Orca: A Component Model and Repository

Alex Brooks[1], Tobias Kaupp[2], Alexei Makarenko[3], Stefan Williams[4], and Anders Orebäck[5]

[1] ACFR, University of Sydney, AUSTRALIA a.brooks@acfr.usyd.edu.au
[2] ACFR, University of Sydney, AUSTRALIA t.kaupp@acfr.usyd.edu.au
[3] ACFR, University of Sydney, AUSTRALIA a.makarenko@acfr.usyd.edu.au
[4] ACFR, University of Sydney, AUSTRALIA s.williams@acfr.usyd.edu.au
[5] Royal Institute of Technology, SWEDEN oreback@nada.kth.se

Summary. This Chapter describes Orca: an open-source project which applies Component-Based Software Engineering principles to robotics. It provides the means for defining and implementing interfaces such that components developed independently are likely to be inter-operable. In addition it provides a repository of free re-useable components. Orca attempts to be widely applicable by imposing minimal design constraints. This Chapter describes lessons learned while using Orca and steps taken to improve the framework based on those lessons. Improvements revolve around middleware issues and the problems encountered while scaling to larger distributed systems. Results are presented from systems that were implemented.

1 Introduction

This Chapter describes Orca: an open-source project which applies Component-Based Software Engineering (CBSE) principles to robotics. Many robotic software projects are presented in this book, however the Orca project stands out by both

- explicitly adopting a component-based approach from the outset, and
- acknowledging the importance of a component market.

While it is not alone in either respect, the Orca project is, to the authors' knowledge, the only framework to combine the two.

The introduction to this Part defined Component-Based Software Engineering, identified its benefits and motivated its application to robotics. The benefits of a component-based approach include

- the ability to build systems quickly by incorporating existing third-party components,
- the ability to build more reliable systems by incorporating components which have been tested across multiple projects,

D. Brugali (Ed.): Software Eng. for Experimental Robotics. STAR 30, pp. 231–251, 2007.

- the software engineering benefits of having a modular system with controlled, explicit dependencies only, and
- the ability to build flexible systems in which individual components can be replaced.

In non-commercial settings, component markets entail the trading of independently-developed components. The recognition of the importance of this has both a practical and a technical impact. Practically, it drives the maintenance of the Orca component repository[6]: an online selection of open-source, interoperable, re-useable robotic components. Significant effort has been devoted to providing sufficient documentation for both the framework itself and for each component, to ensure that components can indeed be deployed by third parties. Technically, the recognition of the importance of component markets drives Orca to make as few assumptions as possible about the systems to which it may be applied, in order to be applicable across as broad a market as possible. Section 2 develops these ideas further, presenting the basic philosophy behind the Orca approach.

The two main principles have remained unchanged since the start of the project in 2001 (under the name of Orocos@KTH). Some implementation details have undergone radical changes, warranting a recent increment of the version number from Orca1 [BKM05] to Orca2. A number of lessons were learned based on experiences with Orca1, resulting in improvements that were made or planned for Orca2. These revolve around middleware-related issues and the problems discovered when scaling to large distributed systems, as discussed in Sections 3 and 4 respectively.

To demonstrate the effectiveness of the framework, Section 5 describes some of the component-based systems that have been built and the re-use that has occurred between those systems. Section 6 compares Orca with the Player project. The comparison is included because Player is the only robotics software project to have established a significant market, and hence has become a de-facto standard in robotics. Finally, Section 7 concludes.

2 Design Philosophy

2.1 Design Minimalism: Impose as Few Constraints as Possible

The design of the Orca framework is conceptually simple. A system consists of a set of components which run asynchronously, communicating with one another over a set of well-defined interfaces. Each component has a set of interfaces it provides and a set of interfaces it requires. The fundamental purpose of the framework is to provide the means for defining and implementing these interfaces. Standardising the definitions and implementations of interfaces ensures that components are likely to be inter-operable, and hence re-useable.

[6] available from http://orca-robotics.sourceforge.net

This statement of purpose is most significant in its omissions. The authors contend that the framework should prescribe as little as possible to achieve the stated aim. This conclusion is reached from the following premises:

1. The benefits of a component-based approach only become apparent when a critical mass of component developers and users arises.
2. There are many different kinds of robotic systems, each with different requirements. For each set of requirements, there are many possible approaches to the system's design.
3. There are many approaches to designing any piece of software (including components) and, correspondingly, many opinions about their relative merits.

The first premise can be motivated by the following simple observation [Szy02]:

> Imperfect technology in a working market is sustainable.
> Perfect technology without any market will vanish.

Indeed, in the absence of a market, the value of a component-based approach is unclear. Producing a general, modular, component-based solution requires significantly more effort and planning than building a specific solution for the specific problem at hand. This extra effort is only worthwhile if there is a reasonable chance of receiving a return on that investment. Szyperski offers the rule of thumb that the break-even point is reached when a component is used across three different projects [Szy02].

In the presence of a component market, the benefits of designing systems to be inter-operable with third-party components become obvious. The alternative to using available components is the re-invention of solutions, which is advantageous only when the home-grown solutions are of significantly better quality. As market size increases this becomes less likely since components act as multipliers in a market, focussing the combined innovation of many individuals.

Examples of the different kinds of robotic systems referenced in the second premise include single-vehicle indoor research systems, large multi-robot outdoor commercial systems and unmanned aerial vehicles. These examples are likely to focus on different requirements, such as a) flexibility, b) scalability, safety and reliability, and c) real-time performance, respectively.

