Proceedings of the 2006 IEEE International Conference on Robotics and Automation
Orlando, Florida - May 2006

Opening the Dialog: Robotics and the Internet

Anthony Cowley
GRASP Lab
University of Pennsylvania
Philadelphia, PA 19104-6228
Email: acowley@seas.upenn.edu

Hwa-chow Oliver Hsu
GRASP Lab
University of Pennsylvania
Philadelphia, PA 19104-6228
Email: hwahsu@grasp.cis.upenn.edu

C.J. Taylor
GRASP Lab
University of Pennsylvania
Philadelphia, PA 19104-6228
Email: cjtaylor@cis.upenn.edu

Abstract— **Inter-component communication has received considerable attention by robotic software architects as various frameworks and toolkits have matured. While the resulting software platforms have proven useful for abstracting hardware interfaces and the complex networking issues that are often present in robot deployment scenarios, they typically present users with arduous paths for integrating new hardware and software, while making no allowance for humans as peers on the network. We explore how mature standards and protocols born of the Internet community can be leveraged to empower software designed for robots and sensor networks. Here we present software approaches that facilitate the integration of new hardware devices, software platforms, and humans in ways that are natural and intuitive for each.**

I. Introduction

Open standards, formats, and protocols are changing the way software developers work. The greater software community has come to understand that there is a network effect related to the success of a particular library, component, or program: the value of a piece of code rises exponentially with the number of other pieces of code with which it can interoperate. In fact, ease of interoperability is often as important, if not more so, than the specific functionality of the component. Yet time and time again, the same functionality is re-engineered because it is simply easier to do so than to wrestle with a systems integration scenario unconsidered by the original developer. This is not only a waste of development time, but also a futile strategy for future growth as the scope of a large software project becomes limited not by the development team's imagination, but by their ability to recreate existing functionality found on other platforms.

The limiting factor addressed in this paper is the context within which a robotics software package is developed. By avoiding the trap of too narrow a focus on the needs of real-time control and the boundaries of a single development team, we have been able to produce an accessible software framework built around adapting to the user, rather than the user adapting to the framework. Robots, sensor networks, and automated systems become significantly more valuable when they become players in the global pool of the Internet, rather than an isolated branch of engineering, and cater to their consumers be they electronic or biological.

II. Background

Software for automated systems, in particular mobile robotics, has thus far concentrated on identifying the key levels of abstraction necessary for efficient development [1] [2]. A general trend is the abstraction of hardware interface so as to minimize entanglement with higher-level behavior development. Typically, sensors and actuators are given software interfaces designed to be comprehensible and appropriate for behavior development. This abstraction serves to create flexibility in the underlying hardware implementation, to the extent that the hardware can be considered a service provider, easily replaced by alternative hardware solutions or software simulations. The goal is not necessarily to encourage hardware ignorance in software developers, but to encourage the reuse of tested, reliable software components [3] [4]. Using the same component in multiple scenarios provides a breadth of experience that allows developers to gain familiarity with commonly used components. This familiarity makes correct usage more likely, and encourages experimentation since the developer already has confidence in basic functionality.

The software framework discussed and extended in this paper, the Remote Objects Control Interface (ROCI), was originally designed with a focus on component design and communication abstraction built atop a strong type system with robust reflection capabilities [5]. The component design aspect of ROCI is based on the notion of creating irreducible translation components. In the ROCI context, a self-contained component is one which translates data from an input form into an output form. Such a component can accomplish its stated functionality with complete independence from context, and is therefore more likely to be used by other developers in novel application settings.

Once component functionality has been reduced to the notion of translation, the data structures that the translator operates over become the defining characteristics of the component itself. Many such components, referred to as Modules in ROCI terminology, have been constructed to interface with hardware devices, or implement particular algorithms. The resulting collection of self-describing components thus define a language with which to program a robot, or, more generally, any sensing and computing platform. The final programming of the target device takes place at a very high level where previously constructed, context-independent components are connected in order to accomplish some multi-stage task. For example, one component may interface with a camera at a device driver level, and output more general purpose video frames. These frames can be translated by another, purely algorithmic component into connected areas matching a predefined color range. The resultant blobs can then be translated into motor commands, which are finally translated into device commands that cause a pan mount to track a particular color. This entire process can be centralized on a single machine, or distributed across a network without any change to component code. Furthermore, the components that accomplish this high-level task may be started as a unit, or dynamically discovered over the ROCI network.

