Robots as web services: Reproducible experimentation and application development using rosjs

Sarah Osentoski, Graylin Jay, Christopher Crick, Benjamin Pitzer, Charles DuHadway, Odest Chadwicke Jenkins

Abstract— **We describe our efforts to create infrastructure to enable web interfaces for robotics. Such interfaces will enable researchers and users to remotely access robots through the internet as well as expand the types of robotic applications available to users with web-enabled devices. This paper centers on rosjs, a lightweight Javascript binding for ROS, Willow Garage's robot middleware framework. rosjs exposes many of the capabilities of ROS, allowing application developers to write controllers that are executed through a web browser. We discuss how rosjs extends ROS and briefly overview some of the features it provides. rosjs has been instrumental in the creation of remote laboratories featuring the iRobot Create and the PR2. These facilities will be available to the community as experimental resources. We describe the overall goals of this project as well as provide a brief description of how rosjs was used to help create web interfaces for these facilities.**

I. INTRODUCTION

Equipping robots with web interfaces will extend the reach and impact of robotic technology. Not only will such interfaces allow researchers and users to access robots remotely through the internet, but they will also greatly expand the ecosystem of robotics applications available to users with web-enabled devices. This paper describes our efforts to create infrastructure to support such applications and devices. Web-enabled robotics will allow users to control robots within their home or workplace through a variety of interfaces on their phone or computer. This technology will also permit the creation of remote robot labs for shared experimentation and developments. Researchers will be able to perform experiments on remote systems to which they would not otherwise have access, and compare results to others in a shared lab environment.

Several robot middleware system have been proposed to enable code sharing among roboticists. These middleware systems include Player/Stage [1], the Carnegie Mellon Navigation Toolkit (CARMEN) [2], Microsoft Robotics Studio [3], YARP [4], Lightweight Communications and Marshalling (LCM) [5], and ROS [6], as well as other systems [7]. These middleware systems provide common interfaces that allow code sharing and reuse. While middleware systems differ in their design and features, they typically provide a communication mechanism, an API for preferred languages, and a mechanism for sharing code through libraries or drivers. Middleware systems typically require developers to code within the middleware framework, and often within a specified build environment.

This paper introduces rosjs[1], a lightweight Javascript binding for ROS that allows web developers to create robot applications. Currently robot application development occurs, at best, within a robot middleware framework. Robot middleware systems are often large and complicated, thus requiring developers acquire a substantial amount of expertise before they can be productive. However, much of this low level complexity can be hidden from the application programmer using abstractions.

rosjs allows developers to program robot applications for the web by exposing ROS topics and services as Javascript objects. rosjs provides security mechanisms that allow users to reserve time on the robot and authenticate their identity using services like Google Calendar. It also provides the ability to perform data logging of user experiences, in service to experiments employing user studies. rosjs was implemented as part of our larger goal of creating experimental infrastructure for remote robotics laboratories. We describe two remote laboratories and how rosjs was instrumental in creating these facilities.

II. PREVIOUS WORK AND BACKGROUND

One of the strengths of rosjs is its support for quickly and easily creating remote user interfaces. Much of the teleoperation work in robotics has traditionally been aimed at tasks where robots operate in environments that are hazardous to human users, such as robotic surgery [8], search and rescue [9], and outer space [10]. In these applications, users are typically experts who have devoted a significant amount of training time to the difficult task of controlling the robot and interacting with its interfaces. Our goal with rosjs is to allow application developers to create interfaces that are intuitive even for novice users.

