# Lightweight tablet devices for command and control of ROS-enabled robots

Andrew Speers, Parisa Mojiri Forooshani, Michael Dicke and Michael Jenkin
Computer Science and Engineering, York University, Toronto, Canada

(speers,pmojirif,jenkin)@cse.yorku.ca, michael.dicke@smail.inf.h-brs.de

***Abstract*—Effective interaction with autonomous devices is a complex problem as the devices themselves are mobile. For the vast majority of devices operating over an extended range a fixed operator or oversight position is inappropriate and mobile oversight becomes essential. Although walking in the vicinity of an autonomous robot with a networked laptop may be appropriate for some environments, such an approach becomes increasingly cumbersome as the robot moves out into the field where weather and other operational concerns are factors. Here we examine the use of lightweight tablet computers as a generic interface to autonomous systems. We demonstrate how ROS-enabled computers can be controlled using Android tablets using standard software toolkits. Tools have been developed for the automatic conversion between ROS messages and the corresponding user interface elements on the tablet and a collection of robot-centric user interface elements have been developed to assist in developing systems for different robot sensors and platforms. Systems are demonstrated for unmanned underwater, surface, flying and rolling ground contact vehicles.**

## I. Introduction

There are very few autonomous systems that are designed to be fully autonomous. Rather, it is intended that such systems operate in response to user instruction and that they will communicate their results and status to a human operator or companion. Given the experimental nature of many such systems it is typically the case that such interaction is via a networked workstation or similar standard computing device. This approach tends to work well for indoor robot systems operating in close proximity to their operator. The operator has access to the underlying software infrastructure and the full set of development and experiment management tools as these are often best supported on this type of platform. As the system moves out of the lab this operational modality becomes less and less desirable. The interaction platform moves first to laptops, then netbooks, ruggedized laptops and the like as more and more portable hardware systems are required. The nature of the problem becomes most readily apparent as experiments take place in inclement weather under environmental conditions that make operation of even ruggedized laptops difficult.

One attractive alternative to the use of commercial off the shelf laptop/workstation hardware is to utilize portable consumer electronics such as IOS, Android, and similar devices. These devices are inexpensive and are hardened to use in the outdoor environment, although they are typically not waterproof. That being said, such devices do not generally support the normal operating system of the robot platform, nor are robot software development/analysis tools well developed for such platforms. For example, they do not generally provide full support for typical robot middlewares (e.g., ROS [1]).

Given their low cost, lightweight nature, and suitability for deployment outdoors there have been a number of efforts to leverage lightweight devices as control interfaces for robotic systems. Early efforts (e.g. [2]) were limited by the lack of standard software infrastructure on the robot platform and software libraries on the lightweight platform. More recent efforts (e.g. [3]) have looked at specific applications and hardware platforms. There have also been efforts (e.g. [4] and [5]) to explore the potential benefits/restrictions of hand-held devices for autonomous systems relative to more traditional interfaces.

Recent advances in robot middleware standardization coupled with standard software toolkits enable the construction of human-robot interaction systems that operate on lightweight tablet and phone platforms. There are a number of potential approaches to this task. The first would be to develop a fully operational ROS environment on the lightweight device and then develop an interaction framework that now has complete access to the underlying ROS infrastructure. This is certainly an attractive approach and there exist efforts, most notably based on rosjava [6], a pure-Java implementation of ROS. ROS' primary target is based on C++ and the Linux environment, and thus such an approach relies on a ROS implementation that lies outside of the core ROS target. An extreme version of the complexities associated with such a task is seen in the rospod project [7], an effort to port ROS to IOS-based devices. Rather than attempting to fully support ROS on a novel hardware platform and OS. Another approach would be to utilize features of HTML5 to map interaction possibilities on the device to a remote ROS environment. This was a central goal of the implementation of the rosbridge [8] package. Rosbridge provides a mechanism within which messages generated within the ROS environment are exposed to an external agent and within which an external agent can inject messages into the ROS environment. This is also a viable approach, although interaction opportunities are limited to the abilities of the HTML5-based browser. A third approach is to exploit the rosbridge mechanism, but rather than utilizing a HTML5-enabled browser, utilize instead a native application running on the lightweight interface device. This is the approach taken here. This allows for more sophisticated user interaction mechanisms – taking advantage of the full capabilities of the lightweight device – while still exploiting the standard rosbridge protocol. By leveraging specific properties of the ROS infrastructure in this paper we demonstrate how Android devices can be used to provide an effective portable robotic interface without the requirement of having a full ROS environment running on the Android platform itself. We have constructed an infrastructure that enables automatic conversion between the representations used by ROS to corresponding user interface elements on the Android platform. A number of test implementations are described that show the capability of the approach to develop interfaces for a range of different hardware platforms.

