Slicer ROS2

Release 0.9

Laura Connolly and Anton Deguet and Aravind Kumar

Apr 24, 2024

CONTENTS:

CHAPTER

ONE

FEATURES

1.1 Overview

This module is designed to enable direct communication between ROS 2 and 3D Slicer.

This module currently supports ROS 2 topics (subscribers and publishers), tf2 (broadcasters and lookups) as well as a parameter client (no server yet).

This module can also be used to visualize a ROS robot in action using a parameter client and tf2 lookups. The robot visualization implementation is following the ROS logic, i.e. the robot description (URDF) is retrieved as a parameter and the real-time link positions are from Tf2. As for RViz, for each robot, you will need a `robot_state_publisher` node that will:

- provide an URDF robot description using a ROS parameter `robot_description`
- update the links positions and broadcast them to tf2

All the functionalities in this module are encapsulated in `vtkMRMLNode` so you use them in your own applications. Since Slicer automatically provides a Python interface for all classes derived from `vtkMRMLNode`, you can develop your application using either C++ or Python.

1.2 Acknowledgment

The initial core developers are:

- Laura Connolly, EE PhD student at Queens University, Kingston, Ontario, Canada
- Aravind S. Kumar, CS Masters student at Johns Hopkins University, Baltimore, Maryland, USA
- Anton Deguet, Associate Research Engineer at Johns Hopkins University, Baltimore, Maryland, USA

This project is supported by:

- The National Institute of Biomedical Imaging and Bio-engineering of the U.S. National Institutes of Health (NIH) under award number R01EB020667, and 3R01EB020667-05S1 (MPI: Tokuda, Krieger, Leonard, and Fuge). The content is solely the responsibility of the authors and does not necessarily represent the official views of the NIH.
- The National Sciences and Engineering Research Council of Canada and the Canadian Institutes of Health Research, the Walter C. Sumner Memorial Award, the Mitacs Globalink Award and the Michael Smith Foreign Study Supplement.

1.3 Publications

- Connolly L, Deguet A, Leonard S, Tokuda J, Ungi T, Krieger A, Kazanzides P, Mousavi P, Fichtinger G, Taylor RH. Bridging 3D Slicer and ROS2 for Image-Guided Robotic Interventions. Sensors (Basel). 2022 Jul 17;22(14):5336. doi: 10.3390/s22145336. PMID: [35891016]; PMCID: [PMC9324680].

CHAPTER

TWO

GETTING STARTED

2.1 Pre-requisites

Before you can start compiling the SlicerROS2 module, you will need:

- Some knowledge of Linux, CMake and ROS 2.
- Ubuntu Linux with ROS 2. This module has been developed and tested using Ubuntu 20.04 with ROS Galactic. It might also work on Ubuntu 22.04 with ROS Humble.
- Slicer 3D built from source is required to build an extension.
  - Ubuntu 20.04 only. Instructions from the Slicer site recommend to install a newer version of Qt on Ubuntu 20.04. This makes the builds with ROS2 quite difficult so instead, we recommend to use Slicer versions up to 5.2.2 and use the Qt libraries installed by default along Ubuntu. After you cloned the Slicer sources, make sure you checkout 5.2.2 using `git checkout v5.2.2`.
  - Before you start compiling Slicer, make sure we use the system/native OpenSSL libraries otherwise you'll get some errors when compiling the Slicer ROS 2 module. You will need to do the following after you ran CMake for the first time. In the Slicer build directory, set `Slicer_USE_SYSTEM_OpenSSL` to `ON` using `cmake . -DSlicer_USE_SYSTEM_OpenSSL=ON -DCMAKE_BUILD_TYPE=Release` or `ccmake`.

  See Slicer build instructions.
- Remember the build directory for Slicer, it will be needed to compile the Slicer ROS 2 module.

Note: If you need to build Slicer from old sources, make sure `CMAKE_CXX_STANDARD` is set to `14` (required to compile Slicer code along ROS 2).

2.2 Compilation

This code should be built with `colcon` as a ROS 2 package. `colcon` is usually installed along ROS 2 but if it isn't, install it with `sudo apt install python3-colcon-common-extensions`. For now, we will assume the ROS workspace directory is `~/ros2_ws` and the source code for this module has been cloned under `~/ros2_ws/src/slicer_ros2_module`.

You will first need to "source" the ROS setup script for ROS 2 (Galactic in this example):

```
source /opt/ros/galactic/setup.bash
mkdir -p ~/ros2_ws/src
cd ~/ros2_ws/src
git clone https://github.com/rosmed/slicer_ros2_module
```

Then build the module using *colcon* while providing the path to your Slicer build directory `Slicer_DIR`:

```
cd ~/ros2_ws
colcon build --cmake-args -DSlicer_DIR:PATH=/home/your_user_name_here/something_
↪something/Slicer-SuperBuild-Debug/Slicer-build -DCMAKE_BUILD_TYPE=Release
```

The option `--cmake-args -DSlicer_DIR...` is only needed for the first `colcon` call. For future builds, you can revert back to just using `colcon build`.

