ROBOTICS TEAM
at Luleå University of Technology

Robotics & AI

LULEÅ UNIVERSITY OF TECHNOLOGY

Graphical User Interface for Robotic Applications

by
Liv Kåreborn

Supervisors
Sina Sharif Mansouri
Christoforos Kanellakis

Developing a GUI

using PyQt5

What is pyqt5?

- PyQt5 is implemented as a set of Python modules. It has over 620 classes and 6000 functions and methods. It is a multiplatform toolkit which runs on all major operating systems, including Unix, Windows, and Mac OS.
- PyQt5's classes are divided into several modules, including the following:
  - **QtCore, QtGui, QtWidgets**, QtMultimedia, QtBluetooth, QtNetwork, QtPositioning, Enginio, QtWebSockets, QtWebKit, QtWebKitWidgets, QtXml, QtSvg, QtSql, QtTest
- The QtCore module contains the core non-GUI functionality. This module is used for working with time, files and directories, various data types, streams, URLs, mime types, threads or processes.
- The QtGui contains classes for windowing system integration, event handling, 2D graphics, basic imaging, fonts and text.
- The QtWidgets module contains classes that provide a set of UI elements to create classic desktop-style user interfaces.

Explanation of some QtWidgets classes

The Qt Widgets Module provides a set of UI elements (classes) to create classic desktop-style user interfaces. Below I have listed all the QtWidgets classes that I have been using.

- QWidget
  - The base class of all user interface objects
- QLabel
  - Text or image display
- QPushButton
  - Command button
- QLineEdit
  - One-line text editor
- QApplication
  - Manages the GUI application's control flow and main settings
- QShortCut
  - Used to create keyboard shortcuts
- QProgressBar
  - Horizontal or vertical progress bar
- QTabWidget
  - Stack of tabbed widgets
- QTableWidget
  - Item-based table view with a default model
- QTableWidgetItem
  - Item for use with the QTableWidget class
- QAbstractItemView
  - The basic functionality for item view classes
- QMessageBox
  - Modal dialog for informing the user or for asking the user a question and receiving an answer
- QFileDialog
  - Dialog that allow users to select files or directories
- QMenu
  - Menu widget for use in menu bars, context menus, and other popup menus

QWidget

The widget is the atom of the user interface: it receives mouse, keyboard and other events from the window system, and paints a representation of itself on the screen. Every widget is rectangular, and they are sorted in a Z-order. A widget is clipped by its parent and by the widgets in front of it.

QApplication

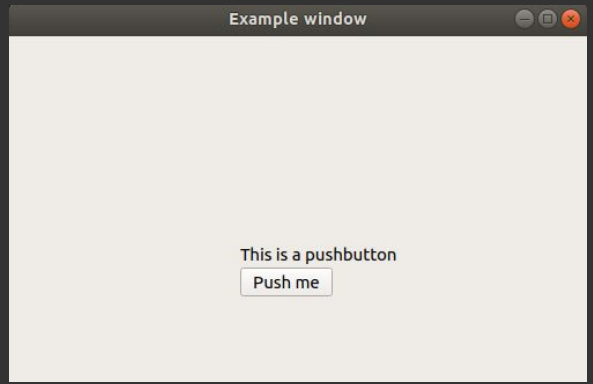
QApplication specializes QGuiApplication with some functionality needed for QWidget-based applications. It handles widget specific initialization, finalization.

Basic GUI- Using QWidget and QApplication

```
class App(QWidget): # creating class with QWidget object
     def __init__(self):
     super(QWidget, self).__init__()
     self.initUI()

     def initUI(self):
     #create mainwindow
     self.setWindowTitle("GUI robotics") # setting window title
     self.setGeometry(500, 200, 400, 500) # setting geometry of
window

if __name__ == '__main__':
     app = QApplication(sys.argv)  # creating QApplication
     ex = App()  # creating instance of QApplication
     sys.exit(app.exec_())   # executing QApplication
```