Even for a specific robotic domain there are widely differing approaches. Consider indoor research robots as an example: three-layer architectures [BFG97] involve little direct communication between sensors and high-level deliberative planners. In contrast, different requirements (particularly architectural) are imposed by a reinforcement learning [SB98] approach in which a learner requires direct access to all sensor data.

The third premise is evidenced by the proliferation of libraries that exist for any given purpose. For example, various GUI toolkits exist, all of which perform similar tasks and have large user-bases.

Given that a critical user mass is required, it follows that a framework should strive to be capable of accommodating as broad a range of projects as possible. Furthermore, given that requirements and approaches vary so widely, it follows that a framework should prescribe as little as possible while achieving its stated aims. Since any assumption is likely to be incompatible with some project or approach, unnecessary assumptions should be avoided.

In particular, Orca avoids making assumptions about any of the following:

1. **Architecture:** system developers should be free to compose a system from any set of components, arranged to form an arbitrary architecture, so long as interfaces are connected correctly. There should be no special component which the framework requires all systems to incorporate.
2. **Interfaces:** individual components should be free to provide or require any set of interfaces. There should be no particular interface which all components must provide or require. It should be easy for component developers to define new interfaces.
3. **Internals:** component developers should be free to provide the implementations of their components' interfaces in any way they choose. The implementation details remain opaque to the rest of the system.

To prescribe design rules for any of these aspects is to make assumptions about all robotics projects, present and future. Should those assumptions prove to be unjustified, the re-usability of both the framework and the set of its components is restricted.

2.2 Impose Necessary Design Constraints

While every attempt is made to avoid imposing constraints on component designers, certain decisions do need to be made to ensure component interoperability. In particular, a framework must choose

- a set of pre-defined interfaces, and
- how communication works between interfaces and how components can implement those interfaces.

Orca is distributed with the set of interfaces used by components already in the repository. New interfaces can be defined as needed. The framework is designed such that a new interface can be added without interfering with any existing interface. It is also important to avoid a proliferation of similar but subtly different (and incompatible) interfaces. In this regard no technical solution will replace old-fashioned communication between developers.

The second point involves the selection of middleware. While Section 3 describes the particular choices made by Orca, more generally this decision requires the consideration of the trade-offs listed below:

Efficiency versus Modularity

In many engineering disciplines, a trade-off often 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) favours the latter approach.

Where projects have elements that need to favour 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.

Footprint

A framework must make a choice regarding the minimum platform to support. Many robotics projects make use of micro-scale platforms such as MICA motes or Khepera robots. While support for these platforms is desirable, it imposes constraints on those who develop for larger platforms. In particular, it constrains the maximum footprint of the framework and precludes the use of more modern languages such as C++. In this trade-off Orca chooses to favour ease-of-use over portability. This does not prohibit the use of less-capable platforms within an Orca system, however they must be incorporated using non-Orca communication.

Real-Time Capabilities

A trade-off exists in whether a framework chooses to provide real-time guarantees for inter-component communications. Many robotics projects require some element of real-time operation, however ensuring that the entire system is real-time imposes unnecessary constraints on those parts of the system which do not require it. While real-time component-based frameworks do exist [Bru01], Orca chooses not to support real-time inter-component communications.

2.3 Offer Useful Design Guidelines

While the imposition of unnecessary design constraints is unhelpful, a framework can certainly assist component developers by providing design guidelines, for example on the design of components' internals or effective system architectures. Providing working code, in the form of libraries or examples, simplifies the implementation of these guidelines. A framework can thus significantly improve the average reliability and maintainability of its components, especially those written by inexperienced developers, as well as improving the ease with which new components can be developed.

A natural consequence of this approach is that conventions will develop. This is beneficial in general: code is easier to understand when a component's

internals follow a convention, and system composers have less to learn if configuration options are consistent. There is an important difference however between these conventions (which are useful and should be allowed to evolve) and interface standards (which are necessary and need to be prescribed). The difference is that the former will arise in those domains in which they make sense, but can be broken, with no disastrous ramifications, in unforeseen cases where they do not apply. Conversely, if interface compatibility is broken the consequences are more severe: if components can no longer communicate they can no longer be replaced between systems, and hence cannot be re-used at all.

The issue of a component's internal behaviour, broached in the Introduction to this Part, provides a useful example. Rather than prescribing the internal behaviour, a framework should define interfaces which expose the necessary functionality (such as *observability* or *controllability*), without the requirement that all components provide those interfaces. If the internal behaviour is suggested rather than prescribed, one can be more aggressive in making those suggestions useful rather than cautiously ensuring that they be general enough to apply to all scenarios.

3 Lessons Learned: Middleware Issues

Middleware refers to the software which acts as an intermediary between different application components [Szy02]. It plays a fundamental role in component-based systems, addressing many issues such as:

1. marshalling and de-marshalling,
2. application-layer communication protocols,
3. versioning,
4. inter-component exception handling, and
5. the means by which interfaces are defined and implemented.

3.1 Middleware for Orca1

There are surprisingly few options for the choice of middleware package. If support for C/C++ on Linux is a requirement (which discounts Microsoft's COM+ [Ewa01] and Sun's Enterprise JavaBeans [Ble01]), the following options exist today:

1. using XML-based technologies such as SOAP,
2. using Ice [Hen04],
3. using CORBA [HV99], or
4. writing custom middleware from scratch.

