Axon: A Middleware for Robotics

by

Michael Morckos

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2012

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The area of multi-robot systems and frameworks has become, in recent years, a hot research area in the field of robotics. This is attributed to the great advances made in robotic hardware, software, and the diversity of robotic systems. The need to integrate different heterogeneous robotic components and systems has led to the birth of robotic middleware. A robotic middleware is an intricate piece of software that masks the heterogeneity of underlying components and provides high-level interfaces that enable developers to make efficient use of the components. A large number of robotic middleware programs exist today. Each one comes with its own design methodologies and complexities. Up to this moment, however, there exists no unified standard for robotic middleware. Moreover, many of the middleware in use today deal with low-level and hardware aspects. This adds unnecessary complexity in research involving robotic behavior, inter-robot collaboration, and other high-level experiments which do not require prior knowledge of low-level details. In addition, the notion of structured lightweight data transfer between robots is not emphasized in existing work. This dissertation tackles the robotic middleware problem from a different perspective. The aim of this work is to develop a robust middleware that is able to handle multiple robots and clients within a laboratory environment. In the proposed middleware, a high-level representation of robots in an environment is introduced. Also, this work introduces the notion of structured and efficient data exchange as an important issue in robotic middleware research. The middleware has been designed and developed using rigorous methodologies and leading edge technologies. Moreover, the middleware's ability to integrate different types of robots in a seamless manner, as well as its ability to accommodate multiple robots and clients, has been tested and evaluated.

Acknowledgements

Foremost, I would like to express my deepest gratitude to my advisor Dr. Fakhreddine Karray, for his continuous support and supervision of my MASc. studies and research. His patience, motivation, immense knowledge and professionalism were the driving force of my research and writing.

I would also like to acknowledge Jamil Abou-Saleh, Sepideh Seifzadeh, and Pouria Fewzee for their assistance and help in the CPAMI laboratory at the University of Waterloo.

I would also like to give special thanks to my thesis readers, Dr. William D. Bishop, and Dr. Andrew Morton for taking the time to review and assess my research work, as well as for their useful and pertinent feedback.

Dedication

This thesis is dedicated to my family. My father *Yousif Morckos* and my mother *Viola Yousif*, my brothers *Mark* and *Matthew*. I owe everything to their immense love, sacrifice, endless support, and encouragement.

Thank you all.

Table of Contents

List of Figures

List of Tables

List of Acronyms and Abbreviations

ACTS Advanced Color Tracking System

API Application Programming Interface

ARCOS Advanced Robotics Control Operating System

ARIA Advanced Robotics Interface for Applications

ARNL Autonomous Robotic Navigation and Localization

CORBA Common Object Request Broker Architecture

COTS Commercially available Off-The-Shelf

CPAMI Centre for Pattern Analysis and Machine Intelligence

DOF Degree of Freedom

EI Entity Interfacer

EM Entity Manager

FSM Finite-State Machine

GNU GPL GNU General Public License

GPB Google Protocol Buffers

GUI Graphical User Interface

HRI Human-Robot Interaction

HTTP Hypertext Transfer Protocol

ICE Internet Communications Engine

IDE Integrated Development Environment

IDL Interface Definition Language

IEEE Institute of Electrical and Electronics Engineers

IP Internet Protocol

LAN Local Area Network

MCL Monte-Carlo Localization

OMG Object Management Group

OSI Open Systems Interconnection

POSIX Portable Operating System Interface

PTZ Pan Tilt Zoom

QoS Quality of Service

RPC Remote Procedure Call

RT Real-Time

RTOS Real-Time Operating System

RTT Round Trip Time

SDK Software Development Kit

SLICE Specification Language for the Internet Communications Engine

SMTP Simple Mail Transfer Protocol

SOAP Simple Object Access Protocol

SSL Secure Sockets Layer

TCP Transmission Control Protocol

UML Unified Modeling Language

URL Uniform Resource Locator

USB Universal Serial Bus

UUID Universally Unique IDentifier

WLAN Wireless Local Area Network

XML Extensible Markup Language

Chapter 1

Introduction

Robotics became a major research area in the 1980's and has been experiencing a rapid growth ever since. Advances in the hardware, sensors, actuators and wireless technologies have opened the door to countless unprecedented opportunities in robotics and applications. A growing number of universities house an advanced robotics lab in at least one department. Robotics is a highly interdisciplinary field, using applications of mechanical, electronics, communication and software engineering.

A typical modern robot consists of a distributed system of hardware components such as sensors and actuators. These components are controlled by sophisticated software ranging from low-level controllers and device drivers to high-level software libraries implemented in high-level programming languages. There is great diversity in robotic software and hardware components. This is attributed to the growing number of robot vendors. However, the robot industry and corresponding markets are far from mature. To date, there are no global standards for robotics. Most of the vendor-provided robotic hardware and software are self-contained and proprietary with minimal or zero support for interoperability. Most robotic systems are incompatible with other robotic systems.

The evolution of middleware has greatly simplified the task of building distributed and loosely coupled applications. According to Bakken *et al.* [19], "middleware is a class of software technologies designed to help manage the complexity and heterogeneity inherent in distributed systems. It is defined as a layer of software above the operating system but below the application program that provides a common programming abstraction across a distributed system.". In robotics, the purpose of middleware is to bring several heterogeneous components into a single cohesive system. The past few years have witnessed the birth of several novel robotic middleware that can accommodate a variety of hardware

components and robots.

However, the distribution requirements extend beyond a single robot. The area of multi-robot control and collaborative task achievement among different robots has long been a hot research topic. Major advances in computer networking and the birth of the Internet have greatly facilitated efficient and reliable communication among separate entities. Thus, there is an ever growing need for generic robotic frameworks and middleware. Many modern robotic middleware programs not only deal with the heterogeneity of components onboard a single robot, but also deal with network heterogeneity by providing a generic network interface to enable multiple robots to interact in a seamless manner. Effectively, robotic middleware provides a framework to integrate several robots into an environment where they can be controlled by a human operator to carry out some tasks and/or collaborate together to achieve a major goal through data and information exchange.

1.1 Motivation

Numerous high-profile robotic middleware programs are currently in use. Robotic middleware programs have varying architectures but typically have the same set of functionalities such as providing hardware and software abstraction layers, resource management, etc. The provided low-level and hardware interfaces provide tight control. While this is desirable in many applications, it can sometimes add unnecessary complexities. As mentioned earlier, robotics is an interdisciplinary field; researchers working with robotics come from different engineering backgrounds and have different needs. For instance, a researcher working on high-level applications such as localization or collaborative robotic behavior may treat the robot as a single entity that provides inputs for his/her application and can be acted upon in return. Such researchers are usually uninterested in lower-level details such as sonar ranging acquisition rate, or low level motor control signals [41].

Thus, the work presented in this thesis is motivated by the existence of the following issues:

- The lack of a generic middleware standard that can accommodate a wide range of heterogeneous robots.
- The (sometimes) unnecessary complexities in many existing middleware programs.
- While existing middleware programs can accommodate many robotic components, they cannot fully replace vendor-provided tools and libraries. For a robot, it is

undeniable that, in most cases, the vendor-provided tools are the best and most efficient way to make use of all of the robot's capabilities.

- In existing middleware programs, and especially those that support networked robots, the exchange of data and information between multiple robots is implicit and not explicitly discussed. The notion of efficient structured information and data exchange between robots is not heavily emphasized in existing work.

1.2 Objectives and Contributions

The work presented in this thesis aims at tackling the problem of robotic middleware from a novel perspective. The main points are as follows:

- Representing a robot as a high-level single entity in the environment rather than fine-grained modeling.
- Providing a system/framework for rapid prototyping of robotic applications and algorithms without worrying about low-level hardware details.
- Providing greater transparency for developers. Namely, network and communications transparency.
- Enabling developers to make full use of vendor-provided libraries and tools, while at the same time allowing for seamless integration with other systems.
- Introducing the notion of structured lightweight data and information exchange between multiple robots in an environment.

1.3 Organization of the Thesis

The remainder of the thesis is organized as follows: Chapter 2 provides a detailed overview of recent advances in the areas of robotic middleware and multi-robot frameworks, along with the latest state-of-the-art work presented in the literature. Moreover, this chapter covers the concepts and challenges of robotic middleware as well as emphasizes some of the related middleware programs, which will be essential for clear presentation of subsequent chapters. Chapter 3 describes, in detail, the architecture of the proposed middleware. This

includes the purpose of the work, design decisions, software designs and methodologies as well as the different components of the system and how they work together. Chapter 4 describes the robotic entities and the experimental setup used to test and evaluate the middleware in a real environment. Chapter 5 discusses the evaluation metrics used to exhaustively evaluate the performance and scalability of the middleware. Chapter 6 concludes the thesis by summarizing the contributions of the presented work and discusses ideas for future research work.

Chapter 2

Background and Literature Review

In this chapter, a detailed discussion of robotic middleware and frameworks is given, including design goals, typical functionalities, architectures, and challenges. Moreover, the current state-of-the-art advances in multi-robot control frameworks and middleware-based architectures are covered. These combined will help explaining how the proposed middleware emphasizes certain elements in multi-robot environments and how it tackles the problem from a different perspective.

2.1 Robotic Middleware

Robots come in different types and configurations. A mobile robot is an automatic machine that is equipped with wheels for cruising. A robotic arm is a sophisticated set of joints and actuators with different degrees of freedom mimicking the human arm and used to manipulate objects. Another example is humanoid robots, whose physical structure resembles that of the human body. A modern robot is usually equipped with a sophisticated array of heterogeneous hardware components. Examples of these are sensors such as infrared sensors and range finding devices such as laser and sonar, cameras, and vision systems, to name a few. These devices are usually controlled by low-level microcontrollers and device drivers. A large number of modern robots can be controlled using computers and come with vendor-provided software components which control the low-level hardware components. Thus, a modern robot can be modeled as a highly distributed system of hardware and software modules. At the robot level, different hardware components can be provided by different vendors. The same applies to software components which can also come in different programming languages. Moreover, different vendors use different communication

protocols between components. While modularity is highly favored in the engineering of sophisticated systems, it is a major issue in robotics integration and configuration. At the robot level, integrating different heterogeneous components in a manner that fully utilizes the robot functionalities is a challenging task. An even more challenging task is integrating different robots into a single cohesive multi-robot framework. The unique nature of robotics comes with its own challenges. Some are listed here:

- Application development for robotics is a difficult task since it sometimes requires knowledge from different engineering fields.
- The heterogeneity of robotic hardware and software components, as well as their communication protocols constitute a major obstacle in multi-robot integration. This results in a steep learning curve and longer development times.
- The lack of a unified standard for robotic communication, or integration means that most multi-robot frameworks are developed in an ad-hoc manner, serving only a specific purpose, and can only be used in a handful of environments.
- Different robots have varying physical properties and offer different services. Making use of all the different services in a meaningful and efficient manner is a challenging task. Moreover, developing "run anywhere" applications that could be deployed on different robots is a very challenging task [41].
- Robots have different and varying operational requirements compared to conventional machines. Some critical robotic systems must operate in a real-time fashion, others may have more relaxed constraints and operate in a "best effort" fashion [41].

There is a growing need for a middle layer, or a *middleware*, for robotics. A middleware, in a distributed system context, is a sophisticated piece of software that enables different distributed components to communicate and manage data in a meaningful manner [8]. The main function of any middleware is to mask all the heterogeneity of components in a distributed system and provide the user with a structured set of high-level services and functionalities that leverage the use of the underlying components. Designing a middleware that can accommodate multiple hardware and software components is one of the most challenging research problems in the field of robotics. A perfect robotic middleware should accomplish the following:

- Reduce robotic software applications development times by providing a high-level abstraction of the underlying components [37].

- Allow for the integration of different types of components in a seamless manner, regardless of their types and configurations, by hiding the heterogeneity of the underlying components.
- Promote reusability and shorter development times by providing commonly used robotic services such as range readings, color tracking, image processing, navigation, localization, and so on, as pluggable components in the system. These services should be independent of any hardware or operating system.
- Provide real-time capabilities for time-critical applications.
- Fully utilize all the underlying functionalities of a robot and ensure minimal overhead and latency.

2.2 Middleware for Networked Robots

Middleware for networked robots, or multi-robot middleware, is a subclass of robotic middleware programs. This type of middleware aims at integrating different entities such as robots, external systems, and sensor networks, into a single framework where entities can use services provided by each other, and collaborate to achieve common goals. Designing middleware programs for networked entities is considered to be a major research problem in robotics. In addition to the inherent impediments encountered in designing conventional robotic middleware, new issues and challenges arise when designing multi-robot middleware programs. A perfect multi-robot middleware should provide the following:

- Collaboration is one of the most common requirements in a multi-robot system. It is important that the middleware provides high-level collaboration frameworks and mechanisms to facilitate development of collaborative operations among multi-robots.
- Be able to integrate different types of systems and components that can be used by robots, such as sensor networks and high performance machines running complex algorithms.
- Since multi-robot environments are highly dynamic and uncertain, the middleware should provide facilities for dynamic and efficient resource discovery. This greatly enhances the robot's ability to be self-adapting and self-optimizing.
- The middleware should provide an efficient structured information and data carrier mechanism across different integrated robots and components.

In multi-robot middleware, the communication model defines the standard and technologies used to implement the communication infrastructure of the middleware, through which different robots and components communicate. The communication model is an indispensable part of any middleware. Different standard and non-standard models are employed by existing middleware programs.

Standard network middleware is a major building component in many existing multi-robot middleware programs. A well known example is the Common Object Request Broker Architecture (CORBA) standard, defined by the Object Management Group (OMG) [7]. CORBA is by far the most used network middleware in robotic middleware. It defines standards for inter-process and inter-platform communications. Moreover, it defines a set of utilities that are inherently important in distributed systems, such as event distribution, distributed notifications, and service discovery. A large number of implementations exist for CORBA [2]. Another relatively new network middleware is the Internet Communications Engine (ICE), developed by ZeroC [17]. ICE is an object-oriented middleware that provides platform and programming language independence, as well as RPC and publish-subscribe functionalities. Some of the interesting features of ICE is that it is object-oriented, built using component-based design methodologies, and supports grid computing.

Besides network middleware technologies, some robotic middleware rely on web technologies such as the Simple Object Access Protocol (SOAP). SOAP is a W3C protocol specification for the exchange of structured information. SOAP uses XML-based messages and relies on application layer protocols of the OSI model, such as HTTP and SMTP, for message transport.

Lastly, some middleware programs rely on non-standard communication models such as custom-implemented shared memory. The rationale behind this is to provide specific low-level functions that cannot be achieved using general purpose technologies.

2.3 Literature Review

The past fifteen years have witnessed numerous research contributions in the area of robotic middlware. Although existing work has focused on different purposes and capabilities, it can generally be split into two major categories. On one hand, the first category tackles the issue of hardware and software heterogeneity from a low-level perspective, focusing on abstracting different types of sensors, actuators and other types of hardware. On the other hand, the second category addresses the problem from a higher level perspective by focusing on the higher level robotic functionalities and human operator experience.

