by Ben Axelrod

THE NEXT BIG THING!

Service Oriented Architectures

Two leading systems, MRDS and ROS, point to the future of robotics

A revolution is coming, and it is time to choose sides. Microsoft + .NET or Linux + Python. Microsoft Robotics Developer Studio (MRDS) and Robot Operating System (ROS) are the major contenders for service-based robotics libraries. The lines have been drawn, and whichever side you choose, you should be prepared for a new paradigm: distributed computing.

MRDS and ROS are two new robotics software architectures that have sprung up in the past few years. At first glance, they could not be more different. ROS is strongly rooted in the open-source movement, (ROS also stands for Robot Open Source), and only runs on Unix-based platforms. MRDS is, well, Microsoft, and as you would expect only runs on Windows. However, once you get past these differences they are actually quite similar. They are both "Service Oriented Architectures," or SOAs for short. This architecture looks to be the next big trend in robotics.

Why are SOAs the next big thing? For a number of reasons. As CPU speeds are starting to level off, computer engineers are finding that the best way to give computers more horsepower is to up the number of cores. This means more

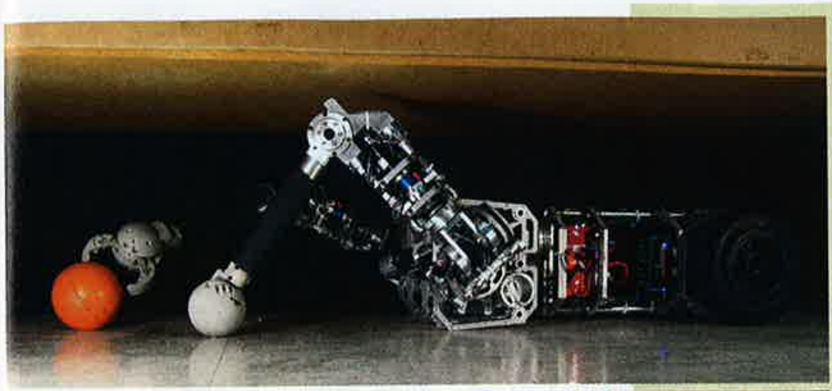
PHOTOS COURTESY OF WILLOW GARAGE, ROD GRUPEN AND COROWARE

threads av
run in par
this power
this direct
The other
More and
computer
Internet a
different
designed

PARALI

Interesting
robotics. L
massively
data at the
"machine.
our menta
lelism as v
ly as usefu

The ot
applies to
said that
of connec
tion, a ro
many laye
the lowes
sensor an
own mic
highest le
mapping
ancing, li
cessing, a
can all tal
to warran
To come
gy, it is
brain ha
these diff
mention
nervous
lower-lev
even som
Again, if
built wit
working
our robot



threads available which in turn means more code can run in parallel... but only if you know how to harness this power. (Even simple microcontrollers are moving in this direction. Take for example the Parallax Propeller). The other reason is due to the ubiquity of the Internet. More and more, what used to be handled on a single computer or mainframe, is now handled through the Internet across multiple machines. This means lots of different systems talking to each other. SOAs are designed for this purpose.

