Middleware for Robots?

Christopher D. Gill and **William D. Smart**
Department of Computer Science
Washington University in St. Louis
One Brookings Drive
St. Louis, MO 63130
United States
{cdgill,wds}@cs.wustl.edu

Abstract

Robotic systems, particularly those involving multiple robots and sensors, are increasingly distributed and heterogeneous. We ask the question whether middleware can be used to build these systems, to alleviate unnecessary complexities of using low-level tools while still preserving necessary levels of system performance. We motivate the answer to that question with preliminary performance studies of a particular open-source middleware implementation, and pose additional open questions to further explore the suitability of middleware in building distributed robotics systems.

Introduction

Robotics systems inherently require significant interaction and coordination of diverse hardware and software elements. Specialized hardware and software may be appropriate at low levels, such as within local sensor busses, or actuator systems. However, the aggregation of such systems, even within a single robot, requires careful consideration of architectural issues to promote flexibility, performance and reuse of both hardware and software. These issues are especially important when we consider use that spans families of robots and embedded systems, types of applications, models of programming, and different kinds of operating environments.

One of the most important problems that must be addressed for a distributed robotics or embedded systems application is the user programming model. How much of the distributed nature of the system should the user have to deal with explicitly, and how much should be hidden by abstractions? The answer to this question is likely to vary from system to system, depending on the user and the application. Some users will not care that their system is distributed, while others will want fine-grained control over certain aspects of the distribution mechanism. A major challenge in designing an architecture for distributed applications of this sort is to provide a programming model that allows both views to work together seamlessly.

In this paper, we discuss the use of middleware for distributed robot control systems. We begin by describing what exactly we mean by "middleware", and discuss its use in the context of distributed robotics and embedded sensor systems. We then present some preliminary experimental results that attempt to quantify bounds on the costs of using such middleware, compared to a simpler information-distribution mechanism. Finally, we conclude by posing some open questions on the use of middleware for distributed robotics applications.

Middleware for Robots

What is Middleware?

Bakken (2001) describes middleware as

> "...a class of software technologies designed to help manage the complexity and heterogeneity inherent in distributed systems. It is defined as a layer of software above the operating system but below the application program that provides a common programming abstraction across a distributed system ... In doing so, it provides a higher-level building block for programmers than Application Programming Interfaces (APIs) such as sockets that are provided by the operating system. This significantly reduces the burden on application programmers by relieving them of this kind of tedious and error-prone programming. Middleware is also informally called "plumbing" because it connects parts of a distributed system with data pipes and then passes data between them."

Middleware resides above the operating system and network layers and provides a consistent distributed programming model to the application developer. Examples of well-known middleware architectures include:

RPC Sun's original client-server procedural programming model.

DCOM Microsoft's distributed component model for Windows platforms.

Java RMI The Java Community's Remote Method Invocation model for the Java language.

CORBA The Object Management Group's (OMGs) standard for language and platform independent distributed object computing.

Why use Middleware?

Middleware is useful for a number of reasons when designing large, distributed system.

Portability Middleware offers a common programming model across language and/or platform boundaries, as well as across distributed end systems. Intel-based computers running Microsoft Windows, programmed in Java can easily communicate with Motorola-based machines running Linux, programmed in C++. Most importantly, this ease of communication happens seamlessly, without the application programmer ever having to worry about it.

Reliability If we have a tried and tested set of middleware, it can be reused and optimized with confidence over many applications. It allows the application programmer to never have to worry about any of the low-level aspects of distributed systems development. The same can also be said for having a well-tested socket library that is reused in many applications, although such a library hides fewer of the low-level details of the communication from the programmer.

Managing complexity Low-level programming abstractions, such as sockets and threads, can be made more accessible through suitable (possibly object-oriented) libraries. However, programming *combinations* of these abstractions can be excessively tedious and error prone. Programming within the context of pattern-aware (Schmidt *et al.* 2000) middleware can drastically reduce both chance of introducing errors into the code, and the amount of pain that the programmer must endure when implementing the system.

Why use CORBA?

In this paper, we argue for using CORBA middleware for distributed robotics and embedded sensor applications. This choice is motivated by two main factors, open standards and platform/language independence.

