

# Is a Common Middleware for Robotics Possible?

William D. Smart

Department of Computer Science and Engineering

Washington University in St. Louis

St. Louis, MO 63130

United States

wds@cse.wustl.edu

**Abstract**—Despite many decades of work in robotics, we are still lacking a well-accepted common software architecture, middleware, and shared low-level functionality. Most research groups still write their own systems, and any software developed by these groups tends to be strongly tied to their architecture, their middleware, and their robots. This makes the sharing of software and, by extension, well-implemented algorithms, extremely difficult, if not impossible. This lack of sharing is seriously impeding robotics research, as we spend our time re-implementing known algorithms and techniques, rather than discovering new ones. In this paper, we outline some of the key problems that stand in the way of developing a widely-accepted set of middleware for robotics. We discuss why the development of this middleware is vital to the long-term success of our field, and offer some suggestions on how to get started.

## I. INTRODUCTION

Despite many decades of work in robotics, we are still lacking a well-accepted common software architecture, middleware, and shared low-level functionality. Most research groups still write their own systems, albeit based on standards from the software engineering community such as CORBA [1]. Any software developed by these groups tends to be strongly tied to their architecture, their middleware, and their robots. This makes the sharing of software and, by extension, well-implemented algorithms, extremely difficult, if not impossible. To use the techniques developed by one group, other groups must re-implement them (in their own architectures, using their own middleware, for their own robots), often from descriptions in published articles. While such articles generally provide complete descriptions of the algorithm, they tend not to discuss the inevitable “tweaking” that real systems need in order to perform well. This means that any re-implementation effort is doomed to spend time fiddling with constants, and working around the (often implicit) assumptions of the algorithm designers.

This wide-spread re-implementation of algorithms wastes valuable research time with unnecessary programming and debugging. Each implementation is likely to have its own set of bugs and idiosyncrasies, leading to different performance than the reference implementation. If many researchers were to use the same implementation of an algorithm, comparison between systems using that algorithm would become more meaningful, bugs in the implementation would be more likely to be found and eliminated (especially in an open-source model [2]), and a huge amount of time and effort would

be saved. A necessary pre-requisite for such implementation re-use is a common communications middleware.

Why has the field of software engineering managed to settle on standard middleware implementations and robotics has not? In this paper, we argue that robotics is inherently a more heterogeneous field, where it is harder to establish (any kind of) standards. Robots also typically have more limited resources than other application areas, and suffer from a higher incidence of hardware and software failures. We look at the nature of these differences, and discuss what they mean for the development and evaluation of effective middleware for robotic systems. We suggest some first steps towards the development of such middleware, and discuss the ramifications on the field if we are successful. Finally, we draw some conclusions about the prospects for being able to implement, evaluate, and deploy effective middleware for robotic systems, and our chances of having more than a handful of people use it.

## II. MIDDLEWARE

Bakken [3] 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.

A number of successful middleware systems are in widespread use today, the most notable of which is, arguably CORBA [1]. Several implementations of the CORBA specification exist, and it is being used in a wide variety of real-world applications. A real-time version of the specification, RT-CORBA [4] allows systems requiring hard real-time performance to also take advantage of this middleware.

In this paper, we take a somewhat broader interpretation of middleware for robotics. We use it to not only include the communications infrastructure between elements of the system, but also the set of low-level, “primitive” behaviors that the control application can call on, where most users will be content with the (more-or-less) default settings of the behavior. By this definition, obstacle avoidance routines, movement to a local waypoint, and returning a sensor measurement are all taken care of by middleware. As more and more aspects of robotics become “solved”, they will find their way into our definition of middleware. At the time of writing, simultaneous localization and mapping (SLAM) for mobile robots could arguably be relegated to a middleware service. It is interesting, however, to note the reaction of researchers in the field to this contention. Those working on SLAM argue that it is madness to consider this to be part of the middleware, since there are many parameters to the popular algorithms, and it is important to have fine-grain control over them. Those interested in just using SLAM to get a position estimate often concede that it would be nice if this *was* part of a common middleware, and one could simply call `localize_robot()` to get a position estimate for the system.

### III. ROBOTS ARE NOT COMPUTERS

Despite our success in developing general-purpose middleware for distributed applications, we have failed to do the same for robotics. Interestingly, even middleware that is widely accepted in other application areas does not enjoy widespread use in robotics. This is, we argue, because of three main factors: the inherent heterogeneity of robotics, the limited resources (computational and otherwise) available on many robot systems, and the high rate of hardware and software failures in fielded systems. In this section, we look at each of these in turn.