A. Communication Primitives

The ROCI framework pays particular attention to communication abstraction as a consequence of the desire to describe functionality using input and output types [6]. The notion that a functional block can be classified by the data structures it operates over requires that some care is taken in designing said data structures. While it may sound a burden, encouraging developers to consider input and output data structures before writing functional code has the benefit of making explicit the gross structure the functional code should follow. The data structures, in this design paradigm, establish the boundaries of the functional code; if there is functionality that does not map directly onto the input or output data structures, then refactoring is most likely needed to ensure that the component is truly singular in purpose. This progressive design review process, where data structure is compared to functional structure, encourages the creation of self-contained components that can be used across multiple projects.

Interface definitions in hand, the ROCI system then supplies each of these interfaces, defined by the aforementioned data structures and referred to as Pins in the ROCI nomenclature, with rich networking functionality. The result is that components can be connected locally in a single process, between processes on a single machine, or between multiple machines across a network. The ROCI Pin machinery guarantees that only compatible interfaces are connected, manages network-transparent polymorphism, automatically creates the most efficient connection between interfaces, and handles network balancing, throttling, buffering, and error handling.

B. User Interfaces: Software

Typical user interfaces (UIs) for robotic systems reflect both the state of robotics and the applications to which they have thus far been employed [7]. Most interfaces have, on some level, been created with the idea of integrating a human as another controller in a robot behavior; a controller that is relied upon to make complex, or critical, decisions. This role of human as just another real-time controller has spawned many UIs centered around the user keenly monitoring continuous data feeds, such as video, in order to maintain situational awareness so that when a decision needs making, the human operator is mentally prepared to do so. Such a role also fits in well with many robotic systems that have required regular human intervention to function correctly. In other words, the ostensible need to have a human always ready to step in when an emergency occurs somewhat lessens the sting of requiring a human operator to ensure any level of correct behavior; the operator is ready and capable, they may as well be utilized.

Of course, autonomous systems have become much more capable over time, and there is little doubt that this trend will continue at a rapid pace. Thus the role of today's human operator can fairly be stated as monitoring continuous data streams in order to infrequently step in when autonomous controllers are not capable of making certain critical decisions. The continuous attention of the human operator is therefore only truly necessary if such critical decisions are particularly time sensitive. As it happens, a large portion of the robotics community has focused on deployment scenarios such as military maneuvers, search and rescue, bio-terrorism, and other scenarios where response time is indeed of the utmost importance. The urgency and specific goals of these types of applications naturally necessitate constant human supervision, so, once again, the constant availability of a human operator is accepted as a requirement even for relatively robust autonomous systems.

An area that naturally demands an effort to minimize human involvement is the fielding of teams of robots. Such efforts have shown that an effective data filter is necessary to limit the volume of data presented to the human operator. While continuous diagnostic data can help the operator maintain situational awareness, automated systems that recognize critical events and present these to the user as compact events have proven effective [7] [8]. Such systems often work similarly to instant messaging desktop software applications: the user is presented with a visual alert designed to attract attention without causing undue disruption whenever there are unhandled messages. Such systems allow a multitude of agents to message a single operator while giving the operator some control over how, at what pace, and in what order, alerts are dealt with.

The development of such alert management systems, having so much in common with existing instant messaging applications present on many computers, prompts a closer look at the operator requirements of modern robotic systems. It is clear that while there are many critical applications for robotics and sensor networks that may require immediate human intervention, there are an increasing number of broad-appeal robotics efforts that can not reasonably require that a human monitor anything continuously. Given this relaxation, different models of interaction may be appropriate.