## II. From ROS messages to a lightweight interaction device

ROS has emerged as a standard software middleware for the development of autonomous systems. Within ROS, overall robot control is modelled as a collection of asynchronous processes that communicate via message passing. Although ROS has been ported to a number of different hardware platforms and bindings exist for a number of different languages, support is primarily targeted towards Ubuntu, with software libraries targeted at C++ and Python. A very limited level of support exists for lower performance devices (e.g., Android platforms) but the limited memory footprint on such devices makes development and deployment difficult. Rather than relying on the less-well-supported ROS environment available on such devices, here we utilize the rosbridge [8] mechanism to enable an external agent to communicate with the ROS environment. This process utilizes the WebSocket [9] protocol as a communication layer which means that provided the external agent has network access to the ROS environment it can be physically located anywhere. Standard libraries exist that support the WebSocket protocol on the Android device (e.g., [10]).

The rosbridge framework communicates ROS messages to and from the external world in the form of YAML (YAML Ain't Markup Language) [11] strings. Such YAML strings can be used directly by an external agent but perhaps the most convenient way is to use JSON [12] to map YAML strings to instances of objects in the environment of the external agent. This approach has a number of advantages in terms of interfacing a lightweight device such as an Android tablet to a ROS-enabled device. First, this approach avoids the requirement of actually running ROS on the lightweight device. This is a critical issue as ROS is not designed for the small memory, bandwidth and computational limits of computer tablets. Second, interfacing with the robot's control environment can be separated to a large extent from the ROS-based computation that is providing vehicle control. Finally, the approach only exposes the specific messages that are key for device operation/management.

The rosbridge YAML-based protocol must deal with the limitations of lightweight devices. First, the communications channels available on the device are reasonably limited in bandwidth. Thus it may not be practical for a device to sample specific ROS message streams at their full bandwidth. Second, certain data streams become particularly large when expressed in YAML. For example, image streams and other binary data sources are encoded as base64 strings in YAML. This allows YAML representations to be represented as seven bit clean ASCII strings. It does, however, make the strings somewhat larger than the original binary source. The process of parsing these strings typically results in another buffer the size of the input string. Given the relatively small memory footprint associated with lightweight devices (16M-64M is typical), these large strings can be problematic. This basic issue is brought into sharp focus when ROS messages attempt to pass extremely large messages (many 100's of megabytes in size). We follow three strategies to addresses these issues

- High bandwidth message streams are throttled within the robot infrastructure itself prior to encoding the stream with rosbridge. This streaming is accomplished so that the throttling introduces a minimum latency in message transmission. This has the advantage of reducing calls on the network bandwidth and on the Android's processor.
- Large data structures within the ROS environment are repackaged so that data that is not required on the tablet is not communicated to it.
- We avoid sending large-format binary data (e.g., image streams) through the rosbridge–YAML–JSON channel. Rather, a web server is instantiated on the robot to transmit these data forms on request in binary format.

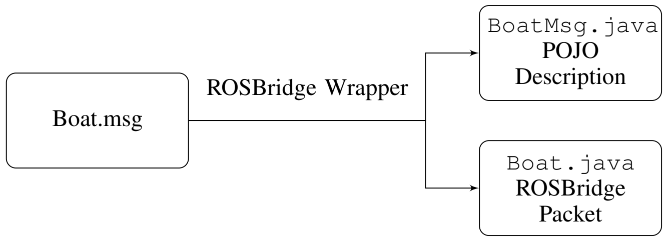

Fig. 1. Automated message POJO generation. ROSBridgeWrapper takes a standard ROS message file and outputs a set of POJOs (Plain Old Java Objects) representing the message. Here the Boat.msg custom message file is passed through the ROSBridgeWrapper application generating two files Boat.java and BoatMsg.java. BoatMsg.java is a POJO representing the fields specific to the custom Boat message. Boat.java represents the additional fields associated with the rosbridge wrapper as well as providing mechanisms for generating the appropriate strings for serializing common tasks directed from the tablet to the robot (subscribing, advertising and publishing). These POJOs are used by a JSON processor to aid in the deserialization of information transmitted from rosbridge over the websocket connection.