#### A. Heterogeneity

One of the defining features of the field of robotics is its breadth and heterogeneity. For some researchers, a robot is a manipulator arm operating to sub-millimeter tolerances with real-time control. For others, it is a standard computer workstation mounted on a set of wheels with some off-the-shelf sensors plugged in. For some, it is a tiny device with an 8-bit processor and some simple binary sensors. Can we really produce a single set of middleware or a single software architecture that will work well in all of these cases?

There are three broad areas in which robot systems can be different from each other: hardware, the users of the robot, and the operational requirements of the system. We will deal with each of these in turn.

1) *The Physical Robot:* Every type of robot is unique. They have different locomotion mechanisms, different on-board computational hardware, different sensor systems, and different sizes and shapes. Since robots must operate in the world, these physical differences matter, and must be accounted for in the control software. This is typically not true of non-robotics applications that run on workstations.

The operating system on these workstations provides a common abstraction on which software can be executed. Software abstractions, such as the Java virtual machine make things even more homogeneous from the programmer’s point of view. This allows software written for one machine to be run easily on a wide class of similar machines with the same underlying abstractions.

While we can, and do, use these low-level abstractions on our robots, they are not enough. Although they allow us to write portable software, when this software must interact with physical devices, the abstraction is no longer sufficient. Consider middleware to provide an obstacle-avoidance routine for a mobile robot. It must know about the locomotion mechanisms, morphology, and sensors of the robot. Even when all of this information is known, writing a generic obstacle avoider that will work for all locomotion mechanisms, using input from all possible sensors is a daunting task. Before we can even think about this we must, at least, establish some common low-level abstractions over sensor measurements and movement through the world.

However, there are a number of problems inherent in deciding on, and working with, these abstractions. When we introduce abstractions, we must give up specific knowledge of the sensors we are using. Often when writing robot control software, we implicitly use our knowledge of sensors, their failure modes, and their placement on the robot. Removing these assumptions makes the software much more difficult to write. Consider the obstacle avoidance example again. On an iRobot B21r mobile robot with a forward-looking SICK laser range-finder, this is not such a hard task. The robot has a circular footprint, and the laser returns (more-or-less) radial distance measurements over the front half of the robot. The robot can turn in place without changing its footprint.

Now, consider writing this same obstacle avoidance routine for a robot of unknown shape, with an unknown number of sensors, arbitrarily mounted on the chassis, with an unknown locomotion system. While it is possible, it is a much harder exercise. All of the unknowns must be discovered somehow, perhaps from configuration files or a dynamic discovery mechanism. Any sensor computation must now explicitly take the position of the sensor into account. Any movement of the robot must consider the shape of the footprint and the locomotion mechanism. We must decide on a reasonable abstraction for the measurements returned by range sensors.

While such generic software is possible, it is much harder to write and debug. Most importantly, for most robot users, there is no value in writing such generic software. Most robotics groups have a small number of types of robots. It is generally more efficient and effective to write specialized low-level software for each of these platforms than it is to write generic “run anywhere” software.<sup>1</sup> This leads to replication of low-level routines for most of the robots even though more traditional elements, such as the low-

<sup>1</sup>Unless, of course, the research focus of the group is on writing generic robotics software.

level communication mechanisms, are shared. Higher-level routines are more likely to be shared across platforms, but these will be intimately tied to the lower-level infrastructure.

The situation is not hopeless, however. Systems such as Player [5] provide a low-level abstraction across a number of common robotics platforms. It does not, however, provide generic access to these systems. Code written for one platform will, most likely, not work on another one unless it is very carefully written. The Player abstraction is very much at the device access level: it is still up to the control software to interpret the numbers returned from the sensors and sent to the actuators.

2) *Robot Users*: In addition to the wide variety of robot platforms, there is a tremendous variety of users of robots. These different users can have very different needs, and can use their robots for very different purposes. Researchers working on core algorithms, such as SLAM or motion planning, require access to all of the parameters and settings of these algorithms. However, they are often happy with the default settings for other aspects of the system. Machine learning and vision researchers often treat the robot as a platform for sensors, and are uninterested in the low-level details of the obstacle-avoidance algorithms.

Encapsulation of irrelevant details is an important feature of any abstraction. How many of us, for example, alter the Maximum Transmission Unit (MTU) in our networking IP implementation? Although it might lead to better performance in some cases, most people are either unaware of it, or feel unqualified to pick a new value. Robotics algorithms are, or should be, the same. Most algorithms should have reasonable default values that are sufficient for the casual user. These defaults should be transparently enforced by the middleware. However, experienced users should be able to access all of the details of the implementation should they choose to do so.

