RCON: Dynamic Mobile Interfaces for Command and Control of ROS-enabled Robots

Robert Codd-Downey and Michael Jenkin
Center for Field Robotics, Department of Electrical Engineering and Computer Science, York University, 4700 Keele Street, M3J 1P3, Toronto, Ontario, Canada

Keywords: ROS, Human-robot Interaction, Teleoperation, iOS.

Abstract: The development of effective user interfaces for an autonomous system can be quite difficult, especially for devices that are to be operated in the field where access to standard computer platforms may be difficult or impossible. One approach in this type of environment is to utilize tablet or phone devices, which when coupled with an appropriate tool such as ROSBridge can be used to connect with standard robot middleware. This has proven to be a successful approach for devices with mature user interface requirements but may require significant software development for experimental systems. Here we describe RCON, a software tool that allows user interfaces on iOS devices to be configured on the device itself, in real time, in response to changes in the robot software infrastructure or the needs of the operator. The system is described in detail along with the accompanying communication framework and the process of building a user interface for a simple autonomous device.

1 INTRODUCTION

True autonomy does not usually exist in today's commercial and academic robots. Typically these systems are designed to take operational directives from a human operator and communicate results back to the operator in a meaningful way. This human robot interaction problem is typically approached using a fixed point control station equipped with monitors and haptic controllers (e.g., Joysticks, Mice). While these control stations offer many benefits they lack the mobility necessary for high situational awareness in extended range applications. A mobile operator equipped with a portable device such as a laptop or netbook offers this situational awareness on hardware compatible with the target robot but commercial versions of such devices are not hardened for operations outdoors. When considering the inclement weather so often encountered with field robotics the tradeoff between cumbersome equipment and environmental protection becomes apparent. Commercial off the shelf mobile electronics such as iOS and Android devices present an inexpensive alternative to custom environment and weather-hardened equipment. Protecting these devices from the elements is inexpensive and has a minimal affect on their portability. That being said, these devices do not generally support the traditional range of robot software development and operational tools offered on laptop/netbook platforms.

The use of mobile devices for human robot interaction has been actively explored in past. PocketCERO (Httenrauch and Norman, 2001), one of the first forays into this area of research, was limited by a lack of standardization across robotic platforms and computational resources available on mobile devices at the time. Implementations on more modern Android devices such as, (Wu and Chen, 2004) have demonstrated human robot interaction using a specific hardware platform.

The acceptance of ROS (Quigley et al., 2009) as a standard middleware for autonomous devices, at least in the research domain, has simplified the development of robot-user interaction substantively. Perhaps the simplest method developing a human-ROS-based robot interaction system on a tablet device would be to deploy a full fledged ROS installation on the target device. This would provide software developed on the device access to the full ROS infrastructure. Unfortunately, ROS's primary development and deployment environment is unix-based and this introduces some difficulties when deploying ROS in non-unix environments. ROSJava (2015) is a pure Java implementation of ROS developed for just such a purpose on the Android platform. On iOS there have

Codd-Downey R. and Jenkin M..
RCON: Dynamic Mobile Interfaces for Command and Control of ROS-enabled Robots.
DOI: 10.5220/0005540000660073
In *Proceedings of the 12th International Conference on Informatics in Control, Automation and Robotics* (ICINCO-2015), pages 66-73
ISBN: 978-989-758-123-6

been similar attempts such as ROSPod (2015) which is still unstable and not a full-featured implementation. Even with a full underlying ROS infrastructure to facilitate development on such devices there are a number of other considerations. Android and iOS devices are limited by available memory capacity, not to mention processing power, and the overhead of an underlying ROS implementation cuts into available resources. Another consideration is the unreliability of wireless communication. ROS assumes a high bandwidth LAN connection between nodes and the master. A user interface that augments the on-robot ROS environment with user interface-based nodes on the robot may have a substantive impact on robot performance if communication becomes unstable. With this in-mind a more prudent approach is to connect to the ROS environment through other means. The ROSBridge (Crick et al., 2011) library enables interaction with a ROS environment from external processes. This interaction is facilitated through commands in the form of JSON (Java Script Object Notation) strings (Crockford, 2006) . These commands are mapped to internal ROS functions and enable external processes to act as a ROS node operating within the normal ROS environment.