Utz *et al.* [46] developed Miro, a mobile robot middleware that tackled the problem of heterogeneity in robotics hardware and software components. The authors of Miro adopted an object-oriented design methodology. They relied on the CORBA standard to facilitate the software development process and ease the integration of different robotic entities, as well providing hardware and operating system independence. Miro is composed of three layers: the device layer, the service layer, and the class framework layer. The device layer is the platform-dependent component of the middleware. It abstracts all the robotic hardware components such as the sensors and actuators. The service layer provides an abstracted interface to hardware components through the use of the Interface Definition Language (IDL) of CORBA. This essentially models sensors and actuators as objects with callable programmatic routines. The use of CORBA IDL eliminates any programming language and location dependencies. The class framework layer lies atop the previous two layers. It provides a collection of common services executed on mobile robots such as localization, path planning, navigation, and logging. The use of cross-platform libraries such as the CORBA-based adaptive communication environment (ACE) [40] and CORBA notification service [27] greatly promotes portability.

In addition, Jung *et al.* [30] proposed a component-based framework for seamless integration of robotic systems. This work is based on the *cisst* package which is a collection of `C++`-based libraries implementing lock-free data structures and providing efficient data exchange mechanisms for communicating threads within a running process. A major part of the *cisst* package is the component framework that models each running thread within a process as a separate component with typical provided and required interfaces. These interfaces enable the components to behave in a client-server manner. Jung *et al.* augmented the data exchange mechanism by using the proxy design pattern and network middleware to enable inter-process communication across networks while preserving the original data exchange mechanism and location transparency. ICE was chosen as the network middleware due to its high performance and its use of the proxy design pattern, which matches the adopted approach in this work.

Lee *et al.* [32] viewed the robotic middleware problem from a more general perspective by proposing a middleware architecture and a software component model for building intelligent service-oriented robotic systems. Lee *et al.* argue that the existing middleware programs are too complex and can only be used in a handful of environments. Lee *et al.* proposed a four-layer middleware architecture. The first and bottom layer is called the operating environment layer. It is responsible for providing real-time capabilities and operating system abstraction. The second layer is the communication layer which houses network middleware programs such as CORBA and ICE. This layer is responsible for providing programming language independence and location transparency. The third layer is

the intelligent service layer. It makes use of component-based software models and design patterns to create a repository of reusable intelligent services. The fourth and final layer is the application layer which provides application developers with a set of powerful tools and APIs for application and user interface development. The second contribution is a simplified and robust component software model. Lee *et al.* argue that component models of other work are either incomplete or too convoluted. The proposed model provides a template for component-based design. In this model, each component has three interfaces: message interface, event interface and data interface. The message interface provides a message interaction interface to other components. The data interface is used for data exchange. The event interface is used for receiving and sending events. Events have the highest priority since they control the flow of execution within the component. There are three types of components: device components, service components, and basic components. Device components can be used to control separate hardware parts on the robot. Commonly used robotic services can be implemented as service components. Basic components are used to implement communication infrastructure and housekeeping facilities for other components.

Ando *et al.* [18] introduced RT-middleware (Robotics Technology Middleware). RT-middleware is a common platform standard for robotic systems based on distributed objects. The standard aims at achieving two major goals. The first is to broaden the robot market through the creation of a standardized and generic architecture that emphasizes modularization of different robotic components. This, according to Ando *et al.*, will remove the barrier between robotic software and hardware manufacturers. The second goal is to increase research efficiency by having researchers focus on implementing their experiments/robotic applications instead of worrying about building robotic systems from scratch. To achieve these two goals, RT-middleware aims at realizing certain capabilities such as real-time execution, synchronization, platform independence and network independence. The building block of RT-middleware is the RT-component. It is a specification standard defined by the Object Management Group (OMG) [7] and modeled on CORBA. An integrated robotic system can be created by combining various network-enabled RT-components. An RT-component is composed of one or more component objects. A component object is a state machine-based processing unit that has input and output ports, as well as command interfaces. A wide scale of robotic modules with different granularity can be implemented using RT-components. An implementation of the RT-middleware standard called the OpenRTM-aist [9] was developed by the National Institute of Advanced Industrial Science and Technology (NIST).

Santos *et al.* [39] proposed an adaptive and real-time producer-consumer-based middleware for information exchange among autonomous robots over a wireless network, and

running on the Linux operating system. The middleware is composed of three main components: a real-time database (RTDB), an adaptive wireless communication protocol, and a task manager called Pman. The RTDB is a replicated blackboard containing a repository of records with their associated data blocks. Each robot in the environment has its own RTDB that stores the robot's state data plus the local state data of every other robot. Each robot periodically broadcasts a subset of the state data stored in its RTDB to every other robot. The blackboard design allows the RTDB to act as a dual-port memory where the application and the communication channel act as the producer and the consumer respectively, or vice-versa. All stored data are timestamped to allow for data age estimation and validation, two things that are crucial in real-time data exchange. The second component of the middleware is the wireless communication protocol. It uses an adaptive time-Division Multiple-Access (TDMA) mechanism that accounts for load variation in the network. The adaptivity ensures minimal access collision between communicating robots. The third part of the middleware is the Pman which compensates for the limited support for time management of processes on Linux by providing real-time services to running applications.

The work done by Song *et al.* [44] is yet another attempt to tackle the problem of robotic middleware in environments with real-time and quality of service (QoS) constraints. Song *et al.* implemented a CORBA-based framework to integrate different robotic and automatic control systems. Since CORBA neither has inherent real-time support nor QoS support, the real-time CORBA (RT-CORBA) specification was used. RT-CORBA provides a set of standard services needed in real-time middleware. An open source implementation of the RT-CORBA called ACE ORB (TAO) was used in an integrated framework consisting of heterogeneous robotic and control entities such as mobile robots, robotic manipulators, and vision systems.

Makarenko *et al.* [35] discussed an upgraded version of Orca, an open-source component-based middleware for a wide range of robots and distributed sensor networks. Orca emphasizes promoting robotic software reuse. According to Makarenko *et al.*, lack of software reuse is a major limiting factor in robotics. Orca provides an infrastructure that supports development of robotic software components and provides developers with the flexibility of specifying component interfaces while not imposing a specific design pattern or architecture. Moreover, Orca provides a collection of APIs enclosed in the `libOrcaIce` library. This library houses a large collection of facilities and patterns commonly used in robotic applications development. It should be noted that the developer has the freedom to choose whether to use the library or not. Orca relies on ICE as its underlying network middleware. All developed components make direct use of ICE.

Bruyninckx *et al.* [20] introduced the Open Robot Control Software (OROCOS), an am-

bitious project that aims at becoming a de facto open standard in the robotics community. The motivation behind OROCOS stemmed from the lack of openness and standardization in the robotics industry where many major companies are producing highly specific robotic equipment using their own proprietary technologies and protocols. Bruyninckx *et al.* intend to follow a highly organized and modular approach in designing and developing OROCOS. OROCOS is split into three types of modules. The first type is the supporting module that contains common robotic services and functionalities, as well various support and utility modules such as real-time support, logging, and so on. The second type is the robotic module. It is a repository of novel robotic algorithms and hardware control software. The third type is any user-developed component. These components can be combined together to form complex robotic platforms. CORBA is chosen as the underlying network middleware and the building block for all user-developed components. Moreover, the development of OROCOS is to be based on a novel incremental process involving detailed discussions among developers, use of open standards such as CORBA IDL and XML, use of novel and original algorithms and techniques from a wide range of domains, as well as applications of software design patterns.

Quigley *et al.* [38] developed ROS, an open-source Robot Operating System that aims at tackling the problem of designing and developing large-scale distributed robotic applications. The design of ROS is based on the microkernel design, but not in the conventional operating system sense. Instead, ROS provides a structured communication layer and application development facilities that are realized by multiple fine-grained interconnecting tools. ROS greatly promotes modularity as one of its cornerstone philosophies. ROS is based on four fundamental concepts: nodes, messages, topics, and services. A node is the elementary building block of a ROS-based application. A node is analogous to a software module. A single application can consist of multiple nodes. Nodes communicate with each other through message passing. Messages range from primitive data types to compound messages. Moreover, nodes can communicate in a publish-subscribe manner by publishing and/or subscribing to different topics. In addition, synchronous communications is possible using services. A service can be defined by a name as well as request and reply messages. ROS-based modules residing on different robots and machines communicate in a peer-to-peer fashion. The rationale behind this is that a centralized communication can become a hinder when different hosts reside in heterogeneous network. Peer-to-peer communication occurs through XML-RPC. Regarding programming language support, ROS supports `C++`, `Python`, `LISP`, and `Octave`. Moreover, it is natively implemented in some of these languages. In addition, ROS provides a programming-language-independent IDL for specifying communication interfaces between different modules. The modularity of ROS led to the incorporation of several external open-source tools such as robotic algorithms,

simulators, and so on. Moreover, various tools were developed to work with ROS. These tools offer multiple services such as debugging, collaborative development, monitoring and visualization, to name a few. ROS is an open-source project that is distributed under a flexible license allowing for the development of both commercial and non-commercial applications.

Côté *et al.* [21] introduced MARIE (Mobile and Autonomous Robotics Integration Environment). It is a distributed component-based middleware that aims to shorten the software development process for robotics by promoting reusability of robotic software components, as well as integration of existing and new robotic environments and modules. MARIE allows for the integration of inherently incompatible robotic frameworks and components. This gives researchers access to a wide range of services and functionalities, while at the same time reducing development time. Côté *et al.* adopted a layered architecture in the design of MARIE to provide multiple abstraction layers, as well as more control on the development details. As such, MARIE is composed of three layers: the core layer, the components layer and the applications layer. The core is the lowest layer. It houses various infrastructure tools such as communication mechanisms, data handling, as well as distributed and low-level operating systems services. The component layer provides facilities for adding reusable components. The application layer is the upper layer. It provides the main framework for developing and integrating robotic applications. The core of any MARIE-based application is the Mediator Interoperability Layer (MIL), which is based on the mediator software design pattern. MIL enables building of distributed robotic systems using heterogeneous components by acting as a central unit that handles communication among the loosely coupled components. MIL houses four main components: the Application Adapter (AA), the Communication Adapter (CA), the Application Manager (AM), and Communication Manager (CM). The AA acts as a unified interface between the MIL and an application, as well as other applications. The CA handles data and information exchange between AAs of different applications. It resolves any protocol incompatibilities. Finally, the AM is responsible for creating and managing different components in the system. Lastly, the CM is responsible for managing communications between components. MARIE uses ACE ORB as its communication infrastructure.

Gerkey *et al.* [24] introduced the Player/Stage, an ambitious multi-robot framework that grew to become a de facto standard in robotic frameworks and middleware. It is argued that the existence of various types of robots, and the mandatory knowledge of network programming are major hinders in robotics research. Player/Stage is a client-server based framework that models a robot as a collection of hardware devices such as sensors and actuators. The framework enables TCP-enabled client programs to control these devices over the network. There are two main parts in Player/Stage: the Player robot

device server and the Stage multi-robot simulator. The Player server implements device drivers for various robotic hardware devices, and provides TCP socket-based interfaces to these devices. Player has multi-client support, and is usually deployed on the robot's onboard computer or any machine that is physically connected to the hardware devices. Player borrows the UNIX model of representing various devices as files. A client can access various devices using the `read` and `write` UNIX system calls. Moreover, Player provides an extensible and flexible device model that allows for implementing sophisticated algorithms in drivers, as well as the exchange of data and information with other drivers. While the Player server is `C/C++`-based and runs on UNIX-like machines, the client programs can be implemented in different programming languages and run on different platforms. This is attributed to the network-based communication. The second part of Player/Stage is the Stage simulator. Stage is a sophisticated multi-robot simulator that enables fast creation of multi-robot simulations that could be later realized in real environments. Moreover, it enables simulation of devices that have no counterparts in the real world. This assists in research and development of new devices. It should be noted that Stage provides simulation models for numerous devices with reasonable accuracy rather than modeling fewer devices with higher accuracy.

Yoo *et al.* [48] presented the Robot Software Communications Architecture (RSCA), a Real-Time (RT) Quality of Service (QoS) middleware specification for networked robots based on the Software Communications Architecture (SCA), a Software-defined radio specification. Similar to works presented earlier, the work presented here tries to tackle the issue of heterogeneity of robotic components. RSCA introduced a specification for standard common interfaces similar to SCA, albeit for robotic applications. These interfaces are split into two groups. The first group is the operating environment interfaces that are used to deploy applications. The second group is the application component interfaces. These interfaces have to be implemented by the robotic application to leverage the functionalities a robot has to offer. The core of RCSA is composed of three main parts: a Real-Time Operating System (RTOS), communication middleware and a core middleware. The RTOS is a POSIX-based operating system that is compliant with an IEEE standard. The communication middleware is based on RT-CORBA. These two provide the essential RT capabilities, as well as various services such as logging and event services. The most important part is the core middleware, also called the core framework. This framework is composed of a set of interfaces and XML-based domain profiles. These interfaces provide a comprehensive set of APIs that are used by robotic applications and/or the other parts of the middleware. The "service" interface is responsible for providing QoS, as well as resource management through devices interfaces. The domain profiles are used to describe various hardware and software configurations. RCSA provides an abstraction layer and

transparency among different robotic applications.

Hernández-Sosa *et al.* [28] proposed CoolBOT, a component-oriented robotic framework implementing mechanisms for dynamic resource management. CoolBOT addresses the issue of software reusability in robotic applications development, as well as adaptive management of low-level resources on a robot, such as processor overhead. A component in CoolBot is represented abstractly as a port automata. This representation breaks a component into three parts: the internal functionality, the external interface, and an automaton modeling the different states of the component. The external interface provides a set of read/write ports. The automaton is composed of external and internal states where the external states can be controlled by the user of the component, and the internal states are responsible for performing the core functionalities of the component. CoolBOT supports two types of components: atomic components and compound components. An atomic component is used for low-level control of hardware devices or for implementing general purpose algorithms. A compound component is composed of other components in a hierarchical fashion. Each compound component is managed by a supervisor component. In execution, components are mapped to threads specific to the host operating system. Different components can communicate through their input ports using the developed Inter Component Communications (ICC), a similar mechanism to Inter Process Communication (IPC) found in operating systems. A component can be either adaptive or non-adaptive. In CoolBOT, adaptive management of resources is defined through two controllable variables: the frequency of operation and the quality level. Moreover, each adaptive component must operate at certain performance levels, which can be adjusted. The supervisor component manipulates the two variables to achieve the desired performance level at runtime. Each component has a set of "adaptive observables" which are monitored externally by the framework. Performance level is continuously adjusted based on these adaptive observables.

Magnenat *et al.* [33] introduced ASEBA, a modular low-level event-based middleware that aims at providing distributed control and efficient resource utilization on multiprocessor-powered robots. The middleware targets robots that have peripheral microcontrollers controlling sensors and actuators. It introduces a new architecture where microcontrollers can communicate through asynchronous messages called events. This design replaces the conventional periodic polling for sensor data and sending actuator commands. Moreover, ASEBA adds to the functionalities performed by microcontrollers. It improves modularity and efficiency by distributing several processing tasks among microcontrollers and have them communicate with the main processor only when needed. This greatly reduces the workload of the main processor and reduces bus latency. To achieve this, ASEBA comes with the AESL scripting language and an Integrated Development Environment (IDE).

AESL is a simple scripting language that is used to implement event firing and reception behaviors. The developed script is then compiled into bytecode using the IDE and runs in a lightweight virtual machine on the microcontroller. These custom designed virtual machines have light memory and processing footprints that are suited for embedded systems.

