Designing a Distributed Real-time Software Framework for Robotics

Yuan-hsin Kuo and Bruce A. MacDonald
ECE Department, University of Auckland, New Zealand
ykuo007 *at* ec.auckland.ac.nz B.MacDonald *at* auckland.ac.nz

Abstract

A distributed real–time robot application framework is designed, to improve the scalability and reusability of software modules. The design is based on real–time CORBA and structured into two layers: the infrastructure layer for basic functionality and the service layer, which includes several reusable services.

1 Introduction

Many robots are equipped with advanced, powerful computing hardware. This enables many new possibilities including distributed robotic applications, since overhead and latency are no longer significant. By deploying a distributed architecture, roboticists can loosen the coupling between software components such as device drivers, algorithms and user interfaces. This type of architecture is superior to a monolithic one in terms of reusability, scalability and availability. Many recent robotic software architectures adopt a distributed design.

The Player/Stage project [Vaughan *et al.*, 2003] is a popular software library, which provides a collection of drivers for many popular robotic devices. It provides a network-oriented device server along with 2D and 3D simulators that are capable of simulating a population of mobile robots. IPC (Inter Process Communication) along with TDL (Task Description Language) are two software packages developed at CMU [Simmons and Apfelbaum, 2004]. TDL extends the standard C++ syntax to provide semantics for task management and it uses IPC for sending messages between servers and clients. CARMEN [Montemerlo *et al.*, 2002], also based on IPC, provides a navigation toolkit for robotics. SIMOO-RT [Becker and Pereira, 2002] is an integrated development environment for real-time systems consisting of a software development framework and model editing tools. CLARAty is an architecture designed for robot automation systems from the Jet Propulsion Laboratory [Nesnas *et al.*, 2003]. It uses a two-tiered design to separate functional and decision layers of a robot application.

The research projects above create custom middleware systems to enable interaction between distributed components. Other projects base their work on existing high-level middleware such as CORBA. MIRO [Utz *et al.*, 2002] is a distributed object-oriented framework for mobile robot application developments. It is based on CORBA and therefore it is also a multi-platform and programming language independent development system. Woo proposed a three-tier software infrastructure [Woo, 2002; Woo *et al.*, 2003] based on CORBA. It is designed around the CORBA Trader Service and relies on CORBA to provide the underlying development environment. OROCOS [OROCOS] is designed to be a software framework for robotic control software. It aims to integrate low-level real-time services provided by operating systems (OSs) and provides a high-level software framework for robotic control software development.

Although high-level middleware can greatly simplify software development, its efficiency is a major concern for roboticists. Robotic applications are different from most desktop applications in that they interact with physical objects, so their actions have stricter timing constraints. If the timing of a particular control signal is wrong, a robot can cause damage to itself or even to humans. Distributed robotic applications must achieve real-time performance at two different levels, the OS level and the middleware level. An OS must appropriately schedule tasks according to their priorities in order to ensure the overall quality of services. Middleware must provide constructs to interface and integrate the real-time features provided by OSs in order to ensure distributed processes are executed at desired priorities.

The next section gives our motivation. Section 2 states design goals. Section 3 explores related work. Section 4 examines middleware technology. Section 5 explains the design. Section 6 evaluates the functionality.

1.1 Project motivation

A key requirement in promotion of software reuse is to loosen the coupling between software modules. Therefore, our environment is designed to support distributed

software development, which automatically makes components more loosely coupled [Szyperski *et al.*, 2002]. It will contain distributed software components that are designed specifically for supporting robotic application developments such as device drivers, development tools, algorithms, control systems and the human-robot interaction component. An important player in this environment is the distributed infrastructure, which manages the communication between these components.

Distributed infrastructure is essentially a concept that covers a collection of network-oriented software subsystems provided by OSs, such as the network device driver and networking protocols. Programmers access this using network application programming interfaces (APIs) provided by the underlying OS. However, these APIs vary across different OSs and not all provide features for the real-time application development needed in robotics. This research project was commissioned to design a software framework to solve these issues and it will serve as the backbone infrastructure for future development projects.

Most middleware technologies used by roboticists today are designed specifically for enterprise software systems, including the CORBA system, the .NET framework and Java Remote Method Invocation (RMI). Their designs do not contain the necessary features for fine-grained control of system resources in order to achieve real-time performance. Therefore, the main objective of this paper is to describe a software framework that is designed around industrial strength real-time middleware.

2 Design Goals

The main purpose of designing a robot application framework is to reduce the time and complexity of the development task. However, there are many other usability requirements that are worth considering for such a framework. The goals of the framework are:

Platform independence so users can choose an appropriate platform for running the applications.

Enhanced Scalability. Support constructs should enhance the scalability of software solutions.

Development Process Simplification. High-level concepts, useful facilities and simplified programming interfaces should support robot program developers.

Real-time Performance to make the software framework usable for robotic applications with soft or hard real-time requirements.

Integration with Existing Infrastructure to allow applications to integrate with the existing functionality provided by underlying distributed middleware.

Promotion for Software Reuse should support object-oriented concepts and provide predefined object interfaces so that modules built on top of the framework have a common ground for interaction.

Programming Language Independence so that application developers will be able to choose an appropriate programming language.