This paper is organized into four major sections; the first reviews earlier efforts to link external user interaction systems to ROS through ROSBridge. The second describes the implementation in Objective-C of a ROSBridge client framework that facilitates communication with ROS. The third describes the implementation and functions of RCON which acts as an interface builder on iOS devices. Finally, to illustrate the effectiveness of the approach a sample UI is constructed and tested using an autonomous system.

2 PREVIOUS WORK

This paper outlines two important advancements from those described in previous works. The first is an iOS implementation of an ROSBridge client library. The second is the development of a flexible robot-interface creation tool that leverages this iOS library to enable the construction of such user interfaces on the device itself. This flexibility would not be possible without the capabilities offered by objective-c objects that enables new class definitions to be created on-the-fly.

In this paper we demonstrate how to leverage the ROSBridge library to allow for the development of a native interface on iOS devices to facilitate effective human-robot interaction. This strategy has been used effectively in the past by Speers et al. (2013) to create a ROSBridge client library for Android and in Codd-Downey et al. (2014) for the Unity3D software platform. These libraries can be used to develop customized interfaces that interact with a target robot. Such predefined interfaces are very effective on production robotic systems in which user-interface requirements can be well specified in advance. However a majority of academic projects are experimental where the development of a customized interface can be considered premature. This may also be true for prototype robots in a commercial setting. Under these circumstances maintaining a user interface that keeps in stride with changes in the robot's architecture can be costly and time consuming. This motivates the need for a dynamic interface that can be reconfigured on the fly based on the needs of the current operator. Such an interface could be used for other purposes that may not have been considered cost effective.

3 ROSBRIDGE iOS FRAMEWORK

ROSBridge (Crick et al., 2011) facilitates communication between an external application and the ROS environment. It exposes messages from the ROS environment to external software using JSON strings, and allows an external agent to inject messages into the ROS environment. In essence it allows an external application to respond to and influence the ROS environment without the overhead of a full ROS implementation. The ROSBridge iOS framework leverages this communication protocol to provide an iOS application access to the ROS environment using native Objective-C constructs. Translating commands to and from JSON strings and facilitating communication with the server is essential to providing a simple and comprehensible framework to developers. An overview of this communication pathway is depicted in Fig. 1, which also lists the full set of ROSBridge commands supported by the iOS framework.

The motivation for choosing iOS as our development platform in part is due to the support of native application development within iOS. This is to be contrasted with the Android platform where applications are run within a virtual machine which adds additional memory and processing overhead. iOS applications are written in Objective-C which is essentially an object oriented framework built overtop C using a preprocessor and an extensive runtime environment. Leveraging this runtime environment, an iOS application is afforded many of the capabilities available only to the compiler in many other languages. These capabilities include but are not limited to; constructing and registering new class definitions, attach-

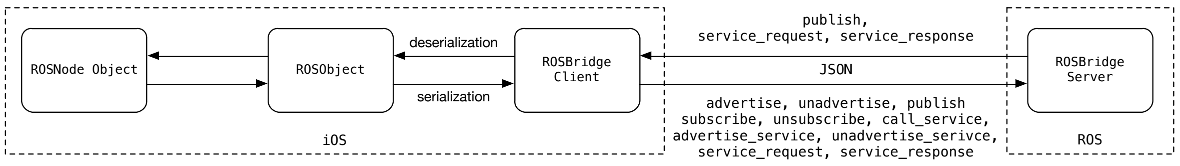

ROSNode Object
ROSObject
deserialization
serialization
ROSBridge Client
publish, service_request, service_response
JSON
advertise, unadvertise, publish subscribe, unsubscribe, call_service, advertise_service, unadvertise_serivce, service_request, service_response
ROSBridge Server
iOS
ROS

Figure 1: Overview of the client communication pathway: The ROSBridge class handles all communication from the client side. All incoming messages are serialized into a instance of a subclass of ROSObject and passed to the corresponding ROSNode implementation. Outgoing messages and commands are serialized from an instance of a subclass of ROSObject and compiled into a JSON string and sent to the ROSBridge server.

ing new methods and variables to a class and registering additional protocol definitions. These capabilities make an excellent fit for the requirements of a dynamic user-interface framework.

3.1 Client Architecture

The client framework has a layout similar to that of ROS (see Fig. 2). All processes that communicate with the ROS environment via the client framework implement the ROSNode protocol. This facilitates bi-directional communication between a ROSNode object and the ROSBridge client framework through pair objects; a ROSObject, and a ROSInfo object. A ROSObject carries information that has been deserialized from either a message, response or request from ROS. The accompanying ROSInfo object contains both a name and a type and is used to identify the specific topic or service which sent the data contained within the ROSObject object.

