QML ROS2 Plugin

Stefan Fabian

May 10, 2024

TABLE OF CONTENTS

CHAPTER

ONE

ACTIONCLIENT

An action client can be used to send goals to an ActionServer. One ActionClient has to have a single type of action but can send multiple goals simultaneously. Process can be tracked using either a goal handle manually or using callbacks.

An ActionClient can be created using the *Ros2 Singleton* as follows:

```
Item {
  // ...
  property var fibonacciClient: Ros2.createActionClient("fibonacci", "action_tutorials_
↪interfaces/action/Fibonacci")
  // ...
}
```

In this example an action client is created using the `action_tutorials_interfaces/action/Fibonacci` action (it is important to use the complete action type here, not any part like the ActionGoal) using the name fibonacci on which an ActionServer should be registered (You can use the action tutorials fibonacci_server).

To send a goal, you can use the *sendGoalAsync* function:

```
if (!fibonacciClient.ready) {
  Ros2.error("Action client not ready yet!")
  return
}
goal_handle = fibonacciClient.sendGoalAsync({ order: numberInput.value }, {
    // These callbacks are optional
    onGoalResponse(goal_handle) {
      if (!goal_handle) {
        // Goal was rejected
        return
      }
      // Handle goal accepted
    },
    onFeedback(goal_hamdle, feedback) {
      // Handle feedback from action server
    },
    onResult(result) {
      // Handle result from action server
      let goalId = result.goalId
      if (result.code == ActionGoalStatus.Succeeded) {
        // Handle success
        let goalResult = result.result
      } else if (result.code == ActionGoalStatus.Canceled) {
        // Handle canceled
```

(continues on next page)

(continued from previous page)

```
    } else if (result.code == ActionGoalStatus.Aborted) {
      // Handle aborted
    }
  }
})
```

The `sendGoalAsync` function takes 2 parameters: the goal and an optional options object with optional callbacks for *onGoalResponse*, *onFeedback* and *onResult*. Both callbacks are optional. It returns a GoalHandle which can be used query to state of the goal or to cancel the goal. The goal_handle passed to the callbacks and the one returned are the same.

1.1 API

class **ActionClient** : public QObjectRos2

Public Functions

QObject ***sendGoalAsync**(const QVariantMap &goal, QJSValue options = QJSValue())

Sends a goal to the action server if it is connected.

Parameters

- **goal** – The goal that is sent to the action server.
- **options** – An object with optional callback members for:
  - goal_response: onGoalResponse(goal_handle) goal_handle will be null if goal was rejected
  - feedback: onFeedback(goal_handle, feedback_message)
  - result: onResult(wrapped_result) where wrapped_result has a *goalId*, result *code* and *result* message.

Returns

null if the action server is not connected, otherwise a *GoalHandle* keeping track of the state of the goal.

void **cancelAllGoals**()

Cancels all goals that are currently tracked by this client.

void **cancelGoalsBefore**(const qml_ros2_plugin::*Time* &time)

Cancels all goals that were sent at and before the given ROS time by this client. Use Time.now() to obtain the current ROS time which can differ from the actual time.

void **cancelGoalsBefore**(const QDateTime &time)

Cancels all goals that were sent at and before the given ROS time by this client. Use Time.now() to obtain the current ROS time which can differ from the actual time.

Signals

void **serverReadyChanged**()

Emitted when the connected status changes, e.g., when the client connected to the server.

Properties

bool ready

True if the *ActionClient* is connected to the ActionServer, false otherwise.

QString actionType

The type of the action. Example: action_tutorials_interfaces/action/Fibonacci.

class **GoalHandle** : public QObjectRos2

Public Functions

void **cancel**()

Sends a cancellation request to the ActionServer.

Properties

qml_ros2_plugin::action_goal_status::GoalStatus status

The goal status in form of an action_goal_status enum value: Aborted, Accepted, Canceled, Canceling, Executing, Succeeded, Unknown

enum qml_ros2_plugin::action_goal_status::**GoalStatus**

Values:

enumerator **Aborted**

enumerator **Accepted**

enumerator **Canceled**

enumerator **Canceling**

enumerator **Executing**

enumerator **Succeeded**

enumerator **Unknown**

CHAPTER

TWO

ARRAYS

Due to the lazy copy mechanism, arrays differ from the standard access in javascript. Because the array is not copied into a QML compatible array container, access happens with methods.

Example: Instead of `path.to.array[1].someproperty`, you would write `path.to.array.at(1).someproperty`.

If you need the array as a javascript array, you can use *toArray* to copy the entire array and return it as a javascript array. The copy is only performed on the first call, subsequent calls should have less overhead.

2.1 API

class **Array**

View on an array field of a message. This allows access on array elements with lazy copy mechanism. Copies of an *Array* point to the same data and modifications of one array will be mirrored by the other.

Public Functions

QVariant **at**(int index) const

If the index is out of the bounds of the array, an empty QVariant is returned.

Parameters
index – Index of the retrieved element.