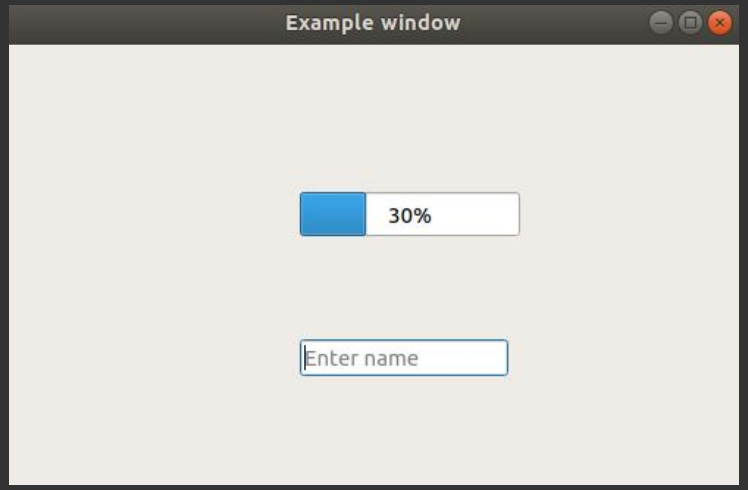
QLabel and QPushButton

```
class App(QWidget): # creating class with QWidget object
      def __init__(self):
      super(QWidget, self).__init__()
      self.initUI()

      def initUI(self):
      #create mainwindow
      self.setWindowTitle("Example window") # setting window title
      self.setGeometry(500, 200, 500, 300) # setting geometry of window
      self.label = QLabel(self)  # creating label
      self.label.setText("This is a pushbutton")
      self.label.move(200, 180)   # moving label
      self.btn = QPushButton('Push me', self) # create pushbutton
      self.btn.move(200,200)
      self.show()
if __name__ == '__main__':
      app = QApplication(sys.argv)  # creating QApplication
      ex = App()  # creating instance of QApplication
      sys.exit(app.exec_())   # executing QApplication
```



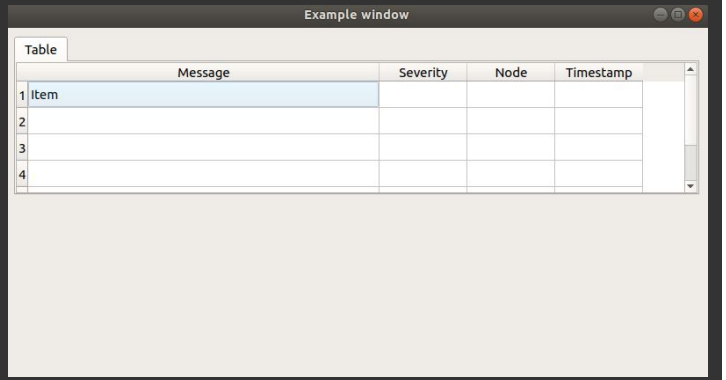
QLineEdit and QProgressBar

```
class App(QWidget): # creating class with QWidget object
        def __init__(self):
        super(QWidget, self).__init__()
        self.initUI()

        def initUI(self):
        #create mainwindow
        self.setWindowTitle("Example window") # setting window title
        self.setGeometry(500, 200, 500, 300) # setting geometry of window
        self.progressBar = QProgressBar(self) # creating progressbar
        self.progressBar.move(200, 100) # moving progressbar
        self.progressBar.resize(150, 30) # rezizing progressbar
        self.progressBar.setMaximum(100) # setting progressbar's maximum
        self.progressBar.setValue(30) # setting value
        self.lineEdit = QLineEdit(self) #creating lineedit
        self.lineEdit.setPlaceholderText("Enter name") # setting palceholder text
        self.lineEdit.move(200,200) # moving lineedit
        self.show() # showing window and widgets
if __name__ == '__main__':
        app = QApplication(sys.argv) # creating QApplication
        ex = App()  # creating instance of QApplication
        sys.exit(app.exec_())   # executing QApplication
```



QTableWidget, QTableWidgetItem, QTabWidget and QAbstractItemView

```
class App(QWidget): # creating class with QWidget object
      def __init__(self):
      super(QWidget, self).__init__()
      self.initUI()
      def initUI(self):
      #create mainwindow
      self.setWindowTitle("Example window") # setting window title
      self.setGeometry(500, 200, 400, 500) # setting geometry of window
      self.table =QTableWidget(self) # creating tablewidget
      self.table.resize(780,180) # rezizing tablewidget
      self.table.setRowCount(6) # setting amout of rows
      self.table.setColumnCount(4) # setting amout of columns
      self.table.setColumnWidth(0, 400) # setting specific column width
      self.table.setItem(0,0, QTableWidgetItem("Item")) #printing "Item" to item (0,0)
      self.table.setHorizontalHeaderLabels(("Message;Severity;Node;Timestamp").split(";")) # setting header labels
      self.table.setSelectionMode(QAbstractItemView.MultiSelection) # enabeling multiselection
      self.tabWidget = QTabWidget(self) # creating tab widget
      self.tabWidget.move(370, 500) # moving tab widget
      self.tabWidget.resize(780,180) # rezizing tab widget
      self.tabWidget.addTab(self.table, "Table") # adding self.table to tab widget
      self.show()
if __name__ == '__main__':
      app = QApplication(sys.argv)     # creating QApplication
      ex = App()  # creating instance of QApplication
      sys.exit(app.exec_())   # executing QApplication
```