We rule out XML-based technologies on the grounds that they are too inefficient for low-level robotic control tasks. Ice is a recent development which was insufficiently mature at the time Orca1 was being designed.

Orca1 implemented both of the remaining options. All communication occurred via a set of communication patterns. The user decided, at compile-time, which middleware option would be used to implement the communication patterns. The choice to implement both options was made because neither is without significant drawbacks.

While the CORBA specification is certainly complete in terms of Orca's middleware requirements, it is also large and complicated. This had two main impacts on the project: firstly, from the project maintainers' point of view, the CORBA C++ API is well-known to be complex and difficult to use [Hen04]. The complexity was overcome by wrapping the CORBA API with a simpler Orca1 API, specialised for Orca1's communication patterns. While this sheltered new users who were not CORBA experts, it required considerable time and effort both to write and to maintain. Secondly, from the point of view of component developers, the most apparent impact of the size and complexity of CORBA was the significant increase in compile times, which did have a tangible effect on usability.

The alternative of writing one's own middleware from scratch also requires significant effort. Orca1 implemented simple TCP-based and UDP-based middleware systems 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. Cross-platform support was minimal and cross-language support was never attempted.

3.2 Middleware for Orca2

Providing multiple middleware implementations allowed Orca1 to support projects with different requirements, and avoided the need to make a single choice when the relative merits of the various options were unknown. This flexibility was paid for in two ways. Firstly, it required extra implementation and maintenance effort, which was detrimental to the quality of each individual middleware implementation.

Secondly, rather than being able to take advantage of the specific properties of each middleware option, the abstract communication API supported only the intersection of features provided by the three options[7]. For example, interfaces had to be specified in a middleware-agnostic manner, making it impossible to take advantage of the full capabilities of CORBA's Interface Definition Language (IDL).

The biggest change in Orca2 is the adoption of a single middleware layer. The authors consider Ice to be sufficiently mature at the time of writing and

[7] Interestingly, Player [GVH03] is moving towards offering a choice of middleware options.

vastly superior to the alternatives. It is far more flexible and reliable than any middleware that is likely to be written from scratch by robotics researchers. Ice can be compiled natively under various operating systems including Windows, Linux and Mac OS X and has several language mappings including C++, Java and Python. Ice has far less overhead when compared with CORBA, especially with respect to complexity (see [Hen04] for a comparison). It is released under the GNU Public License (GPL), however commercial licenses are available for closed-source software. Adopting a single middleware layer allows Orca to take full advantage of the features of the middleware layer rather than crippling each type of middleware to fit a common model.

3.3 Interface Definition

Unable to take advantage of an off-the-shelf interface definition language, Orca1 defined interfaces using *communication patterns* as methods for transferring data and *objects* as the data types which were transferred. This approach was sufficient for many applications, however it had the following deficiencies:

1. There was no clear way to relate associated functionality. For example, a laser server provided both scans and the laser's configuration details. With no obvious way to relate the two, the association had to be performed manually.
2. Polymorphism of interfaces was not possible. Therefore a slightly different *object* resulted in an entirely different (and incompatible) interface.
3. On creation of a new *object*, developers were required to write object-specific serialization code for marshalling, demarshalling, logging to files and reading from files. This presented a practical barrier to framework extensibility.

Instead Orca2 uses Ice's interface definition language, called Slice, to define interfaces. To demonstrate Slice's flexibility an example for a laser scanner is presented in Listing 1. Given a reference to a specific Laser interface, one can easily access both the data and the geometric information *for that laser*. The inheritance of Laser from RangeScanner shows how custom fields can be added to new interfaces while maintaining compatibility with existing interfaces. Defining new interfaces is trivial, and Slice automatically generates all serialization code from the interface definition.

4 Lessons Learned: Scaling to Complex Component-Based Systems

An important lesson learned from the Orca1 experience was the difficulty of scaling up. A certain amount of effort was required to implement each component. It eventuated that the amount of extra effort required to actually

```
// RangeScanner interface

sequence<float>        RangeSequence;

exception CannotImplementConfigurationException extends OrcaException {};

class RangeScannerData extends OrcaObject {
    RangeSequence     ranges;
    float             startAngle;
    float             angleIncrement;
};

interface RangeScanner {
    RangeScannerData      getData();
    RangeScannerConfig    getConfig();
    RangeScannerGeometry  getGeometry();

    void setConfig( RangeScannerConfig config )
        throws CannotImplementConfigurationException;
};


// Laser Scanner interface

sequence<byte>         IntensitySequence;

class LaserData extends RangeScannerData {
    IntensitySequence intensities;
};

interface Laser extends RangeScanner {};
```

Listing 1: Definitions of the RangeScanner and Laser interfaces. Some details are omitted for brevity.

configure, deploy and operate a system of those components was significant. When applied to large distributed systems, such as those described in Section 5, this extra effort could even dominate the total effort required.

This eventuality is not uncommon in distributed systems outside of the robotics field [WSO01]. A number of services and utilities have proved useful for mitigating the problems associated with large systems:

1. naming or trading services, to allow location transparency,
2. event services, to de-couple publishers from subscribers,
3. persistence services, to maintain state beyond a component's lifetime,
4. application servers,
5. deployment utilities,
6. system composition and configuration utilities, and
7. remote monitoring utilities.