Returns
The array element at the given index.

void **spliceList**(int start, int delete_count, const QVariantList &items)

Changes the array content by removing delete_count elements at index and inserting the elements in items. This method can be used to remove, replace or add elements to the array.

> **Warning:** If the operation is not limited to the end of the array, it requires a deep copy of the message array.

Parameters

- **start** – The index at which to start changing the array. If greater than the length of the array, start will be set to the length of the array. If negative, it will begin that many elements from the end of the array (with origin -1, meaning -n is the index of the nth last element

and is therefore equivalent to the index of array.length - n). If the absolute value of start is greater than the length of the array, it will begin from index 0.

- **delete_count** – The number of elements to delete starting at index start. If delete_count is greater than the number of elements after start, all elements from start to the length of the array are removed. If delete_count is 0, no elements are removed, e.g., for a insert only operation.
- **items** – The items that will be inserted at start.

void **push**(const QVariant &value)

Adds the given value to the end of the array.

Parameters
value – The item that is added.

void **unshift**(const QVariant &value)

Adds the given value to the front of the array.

> **Warning:** This requires a deep copy of the message array on first call whereas appending can be done without copying the array.

Parameters
value – The item that is added.

QVariant **pop**()

Removes the last element and returns it.

Returns
The removed element or an empty QVariant if the array is empty.

QVariant **shift**()

Removes the first element and returns it.

> **Warning:** This requires a deep copy of the message array on first call whereas appending can be done without copying the array.

Returns
The removed element or an empty QVariant if the array is empty.

QVariantList **toArray**()

Converts the array to a QVariantList which can be used in place of a JS array in QML. This method performs a deep copy of the message array on the first call.

Returns
The array as a QVariantList.

Properties

`int length`

The length of the array, i.e., the number of elements.

CHAPTER

THREE

EXAMPLES

You can find the described example QML files in the qml_ros2_plugin repo examples directory.

3.1 Subscriber

The subscriber example demonstrates how to create a `Subscriber` in QML using the *QML ROS Plugin*.

You can run the example using the `qmlscene` executable:

```
qmlscene subscriber.qml
```

3.2 Publisher

The publisher example publishes an `example_interfaces/msg/Int32` message on the topic that the subscriber example subscribes to. Coincidentally, the two examples can very well be used together.

To run, run:

```
qmlscene publisher.qml
```

3.3 URDF Tutorial UI

This example combines several of the functionalities provided by this library and presents a user interface for the `urdf_sim_tutorial` diff drive example.

First, launch the example:

```
roslaunch urdf_sim_tutorial 13-diffdrive.launch
```

Next, launch the example UI:

```
qmlscene urdf_tutorial_combined.qml
```

It provides a top down view on the position of the robot and sliders to control the forward and angular movement.

3.4 Logging

This example demonstrates the logging functionality detailed in *Logging*. The "Output logging level" sets the minimum logging level that is printed whereas the "Message logging level" sets the level of the message that is logged when you click the *Log* button.

To run, run:

```
qmlscene logging.qml
```

CHAPTER

FOUR

IMAGETRANSPORT

Seeing what the robot sees is one of the most important features of any user interface. To enable this, this library provides the *ImageTransportSubscription*. It allows easy subscription of camera messages and provides them in a QML native format as a VideoSource.

Example:

```
ImageTransportSubscription {
  id: imageSubscription
  // Enter a valid image topic here
  topic: "/front_rgbd_cam/color/image_rect_color"
  // This is the default transport, change if compressed is not available
  defaultTransport: "compressed"
}

VideoOutput {
  anchors.fill: parent
  // Can be used in increments of 90 to rotate the video
  orientation: 90
  source: imageSubscription
}
```

The *ImageTransportSubscription* can be used as the source of a `VideoOutput` to display the camera images as they are received. Additionally, it can be configured to show a blank image after x milliseconds using the `timeout` property which is set to 3000ms (3s) by default. This can be disabled by setting the `timeout` to 0. If you do not want the full camera rate, you can throttle the rate by setting `throttleRate` to a value greater than 0 (which is the default and disables throttling). E.g. a rate of 0.2 would show a new frame every 5 seconds. Since there is no ROS functionality for a throttled subscription, this means the `image_transport::Subscriber` is shut down and newly subscribed for each frame. This comes at some overhead, hence, it should only be used to throttle to low rates <1. To avoid all throttled subscribers subscribing at the same time causing huge network spikes, the throttled rates are load balanced by default. This can be disabled globally using *ImageTransportManager::setLoadBalancingEnabled* which is available in QML using the singleton `ImageTransportManager`.

4.1 API

class **ImageTransportSubscription** : public QObjectRos2

Properties

QAbstractVideoSurface * videoSurface

Interface for QML. This is the surface the images are passed to.

QString topic

The image base topic (without image_raw etc.). This value may change once the subscriber is connected and private topic names or remappings were evaluated.