Note that the build directory name is `ROS2`, not `slicer_ros2_module`. You will see some CMake warnings mentioning that the module's name (`ROS2`) doesn't follow the ROS2 package naming convention. ROS2 expects all packages to use the "snake_case" naming convention but we decided to ignore this to follow the VTK/Slicer naming convention, i.e. "CamelCase".

If the `Slicer_DIR` is not set properly (or you simply forgot), you should see the following error messages"

```
Could not find a package configuration file provided by "Slicer" with any
of the following names:

SlicerConfig.cmake
slicer-config.cmake
```

At that point, you don't need to clean your ROS workspace. You can fix the issue by running CMake on the build directory for the Slicer module (`ROS2`) `ccmake ~/ros2_ws/build/ROS2`. In CMake, set `Slicer_DIR` to point to your Slicer build directory then hit `c` to configure until you can hit `g` to generate the makefiles. If you prefer a graphical interface, you can use `cmake-gui` instead of `ccmake`. Once `Slicer_DIR` is set, try `colcon build` again (after `cd ~/ros2_ws`).

2.3 Loading the module

You will first need to make sure the environment variables are set properly so the Slicer ROS 2 module can locate all the ROS 2 resources (dynamic libraries and other ROS 2 packages you might need to access):

```
source ~/ros2_ws/install/setup.bash # or whatever your ROS 2 workspace is
```

In the same terminal navigate (`cd`) to your Slicer inner build directory and start Slicer. If you followed the Slicer build instructions, this should look like:

```
cd ~/something_something/Slicer-SuperBuild/Slicer-build
./Slicer
```

The first time you run Slicer, you need to add the module directory in the application settings so that the module can be dynamically loaded.

To do so, open Slicer and navigate through the menus: *Edit* → *Application Settings* → *Modules* → *Additional module paths* → *Add*. The path to add is based on your ROS workspace location as well as the Slicer version (5.3 in this example). It should look like:

```
~ros2_ws/build/ROS2/lib/Slicer-5.3/qt-loadable-modules
```

At that point, Slicer will offer to restart. Do so and then load the module using the drop down menu: *Modules* → *IGT* → *ROS2*

CHAPTER

THREE

ROBOT VISUALIZATION

3.1 Overview

For the robot visualization in Slicer 3D, one first need to properly configure a robot in ROS 2. The default ROS approach requires:

- an XML robot definition file (`UFA`) and usually CAD files for the links.
- a robot state publisher node which will make the URDF description available as a ROS parameter for other nodes. The robot state publisher will also compute the forward kinematics and broadcast the 3D position of each link to tf2.
- a source (publisher) for the current joint positions. This can be an actual robot driver or a script emulating the robot.

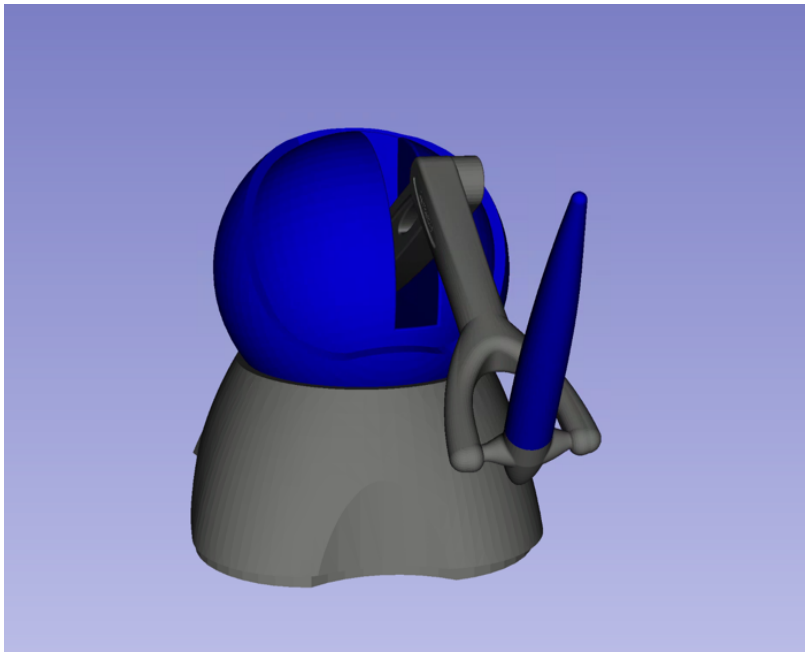
Once the ROS robot is setup properly, the SlicerROS2 robot node can be used to visualize the robot in Slicer. Internally, the robot node will use a parameter node to retrieve the URDF description and tf2 lookups to refresh the position of each link.

3.2 ROS Robot

The following are examples of robots we've used to test the SlicerROS2 module. They cover a serial robot (Phantom Omni) as well as a robot with parallel linkages (dVRK PSM). The SlicerROS2 module should work with any other robots as long as the links CAD files are either `.stl` or `.obj`.