The CORBA standard (Object Management Group 2001b) itself is both open to extension and use, and closed to modification without full review under the OMG standards process. This is good because it means that the standard is relatively stable (between major revisions), should remain general-purpose (since the OMG has members with diverse application interests), but is still subject to change as the technologies involved mature.

CORBA is also platform and language independent. We feel that this is especially important for applications in distributed robotics and embedded systems. Many researchers use C or C++ to program their robots navigation systems, but feel more comfortable implementing their high-level reasoning systems in Lisp. We would like to be able to develop code on our desktops, then deploy in on an embedded processor with the minimum of alterations. The platform and language independence of CORBA makes this much easier.

Robot-Specific Middleware

Now that we have identified what we actually mean by middleware, and motivated our specific interest in CORBA middleware, we go on to look at some of the issues that are relevant to using middleware for robotics and embedded sensor applications.

Data Propagation and Fusion Sensor data are responsible for the vast majority of communications traffic in a distributed robot or embedded sensor system. These data can range from fine-grained readings from a single sensor element to the results of a complex reasoning process. These data can also vary in size from a few bytes in size up to many millions of bytes, and in granularity of transmission from single events to continuous streams. These variations naturally induce different levels of overhead. We look at this issue in some more detail, and give some preliminary experimental results below.

Concurrency, Coordination and Control Many forms of concurrency, coordination and control are useful and often necessary in a distributed system. For example, consider a teleoperation task, where the teleoperated robot is particularly simple, and does not carry many computational resources. Much of the sensor interpretation and reasoning is done in a remote machine, or by a human operator. The controlling system might need to (1) monitor the remote system's bumpers in order to make local navigation decisions, (2) activate a certain high-cost sensor occasionally, and retrieve its readings and (3) record a stream of profiling data summarizing the current state of the remote robot. In the first task, sensitive response to environmental changes would likely need to be handled in an asynchronous manner, which may suggest a fine-grained asynchronous event (O'Ryan & Schmidt 1999) push model, from the remote robot to the local system. In the second case, an asynchronous messaging (Obj 1998) model would support multiple concurrent queries from the local end-system to the remote robot. Finally, the third case might best be handled by a standard distributed object computing (DOC) method invocation from the robot to the local system, to ensure the profile was delivered intact and in sequence.

All of these approaches must be supported seamlessly and with the minimum cost to the programmer in maintenance and feature complexity. We believe middleware can meet this challenge, offering a consistent and reasonable programming environment for distributed robotic systems.

Quality of Service The Quality of Service (QoS) requirements of a robotic system will vary according to the current task, the operating environment and the system loads. For example, timeliness constraints for a slow-moving robot mapping a static office environment might be quite loose, with few severe consequences if they are missed. However, the constraints for a fast-moving autonomous vehicle, traveling over rough terrain are likely to be very tight, with potentially disastrous results if deadlines are missed. We consider some of the QoS implications of middleware in this paper. Furthermore, previous research on real-time performance (O'Ryan *et al.* 2001) and scheduling (Gill, Levine, & Schmidt 2001; Loyall *et al.* 2001; Gill, Cytron, & Schmidt 2002) demonstrates the suitability of middleware for enforcing distributed QoS properties.

Scalable Programming Models System developers require programming models that are scalable. An example of such scalability lies in the complexity of the programming abstractions that are used. A programming model that covers many low-level details may become tedious and error-prone to program unless those details are appropriately encapsulated and can be configured at higher levels of abstraction. For example, a robot might be able to provide raw sensor readings, calibration settings and timing information from its laser range-finder, and allow the various settings to be manipulated by the programmer. However, it should also be able to return a "distance to object" reading, using sensible defaults, for the user who is not interested in the exact settings of the device. This means (at least) two views of the same device, as a generic distance sensor, and as an actual laser range-finder. A good programming model should accommodate both views and allow them to inter-operate seamlessly. It should also be able to scale across sensors. For example, the high-level user should be able to ask for the distances returned by all sensors on the robot, regardless of type, and get reasonable results.[1]

Scalability also deals with how easy it is, from a software point of view, to add another robot or embedded sensor to an existing system. Ideally, there should be little cost in terms of programming time and integration work. These considerations motivate our interest in CORBA middleware. The CORBA programming model allows programmers to extend existing interfaces polymorphically, replace instances of objects, and add new objects to an existing system. Furthermore, CORBA supports implementations that address the performance and footprint issues of scalability, which is the subject of the next section.