Gil *et al.* [25] introduced a data-centric middleware for integrating wireless sensor networks and mobile robots. It is part of the larger European middleware project AWARE [1]. The proposed middleware aims at providing energy efficient mechanisms to enable mobile robots to gather data and information from sensor networks in an environment. This data can be environmental data such as temperature, light and humidity, or localization data that could be used by the robot to locate itself in the environment. The main feature of the middleware is the abstract data-centric representation of environment entities and object. For instance, an object can be a vehicle, a fire, or an animal. Each object has a unique identification. Users can access these information in an abstract way. Moreover, they can provide definitions for certain objects such as dampness for a liquid spill, or high temperature for fire. The middleware components run on both the nodes of the sensor networks and participating mobile robots. The middleware components deployed on sensor nodes are based on the TinyOS.

The OPRoS project, introduced by Song *et al.* [42], is a component-based platform for networked robots that aims at providing a standard model for developing complex robotic applications using COTS components, while promoting reusability and compatibility among heterogeneous robots. The main goal of OPRoS is to support distribution and high-level control through RPC. In OPRoS, a robotic application is composed of one or more loosely coupled software components. OPRoS components come in two types: atomic and composite. On one hand, an atomic OPRoS component is the basic building block of an application. Each atomic component is composed of an execution module based on a FSM and an arbitrary number of ports to communicate with the outside world. There are three different types of ports: method port, data port, and event port. A port can be either a provided (input) or required (output) port. Method ports are used to provide and/or acquire component attributes through client-server method invocation. Data and event ports are used for data and events exchange respectively. These two types of ports use the publish-subscribe pattern for communication. On the other hand, a composite component is composed of one or more atomic components. Sub-components inside a composite component are coupled using a hybrid method that combines both connection-oriented and hierarchical types of composition. Each component on a robot is managed by the component container. The component container is responsible for managing the lifecycle of components and resources.

2.4 Commercial Robotic Systems

There is a number of commercial robotic middleware and frameworks. One of the most famous ones is the Microsoft Robotics Developer Studio (RDS), developed by Microsoft Corporation [5]. It is a Windows-based environment for designing and developing robotic applications. RDS targets academic, commercial, and hobbyist robotic application developers. The main feature of RDS is a programming model and framework that provides facilities for developing asynchronous and state-driven robotic applications. RDS provides a bundle of facilities that provide greater transparency for developers, as well as shorter applications development times. RDS is based on the Concurrency and Coordination Runtime (CCR), a .NET-based concurrency library that masks all complexities of multithreading and synchronization, and provides a clean simple interface to developers. In addition, RDS provides a lightweight state-oriented Decentralized Software Services (DSS) framework that enables applications to interoperate on robots and connected machines. To further speed up development, RDS provides the Visual Programming Language (VPL). VPL is a tool that enables building robotic applications by visually dragging and dropping building blocks. This also promotes reusability by using previously created blocks in multiple applications. For simulation, RDS comes with the Visual Simulation Environment (VSE). VSE is a 3D physics-based simulation tool that enables developer to simulate and test their robotic applications in various settings and environments. RDS supports various types of robots and hardware components. Experienced developers can develop their own RDS-based control modules. The `C#` programming language is the main language used for developing RDS-based applications.

2.5 Chapter Summary

This chapter defined robotic middleware, one of the most challenging research problems in robotics. It discussed the difficulties and challenges associated with designing robotic middleware programs. Moreover, it presented a comprehensive list of features and facilities that should be provided by a perfect robotic middleware.

Some of the state-of-the-art work in the area of robotic middleware and frameworks were covered in this chapter. Moreover, one of the well-known commercial robotic frameworks was covered as well. It can be concluded that a large number of the existing middleware programs deal with low-level and hardware details. As mentioned before, this can add unneeded complexities when carrying out high-level experiments. Moreover, inter-robot data and information exchange is not explicitly emphasized in any of the work. Thus, this

research aims at developing a middleware that provides a high-level modeling of robots, and a structured and efficient inter-robot data exchange carrier.

Chapter 3

Middleware Architecture

This chapter presents the proposed robotic middleware. The work presented in this thesis stems from experiences in working with robotics and fellow researchers and developers. A major issue in a robotics laboratory is how much time it takes to build a demo or an experiment setup involving multi-robots. Setting up a multi-robot experiment can be a tedious and time-consuming process. Aside from implementing the robotic applications, one has to deal with a lot of network programming and concurrency issues to attach different robots into a complete framework. Building a framework can take much longer than implementing the application themselves, which are the main focus of the experiment. In most cases, and due to time constraints, the frameworks developed are ad-hoc with limited or no extensibility and cannot be used in other experiments. The middleware aims at providing a different experience to researchers/developers, as well as ordinary users who have little or no knowledge of robotics.[1] For developers, the middleware aims at minimizing the time spent on setting up experiments/scenarios involving multiple robots. A developer only needs to implement the desired functionalities for a robot in whatever programming language and using whatever APIs are available for that robot. For normal users, the middleware can be used as part of a demonstration framework to enable users who have no prior knowledge of robotics to experiment with different robots and systems. A number of issues that are inherent in robotic middleware were addressed. They are as follows:

[1]The discussion differentiates between two types of people: the researcher/developer and the user. On one hand, the researcher/developer is the person implementing the application and understands how the system works. On the other hand, the user interacts with and controls different robots in the environment through a user interface but does not necessarily know about the system. In a research environment, however, a developer and a user are usually the same person. This notation is used to explicitly differentiate between the different roles of people who make use of the middleware.

- The heterogeneity of robotic software components.
- Rapid and seamless integration of different types of robots.
- Information and data exchange between different entities in a robotic setup.

This work borrows a few ideas from work presented in [35] and [30], such as the use of the Internet Communications Engine (ICE) as the network middleware for the robotic middleware. Unlike many works on robotic middleware, however, it was chosen not to delve into the low-level and hardware details of a robot or model it as a network of components. Instead, the middleware recognizes a robot as a single entity in the environment that possess certain properties and has a number of services to offer to a human operator or other entities. The reasons behind this choice are listed below.

- The main aim of this work is to assist in designing and developing high-level multi-robot applications and experiments where new algorithms and techniques involving robotic behavior, collaboration, reasoning, and so on can be rapidly prototyped and tested. As mentioned before, many of these experiments do not require knowledge of low-level or hardware details.
- Each robot vendor has its own (sometimes proprietary) libraries and protocols for their robots; they are, in most cases, incompatible with libraries provided by other vendors. However, one cannot neglect the fact that vendor-provided frameworks are usually the best in terms of efficiently and easily leveraging all of the capabilities of the underlying robot. The middleware does not aim to replace existing frameworks, but tries instead to introduce a generic-enough protocol that could accommodate different robots with their heterogeneous frameworks. In short, the middleware's main purpose is to enhance the ease of development and reduce the development times for robotic applications.

Regarding the software design and implementation aspects, software design patterns were extensively used in the implementation, such as the state machine and observer patterns [23]. Moreover, a component-based design methodology was followed throughout the design and implementation process. Also, the source code of the middleware was thoroughly documented. These efforts have culminated in an industrial-strength prototype with the following features:

- Modularity: the components of the middleware are independent and loosely coupled. Any component can be modified, moved to a different machine, or even re-implemented in a different programming language without the need to make changes to any components.

- Extensibility: the use of software design patterns, rigorous developing disciplines, and documentation ensures that more functionalities and updates can be easily added in the future. Moreover, the technology used for data and information exchange greatly enhances extensibility.

- Scalability: the middleware is able to accommodate a large number of entities and clients with minimal effect on performance (please see Chapter 5 for more details).

- Performance: while the middleware currently does not include real-time capabilities, it ensures a minimal amount of time for information and data delivery between a client and an entity.

- Safety: robotic environments can be hazardous. Working with mobile-and expensive-equipment always carries the risk of damage and collision. The most basic safety mechanism is to be able to instantly stop or halt a robot if a situation calls for it. The middleware provides a guaranteed safety stop signal that can be acted upon in the robotic application.

In the middleware, an entity can be a robot, sensor network, or any device that can perform/provide services and can be controlled by high-level APIs.[2] Each entity plugged in to the middleware must publish a *profile* detailing the services it has to offer. A typical flow of events in using the middleware can be as follows: When a client program is introduced to the system, the user can browse through a list of all connected entities. The user can then "check out" one or more entities, provided that they are available, and issue various tasks to them. After running some experiments and scenarios, for instance, the user can then release previously "checked-out" entities so that they become available to other users. Figure 3.1 illustrates an abstracted overview of the middleware architecture. The shown components will be discussed in detail.

[2]Only mobile and stationary types of robots were used to test the middleware. Throughout the rest of the thesis, the terms *entity*, *robotic entity*, and *robot* will be used interchangeably.

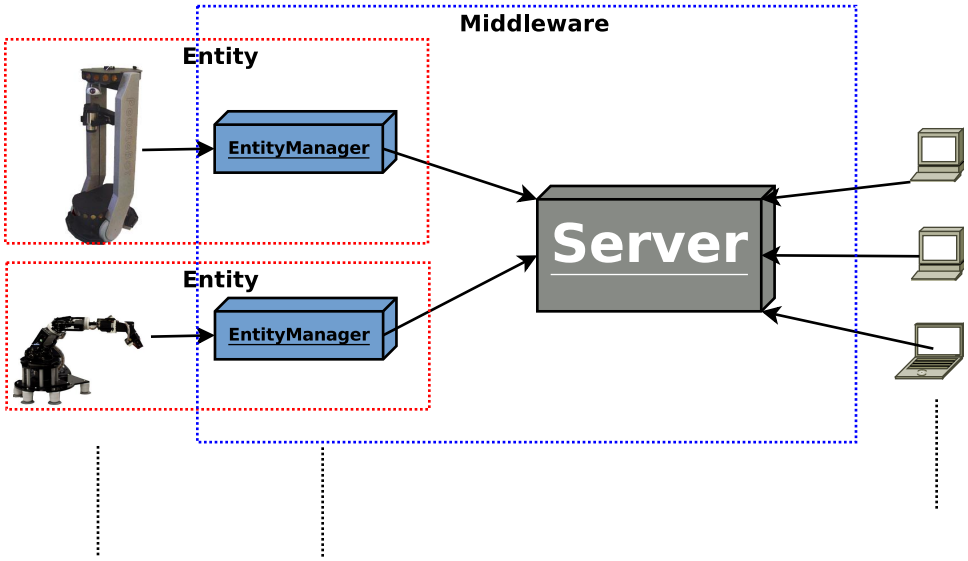


Figure 3.1: Middleware Abstracted Overview.

The discussion starts here by describing what information is needed to model a robot in the environment. This also includes the extensible lightweight data carrier used. Next, the network middleware that was employed as the central hub of the robotic middleware is discussed. Moreover, all the possible data and information that can be exchanged between the different applications in the middleware are discussed. After this follows an expatiation on the middleware architecture and components, and on how they interact and work together.

3.1 Information Base and Data Carrier

This section discusses how each entity is represented in the environment, and what information need to be provided for each entity to make it an active actor in the environment. Moreover, this sections presents the technology that was chosen to have lightweight structured information and data exchange, as well as the rationale behind the choice.

3.1.1 Entity Information

In the middleware, each entity in the environment is recognized by a unique *entity profile*. The entity profile contains a comprehensive set of declarative information, such as the entity's type, location, and services. This information uniquely defines an entity as an active actor in a multi-robot, multi-user environment. The entity profile is composed of two sub-profiles: the *basic profile* and the *services profile*. These XML-based profiles must be defined and created prior to plugging the entity in to the environment. They are discussed in detail in the following subsections.

Basic Profile

The basic profile provides basic information about an entity that can uniquely identify it in the environment. The basic profile for an entity is made available to all connected clients, and it should help a user decide which entity best suits his needs. A typical basic profile should have the following information:

- ***Name***: the name of the entity.
- ***Category***: describes the nature of the entity. In the current version of the middleware, two categories are supported: `ROBOT` and `MACHINE`.
- ***Type***: the type of the entity. Two types are supported: `MOBILE` and `STATIONARY`.
- ***Description***: an optional brief description of the entity.
- ***Xpos***, ***Ypos***, and ***Zpos***: these three parameters define the current position of the entity in an environment.

Figure 3.2 illustrates the basic profile for a mobile robot.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>

<BasicProfile name = "PeopleBot"
            category = "ROBOT"
            type = "MOBILE"
            descr = "Mobile robot with an array of range finding sensors, basic gripper and camera."
            xpos = "1000"
            ypos = "1500"
            zpos = "0" >
</BasicProfile>
```

Figure 3.2: A Sample Basic Profile XML File.

Services Profile

In addition to the basic profile, the developer must also provide the services profile. This profile contains all the details about the services offered by the entity. In the context of the middleware, a service is defined as an executable task that can be performed by an entity upon request from a human operator, or from another entity (such as in collaboration). In addition to its attributes, each service can house two optional sub-components:

- ***Control Parameter***: a control parameter defines how external actors can interact with a service. Control parameters are used to configure a service by providing input and runtime configurations, as well as communicating results and feedback to the requester.
- ***Resource***: a resource represents external data or files that a service might require for execution. Examples of resources are map files, images, or a batch of sensor readings.

For each service, the following attributes should be provided:

- ***Name***: the name of the service.
- ***Global ID***: a global name for the service. This can be used in case the same service is provided by more than one entity, or the service is a sub-service of a composite service. It is the responsibility of the developer to ensure the uniqueness of the global ID.
- ***Type***: the type of the service. There are two supported values: `SMPL` and `CMPLX`. `SMPL` signifies a simple atomic service, while `CMPLX` signifies a composite service that can be composed of other services that are executed in a specific order.
- ***Active***: an optional flag attribute indicating whether the service is enabled or disabled. An enabled service means that it can be requested. A robotic application may enable/disable a service. This can be done, for example, to preserve battery power or resources. However, this choice is left to the developer.
- ***Blocking***: an optional flag attribute indicating whether the service can be executed simultaneously with another service or not. A blocking service could mean that it makes use of resources/components that cannot be shared, or that the simultaneous execution of a second service could interfere with the former.

- ***Pausable***: an optional flag attribute indicating whether the service can be paused in-mid execution or not.
- ***Description***: an optional simple description for the service.
- ***Notes***: optional short notes related to the service.
- ***Service list***: a list of other services. This list should be populated only if the service is `CMPLX`.

As mentioned earlier, each service can have an arbitrary number of control parameters and resources. Each parameter and resource is declared as a separate piece of information having its own attributes and enclosed inside its service. For each parameter the following attributes should be provided:

- ***Name***: the name of the parameter.
- ***Type***: the type of the parameter. A parameter can be an input, output, configuration, or feedback parameter. The allowed values for the type are `INPUT`, `OUTPUT`, `CONFIG`, and `SMPL`.
- ***DataType***: the data type of the parameter. A parameter can be an integer, double, string, or boolean. The allowed values are `INT`, `DBL`, `STR`, and `BOOL`
- ***Units***: an optional attribute signifying the units of the parameter (like millimeters).
- ***Ready***: an optional flag attribute indicating whether the value of the parameter is set or not.
- ***Required***: an optional flag attribute indicating whether the parameter is a mandatory dependency for the owning service or not.
- ***Savable***: an optional flag attribute indicating whether the parameter can be saved or not. If the flag is not set, it means that the parameter can change multiple times, or that it must be set each time its owning service is requested.
- ***Multiple***: this optional attribute is used to indicate that the parameter can store a list of values of the same data type. This can be useful for streaming data or having multiple choices for a single parameter.
- ***Value***: the value of the parameter.

- ***Value list***: list of values of the parameter.

Similar to control parameters, each resource is defined by the following attributes:

- ***Name***: the name of the resource.
- ***URL***: the URL of the resource on the robot's onboard machine.
- ***Ready***: an optional flag attribute indicating whether the resource exists on the robot's machine or not.
- ***Required***: an optional flag attribute indicating whether the resource is required for the owning service to function or not.
- ***Savable***: an optional flag attribute indicating whether the resource is maintained on the robot's machine or not. If the flag is not set it means that that the resource is constantly changing, or that it must be set each time its owning service is requested.

Figure 3.3 illustrates a portion of the services profile for a mobile robot. In the next section the technology that was used as the information and data carrier in the middleware will be discussed, as well as details about how data is stored and transported in an efficient manner.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>

<ServiceProfile count = "7">

<!-- Service #0 -->
<Service parameter_count = "1"
        name = "Simple motion"
        type = "SMPL"
        active = "true"
        blocking = "true"
        pausable = "false"
        global_id = "Walk" >

  <Parameters>
    <Param name = "Distance"
          type = "INPUT"
          data_type = "INT"
          units = "meters"
          value = "1000"
          required = "true"
          set = "true"
          modifiable = "true"
          descr = "Distance to travel" />
  </Parameters>
</Service>

                    <!-- continued -->

</ServiceProfile>
```

Figure 3.3: A Sample Services Profile XML File.