3.2.1 Phantom Omni

The Phantom Omni is an entry level haptic device initially sold by Sensable. Later on, it has been renamed Geomagic Touch or 3DS Touch. The initial version used a FireWire connection. Later models used Ethernet and more recently USB.



We created and used a ROS package for the Phantom Omni: Omni Github Link. This package has no external dependencies and is very light so it's a good way to test the SlicerROS2 module. You can download this repository in your ROS 2 workspace's source directory and then build. Always remember to use `colcon build` followed by `source install/setup.bash` when you downloaded a new package to `src`.

This package contains the URDF, STL meshes, a launch file for the `robot_state_publisher` as well as a dummy script that publishes a joint trajectory so one can see the arm moving around.

To start the `robot_state_publisher`, use:

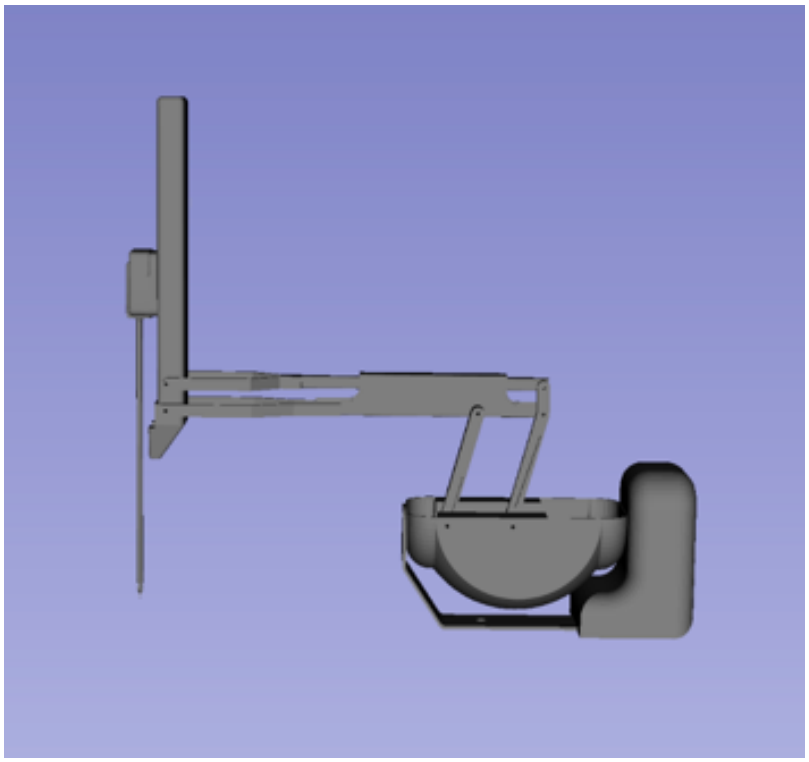
```
ros2 launch sensable_omni_model omni.launch.py
```

Then, to start the robot's dance, use an other terminal:

```
ros2 run sensable_omni_model pretend_omni_joint_state_publisher
```

3.2.2 dVRK PSM

The dVRK PSM (da Vinci Research Kit Patient Side Manipulator) is based on the first generation da Vinci RAMIS system (Robotically Assisted Minimally Invasive Surgery) sold by ISI (Intuitive Surgical systems).



Installing the dVRK code base is not too difficult but it will take a few minutes to compiled. You can find the build instructions for ROS 2 on the dVRK Wiki.

Once you've compiled all the dVRK related packages, you will need 3 terminals to launch the following nodes:

```
# simulated PSM
ros2 run dvrk_robot dvrk_console_json -j ~/ros2_ws/src/cisst-saw/sawIntuitiveResearchKit/
↪share/console/console-PSM1_KIN_SIMULATED.json

# robot state publisher
ros2 launch dvrk_model dvrk_state_publisher.launch.py arm:=PSM1

# test script, make the PSM1 move around
ros2 run dvrk_python dvrk_arm_test.py -a PSM1
```

3.2.3 Cobot

We also tested SlicerROS2 on myCobot by Elephant Robotics, specifically the myCobot 280 M5 Stack. The ROS 2 interface for the device can be found here and drivers can be installed from the Elephant Robotics website.

Assuming the interface (mycobot_ros2) is cloned under the same `ros2_ws`, the state publisher can be started using the following steps:

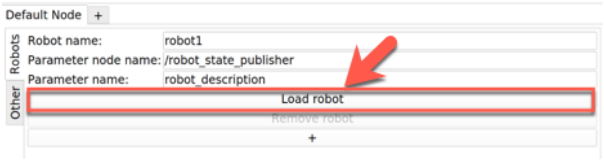
```
cd ~/ros2_ws/src/mycobot_ros2/src/mycobot_ros2/mycobot_280/mycobot_280/config
python3 listen_real.py
```

It's possible that you will need to change the port specified on line 14 of `listen_real.py` depending on your device. The `.dae` files in the robot description also need to be converted to STLs (an online converter will work) and the paths in the URDF file should be updated to reflect this change.