C. User Interfaces: Hardware

User interface issues are not limited to software; the choice of input device often characterizes how a system is used. A large display screen implies a base station that either does not move, or is mounted in a vehicle. Complex keyboard inputs imply that the user will be seated, or at least stationary, to make typing practical. Alternatively, small devices such as PDAs, mobile phones, or wearable computers often suggest operator mobility at the expense of interaction efficiency. Unfortunately, each of these platforms brings with it wildly different hardware capabilities, and thus necessitate different software approaches. Many off-the-shelf hardware platforms have a native software platform and associated API that may differ significantly from the core platform used to develop software for other devices.

Rather than wait for devices that support one's favorite development environment, we believe it far more prudent to embrace the differences between platforms. These differences often bring with them advantages along with the expected inconveniences. Rather than force a preferred development paradigm on the new platform, thus necessitating significant development effort to support each new device, and perhaps inadvertently advocating an overly homogenous future for software development, software frameworks can instead be constructed with unsupported devices in mind.

III. Integration Strategies

The view presented here is that there are a number of distinct integration problems that may be colloquially, yet accurately, viewed as problems in establishing meaningful conversations between entities in a system populated by robotic and human agents. Current software frameworks, as described above, have been primarily concerned with facilitating the development and execution of autonomous, or semi-autonomous, behaviors. As the reliability of these behaviors and the range of problems robots are applied to, increases, collaboration becomes the most critical element in the success or failure of any autonomous system initiative. Importantly, collaboration here is used to refer to interaction between disparate software platforms, hardware platforms, and agents, be they human or electronic. Success in these endeavors requires that all conversations between systems or agents are conducted in languages understandable to, and appropriate for, each entity.

Fortunately, the entire software community has been involved in the same work for many years. Many protocols, file formats, and standards have come and gone as software developers have attempted to benefit from each other's work, but one meta-platform has emerged as the ultimate success story of systems integration in the twentieth century: the Internet. The Internet and, specifically, the World Wide Web are concrete proof of the viability of protocols such as TCP/IP and HyperText Transfer Protocol (HTTP), as well as the associated markup languages and tools such as HyperText Markup Language (HTML), Extensible Markup Language (XML), Cascading Style Sheets (CSS), and Extensible Stylesheet Language (XSL). These standards are defining what a computer is more than any particular hardware feature or software package. The most sensible course for robotics, therefore, is to take practical advantage of the incredibly rich ecosystem these standards have created, and to take design guidance from the mores and values that led to their creation.

A. Platform Heterogeneity

Any assumption of a constant hardware or software platform is an artificial limit on the potential utility of a project. The incalculable value of the Internet as a mechanism for computation, expression, and commerce arises purely from the way that it openly embraces new members. In the field of robotics, integrating hardware and software platforms has implications for both systems-level collaboration and user interface modalities.

Ultimately, the issue is one of determining where in the software stack compatibility is required. If the requirement is to run existing, internally developed software, then the difficulty of porting that software becomes the bottleneck in integrating new hardware platforms, and collaboration with alternative software platforms remains an unanswered question. Simply put, the compatibility requirement should be the most broadly available standard that still meets the application needs of the development team. Today, the protocols associated with the Internet are being built into electronic components at an ever-accelerating rate, and clearly represent some of the most broadly adopted standards across platforms. Moreover, the Internet is home to a bewildering variety of software, thus demonstrating the viability of its common protocols as the foundation of many types of applications, in particular applications that benefit from interoperation and collaboration. The World Wide Web itself is a collaborative creation; one that encourages participation by its ease of use and flexibility of purpose.

Following this lead, ROCI directly addresses three principal network node classes: other ROCI nodes, non-ROCI systems, and humans. At the core of each communication transaction is some data that one node wishes to share with another. This core data is invariant; it is the truth of the transaction, and the technique necessary to effectively make this core accessible to all parties is translation. When built on a strong type system, each piece of core data is not a raw stream of bytes, but instead a structured manifest of information documenting the ancestors of the data type, hierarchical organization created by the designer, names for each field, and type information that imposes limits on the range of each field.

Using this information, ROCI provides static type compatibility checking across network connections, efficient serialization, available universal logging, and a distributed database system for all ROCI-to-ROCI communications. This type of connection is the most highly optimized, and is used for critical real-time controls. While some of the type-dependent features listed above are relatively unique to ROCI, this mode of data transport is roughly equivalent to a custom network protocol implemented by any system.