PARALLELISM AND THE BIOLOGICAL MODEL

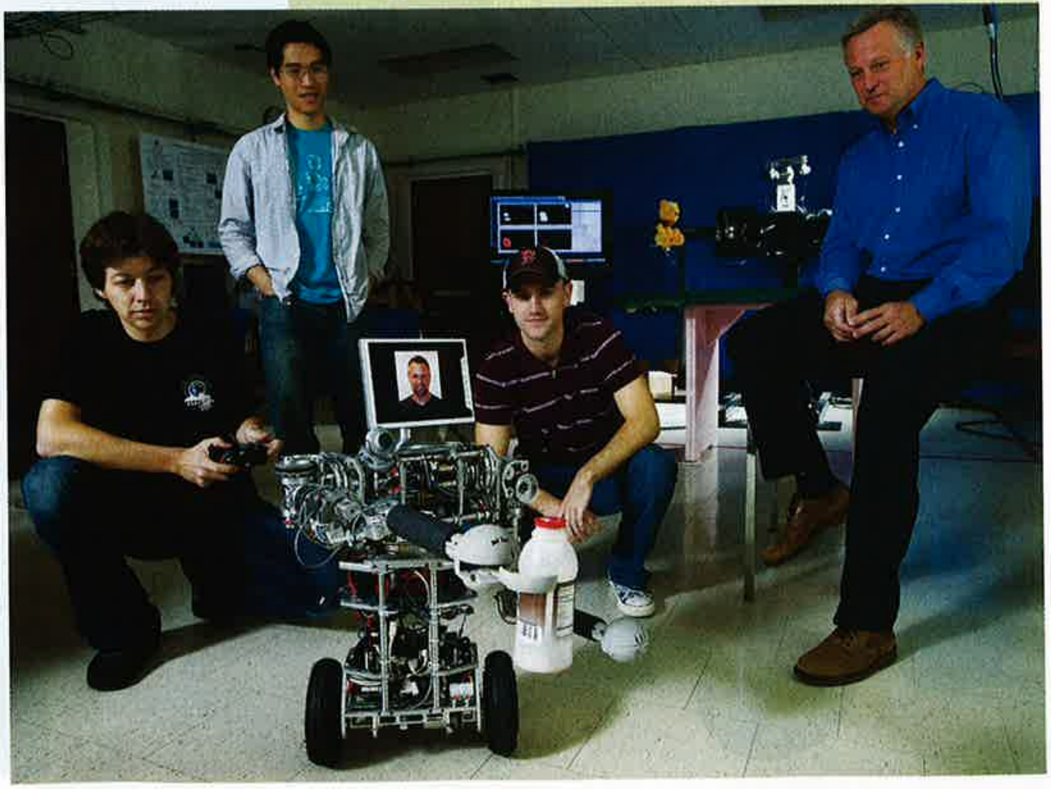
Interestingly, the same factors that spawned SOAs also apply to robotics. Let's first take a look at parallelism. The human brain is massively parallel. The brain's billions of neurons all process data at the same time, giving rise to the planet's most powerful "machine." If we ever want our robots to have even a fraction of our mental capabilities, we will need to take advantage of parallelism as well. It turns out that a single fast computer is not nearly as useful as many, slower computers working in parallel.

The other factor, inter-machine communication, definitely applies to robotics. It has been said that robotics is the science of connectors. Almost by definition, a robot is going to have many layers of computation. On the lowest level, just about every sensor and actuator now has its own microcontroller. On the highest level, vision processing, mapping and navigation, balancing, limb control, speech processing, and behavior selection can all take up enough resources to warrant dedicated processors. To come back to the brain analogy, it is well known that the brain has separate regions for these different functions. Not to mention our spinal cord and nervous system which handle lower-level reactive control and even some simple walking tasks. Again, if biological systems are built with modular controllers working in parallel, shouldn't our robots?

UBOT-5 ASSISTIVE ROBOT

The uBot-5 is the latest generation of an assistive robot that has been in development for some years at the University of Massachusetts Amherst. Computer scientist Rod Grupen, director of UMass Amherst's Laboratory for Perceptual Robotics, www-robotics.cs.umass.edu/Robots, has led the project. Current research by Hee-Tae Jung includes telemedicine for use in stroke rehab and therapy for autism. Scott R. Kuindersma is using the uBot-5 to research "whole body" modeling and control.

The uBot-5 is an assistive robot for aging citizens. It is designed to help with a variety of minor households tasks and administer medical instrumentation. It also allows the client to easily communicate with service providers and loved ones. It can reach objects and pick them up in the same manner as a human, and can retrieve items from confined spaces. The robot's services run on an on-board Pentium-based PC running Windows XP. The uBot-5 teleoperator controls have been ported over to Microsoft Robotics Studio. The uBot-6 will be introduced this summer.