### A. *Mapping ROS messages to the tablet infrastructure*

While the utilization of technologies such as ROS, rosbridge, websockets, JSON and YAML provide a consistent and typically convenient communication pipeline for these portable interfaces, a good deal of work is still required to build a framework suitable for rapid prototyping of such interfaces. At the core of this communication pipeline is the publisher / subscriber, message-centric architecture employed by ROS. As such, the key step in interacting with ROS involves mapping ROS messages to internal structures in the intended application. To accomplish this we have developed a ROSBridgeWrapper application. ROSBridgeWrapper takes a standard ROS message file and creates the corresponding POJOs (Plain Old Java Objects) necessary to encapsulate the message definition as well as basic functionality needed by the tablet framework (see Figure 1). POJOs are generated for all messages of interest to the developer in their particular application and are then usable within the framework (shown in Figure 2) to automatically generate the necessary communication messages as well as assisting in the serialization and deserialization needed to go between the data stream and usable Java objects.

Building a tablet interface involves defining the set of messages that the device is interested in monitoring / displaying as well as the set of messages that the device is interested in transmitting back to the robot, and dealing with services and error conditions that may be relayed through the communication linkage. Figure 3(a) shows this process. For each message type that will be communicated a POJO wrapper is constructed automatically as described above, and how these messages are to be treated must be described to the system.

### B. *Mapping ROS messages to UI elements*

Associated with each POJO wrapper is a callback method. When a given message is received from the ROS environment it is deserialized, an instance of the message is constructed, and its callback method invoked. This provides the necessary interface between a received message and the corresponding user interface element. Figure 3(b) shows the code required to perform this mapping. A large library of standard user interface widgets have been constructed to simplify the design of the tablet display. For example, the message being processed by the code given in Figure 3(b) deals with a SICK laser scanner message, and a standard widget exists for displaying such information.

## III. Sample implementations

Here we demonstrate the efficacy of the approach using a range of different robot platforms. As each of these platforms utilize ROS nodes as their underlying software infrastructure the process of exposing state information and injecting messages into the message stream is very similar across all of the devices. The process of listening to specific message streams involves developing a mapping between the YAML encoding of the ROS message and the corresponding Java object structure. Once accomplished, every message instance within the ROS message stream results in a POJO instance within the lightweight device. Appropriate callbacks are used to transfer this information to the user interface. Commands from the user interface are mapped to the appropriate YAML string and injected into the ROS message framework.

The user interface on the Android platform is based around a number of reused user interface elements. Input is either through the normal touch actions defined on the platform, through the physical orientation of the device relative to gravity/north as sensed by the device itself, or through switches monitored by the Android device.

### A. *Wheeled mobile robot*

A Nomad SuperScout was used to test the controller on a traditional differential drive platform. The SuperScouts (Figure 4(a)) represent one of the more popular platforms used for lab-based robotics research and were prolific in the 1990's when they were first released. The basic sensing modality of the SuperScout platform is in the form of a constellation of 16 ultrasonic sensors located around the perimeter of the robot as well as a matching set of contact bumpers. The original controlling hardware provided by the manufacturer has been upgraded (see [13]) and the robots operate a real-time reactive control architecture that provides environmental safety while responding to general motion commands from external agents [14].

The SuperScout software architecture has been augmented with a set of ROS nodes that encapsulate the capabilities of the robot. Within the ROS network being run are nodes that provide access to all the sensor information as well as the ability to poll and set motion goals ($(x, y, \theta)$) in a global coordinate frame. The robot runs a reactive/subsumption architecture which attempts to keep the robot away from obstacles on way to it's target goal.

The Android interface provides a simple means of setting the goal position of the robot (Figure 4(f)). This is done via four buttons which increment and decrement the x- and y-coordinates of the goal in 0.25 meter increments. The current position of the robot maintained through the robots odometry is also provided to the user on the interface in order to provide some context between the actual vs. desired pose of the robot.

### B. *Aquatic surface craft*

The minnow robot (Figure 4(b)) [15] is an autonomous surface craft with two controllable degrees of freedom (DOF) (rudder and throttle). The device is built around a commercially available RC motorboat. Standard electronics components are employed to interface with the RC boat electronics, and the vessel is augmented with GPS, a video camera, and a tilt compensated compass to provide the necessary onboard sensing capabilities. On-board computation is provided via a Pandaboard running Ubuntu. A ROS-based control and sensing infrastructure is employed to operate the vehicle onboard while wireless Ethernet provides communication offboard. Normally the minnow robots are operated in a fleet within which each individual robot acts as its own ROS master (roscore). A heartbeat strategy is employed to detect communication failures between elements of the fleet.