3 Related work

Player/Stage, MIRO, Woo's structure and SIMOO-RT were selected for more detailed comparison with our goals, because of their availability and relevance. Only SIMOO-RT includes real-time features; however it is not portable across programming languages and platforms.

3.1 The Player and Stage Project

Player/Stage has a network oriented device server *Player* and a robot simulator *Stage*. The Player server is a distributed device repository server that provides clients with network-oriented programming interfaces to access actuators and sensors of a robot. Client programs use proxy objects defined in the Player client library to write and read data to and from the desired device.

Player relies on the low-level TCP transportation protocol to handle communication between clients and servers. Player hides low-level network mechanisms and partially fulfils the requirement for development process simplification. However, when programmers add new device types to the Player server a corresponding client side proxy interface must also be manually added. This process is difficult for programmers who do not have a detailed understanding of the Player protocol and architecture. So Player does not satisfy the requirement for enhancing the scalability.

The strict client/server relationship in the Player architecture imposes a few difficulties in designing reusable software components that are integrated with its infrastructure. This contravenes the requirement for integration with existing infrastructure.

On the client side, Player does not promote separation between reusable control algorithms and the main program flow. Programmers must implement their own abstractions such as applying design patterns [Gamma *et al.*, 1994], in order to build algorithms as reusable modules. So software modules implemented on the client side do not adhere to Player's infrastructure so they cannot be reused by other distributed clients [Szyperski *et al.*, 2002]. If programmers want to implement reusable algorithm components that are integrated into Player's architecture, they must implement them on the server side. These algorithms must be implemented as virtual devices and incorporated into Player's server. For example, a navigation algorithm can be implemented as a virtual positional device driver that wraps around actual device driver components.

The Player library satisfies the requirement for programming language independence reasonably well, since Player includes client-side utilities in C++, Tcl, Java and Python. It is based on an open standard and requires only a TCP/IP library. However, the Player server and its API are written in C++ and do not provide direct support for developing drivers in other languages. So algorithms implemented in other programming languages must find their own way to link with the C++ driver API provided by Player, such as using the Java Native Interface (JNI) or Python-C API. So the server side Player library does not meet the programming language independence requirement.

Unlike many proprietary software development kits shipped with robots, Player does not bind to a particular robot system. It has a modular design; programmers can add support for new hardware devices. Player was developed under the Linux OS; however, it also works on other UNIX variants such as Solaris and FreeBSD that support TCP socket mechanisms. The problem with relying on the UNIX socket APIs is that they are not available on other OSs such as Microsoft Windows. This violates the platform independence requirement.

The overall performance for the Player library is more efficient than other complex solutions based on middleware since it interfaces directly to the low-level TCP socket mechanisms. It is probably the most efficient solution in terms of latency and memory usage. It is the most popular framework. However, it was not designed to fulfil real-time requirements.

3.2 MIRO

Components of MIRO are organised into three layers. The device layer consists of object-oriented interfaces for robot sensors and actuators. The MIRO service layer maps these devices onto distributed objects that can be invoked across the network. Finally, the MIRO class framework provides reusable components such as path-planning and localisation.

MIRO has several advantages over Player in terms of software reuse, since programming interfaces for sensors and actuators are exported as CORBA objects and unlike Player, it does not enforce a client/server relationship between device objects and clients. Firstly, programming interfaces of sensors and actuators are defined in CORBA Interface Definition Language (IDL). This makes it simple to integrate and reuse existing device interface software, which might be implemented in many different programming languages. Secondly, CORBA is an object-oriented middleware which provides many high-level concepts and abstractions compared to Player's socket communication model. So MIRO fulfils the requirements for promotion of software reuse, as well as platform and programming language independence by selecting appropriate technology as a base. Definitions of new types of server side object are done in IDL in MIRO, which is much easier than in Player because CORBA handles much of the work.

MIRO has been tested on several robotic control projects. It is reported that the overhead of CORBA services is relatively insignificant compared to delays introduced by most commercially available robot hardware controllers, where response times are dominated by the latency of the controller. Although MIRO does not provide real-time features to handle predictability.

3.3 Woo's Three-tier Infrastructure

Woo [Woo, 2002; Woo *et al.*, 2003] proposes an infrastructure for CORBA based, distributed robotic applications. The infrastructure has three layers. The application layer consists of user level applications such as the navigation module. The CORBA service layer consists of the CORBA Naming and Trading services that provide for service discovery and advertisement. The middleware layer is the core CORBA system, which manages the communication and configuration.

The service-based architecture can loosen the coupling between client and server because a client can reuse a server without caring about the implementation details.

Since Woo's infrastructure is also based on CORBA, it shares many architectural characteristics that fulfil requirements for programming language independence and development process simplification. However, reusable abstract programming interfaces are not defined, for developers to follow or extend. If a framework does not provide a fixed infrastructure for developers to follow, then independent extensions cannot interoperate, and reusable software development will not be encouraged.

The requirements for platform and programming language independence are achieved by using CORBA. Woo did several performance tests using raw CORBA in both Java and C++, showing that the overheads introduced by CORBA are relatively small. However, as with MIRO and Player it does not support the real-time communication model, so it does not fulfil the requirement for real-time performance.