QString defaultTransport

The default transport passed as transport hint. May be overridden by a parameter. (Default: compressed)

bool subscribed

Whether or not this ImageTransportSubscriber is subscribed to the given topic (readonly)

int networkLatency

The latency from the sender to the received time in ms not including the conversion latency before displaying. This latency is based on the ROS time of the sending and receiving machines, hence, they need to be synchronized. (readonly)

int processingLatency

The latency (in ms) from the reception of the image until it is in a displayable format. (readonly)

int latency

The full latency (in ms) from the camera to your display excluding drawing time. (readonly)

double framerate

The framerate of the received camera frames in frames per second. (readonly)

int timeout

The timeout when no image is received until a blank frame is served. Set to 0 to disable and always show last frame. Default is 3000 ms.

double throttleRate

The update rate to throttle image receiving in images per second. Set to 0 to disable throttling. Default is 0 (disabled).

bool enabled

Whether the subscriber is active or not. Setting to false will shut down subscribers.

class **ImageTransportManagerSingletonWrapper** : public QObject

Public Functions

void **setLoadBalancingEnabled**(bool value)

Sets whether the manager should try to balance throttled subscriptions_ to ensure they don't update at the same time which would result in network spikes.

CHAPTER

FIVE

LOGGING

Logging is done using the *Ros2 Singleton*.

5.1 Output

To log a message you can use one of the following methods `debug`, `info`, `warn`, `error` and `fatal`.

```
Button {
  // ...
  onClicked: Ros2.info("Button clicked.")
}
```

This will produce the following output:

:: code-block:: bash

    [INFO] [1583062360.048922959]: Button clicked.

and publish the following on `/rosout` (unless `NoRos2out` was specified in the `Ros2InitOptions`).

:: code-block:: bash

    header:
        seq: 1 stamp:

            secs: 1583062360 nsecs: 49001300

        frame_id: ‘’

    level: 2 name: “/qml_logging_demo” msg: “Button clicked.” file: “/home/stefan/qml_ros2_plugin/examples/logging.qml” function: “onClicked” line: 130 topics: [/rosout]

The `file`, `function` and `line` info is automatically extracted when you call the log function.

5.2 Set Verbosity

You can change the verbosity, i.e., the minimal level of logging message that is printed (and published if enabled), using `Ros2.console.setLoggerLevel`. By default the logging level is set to *Info*. To enable debug messages you can set it to *Debug* as follows:

```
Ros2.console.setLoggerLevel(Ros2.console.defaultName, Ros2ConsoleLevels.Debug);
```

The first argument to that method is the name of the console to which the logging is printed. These are identifiers used by ros to enable you to change the verbosity of a submodule of your node using `rqt_console`.

You can optionally change to which console you're writing by passing a second argument to the logging function, e.g., `debug("Some message", "ros.my_pkg.my_submodule")`.
This name should contain only letters, numbers, dots and underscores.
Important: The name has to start with "`ros.`".

By default the value of `Ros2.console.defaultName` is used which evaluates to `ros.qml_ros2_plugin`.

Possible values for the console level are: `Debug`, `Info`, `Warn`, `Error` and `Fatal`.

CHAPTER

SIX

PUBLISHERS

A Publisher is used to publish messages on a given topic for delivery to subscribers.

6.1 Simple Example

Contrary to Subscribers, a Publisher can not be instantiated but is created using a factory method of the Ros2 singleton.

```
/* ... */
ApplicationWindow {
  property var intPublisher: Ros2.advertise("/intval", "example_interfaces/msg/Int32",␣
↪10)
  /* ... */
}
```

In order, the arguments are the `topic`, the `type` and the `queueSize` (defaults to 1). Additional QoS options are currently not supported.

To publish a message using our Publisher, we can simply use the `intPublisher` variable defined earlier.

```
SpinBox {
  id: numberInput
}

Button {
  onClicked: {
    intPublisher.publish({ data: numberInput.value })
  }
}
```

where we pass an object with a data field containing the (integer) number of the `SpinBox`. This is according to the `example_interfaces/msg/Int32` message definition.

6.2 API

6.2.1 Publisher

class **Publisher** : public QObjectRos2

Public Functions

unsigned int **getSubscriptionCount**()

Returns
The number of subscribers currently connected to this *Publisher*.

bool **publish**(const QVariantMap &msg)

Sends a message to subscribers currently connected to this *Publisher*.

Parameters
msg – The message that is published.

Returns
True if the message was sent successfully, false otherwise.

Signals

void **advertised**()

Fired once this *Publisher* was advertised. This is either done at construction or immediately after ROS is initialized. Since this is only fired once, you should check if the *Publisher* is already advertised using the isAdvertised property.

Properties

QString type

The type of the published messages, e.g., geometry_msgs/Pose. (readonly)

QString topic

The topic this *Publisher* publishes messages on. This property is only valid if the publisher is already advertised! (readonly)

quint32 queueSize

The queue size of this *Publisher*. This is the maximum number of messages that are queued for delivery to subscribers at a time. (readonly)

bool isAdvertised

Whether or not this publisher has advertised its existence on its topic. Reasons for not being advertised include ROS not being initialized yet. (readonly)

CHAPTER

SEVEN

QUICKSTART

This library provides convenient access of ROS2 concepts and functionalities in QML.