When commanded from another computer with its own ROS master, a python-based graphical user interface is used to control the vehicle. This interface provides the onboard camera view, GPS coordinates, and the speed and heading of the boat. It also provides accessibility to control the throttle and heading of the vessel using button and text widgets. The Android interface shown in Figure 4(g) provides a similar interface. A video feed from the robot is provided in the upper portion of the display. The centre portion shows from left to right the current heading, the roll-pitch of the vessel and it's commanded heading.

### C. *Unmanned underwater vehicle*

Operation of underwater vehicles is complicated by the fact that the vessel is indeed, underwater. This complicates everything from communication among the crew interacting with the device – it is difficult to shout out warnings to other members of the team – to actually observing the device under operation. KROY (see Figure 4(c)), a version of the AQUA2 robotic platform [16]. is a swimming hexapod robot. The device is self contained for power, sensing and computation and utilizes its six fins for mobility. Underwater, communication to the outside

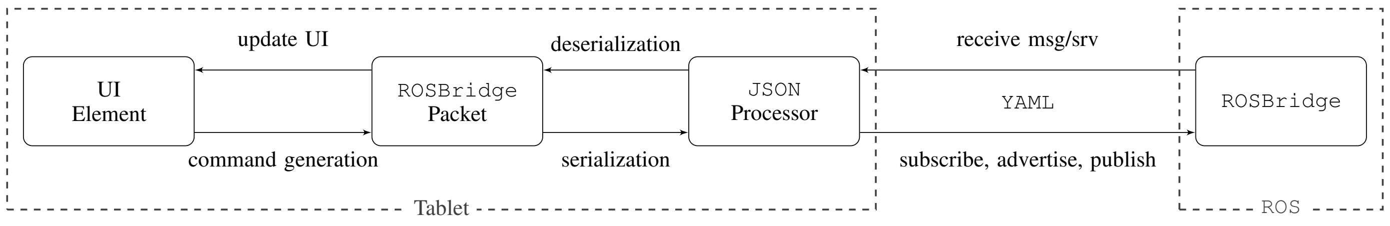

Fig. 2. System diagram showing the communication pathways between the ROS environment and the tablet-based UI elements. When updating the UI from messages generated from the robot side the message is sent over the websocket interface via YAML, this string is processed by a JSON processor and is deserialized into a ROSBridge packet (which is auto-generated from the ROS message file) which updates the UI through a callback which is automatically triggered upon message deserialization. In the opposite direction, changes in the UI can update the POJO of the message which will generate a serialized version of the message that can be transmitted over the websocket and parsed by ROSBridge for insertion into the ROS infrastructure. It should also be noted that the necessary transmissions needed for subscribing and advertising various messages are also triggered from the ROSBridge packet level, passing through the JSON processor and transmitted over the websocket interface to the ROSBridge node.

```
public class RosbridgeCommanderHandler extends ROSBridgeWebSocketHandler {
  static Class[] publishers = {CommanderScan.class, CommanderOdometry.class, CommanderControlMode
      .class};
  static Class[] subscribers = {CommanderTwist.class};

  public RosbridgeCommanderHandler(Activity activity) {
    super(activity, publishers, subscribers,
                  ROSBridgeServiceResponsePacket.class, ROSBridgeStatusPacket.class);
  }
}
```

(a) Defining message interactions

```
public class CommanderScan extends LaserScan {
  public static final String messageName = "base_scan";
  public static final String messageType = "sensor_msgs/LaserScan";

  public void callback(Activity a) {
    ncfrn.yorku.graphics.LaserScan scan = (ncfrn.yorku.graphics.LaserScan) (a.findViewById(R.id.
        commanderLaserScan));
    scan.setScan(this.getMsg());
  }
}
```

(b) Associating a user interface element

Fig. 3. The code that is required to interact with a ROS-enabled robot. In (a) the set of messages that the user interface is interested to receive and communicate with the robot are defined as well as mechanisms to deal with service responses and error conditions. In (b) a link is defined between a message received from the robot and its corresponding user interface element.