Implementing different levels of abstraction into an interface takes more time than just allowing the lowest level of access. Designing a *good* abstraction takes even longer. Again, individual research groups are not likely to implement these different levels, simply because they are not useful to their research agenda. They will implement default interfaces to those systems that they do not need access to, and detailed interfaces to those that they need to fiddle with.

3) *Operational Requirements*: Finally, there is a great variety of operational requirements in robotics. Some systems must operate under hard real-time constraints, some need to run “fast enough”, and some can tolerate a “best effort” level of responsiveness. These requirements lead to very different systems. Components from one level are not likely to work well in another level, either because they do not meet the requirements (in the case of hard real-time systems), or because their implementations and interfaces are overly restrictive (in the case of “best effort” systems). Establishing a common middleware across these requirements seems like an impossible task.

Another source of operation variation is in the availability of human help for the robot. Deep space probes can expect

no help from a human, and must be completely autonomous, rovers on Mars can expect intermittent help (albeit with a time-delay), while robots in a research lab can expect to be coddled and well looked-after by a swarm of graduate students. The availability of human intervention can affect the middleware on the system. If no humans are present, there must be extensive error-detection (for both software and hardware), the ability to identify and quarantine defective elements of the system, and multiply-redundant systems. If, on the other hand, help is close at hand, these elements are not vital to the success of the robot. Once again, research groups that do not need the paranoid level of error-detection and recovery required by extremely remote systems will not, in general, implement it. This makes their middleware and implementations inappropriate for these systems. Conversely, implementation with extreme levels of error-tolerance are often seen as “overkill” for robots in a laboratory setting, wasting computation and other resources checking for events that are rare and easily-fixed by human assistants.

## B. Limited Resources

Mobile robots typically have an extremely limited set of resources, especially computation and power. Everything must run from batteries, so even if it is possible to add a powerful computer, there is always a price to pay in terms of the duty cycle of the robot. Many robots, especially those designed for harsh environments, have further restrictions on their computational abilities, enforced by the need for hardened components. In systems that must operate in hard real-time, we can also see time as a limited resource; we cannot perform extensive processing if we are to meet our real-time deadlines.

These resource limits affect how much off-the-shelf middleware can be used on robotic systems. For example, self-describing sensor data is a proposal that surfaces every so often in the robotics world. The main idea behind this is that every sensor measurement is accompanied by a description of the sensor it came from (type, model, position on the robot, and so on), the type of data that it is (distance measurement, for example), and the characteristics of the measurement (perhaps likely measurement error). XML is a commonly-suggested format for such information. While this is a fine idea for a truly general system, those researchers who have robots equipped with 8-bit microcontrollers are, understandably, aghast at such a proposal. Simply parsing the XML would consume all of their available computational resources.

Systems with more powerful computer systems and longer battery life can better tolerate the richer data representations required by middleware implementations. Fully general systems will need an extensive description language for all aspects of the robot, and a reasoning system to deal with it. In this context, we must pay a price for generality, and this price is computation and, ultimately, battery power.

### C. *Everything fails all the Time*

For traditional distributed applications operating over a network, many of the common hardware failures are hidden from the application by the network protocols or the middleware. For example, problems with the network infrastructure are dealt with by the TCP/IP protocols, and pools of remote servers ensure that there is no single point of failure when a machine goes down. Often implicit in these solutions is the assumption that serious failures are a rare thing, and are generally transient. This allows us to provide some protection against these failures by repetition and replication. If a network packet fails to get through, we resend it. If we have a critical server, we physically replicate it and keep the second machine as a hot-swappable backup of the first.

These solutions, however, are often not practical on a robot. Many failures of hardware are not transient, and we do not have the luxury of replicating our computer hardware, because of space and power limitations. Robots typically have a huge number of electrical and mechanical connectors of all sorts. They also (generally) spend most of their time being shaken, vibrating, and occasionally violently bumping into things. This means that it is almost inevitable that some of these connectors will become loose and fall out, computer cards will dislodge from their slots, and chips will become unseated.

Hardware will can also fail in less obvious ways. Consider the following anecdote (with names omitted to protect the innocent): At a major robotics competition, one team was getting ready for their run. An undergraduate on the team plugged in an extra pan/tilt unit to the main power bus just before the run was to begin. The power draw from this unit was enough to brown-out the laser range-finder, causing it to reset, and begin returning “maximum distance” measurements. The software reading the laser measurements assumed that the numbers it was getting were valid, since the laser range-finder is normally a very reliable piece of equipment. When the robot began its run, it drove straight into a table, and began pushing it, since the obstacle-avoider used the laser measurements, which seemed to indicate no obstacles.