The robotics community has made a few forays into using robots over the internet and shared-time robotics. Burgard

This work was supported in part by the following: PECASE N00014-08-1-0910, NSF Career IIS-0844486, ONR YIP FA9550-09-1-0206. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

S. Osentoski, G. Jay, C. Crick and O.C. Jenkins are with the Computer Science Department, Brown University {sosentos | tjay | chriscrick | cjenkins }@cs.brown.edu

B. Pitzer, C. DuHadway are with Research & Technology Center North America, Robert Bosch LLC { Benjamin.Pitzer | Charles.DuHadway }@us.bosch.com

[1]http://code.google.com/p/brown-ros-pkg/wiki/rosjs

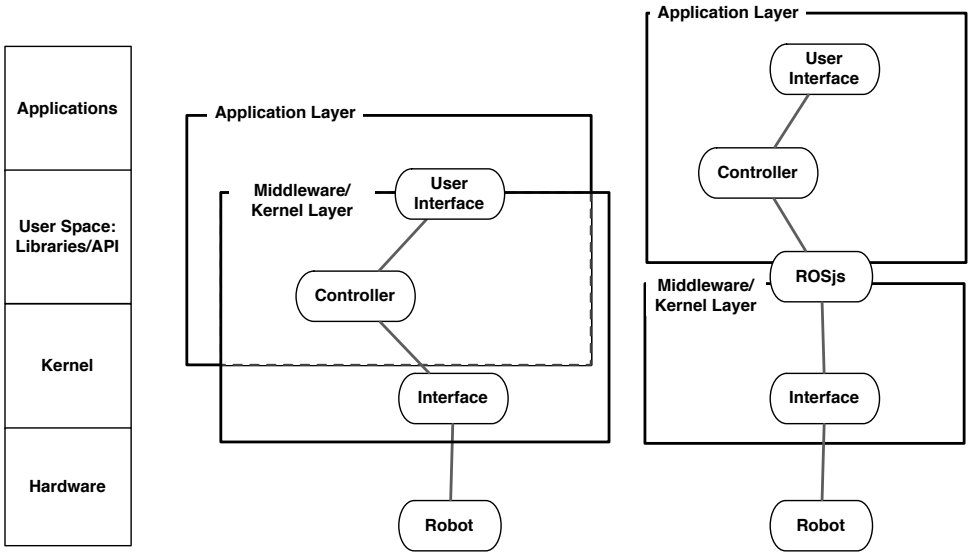

Fig. 1: Recreating traditional abstraction layers in robotics with rosjs. As depicted at left, software development depends on well-established layers of abstraction. Developers and engineers working at each layer possess very different skill sets, but the enterprise succeeds due to well-defined abstractions and interfaces. At present, roboticists must deal with all of these layers at once, limited by both their own skills and by the unwieldiness inherent in poorly-abstracted systems (center). At right, rosjs attempts to establish a clear abstraction boundary to address this problem.

and Schulz have explored handling delay in remote operation/teleoperation of mobile robots using predictive simulation for visualization [11]. Goldberg et al. placed a robot in a garden and allowed users to view and interact with the robot over the web. Users were able to plant seeds, water, and monitor the garden [12], [13]. Taylor and Trevelyan created a remote lab in which users perform tasks involving brightly colored blocks [14]. These early efforts made exciting first steps into examining shared-time robots. Our interest is on creating a community resource to enable researchers to reproduce and share research results. Additionally, previous approaches often require users to download custom software or plugins. rosjs enables the interfaces to be created without requiring any additional software other than an appropriate browser.

A. REVIEW OF ROS

ROS is an open-source robot operating system currently maintained by Willow Garage. ROS uses a peer-to-peer networking topology; systems running ROS often consist of a number of processes called *nodes*, possibly on different machines, that perform the system's computation. Nodes communicate with each other by passing messages. Under ROS, messages are data structures made up of typed fields. Messages may be made up of standard primitive data types, as well as arrays of primitives. Messages can include arbitrarily nested structures and arrays.

Nodes can use two types of communication to send messages within the ROS framework. The first is synchronous and is called a *service*. Services are much like function calls in traditional programming languages. Services are defined by a string name and a pair of messages: a request and a response. The response returns an object which may be arbitrarily complex, ranging from a simple boolean indicating success or failure to a large point cloud data structure. Only one node can provide a service of a specific name.

The second type of communication is asynchronous and is called a *topic*. Topics are streams of objects that are published by a node. Other nodes, "listeners", may subscribe by registering a handler function that is called whenever a new topic object becomes available. Unlike services, listener nodes are unable to use their subscription to the topic to communicate to the publisher. Multiple nodes may concurrently publish and/or subscribe to the same topic and a single node may publish and/or subscribe to multiple topics.

