Orca: Components for Robotics

Alexei Makarenko, Alex Brooks, Tobias Kaupp
ARC Centre of Excellence in Autonomous Systems (CAS)
The University of Sydney, Australia
www.cas.edu.au
{a.makarenko,a.brooks,t.kaupp}@cas.edu.au

***Abstract*— This paper describes Orca: an open-source software project which applies Component-Based Software Engineering principles to robotics. The paper focuses on the technical aspects of the framework which set it apart from other similar efforts. Among them are the adaption of a commercial middleware package, minimalist approach to framework design, and a commitment to multi-language multi-platform support. The paper explains our decision to use Ice – a relatively recent entry in the field of general-purpose middleware packages.**

I. Introduction

This paper is about *Orca* – a software framework for developing robotic systems. It is one of several systems with similar objectives which are currently under development.

Approaches to software development in the context of robotics currently attract considerable attention because creating robotic software is difficult and time-consuming, and perceived to be one of the limiting factors in the progress of robotics today. It is generally acknowledged that the key to making progress in this area is software reuse, which includes both free open-source software and commercially available alternatives.

Orca is an open-source framework for developing component-based robotic systems. It provides the means for defining and developing the building-blocks which can be pieced together to form arbitrarily complex robotic systems. The types of systems we are targeting range from single vehicles to distributed sensor networks. Importantly, we envision usage patterns typical for both academic and commercial environments.

The project's main goal is to promote software reuse in robotics. Many factors contribute to the success or failure of a reuse initiative [12]. Based on an informal survey of the field, it appears that there is a growing consensus that a successful framework would be:

1) open-source;
2) distributed under a license which allows use in commercial applications;
3) modular; and
4) distributed with a repository of tested and documented modules.

These requirements are satisfied by Orca design choices. Orca is an open-source project hosted on *SourceForge.net* and is distributed under a combination of LGPL and GPL licenses. The basic module is a component which can be independently distributed, deployed, and configured. The project distribution includes a repository of components.

Next we list a set of technical requirements which are more controversial and not shared by other design teams. In our opinion, the framework must be:

5) general, flexible and extensible;
6) sufficiently robust, high-performance and full-featured for use in commercial applications; and
7) sufficiently simple for experimentation in university research environments.

Note that these requirements are somewhat contradictory. In particular, items 6 and 7 are not easily combined. To satisfy the given set of requirements, Orca:

- adopts a Component-Based Software Engineering approach without applying any additional constraints (requirements 3, 5);
- uses a commercial open-source library for communication and interface definition (5, 6);
- provides tools to simplify component development but make them strictly optional to maintain full access to the underlying communication engine and services (6, 7); and
- uses cross-platform development tools (5).

This set of design choices sets Orca apart from other projects with similar objectives. The following sections discuss these design choices in turn. A brief comparison with other initiatives is also provided.

II. An Unconstrained Component-Based System

The approach we take is frequently called Component-Based Software Engineering (CBSE) [13]. CBSE offers developers the opportunity to source existing plug-in software components, rather than building everything from scratch. In addition, CBSE offers significant software engineering benefits by enforcing modular systems, which helps control dependencies, reduce maintenance costs and increase system flexibility and robustness.

Fundamental to CBSE are 1) the concept of interfaces considered to be contractual obligations between components, and 2) the choice of a mechanism to implement these interfaces. Strict adherence to a set of predefined interfaces imposes severe constraints on the interactions allowed between components. These constraints are necessary to ensure interoperability of the components.

In the design of a framework for reuse it is certainly possible to go beyond the fundamental interface constraints

but, because Orca aims to be as broadly applicable as possible, we choose not to do so. In particular, we make no prescriptions or assumptions about: component granularity, system architecture, the set of provided or required interfaces, and the components' internal architecture.

A. Arbitrary Component Granularity

Component granularity refers to the size of modules into which the system is broken up. In many engineering disciplines, a trade-off exists between building a monolithic, specific implementation versus a general, modular implementation. The former is usually more efficient but less re-useable or maintainable than the latter. Orca (and CBSE in general) favors the latter approach. Where projects have elements that need to favor efficiency, developers are free to implement the entirety of those elements within a single Orca component. From the point of view of the rest of the system it makes no difference, so long as the same external interfaces are exposed.