While this anecdote is a single example, most robotics researchers have similar tales to tell. The problem is not that the fault could not have been detected but, rather, that it had to be done by the application programmer. Good middleware support would have taken care of this problem and, at the least, turned the “maximum distance” readings into “minimum distance” ones that would have stopped the robot in its tracks before it could assault the furniture.

The moral of this story is that there are more failures on robots than on standard networked computers, and middleware must take care of these failures if at all possible. If the responsibility is left to the application programmer, these exotic failure modes will go uncaught. This also implies that the robustness provided by the networking layer and current middleware is not sufficient for robot systems.

In addition to hardware failures, mobile robots typically

suffer from more software failures than other applications do. An inevitable fact of life for mobile robotics researchers is the need to write code in the field while on deployment. No matter how much one tries to prepare for a deployment in the real world, there is always some additional implementation and tuning that must be done on-site. Although we can institute procedures to make this process less error-prone [6], it is not without risk. Software in the field is often written hastily, racing to meet a deadline, and is almost never tested to the levels that software written in the laboratory is. This makes it much more susceptible to failures. Good middleware of robots should provide as much protection against faulty software elements as possible, to ensure that the effect of failures on the overall system is minimized.

### IV. TOWARDS ROBOTICS MIDDLEWARE

Specifying, designing, implementing, and deploying a widely-accepted middleware for robots is a daunting task. Many good systems exist (see the article by Kramer and Scheutz [7] for a comprehensive survey). However, none of them, with the exception of Player, is particularly widely used. Although the technical aspects of the task are challenging, it seems that winning a wide acceptance in the community is even more difficult. However, if we are to progress as a discipline we must create some common ground. Having a common middleware will allow us to share implementations and avoid time-wasting replication of others’ work. In this section, we suggest some concrete ways to move towards this goal.

In trying to find common ground it is vital to realize that we are never going to convince researchers to use a common software architecture, since everyone has strong (conflicting) opinions on this issue. We are not going to address the benefits and dangers of a common architecture, leaving that battle for another day. We will limit ourselves to discussing common middleware, according to our definition in section II. If we are successful, we will provide a communications infrastructure and low-level components that researchers can compose and add into their own systems.

Although we advocate building an open-source middleware for robotics, we recognize that there is an alternate path to a common standard. If a large company, with sufficient penetration into the research world, were to release a robotics middleware and software architecture, there is a good chance that, over time, this would become the dominant standard. Even if it is not the best (by whatever criteria this is measured), it may become the standard through weight of users. At the time of writing, Microsoft has just released Robotics Studio [8], a software architecture, middleware, and simulation environment for robotics. Only time will tell if this becomes the de facto standard in the years to come, and makes the content of this paper moot.

As we discussed above, robotics is a field characterized by its heterogeneity. However, in order to design a common middleware, we must first decide on a set of common abstractions and data types. At the ICRA 2007 workshop on Software Development and Integration in Robotics, this

subject came up several times. However, although all of our robots live in the same physical world, it was impossible to achieve a consensus of how this world should be represented. For example, measurements of angle are used by almost all robots. However, depending on the application, these can be represented by radians, degrees, Euler angles, or quaternions. Surprisingly strong opinions were voiced regarding the “right” angular representation. If we cannot agree on how to represent something as simple as an angle, what hope is there for us to develop any useful abstractions on which to base middleware?

Our solution to this problem is twofold. First, we must abandon for the moment to fantasy of a common middleware for *all* robots. We must divide the field into possibly arbitrary subfields if we are to make any practical progress. Second, we must choose abstractions that seem reasonable for these subfields, and resign ourselves to the fact that we will, most likely, have picked the wrong ones.

It seems clear to us that different areas of robotics have very different requirements. A high-speed robotic manipulator, requiring hard real-time control, and anchored to the floor is a very different system than a planetary rover. This being the case, it is only logical that we split the field into parts to design middleware for it. If we are successful, we will achieve our goals of timesavings and shared implementations in each sub-field. Although not completely satisfying, this fracturing of the field is vital if we are to make any practical progress. As an example, we might choose a subfield to be “wheeled robots operating indoors, with off-the-shelf sensors”. Although restrictive, this captures a huge number of research robots in university computer science departments (a location close to our own hearts).