What About Speed and Bloat?

One of the most-cited reasons for not adopting middleware-based architectures on robotics platforms is that the additional layers of abstraction will slow things down and cause code bloat. In this section, we address these two issues and provide the results of some empirical experiments to support our claim that middleware offers a net benefit to developers of distributed robotic applications.

Speed Results

To assess the relative time cost of using middleware for data transmission, compared to simple socket data transfer, we conducted two experiments to benchmark the performance of each approach. These benchmarks are intended not as a definitive answer to the question of how much a given system may pay for middleware's flexibility, but rather to establish reasonable upper and lower bounds on those expectations.

Figure 1 shows the average results of ten trials of sending one million pairs of double precision floating point numbers using TAO (Center for Distributed Object Computing Washington University), a high-performance and real-time open-source CORBA-compliant ORB implemented in C++.

[1]Whether or not this is inherently a bad idea to begin with is outside of the scope of this paper.

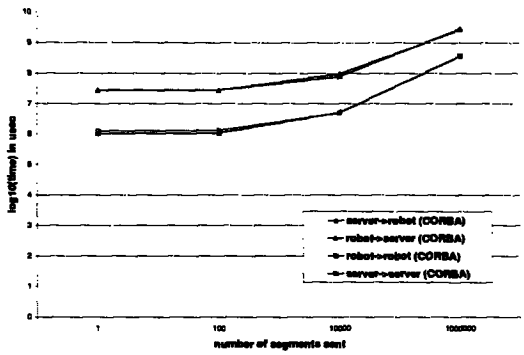

Figure 1: CORBA Data Transmission Benchmark Results

We ran each set of trials locally on a server machine, locally on the robot, from the server to the robot, and from the robot to the server. The robot computer is an Intel Pentium-III 800Mz based system, with 128MB of RAM, running a 2.2.19 linux kernel. The server is a dual processor Pentium-III 1GHz system, with 1GB of RAM, running a 2.4.8 linux kernel. The machines are on different subnets, and the communication bottleneck between them is a standard 802.11 wireless Ethernet link, running at 11Mbits/second.

In each trial we varied the granularity of the transmission by factors of 100, first sending a single buffer, then 100 equal subdivisions of the data, then 10,000 subdivisions, and finally 1,000,000. We note that in the first two runs, of 1 and 100 buffer subdivisions, little overhead was added by the CORBA per-call setup itself. The cost of the transmission includes parameter marshaling, so these represent a baseline for assessing a lower bound on total achievable throughput (several optimizations such as removing endian marshaling where possible could improve on this result for specific cases). We also note that as the per-call overhead increases, the contribution of network overhead relative to the total overhead diminishes, resulting in a convergence of the local and distributed transmission curves.

We contrast these results with an implementation using sockets, that helps us identify an upper bound on the expected throughput in each case. O'Ryan, *et al.*, have made careful studies of the contribution of each infrastructure component in TAO to the overall middleware transmission overhead (O'Ryan *et al.* 2000). Together with our socket results, these profiles help calibrate expectation for the achievable performance of middleware for distributed robotics. The timings for the socket experiments experiments are shown in Figure 2. It should be noted that no marshaling or UN Marshalling of the data is taking place in the socket implementation. This will cause the amount of computation being done to be inherently less than in the CORBA implementation, since the Intel architecture does not natively use network byte order (meaning that all data are converted upon transmission and reception). It does, however, give us a lower bound on the amount of time needed to ship data

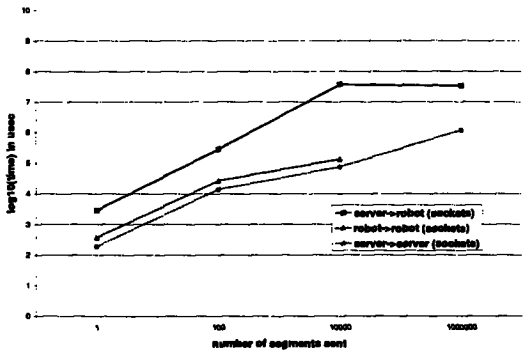

Figure 2: Socket Data Transmission Benchmark Results

between machines.