QMenu and QFileDialog

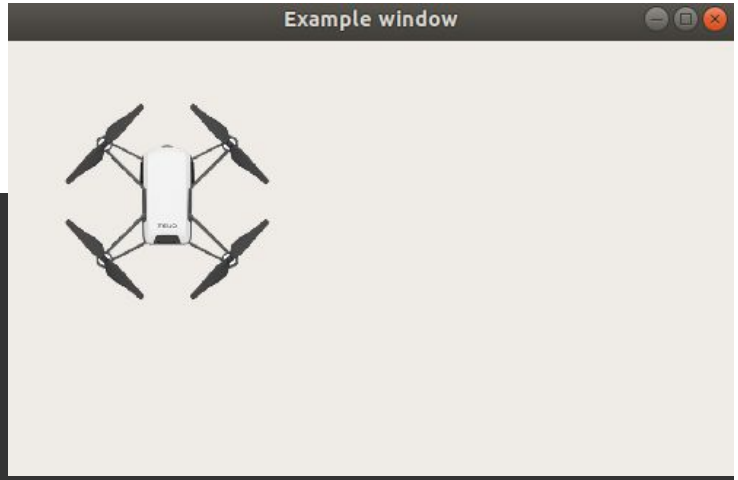
```
class App(QWidget): # creating class with QWidget object
	def __init__(self):
	super(QWidget, self).__init__()
	self.initUI()

	def initUI(self):
	#create mainwindow
	self.setWindowTitle("Example window") # setting window title
	self.setGeometry(500, 200, 500, 300) # setting geometry of window
	self.show()
	def contextMenuEvent(self, event):
	contextMenu = QMenu(self)
	closeAction = contextMenu.addAction("Close")
	fileAction = contextMenu.addAction("File")
	action = contextMenu.exec_(self.mapToGlobal(event.pos()))
	if action == closeAction:
		self.close()
	elif action == fileAction:
		QFileDialog.getOpenFileName(self, "Open file","c:\ ")

if __name__ == '__main__':
	app = QApplication(sys.argv) # creating QApplication
	ex = App()  # creating instance of QApplication
	sys.exit(app.exec_())   # executing QApplication
```

Explanation of some QtGui classes

The Qt GUI module provides classes for windowing system integration, event handling, 2D graphics, basic imaging, fonts and text, etc. Below I have listed all the QtGui classes that I have been using.

- QImage
  - Hardware-independent image representation that allows direct access to the pixel data, and can be used as a paint device
- QPixmap
  - Off-screen image representation that can be used as a paint device
- QKeySequence
  - Encapsulates a key sequence as used by shortcuts
- QColor
  - Colors based on RGB, HSV or CMYK values
- QPainter
  - Performs low-level painting on widgets and other paint devices
- QPen
  - Defines how a QPainter should draw lines and outlines of shapes