7.1 Installation

Note: Currently, only Linux is supported. Other platforms have not been tested.

7.1.1 From Source

To install `qml_ros2_plugin` from source, clone the repo. Now, you have two options: You can either install the plugin in your ROS2 overlay which makes the plugin available only if you've sourced the overlay in your environment. Alternatively, you can enable the global install, to install it system-wide on linux.

Local Install

`cd` into your workspace root directory and `colcon build`. Re-source your `install/setup.bash`.

Global install

`cd` into the repo folder. To install create a build folder, `cd` into that folder and run `cmake -DGLOBAL_INSTALL=ON ..` followed by `make` and `sudo make install`.

```
mkdir build && cd build
cmake -DGLOBAL_INSTALL=ON ..
make -j8 # Replace 8 by the number of cpu cores
sudo make install
```

7.2 Usage

To use the plugin import `Ros2` in QML.

```
import Ros2 1.0
```

Now, you can use the provided components such as `Subscription` and `TfTransform` and the *Ros2 Singleton* to create a `Publisher`, a `ServiceClient`, or an `ActionClient`.

As a simple example, a `Subscription` can be created as follows:

```
Subscription {
  id: mySubscription
  topic: "/intval"
}
```

For more in-depth examples, check out the *Examples* section.

7.2.1 Initialization

Before a `Subscription` can receive messages, a `Publisher` can publish messages, etc. the node has to be initialized.

```
ApplicationWindow {
  /* ... */
  Component.onCompleted: {
    Ros2.init("node_name");
  }
}
```

7.2.2 Shutdown

To make your application quit when ROS shuts down, e.g., because of a `Ctrl+C` in the console, you can connect to the `Shutdown` signal:

```
ApplicationWindow {
  Connections {
    target: Ros2
    function onShutdown() {
      Qt.quit()
    }
  }
  /* ... */
}
```

For more on that, check out the *Ros2 Singleton*.

CHAPTER

EIGHT

ROS2 SINGLETON

The Ros2 singleton provides interfaces to static methods and convenience methods.

In QML it is available as Ros2, e.g.:

```
if (Ros2.ok()) console.log("Ros2 is ok!")
```

8.1 ROS Initialization

First, you need to initialize the node used by your QML application, e.g., in the onCompleted handler:

```
Component.onCompleted: {
  Ros2.init("node_name")
}
```

You can also conditionally initialize by checking if it was already initialized using Ros2.isRosInitialized. As described in the API documentation for *Ros2.init*, you can pass either just the node name or additionally use provided command line args instead of the command line args provided to your executable.

8.2 Query Topics

You can also use the Ros2 singleton to query the available topics. Currently, three methods are provided:

- QStringList queryTopics(const QString &datatype = QString())
  Queries a list of topics with the given datatype or all topics if no type provided.
- QList<TopicInfo> queryTopicInfo()
  Retrieves a list of all advertised topics including their datatype. See TopicInfo
- QString queryTopicType(const QString &name)
  Retrieves the datatype for a given topic.

Example:

```
// Retrieves a list of topics with the type sensor_msgs/Image
var topics = Ros2.queryTopics("sensor_msgs/msg/Image")
// Another slower and less clean method of this would be
var cameraTopics = []
var topics = Ros2.queryTopicInfo()
for (var i = 0; i < topics.length; ++i) {
```

(continues on next page)

(continued from previous page)

```
  if (topics[i].datatype == "sensor_msgs/msg/Image") cameraTopics.push(topics[i].name)
}
// The type of a specific topic can be retrieved as follows
var datatype = Ros2.queryTopicType("/topic/that/i/care/about")
// Using this we can make an even worse implementation of the same functionality
var cameraTopics = []
var topics = Ros2.queryTopics() // Gets all topics
for (var i = 0; i < topics.length; ++i) {
  if (Ros2.queryTopicType(topics[i]) == "sensor_msgs/msg/Image") cameraTopics.
↪push(topics[i])
}
```

8.3 Create Empty Message

You can also create empty messages and service requests as javascript objects using the Ros2 singleton.

```
var message = Ros2.createEmptyMessage("geometry_msgs/msg/Point")
// This creates an empty instance of the mssage, we can override the fields
message.x = 1; message.y = 2; message.z = 1

// Same can be done with service requests
var serviceRequest = Ros2.createEmptyServiceRequest("std_srvs/srv/SetBool")
// This creates an empty instance of the service request with all members set to their
// default, we can override the fields
serviceRequest.data = true
```

8.4 Logging

The Ros2 singleton also provides access to the Ros2 logging functionality. See *Logging*.