Once we have suitably narrowed the scope of our efforts, we must choose suitable abstractions to base our middleware on. The key point to realize here is that we will not please everyone, regardless of how we choose our abstractions. The best that we can hope for is a broad consensus from a plurality of the field. Having settled on these abstractions, we must define a low-level set of routines to manipulate them, extract them from sensors, and send them to actuators. Gerkey [9] has suggested that we implement something akin to the POSIX standards. POSIX is a set of standards for Unix (and Unix-like) operating systems, outlining standard interfaces to the kernel, standard commands, and conformance testing procedures. It is a widely-accepted standard, and has resulted in software that is more easily ported between Unix systems.

In addition to defining low-level abstractions, we must provide graceful degradation of performance in the face of inevitable hardware and software failures. We cannot rely on the application programmer to check for device failures, so we must implement this in the middleware we develop. When a sensor fails, the middleware should select another similar sensor (if one is available), or fail-over to some safe simulated sensor signal (if no replacement is available). While this will obviously have an effect on the control application, it should help avoid catastrophic failures, such

as the one described in section III-C.

#### A. Evaluating Robotics Middleware

We can, of course, evaluate robotics middleware on traditional metrics, such as speed, and responsiveness, but this only tells half the story. In some sense the best measure of the effectiveness of any piece of software is how much that software is used. If every robotics researcher used our middleware, it would probably be fair to call it a success. However, if we are to compare new middleware implementations to existing ones, we must develop a set of metrics that make sense. Although we do not have any conclusive answers to this question, we do have some suggestions about what should be measured (even if we do not know *how* to measure them):

- 1) **Overhead.** How much overhead, in terms of computation and programming effort, do we need to expend to use the middleware, over an existing approach?
- 2) **Robustness to failures.** How well does the middleware deal with hardware and software failures in our system? How many failures (and of what type) can it tolerate before the system becomes useless?
- 3) **Ease of use.** What is the learning curve for moving to our middleware? Can existing software be ported to use the middleware easily?
- 4) **Generality.** Can we build existing software architectures out of our middleware? How difficult is it to do so?
- 5) **Flexibility.** Is the middleware easily extended by users? This is a key feature if the system is to evolve and be widely adopted.

## V. CONCLUSIONS

Progress in robotics is currently being held up by the lack of a common set of abstractions and middleware. Every research group is forced to write their own systems, re-implementing the work of others. This re-implementation, and the subsequent debugging, wastes time that could more profitably be spent on furthering the group’s research agenda. I widely-adopted set of abstractions and middleware would greatly improve the productivity of the field as a whole, while also improving the quality of the shared implementations.

However, the creation of such middleware is not an easy task. In this paper, we have discussed why designing middleware for robotics is a significantly harder task than designing middleware for traditional networked computer systems. There are additional constraints and failure modes that must be addressed if we are to succeed. There is also an astonishing lack of consensus in the field about how such middleware should be constructed, and even if it should be constructed at all.

We do not expect the first version of these abstractions and the middleware that uses them to be complete, or even correct. Several iterations will be necessary, with extensive feedback from the community, before we converge on a good solution. However, we firmly believe that the *only* way to achieve our goals is to roll up our sleeves and get started with

our best guess, rather than trying to decide on the “correct” way to proceed before starting. Although this “try it and see” approach might lead us down the wrong path, it seems clear that the alternative has not yet even got us through the gate.

#### REFERENCES

- [1] Object Management Group, *The Common Object Request Broker: Architecture and Specification (v3.0.3)*, 2004.
- [2] E. S. Raymond, *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O’Reilly, 1999.
- [3] D. Bakken, “Middleware,” in *Encyclopedia of Distributed Computing*, P. Dasgupta and J. E. Urban, Eds. Kluwer, 2000.
- [4] Object Management Group, *Real-Time CORBA (v1.2)*, 2005.
- [5] 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 (ICAR 2003)*, 2003, pp. 317–323.
- [6] W. D. Smart, “Writing code in the field: Implications for robot software development,” in *Software Development for Experimental Robotics*, ser. Springer Tracts on Advanced Robotics, D. Brugali, Ed. Springer, 2007, vol. 30, ch. 5, pp. 93–105.
- [7] J. Kramer and M. Scheutz, “Development environments for autonomous mobile robots: A survey,” *Autonomous Robots*, vol. 22, no. 2, pp. 101–132, 2007.
- [8] Microsoft, “Robotics studio,” <http://msdn.microsoft.com/robotics/>.
- [9] B. Gerkey, 2007, Personal communication.