Once running - make sure your robot is in *Transponder Mode*. More instructions for basic operation of the myCobot can be found in the Gitbook

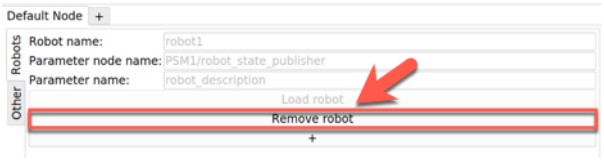
3.3 Slicer Robot

We've simplified loading the robot by adding some shortcuts on the widget UI. The default parameter node name: `/robot_state_publisher` and parameter name: `robot_description` should work for most cases. These are set as the default in the UI. To load a robot with these parameters, press the "Load Robot" button:

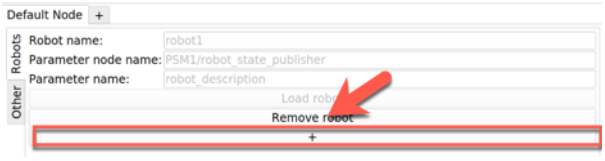


At that point, the robot's model should be loaded and displayed in Slicer. To do this addition programmatically, please see the documentation on the `vtkMRMLROS2RobotNode`. Note that if your robot uses a namespace, you will likely need to prefix this on the parameter node name. The PSM for example requires `PSM1/robot_state_publisher`.

To remove the robot from the scene you can press the "Remove Robot" button.



To add more than one robot, you can press the "+" button and the widget will update with additional line edits for the second robot.



CHAPTER

FOUR

MRML ROS NODES

The high level ROS functionalities (publisher, subscriber, Tf2 broadcast, Tf2 lookup or parameter client) are all encapsulated as Slicer MRML nodes, i.e. derived from the class `vtkMRMLNode` and added to the MRML Scene. This allows to leverage some of the Slicer features:

- Using Slicer node observer pattern to trigger user code when a new ROS message has been received (subscribers, parameters or Tf2 lookups).
- Data visualization using the MRML scene.
- Retrieve node by ID, name or type.
- Python API automatically generated on top of the native C++ code.
- Ability to save and restore your ROS nodes along the MRML scene.

All the SlicerROS2 classes follow the Slicer naming convention, i.e. `vtkMRMLxxxxNode`. We added the `ROS2` "prefix" for all the class names so we're using `vtkMRMLROS2xxxxNode` where `xxxx` represents a ROS functionality. This works fairly well but makes it a bit hard to read for the MRML node than encapsulates a ROS node, i.e. a `vtkMRMLROS2NodeNode`.

4.1 Design

4.1.1 C++ vs Python

Since both Slicer and ROS provide a Python interface we first tried to use Python as the glue between Slicer and ROS. The main issue on Ubuntu is that ROS relies on the system python (version 3.8 on Ubuntu 20.04) while Slicer builds it's own Python interpreter (e.g. 3.9). Since both libraries are not just native Python code, loading (`import`) both Slicer and ROS in the same interpreter is not possible as they rely on different versions of Python C++ libraries. We also considered using the same Python version for both Slicer and ROS, i.e. build everything using Python 3.8 or Python 3.9. Unfortunately, building Slicer against the Ubuntu provided Python interpreter (3.8) is no currently supported by the Slicer "super build". Alternatively, using the Slicer provided Python 3.9 to build the ROS Python bindings (`rclpy`) proved challenging unless one wanted to recompile ROS from scratch instead of using the existing Ubuntu binary packages.

We ultimately decided to implement the SlicerROS2 module in C++ and rely on the VTK/Slicer build to provide the Python bindings. The SlicerROS2 module is compiled against the Slicer and ROS2 libraries which is a bit challenging since both packages have their own CMake macros and have some Python version requirements. The `CMakeLists.txt` provided along SlicerROS2 works but you'll have to ignore some error and warning messages.

4.1.2 Execution Model

One of the challenges of integrating ROS in Slicer is to figure out the execution model. ROS relies heavily on callbacks triggered by external messages. This requires to either use a separate thread to "spin" the ROS event loop or periodically call the ROS "spin" method from the application's main thread. Since we are heavily relying on the MRML scene, using a separate thread is not trivial. Therefore the SlicerROS2 module relies on the main Slicer thread to trigger a periodic call to the ROS spin. We currently use a Qt timer to trigger this periodic call.

Note: The default frequency for the SlicerROS2 module is 50Hz, i.e. 20ms

---

Warning: Since we rely on a Qt timer to trigger the ROS spin, the module will not receive any ROS messages until the GUI is created, i.e. until the ROS module is manually loaded in Slicer.