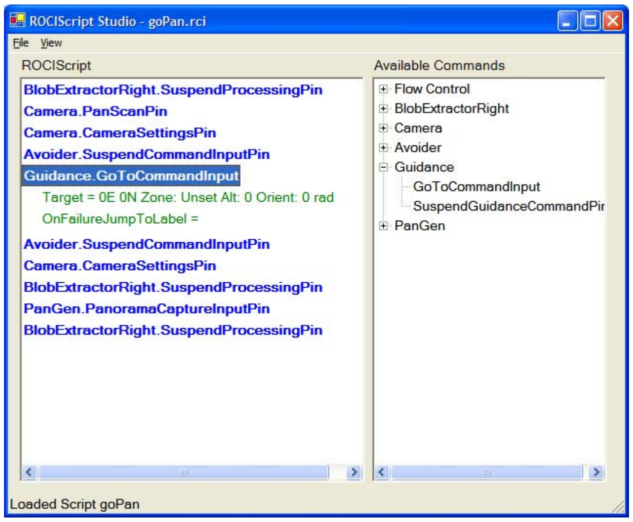
Efficient in many ways, such a system is also completely opaque to non-ROCI nodes. To address this, ROCI provides a simple web interface based on Representational State Transfer (REST) [9]. This interface is completely data driven, with XML as the data representation language. All data transactions on the ROCI network can be serialized as self-describing XML documents and are accessible through descriptive URLs and the basic verbs defined in HTTP. These documents, along with simple HTTP transactions, allow any platform capable of browsing the web to fully collaborate with ROCI nodes.

Finally, humans are only partially served by the availability of XML documents. While readable, these documents are clearly designed for parsing by a machine. To address this, each XML document returned by the web interface references XSL and CSS documents that, together, convert data-driven XML to presentation-ready HTML with flexible CSS styling. Thus a human using a web browser sees familiar web interaction widgets and formatting, while a machine sees easily parsed structured language. The same core data everywhere in the system is automatically serialized as an efficient byte stream, a structured language document, or a visually-pleasing website depending on who, or what, is consuming the data.

B. Human to Computer Communication

Integration between human users and automated systems, be they mobile robots or static sensor networks, involves considering how each naturally interacts with other members if its own class. That is, how humans interact with each other, and how computers most efficiently interact among themselves. While joysticks and control panels are excellent for real-time human interaction, the user experience of creating scripts, or programs, for the system is typically far less intuitive. Many robotic software frameworks do not address scripting specifically, or, if they do, suggest the application of a general purpose programming language to the problem of robot scripting. This model extends the design techniques used in creating the framework and associated components themselves to the domain of high-level task programming, and while it has the benefit of presenting the technical user with a familiar development environment, it does not reflect the target platform. General purpose programming languages are designed for programming general purpose computing machines, and thus reflect that generality and flexibility. This flexibility has made languages such as C, Fortran, and Java extremely successful, but that success is due to the fact that the tool matches the problem. Modern robots and sensor networks, however, represent a more specific high level problem domain. In general, no fixed-vocabulary scripting language captures the specialization built into today's robots and sensor networks.

Consider the situation of one human giving another driving directions. The director specifies steps that are at the highest level reasonably expected to be understood by the recipient. That is, the director does not confuse the issue by specifying how the traveler should contract his muscles, or operate his vehicle. In fact, the entire vocabularies related to those tasks will not appear in the final directions. The vocabulary of the directions will instead consist entirely of a small set of parameterized verbs that are all related to high-level intermediary goals involved in driving an automobile between two places. The low-level activities necessary to accomplish each of these intermediate goals are implied, but not stated. Equally important is that the director does not include instructions that the recipient can not reasonably be expected to execute. Driving directions do not include instructions that assume the recipient can fly, for example. Similarly, the same directive is typically referred to by the same symbolic representation, i.e. word, each time it occurs in the instructions. Variety in language makes for pleasant prose, but confusing directions.



```
BlobExtractorRight.SuspendProcessingPin
Camera.PanScanPin
Camera.CameraSettingsPin
Avoider.SuspendCommandInputPin
Guidance.GoToCommandInput
    Target = 0E 0N Zone: Unset Alt: 0 Orient: 0 rad
    OnFailureJumpToLabel =
Avoider.SuspendCommandInputPin
Camera.CameraSettingsPin
BlobExtractorRight.SuspendProcessingPin
PanGen.PanoramaCaptureInputPin
BlobExtractorRight.SuspendProcessingPin
```

```
⊞ Flow Control
⊞ BlobExtractorRight
⊞ Camera
⊞ Avoider
⊟ Guidance
    GoToCommandInput
    SuspendGuidanceCommandPir
⊞ PanGen
```