ROSNode objects are registered with the main ROSBridge class whenever they initiate a command that necessitates a response. Each ROSNode is identified using a globally unique string (similar to that of a process and thread identifier within ROS) which aids message routing. ROSNodes have the ability to use all functions within ROS' publisher/subscriber framework and request/response framework. The main ROSBridge class also provides easy access to the ROSAPI – the set of service calls provided by the ROSBridge server – through a set of simple methods and callbacks.

3.2 Mapping ROS Messages to Classes

Key to developing a user interface that interacts with ROS is the structure of the relationship between ROS messages and the software infrastructure within the user interface itself. Incoming messages could be dealt with using two different approaches: the first approach converts JSON strings to their generic object counterparts (Dictionary, Array, String, Number), the second approach converts JSON strings to the instance of a predefined class representing the message. The benefit of the first approach is that message types do not have to been known beforehand. Using predefined classes carries the benefit of using literal syntax and avoiding type casting. Mapping ROS messages to predefined classes is complicated because a JSON object does not carry type information. Speers et al. (2013) solved this problem by requiring all message classes to be registered with the system, the corresponding class is that which matches the layout of the JSON Object. In their system new messages cannot be added at runtime. The flexibility of the Objective-C runtime allows for both approaches to be taken simultaneously. Relying on a standard naming scheme for message classes (e.g. /turtle1/cmd_vel becomes _turtle1_cmd_vel) an instance of the class can be created at runtime assuming a corresponding definition has been registered with the runtime environment by the application executable. If the class is not registered a new class definition can be generated on the fly and registered with the runtime environment.

Traditionally Objective-C development required that a developer actively manage memory within the language's reference counting framework. This meant acquiring/releasing ownership of an object by manually calling the objects retain/release methods to increment and decrement the corresponding instance reference count. In recent years there has been a push in iOS software towards Automatic Reference Counting (ARC) where the compiler adds these methods automatically. Including ARC in a project allows a developer to leave memory management to the compiler, which results in more time actively developing and less memory related bugs. A major caveat in the use of ARC is that the compiler forbids the use of a dealloc, retain and release method. This is especially problematic when constructing new class definitions at runtime, as not implementing a dealloc method for new classes results in catastrophic memory leakage. Fortunately ARC can be disabled on select source code files, however, best practices would dictate that files compiled without ARC support be kept to an absolute minimum. In our case all interaction with the runtime environment is contained within the ROSObject class.

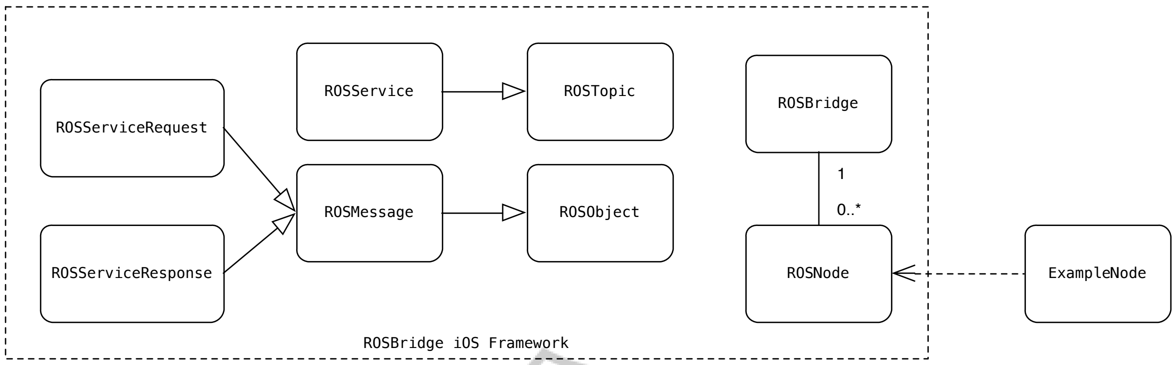


Figure 2: Overview of the client framework: The ROSBridge class handles all client side communication to and from the server. All incoming messages are deserialized into ROSMessage objects and passed to the corresponding ROSNode implementation. Outgoing messages and commands are serialized into JSON strings and sent to the ROSBridge server.