4.1.3 Templates vs Inheritance

The two packages also differ in their design patterns. Slicer (and VTK) strongly relies on base classes and inheritance to allow runtime decisions. Meanwhile, ROS heavily relies on templates and type traits which are handled at compilation time. Most ROS communication mechanisms only support a finite number of data types (e.g. parameters are booleans, integers, floating points, strings or vector of aforementioned types, tf2 uses transforms only…) so this is not a major issue.

The main difficulty lies in supporting ROS topics and services For your code, we ended up using templates for our internal data structures and add some macros to generate the user classes. These macros use template specialization and add some methods to create a C++ class that can be used within Slicer (including the Python bindings generation).

4.1.4 Coordinate Systems and Units

The SlicerROS2 module will automatically convert between the default 3D frames conventions in Slicer and ROS. Slicer (and by extension all VTK objects in Slicer) follow the RAS convention and distances are provided in millimeters. Meanwhile uses the RHS convention and SI units (meters).

4.2 Nodes

The SlicerROS2 module always creates a default ROS node (internally a `rclcpp::node`). This node is both a ROS node and a MRML node, hence the unfortunate name `vtkMRMLROS2NodeNode`. This node is added to the MRML scene and should be used to add your custom `vtkMRMLROS2` nodes (topics, parameter…). It is possible to add more ROS nodes in SlicerROS2 but this feature has not been tested extensively for the first release.

To retrieve the default ROS node:

Python

C++

```
rosLogic = slicer.util.getModuleLogic('ROS2')
rosNode = rosLogic.GetDefaultROS2Node()
```

If you're modifying the Slicer ROS module, you can access the default ROS node directly using the `GetDefaultROS2Node` method.

```
vtkMRMLROS2NodeNode * rosNode = this->GetDefaultROS2Node();
```

If you're working on a new module that relies on the SlicerROS2 module, you can use the `GetModuleLogic` method and then the `GetDefaultROS2Node`.

```
vtkMRMLAbstractLogic * logic = this->GetModuleLogic("ROS2");
if (logic == nullptr) {
  vtkErrorMacro(<< "ROS2 logic not found");
} else {
 vtkSlicerROS2Logic * rosLogic =
      vtkSlicerROS2Logic::SafeDownCast(logic);
  if (rosLogic == nullptr) {
    vtkErrorMacro(<< "Found what should be the default ROS2 logic but the type is wrong
↪");
  } else {
    vtkMRMLROS2NodeNode * rosNode = rosLogic->GetDefaultROS2Node();
    // now we can use the node
  }
}
```

For all other cases, you can use node ID of the default ROS node (`vtkMRMLROS2NodeNode1`) to retrieve it from the MRML scene.

```
vtkMRMLNode * node = scene->GetNodeByID("vtkMRMLROS2NodeNode1");
if (node == nullptr) {
  vtkErrorMacro(<< "ROS2 default node not in scene");
} else {
  vtkMRMLROS2NodeNode * rosNode =
      vtkMRMLROS2NodeNode::SafeDownCast(node);
  if (rosNode == nullptr) {
    vtkErrorMacro(<< "Found what should be the default ROS2 node but the type is wrong");
  } else {
    // now we can use the node
  }
}
```

We don't recommend that you delete the default node. If you create another node and need to delete it, use the method `vtkMRMLROS2NodeNode::Destroy`.

4.3 Topics

In ROS, publishers and subscribers can send/receive any type of ROS message (defined in *.msg* files). These *.msg* files are then parsed by a code generator that creates the C++ code needed to support said message. All the classes and functions needed for ROS topics can then be templated and uses the "type traits pattern" since all the messages have a similar API. On the other hand, Slicer tends to avoid template for end-user classes. So we created a set of basic publishers and subscribers to convert messages between ROS and Slicer.

Table 1: Publishers and subscribers

Slicer type	ROS type	SlicerROS2 "name"
std::string	std_msgs::msg::String	String
bool	std_msgs::msg::Bool	Bool
int	std_msgs::msg::Int64	Int
double	std_msgs::msg::Float64	Double;
vtkIntArray	std_msgs::msg::Int64MultiArray	IntArray
vtkDoubleArray	std_msgs::msg::Float64MultiArray	DoubleArray
vtkTable	std_msgs::msg::Int64MultiArray	IntTable
vtkTable	std_msgs::msg::Float64MultiArray	DoubleTable
vtkMatrix4x4	geometry_msgs::msg::PoseStamped	PoseStamped
vtkDoubleArray	geometry_msgs::msg::WrenchStamped	WrenchStamped
vtkTransformCollection	geometry_msgs::msg::PoseArray	PoseArray

For example, if you need to create a publisher that will take a *vtkMatrix4x4* on the Slicer side and publish a *geometry_msgs::msg::PoseStamped* on the ROS side, the full SlicerROS2 node name will be *vtkMRMLROSPublisherPoseStampedNode*.