8.5 IO

You can also save and read data that can be serialized in the yaml format using:

```
var obj = {"key": [1, 2, 3], "other": "value"}
if (!Ros2.io.writeYaml("/home/user/file.yaml", obj))
  Ros2.error("Could not write file!")
// and read it back
obj = Ros2.io.readYaml("/home/user/file.yaml")
if (!obj) Ros2.error("Failed to load file!")
```

8.6 API

class **TopicInfo**

Properties

QString name

The name of the topic, e.g., /front_camera/image_raw.

QStringList datatypes

The datatype(s) of the topic, e.g., sensor_msgs/msg/Image.

class **IO**

Public Functions

bool **writeYaml**(QString path, const QVariant &value)

Writes the given value to the given path in the yaml format.

Parameters

- **path** – The path to the file.
- **value** – The value to write.

Returns

True if successful, false otherwise.

QVariant **readYaml**(QString path)

Reads a yaml file and returns the content in a QML compatible structure of 'QVariantMap's and 'QVariantList's.

Parameters

path – The path to the file.

Returns

A variant containing the file content or false.

class **Ros2QmlSingletonWrapper** : public QObject

Public Functions

void **init**(const QString &name, quint32 options = 0)

Initializes the ros node with the given name and the command line arguments passed from the command line.

Parameters

- **name** – The name of the ROS node.
- **options** – The options passed to ROS, see ros_init_options::Ros2InitOption.

bool **ok**() const

Can be used to query the state of ROS.

Returns
False if it's time to exit, true if still ok.

qml_ros2_plugin::*Time* **now**() const

Returns
The current time as given by the node's clock (if initialized, otherwise rclcpp::Time())

QString **getName**()

Returns the name of the node. Returns empty string before ROS node was initialized.

QString **getNamespace**()

Returns the namespace of the node. Returns empty string before ROS node was initialized.

QStringList **queryTopics**(const QString &datatype = QString()) const

Queries the internal ndoe for its topics or using the optional datatype parameter for all topics with the given type.

Parameters
datatype – The message type to filter topics for, e.g., sensor_msgs/Image. Omit to query for all topics.

Returns
A list of topics that matches the given datatype or all topics if no datatype provided.

QList<qml_ros2_plugin::*TopicInfo*> **queryTopicInfo**() const

Queries the internal node for its topics and their type.

Returns
A list of *TopicInfo*.

QStringList **queryTopicTypes**(const QString &name) const

Queries the internal node for a topic with the given name.

Parameters
name – The name of the topic, e.g., /front_camera/image_raw.

Returns
The types available on the topic if found, otherwise an empty string.

QVariant **createEmptyMessage**(const QString &datatype) const

Creates an empty message for the given message type, e.g., "geometry_msgs/Point". If the message type is known, an empty message with all members set to their default is returned. If the message type is not found on the current machine, a warning message is printed and null is returned.

Parameters
datatype – The message datatype.

Returns
A message with all members set to their default.

QVariant **createEmptyServiceRequest**(const QString &datatype) const

Creates an empty service request for the given service type, e.g., "std_srvs/SetBool". If the service type is known, an empty request is returned with all members of the request message set to their default values. If the service type is not found on the current machine, a warning message is printed and null is returned.

Parameters
datatype – The service datatype. NOT the request datatype.

Returns
A request message with all members set to their default.

QObject *`getLogger`(const QString &name = QString())

Get the logger with the given name or if no name is passed, the node's logger.

Parameters
`name` – (Optional) Name of the logger.

Returns
An instance of Logger wrapping the requested rclcpp::Logger.

QJSValue `debug`()

Logs debug with the node's logger.

QJSValue `info`()

Logs info with the node's logger.

QJSValue `warn`()

Logs warn with the node's logger.

QJSValue `error`()

Logs error with the node's logger.

QJSValue `fatal`()

Logs fatal with the node's logger.

QObject *`createPublisher`(const QString &topic, const QString &type, quint32 queue_size = 1)

Creates a *Publisher* to publish ROS messages.

Parameters

- `type` – The type of the messages published using this publisher.
- `topic` – The topic on which the messages are published.
- `queue_size` – The maximum number of outgoing messages to be queued for delivery to subscribers.

Returns
A *Publisher* instance.

QObject *`createSubscription`(const QString &topic, quint32 queue_size = 1)

Creates a Subscriber to createSubscription to ROS messages. Convenience function to create a subscriber in a single line.

Parameters

- `topic` – The topic to createSubscription to.
- `queue_size` – The maximum number of incoming messages to be queued for processing.

Returns
A Subscriber instance.

QObject *`createSubscription`(const QString &topic, const QString &message_type, quint32 queue_size = 1)

Creates a Subscriber to createSubscription to ROS messages. Convenience function to create a subscriber in a single line.

Parameters

- `topic` – The topic to createSubscription to.

- **message_type** – The type of the messages to subscribe to on the topic.
- **queue_size** – The maximum number of incoming messages to be queued for processing.

Returns
A Subscriber instance.