B. No System Architecture

Orca is an implementation framework and not an architecture. In other words, any architecture, be it a centralized blackboard, strictly-layered, strictly-decentralized, or mixed, should be implementable with Orca. System developers are free to compose a system from any set of components, arranged to form arbitrary architectures, so long as interfaces are connected correctly. There is no special component which the framework requires all implemented systems to incorporate.

C. Arbitrary Interfaces

Individual components are free to provide or require any subset of valid interfaces. There is no particular interface which all components must provide or require, and it is easy for component developers to invent new interfaces.

D. No Component Architecture

Component developers are free to provide the implementations of their components' interfaces (which are opaque to the rest of the system) in any way they choose.

III. Ice Middleware

To implement a distributed component-based system, one must be able to define interfaces and make a choice of communication mechanism. In the case of cross-platform operation involving different operating systems, the software which provides such functionality is typically referred to as *middleware*. Given the realities of robotic software development we consider support for C/C++ on Linux to be essential. This rules out Microsoft's COM+ and Sun's Enterprise JavaBeans. With this in mind, the following options exist today:

- using XML-based technologies such as SOAP;
- using CORBA [1];
- writing custom middleware from scratch; or
- using Ice [7].

We discount XML-based technologies on the grounds that they are too slow and inefficient for low-level robotic control tasks. While CORBA is sufficiently flexible for Orca's middleware requirements, it is also large and complex. Experience with CORBA in earlier versions of Orca showed this complexity to be problematic. In comparison, Ice offers a much smaller and more consistent API, superior feature set and similar performance [7].

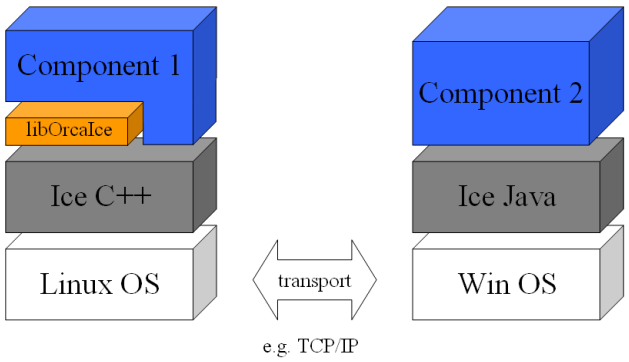


Fig. 1. Two Orca components written in different languages and executing on different operating systems. Both can be acting as a server or as a client. The optional libOrcaIce distributed with Orac simplifies component development in C++.

Earlier versions of Orca also experimented with writing custom middleware from the ground up. While communicating over a socket is simple, implementing middleware sufficiently flexible and reliable to support Orca's requirements involved re-implementing (and maintaining) large parts of CORBA functionality, which is a non-trivial task. Our conclusion was that it is unrealistic to expect robotics researchers to have the time or skills to develop middleware to the same standards as commercial products.

Note that compared to a previous publication [2] we have reversed our opinion on using a higher-level abstract transport-independent mechanism. Experience has shown that, in addition to the significant development effort, the feature set available to the abstract interface is necessarily limited to the lowest common denominator of the lower-level implementation alternatives.

These arguments led to the decision to choose Ice as the single middleware foundation for our project. The rest of this section briefly describes what Ice is and what features it offers.

A. Ice

The Internet Communications Engine (Ice) [8] is a modern proprietary implementation of middleware ideas similar in spirit to CORBA.

Slice is the Specification Language for Ice, analogous to IDL for CORBA. Slice is used to define interfaces – a contract between clients and servers. Slice specifications are then compiled into various programming languages. Ice supports C++, Java, Python, PHP, C#, and Visual Basic. Ice clients and servers can work together regardless of the programming language in which they are implemented.

The Ice core library manages all the communication tasks using a protocol which includes optional compression and

support for both TCP and UDP. Ice optionally provides a flexible thread pool for multi-threaded servers.

Ice builds natively under various operating systems including Linux, Windows, and MacOS X. Figure 1 shows an example of two Orca components written in different languages and running on two different operating systems.