3.1.2 Google Protocol Buffers

The notion of an extensible data and information exchange was not greatly emphasized in robotic middleware. On one hand, some of the works such as those presented in [46], [44], [35], [20], and [43], rely directly on RPC mechanisms provided in the underlying network middleware such as ICE and CORBA to exchange data and information. For instance, sonar readings from a mobile robot can be transmitted as function arguments in an RPC. Since RPC interfaces are specified using Interface Definition Languages (IDL), this approach does not allow for much extensibility in case new types of data and information are frequently included in the system through growing needs or the addition of new robotic entities. In that case, the interfaces have to be frequently modified to support the change. In addition, it is almost impossible to specify all of the needed information and their exchange interfaces at development time. On the other hand, in the ORiN project [36], robotic specifications can be written in XML and exchanged over SOAP. In addition, [22] and [38] use XML-RPC over HTTP for data exchange. While XML is an ideal choice for extensible and structured information exchange, it is also verbose and heavyweight. Moreover, the use of HTTP certainly adds extra overhead. It is believed that information exchange in a robotic environment is a critical issue even without real-time constraints, since in various cases it is essential that data reach a robot within a reasonable amount of time to ensure that the desired effect takes place or to avoid any accidents.

To have a structured lightweight information exchange, the Google Protocol Buffers (GPB) were used [12]. GPB are a collection of platform-and language-independent mechanisms for serializing structured data. GPB are Google's *lingua franca* for the fast inter-machine exchange of structured information. According to [12], "GPB are 20 to 100 times faster than XML and 3 to 10 times smaller." GPB are independent of the transport medium or protocol. This powerful feature enables GPB to be used over any RPC framework or transport protocol. Moreover, programmatic data access is simple and straightforward. The emphasis on simplicity and performance in GPB as well as their powerful features, qualify GPB as an ideal choice for information and data exchange in the middleware.

To create a type of information record, one has to write the specification as a GPB *message*. A message is composed of uniquely numbered fields. Each field has a name and a value and can be either *required* (must be included), *optional*, or *repeated* (a list of values of the same data type). A message file (which has the extension `.proto`) is then compiled using the GPB compiler into a simple data access class [12]. The compiler supports translation into multiple programming languages, such as `C++`, `Java`, and `Python`. The generated class provides functions to serialize/parse an information record specified by the message file to/from raw bytes. The serialized information can then be sent over the

wire to its destination, where it can be parsed and used. The binary serialization format of GPB is compact and both forward-and backward-compatible.

3.1.3 Google Protocol Buffers Messages

All data declared in the entity profile are parsed from the provided profile XML files and stored in programmatic classes corresponding to GPB message specifications. These classes have automatic serialization and deserialization functions that are used when transmitting and receiving data over the network. The current version of the middleware has six message types. They are listed as follows:

- ***Service***: the service message stores all the information needed to define a robotic service. Figure 3.4 illustrates a portion of the Service message file. One of the most important fields of Service is the "entityId", which stores the owning entity-unique identification.
- ***CtrlParam***: the ctrlparam message stores all the information needed to describe a control parameter of a service. A service can have zero or more control parameters. This information includes the name, data type, and all the other attributes obtained from the service profile file. The ctrlparam record does not store the actual parameter value(s), but rather the metadata of parameters.
- ***Param***: this message stores raw parameter information. It is used for the actual exchange of parameter data.
- ***Resource***: the resource message stores metadata needed to locate and identify a certain resource.
- ***Basic Profile***: the basic profile message stores all basic information on an entity. This information is obtained from the basic profile file.
- ***ByteStream***: this message is used as a carrier for a stream of raw data. It can store an arbitrary-sized array of bytes.

The middleware messages contain additional information for internal housekeeping and data transmission. These fields of information need not be specified in the entity profile and the developer need not be aware of them.

```
package Data;

message Service
{
  enum Type
  {
    SMPL = 0;
    CPLX = 1;
    ITRAV = 2;
  }

  required string entityId = 1;
  required int32 id = 2;
  required string name = 3;
  required Type type = 4;

  optional int32 depCount = 5 [default = 0];
  optional int32 fixedDepCount = 6 [default = 0];
  optional int32 baseDepCount = 7 [default = 0];
  optional bool isActive = 8 [default = true];
  optional bool isExec = 9 [default = false];
  optional bool isBlocking = 10 [default = true];
  optional bool isPausable = 11 [default = false];
  optional string descr = 12;
    /* ... continued ... */
}
```

Figure 3.4: The "Service" Message File.

A powerful feature of GPB is that additional fields can be easily introduced into a message or new messages can be added without worrying about modifying the communication medium or RPC interfaces. Moreover, existing applications do not need to be modified, unless they make use of the additional fields. Even so, the change will be minimal. Throughout the development cycle, numerous modifications and additions were made to the message types without any modification to the network middleware or RPC interfaces.

3.2 Network Middleware

Network middleware is a sophisticated piece of software that mediates between different distributed applications and the network in which they reside. Network middleware is considered a vital component in many network-based distributed systems since it eliminates the boundaries between distributed applications. Some of the functionalities a network middleware (and a distributed system in general) can provide are the following [45]:

- Network transparency: a typical network middleware hides all networking details from developers and provides network transparency to application developers.
- Scalability: a good network middleware can accommodate a growing number of applications and resources with graceful degradation in performance. Moreover, it can

support wide geographical distribution and can be easily managed, regardless of the number of distributed components.

- Easy access to resources: a network middleware should enable different distributed applications to access other resources and applications in an easy, efficient manner.

In the proposed robotic middleware, the network middleware is the central hub. It is the central component that handles all communications between the different robotic entities and controlling client programs.

There are a number of high-profile middleware technologies in use today. The CORBA standard specified by the OMG is still considered to be the de facto communications standard for heterogeneous software components running on multiple machines. As mentioned before, CORBA enjoys widespread use and has many implementations [2]. Various works in robotic middleware and distributed robotic systems are based on CORBA, such as those presented in [20], [46], [44], [43], [18], [26], [31], and [29].

Due to the design goals and the well-known reputation of CORBA, it was the natural choice for a network middleware. However, based on a number of technical reports and developer experiences, CORBA suffers from numerous shortcomings, such as [13]:

- Since CORBA was first conceived, it has always suffered from "design by committee" issues. A large number of complex functionalities were included in the standard that were either unimplementable or useless.
- Failure of the OMG to keep up with the rapid rise of web technologies.
- There are only a handful of implementations that fully implement the major features of the CORBA standard. Many of the existing implementations are either inadequate or incomplete.
- CORBA adds unnecessary development complexities and has a steep learning curve.
- CORBA lacks some important features, such as security and firewall traversal mechanisms.

SOAP technology is Web-oriented, and the use of XML for communication adds overhead. It is essential to have minimal communication overhead between entities in a robotic environment. Consequently, the ICE network middleware was an ideal choice for the middleware. Details are given in the next subsection.

3.2.1 The Internet Communications Engine

The first version of the ICE middleware was developed by ZeroC, Inc .[17], in 2003. It is regarded as an efficient alternative to other middleware programs such as CORBA and SOAP. From a design perspective ICE was greatly influenced by CORBA. However, it is simpler, smaller and more powerful. The reasons behind choosing ICE for the network middleware are the following:

- ICE has an efficient, component-based architecture. A great emphasis is put on eliminating any network overhead and bottlenecks.
- ICE makes use of a number of software design patterns, such as the proxy pattern.
- ICE fully supports multiple programming language and platforms. It supports `C++`, `Java`, `Python`, and other object-oriented languages. It also runs on various platforms, such as Windows and GNU/Linux.
- The Specification Language for ICE (SLICE), which is ICE's IDL, provides an easy and extensible way of specifying interfaces.
- ICE includes a firewall traversal mechanism, a feature that is missing in most network middleware programs.
- ICE compares favorably with other middleware programs and RPC technologies-such as `Java RMI`, and `WCF`-in terms of performance and scalability [3].
- A number of high-profile technology and defense companies are using ICE. Some of these companies are Hewlett-Packard, Skype, SGI, BAE Systems, and Lockheed Martin [16]. This indicates that ICE is a highly mature, proven, powerful tool.

A handful of robotic middleware programs and frameworks are starting to adopt ICE as their network middleware. Some of these are the works presented in [30] and [35]. The middleware has a multi-tier architecture where the clients are the robotic entities and front-end client programs.

3.2.2 ICE Services

In addition to providing a fully-featured and high-performance RPC platform, ICE also provides a comprehensive bundle of services and functionalities, such as data persistence

and event distribution. These types of services are widely used in distributed applications and mission critical systems. ICE services are designed to have high scalability, performance, and fault-tolerance. These services are discussed in the following subsections.

Glacier2

Glacier2 is a central service of ICE that implements firewall traversal mechanisms allowing any number of clients and servers to communicate freely across firewalls. Glacier2 enables servers to "push" data and updates to clients through a bidirectional connection feature, even if clients are residing behind a firewall that disallows incoming connections. In addition, Glacier2 provides security, authentication, and filtering mechanisms. Clients communicate with Glacier2 over SSL. Moreover, Glacier2 can control which resources and objects are accessed by the clients on the server(s). Deployment of Glacier2 requires minimal changes on the client side and no changes on the server side.

IceStorm

IceStorm is an efficient publish-subscribe mechanism that supports topic graphs. IceStorm allows for a federation of servers. This allows for scalability limited only by communication. Moreover, IceStorm provides persistent storage for topic subscriptions so that participants need not be reconfigured when restarted. Unlike many publish-subscribe frameworks where a message is composed of structured data, IceStorm messages are strongly typed classes and represented by operation invocations.

IceGrid

IceGrid is a collection of frameworks providing services for deployment and maintenance of high-performance grid computing applications using the collective power of multiple networked machines. IceGrid provides four main services:

- Location service: provides server discovery facilities, as well as load balancing and reliability mechanisms. Clients are directed to the most available server and can be redirected to other servers in case of failures without experiencing any interruptions. Moreover, servers need not be running at all times and can be activated on demand.

- Deployment service: provides facilities and tools to simplify and automate the process of deploying and maintaining server programs and data over multiple hosts in a network.
- Resource allocation service: this service facilitates the distribution of grid resources and clients among servers.
- Administration service: this service provides a powerful set of tools for control and configuration of all deployed applications and servers.

IcePatch2

IcePatch2 is an efficient service integrated with IceGrid for the distribution and patching of updates to different remote components. IcePatch2 employs compression and checksum techniques to ensure efficient transmission and data integrity.

Freeze and FreezeScript

The Freeze service provides facilities for persistent storage of ICE objects into a Berkeley DB [10] database. Freeze enables retrieval and instantiation of ICE objects on demand, such as instantiating servant objects based on a client's request. Moreover, Freeze enables applications to update and re-store objects. FreezeScript provides inspection and debugging tools for the stored objects.

3.2.3 SLICE

The Specification Language for the Internet Communications Engine (SLICE) is the provided IDL for ICE. Like any typical IDL, SLICE is a purely declarative language that is used to define object interfaces, operations, and exceptions that can arise from using these interfaces. SLICE effectively defines a contract between a client and a server in an application. The actual implementation of the interfaces must be provided by the application developer. The separation between interfaces and their implementations means that different components within an application need not be implemented in the same programming language. Moreover, SLICE provides mechanisms for specifying semantics for object persistence (in conjunction with Freeze).

Interface specifications must be written in a file with the extension "`.ice`" and compiled into the target programming language using the SLICE compiler. SLICE supports

a wide variety of object-oriented programming languages such as `C++`, `Java`, `Python`, `C#`, `Objective-C`, `Ruby`, and `PHP`. The generated source code files provide the link between the application and the ICE runtime environment and networking layer. The generated source code must be compiled and linked with the application logic code on both the client and the server sides.

3.3 Entity Component

The entity component is the part of the middleware that resides on the robot's onboard machine. This vital component is responsible for providing complete transparency to the developer by hiding all networking and concurrency details while presenting a simple and extensible interface that can easily interface with custom-developed robotic applications. The entity component is composed of two modules: the Entity Manager and the Entity Interfacer. These two modules are discussed in detail in the following subsections.

3.3.1 The Entity Manager

The Entity Manager (EM) is a modular piece of software that runs on every robotic entity machine that is an active actor in the environment. As its name implies, the EM manages the underlying robotic entity and acts as its interface to the outside environment. All a developer needs to do is to implement his custom robotic program and plug it in to the EM to connect the robot to the environment. A great emphasis was put on making the job of the developer as easy and fast as possible. The main goal is to have the developer worry about implementing his robotic application, and nothing more. To achieve that, the design goals for the EM are as follows:

- Providing complete transparency to the developer who is implementing the robotic application. The developer does not need to worry about details such as queueing, concurrency, multithreading issues, or networking.
- Providing reliability and safety features. The main implemented feature at the moment is a stop signal that is dispatched instantly in cases of client requests, disconnections, or improper behavior on the client side. The stop signal is guaranteed to reach the application in a minimal amount of time. However, it is the responsibility of the developer to make meaningful use of the signal.

- Maximizing the automation of the process. All a developer needs to do is to run his own application and the EM. Zero intervention is required after that, except in cases of severe errors.

The core design of the EM is based on the state machine software design pattern [23]. The EM uses different states to model the different states a robot can be in while running as an entity in the environment. Figure 3.5 illustrates the simplified UML state machine diagram of the EM. There are nine different states in the current version of the EM. They are explained as follows:

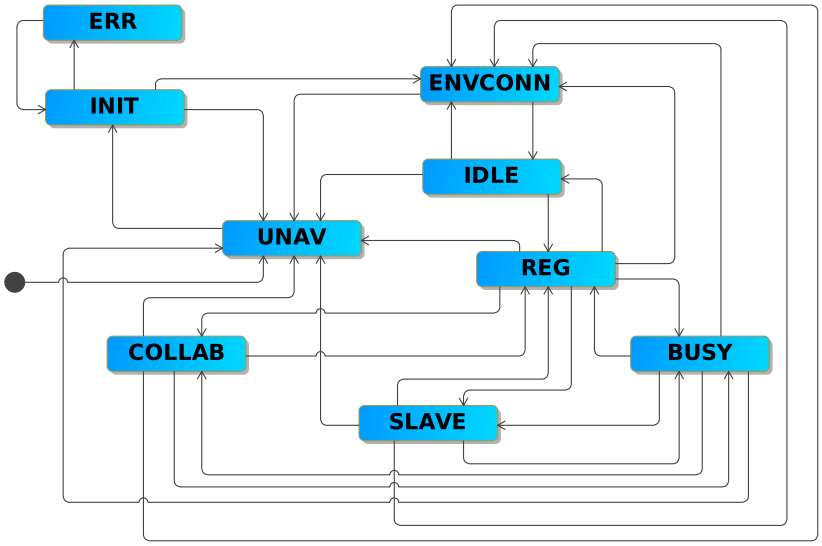


Figure 3.5: Entity Manager Simplified UML State Machine Diagram.