4.3.1 Publishers

To create a new publisher, one should use the MRML ROS2 Node method `vtkMRMLROS2NodeNode::CreateAndAddPublisherNode`. This method takes two parameters:

- the class (type) of publisher to be used. We provide some publishers for the most common data types (from Slicer to ROS messages). The full list can be found in the Slicer ROS logic class (`Logic/vtkSlicerROS2Logic.cxx`) in the method `RegisterNodes`.
- the topic name (`std::string`).

Publishers are triggered by calling the `Publish` method.

Python

C++

```
rosLogic = slicer.util.getModuleLogic('ROS2')
rosNode = rosLogic.GetDefaultROS2Node()
pubString = rosNode.CreateAndAddPublisherNode('vtkMRMLROS2PublisherStringNode', '/my_
↪string')
# run `ros2 topic echo /my_string` in a terminal to see the output
pubString.Publish('my first string')

pubMatrix = ros2.CreateAndAddPublisher('vtkMRMLROS2PublisherPoseStampedNode', '/my_matrix
↪')
# run `ros2 topic echo /my_matrix` in a terminal to see the output
```

(continues on next page)

(continued from previous page)

```
mat = vtk.vtkMatrix4x4()
mat.SetElement(0, 3, 3.1415) # Modify the matrix so we can see something changing
pubMatrix.Publish(mat)
```

```
auto pubString = rosNode->CreateAndAddPublisherNode("vtkMRMLROS2PublisherStringNode", "/
↪my_string");
// run ros2 topic echo /my_string in a terminal to see the output
pubString->Publish("my first string");
```

To remove the publisher node, use the method `vtkMRMLROS2NodeNode::RemoveAndDeletePublisherNode`. This method takes one parameter:

- the topic name (`std::string`)

4.3.2 Subscribers

To create a new subscriber, one should use the MRML ROS2 Node method `vtkMRMLROS2NodeNode::CreateAndAddSubscriberNode`. This method takes two parameters:

- the class (type) of subscriber to be used. See :*Publishers*.
- the topic name (`std::string`).

Subscriber nodes get updated when the ROS2 node is spun. Users can set their own callback to act on newly received messages using an observer on the MRML ROS subscriber node. The last message received can be retrieved using `GetLastMessage`.

Python

C++

```
rosLogic = slicer.util.getModuleLogic('ROS2')
rosNode = rosLogic.GetDefaultROS2Node()
subString = rosNode.CreateAndAddSubscriberNode('vtkMRMLROS2SubscriberStringNode', '/my_
↪string')
# run `ros2 topic pub /my_string` to send a string
m_string = sub.GetLastMessage()
# alternate, get a string with the full message
m_string_yaml = sub.GetLastMessageYAML()
# since the subscriber is a MRML node, you can also create an observer (callback)
# to trigger some code when a new message is received
observerId = subString.AddObserver('ModifiedEvent', myCallback)
```

```
auto pubString = rosNode->CreateAndAddSubscriberNode("vtkMRMLROS2SubscriberStringNode",
↪"/my_string");
// run ros2 topic echo /my_string in a terminal to see the output
pubString->Publish("my first string");
```

To remove the subscriber node, use the method `vtkMRMLROS2NodeNode::RemoveAndDeleteSubscriberNode`. This method takes one parameter:

- the topic name (`std::string`)

4.4 Parameters

This version of SlicerROS2 only supports parameter clients, i.e. retrieving parameters held by other ROS nodes (i.e. ROS processes running along Slicer).

The parameter MRML ROS node is slightly different from the other MRML ROS nodes implemented in SlicerROS2. A ROS parameter is fully identified by the ROS node *n* that hold the parameter and the parameter name *p* so we could have an implementation that would require one MRML ROS node for each parameter. In practice, this can lead to way too many MRML nodes. For example, if you have two parameters *p1* and *p2* held by the same ROS node *n1*, we would have to create two MRML ROS nodes, *n1p1* and *n1p2*. Since the ROS 2 libraries provide a single message to retrieve all the parameters help by a single node, we decided to require one MRML ROS node per ROS node observed. This MRML ROS node can then observe all the parameters held by the ROS node.

To create a new parameter node, one should use the MRML ROS2 Node method `vtkMRMLROS2NodeNode::CreateAndAddParameterNode`. This method takes one parameter:

- the name of the ROS node that holds the parameters you wish to retrieve (`std::string`). This is an actual ROS node, running along Slicer.

Once the *vtkMRMLROS2ParameterNode* observing the ROS node *n1* is added, one needs to specify which parameters to observe using the method *vtkMRMLROS2ParameterNode::AddParameter*.

Parameter nodes get updated when the ROS2 node is spun. Users can set their own callback to act on newly received messages using an observer on the MRML ROS parameter node.

