0.25, -0.25, 0.0), (0.0, 0.0, $-\frac{\pi}{4}$)

Table 1: Static ROS 2 Transformations for Inventory Slots

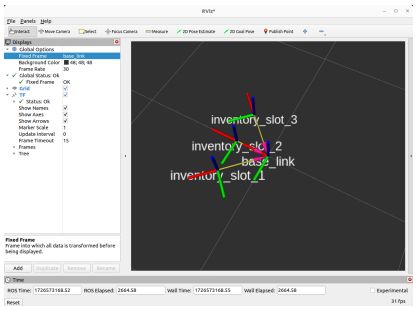


Figure 3: Inventory transformation frame example.

Ensure the node can run using the following command:

```
ros2 run inventory inventory_transformations
```

Task 3 - Package: inventory - Node: inventory_manager - 3 Marks

The inventory manager is a simple server that stores the current state of the inventory system. Using the **Inventory.srv** interface on service topic **inventory**, a client requests the commands **put_in_inventory** or **get_from_inventory** to add the **item_id** to add it to the inventory or remove it from the inventory, respectively. The inventory fills in ascending order, Ie slot_1 before slot_2 before slot_3. If the inventory space is empty use a value of -1 to represent this state.

Inventory manager server

put_in_inventory

Request:

- command = put_in_inventory
- item_id = Item ID

Response:

- response = 1 # If slot_1 is empty
- response = 2 # If slot_2 is empty
- response = 3 # If slot_3 is empty
- response = -1 # If inventory is full

Action:

Updates the internal state of the relevant slot with the new item if the inventory is not full.

get_from_inventory

Request:

- command = get_from_inventory
- item_id = Item ID

Response:

- response = 1 # If item_id is in slot 1
- response = 2 # If item_id is in slot 2
- response = 3 # If item_id is in slot 3
- response = -1 # If item_id is not in the inventory

Action:

Updates the internal state of the relevant slot if the item is found. If multiple items with the same ID are in the inventory return the location of the first slot the item is in, based on an ascending slot order.

Invalid condition

- Invalid if the command is not either action listed above
- Invalid if the requested item_id is < 0

Response:

- response = -2 # If request is invalid

Inventory status

Inventory status message publisher

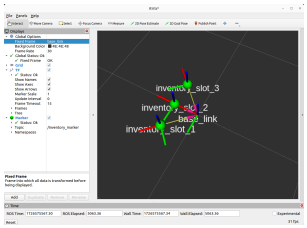
At a frequency of 1Hz publish a filled out **InventoryStatus.msg** message to the topic **inventory_status**.

Inventory visualisation publisher

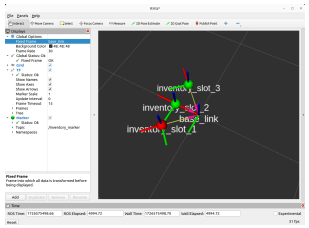
To visualise the state of the inventory you must publish a **Marker** message to the topic **inventory_marker**. You should publish at a rate of 1Hz and may use the same callback function used to publish the status message. A red ball is used to represent if the inventory space is occupied, whereas a green ball indicates the inventory slot is free. The marker parameters can be seen:

```
marker type = SPHERE
marker action =  ADD
markers scale = [0.1,0.1,0.1]
markers color = [0.0,1.0,0.0] \# if empty
markers color = [1.0,0.0,0.0] \# if occupied
markers color a = 1.0 \# opacity
```