MICROSOFT ROBOTICS DEVELOPER STUDIO

The first contender, MRDS, first released in December 2006, is a product of Microsoft Research. It is closed-source, but completely free as of version 2008 R3. MRDS is built on top of Microsoft's .NET Framework. This means that you have a wide choice of programming languages, although MRDS only supports C#, VB.Net, C++/CLI, and IronPython. It also means that you have the backing of a huge and powerful library with lots of helpful subroutines.

MRDS consists of two core libraries: Concurrency and Coordination Runtime (CCR), and Decentralized System Services (DSS). CCR is a lightweight messaging library which lets you spawn, iterate, and join threads quite easily. DSS is a state-oriented service model that leverages CCR and gives you higher-level functionality like web browser integration. All MRDS services can be inspected and configured through a web browser, which reduces a lot of operations to a simple point-and-click. However, users will still need to be familiar with DOS command line tools, as not everything can be done through the browser.

In addition to CCR and DSS, MRDS includes a Visual Programming Language (VPL) and a wrapper for the NVidia PhysX 3D physics simulator. While VPL is targeted to non-programmers, CCR and DSS are robust enough for enterprise level work. MRDS has a multitude of corporate partnerships. One application of note is balancing the load on the MySpace servers.

ROBOT OPERATING SYSTEM

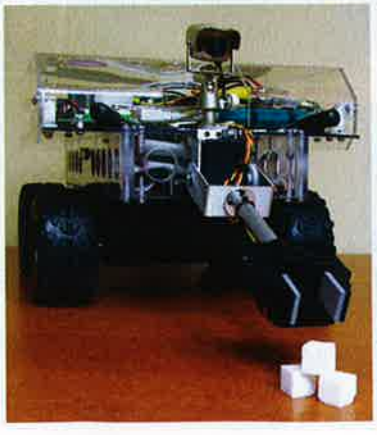
In the second corner, we have ROS which is the primary software platform of Willow Garage. It celebrated its 1.0 release in January, 2010, and first stable distribution "Box Turtle" in March. ROS is completely open source, and was jointly developed by Willow and Stanford University. ROS calls itself a "meta-operating system for your robot" because it gives you standard OS-like tools for working with ROS on your robot. ROS is only supported on Unix-like operating systems: Linux, OS-X, and Cygwin on Windows. ROS is written in both Python and C++ which provides a nice mix of power and portability. Additionally, message types and other configuration files are specified in simple plain-text files which get parsed into code by the client you use. Currently, C++ and Python clients are supported, with additional Octave, Lisp, and Java clients having experimental status.

Being partially developed at Stanford, it is easy to understand that ROS has many ties to the robotics academic world. Currently, 22 universities maintain ROS software stacks. Additionally, the ROS development team significantly overlaps with the development teams for OpenCV and Player. This ensures that ROS has seamless integration with these common academic robotics libraries. It also means that ROS developers can leverage lots of preexisting robust code. While ROS has a wide variety of packages, a majority were created for mobile manipulation research. Mobile manipulation is one of the topics at the forefront of robotics academic research. This is what Willow Garage's PR2 robot was designed for.

Similar to MRDS, ROS has a variety of command line tools to create new packages and get info on them. However, ROS takes this to a new level. Its multitude of tools provides a solid infrastructure to manage your system and keep third party packages working nicely with each other. ROS's main interface is the terminal. Linux

COROWARE ROBOTS

Coroware Corobots have been using Microsoft Robotics Developers Studio for years. Corobots are used for research at a variety of colleges and universities, including Vassar College, and come in a variety of configurations suited to both indoor and rough outdoor environments. Check out Coroware's website to see more, www.coroware.com.



WILLOW GARAGE PR2

Willow Garage's PR2 robot uses ROS running in a Linux operating system. It is a purpose designed research platform that is being given away to approximately 10 research centers at no cost to spur robotics research. Films of the PR2 autonomously folding towels can be found at the Willow Garage website, www.willowgarage.com. PR2 is intended to do useful work of all kinds in human environments, and its designers hope that developers will take it far as a groundbreaking platform.

users will feel at home using "roscd" in place of "cd", and "rosls" in place of "ls", just to name a few. Additionally, these tools support the "pipe operator," which allows you to funnel the output of one tool into another to create powerful command sequences.