ROS supports a limited number of types to encode parameters (ROS2 parameters). Most types are supported in SlicerROS2 and we provide methods to determine the type at runtime. The recommended steps are:

- check if the parameter exists using `IsParameterSet(parameterName)`
- get the parameter type using `GetParameterType(parameterName)`
- use the correct accessor based on the parameter's type, for example `GetParameterAsString(parameterName)`

Table 2: Parameter types and accessors

Type	Accessor
"bool"	`GetParameterAsBool`
"integer"	`GetParameterAsInteger`
"double"	`GetParameterAsDouble`
"string"	`GetParameterAsString`
"bool_array"	`GetParameterAsVectorOfBools`
"integer_array"	`GetParameterAsVectorOfIntegers`
"double_array"	`GetParameterAsVectorOfDoubles`
"string_array"	`GetParameterAsVectorOfStrings`

For convenience, we also provide the method `vtkMRMLROS2ParameterNode::PrintParameter` which will provide a human readable description of the parameter whatever the type is (using `std::string`).

Python

C++

```
rosLogic = slicer.util.getModuleLogic('ROS2')
rosNode = rosLogic.GetDefaultROS2Node()
# setup to get parameter robot_description from node state_publisher
parameterNode = rosNode.CreateAndAddParameterNode('state_publisher')
```

(continues on next page)

(continued from previous page)

```
parameterNode.AddParameter('robot_description')
# check if parameter is set, C++ example is more detailed
if parameterNode.IsParameterSet('robot_description'):
  # then check the type
  if parameterNode.GetParameterType('robot_description') == 'string':
    robotDescription = parameterNode.GetParameterAsString('robot_description')
```

Setup:

```
// setup to get parameter robot_description from node state_publisher
auto parameterNode = rosNode->CreateAndAddParameterNode("state_publisher");
parameterNode->AddParameter("robot_description");
// add a callback
parameterNode->AddObserver(vtkMRMLROS2ParameterNode::ParameterModifiedEvent,
                           this, &myClassType::Callback);
```

Callback:

```
// ideally in callback but can be used in a busy loop
if (!parameterNode->IsParameterSet("robot_description")) {
  vtkErrorMacro(<< "parameter \"robot_description\" is not set");
  return;
}
if (parameterNode->GetParameterType("robot_description") != "string") {
  std::string outType = parameterNode->GetParameterType("robot_description");
  vtkErrorMacro(<< "parameter \"robot_description\" is of type " << outType << " and not␣
↪string.");
  return;
}
std::string robotDescription = parameterNode->GetParameterAsString("robot_description");
```

To remove the broadcaster node, use the method `vtkMRMLROS2NodeNode::RemoveAndDeleteParameterNode`. This method takes one parameter:

- the monitored node name (`std::string`) i.e. 'robot_state_publisher'

4.5 Tf2

For Tf2, there is no need to support multiple data types since Tf2's API exclusively uses `geometry_msgs::msg::TransformStamped`. On the Slicer side, the classes `vtkMRMLROS2Tf2BroadcasterNode` and `vtkMRMLROS2Tf2LookupNode` support both `vtkMatrix4x4` and `vtkMRMLTransformNode`.

Tf2 lookups use a Tf2 buffer to store all the Tf2 messages (broadcasts) sent by all the ROS nodes. For the SlicerROS2 module, we decided to add a Tf2 buffer as a private data member of the `vtkMRMLROS2NodeNode` since most users will never need a direct access to the Tf2 buffer. The Tf2 lookups are performed when the node node is spun.

4.5.1 Broadcasts

To create a new Tf2 broadcaster, one should use the MRML ROS2 Node method `vtkMRMLROS2NodeNode::CreateAndAddTf2BroadcasterNode`. This method takes two parameters:

- the parent ID (`std::string`)
- the child ID (`std::string`)

Broadcasters are triggered by calling the `Broadcast` method. It is also possible to set the Tf2 broadcast as an observer for an existing `vtkMRMLTransformNode` using the method `ObserveTransformNode`. The broadcast will then automatically occur when the observed transform node is modified.

Python

C++

```
rosLogic = slicer.util.getModuleLogic('ROS2')
rosNode = rosLogic.GetDefaultROS2Node()
broadcaster = ros2Node.CreateAndAddTf2BroadcasterNode('Parent', 'Child')
# Broadcast a 4x4 matrix
broadcastedMat = vtk.vtkMatrix4x4()
broadcastedMat.SetElement(0, 3, 66.0) # Set a default value
broadcaster.Broadcast(broadcastedMat)
```

```
auto broadcaster = rosNode.CreateAndAddTf2BroadcasterNode("Parent", "Child");
# Broadcast a 4x4 matrix
vtkSmartPointer<vtkMatrix4x4> broadcastedMat = vtkMatrix4x4::New();
broadcastedMat->SetElement(0, 3, 66.0);
broadcaster->Broadcast(broadcastedMat);
```

To remove the broadcaster node, use the method `vtkMRMLROS2NodeNode::RemoveAndDeleteTf2BroadcasterNode`. This method takes two parameters:

- the parent ID (`std::string`)
- the child ID (`std::string`)

4.5.2 Lookups

To create a new Tf2 lookup, one should use the MRML ROS2 Node method `vtkMRMLROS2NodeNode::CreateAndAddTf2LookupNode`. This method takes two parameters:

- the parent ID (`std::string`)
- the child ID (`std::string`)

The class `vtkMRMLROS2Tf2LookupNode` is derived from `vtkMRMLTransformNode` so it can be used as any other transformation in the MRML scene.