4 RCON

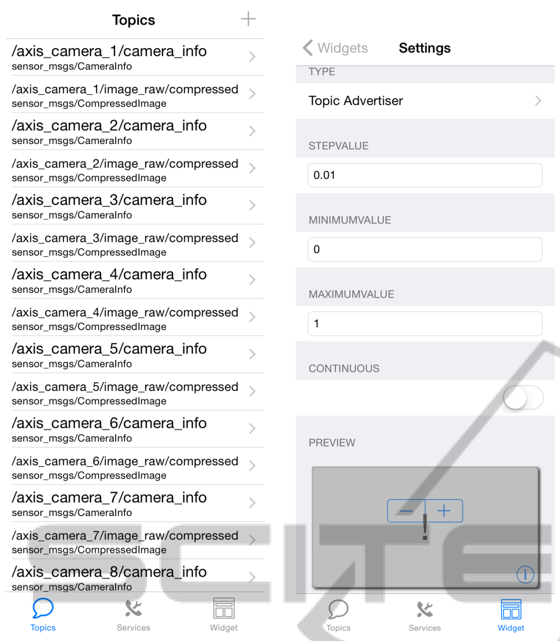
RCON is an iOS application that is inspired by the ROS tool rViz and aims to provide not only visualization features, but command functions as well. RCON provides a similar look and feel to rViz and enables a user to construct and use a custom set of interface elements (widgets) on-the-fly. These widgets have access to the full underlying structure of ROS messages and can display information from a subscribed topic, create data to publish on a topic, call an advertised service or generate a response to a service.

RCON treats a set of user configured widgets as a canvas. These canvases can be created, destroyed and renamed. Changing from the active canvas to another canvas automatically stops the operation of the current set of widgets and starts the operation of the widgets within the new canvas. Canvases afford the user with the ability to create different interfaces for different robots and tasks. A canvas be configured by adding, removing, repositioning and resizing its widgets. The views shown in Fig. 3 depict the stages of adding widgets to the currently active canvas. The final stage of the widget addition process involves configuring the per-widget settings that affect its behaviour. For instance, a Stepper Widget requires a minimum value, maximum value, step value and a indicator of how often to send updates.

4.1 Available Widgets

Interaction with the ROS environment is facilitated solely through the use of the predefined widgets. Each widget has its own set of customizable attributes that define its behaviour. Additionally, each widget is only capable of interacting with certain message types or services. The widgets available with RCON include:

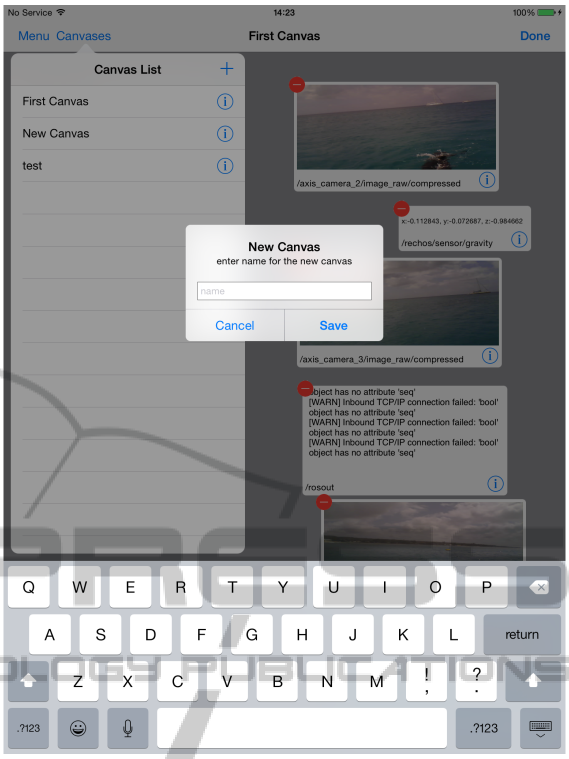
- **Button** - A button widget is used to call services and can be configured to call separate services upon being pressed and released.
- **Label** - A label widget displays raw data from a large number of message and primitive types. (Topic Subscriber only).
- **Text Field** - A text field widget displays raw data from a large number of messages and primitive types, it can also be used to publish data to a topic, if text is properly formatted.
- **Switch** - A switch widget can publish or subscribe to a std_msg/Bool message or bool primitive types.
- **Slider** - A slider widget can publish or subscribe to all standard number messages or types.
- **Stepper** - A stepper widget is capable of publishing to all standard numerical messages and primitive types.
- **Image** - An image widget displays data from sensor_msgs/Image, sensor_msgs/CompressedImage and is also capable of interpreting and generating images from other messages (e.g., sensor_msgs/LaserScan and nav_msgs/OccupanyGrid messages).
- **Text View** - A text view widget displays a large scrollable text field that remembers previous messages. Useful as either a publisher/subscribe of string type messages (including ros_graph/Log messages).
- **Map** - A map widget displays the world map showing visual GPS coordinate information and can be used to both subscribe to and publish on topics of type sensor_msgs/NavSatFix.