Fig. 1. Commands implemented by the mobile robot program being scripted here are shown in a toolbox area on the right of the script creation tool. These commands can be dragged into the script area on the left. Parameters for each command are automatically extracted, and type inspection is used to generate appropriate UI elements such as drop-down boxes, or list boxes. The script creation utility also supports a plug-in architecture for generating more complicated parameters, such as selecting waypoint locations from a map interface.

Note here that instructions given at too low a level will often include unwanted variety as there are usually many ways to construct a high level concept from low level components. Thus the vocabulary used in tasking a human is chosen such that it is capable of expressing all necessary functionality while simultaneously minimizing the instruction set and the number of instruction instances necessary to accomplish the goal.

The scripting of automated systems must be done with the same semantic mapping between instructions and behavioral capabilities. With this in mind, we have created a scripting component, ROCIScript, that manages dynamic script block injection and synchronization between scripted agents. More importantly, the scripting model is based on the creation of an application-specific scripting domain language gleaned from the components that make up the robot software, Fig. 1. The code that will run on the robot is inspected to ascertain what script commands are implemented, and these commands are presented to the script author as parametric verbs to be serially listed in a simple script structure. There is no large, fixed script vocabulary that might include elements not supported by the system being scripted [10]. Instead, the only two always available script instructions enable synchronization between agents and script flow control via label-based jumps, while the working instruction set consists entirely of instructions relevant to the scripted system's designed capabilities. In this fashion, an appropriate language for scripting each system is automatically constructed, thus establishing a contract between the script author and the system being scripted that specifies what constitutes a syntactically valid script.

Once a piece of ROCIScript has been created, it is serialized into an XML file. This file lists the components that implement the instructions used in the script, so that the script can be used across a variety of systems that make use of the same components, and the script instructions themselves. Using XML as a storage format allows any software with access to an XML processing library to re-parameterize the script with basic DOM manipulation. In conjunction with the ROCI web interface, high-level behavioral scripting of ROCI nodes is made available to any platform capable of XML manipulation and HTTP transactions. These requirements are easily met by many robust, well-documented libraries available in any modern development environment.

C. Computer to Human Communication

The scripting system above describes a mechanism to guide script authors, be they human or automated, in the creation of scripts that can always be understood by their recipients, express goals in as high a level as is supported by the specific target system, and are communicable via widely-supported protocols. This activity can be seen as making the script writing capabilities of the script author accessible to, and consumable by, the system to be scripted. Of course, it would also be advantageous to have data and services flow in the other direction in a manner as natural to the recipient as ROCIScripts are to ROCI nodes.