- ***Unavailable (UNAV)***: this is the initial state of the EM. The EM will first continuously try to establish a link to the robotic application. Once the link is established, the EM switches to the *INIT* state. It should be noted that the EM will always switch to *UNAV* whenever the link to the entity is broken, regardless of the current state.
- ***Initialize (INIT)***: in this state the EM will attempt to parse the entity profile (basic and services profiles) provided by the developer. In case of parsing errors the

EM will switch to the *ERR* state. Otherwise, the EM will switch to the *ENVCONN* state.

- ***Environment Connect (ENVCONN)***: in this state the EM will continuously attempt to connect to the server. After a connection is established, the EM will then upload the entity's basic profile to the server. This finalizes the connection and registration process.

- ***Idle (IDLE)***: this is the default state of any connected robot. In this state the EM is awaiting check-out requests from a client. Whenever a check-out request is received and granted, the EM uploads the service profile to the user's client program and switches to the *REG* state. It should be noted that the EM will always switch to *IDLE* whenever the client releases the entity or the connection with the client is lost, regardless of the current state.

- ***Registered (REG)***: in this state the entity waits for the user's data and tasks. When data are received (such as service control parameters or resources), the EM relays this data to the robotic application and returns the status to the client (whether the update was granted or denied). Whenever a task is received, the EM will switch to either the *COLLAB*, the *SLAVE*, or the *BUSY* state, depending on the requested service.

- ***Collaborative (COLLAB)***: in this state the entity will act as a master to one or more entities involved in a collaborative task. The EM will dispatch collaboration requests to all participating entities. Since the collaborating entities must have been checked out by the same client, the client ID is included in the request for the purpose of authentication. After all participating entities authenticate and acknowledge the request, the master entity will proceed to dispatch tasks in the pre-determined order.

- ***Slave (SLAVE)***: the EM will switch to this state whenever a collaboration request is received from a master entity. The EM will patiently await for one or more tasks from the master entity.

- ***Busy (BUSY)***: after a task is granted by the robotic application, the EM will switch to *BUSY*. The EM will remain in this state for as long as it takes for the task to complete. Then it will switch to the previous state. There are two special cases to consider in *BUSY* state:

  - Whenever a stop command is received, the EM instantly dispatches a stop signal to the robotic application and switches to the *REG* state.

  - Whenever the user releases the entity or abruptly disconnects, the EM will dispatch a stop signal to the robotic application and switches to the *IDLE* state.

- ***Error (ERR)***: the EM will switch to this interactive state if it failed to parse the entity profile. The EM will notify the user of the error and provide an option to either fix the files or quit. User intervention is needed at this state.

The current version of EM supports three special requests, or commands. In the context of the middleware, a command is a special message that has a higher precedence over other messages, and can interrupt the normal behavior, or flow of execution of an entity. The three commands are listed below.

- Stop command: this command will cause the EM to instantly dispatch a stop signal to the robotic application. The stop command should halt all physical activities of a robot, and it is up to the application developer to make effective use of the signal. It should be noted that a stop command, regardless of its cause, has the highest priority among other commands and will instantly halt all running services or operations at any state.

- Reset command: this command causes the EM to halt execution of a task (if any) by sending a stop signal to the underlying robot, and then proceeds with the next task in the queue (if any).

- Clear command: this command will halt the execution of any task and clears the EM's tasks queue.

The UML class diagram of the EM is illustrated in Figure 3.6. The main classes of the EM are explained as follows:

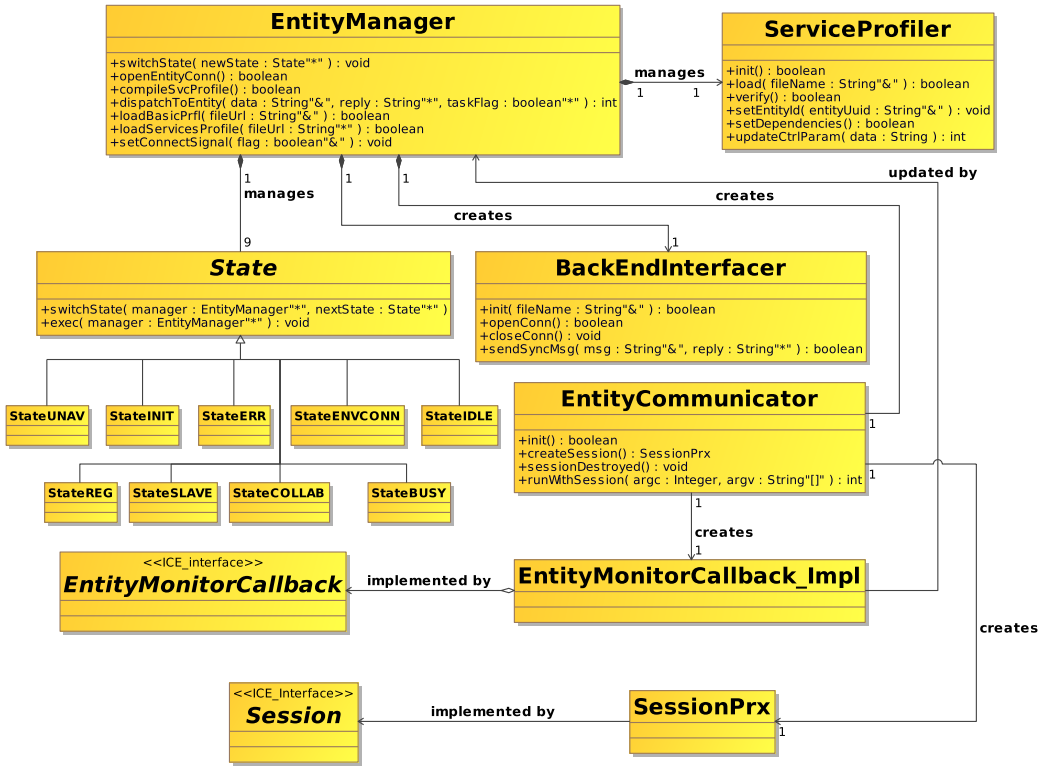


Figure 3.6: Entity Manager Simplified UML Class Diagram.

EntityManager

This is the central class of the EM module. It acts as the control hub for all the other parts. The `EntityManager` performs the following functions:

- Instantiating and managing the other classes of the EM.
- Creating and maintaining the communications link with the robotic application through the Entity Interfacer (explained later).
- Maintaining all of the data and information related to the state of the robot, as well as incoming and outgoing data queues.

ServiceProfiler

The `ServiceProfiler` class acts as a registry for the robot's services. It stores and manipulates all data and information for all services. This completely eliminates the need for implementing complex routines to keep track of the parameters and resources of a service or keeping track of its dependencies and readiness. The functionalities of `ServiceProfiler` can be summarized as follows:

- Maintaining all data and information for each service such as its control parameters, resources, and dependencies.
- Ensuring that all the dependencies of a service are satisfied before dispatching a task request to the robot.
- Toggling services on/off according to commands from the robotic application.

BackEndInterfacer

This class is one of the two classes in the communications part of the EM. The `BackEndInterfacer` is responsible for creating and maintaining a TCP socket-based communications link with the robotic application. Its functionalities can be summarized as follows:

- Efficient information and data exchange with the robotic application. Data include parameters, resources, task requests, raw data, and text-based general messages.
- Periodic heartbeat pings to ensure liveliness of connection.

State

`State` is the parent class of the nine state classes. The core functionalities and logic of the EM and the state machine management are distributed among all of the `State`-based classes. Each `State` sub-class implements the logic of one of the EM states discussed earlier. The classes are as follows:

- `StateUNAV`
- `StateINIT`

- `StateENVCONN`
- `StateIDLE`
- `StateREG`
- `StateCOLLAB`
- `StateSLAVE`
- `StateBUSY`
- `StateERR`

EntityCommunicator

This is the second class in the communications part of EM. `EntityCommunicator` is responsible for creating and maintaining the communications link between the EM and the server. It dispatches all outgoing data to the server and updates the `EntityManager`'s queues with incoming data.

SessionPrx

This class is the generated proxy implementation of the `Session` *Glacier2*-based ICE interface. The EM uses functions defined in this interface to interact with the server and client programs. Figure 3.7 illustrates a portion of the interface definition. The key functions in this interface are as follows:

- `sendBasicPrfl`: this function is used to upload the entity basic profile to the server.
- `sendSvcPrflRecord`: this function is used to send a service record to the controlling client (who has just checked out the entity). The server will relay these data to the client program.
- `sendToClient`: this function is used to send a simple string-based message to the controlling client. This is used for sending informative messages, updates, and feedbacks.
- `sendToClientDataSeq`: this function is used to send a stream of raw data to the controlling client. Raw data can be sensor readings, a list of controlling parameters, and so on.

EntityMonitorCallBack_Impl

This class is the implementation of the `EntityMonitorCallBack` callback ICE interface (illustrated in Figure 3.8). This interface acts as the contract between the EM and the server. The server calls functions in this interface to push data and updates to the EM. Some of the important functions in the interface are the following:

- `newClientMsg`: this function is used to relay simple client messages and requests to the robot, such as a check-out request message.
- `newClientMsgDataSeq`: this function is used to relay a sequence of raw data from the controlling client to the robot. This data could be a resource file, for instance.
- `stopSignal`: this function relays a stop command from the client to the entity. Moreover, the server uses this function to alert the EM in case the client releases the robot or abruptly disconnects.
- `resetSignal`: this function relays a reset command to the robot.
- `clearSignal`: this function relays a clear command.

```
#include <Ice/BuiltinSequences.ice>
#include <Glacier2/Session.ice>

#include <commutil.ice>

  /* ..... */

  interface Session extends Glacier2::Session
  {
    void setCallback (EntityMonitorCallback *entityCb);
    ["ami"] long sendBasicPrfl (string entityType, string msg);
    ["ami"] long sendSvcPrflRecord (string entityType, int recordId,
                                    int count, CommUtil::DataRecord svcRecord);
    ["ami"] long sendToClient (string clientId, string entityType,
                               string msgType, string msg);

                /* ... continued ... */
  };

  /* ... continued ... */
```

Figure 3.7: Portion of the Entity Session SLICE Interface.

```
#include <Ice/BuiltinSequences.ice>
#include <Glacier2/Session.ice>

#include <commutil.ice>

  /* ..... */

  ["ami"] interface EntityMonitorCallback
  {
    void newClientMsg (long timestamp, string clientId, string entityType,
                       string msgType, string msg);
    void newClientMsgDataSeq (long timestamp, string clientId,
                              string entityType, string msgType,
                              CommUtil::DataSeq dataSeq);
    void newClientMsgDataRec (long timestamp, string clientId,
                              string entityType, string msgType,
                              CommUtil::DataRecord dataRec);

                  /* ... continued ... */
  };

  /* ... continued ... */
```

Figure 3.8: Portion of the Entity Callback SLICE Interface.

3.3.2 The Entity Interfacer

The Entity Interfacer (EI) is the second module of the entity component of the middleware. The current version of the EI provides a TCP socket-based communications link with the EM, as well as a simple and extensible API that the robotic application must interface with to be able to be seamlessly plugged in to the environment. The design goals of the EI are as follows:

- Simplicity: the current implementation of the EI provides a small set of easy-to-use functions. All the developer needs to do is to set a required number of callback functions in his application to enable the EI to feed the robotic program with all the needed information and data. Moreover, the robotic program interacts with the environment through the EI function set.
- Transparency: the EI hides all details involving the transmission, packing, and unpacking of data and so on. The developer does not need to know how data are handled and transmitted.
- Robustness: the EI is completely reliable and thread-safe. Moreover, its communications function runs in a totally isolated thread and external applications cannot tamper with its execution flow.

- Safety: the EI will instantly dispatch a stop signal to the entity application the instant the link to the EM is broken. Since the EI is closely associated with the robotic application, the stop signal will never be lost.

Currently, there is only a `C++` implementation of EI. However, its modular and object-oriented design allows for fast porting to other programming languages such as `Java` or `Python`. Figure 3.9 illustrates the UML class diagram of the EI. The main classes of the EI are explained as follows:

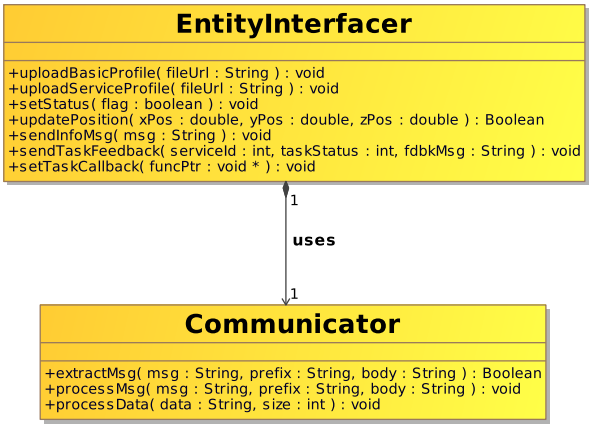


Figure 3.9: Entity Interfacer UML Class Diagram.

EntityInterfacer

This class provides the main API for robotic applications developers. A robotic application can interact with the environment by calling functions in this class. Moreover, this class interacts with the robotic application through a set of callbacks. Some of the key functions provided to the developers are as follows:

- `uploadBasicProfile`: used to upload the developer-defined basic profile file to the EM.
- `uploadServiceProfile`: similar to the previous function, this function is used to upload the services profile file.

- `setStatus`: used to set the current status of the entity. In the middleware semantics, a checked-out entity can be either busy executing a task or ready to receive a task request.
- `sendInfoMsg`: this function can be used to send simple informative messages to the controlling client program.
- `sendTaskFeedback`: this function is used to report a task feedback to the controlling client. Each task is associated with a unique service.

Communicator

This class represents the communication port of the EI, and is responsible for establishing and maintaining the communications link with the EM. It handles all operations related to the transmission, packing and unpacking of data. This provides complete transparency to the developer.

3.4 Server Component

The server component is the central hub of the middleware. Its main functions are to keep track of all connected entities and clients, as well as providing efficient and reliable message routing between different clients and entities. The server is implemented in `C++` and ICE. The main design goals of the server are as follows:

- Minimal information and data are maintained per entity or client.
- The server should be stateless, it should not interpret passing messages and is invariant of the current state of an entity or a client.
- An entity or a client are universally identifiable in the environment by a Universally Unique IDentifier (UUID). Clients and entities are able to communicate using their UUIDs.
- The server should provide simple and extensible communication interfaces to ensure flexibility and easy expansion.

The server is composed of two sub-modules: the *entity* server and the *front-end* server. These two loosely coupled modules communicate through a publish-subscribe mechanism based on IceStorm to bring about the desired functions. The current version of the server supports GNU/Linux-based platforms. It can be easily ported to other platforms. The server components are discussed in detail in the following subsections.

