University of Puerto Rico-Mayaguez

Department of Electrical and Computer Engineering

ROS-Plastron

Israel J. Lopez Toledo
Mariam C. Saffar Pérez
Moises Garip
ICOM 4036
December 11, 2018

Introduction

As robotics extends further into our daily lives, it becomes important that those with little to no experience be able to take part in this growing market. The Robotic Operating System (ROS) is an open-source, meta-operating system for robots. It provides the services expected from an operating system, including hardware abstraction, low level device control, implementation of commonly used functionality, message passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers. ROS is the most popular open source robotics platform available, held up by a large community that continuously works on improving it. Although the ROS platform eases software development in robotics by providing libraries and tools to developers, there is still a big learning curve that developers must overcome in order to start working with ROS. There is also a big amount of repetitive code involved in ROS, for example every ROS node has the same code structure.

Our solution to the previously stated problems is ROS-Plastron, a programming language designed to simplify the use of ROS. The main purpose of ROS-Plastron is to make ROS more approachable, making it accessible to a wider user base. ROS-Plastron will ease node creation based on user's specifications and it will ease the manipulation of existing nodes. Nodes are processes within ROS, that utilize the platform's client library to communicate with other nodes. This communication can be through Subscription, which is based on the indirect message passing communication model, or through Services, which is based on the Client-Server communication model. The data transferred between communicating nodes are called messages, they are a simple data structure, comprising typed fields. In general ROS-Plastron will help inexperienced users easily overcome ROS's learning curve, while experienced developers will be able to take advantage of ROS-Plastron's features to quicken their project development.

Language Tutorial

ROS_Plastron currently supports two features: Node Creation and Existing Node Manipulation. The Node Creation feature allows users to generate four types of node templates: Subscribers, Publishers, Clients and Servers. The Existing Node Manipulation feature allows users to load existing node files into the system, perform changes to them (i.e unsubscribing a node from a topic and subscribing it to another topic) and then generate the node files. In the future ROS_Plastron will support other features like: Custom Message Creation and URDF Files Creation.

To utilize ROS_Plastron the user should first make sure he has Python 3.4 and the PLY package installed. Then he needs to download the ROS_Plastron zip file from github and extract its content. Finally, he needs to execute ROS_Plastron/src/main.py in order to run the language. The following example is a script that generates a client node.

```
create_node 'client' as node1
node1 client_requests 'add_two_ints' of service_type AddTwoInts with input 5,6
node1 client_requests 'waypoint_clear' of service_type WaypointClear with input none
node1 client_requests 'set_bool' of service_type SetBool with input True
generate_node node3
```

The first line of the example code creates a node of name “client”, the system will reference this node as “node1”. Lines two to four state that the node will request those specified services of the specified service types and it will provide the specified inputs as the request’s parameters.
The next example is a short script that loads and manipulates an existing node.

```
load subscriber_sub1 as node2
unsubscribe node2 from 'chatter'
subscribe node2 to 'topic_added_after_load' of message_type String
generate_node node2
```

The first line of the example loads the existing node file “subscriber_sub1”, the system will reference this node as “node2”. The next line unsubscribes node2 from its currently subscribed topic “chatter”. The next line subscribes node2 to a new topic called “topic_added_after_load”, which is of type “String”. And the last line generates the modified node template.

Reference Manual

The following are ROS_Plastron grammar rules:

- Create a node in the system:

  create_node 'NODE_NAME' as VARIABLE
- Subscribe a node to a topic:

  subscribe VARIABLE to 'TOPIC_NAME' of message_type MESSAGE_TYPE
- Unsubscribe a node from a topic:

  unsubscribe VARIABLE from 'TOPIC_NAME'
- Publish to a topic:

  VARIABLE publish MESSAGE to 'TOPIC_NAME' of message_type MESSAGE_TYPE
- Stop publishing to a topic:

  VARIABLE stop_publishing to 'TOPIC_NAME'
- Node requests a service:

  VARIABLE client_requests 'SERVICE_NAME' of service_type SERVICE_TYPE with input PARAMETERS
- Stop requesting a service:

  VARIABLE stop_request 'SERVICE_NAME'
- Node advertises a service:

  VARIABLE provides_service 'SERVICE_NAME' of service_type SERVICE_TYPE
- Stop providing a service:

  VARIABLE stop_service 'SERVICE_NAME'
- Create message:

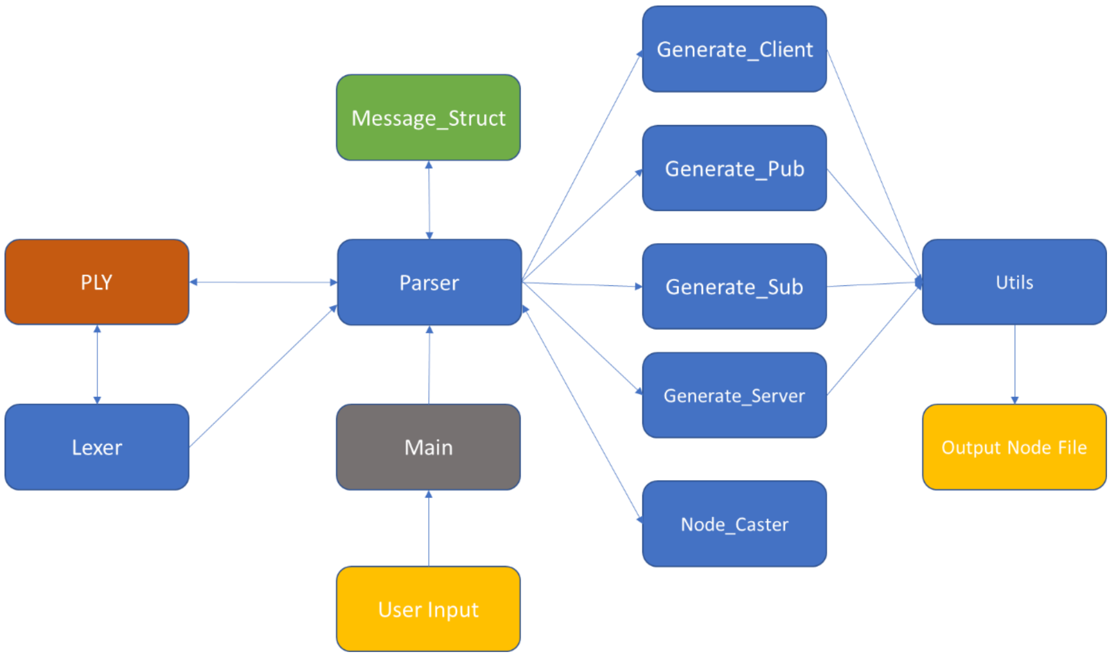
  create_message MESSAGE of message_type MESSAGE_TYPE and input USER_INPUT
- Generate template:

  generate_node VARIABLE
- Load existing node:

  load FILE_NAME as VARIABLE

Language Development

Translator Architecture



Module Interface

The main module of ROS_Plastron, *main.py*, is the module that takes care of capturing the user input. The module passes the user input to the parser, *plastron_parser.py*, who processes the input with the help of the lexer, *plastron_lex.py*, and the PLY package. After processing the input, the parser then stores the appropriate user input, if the user created a message then it creates a message object utilizing *message_struct.py*. If the user is attempting to generate a node, the parser calls one of the generate node modules (*generate_client.py*, *generate_server.py*, *generate_pub.py*, *generate_sub.py*) and sends it the appropriate data. The generate node module processes the data sent by the parser and once the template is ready it utilizes the module *utils.py* to create the node file. If the user is attempting to load a node into the system, the module *node_caster.py* parses the node file and loads the appropriate data into the system.

Development Environment

The following tools were used to develop ROS_Plastron:

- Python 3.4 : Language utilized to develop the project
- PLY: Package lex and yacc parsing tools
- PyCharm: IDE utilized to develop the project
- Github: Platform hosting the source code.