Bloat

One argument against middleware is that it adds unnecessary bloat to a system. While full-featured middleware implementations such as CORBA-compliant ORBs may have significant memory footprints, subsets of such middleware can offer many of the benefits without the excessive overhead. Current research is exploring various ways to ensure only the necessary features of middleware are provisioned, and thus reduce the overall footprint on a case-by-case basis.

The key insight is that the footprint of middleware is only appropriate or inappropriate with respect to a *particular* allocation of resources. Resources tend to be aggregated into hierarchies in distributed systems, with resource-rich backbones connecting islands of adequate but diminished resources, which may in turn connect resource-attenuated leaves of the overall sensor/robot network. Three main areas of research are addressing these issues at the leaves of provisioning hierarchy:

- RMI/JINI (Sun Microsystems 1999) - this approach allows a reduced-feature protocol bridge to inter-operate with a full-featured Java middleware infrastructure. Specifically, Jini allows devices with resource constraints too stringent to support RMI, to interact with endsystems that can support RMI. Thus, combining RMI with Jini allows devices at very small resource scales to be used within a middleware context.
- CORBA/Smart Transducers (Object Management Group 2001a) - this approach takes a similar approach to RMI/Jini, but removes language specificity. In particular, CORBA supports a variety of language and platform choices. The Smart Transducers specification allows individual devices to be addressed as though they were on endsystems supporting a complete CORBA ORB. Thus, the CORBA/Smart Transducers approach offers additional transparency to programmers of these kinds of systems.
- CORBA/NEST (Subramonian, Gill, & Sharp 2001; Subramonian & Gill 2002) - this approach addresses reduction of the middleware infrastructure itself, from a perspective of *composing* only the needed features into a customized and inter-operable version of the full-featured middleware. We believe this approach is the most suitable to systems such as distributed robotic and sensor systems in which different QoS properties such as timeliness and footprint must be interchanged with features required by each application. By provisioning exactly the policies and mechanisms needed, and more importantly supporting integration at the point of composition, we believe sophisticated systems design trade-offs can be achieved without an undue burden of complexity on the system developers.

Conclusions: Applicability of Middleware

Although we strongly believe that CORBA-based middleware offers great advantages for robotics and embedded sensor applications, we do not claim that it is a good solution for *all* such systems. The applicability of a middleware solution depends heavily on the hardware of the system on which it must be deployed, and the applications that it will be running.

Many robotics research platforms are based on common off-the-shelf (COTS) hardware, with computational and networking abilities comparable to a desktop computer workstation, and run a "standard" operating system that controls most of their operation. We believe that it is this class of system that will most benefit from the application of middleware. Arguments can be made that certain QoS needs cannot be met by such a COTS solution, and that specialized hardware, operating systems and communications infrastructure must be used. However, there is some evidence that a relatively cheap COTS system can meet relatively stringent real-time QoS demands. Earlier experiments (Levine *et al.* 1999) on several general-purpose and real-time operating systems indicated that under certain conditions general-purpose operating systems can perform in some areas as well as, or in some cases better, than some RTOSs. For example, the results of these experiments showed that thread context switching overhead had lower latency and jitter in a COTS open-source Linux kernel than in all but one RTOS, as long as the number of threads per CPU was small.

Open Questions

Rather than drawing conclusions from the preliminary experiments presented in this paper, we will instead end by asking some questions. We consider these questions to be important ones that should be addressed in any further research into the use of middleware on distributed robotic and embedded sensor systems.

- What are the real costs of deploying CORBA-based solutions on standard robotic platforms, such as the RWI type of robot? Furthermore, are there configurations of system features that increase or decrease these costs?
- What are the relevant round-trip latency and data throughput transmission profiles for *realistic* data transfer patterns in common robotics use-cases?