QObject *createServiceClient(const QString &name, const QString &type)

Creates a *ServiceClient* for the given type and the given name.

Parameters

- **name** – The name of the service to connect to.
- **type** – The type of the service. Example: example_interfaces/srv/AddTwoInts

Returns
An instance of *ServiceClient*.

QObject *createActionClient(const QString &name, const QString &type)

Creates an *ActionClient* for the given type and the given name.

Parameters

- **name** – The name of the action server to connect to.
- **type** – The type of the action. Example: action_tutorials_interfaces/action/Fibonacci

Returns
An instance of *ActionClient*.

Signals

void initialized()

Emitted once when ROS was initialized.

void shutdown()

Emitted when this ROS node was shut down and it is time to exit.

CHAPTER

NINE

SERVICES

You can create a `ServiceClient` using the *Ros2 Singleton* Here's a short modified example of the service example provided in the *Examples*.

```
Button {
  property var serviceClient: Ros2.createServiceClient("/add_two_ints", "example_
↪interfaces/srv/AddTwoInts")
  onClicked: {
    var result = serviceClient.sendRequestAsync(
      { a: 1, b: 3 },
      function (result) {
        textResult.text = !!result ? ("Result: " + result.sum) : "Failed"
      })
  }
}
```

In the first step, a service client is created. Here, the first argument is the `service` that is called and the second is the `type` of the service.

When the button is clicked, the service client is used to send a request with the first argument being the `request` and the second is a callback that is called once the service returns. The callback receives the result of the service call or the boolean value `false` if the call failed. The code will continue execution while the service call is processed, hence, if you want to prohibit concurrent calls to the same service, you'll have to add your own logic to check whether a service call is currently active.

9.1 API

class **ServiceClient** : public QObjectRos2

Public Functions

ServiceClient(QString name, QString type)

Parameters

- **name** – The service topic.
- **type** – The type of the service, e.g., "example_interfaces/srv/AddTwoInts"

bool **isServiceReady**() const

Returns whether the service is ready.

void **sendRequestAsync**(const QVariantMap &req, const QJSValue &callback)

Calls a service asynchronously returning immediately. Once the service call finishes, the optional callback is called with the result if provided.

Parameters

- **req** – The service request, i.e., a filled request message of the service type.
- **callback** – The callback that is called once the service has finished. If the request failed, the callback is called with false.

Properties

bool ready

True if the *ServiceClient* is connected to the Service and the Service is ready, false otherwise.

CHAPTER

TEN

SUBSCRIPTION

A subscription listens for messages on a given topic.

10.1 Simple example

First, let's start with a simple example:

```
Subscription {
  id: mySubscription
  topic: "/intval"
}
```

This creates a subscription on the topic `/intval`. The message type will be determined when the subscription is established. Let's assume the topic publishes an `example_interfaces/msg/Int32` message.

The `example_interfaces/msg/Int32` message is defined as follows:

```
int32 data
```

We can display the published value using a text field:

```
Text {
  text: "Published value was: " + mySubscription.message.data
}
```

Whenever a new message is received on `/intval` the message property is updated and the change is propagated to the text field. Thus, the text field will always display the latest received value.

10.2 Full example

In most cases, the above Subscription is sufficient. However, the Subscription has more properties to give you more fine-grained control.

```
Subscriber {
  id: mySubscriber
  topic: "/intval"
  // Using messageType sets the type explicitly, will not connect to a
  //  publisher if the type does not match
  messageType: "example_interfaces/msg/Int32"
```

(continues on next page)

(continued from previous page)

```
  throttleRate: 30 // Update rate of message property in Hz. Default: 20
  queueSize: 10
  enabled: true // Can be used to pause/unpause the subscription
  onNewMessage: doStuff(message)
}
```

The `queueSize` property controls how many incoming messages are queued for processing before the oldest are dropped. Note that due to the `throttleRate` messages may be dropped even if the `queueSize` is large enough.

The `throttleRate` limits the rate in which QML receives updates from the given topic. By default the Subscriber polls with 20 Hz on the UI thread and will notify of property changes with at most this rate. This is to reduce load and prevent race conditions that could otherwise update the message while QML is using it since the subscriber is receiving messages in a background thread by default.

Using the `enabled` property, the subscription can be enabled and disabled. If the property is set to `false`, the subscription is shut down until it is set to `true` again and subscribes to the topic again. For example, the state of a Subscription can be toggled using a button:

```
Button {
  id: myButton
  state: "active"
  onClicked: {
    mySubscription.enabled = !mySubscription.enabled
    state = state == "active" ? "paused" : "active"
  }
  states: [
    State {
      name: "active"
      PropertyChanges {
        target: myButton
        text: "Unsubscribe"
      }
    },
    State {
      name: "paused"
      PropertyChanges {
        target: myButton
        text: "Subscribe"
      }
    }
  ]
}
```

Whenever a new message is received, the newMessage signal is emitted and the message is passed and can be accessed as `message` which technically refers to the received message and not the message property of the Subscriber. Untechnically, they are the same, though.