3.4.1 Entity Server

The entity server is a modular piece of software that acts as a scalable registry and message router for all connected entities. The entity server performs the following functions:

- Keeping track of all online entities.
- Efficient and scalable message routing from entities to controlling clients and vice-versa (through the front-end server).
- Efficient and scalable message routing between different entities.
- Keeping the front-end server updated with the latest changes in connected entities' status.
- Instantly dispatching a stop signal to checked out entities whose client abruptly disconnects without properly releasing them. (through an update from the front-end server).

As per the design goals, the entity server maintains a minimal amount of information on each entity and does not interpret passing messages or data. For each connected entity, the entity server maintains the following information:

- An entity UUID-to-callback adapter mapping (for pushing data and updates to each connected entity).
- An entity UUID-to-basic profile mapping.

Figure 3.10 illustrates the simplified UML diagram of the entity server. The main classes of the server are discussed follows:

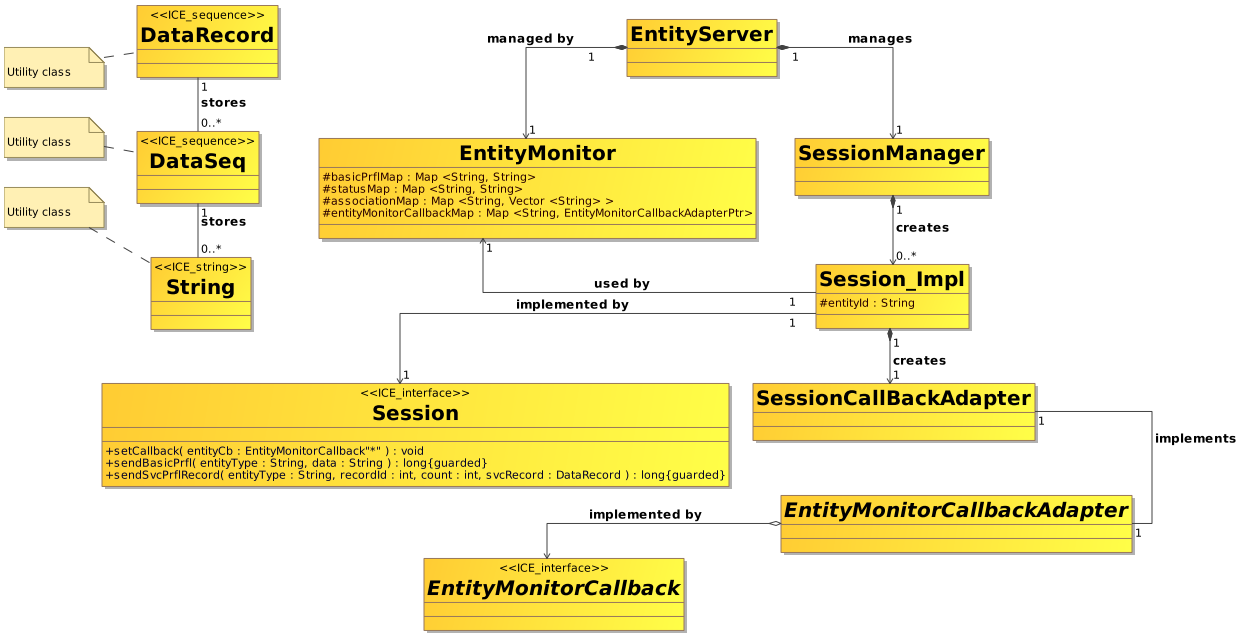


Figure 3.10: Simplified UML Class Diagram of the Entity Server.

EntityServer

This is the main controller class of the entity server. It is responsible for initiating the ICE runtime, as well as initializing the communicator to accept incoming entity connections and instantiating and maintaining the `EntityMonitor` and the `SessionManager`. Moreover, it performs other functionalities that are typical for a server, such as thread pool and resources management.

EntityMonitor

The `EntityMonitor` is the central class of the server. It implements the entire logic and behavior of the server. The main functionalities of the class are as follows:

- Maintaining a UUID-to-callback adapter mapping for each active entity session. The callback adapter is used to communicate with the owning entity through the ICE push mechanism.

- Storing and maintaining the basic profiles of all connected entities.
- Updating the front-end server with any change in the status of the connected entities.

SessionManager

As its name implies, this class is responsible for creating and maintaining a session for each connected entity. Moreover, it is responsible for the destruction and cleanup of sessions.

Session

The `Session` class is the generated `C++` interface of the ICE session interface shown in Figure 3.7. This class implements a Glacier2-based "push" session and specifies all of the operations through which the EM of a robot can interact with the server. Moreover, the class houses all of the mechanisms needed for marshalling/unmarshalling requests, as well as communicating with the server.

Session_Impl

This class is the custom implementation of the `Session` interface. A connected entity will call its session functions to interact with the entity server. The `Session_Impl` will in turn invoke the designated functions in the `EntityMonitor`. A `Session_Impl` object is created for each connected entity. The implemented functions of this class are the same those in Section 3.3.1.

EntityMonitorCallBack

The `EntityMonitorCallBack` class is the generated implementation of the SLICE callback interface. It specifies all function callbacks through which the entity server can interact with the EM of a connected entity. Figure 3.8 illustrates a portion of the interface implementation.

EntityMonitorCallBackAdapter

As with the `Session_Impl`, this class is the custom implementation of the `EntityMonitorCallBack` ICE interface (shown in Figure 3.8). The `EntityMonitor` maintains an instance of this class for each connected entity. The server uses the

`EntityMonitorCallBackAdapter` functions to "push" data and updates to the EM of the owning entity. The functions of this class are equivalent to those mentioned in Section 3.3.1.

3.4.2 Front-End Server

The front-end server functionalities are very similar to those of the entity server, except that the former is responsible for handling all connected clients. The front-end server maintains a minimal amount information for each connected client. The functionalities of the front-end server are summarized as follows:

- Keeping track of all connected clients.
- Keeping all clients updated with the latest list of entity basic profiles (through updates from the entity server).
- Handling check out and release requests from clients.
- Efficient and scalable message routing from controlling clients to checked-out entities and vice-versa (through the entity server).
- Instantly notifying the entity server of any dropped clients who had previously checked-out entities but went offline before properly releasing them.
- Parsing and uploading of global services profiles (if available) to be downloaded by connected clients on request.

The front-end server maintains the following data:

- A mapping between each client's UUID and its callback adapter (similar to the entity server).
- A copy of the UUID-to-entity basic profile mappings (which are frequently updated by the entity server). This information is sent to each recently connected client. Any updates are immediately relayed to all connected clients.
- A mapping between each client and its checked-out entities. This is essential to ensure that messages reach their correct destination and that the designated entities are notified in case their controlling client abruptly disconnects.

The UML class diagram of the front-end server is illustrated in Figure 3.11. The descriptions of the front-end server classes are given below.

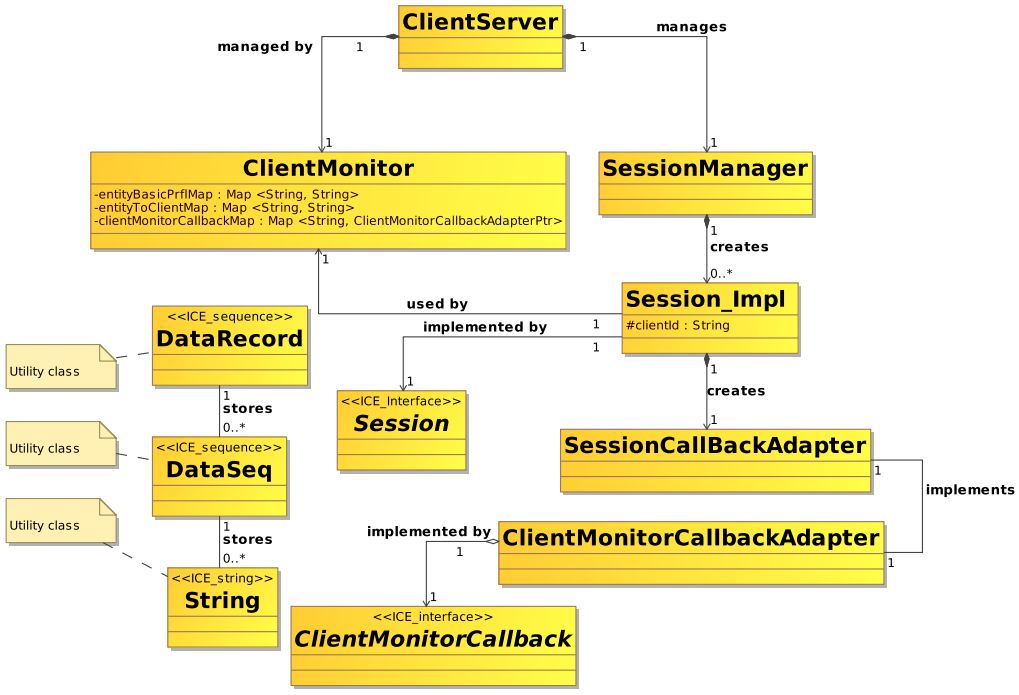


Figure 3.11: Simplified UML Class Diagram of the Front-End Server.

FrontEndServer

This is the main controller class of the front-end server. Its functions are very similar to that of the `EntityServer` class of the entity server. The `FrontEndServer` instantiates and maintains the `EntityMonitor` and the `SessionManager`.

ClientMonitor

The `ClientMonitor` is the central class of the server. It implements the entire logic and behavior of the server. The main functionalities of the class are as follows:

- Maintaining a UUID-to-callback adapter mapping for each active client session. The callback adapter is used to communicate with the owning client through the ICE push mechanism.
- Storing and maintaining a mapping between each client and its checked-out entities. This is used for relaying data and updates to the designated entities (via the entity server) and vice-versa.

SessionManager

This class is responsible for the creation, maintenance and destruction of client sessions.

Session

The `Session` class is the generated implementation of the ICE session interface declared in "`frontendcomm.ice`" (for the front-end server). Figure 3.12 illustrates a portion of the interface implementation.

Session_Impl

This class is the custom implementation of the `Session` interface. A connected client will call its session functions to interact with the server. The `Session_Impl` will in turn invoke the designated functions in the `ClientMonitor`. Some of the important functions in `Session_Impl` are:

- `sendRequest`: this function is used to send various types of requests to both the server and connected entities. For the server, it can be used to request status updates for entities, as well as for downloading the global services profile (please see Appendix A for more information). For an entity, the function is used to send check-out and release requests.
- `sendToEntityDataSeq`: this function is used to send a sequence of raw data to a previously checked-out entity.
- `sendStopSignal`, `sendClearSignal`, and `sendResetSignal`: these functions are used to send stop, clear, and reset commands respectively (please see Section 3.3.1).

ClientMonitorCallBack

This class is the generated implementation of the SLICE callback interface, defined in `frontendcomm.ice`. It specifies all function callbacks through which the server can interact with the client program. Figure 3.13 illustrates a portion of the interface implementation.

ClientMonitorCallBackAdapter

As with the `Session_Impl`, this class is the custom implementation of the `ClientMonitorCallBack` interface. The `ClientMonitor` maintains an instance of this class for each connected client. The server uses the `ClientMonitorCallBackAdapter` functions to "push" data and updates to the client program. Some of the important functions of `ClientMonitorCallBackAdapter` are:

- `newBasicPrflMsg`: this function is used to relay the basic profile of an entity to the client.
- `newSvcRecordMsg`: this function is used to relay the raw data of a service record originating from a recently checked-out entity.
- `newEntityMsgDataSeq`: this function is used to relay a sequence of raw data from a checked-out entity to the controlling client.
- `newUpdateMsg`: this function relays a simple string-based message from an entity to its controlling client.

```
#include <Ice/BuiltinSequences.ice>
#include <Glacier2/Session.ice>

#include <commutil.ice>

  /* ..... */

  interface Session extends Glacier2::Session
  {
    void setCallback (ClientMonitorCallback *clientCb);
    ["ami"] long sendToEntity (string entityId, string entityType,
                               string msgType, string msg);
    ["ami"] long sendToEntityDataSeq (string entityId, string entityType,
                                      string msgType, CommUtil::DataSeq dataSeq);

                /* ... continued ... */
  }

  /* ... continued ... */
```

Figure 3.12: Portion of the Client Session SLICE Interface.

```
#include <Ice/BuiltinSequences.ice>
#include <Glacier2/Session.ice>

#include <commutil.ice>

  /* ..... */

  ["ami"] interface ClientMonitorCallback
  {
    void newBasicPrflMsg (long timestamp, string entityId, string entityType,
                          string msg);
    void newSvcRecordMsg (long timestamp, string entityId, string entityType,
                          int recordId, int count,
                          CommUtil::DataRecord svcRecord);
    void newEntityMsg (long timestamp, string entityId, string entityType,
                       string msgType, string msg);

                /* ... continued ... */
  }

  /* ... continued ... */
```

Figure 3.13: Portion of the Client Callback SLICE Interface.

3.4.3 Inter-server Communication

As mentioned earlier, the two servers communicate through a publish-subscribe mechanism. This might seem an overkill since publish-subscribe is usually used where multiple entities subscribe to a specific topic and receive notifications related to that topic. The reason for choosing such a mechanism is to ensure that the two servers are loosely coupled and to have a simple and extensible communication interface between them. In the current version of the middleware the entire server component runs on a single machine. Taking into account future expansions which might involve running the two servers on two different machines or having duplicate servers (for scalability), the publish-subscribe mechanism is the ideal way to ensure minimal effort in expansion. Figures 3.14 and 3.15 illustrate portions of the topic interface between the two servers.

```
#ifndef ENTITYTOPICS_ICE_
#define ENTITYTOPICS_ICE_

#include <commutil.ice>

module EntityComm
{
  interface EntityToFrontEndComm
  {
    void entityBasicPrflUpdate (string entityId, string entityType, string msg);
    void entitySvcRecord (string entityId, string entityType, int recordId, int count, \
      CommUtil::DataRecord svcRecord);
    void entityMsg (string entityId, string entityType, string msgType, string msg);
    void entityMsgDataSeq (string clientId, string entityType, string msgType, \
      CommUtil::DataSeq dataSeq);
    void entityMsgDataRec (string clientId, string entityType, string msgType, \
      CommUtil::DataRecord dataRec);
    void entityUpdate (string entityId, string entityType, string msgType, string msg);
  };
};

#endif // ENTITYTOPICS_ICE_
```

Figure 3.14: The Entity Server Topic SLICE Interface.

```
#ifndef FRONTENDTOPICS_ICE_
#define FRONTENDTOPICS_ICE_

#include <commutil.ice>

module FrontEndComm
{
  interface FrontEndToEntityComm
  {
    void clientMsg (string clientId, string entityId, string entityType, string msgType, \
      string msg);
    void clientMsgDataSeq (string clientId, string entityId, string entityType, string msgType, \
      CommUtil::DataSeq dataSeq);
    void clientMsgDataRec (string clientId, string entityId, string entityType, string msgType, \
      CommUtil::DataRecord dataRec);
    void clientRequest (string clientId, string entityId, string entityType);
    void clientUpdate (string clientId, string entityId, string entityType, string msgType, \
      string msg);
    void stopSignal (string clientId, string entityId, string entityType, string msg);
    void clearSignal (string clientId, string entityId, string entityType, string msg);
    void resetSignal (string clientId, string entityId, string entityType, string msg);
  };
};

#endif // FRONTENDTOPICS_ICE_
```

Figure 3.15: The Front-End Server Topic SLICE Interface.