Unlike many other robot middleware systems, ROS is more than a set of libraries that provide only a communication mechanism and protocol. Instead, nodes are developed within a build system provided by ROS. The intent is that a system running ROS should be comprised of many independent modules. The build system is built on top of CMake [15], which performs modular builds of both nodes and the messages passed between them.

B. WEB APPLICATIONS AND JAVASCRIPT

Computing paradigms have developed over the years, from batch systems to timeshared mainframes to standalone desktops to client-server architectures to ubiquitous web-based applications. Current technology allows transparent administration, redundant storage, and instantaneous deployment of software running on wildly heterogenous platforms, from smartphones to multicore desktops. This relatively new

and extremely fertile ecosystem has spawned a population of users who understand basic web technologies such as HTML and Javascript [16]. Familiarity with basic web technologies extends beyond expert application developers to users who would not necessarily call themselves programmers, but who nevertheless use the web for all manner of creation and communication and are familiar with the basic technologies. One of the goals of rosjs is to broaden robotics to this vast untapped population of writers, artists, students, and designers. Javascript has become the default language of the web and as such is one of the most popular languages in the world. We hope to leverage a small part of that popularity to open robotics to an entirely new audience and to make working with robotics easier for those who are already familiar.

rosjs is designed to integrate ROS with the web as unobtrusively and universally as possible. Its only advanced dependency is on the HTML5 [17] technology of websockets. Currently browsers such as Safari, Opera, and Chrome fully support them, as does the nightly build of Firefox. Universality has been one of the key factors in the success of the web, and accordingly rosjs is implemented as a simple Javascript library, completely agnostic with respect to preferred development frameworks. It uses JSON, the Javascript object syntax itself, to communicate with its backend.

III. ROSJS

rosjs was designed to meet the needs of developers with web programming experience. There are multiple advantages to the ability to develop robot applications in the browser. The first is that web browsers are a familiar interfaces that are widely used, even by nontechical users. Additionally, allowing users to access robots through the internet may provide insights into new applications for robotics, as well be used as a tool to recruit potential scientists to the field. Javascript allows for rapid and flexible user interface and visualization development. Applications developed within a web browser are also portable across platforms, and updates and new functionality can be easily provided.

rosjs was also designed to provide an additional level of abstraction on top of ROS, as depicted in Figure 1. As a point of comparison, traditional operating system architectures build on top of hardware with increasing levels of abstraction. The current state of many robot middleware packages is that application developers must do a significant amount of developing within the middleware layer. At a minimum they must understand the build and transportation mechanisms of the middleware package. rosjs adds another layer of abstraction by presenting itself as a library/api. This allows experienced ROS developers to prepare a ROS environment within which application developers with less robotics experience can create applications.

ROS abstracts individual robot capabilities, allowing robots to be controlled through messages. It also provides facilities for starting and stopping the individual ROS. rosjs encapsulates these two aspects of ROS, presenting to the user a unified view of a robot and it's environment. If we continue with the operating system metaphor: a modern operating system allows a programmer to interact with hardware through a standardized API. The operating system supports a division of labor between installing and maintaining a particular collection of device drivers (a function often performed by IT personnel) and using the capabilities of that hardware (a function most performed by programmers). rosjs supports a similar division of labor by encapsulating the ROS development environment into a library. It is relatively easy for an experienced ROS user to prepare a ROS environment and to allow a less experienced user leverage this environment through rosjs. rosjs users need only understand the core concepts of topics and services. Other details of ROS such as launch files, starting a roscore, and networking become superfluous.

By wrapping ROS, rosjs extends its capabilities. For example, by acting as an intermediary between ROS and the user, rosjs is able to provide authentication and other security mechanisms that ROS itself lacks. We describe these and some additional features in the next subsections.