Lookup nodes get updated when the ROS2 node is spun. Users can set their own callback to act on updated transformations using an observer on the MRML ROS subscriber node. The last transformation received can be retrieved using `GetMatrixTransformToParent`.

Python

C++

```
rosLogic = slicer.util.getModuleLogic('ROS2')
rosNode = rosLogic.GetDefaultROS2Node()
lookupNode = self.ros2Node.CreateAndAddTf2LookupNode('Parent', 'Child')
# get the transform "manually"
lookupMat = lookupNode.GetMatrixTransformToParent()
# or use an observer
observerId = lookupNode.AddObserver(slicer.vtkMRMLTransformNode.TransformModifiedEvent,␣
↪observer.Callback)
```

```
auto lookup = rosNode.CreateAndAddTf2LookupNode("Parent", "Child");
# Broadcast a 4x4 matrix
vtkSmartPointer<vtkMatrix4x4> lookupMat = vtkMatrix4x4::New();
lookupMat->GetMatrixTransformToParent(lookupMat);
```

To remove the lookup node, the method `vtkMRMLROS2NodeNode::RemoveAndDeleteTf2LookupNode`. This method takes two parameters:

- the parent ID (`std::string`)
- the child ID (`std::string`)

4.6 Robots

To create a new Robot node, one can either use the UI (instructions in Section 3.3) or create the robot programmatically with the following commands. The convenience function `vtkMRMLROS2NodeNode::CreateAndAddRobotNode` was added to the module logic that accepts three arguments (`std::string robotName`, `std::string parameterNodeName`, `std::string parameterName`).

Python

C++

```
rosLogic = slicer.util.getModuleLogic('ROS2')
rosNode = rosLogic.GetDefaultROS2Node()
rosNode.CreateAndAddRobotNode('PSM','PSM1/robot_state_publisher','robot_description') #␣
↪Using the PSM as an example
```

```
auto robot = rosNode.CreateAndAddRobotNode("PSM","PSM1/robot_state_publisher","robot_
↪description");
```

The robot node creates an observer on the parameter node that contains the robot description. If the parameter node is modified (indicating that the robot description is available), it begins the process of loading the visuals for the robot into the scene. This process involves: parsing the URDF file, creating a list of Tf2 lookups in the scene, creating the models for each link of the robot and applying the correct color and offset position relative to the base of the robot. Once the visuals have been created, the Tf2 lookups start to check the Tf2 buffer and update the position of the model according to the joint state publisher.

To remove the robot, use the "Remove robot" button on the UI or the method `vtkMRMLROS2NodeNode::RemoveAndDeleteRobotNode`. This method takes one parameter:

- the robot name (`std::string`)

CHAPTER

FIVE

LIMITATIONS

- We only support STL and OBJ meshes for the `visual` defined in the URDF. If any `visual` is defined using a `geometry` (sphere, box…) or another mesh format, it will not be displayed in Slicer
- ROS namespaces are not supported yet. The current implementation allows multiple nodes but doesn't provide a parameter to set the ROS namespace. Let us know if this is something you need.
- The following ROS functionalities are missing:
  - Services
  - Parameter server (`vtkMRMLROS2ParameterNode` only works as a client)
- The current implementation assumes that all ROS2 nodes (`vtkMRMLROS2NodeNode`) added to the scene should be spun by the module's logic. This doesn't provide an option for users to control how their ROS2 nodes spin (including rate).
- Saving and reloading the scene as a MRML scene has not been extensively tested and might not work.

CHAPTER

SIX

TESTS

6.1 Introduction

We provide some unit tests for the Slicer ROS module. The unit tests rely on the automatically generated Python wrapper to to test the C++ MRML nodes. These tests assumes you've installed the default ROS2 desktop packages and use the turtlesim examples provided along ROS2.

To test the MRML ROS2 parameter node, the ROS2 turtlesim nodes are started in the background so you will see some windows popping up.

To test the subscribers and publishers, the unit tests subscribe and publish to the same topic so we can send and receive from the same node. This also indirectly tests the conversion methods.

Finally, for Tf2 lookups and broadcasts, we use the strategy used to test the subscribers and publishers. The test broadcasts a known transform and uses a lookup to retrieve the value of the Tf2 buffer.

6.2 Running the unit tests

```
tests = slicer.util.getModuleLogic('ROS2Tests')
tests.run()
```

The tests intentionally attempts to perform commands that should fail therefore you will see a few error and error messages displayed in the Python console. To see the result of the tests, you will have to scroll up.

CHAPTER

SEVEN

LINKS

- SlicerROS2: https://github.com/rosmed/slicer_ros2_module/
- ROS for Medical Robotics: https://rosmed.github.io/
- Slicer: https://www.slicer.org/
- ROS2: https://www.ros.org/