B. Ice Services

Ice middleware comes with several services which provide additional functionality. Some of them are used extensively in Orca.

Centralized Registry. `IceGrid Registry` resolves symbolic names of services to their addresses on the network.

Application Server. `IceBox` is a simple application server that can orchestrate the starting and stopping of a number of application components. Application components can be deployed as a dynamic library instead of as a process. There are several advantages in using this architecture. The two main ones are: ease of deployment and administration, and optimization of communication between components within the same application server.

Event Service. A common cause of difficult-to-debug faults in larger systems tends to be the coupling between publishers and subscribers of information. Two problems can occur when communication is unreliable or clients are slow. Firstly, slow clients can delay the publishing component's thread, causing problems that appear to be related to the publisher's algorithm. Secondly, a slow client can starve faster clients, causing problems that appear to be related to the faster clients. `IceStorm` is a stand-alone component which receives data from a single server and disseminates it to multiple clients. It relieves components of the burden of managing subscribers and handling delivery errors. Publishing to a local event service ensures that network delays do not interfere with the operation of the publishing algorithm. `IceStorm` is efficient, since it forwards messages without needing to marshal or demarshal them. It can be configured with multiple threads, to weaken the dependencies between clients.

Other features and services which are not currently used in Orca but which are clearly relevant to industrial applications include: persistent storage, deployment and activation services, load balancing, management of software updates, firewall, etc. A secure communication layer is available and may also be important in commercial applications. Availability of a light-weight embedded version of Ice is very promising for deployment on small computing devices.

C. On Custom vs. General Middleware

The implementation of a component-based system consists of the components themselves and the infrastructure necessary to allow components to work with each other. As developers of robotics software we want to concentrate our efforts on the component code, which is useful for robotics applications, while minimizing the amount of effort spent on writing and maintaining the infrastructure code. The effort required for software development is notoriously hard to judge but code size does give a meaningful approximation.

Figure 2 plots the size of the Orca project as it evolved over time. The sizes of the components and the infrastructure are tracked and plotted separately. The component code size is shown as positive numbers and the infrastructure code size is shown as negative numbers (because this is something we want to minimize). Of particular interest is a large reduction in the size of the Orca infrastructure which was facilitated by the adoption of the Ice middleware. The size of the relevant parts of Ice is also calculated and shown as underpinning the Orca infrastructure. The trend we would like to see in the future is the growth in the size of the components with no or small changes in the size of the Orca infrastructure.

The figure also shows the equivalent numbers for the Player/Stage project [6] which employs custom middleware. The version released in 2006 introduced several new features which required a significant rewrite of the middleware layer with a visible increase in the code size.

IV. Strictly Optional Tools

Ease of use is certainly one of the main factors contributing to popularity of a particular software tool. We consider the Player API to be the gold standard of simplicity, particularly on the client side. In our experience, using Ice API directly falls somewhere in between the simplicity of a custom robotic-specific middleware like Player and the complexity of CORBA.

To lower the entry barrier for developers, Orca provides a library called `libOrcaIce` which is intended to provide a simplified API sufficient for the majority of usage patterns encountered in the development of robotics components. It is important to emphasize that the use of `libOrcaIce` and all other Orca utilities is strictly optional, and it is possible to write new components and communicate with the existing ones without using any of the supplied tools.

Even if a component is written with the help of `libOrcaIce`, it is still possible to use the full range of Ice functionality. This fact is shown graphically in Figure 3 where Component 1 has access to both `libOrcaIce` and the underlying `libIce` for C++.

While we avoid enforcing particular design patterns for either systems or components, we do provide guidelines (and working code!) for designs that have worked well in the past. One useful feature of `libOrcaIce` is the transparent integration of component code into a stand-alone application and an `IceBox` service. Figure 3 demonstrates how the same component code can be easily packaged as either a stand-alone application or a service deployable inside an application server.