A. FEATURES

1) Exposure of ROS Services and Topics: To use rosjs users first create a rosjs object that is can be used to interact with a robot using ROS services and topics.

rosjs provides interfaces to both publish and subscribe to ROS topics. rosjs allows users to publish through a simple publish command. rosjs handles listening to topics by registering handler callbacks. The developer first registers a callback with the rosjs object and then uses a service call to register your interest in the topic with rosjs. Additionally, developers can publish topics

rosjs interacts with ROS services via the callService function, one of the members of the ROS object. Under rosjs services act as they do in ROS.

rosjs also provides a number of services that are accessed as any other ROS service. However, these services are only accessible from rosjs and cannot be used by another ROS node. rosjs provides services that:

- allow the user to access the currently available topics and services
- list the topics rosjs is subscribed to
- access the type of message of the topic/service
- access objects associated with the type of a requested topic/service
- handle key authentication for rosjs

2) Security: When developing a robotic web application that will be broadly available over the internet, security is a fundamental concern. rosjs provides two forms of security: protected services/topics and key authorization. Protected topics and services are necessary when there are critical services that the client should not interfere with – for example, those that enforce human or robot safety. The mechanism is straight-forward; protected topics and services simply start with the string "protected". Attempts to subscribe or publish to protected topics will not result in message transfer. Calls upon protected services will return an empty object.

Key authorization allows the developer to limit access to a robotic web application to specific users. rosjs implements a generic authentication mechanism based on JSON with padding (jsonp). If provided with the URL of an appropriate web-service, rosjs will use it to authenticate potential clients. The jsonp web service should simply take in an alphabetic key and return an amount of time (in seconds) that the key is authorized to use rosjs. When operating in this mode, rosjs expects the first action any client will take is to provide such a alphabetic key. Any other action results in being disconnected. Once authorized, a client can take normal actions until the time it was allotted by the authorization web service expires. Using this simple infrastructure, developers can easily bind rosjs to any authentication mechanism they choose. On machines where appropriate SSL libraries are available, rosjs will automatically support SSL (https) keyurls. rosjs does not take any steps to throttle connection attempts nor does it provide any other form of DOS (denial of service) protection. Users are encouraged to handle this issue at the port or firewall level.

3) Data Logging: A data logging mechanism is key for recording data from experiments. rosjs provides a mechanism for developers and users to save data. The data can be saved locally to a file that can then be uploaded. ROS developers can still log data using ROS on the server side but the additional information can be used to collect data about the user's experience. This functionality is likely to be necessary for for fields where user studies are necessary such as Robot Learning from Demonstration (LfD) and Human Robot Interaction.

B. UNDERLYING TECHNOLOGY

The major underlying design decision of rosjs was to use HTML 5 websockets as the transportation layer. In order to perform control, robotic web applications must be able to transmit data from the robot to the user quickly. Websockets provide the ability for the bidirectional communication between the browser over a single-socket TCP/IP connection. Websockets lack the message overhead of technologies like AJAX which require HTTP headers for each message. The use of websockets generally results in higher speeds and lower latency.

rosjs currently uses a custom websocket implementation since a standard version does not yet exist. We hope to move to a community provided library or implementation when one becomes available. One of the benefits of using websockets is that the server hosting the robot application can be different from the server hosting rosjs.

IV. REMOTE LABORATORY

While middleware systems allow for code sharing and reuse, many researchers are limited by the overhead (and sometimes pure impossibility) of reproducing results on similar platforms. Large platforms like mobile manipulators are expensive and difficult to obtain for researchers at smaller institutions or companies. It is rare for researchers to have access to common platforms, let alone shared data.

A remote robotic laboratory would allow researchers to run experiments and compare against results produced on a common platform. rosjs was developed as part of our work towards developing the experimental infrastructure for the creation of remote robotic laboratories.

We envision experimental facilities in which users will schedule experiments using tools like Google calendar. Often, researchers will be able to code the entire robot interaction within a browser, making use only of the available robot capabilities exposed through rosjs. For more elaborate needs, the robot can be given a pointer to an subversion (svn) or git repository containing the desired experimental code, which will then be run locally. To encourage code sharing, preference for time on the robot will be given to code stored in public repositories. Data resulting from experimental trials will be uploaded to a public repository and linked to code used in the corresponding trial. Since this facility will be available on the web it will enable communities that require user studies, such as human-robot interaction and learning from demonstration, to acquire a significant amount of data – finally moving them beyond "proof-of-concept" demonstrations.