3.4 SIMOO-RT

SIMOO-RT is an integrated development environment designed to support the whole development cycle of real-time industrial automation systems. It consists of a model-editing tool (SIMOO-RT MET) and an object-oriented software development framework (SIMOO-RT). It is based on the Active-Objects/C++ programming language, which provides an RMI based communication model and the message publishing model (publisher/subscriber communication scheme). This extensive support for a distributed computing model makes

a SIMOO-RT solution both scalable and reusable. The integrated modelling tool makes it is easy to build well-designed reusable software modules.

SIMOO-RT [Becker and Pereira, 2002] supports some real-time features such as association of a time constraint with requests, and fault tolerance mechanisms. The framework provides support for two types of timing requirement: periodicity and deadlines. SIMOO-RT allows the programmer to define actions to be taken when a specified deadline is violated. So SIMOO-RT satisfies the real-time performance requirement.

SIMOO-RT adds some primitives to C++ in order to support various real-time features. It does not have bindings to other programming languages so SIMOO-RT was not designed to support multiple programming languages. Furthermore, it only runs on a limited number of UNIX-like platforms such as QNX and Linux, which violates the requirements for programming language and platform independence.

4 Distributed Middleware Technology

The design of our software framework is based on existing middleware instead of developing proprietary solutions, for several reasons. The development of fully featured middleware is a complex task. Multiple OSs and different operational contexts must be supported; many existing middleware technologies already achieve this goal. Rapidly evolving technology trends and the cost of debugging makes maintaining proprietary middleware systems a difficult task [Schmidt and Vinoski, 2001]. Many existing middleware systems already have components and services that are powerful and generic. A custom middleware system cannot easily integrate with the functionality of existing middleware systems.

4.1 Evaluation of existing middleware

Several existing middleware technologies such as CORBA, Microsoft .NET, IBM SOM, ICE, SOAP, RTC and Sun's Java/RMI were evaluated. These distributed middleware technologies have similar functionality but target different types of application.

The simplest form of middleware is message oriented middleware (MOM) [Woo, 2002; Gomaa, 2000]. Most MOM technologies are only a thin software layer on top of the raw communication model provided by the underlying OS. MOM takes data given by programmers and creates messages by adding headers for describing data sizes and types. After messages are created, MOM then forwards them to the client side.

Remote Procedure Call (RPC) and Object Request Broker (ORB) style middleware provide higher levels of abstraction than MOM [Gomaa, 2000]. RPC allows functions to be invoked across the network. ORB style middleware extends RPC by integrating object-oriented software concepts. Compared to ORB, RPC style middleware is less popular and usually provides fewer facilities for solving distributed computing problems

ORB middleware can promote interoperability between research projects because it enables users to build distributed systems by placing together objects from different developers and environments.

4.2 CORBA and Real-time CORBA

CORBA is an ORB style middleware that was designed to replace old RPC. Each CORBA object server implements some interface defined in IDL. Interfaces defined in IDL serve as contracts between distributed clients and servers in a CORBA system. IDL definitions are independent of programming languages; CORBA vendors usually provide utilities that convert an IDL file into supported programming language mappings. Currently, CORBA language specifications are available for C, C++, Ada, Python, Perl and Java.

In addition to the low-level transportation specifications, CORBA also has a set of high-level services such as the Trader Service and Naming Service.

The real-time CORBA specification [OMG, 2002] extends the core CORBA model to support real-time constructs. Currently real-time CORBA implementations are available for Java and C++. Real-time CORBA specifies programming interfaces that allow applications to configure and control processor, communication and memory resources. Real-time CORBA adds programming interfaces to access low-level schedulers, communication resources and thread managers of the target OS. It is possible to achieve a similar effect in classic CORBA by sending priority values through CORBA method invocations and manually invoke native OS APIs. However, this will not be portable across different OSs. Furthermore, the underlying CORBA machinery is not aware of the priorities. Real-time CORBA solves these issues by integrating priority models into the CORBA protocol and introducing several other mechanisms to enhance the predictability of a CORBA system.

Priority assignments

Real-time CORBA allows applications to configure OS schedulers via priority assignments. There are two different priority models supported in real-time CORBA: the Server-Declared and Client-Propagated model [Schmidt and Vinoski, 2002a].

The *Server Declared* Priority model allows services to be activated at certain priorities. For example, two servants may be activated in two different thread contexts running at high and low priorities and invocations made by clients will execute at the target service's execution priority.

In the *Client Propagated* model the execution priority of a request is transmitted with the request made by

the client. For example, there may be one servant running at the server side that is activated with the client-propagated priority assignment policy. Supposing the client side has three processes running in two different priorities. When they make requests to the service; their requests are executed in different priorities according to their own priority.

Thread Pools and Synchronisers

In classic CORBA systems, there are no guarantees on how requests will be processed by the ORBs. Some ORBs allocate a thread for each request and this can result in an excessive number of threads. When the number of requests is large, this concurrency model consumes a large amount of system resources and may drag down the system performance as the computer spends more time switching between threads. To solve this problem, real-time CORBA provides thread-pooling, which is a technique to ensure client requests are processed within a bounded amount of CPU and memory resource [Schmidt and Vinoski, 2002b].

Different priority thread-pools in real-time CORBA can be associated with multiple Portable Object Adaptors (POAs). As requests arrive, the POAs grab threads out of the appropriate thread-pool. If all threads are occupied in a thread-pool, newly arrived requests will be buffered until there are free ones available to process it.