QPixmap



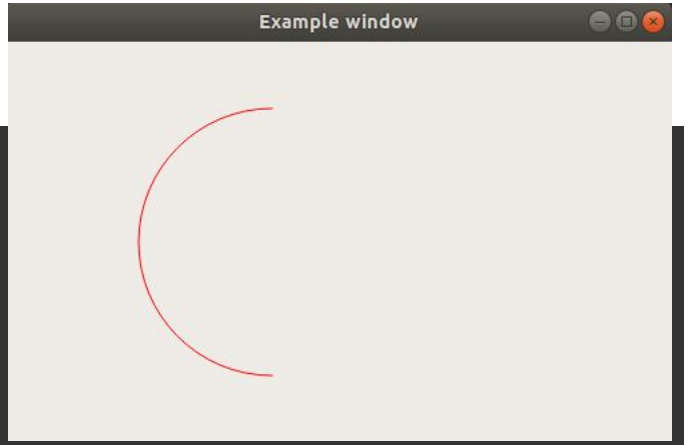

```
class App(QWidget): # creating class with QWidget object
      def __init__(self):
      super(QWidget, self).__init__()
      self.initUI()

      def initUI(self):
      #create mainwindow
      self.setWindowTitle("Example window") # setting window title
      self.setGeometry(500, 200, 500, 300) # setting geometry of window
      self.imageLabel = QLabel(self)
      pixmap = QPixmap("/home/user/.../drone.png")
      pixmap = pixmap.scaled(200, 200, Qt.KeepAspectRatio, Qt.FastTransformation)
      self.imageLabel.setPixmap(pixmap)
      self.imageLabel.move(10, 10)
      self.show()

if __name__ == '__main__':
      app = QApplication(sys.argv)  # creating QApplication
      ex = App()  # creating instance of QApplication
      sys.exit(app.exec_())   # executing QApplication
```

QImage

To add a QImage to a QLabel the QImage first needs to be converted to a QPixmap

```
class App(QWidget): # creating class with QWidget object
      def __init__(self):
      super(QWidget, self).__init__()
      self.initUI()

      def initUI(self):
      #create mainwindow
      self.setWindowTitle("Example window") # setting window title
      self.setGeometry(500, 200, 500, 300) # setting geometry of window
      image = QImage()
      self.imageLabel = QLabel(self)
      self.imageLabel.setPixmap(QPixmap.fromImage(image)) #Setting QImage imageLabel
      self.imageLabel.move(10, 10)
      self.show()

if __name__ == '__main__':
      app = QApplication(sys.argv)  # creating QApplication
      ex = App()  # creating instance of QApplication
      sys.exit(app.exec_())   # executing QApplication
```

QPainter, QPen and QColor




```
class App(QWidget): # creating class with QWidget object
      def __init__(self):
      super(QWidget, self).__init__()
      self.initUI()

      def initUI(self):
      #create mainwindow
      self.setWindowTitle("Example window") # setting window title
      self.setGeometry(500, 200, 500, 300) # setting geometry of window
      self.show()
      def paintEvent(self, e): # event handler
      painter = QPainter(self) #creating painter
      painter.setRenderHint(QPainter.Antialiasing) # smoothes edges
      painter.setPen(QColor(250,0,0)) # creating pen
      painter.drawArc(100, 50, 200, 200, 90 * 16, 180 * 16) # drawing with pen

if __name__ == '__main__':
      app = QApplication(sys.argv)  # creating QApplication
      ex = App()  # creating instance of QApplication
      sys.exit(app.exec_())   # executing QApplication
```

Explanation of some QtCore classes

All other Qt modules rely on the QtCore module. It module contains core non-GUI functionality.Below I have listed all the QtCore classes that I have been using.

- Qt
  - The Qt namespace feature serves as a tool to handle certain scenarios involving multiple configurations of Qt more gracefully.
- pyqtSignal
  - More information regarding signals and slots will follow
- pyqtSlot
  - More information regarding signals and slots will follow

Signals and Slots in PyQt5

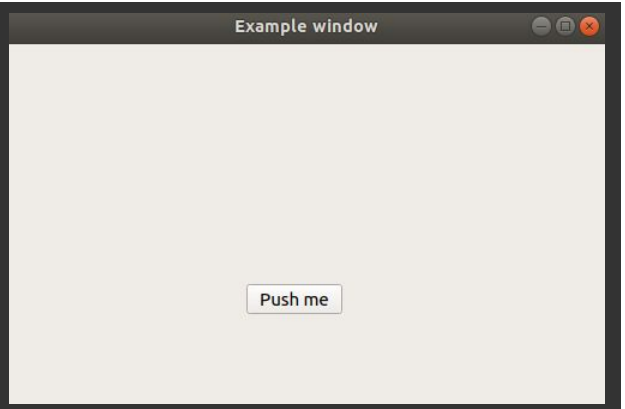
Unlike a console mode application, which is executed in a sequential manner, a GUI based application is event driven. Functions or methods are executed in response to user's actions like clicking on a button, selecting an item from a collection or a mouse click etc., called events.

Widgets used to build the GUI interface act as the source of such events. Each PyQt5 widget, which is derived from QObject class, is designed to emit 'signal' in response to one or more events. The signal on its own does not perform any action. Instead, it is 'connected' to a 'slot'. The slot can be any callable Python function.