3.5 Overall Architecture

Figure 3.16 illustrates the overall architecture of the middleware, showing how the different components interface with each other. It also shows how a robotic application and a client

program interface with the middleware.

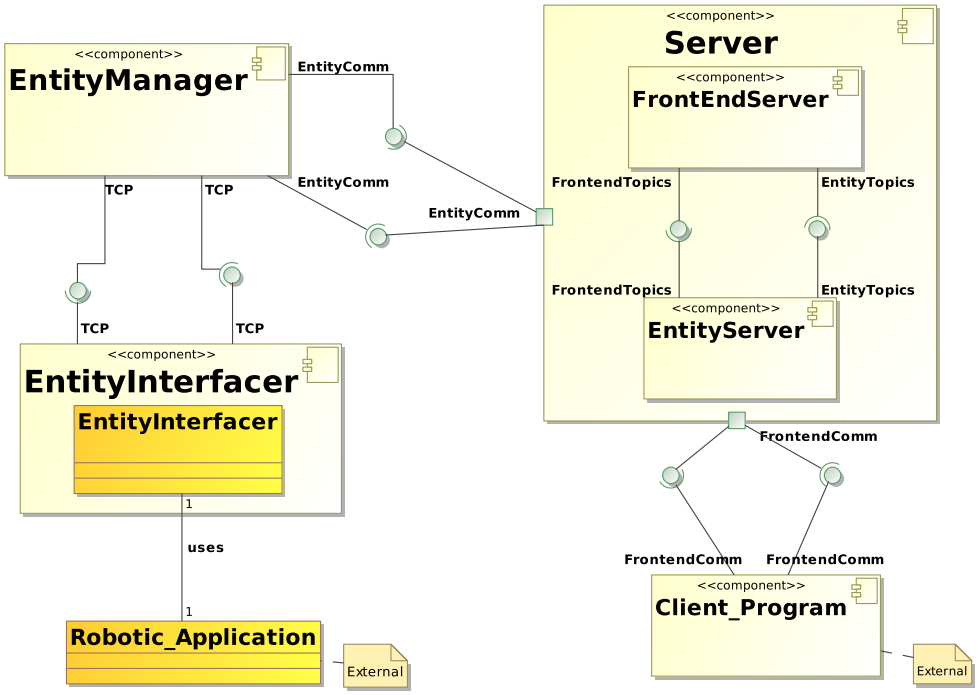


Figure 3.16: Middlware UML component diagram.

3.6 Chapter Summary

In this chapter, the proposed middleware was described in a bottom-up fashion, starting from the information base and technologies used, and going up the system architecture, components, and integration. Firstly, the representation of a robotic entity in a multi-robot environment based on the middleware is described, as well as the information the robotic developer needs to provide for each robot. The Google Protocol Buffers was chosen as an extensible and lightweight mechanism for data and information exchange in the middleware. Moreover, the Internet Communications Engine was used as the network middleware. This chapter also described the architecture of the proposed system. Each component was described in detail including its design goals and choices, as well as its

software and implementation details. Moreover, the discussion covered how the different components work together in the system. The middleware models each entity in the environment as a single actor. A great emphasis was put on reducing the development times of experiments and demos involving multi-robots, as well as making the life of developers as easy as possible. The adherence to rigorous and professional design and implementation methodologies helped to produce a robust and modular system that can be easily expanded and updated.

Chapter 4

Experimental Setup for Multi-Robot System

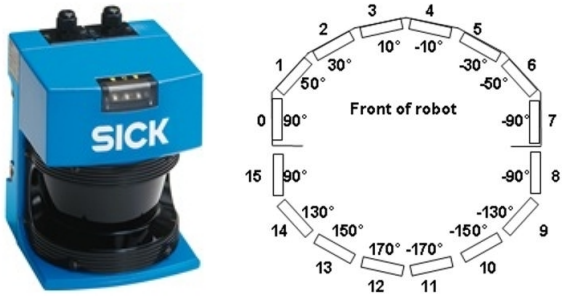
The middleware was deployed in a real environment to test and demonstrate its ability to accommodate heterogeneous robot units. To achieve that, two different types of robots were used: a multi-purpose mobile robot and a stationary platform mounted robotic arm. Each robot comes from a different vendor and has its own software libraries and APIs. A testing module was implemented for each robot. Each module is composed of a set of services ranging from simple ones, such as travelling an arbitrary distance, to complex ones such as tracking and picking colored objects. Moreover, a sophisticated collaborative scenario was implemented to fully test the performance and capabilities of the middleware. The entity component of the middleware seamlessly integrated the two robotic applications into a highly reliable multi-robot research environment. The next sections discuss the different robotic entities in detail including their hardware features, SDKs, and the applications developed for each one. Moreover, the discussion will cover a collaborative scenario that involves the two robots working together to achieve a goal.

4.1 PeopleBot Mobile Robot

PeopleBot [11], shown in Figure 4.1, is a multi-purpose differential-drive robot developed by MobileRobots, Inc. [4], for research and applications involving Human-Robot Interaction (HRI), cooperative robotics, performance, exhibition, education, and much more. It is equipped with chest-level extension and a touch screen to facilitate interaction with users.

Figure 4.1: The PeopleBot Mobile Robot [11].

PeopleBot comes with a set of sensors, as well as range-finding devices for navigation and obstacle avoidance purposes. For range-finding, PeopleBot features two arrays of sonar sensors at the bottom base and the top extension. Moreover, PeopleBot can be optionally equipped with the SICK LMS-200 laser device for higher navigational and obstacle avoidance accuracy. Figure 4.2 illustrates the laser device and the lower sonar array found on the PeopleBot. In addition to range-finding devices, PeopleBot is able to recognize objects and people, as well as track colors using an optional 120-degrees pan/tilt/zoom (PTZ) camera which can be installed on top or underneath the top deck. PeopleBot is equipped with infrared table sensors, as well as a 2-DOF gripper with pressure sensors to sense tables and grab objects respectively. Moreover, an array of forward and backward bumpers equipped with limit switches enables PeopleBot to navigate in narrow spaces with reasonable accuracy. The provided joystick enables basic manual driving control of the robot.



(a) SICK LMS [11]. (b) Lower Sonar Array [47].

Figure 4.2: PeopleBot Range-Finding Devices.

4.1.1 Pioneer SDK

PeopleBot, along with different mobile robots developed by MobileRobots, Inc., is equipped with an onboard computer and comes with the Pioneer SDK [6]. Pioneer SDK is a collection of high-level software libraries that can run on both Linux and Windows platforms, and provides a powerful framework for developing robotic applications. The central component of the SDK is the Advanced Robotics Interface for Applications (ARIA) core library. ARIA provides a collection of APIs and a framework for controlling and communicating with all underlying robotic hardware components, as well as installed accessory devices. ARIA provides a rich set of utilities for writing robotic control software, as well as cross-platform applications. In addition, ARIA provides different levels of control by allowing simple control of hardware, as well as providing optional high-level control actions combining several simple motion behaviors into a single unified behavior. One of the core utilities provided by ARIA is the ArNetworking library which provides a simple extensible client-server networking framework for writing networked robotic control applications. ArNetworking provides facilities for multi-robot information exchange. It includes a set of predefined services for remote control of robots, as well as remote data and information exchange. Some of these services are teleoperation, sensor readings, graphical visualizations, and notifications. ARIA supports two types of control connections: a serial-based connection to the robot's microcontroller, and a TCP-based connection to a simulator tool such as MobileSim (discussed later). An ARIA-based application will first attempt to connect to the robot through the serial interface, and if unsuccessful, will then attempt to connect to the simulator. The order of precedence can be specified by the developer. ARIA is written in `C++` and provides wrappers for other programming languages, such as `Java` and

`Python`. It should be noted that ARIA is open-source software released under the GNU GPL license. The other components of the Pioneer SDK are described below:

- **ARNL**: the Autonomous Robotic Navigation and Localization (ARNL) is a collection of proprietary libraries based on ARIA that provide intelligent localization and navigation services enabling the robot to keep track of its position and intelligently plan its trip to a destination point. ARNL provides three different techniques for localization. The first technique uses laser readings from the installed SICK LMS range-finding device along with odometry readings to accurately determine the robot's position through the application of the Monte-Carlo Localization (MCL) algorithm. The second technique is similar to the first one except that it uses sonar readings. The third technique is used for outdoor localization. It infuses readings from an optional GPS device into the MCL algorithm. For navigation, ARNL employs a grid-based search method to compute the shortest safe path to a destination, taking into account any detected obstacles or forbidden zones. Throughout its trip, the robot is fed accurate translational and rotational velocities data. Moreover, ARNL continuously computes the robot's path during its trip to account for dynamic changes in the environment. Like ARIA, ARNL is written in `C++` and provides wrappers for other programming languages, such as `Java` and `Python`.

- **Mapper3**: it is a tool for creating environment maps that can be used as the basis for intelligent localization and navigation, either in a real environment or a simulation. Drawing a map starts by having the robot thoroughly scan its environment using its SICK range-finding laser device. This process produces a 2-dimensional map file (`.2d`) that is processed by Mapper3 to produce the finalized map file (`.map`). This map file can be further edited by adding obstacles, specifying home and target points, forbidden zones, and so on.

- **MobileSim**: it is a rich graphical-based robot simulator that enables accurate simulation of robotic applications before deployment on real robots. MobileSim simulates the robot's serial control connection through a TCP-based connection. MobileSim can accurately simulate a real environment map including walls, obstacles, sectors, and designated locations using the provided Mapper3-created map file. MobileSim provides various features such as support for a wide range of MobileRobots robots, simulation of multiple robots, configurations of a robot's range-finding devices, interactive control of objects in the environment including robots and obstacles, as well as many useful features [6]. A user can add custom devices and robots to MobileSim by including an ARIA parameter file.

- **MobileEyes**: it is a client program providing a graphical-based user interface for monitoring and controlling a robot remotely. MobileEyes can connect remotely to applications running on the robot or MobileSim through ARIA and ArNetworking. Through MobileEyes, a user can load an environment map and view information such as the robot's position on the map, sensor readings, and battery levels [6]. MobileEyes also enables teleoperation of robots, configuration of control parameters, and controlling navigation and localization software. Moreover, it can interface with installed accessory devices, such as the onboard PTZ camera.

- **ACTS**: the Advanced Color Tracking System (ACTS) is an easy-to-use cross-platform video processing software that can be used to identify and track colored objects. ACTS supports 32 individually trainable color channels and can process 30 frames per second [6]. As such, it can track up to 320 mobile objects under different lighting conditions. Information on tracked objects can be easily retrieved by ARIA-based programs through the provided APIs. Moreover, ACTS comes with a graphical-based training application that is used to configure and tune the different channels. ACTS is a highly useful tool for applications involving object tracking, vision, and HRI.

- **ARCOS**: the Advanced Robotics Control Operating System (ARCOS) is a low level operating system that is considered the brain of any MobileRobots robot. It is responsible for managing all the low-level details of the robot, such as motor control, firing the sonar, collecting and reporting sonar and odometry data, as well as stalling the motors in response to emergency protection triggers such as a triggered bumper. ARCOS is based on a client-server architecture where the server handles all the robot's low-level details and the client (an ARIA-based application) controls the robot through a serial-based connection to the server [11]. Moreover, ARCOS provides a client-server interface which can be used to achieve tighter low-level control, or to implement custom control routines and reactive planning applications.

Two identical PeopleBot robots were used in the experiment. An identical set of services was implemented for each one. They are discussed in the following subsection.

4.1.2 Implemented Services

A number of services were developed for the PeopleBot. These services range in complexity and utilize all the available hardware components installed on the robot. Each service has a number of input and configuration parameters, as well as feedback parameters. Some

complex services are standalone services while others are composed of simpler services or built on top of others. `C++` was used for implementation since it is natively supported by the Pioneer SDK. The different services were implemented in different classes and grouped into a single module. A total of six services were developed.

Driving

This service implements a very simple straight motion of the robot for an arbitrary distance at a given velocity. Range readings from the sonar arrays and the SICK LMS laser device, as well as the bumpers are used to avoid hitting an obstacle in the robot's path. Any range reading beyond a certain threshold, or a triggered bumper will instantly send a stall signal to the robot's motors. The inputs to this service are the distance (in millimeters), velocity (speed and direction), and the range reading threshold.

Turning

This service implements simple rotational motion of the robot. The infrared table sensors are employed to avoid hitting any table or elevated surface while turning. A triggered table sensor will instantly stall the motors and cancel the turning. The inputs to this service are the rotational velocity and an angle value.

Operate Gripper

This service is used to demonstrate the capabilities of the 2-DOF gripper. The inputs are the direction of the gripper and a boolean flag storing the desired gripper mode (opened/-closed).

Wander with Collision Avoidance

This service implements random wandering in an environment while employing a fuzzy-logic-based obstacle avoidance mechanism. Readings from the range-finding devices are used as an input to the obstacle avoidance mechanism, which, in turn, controls the robot's translational and rotational velocities in an intelligent manner. Moreover, the bumpers and table sensors are used to avoid hitting obstacles in narrow spaces and hitting tables respectively.

Navigation

This complex service involves the robot navigating to a designated location in an environment. This service employs the fuzzy logic-based obstacle avoidance mechanism discussed earlier, as well as a simple localization technique using a map of the environment. The input to this service is a map file of an environment and a designated location.

Table Object Tracking and Grabbing

This complex service employs the gripper and the PTZ camera to track and grab a colored object placed on a table edge. The input to this service is a single value specifying a target color. The robot starts by scanning its environment looking for a significantly sized object having the specified color. After an object is found the robot will cautiously close in on the object while continually adjusting its translational and rotational velocities to ensure an accurate gripping position. After approaching the table, the robot will deploy its gripper to grab the object.

4.2 Cyton Alpha Robotic Arm

The Cyton Alpha, shown in Figure 4.3, is a 7-DOF 1G robotic arm mimicking the human arm, and is developed by Robai [14] for the purposes of performing lightweight tasks, prototyping, and research. The Cyton Alpha features 7-DOF movement capability plus a single end effector gripper. The "kinematic redundancy" of Cyton Alpha combined with the high performance servo motors operating the joints contributes to the arm's fluid and accurate motion, as well as its ability to navigate around obstacles and narrow spaces. The Cyton Alpha can be controlled by any machine using a USB interface.

Figure 4.3: The Cyton Alpha Robotic Arm [14].

4.2.1 Actin-SE Control Software

All robotic arms developed by Robai come with Actin-SE, a powerful cross-platform configurable software package [14]. Actin-SE provides a rich set of APIs and tools for programmatic control of the arm, as well as an optimized control system to reduce development time. The provided optimized control system takes into account the great difficulty and the countless ways of manipulating the joints to achieve a desired motion or posture. Acting upon the provided requirements, the control system will handle simultaneous movements of the joints while avoiding collisions, singularities, joint limits, as well as maintaining optimized strength, accuracy, and reduced kinetic energy. Moreover, Actin-SE provides the Cyton Viewer (shown in Figure 4.4), which is a front-end GUI and kinematic simulator featuring 3D visualization of the robotic arm, as well as various simulation and configuration tools. Actin-SE is implemented in `C++` and supports Windows, Linux, and Apple platforms. The other functionalities and features provided by Actin-SE are listed below.

- **End-effector control**: enables direct control of the arm with any 3D input device, or computer mouse.
- **Kinematic and dynamic simulator**: useful for the purposes of simulating operation routines and environments.
- **Network interface**: enables remote control of the arm over TCP/IP.