Each priority group is allocated with a number of threads and high-priority groups can borrow unused threads from low-priority ones.

Protocol Selection and Explicit Binding

Real-time CORBA allows configuration with different communication protocols. OMG specifies only TCP/IP at present. Real-time CORBA supports connection pre-establishment where clients can control connections to servers in order to minimise the latency on initiating connections; normally CORBA establishes connections on demand. Real-time CORBA also introduces the Priority-Band communication model, allowing applications to establish special private channels for services running at different priority ranges. This improves the overall determinism [Schmidt and Vinoski, 2002c].

4.3 Java, RMI, Distributed Real-time Java

Sun Microsystems first released Java in 1995. Java is now a complete platform for developing software solutions. The standard edition of Java contains a CORBA implementation as well as its own API for distributed computing. Java RMI is also an ORB-style middleware, it uses stub and skeleton objects to hide communication details and to allow methods to be invoked across the network. The RMI Registry is similar to the naming service of CORBA, allowing objects to be looked up by name. Java RMI introduces distributed garbage collection to automate much object lifecycle management. However Java RMI does not support interoperation with other programming languages. It is possible to achieve language interoperability through a combination of Java RMI and Java native interface (JNI). Nevertheless, it is not as straight forward as in CORBA where there is native support for multiple languages.

In order to extend the interoperability between Java RMI and CORBA, Sun and IBM developed the RMI over IIOP mechanism, which allows developers to design and implement remote interfaces using the native Java programming language syntax rather than IDL.

Ordinary Java RMI does not provide real-time features as required for real-time applications. The distributed real-time Java specification is a project to add read-time capability to Java's native RMI model. It introduces extensions to raw RMI to facilitate distributed real-time programming in Java [Jensen, 2000]. It naturally inherits all interesting features of Java such as platform independence and Java's dynamic nature. However, this project is still in the early stage of development and only a primitive initial specification is available.

4.4 SOAP

Simple Object Access Protocol (SOAP) [Seely, 2002; W3C, 2003] is an object-oriented distributed middleware designed to reuse the existing Internet infrastructure. Its predecessor is XML-RPC, an XML (eXtensive Markup Language) version of RPC. As with other middleware, SOAP provides object-oriented abstractions to the underlying protocol usage. While other middleware such as CORBA use their own application protocol, SOAP reuses existing ones such as Hyper Text Transport Protocol (HTTP), File Transport Protocol (FTP) and Simple Mail Transport Protocol (SMTP). Most SOAP implementations use HTTP.

The SOAP specification is open; it can be implemented on any platform for any programming language. To name a few, Java, C++, Allegro Lisp, Python and Perl have implementations of SOAP and most are available on Linux, FreeBSD and Windows. On the down side, SOAP is often criticised for its performance. XML is used to describe method invocation. XML documents are much larger in size than binary protocols such as CORBA's IIOP, and so it takes longer to transport and process. So SOAP might not be useful for resource constrained embedded systems such as small robots. Also, SOAP was not designed with real-time performance in mind. Therefore, there are no built-in features to ensure the quality of service for a SOAP server.

4.5 COM, DCOM and .NET

COM (Component Object Model) is used in many Microsoft software products [Szyperski *et al.*, 2002]. An

important concept introduced by COM is to use object aggregation over inheritance. COM avoids multiple inheritance altogether by supporting multiple interfaces in terms of object aggregation. This concept has great influence on many open-source software projects such as the XPCOM framework from [Mozilla] and the UNO component model from [OpenOffice.org]. Distributed COM (DCOM) transparently expands the concepts of COM to a distributed manner by using stub and proxy objects to handle inter-process communication.

The .NET framework is a new generation of distributed component technology designed by Microsoft. It includes a language-neutral development platform (CLI), which is similar to CORBA. In addition, it also defines an intermediate language format, much like Java's byte code system, to ensure a compiled program can be executed on any platform where an implementation of the .NET platform is available.

There are open-source projects such as Mono [Schönig and Geschwinde, 2003] and DotGNU [DotGNU] trying to implement compatible .NET frameworks on Unix platforms. Mono did implement proprietary components of .NET framework in order to be compatible with Microsoft's version of .NET.

4.6 RTAI and RTC

Real-Time Application Interface (RTAI) [Dozio and Mantegazza, 2003] is an open-source software project to add real-time capabilities to the standard Linux kernel. It can be installed by patching a standard Linux kernel.

RTC [Heimfarth *et al.*, 2003] is a MOM for real-time communication on top of the RTAI. It provides functions for sending and receiving messages. Since it is tightly coupled with RTAI, it is only available on the Linux OS. RTC had been evaluated under several different system loads. The evaluation result shows a maximum delay of 2.57 *ms* and minimum delay of 2.52 *ms* when sending 1 Kb of data, independent of the configured load.

4.7 Middleware Evaluation

All middleware technologies introduced above support location and protocol transparent communication. They each provide a higher level message format specification and hide the low-level transmission protocol details from programmers' point of view. This fulfils the development process simplification requirement. While RTC is a MOM, others such as Java RMI, Microsoft .NET and CORBA are ORB style middleware that support a higher level object-oriented abstraction, which satisfies the requirement for promotion of software reuse. These abstractions are important factors for supporting reusable software development.