In PyQt5, connection between a signal and a slot can be achieved in different ways. In the following slides I will present one way of doing this.

widget.signal.connect(slot_function)

```
class App(QWidget): # creating class with QWidget object
      def __init__(self):
      super(QWidget, self).__init__()
      self.initUI()
      def changeLabelText(self):
      self.label.setText("Pushbutton has been pressed")
      def initUI(self):
      #create mainwindow
      self.setWindowTitle("Example window") # setting window title
      self.setGeometry(500, 200, 500, 300) # setting geometry of window
      self.label = QLabel(self)  # creating label
      self.label.resize(200, 50)
      self.label.move(150, 100)   # moving label
      self.btn = QPushButton('Push me', self) # create pushbutton
      self.btn.move(200,200)
      self.btn.clicked.connect(self.changeLabelText) # sending signal
      self.show()
if __name__ == '__main__':
      app = QApplication(sys.argv)  # creating QApplication
      ex = App()  # creating instance of QApplication
      sys.exit(app.exec_())   # executing QApplication
```



Connecting the GUI to ROS

After creating a GUI you need to connect it to the data you want to display and handle. In my case the data is coming from a drone. By subscribing and publishing data to the drone and sending service calls different functionalities can be enabled and the drone can be controlled and observed via the GUI.

Depending on what type of information you are interested in you will need to subscribe to different topics, and handle the subscribed data in different ways. However, by observing the GUI I have created the way of doing this can be learned and adapted to specific situations.

Subscribing to rostopic

1. Initialize a node in the main
   a. rospy.init_node('node_name', anonymous=True)
2. Subscribe to topic
   a. self.imageSub = rospy.Subscriber("topic_name", MsgType, self.callbackMethod, queue_size = 10)
3. Create a callback method where the subscribed data is sent
   a. def callbackMethod(self, data):
      self.signal.emit(data)
4. Emit data from callback method through a signal/slot connection to a method where it is used.

```
class App(QWidget): # creating class with QWidget object
        signal = pyqtSignal(Imu) # creating signal
        def __init__(self):
        super(QWidget, self).__init__()
        rospy.Subscriber("topic_name", Msg_type, self.imuCallback, queue_size = 10)
        self.initUI()

        @pyqtSlot(Imu) # creating slot
        def changeLabelText(self, data):
        print(data.angular_velocity.x) # extracting data
        self.label.setText(str(data.angular_velocity.x)) # printing data to label

        def imuCallback(self, data): # callbackfunktion for reading imu
        self.signal.emit(data) # emiting data from signal to slot

        def initUI(self):
        #create mainwindow
        self.setWindowTitle("Example window") # setting window title
        self.setGeometry(500, 200, 500, 300) # setting geometry of window
        self.label = QLabel(self)  # creating label
        self.label.resize(200, 50)
        self.label.move(150, 100)   # moving label
        self.show()
        self.signal.connect(self.changeLabelText) #connecting signal to slot

if __name__ == '__main__':
        app = QApplication(sys.argv) # creating QApplication
        rospy.init_node('gui_data', anonymous=True) # initializing node
        ex = App()  # creating instance of QApplication
        sys.exit(app.exec_())   # executing QApplication
```

This example shows how you can subscribe to a topic and continuously update a label in the GUI with it's value.

Since topics generally publish data at a high rate, a callback function that emits data to a slot function is to prefer. This is because signal/slots can queue data in a good way which helps you avoid segmentation fault.

By using signals and slots like I'm doing in this example, you will also be able to send data between classes.

Publishing data to rostopic

Start by making sure you have initialized a node in the main. Important to know that it can be the same node as for subscribing.

```
pub = rospy.Publisher("topic_name", MsgType, queue_size=10)
pub.publish(data) #publishing data to topic_name
```

Calling rosservice

Start by making sure you have initialized a node in the main. Important to know that it can be the same node as for subscribing and publishing.

```
def emergencyLand(self):
    rospy.wait_for_service("service_name", 1)
    emergency_land = rospy.ServiceProxy("service_name", MsgType)
    emergency_land()
```

Guide to connecting two servers to one master

And sending files in between using Paramiko

Connecting two computers to one master

Computer 1 (running the master)