Availability of these tools is an important consideration when selecting a middleware package or robotics framework. This Section describes the problems that were found to be most problematic and the tools used to solve them.

4.1 Composition and Configuration

Each Orca1 component was configured with a separate file which defined its inter-component connections and modified its behaviour. Defining all the connections by hand in a large system was tedious and error-prone. When system misbehaviour was caused by a misconfigured component, identifying the cause involved inspecting multiple files on multiple filesystems.

The solution attempted in Orca1 was to provide structured information regarding how each component could be connected and configured. This information was used to validate hand-written configurations and to generate configurations using `GOrca`: a graphical system composition and configuration utility similar to Simulink. `GOrca`, shown in Figure 1, parsed a set of component definition files to create a library of available components. Users could drag components from the library onto a project, connect their interfaces graphically and set configuration parameters textually. Finally, `GOrca` would export a set of valid configuration files representing the entire system.

While `GOrca` provided an effective means of generating configuration files, the number of generated files soon became unmanageable. Orca2 simplifies this further using Ice's application server, called `IceBox`. Developing components as dynamically-loadable libraries allows them to be deployed either as stand-alone components or as services within an `IceBox`. Configuration is simplified because an `IceBox` has a single configuration file which specifies the set of components to instantiate and their individual configuration details. In addition, invocations between components within an `IceBox` are optimised by Ice, taking advantage of the fact that they co-exist within the same process[8].

4.2 System Deployment

The deployment of a component-based system entails ensuring binary synchronisation across all hosts (either by copying source code and compiling locally, or by copying binaries), copying configuration files, then starting and stopping components. The amount of time spent tracking down failures resulting from unsynchronised binaries was significant, and having to open one remote shell per component was onerous.

[8] A Player server may also be comsidered an application server. The important difference in Orca2 is that communication is identical whether a component is a stand-alone application or part of an `IceBox`. The ramifications of this difference are explained in Section 6.

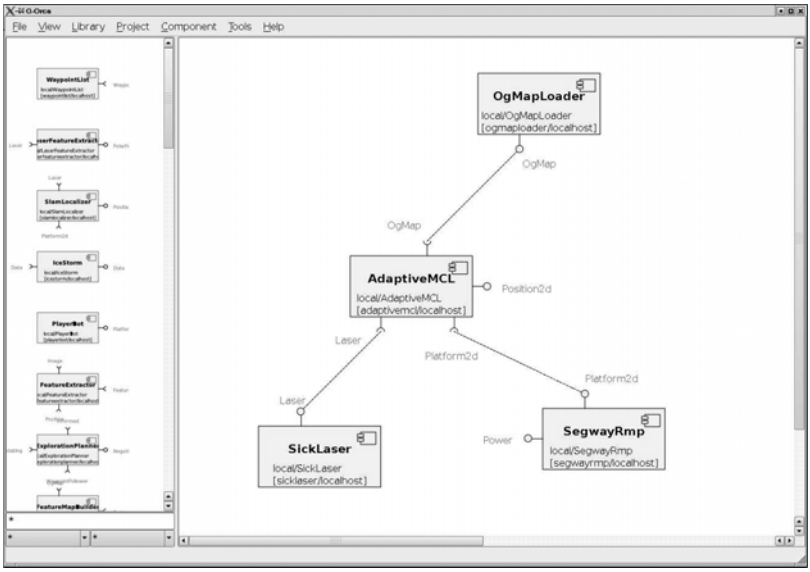


Fig. 1. `GOrca`: a graphical tool to assist component configuration. Components can be dragged from the library on the left to the projcet on the right and connected graphically. `GOrca` ensures that connections are valid. Right-clicking a component allows its configuration parameters to be modified.

To deal with this issue in Orca1, `orcad` was developed. `orcad` is a daemon which deploys all components on a host. It provides an Orca interface over which remote components can provide the names of components to be started or stopped on that host, along with the configuration files with which they should be started. It can then be queried, reporting the set of running components and the exit status of any which have died. This solved the problem of starting and stopping components and allowed all configuration files to be stored centrally.

Ice ships with a deployment service called `IceGrid`, which essentially solves the same problems as `orcad`: allowing a central component to start and stop components remotely. In addition it provides a standard solution for the binary synchronisation problem and reports statistics such as CPU, memory and network usage.

4.3 Remote Monitoring

As a system grew beyond a handful of components or hosts, the mean time between failures in the system became far smaller than the mean time between failures of the least reliable constituent component. In addition to algorithmic failures within components, errors occur due to both communication failures (which are not unexpected when dealing with mobile robots) and the interactions between components. Detecting and localizing faults was difficult.

Fault detection was problematic because, as individual component reliability increased, components would try to detect and recover from faults internally. While this behaviour is desirable, the result was that overall system performance would mysteriously degrade. Even when faults were detected, it could be difficult to determine which component was at fault or even on which host the problem was occurring. These problems were exacerbated by the introduction of `orcad` since trace statements directed to the console became unavailable.

Orca2 provides a remote logging API and a central monitoring component to monitor trace, heartbeat and error messages from any subset of a system (including the entirety). The level of detail required can be filtered on a per-component basis. In-keeping with the Orca approach presented in Section 2 it is not mandatory that components provide this remote logging interface, however the benefits are obvious and Orca2 provides libraries to simplify its implementation.