Topics
/axis_camera_1/camera_info
sensor_msgs/CameraInfo
/axis_camera_1/image_raw/compressed
sensor_msgs/CompressedImage
/axis_camera_2/camera_info
sensor_msgs/CameraInfo
/axis_camera_2/image_raw/compressed
sensor_msgs/CompressedImage
/axis_camera_3/camera_info
sensor_msgs/CameraInfo
/axis_camera_3/image_raw/compressed
sensor_msgs/CompressedImage
/axis_camera_4/camera_info
sensor_msgs/CameraInfo
/axis_camera_4/image_raw/compressed
sensor_msgs/CompressedImage
/axis_camera_5/camera_info
sensor_msgs/CameraInfo
/axis_camera_5/image_raw/compressed
sensor_msgs/CompressedImage
/axis_camera_6/camera_info
sensor_msgs/CameraInfo
/axis_camera_6/image_raw/compressed
sensor_msgs/CompressedImage
/axis_camera_7/camera_info
sensor_msgs/CameraInfo
/axis_camera_7/image_raw/compressed
sensor_msgs/CompressedImage
/axis_camera_8/camera_info
sensor_msgs/CameraInfo

Widgets
Settings
TYPE
Topic Advertiser
STEPVALUE
0.01
MINIMUMVALUE
0
MAXIMUMVALUE
1
CONTINUOUS
PREVIEW

Figure 3: Adding a widget to the current canvas can be initiated through one of three pathways, selecting a target topic, service or widget. After choosing a topic or service a user is then displayed a list of compatible widgets. After selecting a widget it must then be configured using the widgets settings panel. This settings panel displays a list of configurable variables that can be used to modify the widgets behaviour. Here the selected topic or service can be further altered. When a widget has been properly configured it can be dragged to the desired location on the canvas.

- **Accelerometer** - An accelerometer widget displays data from an accelerometer and publishes this to a sensor_msgs/Imu topic.
- **Gyroscope** - A gyroscope widget displays data from a gyroscope and publishes this to a sensor_msgs/Imu topic.
- **Magnetometer** - A magnetometer widget displays data from a magnetometer and publishes this to a sensor_msgs/Imu topic.
- **Camera** - A camera widget displays images from the onboard camera and can publish this image to a sensor_msgs/CompressedImage topic.
- **Multi-Touch** - A multi-touch widget displays multitouch interactions from the user and publishes planar information using a variety of types within geometry_msgs.
- **Compound** - A compound widget can be configured by grouping together other widgets to form new messages, requests or responses. Using this widget message or service types not previous known to the application can be created.


No Service
14:23
100%
Menu Canvases
First Canvas
Done
Canvas List
First Canvas
New Canvas
test
New Canvas
enter name for the new canvas
name
Cancel
Save
/axis_camera_2/image_raw/compressed
x:-0.112843, y:-0.072687, z:-0.984662
/rechos/sensor/gravity
/axis_camera_3/image_raw/compressed

```
object has no attribute 'seq'
[WARN] Inbound TCP/IP connection failed: 'bool'
object has no attribute 'seq'
[WARN] Inbound TCP/IP connection failed: 'bool'
object has no attribute 'seq'
[WARN] Inbound TCP/IP connection failed: 'bool'
object has no attribute 'seq'
```

/rosout

Figure 4: An interface (canvas) within RCON can be modified by adding/deleting widget or resizing (via a pinch gesture) or repositioning (via a touch and drag). Canvases can also be created, renamed and are hot swappable.

These widgets encompass a large percentage of the standard user interface elements available within iOS. Their capabilities as widgets stem from their configurable attributes and visual interface within iOS. For instance, the Stepper widget is not able to function as a subscriber because its view (a plus and a minus button) does not include feedback as to its current value.