world is limited to its onboard cameras, signals through a small number of visual ports, its actions, and an optical fibre tether. One effective mechanism for communicating with the device is through a waterproof tablet connected to this tether [17]. In this mode divers operating underwater control the device while in close proximity of the robot. Earlier versions of this underwater tablet utilized a full-sized PC tablet encased in a waterproof housing. Although this was an effective strategy the large displacement of the resulting housing complicated operations [18]. The Android-based version of this approach is much more compact. The primary user display is shown in Figure 4(d). The display is broken down into three columns. The left most column shows instantaneous leg angles and leg temperatures providing a snapshot of the current state of the low level systems on the vehicle. Below this the current heading and pose of the vehicle are displayed via a compass and an artificial horizon. The center panel shows a live video feed from the robot (in Figure 4(d) the live video feed is substituted by a static image) followed by the current commanded speed of the vehicle and the orientation of tablet housing relative to gravity (indicated by the red circle in the center of the display). The right panel shows a set of potential inputs. Given the switch-based input paradigm available underwater it is not possible for the operator to just "select" an input by pressing on it. Rather the operator cycles through the various active buttons using one of the switches while one of the positions of the second switch causes the current button to be executed.

### *D. ARDrone*

A wide range of commercial (see [19] for a review) and home-grown quadrotor designs exist (e.g., [20]). Here we describe an Android-based control system for the the Parrot ARDrone. The Parrot ARDrone platform (Figure 4(d)) is a low budget and lightweight UAV distributed as a high-tech