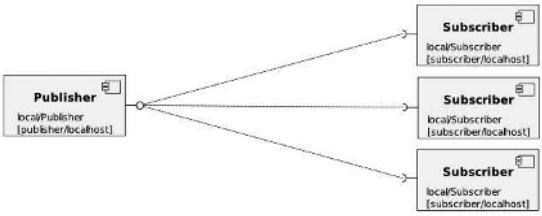
4.4 Event Service

A common cause of difficult-to-debug faults in larger systems was the coupling between publishers and subscribers. 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.

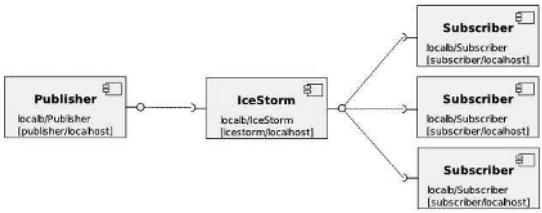
The problem of server-client coupling is solved in Orca2 using Ice's event service, called `IceStorm`. An event service is a stand-alone component which receives data from a single server and disseminates it to multiple clients, as illustrated in Figure 2. It relieves components of the burden of managing 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.

5 Results: Systems and Re-use

This section describes several systems implemented using Orca1. There is sufficient variation in system types to illustrate the flexibility of the project's philosophy and implementation. The description of the systems highlights the main objective of the project: software reuse. In addition to the highlighted reuse, all projects re-use common infrastructure such as graphical user interface components, `GOrca`, `orcad` and components for logging and replaying data streams and monitoring system status. Robotic platforms used in these systems are shown in Figure 3.



(a) Publish/subscribe without an Event Service



(b) Publish/subscribe with an Event Service

Fig. 2. The use of an event service such as `IceStorm` relieves the publishing component of the burden of managing subscribers and handling delivery failures. A multi-threaded event service weakens dependencies between clients.

5.1 Indoor Multi-vehicle Cooperative Exploration Research

Figure 4 shows the component diagram for a research project on cooperative indoor exploration [MDW06], using three Pioneer robots. Many of the high-level components are specific to the particular exploration approach, however the low-level hardware components, occupancy-grid server, obstacle avoidance algorithm, localisation algorithm and path plan executor are all re-used from other projects.

5.2 Industry-Funded Multi-vehicle Project

This project involves up to eight identical Segway RMP vehicles, where each robot's component diagram is as shown in Figure 5. The project was the source of many of the lessons from Section 4, since system reliability was paramount. The low-level hardware servers, navigation algorithms, map server and joystick interface are re-used between projects. A point of interest is the parallel use of two kinds of localisers: AdaptiveMCL, which uses a particle-based representation of uncertainty [Thr02] and SlamLocaliser, which uses a Gaussian representation [DNDW01]. The two localisers demonstrate the principle of replaceability: the PathPlanExecutor could be switched between the two if problems occurred with either one.

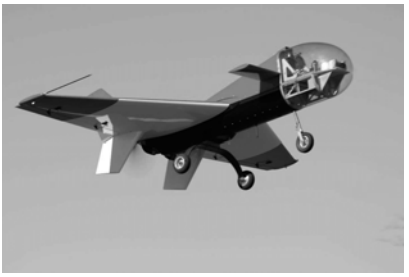
(a) A Pioneer indoor robot equipped with a SICK laser

(b) A Pioneer2AT equipped with two cameras and a SICK laser

(c) A Segway RMP

(d) A Pioneer indoor robot equipped with a camera and a SICK laser

(e) An Argo unmanned all-terrain ground vehicle



(f) A Brumby unmanned air vehicle

Fig. 3. Unmanned robotic platforms used in the systems described in this Section.

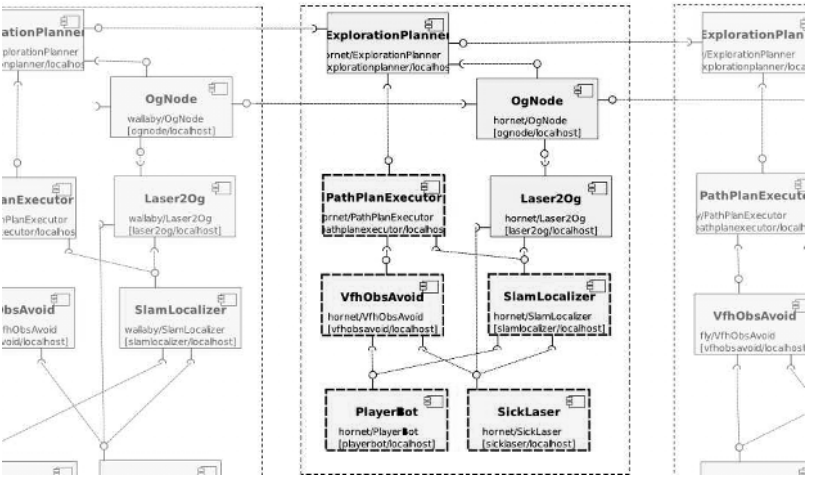


Fig. 4. Component diagram of one of three robots in a multi-vehicle cooperative exploration research project. Links leading left and right connect to other robots. In this and other system diagrams, highlighted components are those which were re-used across more than one project.