4.2 Use Cases

RCON makes it possible to create any type of user interface one so desires. However, there is a particular set of tasks for which RCON is expressly capable, as described below.

Sensor Monitoring. Monitoring all sensors onboard a robot can be a daunting task. This is especially true for large complex robots such as the PR2 and Atlas robots, where monitoring all of the sensors at once is effectively impossible. In order to manage this problem it is common to group sensors either by location or function. Thus a different group is monitored depending on the current task. However, unexpected behaviour on the part of the robot may require that data from another sensor be accessed to assess the problem. In this case a new widget can be added and others moved around or resized to make room for the additional view.

Using the camera widget an user can also publish image streams of the robot while in operation. This enables a user to simultaneously record the actions of the robot while closely monitoring its sensors, assuming the necessary topics are already being recorded by the rosbag record utility. Normally this test requires an additional person with a separate device capable of recording video or taking pictures.

Motion Controller. One shortcoming of laptop and netbook based human-robot interaction is their inability to capture inertial data. Although motion controllers such as the PlayStation Sixaxis, Dualshock, etc. can provide this functionality, they add additional complexity to the process of actually controlling a remote device. Such sensors are typically integrated within tablet/phone devices and can be easily adopted to control a robot as demonstrated by Speers and Jenkin (2013) . Using a compound widget containing an accelerometer, gyroscope and magnetometer widgets, within RCON a complete sensor_msgs/Imu message can be compiled and sent to the robot where this interaction can be carried out. One should be aware how this model differs slightly from the approach taken in Speers and Jenkin (2013) . Lacking a user scriptable engine RCON can't process and interpret new results. As a result calculations that turn raw sensor data, into user commands must take place on board the robot.

Discovery. Most human-robot interaction within an academic setting takes place between a robot and an experienced user. Within such a setting the user is often intimately familiar with the robot and its capabilities. A novice user typically does not have this acquired knowledge. RCON provides a novice user with an intuitive tool to quickly discover a robot's features and capabilities. Browsing through the list of topics reveals onboard sensors. Browsing through the services reveals the robot's command structure.

Collaborative Interfaces. Multiple operators can be connected to a robot simultaneously using RCON on separate devices. Operators interact with the robot but also with each other as well. For example, subscribing to the image topic published by another operator's camera enables one operator to see what the other sees. A chat room of sorts could be set up using the TextView widget for viewing messages and TextView/TextField widget for sending messages. This facilitates cooperation between operators that are interacting with the robot under different capacities (e.g., controlling and monitoring) or those who are operating separate sensor/mobility systems, or event

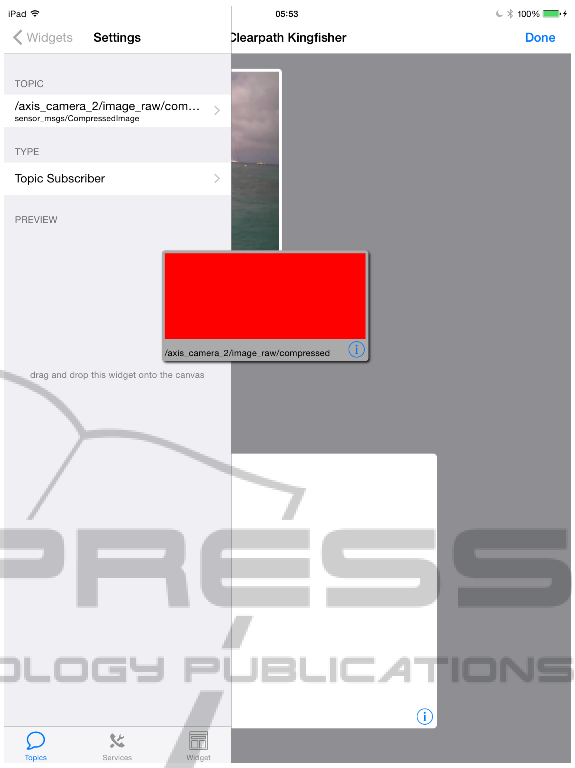

iPad
05:53
100%
Widgets
Settings
Clearpath Kingfisher
Done
TOPIC
/axis_camera_2/image_raw/com...
sensor_msgs/CompressedImage
TYPE
Topic Subscriber
PREVIEW
/axis_camera_2/image_raw/compressed
drag and drop this widget onto the canvas
Topics
Services
Widget

Figure 5: After a widget has been fully and properly configured the warning overlay on the preview disappears. This signifies that the widget is ready to be added to a canvas. Adding a widget to the current canvas can be done by dragging and dropping the preview to the desired location on the canvas.

separate multiple robots under a single ROS master process.