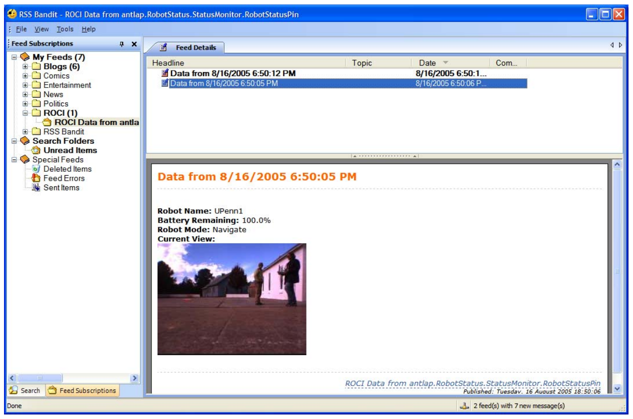
Typically, data collected by sensor networks or mobile robots has been delivered for human consumption through the use of custom applications. While these applications provide an opportunity for exploring new interface methods, they are also an inconvenience to many users and hinder integration at the desktop level. First, while custom applications offering new ways to interact with data streams may offer some benefits to the user willing to learn a new way of consuming data, those benefits must be weighed against the cost of inconvenience for the user. Many of today's systems prove to be systems designed by engineers for engineers, a situation well-suited to research activities, but one that ignores the many potential users of robotics. Most computer users have data acquisition, perusal, and dissemination habits that they have slowly developed over time as they have become comfortable with some mechanisms and rejected others. The most successful, and familiar, methods of interaction today are the World Wide Web, and the epistolary ease of email. People have become comfortable with these methods of acquiring information from friends, colleagues, and corporations; they have established these channels into their personal data-sphere. To require a new, unfamiliar, custom application is to burden the user unnecessarily.

The second way in which custom data interfaces hinder adoption is their segregation of the new data streams from the ones the user is accustomed to. The user familiar with reading and writing email, subscribing to RSS feeds of favorite web sites, and general web forms interaction is inadvertently being forced to acknowledge that data generated by robots is in some way different since it can only be consumed by a new application. This fencing off of certain data by virtue of the way it was collected and published discourages use and makes it far more difficult to combine with existing data channels.

The solution we have implemented in the ROCI software framework is an architecture for formatting and dissemination extensions. These Pin Exporter extensions, or plug-ins, can inspect any interface data structure—i.e. ROCI Pin—to translate or format the actual data in a way that is appropriate for a given export mechanism. Exporters now exist for maintaining RSS feeds for outputs, or for emailing the output data to any number of recipients, Fig. 2. The philosophy being espoused here is that, to the automated system, a human is just another service like any other software component, but the human works best when data is delivered in familiar ways. In this way, data efficiently consumed and acted upon by automated systems can also be efficiently consumed and acted upon by humans on the network. For example, a security system may have a component that recognizes individuals. The output of this recognizer will be used as an input to other components, ultimately opening a door, running a program, or taking some automated response. This output may also be of some interest to humans not actively monitoring the output. While the automated system may respond immediately, it may be more useful to allow a human user to consume data in the manner of his or her own choosing. Thus the availability of an RSS feed that embeds a picture of each recognized user, along with contextual information such as time, length of visit, etc. offers a convenient way to keep tabs on system usage without requiring any step out of the ordinary for someone who already monitors website updates via RSS. Similarly, a domestic aid robot, perhaps a vacuum cleaner, may encounter an unfamiliar obstacle while performing its daily tasks. This encounter should trigger some automated responses, such as stopping motors, but the data associated with the encounter can also be emailed to the owner. This type of operator notification represents a usage scenario quite apart from military or emergency response situations where immediate action is required. In the case of ROCI, a simple, structured response can be emailed back to the robot, perhaps to instruct it to resume its duties [1]. Or maybe the user just wants a weekly email from the house-cleaning robot confirming that everything is working properly. In any scenario, the goal is to provide the desired data to the human so as to be most easily consumed. If the user needs to install and use a new application, or, more generally, step out of their usual, comfortable data channels to check on robot status, the experience suffers.

The exporter system discussed here is not simply another interface mechanism that every component author needs to support. Instead, the human is consuming the exact same data as other system components are. The data is simply being formatted and delivered in a way designed to maximize user comfort and convenience. For ROCI-to-ROCI communication, data is formatted and delivered via the most efficient means possible, binary encoding and lean network packets. For part-

[1]Email responses are made possible by running an email server application modified to route emails to the ROCI web interface.



Headline	Topic	Date	Com...
Data from 8/16/2005 6:50:12 PM		8/16/2005 6:50:1...	
Data from 8/16/2005 6:50:05 PM		8/16/2005 6:50:06 P...	