Java RMI does not natively support multiple languages so Microsoft .NET, CORBA and SOAP are clear

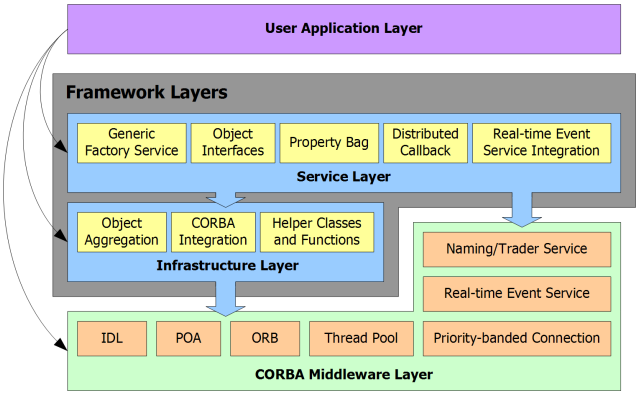

Figure 1: Framework Layers

winners. Specifications for CORBA and SOAP are open and can be implemented on any platform so both of them meet the requirement for platform independence. Finally, CORBA contains the real-time CORBA specification, which provides support for features such as the thread-pooling and priority assignment so it also satisfies real-time performance requirement.

As a result, CORBA along with its real-time specification was chosen for implementing our framework since: a) it is independent of language and platform, b) it is capable of integrating with real-time OSs, and c) it provides high-level object-oriented abstractions for low-level communication concepts. CORBA does however present some risks to reusability and to the development process. It is a specification and users much ensure compatibility between ORB implementations. It has many features and the learning curve is steep. It is not a binary level interface and the overheads in performance and programming are higher than other technologies. Thus it is important to develop a useable CORBA based framework.

5 Design

The architecture of the framework is split into two layers (see Fig. 1). The infrastructure layer contains classes and functions that are a foundation both for services in the service layer and for user applications to build upon. The service layer provides reusable high-level services, which users can deploy in their applications. A programmer can interact directly with the infrastructure layer to create new services as well as combining existing services provided in the service layer into an application.

A framework could be implemented as a wrapper, presenting an API and hiding the distributed software infrastructure from the application programming. Alternatively, the framework could present services, allowing the application programmer to have access to the underlying infrastructure. The service based approach was chosen because it makes the system more extensible by

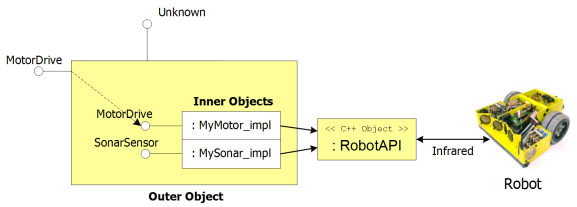

Figure 2: Object–aggregation example

not limiting the access for extension to "hot spots," as the wrapper method does [Fayad *et al.*, 1999]. The functionality of the framework may mix with both CORBA services and newly defined object interfaces, making the system more scalable. The main disadvantage is that programming is a little more complex. Helper classes and functions are included, to counter this problem.

5.1 The Infrastructure Layer

Currently, the most widely used technique for enhancing software reusability is the *Interface-based Programming* technique [Szyperski *et al.*, 2002; Löwy, 2003], which is available in most distributed component-oriented middleware such as CORBA, Microsoft .NET and Java/RMI. Interfaces of services are separated from their implementations. The interfaces become contracts between clients and services; changes that happen in service implementations will not affect clients.

A service in component-oriented middleware usually exposes multiple interfaces to clients. Traditionally this is achieved by using the multiple inheritance feature of CORBA IDL, where an implementation of a service inherits from multiple predefined interfaces. However this suffers from problems associated with multiple inheritance such as method name clashes where methods with the same name have different meanings in parent classes [Szyperski *et al.*, 2002].

This framework uses the *object-aggregation* concept, as used by UNO [OpenOffice.org], Bonobo [GNOME], COM and XPCOM [Mozilla, 2003]. The object-aggregation technique achieves a similar effect while avoiding the problems of multiple inheritance [Szyperski *et al.*, 2002]. Unlike multiple inheritance, implementations of interfaces using object–aggregation are stored as *inner interfaces* that are aggregated inside the *outer interface*. The inner interfaces are revealed by clients' queries. Fig. 2 illustrates the runtime collaboration where instances of two classes are aggregated together and initially the aggregated object exposes the **MotorDrive** interface as its outer interface, but can morph to expose the **SonarSensor** interface upon request. This has much the same effect as multiple inheritance but avoids difficulties such as method name classes.

This layer also provides facilities to integrate and simplify the real-time CORBA API. This covers the *Server-Declared/Client-Propagate Priority Assignment Model*, *Priority-banded Connection* and *Threadpool* components of real-time CORBA.