5 EXAMPLE UI CONSTRUCTION

Constructing a new interface for controlling a robot using RCON is quite simple especially if one is familiar with the target platform. In this example we will demonstrate the interface construction process for a Clearpath^{TM}Kingfisher (Fig. 6) that has been outfitted with additional sensors (camera array and depth sensor (Codd-Downey et al., 2014b) with integrated accelerometer). The first step in this process is to connect to the rosbridge server running on the robot. When a connection has succeeded RCON will display the list of topics and services that are being advertised by the device. The first interface element we will create is a multi-touch widget. This widget will be used to send velocity commands to the robot in the form of a geometry_msgs/Twist message. To add this widget to the current canvas select the touch widget from the widget list panel, this

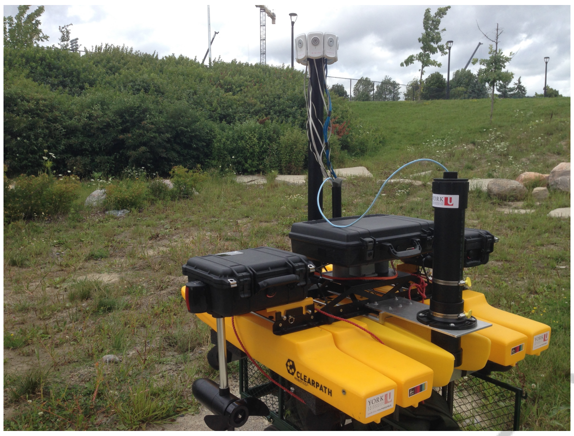


Figure 6: Overview of the Kingfisher Robotics Platform. The unmodified platform comes equipped with an IMU, a GPS antenna, a radio transceiver and differential drive capability. Additional sensing capabilities have been added to the platform, these include a panoramic array of camera and a SONAR depth sensor.

will bring up a settings panel where the widget can be configured. The touch widget has two basic settings (type and topic), in this case the type should be set to publisher and the topic set to /cmd_vel, when the widget has been properly configured the preview shown on the settings panel can be dragged on to the canvas as depicted in Fig. 5. Next we add two image widgets to the canvas so that live camera data from the robot can be viewed on screen. To do this you select the topic which corresponds to the desired camera feed (sensor_msgs/CompressedImage or sensor_msgs/Image) from the topics panel. This will display a prompt asking whether you wish to subscribe to the topic or publish a message to it. After selecting subscribe, a list of widgets that can act as subscribers to the selected topic is displayed, of these widgets we are interested in the image widget. Adding widgets via this process sets the type and topic settings so the preview can be immediately added to the canvas via a drag and drop gesture. This process can be repeated to add a second image widget to the canvas as illustrated in Fig. 7.

When the canvas has been populated with the desired widgets the canvas can be further configured by repositioning and resizing widgets to achieve the desired layout. The interface constructed in this example is shown in Fig. 7. The touch widget that was added to the canvas can be used to send motion commands to the robot using a pan gesture to illustrate the desired motion. This simple interface and the process involved in its construction demonstrates how RCON can be used to create effective interfaces for interacting with robots.

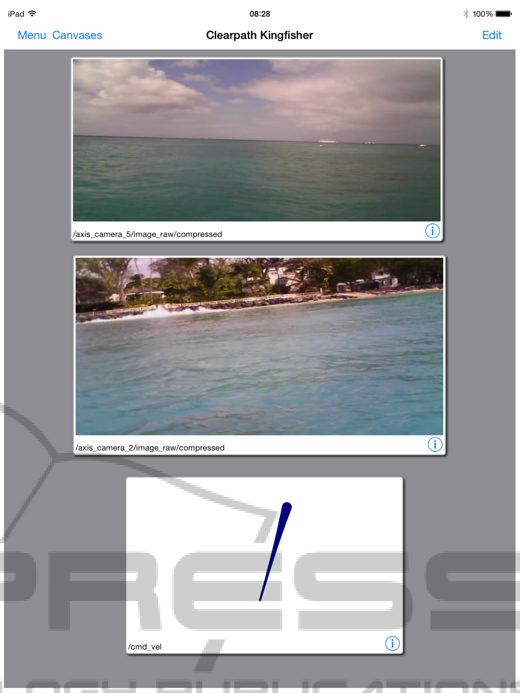

iPad
08:28
100%
Menu Canvases
Clearpath Kingfisher
Edit
/axis_camera_5/image_raw/compressed
/axis_camera_2/image_raw/compressed
/cmd_vel

Figure 7: This example interface consists of two image widgets and a touch widget. The image widgets display live camera data published by the robot. The touch widget can be used to send velocity messages to the robot using touch gestures.