Finally, there's also the messageType property which holds the type of the received message, e.g., `example_interfaces/msg/Int32`. If it isn't set, the type is determined from the first available publisher, otherwise, the subscription will only connect to publishers with the correct message type.

10.3 API

class **Subscription** : public QObjectRos2

Public Functions

unsigned int **getPublisherCount**()

Returns
The number of publishers this subscriber is connected to.

Signals

void **newMessage**(QVariant message)

Emitted whenever a new message was received.

Parameters
message – The received message.

Properties

QString topic

The topic this subscriber subscribes to.

quint32 queueSize

The maximum number of messages that are queued for processing. Default: 10.

QVariant message

The last message that was received by this subscriber.

QString messageType

Set to a specific type to subscribe to that type, e.g., geometry_msgs/msg/Pose, otherwise the type is automatically detected and if the topic has multiple available types, one is arbitrarily selected.

int throttleRate

Limits the frequency in which the notification for an updated message is emitted. Default: 20 Hz.

bool enabled

Controls whether or not the subscriber is currently enabled, i.e., able to receive messages. Default: true.

bool subscribed

Indicates whether a subscription is active or not.

CHAPTER

ELEVEN

TF TRANSFORMS

There are two methods for looking up **tf2** transforms.

11.1 Component

The `TfTransform` component can be used to createSubscription to transforms between two frames.

```
TfTransform {
  id: tfTransform
  active: true // This is the default, if false no updates will be received
  sourceFrame: "turtle1"
  targetFrame: "world"
}
```

It provides a `valid` property that indicates if a valid transform has been received. If it is valid, it contains a `transform` property with the stamped transform `geometry_msgs/msg/TransformStamped` and for convenience also a `translation` and a `rotation` property which refer to the translation and rotation in the transform.

Using the `rate` property, you can also change the maximum rate at which the transform is updated.

11.2 Static

You can also use the `TfTransformListener` singleton to look up transforms if you just need it once.

```
Button {
  text: "Look Up"
  onClicked: {
    var transformStamped = TfTransformListener.lookUpTransform(inputTargetFrame.text,␣
↪inputSourceFrame.text)
    if (!transformStamped.valid)
    {
      transformResult.text = "Transform from '" + inputSourceFrame.text + "' to '" +␣
↪inputTargetFrame.text + "' was not valid!\n" +
                             "Exception: " + transformStamped.exception + "\nMessage: "␣
↪+ transformStamped.message
      return
    }
    transformResult.text = "Position:\n" + printVector3(transformStamped.transform.
↪translation) + "\nOrientation:\n" + printRotation(transformStamped.transform.rotation)
```

(continues on next page)

(continued from previous page)

```
  }
}
```

Use the provided `Ros2.now()` static methods to look up at specific time points. For the latest, you can pass `new Date(0)`. Be aware that in JavaScript durations are given in milliseconds.

Warning: Be aware that *canLookUp* can return a `boolean` value or a `string` error message. You should explicitly test for that since strings are truthy, too.

11.3 API

class **TfTransformListener** : public QObject

Public Functions

bool **isInitialized**() const

Returns true if the *TfTransformListener* was already initialized, false otherwise. This does not mean it will already receive tf2 data as Ros2 may not be initialized yet.

QVariant **canTransform**(const QString &target_frame, const QString &source_frame, const rclcpp::Time &time = rclcpp::Time(0), double timeout = 0) const

Checks if a transform is possible. Returns true if possible, otherwise either false or if available a message why the transform failed.

Parameters

- **target_frame** – The frame into which to transform.
- **source_frame** – The frame from which to transform.
- **time** – The time at which to transform in seconds.
- **timeout** – How long to block before failing in milliseconds. Set to 0 for no timeout.

Returns
True if the transform is possible, otherwise an error message (string) if available, false if not.

QVariant **canTransform**(const QString &target_frame, const rclcpp::Time &target_time, const QString &source_frame, const rclcpp::Time &source_time, const QString &fixed_frame, double timeout = 0) const

Checks if a transform is possible. Returns true if possible, otherwise either false or if available a message why the transform failed.

Parameters

- **target_frame** – The frame into which to transform.
- **target_time** – The time into which to transform.
- **source_frame** – The frame from which to transform.
- **source_time** – The time from which to transform.
- **fixed_frame** – The frame in which to treat the transform as constant in time.

- **timeout** – How long to block before failing in milliseconds. Set to 0 for no timeout.

Returns
True if the transform is possible, otherwise an error message (string) if available, false if not.

QVariantMap **lookUpTransform**(const QString &target_frame, const QString &source_frame, const rclcpp::Time &time = rclcpp::Time(0), double timeout = 0)

Get the transform between two frames by frame id.

Parameters

- **target_frame** – The frame to which the data should be transformed.
- **source_frame** – The frame where the data originated.
- **time** – The time at which the value of the transform is desired. Set to 0 for latest.
- **timeout** – How long to block before failing in milliseconds. Set to 0 for no timeout.