Figure 2 depicts our vision of how a remote robotics lab will work specifically in the case of learning from demonstration. Users schedule time on the robot and can interact with the it using the remote interface. This interface can be used to demonstrate tasks or to visualize the robot as it performs tasks provided by custom built controllers. During each session data is logged and stored in a publicly available repository. Custom controllers and learning algorithms, provided in public code repositories, can use the data and provide policies for desired tasks on the robot.

There are many technical challenges to address when creating such a remote lab. The functionality provided by rosjs is instrumental to overcoming them. A web interface is required so that users can work with the robot remotely. The user must have some way of controlling the robot, either with code or through teleoperation. Users must also be able to visualize the result of the control. Security measures are required for the safety of the robot. In the next section, we describe two remote labs, pictured in Figure 3, and describe how they handle these requirements.

Fig. 3: Two different remote labs currently under development. The first depicts an iRobot Create learning to navigate a maze. The second contains a PR2 that can manipulate objects on a table.

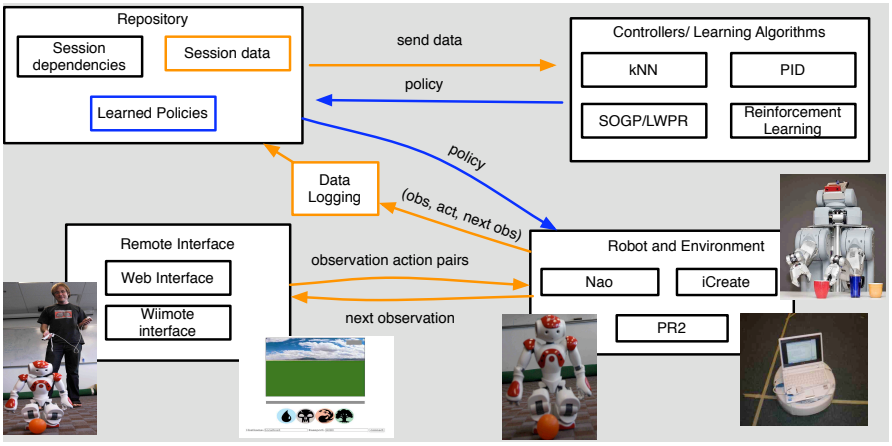

Fig. 2: Conceptual overview of a remote laboratory. Users can interact with robot platforms, log and retrieve data in a public repository, and access both standard and custom controllers and learning algorithms.

V. Example Robotic Experimental Environments

A. iCreate Maze

The iCreate Maze allows users to demonstrate the correct path through a maze to an iRobot Create. The website allows a user to access the robot if it is not currently in use. Users are randomly presented with one of two interfaces. The first shows a compressed video feed from the robots camera. The second shows a visualization of the robot's perceptual space.

1) Augmented Reality Aided Visualization: In order to create a lightweight perceptual space we employed Augmented Reality (AR) tags. We use AR tags in robotics domains, like the maze navigation, to augment or replace vision algorithms. AR tags can be used to mark and identify locations or objects of interest. The tags are recognized using our publicly available *ar_recog* ROS node[2]that wraps the ARToolkit [18], a software library for building AR applications. ar_recog returns the position and id of the tags within the current visual frame. Thus, tags can be easily visualized over the web since only the location information must be sent and the browser can render the appropriate visualization in terms of scale and location.

B. PR2 Manipulation

Interacting with a mobile manipulator such as the PR2 is a more complex task than an iCreate. Thus the remote lab interface for the PR2 is substantially more complex and includes 3d visualization of the robot using WebGl as well as authentication procedures for security.