V. Cross-platform Tools

Orca was designed with the intention of being used on various platforms. To this end, we use CMake (cross-platform make) [9] for its build system. Ice has language mappings to many languages and builds natively under several operating

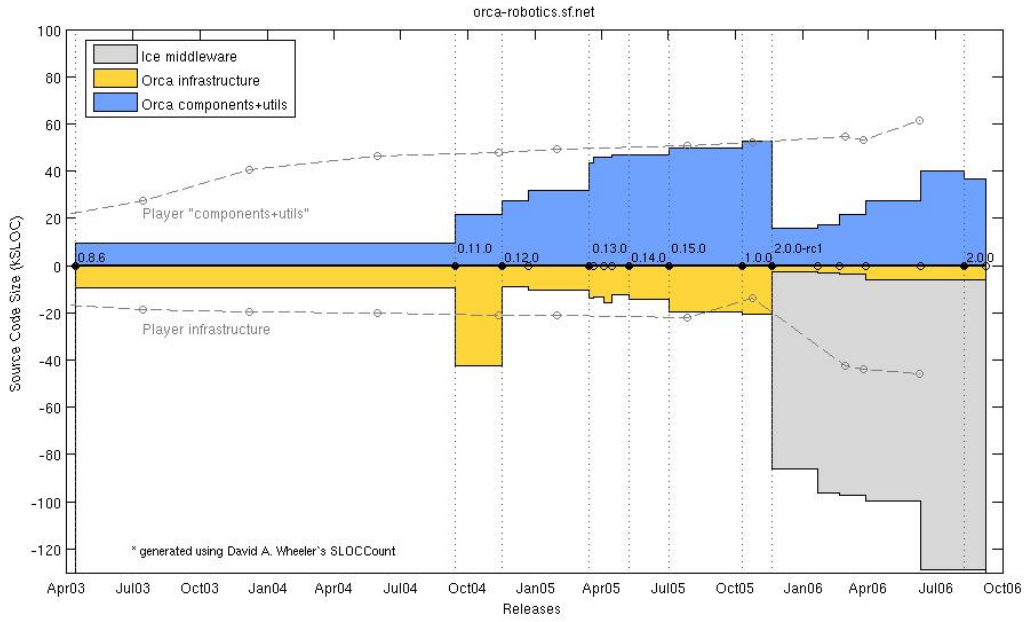
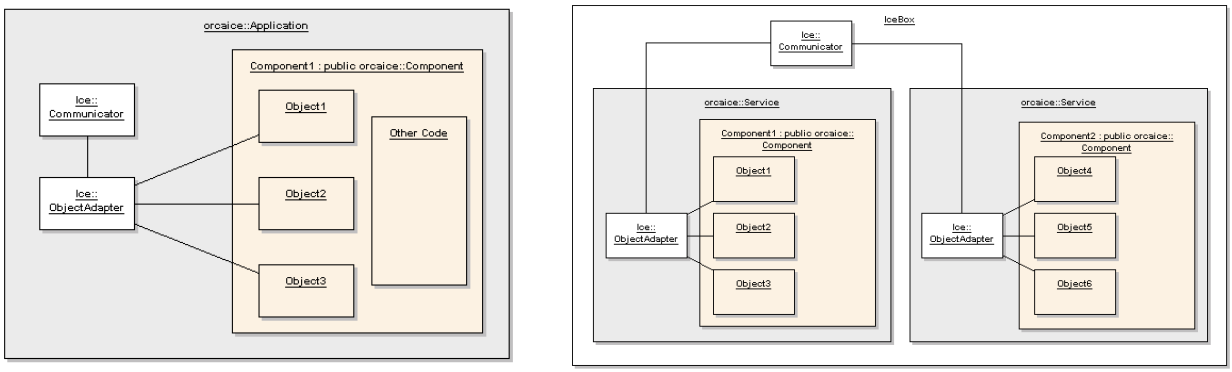


Fig. 2. History of project size for the Orca project. Player/Stage data is shown for comparison. Full details and data are available from the project website [11]. SLOCCount tool [5] was used.



(a) (b)

Fig. 3. "Write-once deploy-twice" feature of libOrcaIce. The main idea is to have the internals of the component used unmodified in both stand-alone and application server environments. (a) Integration of component code into a stand-alone application. White color stands for standard Ice objects which we do not modify, e.g. the Communicator – the Ice run-time object. Gray color stands for the objects which are part of libOrcaIce and used unmodified by component developers. Cream color stands for component-specific code, written by component developers. (b) Integration of component code into an IceBox service. Two services are shown running side by side. Under this configuration, communication between the two services is optimized to function calls.

systems. The majority of components in the Orca repository use the Linux/C++ combination. The core of the project compiles on Windows and there is currently a simple example in Java. For GUI development we use TrollTech's Qt – a multi-platform C++/Java GUI framework.