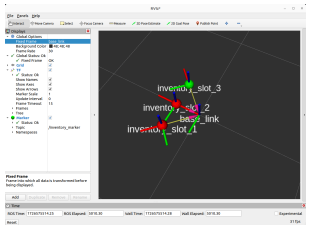
Example package operation



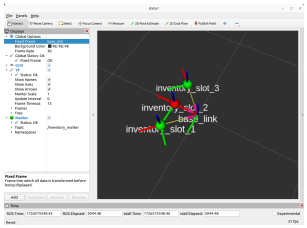
Default State



command: put_in_inventory
item_id: 2
response: 1



command: put_in_inventory
item_id: 11
response: 2



command: get_from_inventory
item_id: 2
response: 1

Task 4 - Package: perception - Node: perception - 3 Mark

The perception node aims to act as an interface between an item recognition implementation and the rest of the system. For this assignment you do not need to perform item recognition, rather it will be simulated by an external publish in the form of a **Camera.msg** message. When the perception node receives a **Camera.msg** message from the topic **camera** it dynamically broadcasts a transformation frame between the frame **camera_link** and a new frame named after the **item_id**. This dynamically broadcasted frame will then be used in a lookup operation with the brain node to assess if the item is close enough to be picked up. With a frequency of 1Hz, the perception node must also publish a message of type **Int32MultiArray.msg** to the topic **perception_status**. The message should contain a list of all the item ids seen in the previous 5 seconds by the perception node.

Inventory camera subscriber

On receiving a **Camera.msg** message on the topic **camera** publish the following dynamic transformation to the transformation tree.

Parent Frame	Child Frame	Transform (Translation, RPY)
camera_link	msg.item_id	(msg.x, msg.y, msg.z), (0.0, 0.0, 0.0)

Table 2: Dynamic ROS 2 Transformation for perception node

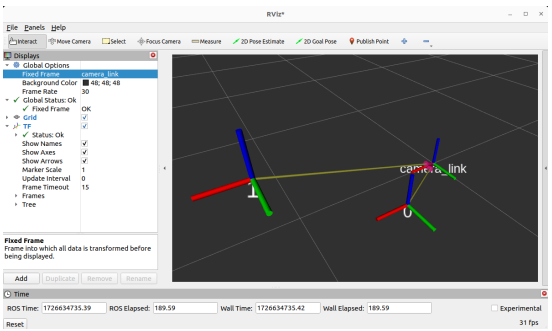


Figure 4: Example of resulting transformation observed by the camera.

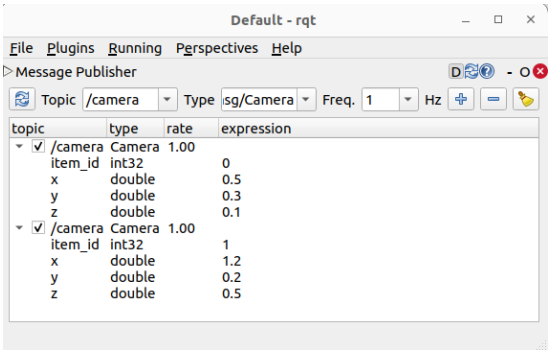


Figure 5: RQT used to publish simulated camera messages.

Inventory status publisher

With a frequency of 1Hz, the perception node must also publish a message of type **Int32MultiArray.msg** to the topic **perception_status**. The message should contain a list of all the item ids seen in the previous 5 seconds by the perception node. The resulting item list does not need to be sorted and only the item_ids have the be published. The message definition for the **example_interfaces.msg Int32MultiArray** can be found: Here

Task 5 - Package: brain - Node: brain_transformations - 1 Mark

The **brain_transformations** node serves as a static transformation broadcaster that publishes the locations of the **arm_link** and **camera_link** to **base_link**. Additionally a frame between **map** and **base_link** should be published.

Parent Frame	Child Frame	Transform (Translation, RPY)
map	base_link	(1.0, 1.0, 0.0), (0.0, 0.0, 0.0)
base_link	arm_base	(0.1, 0.0, 0.2), (0.0, 0.0, 0.0)
base_link	camera_link	(-0.5, 0.0, 0.5), (0.0, $\frac{\pi}{6}$, 0.0)