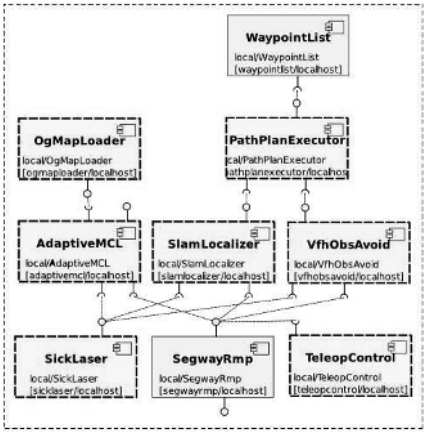


Fig. 5. Component diagram of a single SegwayRMP. The project involves up to eight identical copies of this diagram.

5.3 Single-Vehicle Research

The system in Figure 6 was used for research in sensor-centric localisation on a single Pioneer2AT, both indoors and outdoors [BUM06]. While system management and reliability of the low-level components was not so important for this project, it was helpful to be able to use many of the same components that were hardened by industrial projects.

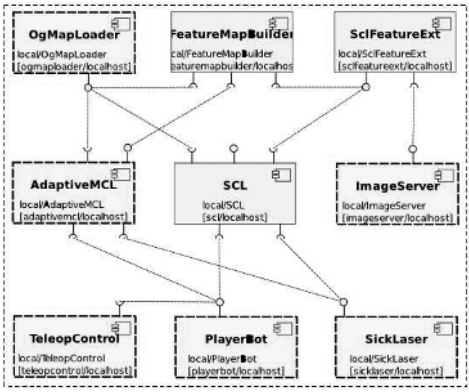


Fig. 6. Component diagram of the system deployed on a Pioneer2AT, used for single-vehicle research in sensor-centric localisation.

5.4 Large-Scale Multi-vehicle Outdoor Research

Figure 7 depicts the component diagram for the ANSER project: a large-scale outdoor decentralised data fusion [URO06] research project involving manned and unmanned ground vehicles, unmanned air vehicles and human operators. This is the most complex system to which Orca has been applied, highlighting many of the system scalability issues discussed in Section 4. The system is also interesting in that it incorporates air vehicles with strong real-time requirements. To accommodate this, the flight control software is written without Orca. To the rest of the system the air vehicles appear as 'black boxes' which provide their positions and observations over Orca interfaces.

6 Comparison with the Player Framework

Player [GVH03], originally designed and developed as a *robot device server*, has become the most successful mobile robotics re-use project to date. Its simplicity, reputation for reliability, good documentation and support from developers all contributed to its success. Player's design is similar to Netscape's

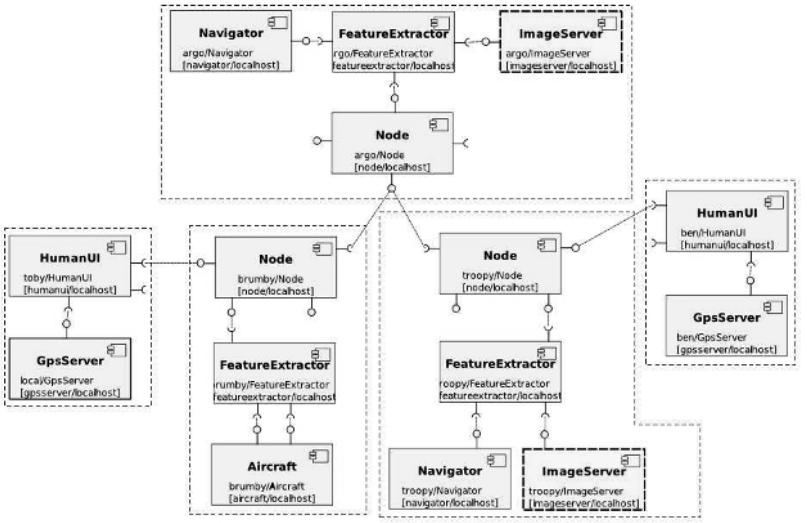


Fig. 7. Component diagram for the ANSER project. Dashed lines indicate the borders between physical platforms. The top-most platform is the Argo unmanned ground vehicle. From left to right, the lower boxes indicate a human operator, a Brumby unmanned air vehicle, a manned four-wheel drive vehicle and a second human operator.

plugin architecture, and is component-based in that sense: a monolithic server houses a set of modular devices. This has contributed to Player's successful re-use: devices conform to well-defined interfaces and correspond to well-understood algorithms or entities, allowing third parties to deploy and configure them without ever seeing the source code. When compared with Orca, Player's most important design decisions are its use of custom middleware and its delineation of server space from client space.

6.1 Middleware

Section 3 discussed both the central role of middleware for component-based frameworks, and the effort involved in implementing and maintaining the required functionality. We believe that it is unrealistic to expect robotics researchers to have the skills or time to develop middleware to the same standards as experienced middleware professionals working full-time on commercial products. By leveraging commercial middleware we expect Orca to provide more flexible and reliable communications infrastructure.