6 DISCUSSION

RCON offers a wide variety of applications in the field of human-robot interaction. The most obvious and notable is the ability to create interfaces that can control and monitor a robot's activity on-the-fly. This ability can eliminate the need to devote costly resources into the development of custom user interface applications. Multiple operators can choose to customize an interface specific to their needs, whether that be the size/placement of certain widgets due to handedness/preference or the inclusion of additional widgets for increased awareness.

Robots still in the development stage frequently require debugging within the field. This process ordinarily requires feedback from low-level systems and accessing this information can be difficult. With RCON specialized one-off debugging an interface can be created easily. If a custom look-and-feel is required later in the production stage the ROSBridge client library allows such development to progress without worrying about the underlying communication between ROS. For instance, much of the screen real-estate used by RCON could be used for other purposes, and custom views could facilitate greater information exchange. This is of particular interest to those who wish to develop simultaneously on iOS and Android, as the AppPortable toolkit can translate from Objective-C to Java, albeit with minor modification using compiler directives. Native Android development does not provide this functionality and the reverse process is not currently available.

At the moment, there is a caveat to proliferating this technology. ROSBridge does not provide any form of security to inhibit access to the core ROS environment. This could result in a nefarious third party gaining unwanted access to the robot. The most effective method to prevent against this is to password protect the wireless network in use by the robot. Additionally the port on which the ROSBridge server is running can be changed from the default port and kept secret.

ACKNOWLEDGEMENT

The support of NSERC Canada and the NSERC Canadian Field Robotics Network (NCFRN) is gratefully acknowledged.

REFERENCES

Codd-Downey, R., Forooshani, P., Speers, A., Wang, H., and Jenkin, M. (2014a). From ROS to unity: Leveraging robot and virtual environment middleware for immersive teleoperation. In *IEEE International Conference on Information and Automation (ICIA 2014)*, pages 932–936.

Codd-Downey, R., Jenkin, M., and Speers, A. (2014b). Building a ros node for a nmea depth and temperature sensor. In *Proc. 11th Conf. on Informatics and Control, Automation and Robotics (ICINCO)*, Vienna, Austria.

Crick, C., Jay, G., Osentoski, S., Pitzer, B., and Jenkins, O. C. (2011). Rosbridge: ROS for non-ROS users. In *Proceedings of the 15th International Symposium on Robotics Research*.

Crockford, D. (2006). Rfc4627: Javascript object notation. In *Fiedler, M., Möller, S., & Reichl, P.(2012). Quality of Experience: From User Perception to Instrumental Metrics (Dagstuhl Seminar 12181). Dagstuhl Reports*, volume 2, pages 1–25.

Httenrauch, H. and Norman, M. (2001). Pocketcero - mobile interfaces for service robots. In *In Proceedings of the Mobile HCI, International Workshop on Human Computer Interaction with Mobile Devices*.

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y. (2009). ROS: an open-source robot operating system. *ICRA workshop on open source software*.

ROSJava (2015). ROSJava - ros wiki. http://wiki.ros.org/rosjava. Accessed: 2015-02-25.

ROSPod (2015). ROSPod - ros wiki. http://wiki.ros.org/rospod. Accessed: 2015-02-25.

Speers, A., Forooshani, P. M., Dicke, M., and Jenkin, M. (2013). Lightweight tablet devices for command and control of ros-enabled robots. In *16th International Conference on Advanced Robotics (ICAR 2013)*, pages 1–6. IEEE.

Speers, A. and Jenkin, M. (2013). Diver-based control of a tethered unmanned underwater vehicle. In *Proceedings of the 10th International Conference on Informatics in Control, Automation and Robotics (ICINCO 2013)*, pages 200–206.

Wu, S. and Chen, Y. (2004). Remote robot control using intelligent hand-held devices. In *The Fourth International Conference on Computer and Information Technology, 2004. CIT'04.*, pages 587–592. IEEE.