Table 3: Static ROS 2 Transformations for Robot Components

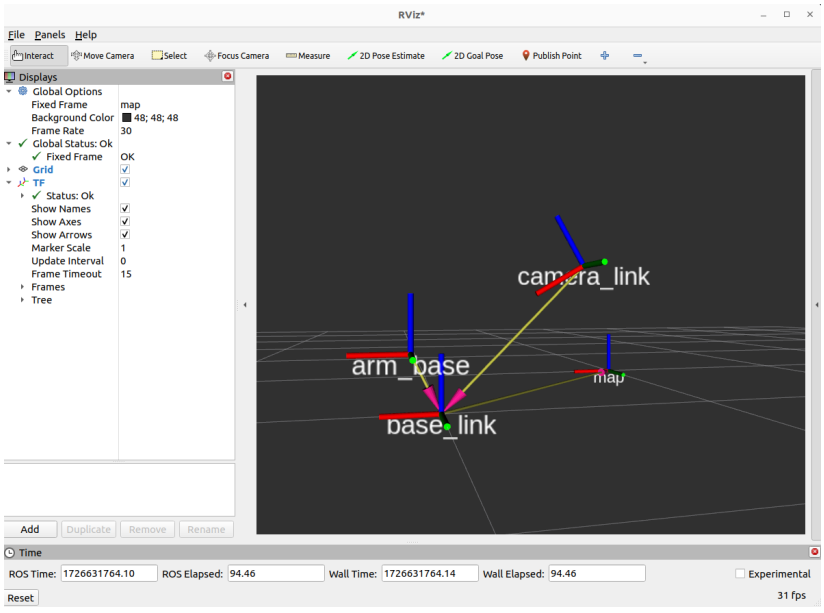


Figure 6: Inventory transformation frame example.

Ensure the node can run using the following command:

```
ros2 run brain brain_transformations
```

Task 6 - Package: brain - Node: brain - 3 Mark

The brain node acts as the interface between the operator and the system. It is primarily a server that accepts or rejects commands, if accepted, a routine and state machine is used to complete the command. The node receives commands from the service **Command.srv** on topic **command**. Status messages in the form of **BrainStatus.msg** are published to the topic **brain_status** when relevant.

Brain command server

Command

Request:

- command = "pick" or "place"
- item_id = Item ID

Response:

- accept = "accepted"
- accept = "busy"
- accept = "malformed command"

Action:

A visual representation of the server action can be found in Figure 1.

If the command is not **"pick"** or **"place"** respond with **"malformed command"**.

If the node is busy fulfilling the previous command respond with **"busy"**.

If the command is **"pick"** follow these steps:

- Set status to busy.
- Check if the item transformation has been published within the last 5 seconds. If not, publish an error status message and reset.
- Check if the Euclidean distance of the item is within <= 1m of the frame **"arm_link"**. If not, publish an error status message and reset.
- Call the inventory manager server with the command **"put_in_inventory"**. If full, publish an error status message and reset.
- Call the arm movement with the item position relative to **"arm_link"** and the desired inventory location. Publish arm status messages.
- Reset node.

If the command is **"place"** follow these steps:

- Set status to busy.
- Call the inventory manager server with the command **"get_from_inventory"**. If not in inventory, publish an error status message and reset.
- Call the arm movement with the pose (xyz) = (0.5,0.0,0.0) and the items inventory location. Publish arm status messages.
- Reset node.

Brain status publisher

Using the **BrainStatus.msg** message, publish relevant status updates to the topic **brain_status** following the format below. You must use these status message formats and may not add your own.

Command request

For every command received by the system publish the following status:

```
"UPDATE: new request {request.command} item {request.item_id} - Status {response.accept}"
```

Inventory status

If the item is not in the inventory:

```
"ERROR: item {item_id} is not in inventory"
```

If the inventory is full:

```
"ERROR: inventory is full"
```

Adding to the inventory:

```
"UPDATE: added item {item_id} to inventory slot {inventory_slot}"
```

Getting from the inventory:

```
"UPDATE: placing item {item_id} from inventory slot {inventory_slot}"
```

Transformation status

If the transformation to the item does not exist:

```
"ERROR: cannot find transformation to item {item_id}"
```

If the item is $>$ 1m from **"arm_link"**:

```
"ERROR: item {item_id} is too far"
```

Arm client status

If requesting an arm movement from the arm server:

```
"UPDATE: arm movement request"
```

If arm movement from the arm server finishes:

```
"UPDATE: arm movement finished"
```

Arm client

The brain node acts as a client and calls the arm server. Check the appendix for the control interface required by the provided arm server.

NOTE: Testing with RQT

If you use RQT to generate the **Camera.msg** message then the message will still be published even after the brain does a pick command. This is unlike a physical system, in which the arm would pick up the item and remove it from the frame and thus it would no longer be seen. This is a caveat that has to be made for this assignment.