VI. Related Work

The idea of applying CBSE principles to the field of mobile robotics is not new. Only a few initiatives fulfil all the requirements listed in the Introduction. By far the biggest constraining factor is the availability of reusable components.

Several existing projects in mobile robotics do maintain large repositories of well-tested re-useable implementations of algorithms for mobile robotics. The two most popular are CARMEN [10] and Player [14]. CARMEN, the Carnegie Melon Robot Navigation Toolkit, communicates between components using a custom communication library IPC. It is designed specifically for single-robot systems. Its use is limited to Linux and C++.

While Player was not designed explicitly with CBSE principles in mind, it employs many component-based ideas. As the most successful mobile robotics re-use project to date, Player has arguably become the de-facto standard. Client libraries exist for several languages on different platforms, but the server side of Player is limited to Linux and C++. For a more

detailed comparison of Orca and Player see [3].

Orocos [4] is based on CORBA with applications primarily in the field of robotic manipulators. Its use is also limited to Linux and C++.

VII. Conclusion

We presented the current state of the Orca project. Main directions of current work are learning how to use Ice features effectively and improving reliability of existing components.

Deployment and management of components becomes increasingly important for larger and longer-running systems. Similar to other systems, we currently use plain-text configuration files and run executables from the command line. This approach offers simplicity which is hard to beat. It has its limitations however, and we are learning to use the `IceGrid` service which offers powerful features for service activation, administration, software distribution, etc.

DARPA Grand Challenge-3 is an autonomous ground vehicle competition and one of the teams is using Orca as its software platform. This experience will drive improvements to the framework and will contribute components to the repository.

Acknowledgment

This work is supported by the ARC Centre of Excellence programme, funded by the Australian Research Council (ARC) and the New South Wales State Government.

References

[1] F. Bolton. *Pure CORBA*. Sams, 2001.

[2] A. Brooks, T. Kaupp, A. Makarenko, A. Orebck, and S. Williams. Towards component-based robotics. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2005)*, pages 163–168, 2001.

[3] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, and A. Oreback. Orca: a component model and repository. In D. Brugali, editor, *Principles and Practice of Software Development in Robotics*. Springer, 2006 (In Press).

[4] H. Bruyninckx. Open robot control software: the OROCOS project. In *IEEE Int. Conf. on Robotics and Automation (ICRA'01)*, volume 3, pages 2523–2528, 2001.

[5] D.Wheeler. SLOCCount. `http://www.dwheeler.com/sloccount`.

[6] B.P. Gerkey, R.T. Vaughan, and A. Howard. The Player/Stage project: Tools for multi-robot and distributed sensor systems. In *The 11th International Conference on Advanced Robotics (ICAR'03)*, pages 317–323, Coibra, Portugal, 2003.

[7] M. Henning. A new approach to object-oriented middleware. *IEEE Internet Computing*, 8(1):66–75, 2004.

[8] M. Henning and M. Spruiell. *Distributed Programming with Ice*. 2006.

[9] Kitware Inc. CMake: Cross-platform Make. `http://cmake.org`.

[10] M. Montemerlo, N. Roy, and S. Thrun. Perspectives on standardization in mobile robot programming: The Carnegie Mellon navigation (CARMEN) toolkit. In *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 2436–2441, 2003.

[11] Orca Project. Orca2. `http://orca-robotics.sf.net`.

[12] D.C. Schmidt. Why software reuse has failed and how to make it work for you. *C++ Report*, 11(1), 1999.

[13] C. Szyperski, D. Gruntz, and S. Murer. *Component software : beyond object-oriented programming*. ACM Press; Addison-Wesley, New York London, 2nd edition, 2002.

[14] R.T. Vaughan, B.P. Gerkey, and A. Howard. On device abstractions for portable, reusable robot code. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003)*, pages 2421–2427, 2003.