1) 3D Web Visualization: One of the most useful tools in robot development is a 3D visualization environment. The remote lab interface contains a widget that provides this functionality. Whether robot models, poses, maps, or custom visualization markers, the visualization widget can

[2]http://code.google.com/p/brown-ros-pkg/wiki/ar_recog

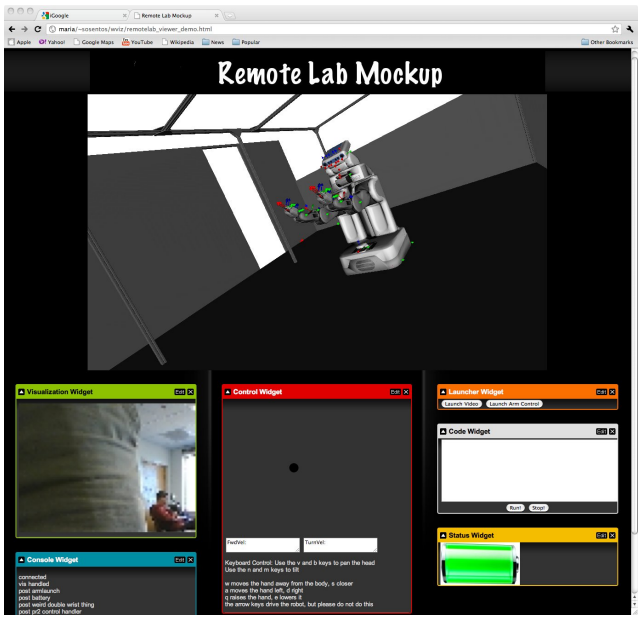

Fig. 4: Visualization of a remote lab interface that can be used to teleoperate a PR2 for tasks such as object manipulation.

display views of various types of robot data. The underlying technology for visualizing 3D data on the web is WebGL, a 3D graphics API implemented in a web browser without the use of plug-ins. Similar to OpenGL, WebGL provides a programmatic interface for 3D graphics using the OpenGL ES 2.0 standard. Our 3D visualizer provides an interface to WebGL based on world objects and other high level classes. It also provides a scene graph, which is an object-oriented representation of the 3D world. Such a representation is especially suitable for robotic development, since data is rep-

resented in various interdependent frames. It also simplifies the extension to new data types since scene nodes can be implemented for any data type and inserted into the graph, as long as they adhere to a common interface.

2) Authentication using Google Calendar: Earlier in the paper we described rosjs's generic authentication mechanism. In this section we describe a concrete example of how the authentication mechanism was implemented for the PR2 Manipulation remote lab. In this lab authorization was tied to Google's calendar and OpenID services. This involved writing two web services: the key authenticator and key provider. The authenticator is little more than a wrapper that speaks the jsonp format expected by rosjs (see Section III-A.2). The key provider handles communicating with Google.

A typical successful authentication proceeds as follows. The user arrives at a login page for the robot. This page uses Javascript to speak with the key provider and receives a novel alphabetical challenge string. The page then constructs an OpenID authentication request URL that it attaches to a "login" link on the page. When the user clicks this link, they are taken to Google's servers to perform the actual login. The URL encodes both the challenge and the location of the robot control page. If the user successfully logs into a Google account, Google will send the user to the robot control page and pass on (again, through URL parameters) the challenge, the relevant account information, and a confirmation token. The robot control page passes this information onto the provider which files it into persistent storage. In exchange it receives a key associated with the storage location. In the current system, it is this key that acts as the authentication expected by rosjs. The robot control page can now start a rosjs connection and use the key to authenticate itself, matched against the corresponding token. The provider thereby knows what Google account is making an authentication request, checks for a current appointment with Google's calendaring system, and determines the amount of time remaining in the appointment. Finally, the authenticator passes this information back to rosjs.