Data from 8/16/2005 6:50:05 PM

Robot Name: UPenn1
Battery Remaining: 100.0%
Robot Mode: Navigate
Current View:

ROCI Data from antlap.RobotStatus.StatusMonitor.RobotStatusPin
Published: Tuesday, 16 August 2005 18:50:06

Fig. 2. ROCI Pins—the abstraction used to communicate data between components—can export their data to plain XML available over a RESTful Web interface, Email, or RSS (shown).

ner platforms, the very same data is always accessible as XML. Finally, for humans, that same data is available through a web browser, or formatted and delivered in a manner as natural for the receiver as possible. The sum of the parts described here is a loosely-coupled MVC (Model-View-Controller) architecture in which every interface data structure is automatically—free to the developer—available as a data source, the ROCI Kernel acts as access controller, and Pin Exporter extensions provide custom views of the data source.

IV. Conclusion

A key aspect of ROCI's design has always been the reduction of redundancy. Sometimes referred to as the DRY (Don't Repeat Yourself) principle in programming, ROCI's original design applied this methodology to the classification of translation components by virtue of their input and output types. The notion that data structures are related to the functionality of the components implies that the programmer encodes some descriptive content of the functional code in the process of data structure definition. This content can be extracted by a reflective type system and used to automatically classify components without requiring any additional effort on the part of the programmer. It became clear that manually-created component descriptions, be they structured or not, restate at least some of what is already stated in code. The extensive use of type information throughout ROCI, and many components designed for it, have allowed ROCI developers to extract more value from their code than is generally expected. The addition of scripting services built upon the same type definitions, and the robust accessibility offered via the web interface and Pin Exporter mechanism, further increase this value. These simplifying technologies combine to make possible a new level of focus on the user.

User experience is not just a marketing term; it is instead an important indicator of how well any user, be they human or computer, can make use of a service. A good experience indicates useful software that is more likely to be used. Thus effort spent determining how best to serve each end user is repaid by the contributions of those users. It is this focus on helping each node on the network do what it does best that makes the network itself far more effective. The emphasis on interoperation and communication permits us to quickly experiment with and integrate novel methods of user-robot interactions without spending time porting code or training users. Like the Web itself, robotics can be a powerful collaborative technology when designed with usability and interoperability in mind.

References

[1] R. Vaughan, B. Gerkey, and A. Howard, "On Device Abstractions For Portable, Reusable Robot Code," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robot Systems*, 2003, pp. 2121–2427.

[2] B. Gerkey, R. Vaughan, K. Stoy, A. Howard, G. Sukhatme, and M. Mataric, "Most Valuable Player: A Robot Device Server for Distributed Control," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2001, pp. 1226–1231.

[3] N. Nesnas, A. Wright, M. Bajracharya, R. Simmons, T. Estlin, and W. Kim, "CLARAty: An Architecture for Reusable Robotic Software," in *Proceedings of SPIE Aerosense Conference*, 2003.

[4] M. Montemerlo, N. Roy, and S. Thrun, "Perspectives on Standardization in Mobile Robot Programming: The Carnegie Mellon Navigation (CARMEN) Toolkit," in *Proceedings of the 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2003, pp. 2436–2441.

[5] A. Cowley, H. Hsu, and C. Taylor, "Modular Programming Techniques for Distributed Computing Tasks," in *Proceedings of the 2004 Performance Metrics for Intelligent Systems (PerMIS) Workshop*, 2004.

[6] ——, "Software Design for Distributed Sensing and Computing Tasks," in *Proceedings of SPIE Vol. 5609 Mobile Robots XVII*, 2004, pp. 135–144.

[7] L. Chaimowicz, et al, "Deploying Air-Ground Multi-Robot Teams in Urban Environments," in *Proceedings of the 2005 International Workshop on Multi-Robot Systems*, 2005.

[8] B. Satterfield, S. Jameson, H. Choxi, and J. Franke, "A Role-Based Approach to Unmanned Team Operations," in *Association for Unmanned Vehicle Systems International conference*, Baltimore, MD, June 2005.

[9] R. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[10] T. Berners-Lee. Principle of Least Power. [Online]. Available: http://www.w3.org/DesignIssues/Principles.html#PLP