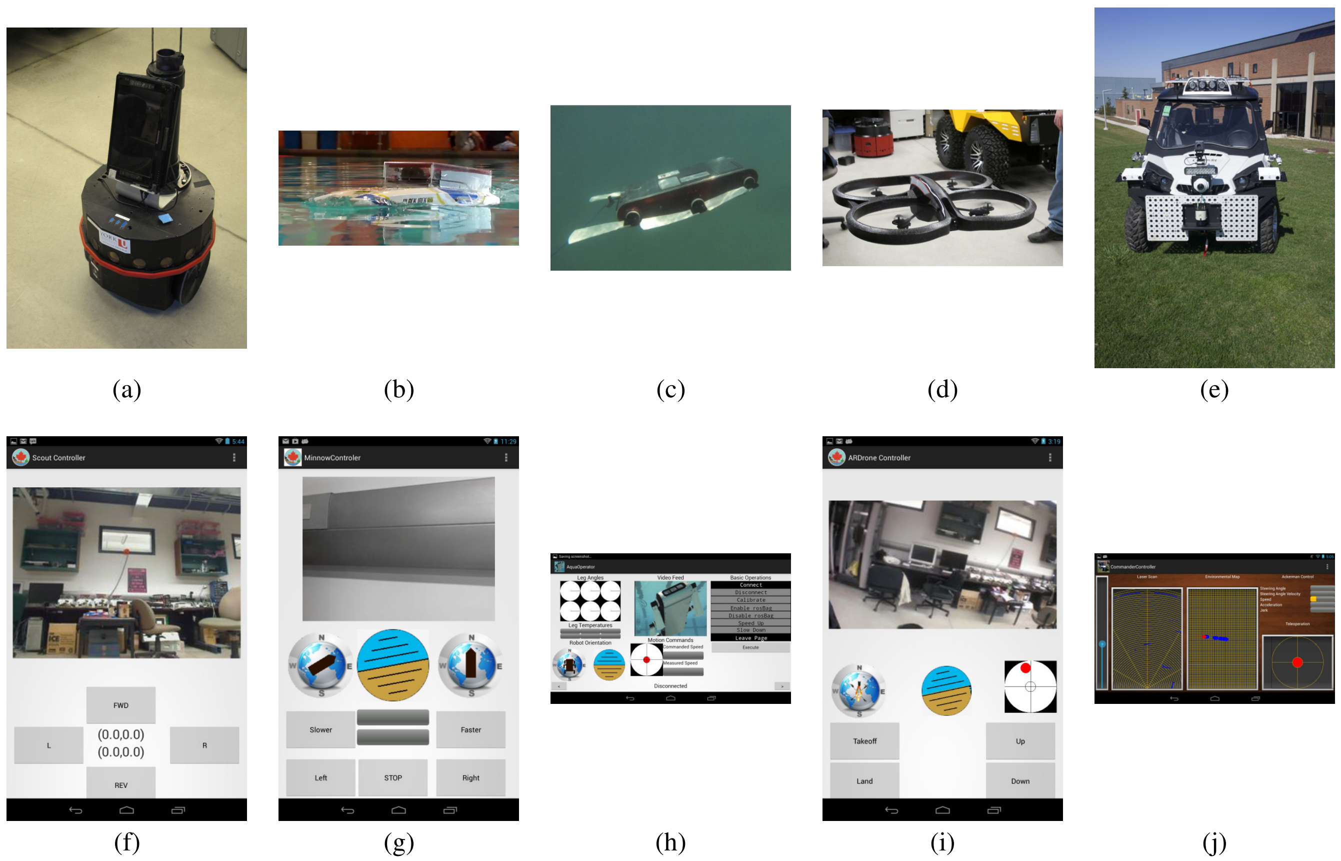

Fig. 4. Example systems. Several robots (a)-(e) running ROS at their core have had interfaces developed for monitoring and control purposes in order to validate the core aspects of the tablet framework. Wheeled, air, surface and underwater vehicles have been tested with varying degrees of control. See text for descriptions of the vehicles and their corresponding tablet interfaces.

toy. For research purposes it is possible to interface the device to ROS through a number of different packages. We choose to use the ardrone_autonomy package which offers abstracted control features and access to onboard cameras and IMU data. Figure 4(i) shows the lightweight user interface developed to control the device. As with the underwater robot example, here we use the onboard orientation sensor of the Android as the primary input mechanism. Sensors from the ARDrone are reflected in the user interface as well as the live video streamed from the device.

### *E. Commander*

The Commander vehicle (Figure 4(e)) is a joint effort between MacDonald, Dettwiler and Associates (MDA) and the Canadian Space Agency (CSA). A Bombardier vehicle has been retrofitted to be computer controlled with various vision-based and laser sensors mounted on the vehicle for teleoperation and autonomous purposes. The corresponding interface Figure 4(j) displays information for the forward-facing laser scanner, odometry information and Ackermann drive statistics from the vehicle and allows for control using an on-screen joystick and a linear slider selecting the maximum velocity of the vehicle.

## IV. Summary and Future Work

The development of a common robot middleware coupled with a standard transport infrastructure enables a wide range of devices to be used for robot control. Here we have shown how rosbridge and JSON can be leveraged along with standard Android user interface elements to build a package for the development of user interface elements for autonomous robots. These lightweight controllers can be easily hardened for difficult environments (e.g., underwater), have reasonably long operational cycles and are light and portable. Furthermore, they typically support a range of different networking options including cellular phone signals, at a very competitive cost.

These lightweight devices support a large range of interaction possibilities. A set of simple interaction elements (e.g., compasses, artificial horizon, etc.) can be easily re-used for different robots and different applications on the same robot. The devices themselves offer a wide range of input mechanisms, from multi-touch surfaces to IMU-based sensors. They can also be interfaced to standard joysticks and the like. For very hostile environments (e.g., underwater) it is possible to augment Android's and similar devices with external switches which makes them useful as user interface elements even in the harshest environments.

The use of such devices is not without its limits. The devices typically have complex application lifecycles, and have limited memory, communication bandwidth and computational resources. The limited computational resources are of a particular concern. The message-based model of ROS coupled with the high bandwidth and potentially large volume of data available on the vehicle can make it impractical to present all of this information to the lightweight device. We have found that it is possible to deal with these restrictions by (i) throttling packets that are to be sent to the tablet, (ii) downsampling data on the robot prior to presenting the data to the tablet, and (iii) by presenting large-scale binary data (e.g., image streams) through a parallel data transport mechanism.

Ongoing work is examining the use of this framework for the creation of custom interfaces for specific field trials / tasks. For example, the interface being used while attempting to have a robot build a 3D model of it's surroundings could emphasize displaying the current state of the model being built, frame rate and values of the pertinent sensors, available storage space on the device and other such task-specific information, while the interface for the device while driving it from one location to another might be more concerned with vehicle safety and stability. For each condition only the necessary messages would be sampled. This level of customization maximizes the usefulness of the interaction device while at the same time minimizes usage of the network bandwidth. This concept could also extend to controllers designed for coordinating and monitoring a team of robots. Overseeing the status of such multi-robot trials is an extremely difficult task. Direct control of all vehicles via a single person is impractical. Finding ways to monitor such trials and potentially determining appropriate methods for human interactions with the robot team during these trials are interesting and unique challenges. The modular and standardized nature of the controller architecture is particularly well suited for exploring such work.