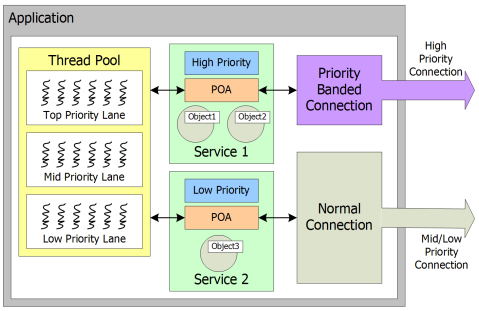

Figure 3: Allocation of POAs

Server

```
// initialise the system, passing true (DAFRA::T) for the last
// argument means that this is a server application
DAFRA::initialise (argc, argv, DAFRA::T);

// creates two device interface objects
SonarSensor_impl aSonarSensor;
MotorDriver_impl aMotorDriver;

// aggregate motor driver object using object aggregation
aSonarSensor.add_interface (&aMotorDriver);

// activates this aggregated object
aSonarSensor.activate (DAFRA::TOP_PRIORITY);

// advertise this aggregated object by customised function
my_advertise (aSonarSensor);

// start the main loop of this application
DAFRA::main_loop ();
```

Client

```
// initiate the client application
DAFRA::initialise (argc, argv, DAFRA::F);

// get object referernce from somewhere
CORBA::Object_var MyObj = get_object_reference()

// down-casting the object reference string to desired types
SonarSensor_var aSonarSensor =
DAFRA::unknown_cast <SonarSensor>(MyObj);
MotorDriver_var aMotorDriver =
DAFRA::unknown_cast <MotorDriver>(MyObj);

// do some work with these two object references
// (e.g. an obstacle avoidance algorithm)
```

Figure 4: Example server and client code.

The design simplifies and somewhat restricts the configuration of CORBA services, by limiting support to only create, activate, advertise, and configure (as do Player/Stage and MIRO). The task of allocating Portable Object Adaptors (POAs) is automated for the programmer, as illustrated in Fig. 3. A POA is allocated for each service. Other simplifications to the programming are: limiting priority values to one of three, and hiding real time CORBA configuration. Helper functions and templates are also provided, for automating the framework adaptation, for accessing CORBA functionality, and down casting object references. More advanced CORBA functionality is moved to the service layer. Fig. 4 shows the crucial parts of the code for an example server and client.

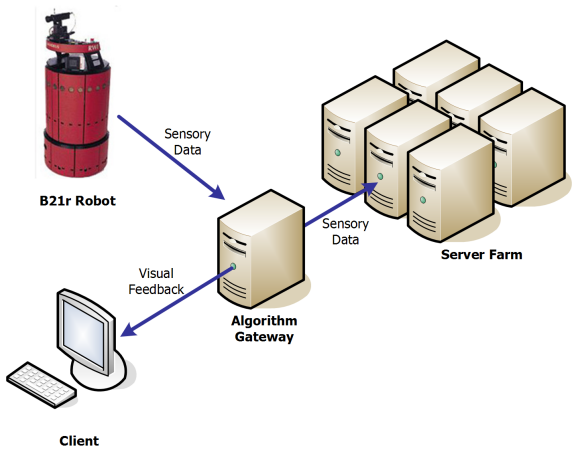

Figure 5: Server Farm

5.2 The Service Layer

This layer introduces several reusable generic components. They rely on the infrastructure layer as the foundation and provide distributed services for their clients. The main motivation behind the creation of these services is to provide reusable facilities, which solve issues that frequently occur in robotic applications.

Generic Factory Service

Robotic applications frequently need large computation power for handling complex obstacle avoidance or image processing algorithms. One common solution for this requirement is to deploy a server farm [Löwy, 2003] environment (see Fig. 5). CORBA services are used to manage such a farm, rather than creating a CORBA interface to an existing clustering system such as OpenMosix or Beowulf. This framework introduces the generic factory service to support distributed processing in order to fulfil the requirement for enhancing the scalability.

Two types of servant give services from the farm. The single call servant serves just one call and is removed. The removable servant lives while clients require it. It outlives the client call, enabling a number of clients to request services, until a client removes it. This is useful for implementing finite state machines, for example.

Distributed Callback

Frequently robotic applications need event-based communication where clients must be notified on a change of the server's state. Sonar and other range sensors are good candidates for using this service. The client need not continually query (ie poll) the sensors, because the sensor service will notify the client when new data is available. The service layer of our framework provides a distributed callback service for this purpose. This feature fulfils the requirement for development process simplification by providing ready-to-use facilities tailored specifically for robot software development. The structure of distributed callback is essentially a distributed version of the *Observer* pattern as explained in the GoF design pattern book [Gamma *et al.*, 1994].

There are two main uses of the distributed callback component, the event-source and event-sink. Event sources are objects that trigger events and notify their observers. The observers are event sinks that register themselves to event sources in order to receive events.

Publisher

Although the distributed call-back component satisfies the requirement for development process simplification, it has several drawbacks that cause scalability problems:

One-way methods are not always one-way The implementation of call-back uses CORBA's "one-way" keyword for the asynchronous update method of the event sink. However the CORBA specification allows implementations to bypass the "oneway" keyword and make all one-way methods synchronous. This can become a serious performance impact when the number of observing clients grows.

Scalability issues for large numbers of clients The distributed call-back works fine for small numbers of clients. However, as the number of clients grows, the server may spend much CPU time processing event sinks and not be able to perform the task that it was originally assigned.

In order to support scalable event-based communication, this framework introduces a Publisher service that integrates the CORBA real-time Event Services. This service is useful for large-scale robotic systems.