TERMINOLOGY DIFFERENCES

It should be noted that the terminology between these two libraries differs significantly. MRDS uses a web-services nomenclature, whereas ROS uses a distributed systems nomenclature. At times, the naming conflicts can be quite confusing. For example, a node in ROS is called service in MRDS, and a service in ROS is just a pair of messages in MRDS. In this article, I use terminology that is more closely aligned with MRDS. However, if you understand the high-level concepts outlined here, you should have an easier time learning either system.

Once you have chosen your poison, you will find that writing code for either of these packages is a little bit different from other types of programming you may have done – especially if you are not familiar with distributed systems or service architectures. Despite differences in naming, implementation, and general coding philosophies, both MRDS and ROS follow a pretty similar design pattern. Understanding SOAs is a big first step to learning and working with these libraries.

SERVICE ORIENTED ARCHITECTURE

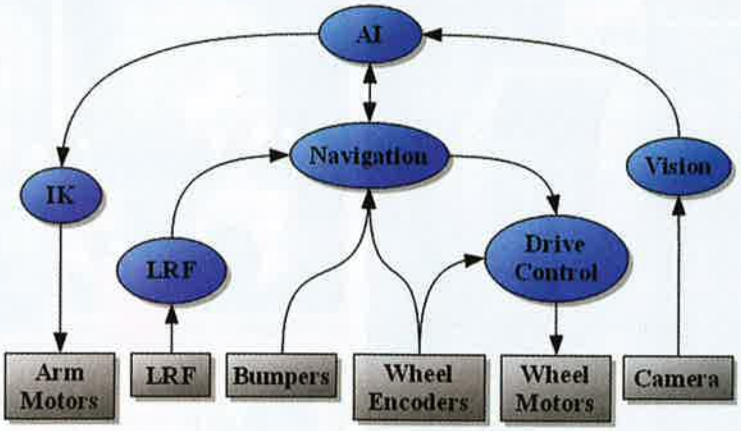
First, a little history. SOAs are not new. Web services have been around for about a decade now. Typically, when one talks about SOAs, they specifically mean web services which are a collection of standards and technologies such as XML, SOAP, and WSDL. Here, I use the term SOA to mean the general architecture of your code. In a SOA, each part of your application runs in an isolated service, and they talk to each other through messages. There are two basic messaging schemes: request/reply, and publish/subscribe.

Request/reply can be thought of as a "pull" scheme, where a service always has to pull requested data out of another service. This is the most basic type of messaging. In general the "request" message need not actually be a request for data; of course it can send data or simply be an indication to another service. The bottom line with request/reply messages is that if you want data repeatedly, you have to ask for it each time.

The other type of messaging, publish/subscribe, can be thought of as a "push" scheme. In this pattern, one service spews messages continuously, without knowing or caring if any other service is listening. It is "pushing" the data. Then, one or many other services can subscribe to this data stream. This pattern is analogous to a "callback" or "interrupt" because the service that is subscribed to the data stream will get notified as soon as there is a message. This pattern is particularly useful for low-level sensors that generate new readings at a constant rate, and you may have many services that want to subscribe to and inspect this data.

INFORMATION FLOW

Here is an example of a highly simplified SOA for a generic mobile robot with an arm. At the bottom of the diagram, the gray rectangles represent the actual hardware of the robot. The blue ellipses above represent services. The arrows represent the flow of data through the system. As you can see, it is arranged somewhat hierarchically, with "low-level" services closer to the hardware, and "high-level" services separated from the hardware. The higher up the service, the "brainier" it is. Constructing your system in a hierarchical manner like this forces you to maintain good design patterns. It also helps with debugging because you will have a clear understanding of the dependencies between services.