Returns
A map containing a boolean valid field. If valid is true it also contains the transform. If valid is false, it might contain more information, e.g., an exception field with the name of the exception and a message field containing more information about the reason of failure.

QVariantMap **lookUpTransform**(const QString &target_frame, const rclcpp::Time &target_time, const QString &source_frame, const rclcpp::Time &source_time, const QString &fixed_frame, double timeout = 0)

Get the transform between two frames by frame id.

Parameters

- **target_frame** – The frame to which the data should be transformed.
- **target_time** – The time to which the data should be transformed. Set to 0 for latest.
- **source_frame** – The frame where the data originated.
- **source_time** – The time at which the source_frame should be evaluated. Set to 0 for latest.
- **fixed_frame** – The frame in which to assume the transform is constant in time.
- **timeout** – How long to block before failing in milliseconds. Set to 0 for no timeout.

Returns
A map containing a boolean valid field. If valid is true it also contains the transform. If valid is false, it might contain more information, e.g., an exception field with the name of the exception and a message field containing more information about the reason of failure.

void **unregisterWrapper**()

If the count of wrappers gets to zero, the resources of this singleton will be freed.

class **TfTransform** : public QObject

Represents a tf transform between source and target frame.

Properties

QString sourceFrame

The source frame of the tf transform, i.e., the frame where the data originated.

QString targetFrame

The target frame of the tf transform, i.e., the frame to which the data should be transformed.

bool enabled

Whether this tf transform is enabled, i.e., receiving transform updates.

QVariantMap transform

The last received transform as a geometry_msgs/msg/TransformStamped with an added boolean valid field and optional error fields. See *TfTransformListener::lookUpTransform*

QVariantMap message

An alias for transform.

QVariant translation

The translation part of the tf transform as a vector with x, y, z fields. Zero if no valid transform available (yet).

QVariant rotation

The rotation part of the tf transform as a quaternion with w, x, y, z fields. Identity if no valid transform available (yet).

qreal rate

The maximum rate in Hz at which tf updates are processed and emitted as changed signals. Default: 60 Note: The rate can not exceed 1000. Setting to 0 will disable updates.

bool valid

Whether the current transform, i.e., the fields message, translation and rotation are valid.

CHAPTER

TWELVE

TIME

To preserve the accuracy and allow for compatible serialization of message objects, anonymous wrappers for *Time* and *Duration* were introduced. These wrapper types are used for time and duration fields in received messages and the current time can be obtained using the `Ros2.now()` method of the *Ros2 Singleton*.

Example:

```
property var currentTime: Ros2.now()
```

Both wrapper types can be converted to QML/JavaScript `Date` objects using the *toJSDate()* method at the cost of micro- and nanosecond accuracy.

Please note that due to limitations in QML and JavaScript mathematical operations for Time and Duration are not possible.

12.1 API

class **Time**

Represents a point in time for a time source.

Properties:

- seconds: Floating point value containing the seconds passed since epoch. Depending on the size of double, this may have significant precision loss. Use nanoseconds for precision.
- nanoseconds: unsigned integer containing the nanoseconds since epoch.
- clockType: The type of the time source. See Ros2ClockTypes

Public Functions

inline bool **isZero**() const

Whether the time represented by this instance is zero.

inline QDateTime **toJSDate**() const

A JS Date representing the value stored in this instance. Since JS Dates only have millisecond accuracy, information about microseconds and nanoseconds are lost. The time is always rounded down to prevent the JS Date from being in the future.

Properties

double seconds

The time in seconds (since 1970) as a decimal value. (Possible loss in precision)

quint64 nanoseconds

The time in nanoseconds (since 1970) as an unsigned integer.

quint32 clockType

The clock type of this time point. See Ros2ClockTypes.

class **Duration**

Represents a duration for a time source.

Properties:

- seconds: Floating point value containing the seconds passed between two time points. Can be negative. Depending on the size of double, may have significant precision loss. Use nanoseconds if you need a precise measurement.
- nanoseconds: Signed integer containing the nanoseconds that have passed between two timepoints.

Public Functions

inline bool **isZero**() const

Whether the duration represented by this instance is zero.

inline double **toJSDuration**() const

A JS duration representing the value stored in this instance. JS measures differences between two Dates as a (floating point) number of milliseconds.

Properties

double seconds

The time in seconds that has passed between two timepoints. (Possible loss in precision)

qint64 nanoseconds

The duration in nanoseconds.

enum qml_ros2_plugin::ros_clock_types::**Ros2ClockTypes**

Values:

enumerator **Uninitialized**

Uninitialized.

enumerator **Ros**

ROS time.

enumerator **System**

System time.

enumerator **Steady**

Steady clock time.

For instructions on how to setup the QML ROS2 plugin and a quick getting started guide, check the *Quickstart*.

More in-depth examples can be found in the examples folder as described in *Examples*.

INDEX

Q