The CORBA real-time Event service provides two communication models, the push model and the pull model. To simplify the programming interface, this framework only supports the push-model since it is more commonly used in event-driven applications.

The real-time Event service extends the original CORBA Event Service with real-time capabilities and filters. The supplier of an event must provide two IDs: the source ID and the target ID for each of the events it provides. The real-time Event service does not specify how these IDs should be generated and distributed to consumers. For ID generation, programmers usually have to use customised algorithms for generating the two types of ID. However, this process is error-prone because programmers may generate the same ID for different events. Also, distributing IDs can be a difficult task as there are no standard methods. To solve these problems, this framework introduces the Publisher service to handle the generation and distribution of IDs.

Object Interfaces

The framework defines a set of object interfaces that are intended to become the foundation for extensions for interaction as required for promoting software reuse. These object interfaces are defined in CORBA IDL to achieve platform and language independence [Utz *et al.*, 2002]. Object interfaces declare method templates that serve as contracts between services and their users. These method templates are overridden by service implementations to provide actual behaviours. The architecture for the set of object interfaces follows the object aggregation concept. All object interfaces solely inherit from the `Unknown` interface so they can be aggregated with other objects in the framework. For example, a laser scanner service can aggregate with a distributed callback component so that clients can be notified when new range readings are available in the laser scanner.

Property Bag Component

A problem arises when interfacing robotic hardware systems to software; hardware devices can have product-specific extensions. For example, while all laser scanners will allow programmers to fetch range readings, not all of them will give programmers the ability to control the scanning frequency. Other frameworks such as Player and MIRO define comprehensive software interfaces for hardware devices so that they can cover all possible attributes. However, some hardware can have strange, unpredictable parameters. Programmers will then have to add a new programming interface to describe the hardware.

There are two issues related to this approach. Firstly, whenever new interfaces are defined, they must be distributed system-wide; otherwise other services would not know how to communicate to these new extensions. This requires bringing these service processes down, recompiling them to add support for new interfaces and reactivating them. Sometimes, a service might contain useful control algorithms that are too important to be brought down. This makes implementations of these new interfaces inaccessible by these services. Secondly, defining too many object interfaces in the framework may hurt the reusability. Programmers may be tempted to create new versions of existing classes just because new devices have different parameters.

The framework introduces a property bag service to reduce the number of new interfaces that need to be defined when adding new functionality. Each property is a name-value pair that can be updated dynamically to the bag. This feature is inspired by the Bonobo [GNOME] framework. Because properties are dynamic, adding new features does not invalidate the object interface for the property bag component and therefore a client does not need to know any new interface. For example a laser scanner object might be an aggregation of a range sensor object and a laser scanner properties bag, enabling new kinds of laser scanner objects with different properties to be added on the fly, without the need for additional laser objects to be added. This is essentially a realisation of the *Strategy* design pattern [Gamma *et al.*, 1994].

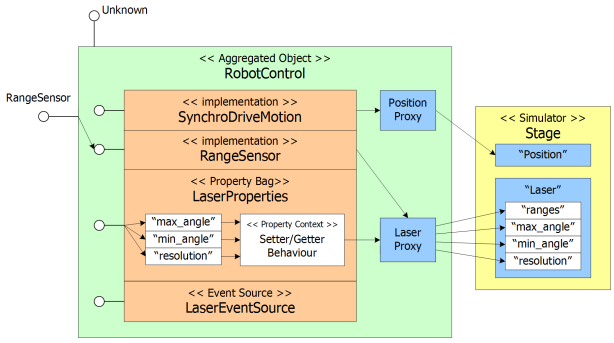

Figure 6: Server Architecture of the Example

6 Functionality Evaluation

Fig. 6 shows an example of the framework at work. A server is responsible for interfacing a robot in the Stage simulator to the CORBA infrastructure through Player Client libraries. The client uses a simple obstacle avoidance algorithm to control the robot.

The server aggregates implementations of two predefined IDL interfaces, *SynchroDriveMotion* and *RangeSensor*. They describe motor and sensor interfaces of a robot in the Stage simulator. Attributes such as the control of resolution, maximum and minimum scanning angle are stored in *LaserProperties*, which is an instance of the Property Bag service. An instance of the Event Source service is also aggregated so clients can register themselves to be notified of new laser scanner data. A client would get object references to the server service and use helper functions to retrieve references from the aggregated service. It then gets laser scanner readings from the reference to *RangeSensor*, computes the desired movement using a simple obstacle avoidance algorithm and sends movement commands back using the reference to *SynchroDriverMotion*. It can attach itself to *LaserEventSource* to receive updates for the laser scanner readings rather than querying.

The framework principally supports multiple programming languages and platforms. The framework uses CORBA to hide low-level network location and protocol details and provide mechanisms to further simplify its programming interface so it satisfies the requirement for simplification of the development process. To handle large-scale application development, this framework introduces the generic factory service to allow services to be deployed in server farm environments. Furthermore,

this framework provides features to support reusable software development. Finally this functionality integrates nicely with the CORBA system so it fulfils the requirement for integration with existing infrastructure.

7 Conclusion

The distributed robotics framework is based on industrial-strength distributed real-time middleware to achieve real-time performance. Furthermore, it provides additional software components and functionality to simplify the robotic application developments. The framework meets all functional requirements. Performance evaluation will be reported in future.