SOA for a generic mobile robot with an arm.

The downward pointing arrows are typically request/reply type messages. This is because the higher level services are directly controlling the lower ones. The data in each of these messages is usually some sort of control command. For example, the artificial intelligence (AI) service tells the inverse kinematics (IK) service where to move the arm, or the navigation service where to drive the robot. Following the path of the downward arrows will lead you to the robot's hardware outputs, the motors.

Similarly, following the upward arrows will take you from the robot's hardware sensors, the laser range finder (LRF), bumpers, encoders, and camera, up to the brains of the robot. These are mostly publish/subscribe type messages, because the sensors generate data as fast as they can and simply broadcast this information. You can see how both the navigation and drive-control services subscribe to the wheel encoder data. Both MRDS and ROS are designed to handle this type of broadcast efficiently.

As alluded to above, a service may not necessarily know or care about the other services running, or even interacting with it. This loose coupling is possible due to the message passing interface. When services talk to each other they only see this interface. If you set up your message definitions properly, you can turn each interservice interface into an abstraction layer. This is where the real power of SOAs comes into play. These abstraction layers allow for easier code re-use and exchange. In our simple example from above, the LRF service implements a standard generic set of messages which provide laser scan data. So if you change your hardware to use a SICK LRF instead of a Hokuyo LRF for example, you would only need to change the LRF service. The navigation service would not even know there was a change in your system. The new SICK LRF service you use might not even be written by you. All this can be done without changing a single line of source code.

This service isolation provides robustness to your system. If one service goes down, all other services will not necessary go down with it. Of course data may back up, and things can still break; but at least there is an opportunity to recognize the fault and restart the service if possible. This isolation also means that you can scale your system quite easily. For example, if you find that the vision service is taking up too much processing power and dragging the entire system down, you can simply put that one service on a separate computer. Because services talk over sockets, they don't care if they are on the same computer, or across the globe over the Internet. In general, moving a service onto another computer, does not require any code changes. It usually only requires modification of some configuration and start-up files so that all services know how to find each other.

CODE COMPLEXITY

Of course there is a cost to pay for the flexibility and power of a service architecture, and that is complexity. Your code is no longer a single string of instructions. Gone are the days of simply setting a breakpoint and stepping through your code. Now, your code is spread over multiple applications and multiple threads, each sending and receiving asynchronous messages. Needless to say following the flow of execution of your code can be tough. On top of the normal coding bugs, there are a host of new and much more onerous issues such as deadlocks, starvation, port/firewall issues, message type mismatches, and improperly implemented services. Both MRDS and ROS have tools for logging debug messages and examining message histories, but in large systems, even these tools can break down.

CONCLUSION

Make no mistake; both the MRDS and ROS libraries have very steep learning curves. But I believe climbing that mountain is well worth it. For users wishing to learn, my suggestion is to read all the documentation you can get your hands on and do the tutorials. They are there for a reason. Thankfully, both libraries have large active communities that you can take full advantage of. MRDS has extensive forums, and ROS has an email list with lightning-fast response times.

SOAs are an important and powerful paradigm for robotics today. If you want to be a part of the modern robotics community, a solid understanding of SOAs and distributed systems is becoming a requirement. The popularity of two new libraries: MRDS and ROS is an example of this. So, which side are you on?

Editors note: Ben Axelrod is a Research Scientist for iRobot Corporation. He currently lives in Burlington, Massachusetts with his wife and son. He has previously worked for Coroware, Inc., Microsoft and Iguana Robotics, Inc. At Microsoft he helped launch the first preview release of MRDS. ◉

Links
CoroWare Inc., www.coroware.com, (800) 641-CORO

Laboratory for Perceptual Robotics, www-robotics.cs.umass.edu/Robots, (413) 577-0618

Microsoft Robotics Developer Studio (MRDS), www.microsoft.com/robotics

Robot Operating System (ROS), www.ros.org

Willow Garage, www.willowgarage.com, (650) 475-2700

For more information, please see our source guide on page 89.