## ACKNOWLEDGMENTS

The technical and diving assistance of Jim Zacher and Heather Jenkin is gratefully acknowledged. This project would not have been possible without the generous financial support of the Canadian Foundation for Innovation and support by the Natural Sciences and Engineering Research Council (NSERC) through the NSERC Canadian Field Robotics Network (NCFRN).

## SOURCE CODE

The framework described in this paper along with example widgets and robot interfaces is available for download at http://vgrlab.cse.yorku.ca.

## REFERENCES

[1] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Y. Ng, "ROS: an open-source robot operating system," in *Proc. Open-Source Software workshop at the International Conference on Robotics and Automation (ICRA)*, 2009.

[2] S. Wu and Y. Chen, "Remote robot control using intelligent hand-held devices," in *Fourth Int. Conf. on Computer and Information Technology*, Wuhan, China, 2004.

[3] N. Fung, "Light weight portable operator control unit using an Android-enabled mobile phone," in *Proc. SPIE Unmanned Systems Technology XIII*, Orlando, FL, 2011.

[4] P. Rouanet, J. Béchu, and P.-Y. Oudeyer, "A comparison of three interfaces using handheld deives to intuitively drive and show objects to a social robot: the impact of underlying metaphors," in *Proc. 18th IEEE Int. Symp. on Robot and Human Interactive Communication*, Toyama, Japan, 2009.

[5] L. Bento, R. Prates, and L. Chaimowicz, "Designing interfaces for robot control based on semiotic engineering," in *Proc. IEEE Int. Conf. on Systems, Man, and Cybernetics*, Anchorage, Alaska, 2011.

[6] ROSJAVA, "Rosjava wiki," http://wiki.ros.org/rosjava, accessed September 19, 2013.

[7] W. Garage, "Rospod wiki," http://wiki.ros.org/rospod, accessed September 19, 2013.

[8] Brown University, "rosbridge," http://www.rosbridge.org, Feb. 2013, accessed February 10, 2013.

[9] I. Hickson, "The WebSocket API," W3C Working Draft 16 August 2010, W3C.

[10] Tavendo, "Autobahn websocket," http://autobahn.ws, Feb. 2013, accessed February 10, 2013.

[11] S. Ben-Kilko, "Yaml ain't markup language," http://www.yaml.org, accessed March 12, 2013.

[12] D. Corckford, "The application/json media type for JavaScript object notation JSON," Netowrk Working Group RFC 4627, 2006.

[13] A. Chopra, M. Obsniuk, and M. Jenkin, "The Nomad 200 and the Nomad Superscout: Reverse engineered and resurrected," in *Proc. 3rd Canadian Conf. on Computer and Robot Vision (CRV)*, Quebec City, Canada, 2006, pp. 55–62.

[14] M. Robinson and M. Jenkin, "Reactive low level control of the ARK," in *Proc. Vision Interface*, 1994, pp. 41–47.

[15] A. Calce, P. M. Forooshani, A. Speers, K. Watters, T. Young, and M. Jenkin, "Autonomous aquatic agents," in *Proc. ICAART*, Barcelona, Spain, 2013.

[16] G. Dudek, M. Jenkin, C. Prahacs, A. Hogue, J. Sattar, A. German, H. Liu, S. Saunderson, A. Ripsman, S. Sinhon, L. Torres, E. Milios, P. Zhang, and I. Rekleitis, "A visually gudied swimming robot," in *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, 2005, pp. 1749–1754.

[17] B. Verzijlenberg and M. Jenkin, "Swimming with robots: human robot communication at depth," in *Proc. IEEE/RSJ Int. Conf. on Robotics and Intelligent Systems (IROS)*, 2010.

[18] A. Speers and M. Jenkin, "Diver-based control of a tethered unmanned underwater vehicle," in *Proc. ICAR*, Reykjavik, Iceland, 2013.

[19] S. Gupte, P. Mohandas, and J. Conrad, "A survey of quadrotor unmanned aerial vehicles," in *Proc. of IEEE Southeastcon*, 2012.

[20] H. Lim, J. Park, D. Lee, and H. J. Kim, "Build your own quadrotor: open-source projects on unmanned aerial vehicles," *IEEE Robot. Automat. Mag.*, vol. 19, pp. 33–45, 2012.