In an attempt to provide a more quantitative analysis, Figure 8 measures the number of lines of source code devoted to implementing middleware compared with the number devoted to actual algorithms and hardware

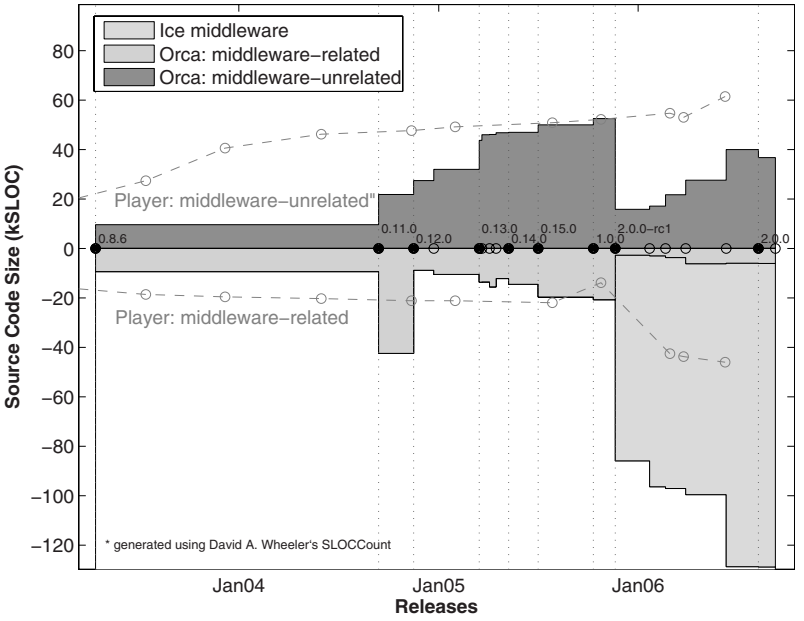


Fig. 8. A measure of the number of Source Lines of Code (SLOCs) devoted to implementing middleware-related (on the positive y-axis) and middleware-unrelated (on the negative y-axis) functionality, for various releases of both the Orca and Player projects. Orca adopted Ice for release 2.0.0-rc1 in late 2005.

drivers, for both Orca and Player[9]. Ideally, the amount of middleware-related code should remain constant and small, while the middleware-unrelated code should grow. Several points are apparent from Figure 8. Firstly, for both projects, the middleware-related effort is clearly significant. Secondly, for the Orca project, the adoption of a single commercial middleware product in late 2005 resulted in a dramatic reduction in the amount of middleware-related code written or maintained by robotics researchers. Finally, the total quantity of middleware-related code used by Orca2 is significantly larger than the quantity used by Player. We believe that this reflects the fact that Ice pro-

[9] Measurements were made using David A. Wheeler's SLOCCount. For recent Player releases, "middleware-related" counts the contents of "libplayercore", "client_libs", and "libplayertcp", while "middleware-unrelated" counts the contents of "server" and "utils". Automatically-generated bindings are not considered. For recent Orca releases, "middleware-related" counts the contents of "libOrcaIce", while "middleware-unrelated" counts the contents of the "utils" and "components" directories. The Ice measurement considers only the Ice features used by Orca, and only the C++ distribution. Full details can be found at http://orca-robotics.sf.net

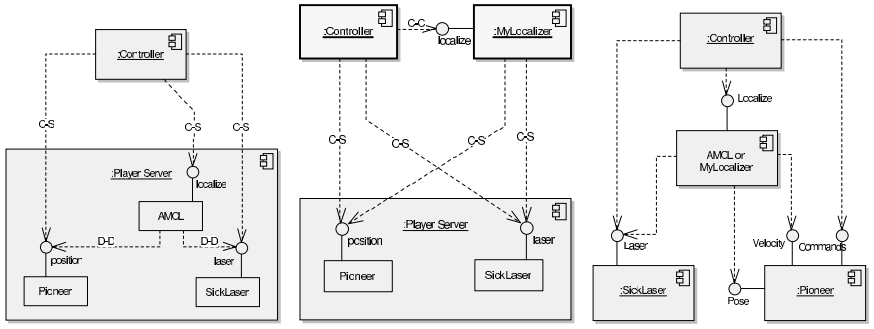
vides significantly more functionality and flexibility than Player middleware. This also provides some measure of the implementation effort which would be required for Player middleware to provide a similar level of functionality.

6.2 Client-Client Interaction

Player's delineation of server space from client space has several important consequences. Consider a single-robot system including components for hardware abstraction, localisation and control. Figure 9 shows how this might be implemented using either Orca or Player. There are several possible communication mechanisms in the Player model:

1. Inter-device communication, either within or between Player servers
2. Client-Server communication
3. Inter-client communication

Inter-client communication is not defined by Player, and must be supplied by users. As such, software modules that rely on custom inter-client communication mechanisms do not conform to any particular standards, and are not re-useable in the same way as Player devices are. To make the 'MyLocaliser' and 'Controller' modules independently re-useable under the Player model, MyLocaliser must be ported to server space. This requires significant effort,



(a) The setup using standard Player devices and a controller in client space.

(b) Using one's own localiser in client space requires the use of client-client communication.

(c) An Orca system: only one type of communication is required.

Fig. 9. Three possible implementations for a system with hardware servers, a localiser (either Player's Adaptive Monte Carlo Localiser (AMCL) or a hand-written localiser) and a controller, using either (a)-(b) Player or (c) Orca. The map-serving component (required for AMCL) is omitted for clarity. For Player, communication links are labelled Client-Server (C-S), Device-Device (D-D) or Client-Client (C-C). The use of non-standard Client-Client communication in (b) affects the re-usability of both the 'MyLocaliser' and 'Controller' implementations.

a possible re-design to fit the Player Device model, and code changes to the Controller.

Rather than developing in client space then porting to server space, one might develop directly in server space. In practice this doesn't often happen due to the extra complexity of conforming to the Player device model, and the problems of developing within a monolithic server where all components must be started and stopped simultaneously.