1. Add following lines to **/etc/hosts file**

   127.0.0.1 <tab> localhost
   127.0.1.1 <tab> <hostname for master>
   <ip to master> <tab> <hostname to master>
   <ip to slave> <tab> <hostname to slave>
2. Add following lines to **~/.bashrc file** (in the bottom of the file)

   export ROS_MASTER_URI=http://<hostname or ip of master>:11311
   export ROS_HOSTNAME=<hostname to master>
   export ROS_IP=<ip to master>

Computer 2 (slave)

1. export ROS_MASTER_URI=http://<hostname or ip of master>:11311 (needs to be done in every new terminal)
2. to **/etc/hosts file**
   a. 127.0.0.1 <tab> localhost
      127.0.1.1 <tab> <hostname for slave>
      <ip to slave> <tab> <hostname to slave>
      <ip to master> <tab> <hostname to master>
3. Add following lines to **~/.bashrc file** (in the bottom of the file)
   a. export ROS_HOSTNAME=<hostname to slave>
      export ROS_IP=<ip to slave>

Paramiko- Secure connections to remote machines

The Paramiko module gives an abstraction of the SSHv2 protocol with both the client side and server side functionality. As a client, you can authenticate yourself using a password or key and as a server, you can decide which users are allowed access and the channels you allow.

I have been using Paramiko to create a connection between the computer running the GUI and my platform. Thanks to this connection I have been able to save the recorded rosbag to the platform and send files between the servers. To make paramiko work properly I needed to edit the ~/.bashrc file to enable interactive connection between the servers.

1. open ~/.bashrc
2. uncomment the lines directly under the line

 # if not running interactively, don't do anything

3. add the lines

 # if not running interactively, don't do anything

[-z "$PS1"] && return

4. under the line

source/opt/ros/melodic/setup.bash

Downloading file from remote server

```
      #Methos for downloading file from platform
      def downloadFile(self):
      filename = self.downloadFileLabel.text()
      if filename == "":
            self.messageBox("Enter filename")
      else:
            downloadFilepath = self.downloadFilepath + filename
            downloadLocalpath = self.downloadLocalpath + filename
            ssh = paramiko.SSHClient() # creating representation of session with SSH server
            ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy()) # set policy to use when connecting to
servers without a known host key
            ssh.connect(self.master_ip,self.port,username=self.username, password=self.password) #
connecting to server
            sftp_client = ssh.open_sftp() # open an SFTP session on the SSH server
            sftp_client.get(downloadFilepath,downloadLocalpath) # getting file from server
            sftp_client.close() # close SFTP session
            ssh.close() # closing session
```

Uploading file to remote server

```
#Method for uploading file to remote server
def uploadFile(self):
      try:
            fname = QFileDialog.getOpenFileName(self, "Open file","c:\ ")
            uploadLocalpath = fname[0]
            filename = uploadLocalpath.split("/")[-1]
            uploadFilepath = self.uploadFilepath + str(filename)
            ssh = paramiko.SSHClient() # creating representation of session with SSH server
            ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy()) # set policy to use when connecting to
servers without a known host key
            ssh.connect(self.master_ip,self.port,username=self.username, password=self.password) #
connecting to server
            sftp_client = ssh.open_sftp() # open an SFTP session on the SSH server
            sftp_client.put(uploadLocalpath, uploadFilepath) # put file on server
            sftp_client.close() # close SFTP session
            ssh.close() # closing session
      except:
            pass
```

How to start recording .bag on remote server

```
def record(self):
	ssh=paramiko.SSHClient() # creating representation of session with SSH server
	ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy()) # set policy to use when connecting to servers without a known host key
	ssh.connect(self.master_ip,self.port,username=self.username, password=self.password) # connecting to server
	stdin,stdout,stderr=ssh.exec_command("rosbag record -a") #record all topics, save input output and error
	ssh.close() # closing session
```

How to stop recording .bag on remote server

```
def stopRecord(self):
	ssh=paramiko.SSHClient() # creating representation of session with SSH server
	ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy()) # set policy to use when connecting to servers without a known host key
	ssh.connect(self.master_ip,self.port,username=self.username, password=self.password) # connecting to server
	stdin,stdout,stderr=ssh.exec_command("rosnode list") # exec command and saving input, output and error
	list_output = stdout.read() # read output
	for string in list_output.split("\n"):
	if (string.startswith("/record")):
		killing = ssh.exec_command("rosnode kill " + string) # stop recording
	ssh.close() # closing session
```

Thank you

Contact: Liv Kåreborn