This scheme, as currently implemented, does not provide any protection against denial of service (DOS) attacks nor man in the middle attacks (the one-time key is given to rosjs over an open socket). It does protect against replay attacks and identity theft. This seems to be a common compromise amongst commercial web services. For example, both Facebook and Google use SSL only during their authentication stage and thereafter are more open to session hijacking. However, it should be noted that the authentication scheme detailed here represents the beginning rather than the end of what is possible. Because rosjs supports an outside (optionally SSL enabled) communication channel, developers are free to implement more complicated challenge-response mechanisms if they so choose.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we described rosjs, a Javascript library for ROS, that allows developers to expose robot functionality as web services. rosjs allows developers to create robot applications that can be used in the web browser and extends ROS to provide security and data logging mechanisms. We also described infrastructure we have developed to allow remote experimental robot laboratories and showed progress on two remote laboatories. We plan to continue extend the remote laboratories to add functionality and additional tasks and provide these labs to the research community. We also plan to expand our suite of remote lab capabilities and infrastructure, to include supporting hobbyists in deploying remote labs of their own. We are currently developing a soccer-playing setup using NXT-based Lego robots in support of this goal.

REFERENCES

[1] B. Gerkey, R. T. Vaughan, and A. Howard, "The player/stage project: Tools for multi-robot and distributed sensor systems," in *Proceedings of the 11th International Conference on Advanced Robotics*, 2003, pp. 317–323.

[2] K. Wyobek, E. Berger, H. V. der Loos, and K. Salisbury, "Perspectives on standarization in mobile robot programming: The Carnegie Mellon Navigation (CARMEN) Toolkit," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2003, pp. 2436–2441.

[3] J. Jackson, "Microsoft robotics studio: A technical introduction," *Robotics & Automation Magazine, IEEE*, pp. 82–87, 2007.

[4] G. Metta, P. Fitzpatrick, and L. Natale, "YARP: Yet Another Robot Platform," *International Journal on Advanced Robotics Systems*, pp. 43–48, 2006.

[5] A. Huang, E. Olson, and D. Moore, "LCM: Leightweight Communications and Marshalling," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2010.

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

[7] J. Kramer and M. Scheutz, "Development environments for autonomous mobile robots: A survey," *Autonomous Robots*, pp. 101–132, 2007.

[8] A. Okamura, "Methods for haptic feedback in teleoperated robot-assisted surgery," *Industrial Robotics*, pp. 499–508, 2004.

[9] J. L. Casper and R. R. Murphy, "Workflow study on human-robot interaction in USAR," in *Proccedings of the 2002 IEEE International Conference on Robotics & Automation)*, 2002, pp. 1997–2003.

[10] H. Aldridge, W. Bluethmann, R. Ambrose, and M. Diftler, "Control architecture for the robonaut space humanoid," in *Proccedings Humanoids 2000, The 1st IEEE/RAS Conference on Humanoid Robots*, 2000.

[11] W. Burgard and D. Schulz, *Beyond webcams: an introduction to online robots*. Cambridge, MA, USA: MIT Press, 2002, ch. Robust visualization for online control of mobile robots, pp. 241–258.

[12] K. Goldberg, S. Gentner, C. Sutter, and J. Wiegley, "The mercury project: A feasibility study for internet robots," *Robotics & Automation Magazine, IEEE*, pp. 35–39, 1999.

[13] K. Goldberg, H. Dreyfus, A. Goldman, O. Grau, M. Gržinić, B. Hannaford, M. Idinopulos, M. Jay, E. Kac, and M. Kusahara, Eds., *The robot in the garden: telerobotics and telepistemology in the age of the Internet*. Cambridge, MA, USA: MIT Press, 2000.

[14] K. Taylor and J. Trevelyan, "A telerobot on the world wide web," in *National Conference of the Australian Robot Association*, 1995.

[15] K. Martin and B. Hoffman, *Mastering CMake: A cross-platform build system*. Kitware Inc, 2008.

[16] *ECMA-262: ECMAScript Language Specification, 5th ed*, E.C.M.A. International Std., December 2009. [Online]. Available: http://www.ecma-international.org/publications/standards/Ecma-262.htm

[17] M. Pilgrim, *HTML5: Up and Running*. O'Reilly Media, 2010.

[18] H. Kato and M. Billinghurst, "Developing ar applications with artoolkit," in *ISMAR*, 2004, p. 305.