References

[Becker and Pereira, 2002] L. B. Becker and C .E Pereira. SIMOO-RT - An Object-oriented Framework for the Development of Real-time Industrial Automation Systems. *IEEE Trans. on Rob. and Auto*, 18(4), Aug 2002.

[DotGNU,] DotGNU. GNU. http://www.dotgnu.org/.

[Dozio and Mantegazza, 2003] L. Dozio and P Mantegazza. Real-time distributed control systems using rtai. In *Proceedings of the 6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 2003.

[Fayad *et al.*, 1999] M. E. Fayad, D. C. Schmidt, and R. E. Johnson. *Building Application Frameworks*. Wiley, 1999.

[Gamma *et al.*, 1994] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison Wesley, 1994.

[GNOME,] GNOME. Bonobo Document Model. http://developer.gnome.org/arch/gnome/componentmodel/bonobo.html.

[Gomaa, 2000] H Gomaa. *Designing Concurrent, Distributed and Real-time Applications with UML*. Object Technology Series. Addison Wesley, 2000.

[Heimfarth *et al.*, 2003] T. Heimfarth, M. Götz, F. J. Rammig, and F. R. Wagner. Rtc: A real-time communication middleware on top of rtai-linux. In *Proceedings of the 6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 2003.

[Jensen, 2000] E. D. Jensen. Distributed real-time specification. http://www.jcp.org/en/jsr/detail?id=50, Aug 2000.

[Löwy, 2003] J. Löwy. *Programming .Net Components: Design and Build Maintainable Systems Using Component-Oriented Programming*. O'Reilly & Associates, Inc., CA, April 2003.

[Montemerlo *et al.*, 2002] Michael Montemerlo, Nicholas Roy, and Sebastian Thrun. CARMEN. Carnegie Mellon Robot Navigation Toolkit. http://www-2.cs.cmu.edu/~carmen/, 2002.

[Mozilla, 2003] Mozilla. XPCOM - Cross Platform Component Object Model. http://www.mozilla.org/projects/xpcom/, 2003.

[Nesnas *et al.*, 2003] A. Nesnas, I.A.and Wright, M. Bajracharya, R. Simmons, and T. Estlin. CLARAty and Challenges of Developing Interoperable Robotic Software. In *Proc. IEEE International Conference on Intelligent Robots and Systems (IROS)*, Nevada, Oct 2003.

[OMG, 2002] OMG. *Real-Time CORBA Specification*. Object Management Group, Inc., 1.1 edition, August 2002.

[OpenOffice.org,] OpenOffice.org. UNO - the **U**niversal **N**etwork **O**bjects Model. http://udk.openoffice.org.

[OROCOS,] OROCOS. Open Robot Control Software Open Robot Control Services. http://www.orocos.org.

[Schmidt and Vinoski, 2001] D.C. Schmidt and S. Vinoski. Object Interconnections: Real-time CORBA, Part 1: Overview and Motivation. *C/C++ Users Journal*, December 2001.

[Schmidt and Vinoski, 2002a] D.C. Schmidt and S. Vinoski. Object Interconnections: Real-time CORBA, Part 2: Applications and Priorities. *C/C++ Users Journal*, Jan 2002.

[Schmidt and Vinoski, 2002b] D.C. Schmidt and S. Vinoski. Object Interconnections: Real-time CORBA, Part 3: Thread Pools and Synchronizers. *C/C++ Users Journal*, March 2002.

[Schmidt and Vinoski, 2002c] D.C. Schmidt and S. Vinoski. Object Interconnections: Real-time CORBA, Part 4: Protocol Selection and Explicit Binding. *C/C++ Users Journal*, May 2002.

[Schönig and Geschwinde, 2003] H. Schönig and E. Geschwinde. *Mono kick start*. SAMS, Sept 2003.

[Seely, 2002] S. Seely. *SOAP: Cross Platform Web Service Development using XML*. Prentice-Hall PTR, Prentice-Hall, Inc., Upper Saddle River, NJ 07458, 2002.

[Simmons and Apfelbaum, 2004] R. Simmons and D. Apfelbaum. TDL: Task Decription Language. http://www-2.cs.cmu.edu/~tdl/, 2004.

[Szyperski *et al.*, 2002] C. Szyperski, D. Gruntz, and S. Murer. *Component Software*. Addison Wesley, 2nd edition, November 2002.

[Utz *et al.*, 2002] H. Utz, S. Sablatnög, S. Enderle, and G. Kraetzschmar. Miro - middleware for mobile robot applications. *IEEE Transactions on Robotics and Automation*, 18(4), August 2002.

[Vaughan *et al.*, 2003] R. T. Vaughan, B. P. Gerkey, and A. Howard. On device abstractions for portable, reusable robot code. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems*, October 2003.

[W3C, 2003] W3C. Simple Object Access Protocol (SOAP) 1.2 specifications. http://www.w3.org/TR/soap/, June 2003.

[Woo *et al.*, 2003] Evan Woo, Bruce A. MacDonald, and Félix Trépanier. Distributed mobile robot application infrastructure. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 1475–80, Las Vegas, October 2003.

[Woo, 2002] E. Woo. A distributed application infrastructure for mobile robots. Master's thesis, The University of Auckland, November 2002.