- Can user studies identify how middleware helps or hinders ease of programming? Furthermore, can these studies measure the perceived performance impact of middleware on robot usability, from the user's perspective?
- What are the abstractions that are *useful* to people building and using robots? Which abstractions hold over different communities of use (for example ML researchers, path-planning researchers, military, etc.)? Which abstractions extend or specialize the more general abstractions for particular domains?
- Can CORBA-based approaches be combined with simpler strategies (such as sockets), so that the strengths of one can make up for the weaknesses of another?
- How does the minimum transmission (MTU) size on a network affect the tradeoffs between sockets and CORBA? Between small, frequent packets and larger, less frequent ones?

References

Bakken, D. 2001. Middleware. In Urban, J., and Dasgupta, P., eds., *Encyclopedia of Distributed Computing*. Kluwer. to appear.

Center for Distributed Object Computing. Washington University. TAO: A High-performance, Real-time Object Request Broker (ORB). www.cs.wustl.edu/~schmidt/TAO.html.

Gill, C. D.; Cytron, R.; and Schmidt, D. C. 2002. Middleware Scheduling Optimization Techniques for Distributed Real-Time and Embedded Systems. In *Proceedings of the 7^{th} Workshop on Object-oriented Real-time Dependable Systems*. San Diego, CA: IEEE.

Gill, C. D.; Levine, D. L.; and Schmidt, D. C. 2001. The Design and Performance of a Real-Time CORBA Scheduling Service. *Real-Time Systems, The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware* 20(2).

Levine, D. L.; Flores-Gaitan, S.; Gill, C. D.; and Schmidt, D. C. 1999. Measuring OS Support for Real-time CORBA ORBs. In *Proceedings of the 4^{th} Workshop on Object-oriented Real-time Dependable Systems*. Santa Barbara, CA: IEEE.

Loyall, J.; Gossett, J.; Gill, C.; Schantz, R.; Zinky, J.; Pal, P.; Shapiro, R.; Rodrigues, C.; Atighetchi, M.; and Karr, D. 2001. Comparing and Contrasting Adaptive Middleware Support in Wide-Area and Embedded Distributed Object Applications. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21)*, 625–634. IEEE.

Object Management Group. 1998. *CORBA Messaging Specification*, OMG Document orbos/98-05-05 edition.

Object Management Group. 2001a. *Smart Transducers Interface*. Object Management Group, OMG Document orbos/2001-06-03 edition.

Object Management Group. 2001b. *The Common Object Request Broker: Architecture and Specification*, 2.5 edition.

O'Ryan, C., and Schmidt, D. C. 1999. Applying a Real-time CORBA Event Service to Large-scale Distributed Interactive Simulation. In *5^{th} International Workshop on Object-oriented Real-Time Dependable Systems*. Monterey, CA: IEEE.

O'Ryan, C.; Kuhns, F.; Schmidt, D. C.; and Parsons, J. 2000. Applying Patterns to Develop a Pluggable Protocols Framework for ORB Middleware. In Rising, L., ed., *Design Patterns in Communications*. Cambridge University Press.

O'Ryan, C.; Schmidt, D. C.; Kuhns, F.; Spivak, M.; Parsons, J.; Pyarali, I.; and Levine, D. L. 2001. Evaluating Policies and Mechanisms to Support Distributed Real-Time Applications with CORBA. *Concurrency and Computing: Practice and Experience* 13(2):507–541.

Schmidt, D. C.; Stal, M.; Rohnert, H.; and Buschmann, F. 2000. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. New York: Wiley & Sons.

Subramonian, V., and Gill, C. 2002. OMG Workshop On Embedded & Real-Time Distributed Object Systems. In *Experiences with Middleware for a Networked Embedded Software Technology Open Experimental Platform*. Object Management Group.

Subramonian, V.; Gill, C.; and Sharp, D. 2001. OOPSLA 2001 Workshop Towards Patterns and Pattern Languages for OO Distributed Real-time and Embedded Systems. In *Towards a Pattern Language for Networked Embedded Software Technology Middleware*. ACM.

Sun Microsystems. 1999. Jini Connection Technology. www.sun.com/jini/index.html.