Test Methodology

The testing approach that was utilized throughout this project was Incremental Integration Testing, this type of testing is a bottom-up approach for continuous testing. During this project each module was tested individually to assure each functionality worked properly. After testing was done on a module, it was attached to the main module and then the main module was tested to assure that the new functionality did not affect other functionalities or the performance of the whole project.

To test the lexer a series of invalid tokens and characters were used as input to observe if they were handled properly. The parser was tested by utilizing a series of invalid grammar rules, this was to see if the parser handled these errors properly. The generation modules were tested first by sending them fixed values and observing their behavior. After it was made sure that the modules were working properly, they were attached to the project and tested with input from users. This input from the user included invalid cases, like mixing a subscriber node with a publisher node or large amounts of operations, for example a publisher that published ten different messages. The node caster module was first tested by providing it directly with node files that had small amounts of operations. Once it was working properly, it was added to the main code and it was provided larger node files to observe its behavior and the main module's behavior, since it needed to process the data parsed by this module.

Test Programs

- **Example 1: Creating a Publisher Node**

```
create_node 'publisher' as node1
create_message msg1 of message_type Point and input 5,6,7
create_message msg2 of message_type Vector3 and input 1,3,0
create_message msg3 of message_type String and input "hello_World"
create_message msg4 of message_type Quaternion and input 8,9,10,11
create_message msg5 of message_type Pose2D and input 1,2,80
node1 publish msg1 to 'chatter' of message_type Point
node1 publish msg2 to 'chatter1' of message_type Vector3
node1 publish msg3 to 'chatter2' of message_type String
node1 publish msg4 to 'chatter3' of message_type Quaternion
node1 publish msg5 to 'chatter4' of message_type Pose2D
generate_node node1
```

- **Example 2: Creating a Subscriber Node**

```
create_node 'subscriber' as node2
subscribe node2 to 'chatter' of message_type String
generate_node node2
```

- **Example 3: Creating a Client Node**

```
create_node 'client' as node3
node3 client_requests 'add_two_ints' of service_type AddTwoInts with input 5,6
node3 client_requests 'waypoint_clear' of service_type WaypointClear with input none
node3 client_requests 'set_bool' of service_type SetBool with input True
generate_node node3
```

- **Example 4: Creating a Server Node**

```
create_node 'server' as node4
node4 provides_service 'add_two_ints' of service_type AddTwoInts
node4 provides_service 'waypoint_clear' of service_type WaypointClear
node4 provides_service 'set_bool' of service_type SetBool
generate_node node4
```

- **Example 5: Loading and modifying a node**

  load subscriber_sub1 as node5
  unsubscribe node5 from 'chatter'
  subscribe node5 to 'topic_added_after_load' of message_type String
  generate_node node5

Conclusion

The result of this work was a functional language in its early stages, that can simplify the creation and manipulation of ROS nodes. The two proposed main features were successfully implemented and are fully operational. ROS_Plastron's grammar was designed to be as close as possible to English sentences. This resulted in an intuitive grammar for users who had no prior experience in programming. The down side of this design was that the language became more verbose than common programming languages and it also resulted being harder to develop for the programmers behind the project. The biggest challenge faced during this project was implementing the load node feature. This is because it is complicated to parse node files that are developed by users, since they don't necessarily follow a standard format. This problem was handled for the moment by only allowing to parse nodes that have been created by ROS_Plastron. By limiting the system to only nodes that it has created before, the system has an easier time finding the information it needs from the node files. Another reason this feature was implemented this way was that if the user is completely inexperienced in programming, it is safe to assume that he will create most of his nodes utilizing ROS_Plastron and therefore the nodes loaded into the system will be most likely ROS_Plastron created nodes. This project was focused specifically in nodes, but ROS_Plastron has potential to make other aspects of ROS easier for users. Some of the future features that the team is interested in developing for ROS_Plastron are: custom message creation, URDF file creation, support for node creation and manipulation for C++ node files and support for node creation and manipulation for nodelettes. Some of these features were presented in the proposal but they fell out of the scope of the project, since the focus was on the creation and manipulation of python node files as previously mentioned. In conclusion ROS_Plastron has potential to become a powerful tool in the field of robotics software development.