System interface verification

3 interface test nodes have been provided, These nodes serve as an interface test to ensure your system matches the specification. It however does not test the functionality or correctness of your implementation. If you can not build the **interface_verification** package, or you get a different output from the one shown below, your system interface is incorrect.

You should develop your tests to ensure system correctness. If you wish to write unit tests for your packages you may follow the guide: Testing.

```
mitchell@GLaDOS:~/Documents/4231_assignment$ ros2 run interface_verification brain_verification
[INFO] [1726641112.688308199] [brain_verification]: Brain verification node launched
[INFO] [1726641112.688615464] [brain_verification]: BrainStatus topic interface is correct
[INFO] [1726641112.688800817] [brain_verification]: Command has the correct response
```

```
mitchell@GLaDOS:~/Documents/4231_assignment$ ros2 run interface_verification inventory_verification
[INFO] [1726641092.251362620] [inventory_verification]: Inventory service interface is correct
[INFO] [1726641092.421768929] [inventory_verification]: Inventory status interface is correct
```

```
mitchell@GLaDOS:~/Documents/4231_assignment$ ros2 run interface_verification perception_verification
[INFO] [1726641056.090003056] [perception]: Perception node launched
[INFO] [1726641056.402419552] [perception]: PerceptionStatus topic interface is correct
[INFO] [1726641057.084109924] [perception]: Published Camera message: item_id=1, x=0.0, y=0.0, z=0.0
```

Figure 7: Verification outputs

Compiling

Your solution will be compiled using the provided ./compile.sh script. It is recommended that you use this script as it specifically compiles the interfaces package before the remaining packages. This is to prevent errors from packages that require interfaces to be sourced.

Marking

Your system will be auto-tested. This assignment is worth 15 course marks, 12 marks are awarded for specific node implementations, and 3 marks are awarded based on your entire system's ability to perform.

Individual component marks

Each package will be tested individually by running the associated package launch file and evaluating the output for a set of simulated inputs. If your package does not launch with the required launch file format you will not receive marks for the package.

Tasks	Mark (s)
`brain_transformation`	1
`brain`	3
`interfaces`	1
`inventory_transformation`	1
`inventory_manager`	3
`perception`	3

System completeness - 3 Marks

The system completeness uses all three of the submitted packages to evaluate your complete system performance. You will only be eligible for these marks if your system launch file launches all 5 nodes and the interface package is complete. Your system (brain, perception, and inventory) must launch with the following command:

```
ros2 launch brain system_launch.py
```

You do not need to launch the arms package in this launch file.

Submission

Due: 11th October 2024

You must submit a compressed zip file containing your workspace to Moodle. Please delete the build, install, and log directories before compressing and submitting.

Plagiarism

If you are unclear about the definition of plagiarism, please refer to What is Plagiarism? — UNSW Current Students. You could get zero marks for the assignment if you were found:

- Knowingly providing your work to anyone and it was subsequently submitted (by anyone), or
- Copying or submitting any other person's work, including code from previous students of this course (except general public open-source libraries/code). Please cite the source if you refer to open-source code.

You will be notified and allowed to justify your case before such a penalty is applied.

Generative AI

Code written with a generative AI (ChatGPT) is allowed but must be clearly labelled with in-line comments and mentioned at the top of the file.

Late policy

UNSW has a standard late submission penalty of:

- 5% per day,
- for all assessment tasks where a penalty applies,
- capped at five days (120 hours) from the assessment submission deadline. In case of an approved Equitable Learning Plan (ELP) Providion, special consideration or short extension, the late penalty applies from the date of the approved time extension. After five days from the original or extended deadline, a student cannot submit an assessment, and
- No permitted variation.

Appendix - Package: arm - Node: arm

A simulated arm service package is supplied. When called, the arm server waits a random period before responding, this is to simulate a physical arm moving an unspecified amount of time. You must launch this package in your system launch file.

Arm Server

put_in_inventory

Request:

- command = "pick" or "place"
- x = x position relative to arm_link
- y = y position relative to arm_link
- z = z position relative to arm_link
- inventory_slot = Inventory slot to interact with

Response:

- success = 1 if the movement was completed
- success = 0 if the movement failed to complete

Action:

Stalls the response for a random time between 5-10 seconds. Simulates a physical arm movement.