In contrast, there is only one inter-component communication mechanism in the Orca framework, and no distinction between server space and client space. This improves system flexibility: the localiser in Figure 9 can be replaced without requiring changes to other parts of the system. It also improves the ability of a component repository to grow: both the 'MyLocaliser' and 'Controller' modules can be added to an Orca-based component repository with no modifications. We expect this contrast to be most significant in systems containing modules which act both as clients and servers, as the localiser does in Figure 9.

In summary we hope that, by designing the framework to be general and flexible and by leveraging commercial-grade middleware, Orca will be more able to scale to large complex systems and to provide the reliability required for industrial systems.

7 Conclusion

This chapter discussed the practical application of CBSE techniques to mobile robotics. The Orca project provides the implementation of a component model and promotes the development of re-useable components for the robotics community. While simpler models may be sufficient for small projects, Orca addresses many of the issues that occur as system size increases. This allows the same components to be re-used in both simple and complex distributed systems.

References

[BFG97] R. Bonasso, R. Firby, E. Gat, D. Kortenkamp, D. Miller, and M. Slack, *Experiences with an architecture for intelligent, reactive agents*, Journal of Experimental and Theoretical AI (1997).

[BKM05] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, and A. Oreback, *Towards component-based robotics*, Proc. IEEE/RSJ Intl. Conference on Intelligent Robots and Systems, 2005.

[Ble01] D. Blevins, *Overview of the enterprise JavaBeans component model*, Component-based software engineering : putting the pieces together, Addison-Wesley, 2001.

[BMU05] A. Brooks, A. Makarenko, and B. Upcroft, *Gaussian process models for indoor and outdoor sensor-centric localisation*, submitted to International Journal of Computer Vision (2005).

[Bru01] H. Bruyninckx, *Open robot control software: the OROCOS project*, IEEE International Conference on Robotics and Automation (ICRA'01), vol. 3, 2001, pp. 2523–2528.

[Brug06] Brugali, D. and Brooks, A. and Cowley, A. and Côté, C. and Domnguez-Brito, A.C. and Létourneau, D. and Michaud, F. and Schlegel, C. *Trends in Component-Based Robotics*, In D.Brugali (Ed.) *Software Engineering for Experimental Robotics*, Spinger STAR series, 2006

[DNDW01] M. W. M. G. Dissanayake, P. M. Newman, H. F. Durrant-Whyte, S. Clark, and M. Csobra, *A solution to the simultaneous localization and map building (SLAM) problem*, IEEE Transactions on Robotic and Automation (2001), 229–241.

[Ewa01] T. Ewald, *Overview of COM+*, Component-based software engineering : putting the pieces together, Addison-Wesley, 2001.

[GVH03] B. Gerkey, R. Vaughan, and A. Howard, *The player/stage project: Tools for multi-robot and distributed sensor systems*, Proc. Intl. Conf. on Advanced Robotics, 2003, pp. 317–323.

[Hen04] M. Henning, *A new approach to object-oriented middleware*, IEEE Internet Computing **8** (2004), no. 1, 66–75.

[HV99] M. Henning and S. Vinoski, *Advanced CORBA programming with c++*, Addison-Wesley, 1999.

[MMDW06] G. Mathews, A. Makarenko, and H. Durrant-Whyte, *Information-based decentralised exploration for multiple robots*, Submitted to Proc. IEEE Intl. Conf. on Robotics and Automation, 2006.

[SB98] R. Sutton and A. Barto, *Reinforcement learning: An introduction*, MIT Press, Cambridge MA, 1998.

[Szy02] C. Szyperski, *Component software - beyond object-oriented programming*, Addison-Wesley / ACM Press, 2002.

[Thr02] S. Thrun, *Particle filters in robotics*, Proceedings of the 17th Annual Conference on Uncertainty in AI (UAI), 2002.

[UOK05] B. Upcroft, L. Ong, S. Kumar, M. Ridley, T. Bailey, S. Sukkarieh, and H. Durrant-Whyte, *Rich probabilistic representations for bearing only decentralised data fusion*, Proc. IEEE Intl. Conf. on Information Fusion, 2005.

[VAUG06] Vaughan, R.T. and Gerkey, B.P., *Reusable Robot Software and the Player/Stage Project*, In D.Brugali (Ed.) *Software Engineering for Experimental Robotics*, Spinger STAR series, 2006

[WSO01] N. Wang, D. Schmidt, and C. O'Ryan, *Overview of the CORBA component model*, Component-based software engineering : putting the pieces together, Addison-Wesley, 2001.

[MDW06] G. Mathews and H. Durrant-Whyte, *Scalable Decentralised Control for Multi-Platform Reconnaissance and Information Gathering Tasks*, Proc. Intl. Conf. on Information Fusion, 2006.

[BUM06] A. Brooks, B. Upcroft, and A. Makarenko, *Gaussian Process Models for Sensor-Centric Localisation*, Proc. IEEE Intl. Conf. on Robotics and Automation, 2006.

[URO06] B. Upcroft, M. Ridley, L.L. Ong, B. Douillard, T. Kaupp, S. Kumar, T. Bailey, F. Ramos, A. Makarenko, A. Brooks, S. Sukkarieh, and H.F. Durrant-Whyte, *Multilevel State Estimation in an Outdoor Decentralised Sensor Network*, 10th International Symposium on Experimental Robotics, 2006.