- **Movie recording/playback**: enables recording and replaying of the arm's movements and simulations. These can be produced into video clips.
- **Data capture and dynamic plotting**: all kinematic, dynamic, and diagnostic data can be captured and stored in other tools' formats such as Matlab and Mathematica. Moreover, the captured data can be plotted dynamically in real time.

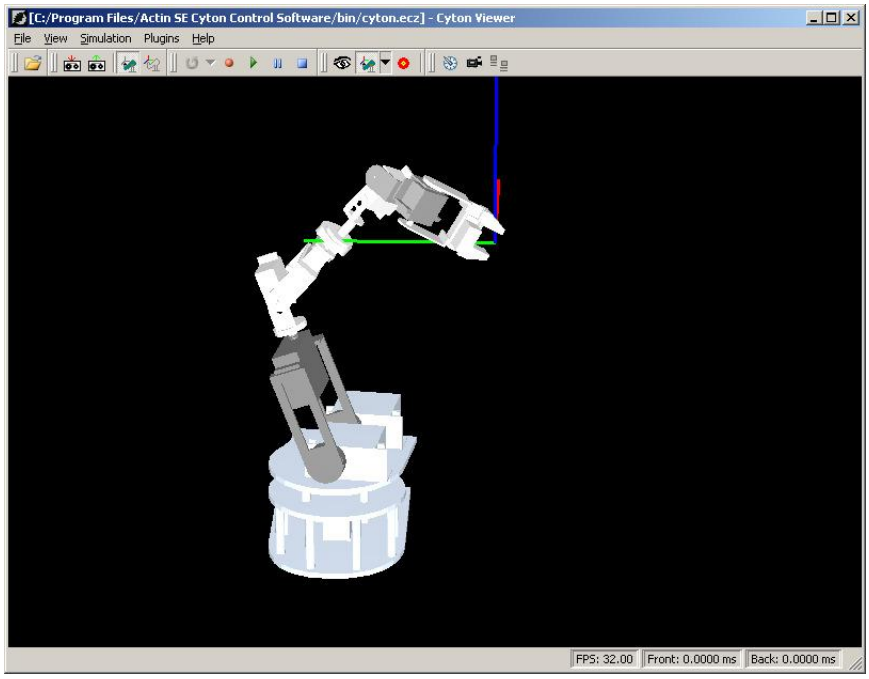


Figure 4.4: Cyton Viewer [14].

As with PeopleBot, two identical Cyton Alpha arms were available. The implemented services are discussed in the following subsection.

4.2.2 Implemented Services

A total of three services were developed for the Cyton Alpha arm. They are discussed in the following subsections.

Reach Demo

This service is a simple demo demonstrating the reach limit of each of the joints of the arm. The service simply controls the servo motor of a joint to the minimum and maximum values of its angular range, and it successively repeats the process for each joint.

Control

This service enables control of individual joints as well as combined control. This can be considered as a tight manual control of the arm. The inputs to this service are the angular values for one or more joints.

Grab and Place

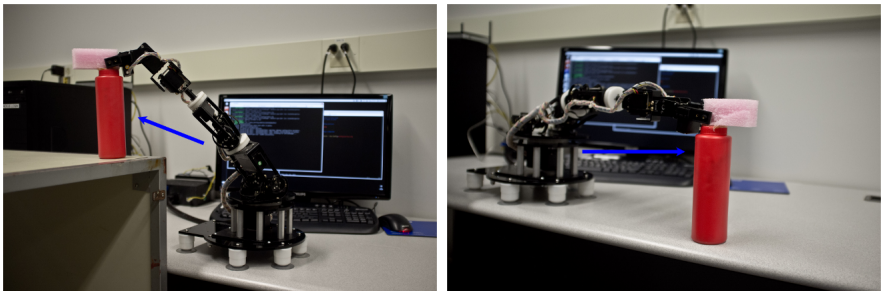
This sophisticated service controls the arm to pick an object from an elevated surface and place it accurately on a lower surface or a table edge. This service is particularly interesting since it is used to demonstrate collaboration with the PeopleBot mobile robot.

4.3 Collaborative Service

To test the middleware rigorously, a collaborative service was created involving a PeopleBot robot and a Cyton Alpha arm. In this scenario, the two robots work together to transport an object placed on an elevated surface at one side of a room to a table located at the opposite side. Figure 4.5 illustrates the initial setup in the CPAMI laboratory.

Figure 4.5: PeopleBot and Cyton Alpha at Initial Setup.

All a human operator needs to do is designate a master entity and issue a task request for the service to it. The master entity will coordinate the other involved entities and manage the order of execution of the different tasks. It makes no difference which entity is designated as the master. Firstly, the Cyton Alpha will execute the "Grab and Place" service (illustrated in Figure 4.6). It will then inform the PeopleBot of the success/failure of the task, or commands the PeopleBot to execute the next task, depending on whether the arm is the slave or the master respectively.



(a) Picking from an Elevated Surface. (b) Placing on the Table Edge.

Figure 4.6: Cyton Alpha in Action.

Next, the PeopleBot will execute the "Table Object Tracking and Grabbing" service. Then, if successful, it will either proceed to execute the "Navigation" service or waits for the command to execute "Navigation", depending on whether PeopleBot is the master or the slave respectively. Lastly, and after successfully reaching its destination, PeopleBot will operate its gripper to accurately place the object on the table. Figures 4.7 and 4.8 illustrate the PeopleBot in action.

(a) (b)

Figure 4.7: PeopleBot performing "Table Object Tracking and Grabbing".

(a) (b) (c)

Figure 4.8: PeopleBot Performing "Navigation". The Target is the Front of the Opposite Table.

4.4 Chapter Summary

This chapter discussed two different robotic platforms that were used to test the proposed middleware. The discussion covered the hardware features and the provided software libraries for each robot. In addition, the discussion highlighted the different services implemented for each robot. These services vary in complexity and make use of all the physical capabilities of the robot. Moreover, it was illustrated how heterogeneous robots could perform collaborative tasks using the middleware by having the PeopleBot mobile robot and the Cyton Alpha arm work together to transport an object across the laboratory.

Chapter 5

System Evaluation

This chapter discusses preliminary evaluation of the proposed middleware. Since the main aim is to design and develop an industrial grade distributed robotic middleware that could accommodate multiple entities and clients, the middleware was subjected to a series of performance and stress tests. The performance tests are tailored to match the needs of a robotic research environment, and so, they differ slightly from typical distributed systems and servers evaluation criteria.

The current version of the middleware is intended to be used within a typical university robotics laboratory. A series of performance tests were carried out at the Centre for Pattern Analysis and Machine Intelligence (CPAMI) at the University of Waterloo. The server component was deployed on a 2.66 GHz. Intel Quad Core machine with 4 GB of RAM. The tests relied on the provided high speed LAN and WLAN in the laboratory. Moreover, a barebones console-based client program was developed to carry out the experiments.

5.1 Latency

Latency, in the networking and communications context, is the transmission delay between a source and a destination in a network. In a typical packet switching network, latency can be measured in two different ways. The first is the one-way latency, which is the time it takes a request to travel from a source to a destination over the network. The second is the two-way latency, also known as the round-trip time (RTT), is the one-way latency from the source to the destination plus the one-way latency of the reverse trip (from the destination to the source). Pinging is the most common way of measuring latency and it is relatively accurate since it does not involve any processing on the destination side.

Since the middleware is designed to handle multiple clients and entities, it is critical to measure how long it takes a client to reach its checked-out entities, and how this time changes with an increasing number of connected clients and entities. Thus, several experiments were carried out to measure the latency between clients and entities. Unlike conventional latency measurements which merely involve pinging the destination, two different one-way latency measurements were carried out between clients and entities as follows:

- Latency of transmitting batches of raw data: as mentioned before, a client is able to send resources to an entity. A resource can be an image file, for instance. To simulate data, the GPB `bytestream` message was used to pack 5000 5-ASCII character strings. This size was chosen to simulate substantial network traffic.
- Latency of transmitting a stop signal: a client can send a stop signal to an entity which will halt the execution of the current task and/or physically stop the entity. For obvious safety reasons, such as hazards and collision avoidance, it is crucial that the stop signal reaches an entity in the least amount of time.

To simulate large numbers of entities, multiple EMs were evenly distributed on four different machines running Ubuntu Linux. Taking into account that many robots, and especially mobile robots, rely on wireless-based communication, the tests were duplicated to record latencies for both wired and wireless entities. Due to the inherent unreliability of networks, and to ensure that the recorded latency values are accurate, all received data on each entity is validated to ensure that it is uncorrupted and unaltered. The server was modified to allow for a client to stream data/signals to all connected entities, which is not the case in normal operation. This was done to allow for load testing. Furthermore, connection timeouts were set to two seconds (for both entities and clients).

5.2 Single Client - Multiple Entities

In this experiment, several runs were carried out to measure how the transmission latency between a single client and multiple entities is affected by the number of connected entities. The experiment involves exponentially increasing number of entities from 1 up to 2048 while recording the latency for each number of entities. For each run, an equal number of wired and wireless entities existed in the environment.

5.2.1 Raw Data Stream

In this experiment the client is continuously sending a batch of raw data to each connected entity every 500 milliseconds through successive RPCs of this function:

```
long sendData (long clientTimestamp, string entityId, string entityType,
string msg, CommUtil::DataSeq dataSeq);
```

The `dataSeq` stores the serialized `bytestream`. The total size of transmitted data (including function arguments) per call is 35137 bytes, or 281096 bits. Table 5.2.1 summarizes the obtained results. Figure 5.1 illustrates a plot of the average latency against the number of entities.

	Average Latency (ms)	
Number of Entities	LAN	WLAN
1	1.8	141.5
2	1.7	57.7
4	2.7	57.7
8	4.4	57.5
16	7.9	57.1
32	16.1	56.4
64	30.9	54.2
128	61.1	58.4
256	157.6	108.4
512	249.7	246.7
1024	402.6	698.9
2048	823.7	901.8

Table 5.1: Single Client - Multiple Entities Data Stream Average Latency.

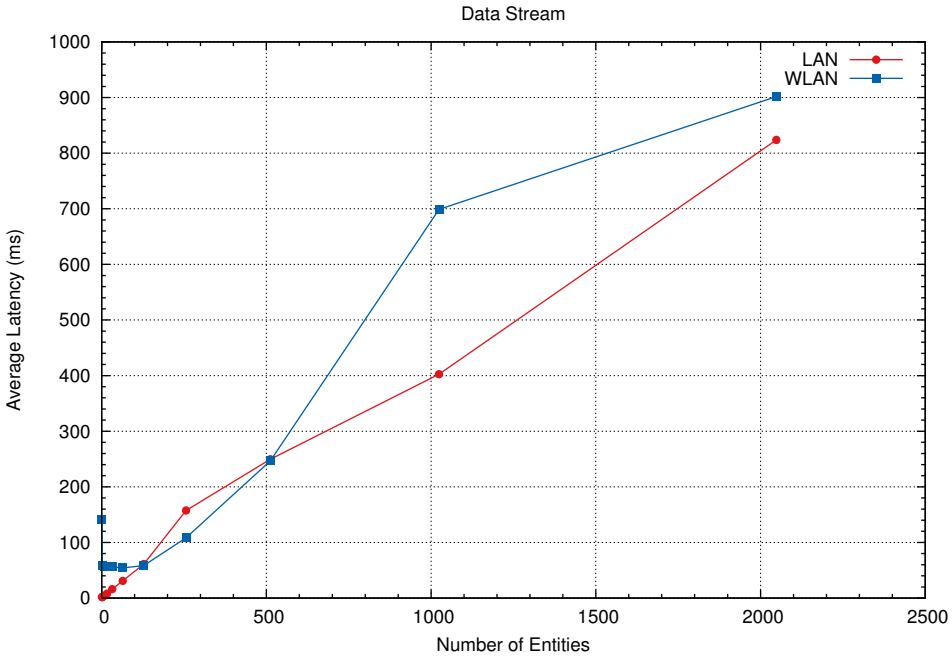


Figure 5.1: Single Client - Multiple Entities Data Stream Average Latency Plot.

5.2.2 Stop Signal

In this experiment the client is continuously sending a stop signal to each connected entity every 500 milliseconds through successive RPCs of this function:

```
long sendStopSignal (long clientTimestamp, string entityId,
  string entityType, string msg);
```

Table 5.2.2 summarizes the obtained results. Figure 5.2 illustrates a plot of the average latency against the number of entities. The results clearly demonstrates that it takes less than 1 second for a stop signal to reach any entity in an environment with a large number of entities.

	Average Latency (ms)	
Number of Entities	LAN	WLAN
1	0.2	0.3
2	0.3	0.3
4	0.6	0.5
8	0.9	2.3
16	1.7	1.6
32	2.6	2.4
64	4.8	5.1
128	10.6	8.9
256	18.2	17.8
512	35.3	39.5
1024	72.3	83.6
2048	160.7	402.8

Table 5.2: Single Client - Multiple Entities Stop Signal Average Latency.

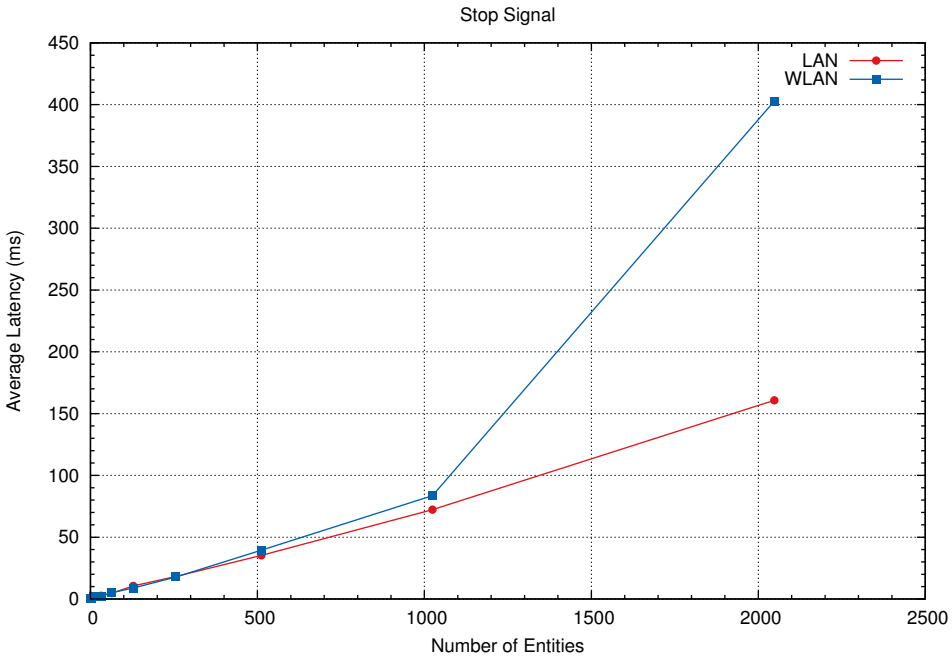


Figure 5.2: Single Client - Multiple Entities Stop Signal Average Latency Plot.

5.3 Multiple Clients - Multiple Entities

A number of load testing experiments were carried out to measure how the transmission latency between a client and its checked-out entities is affected by the number of both clients and entities in the environment. As with the previous test, an equal number of wired and wireless entities existed in the environment for each run. In this experiment, however, a fixed number of 256 entities were connected to the environment prior to testing; they are split into two categories as follows:

- Standard entities: this group consisted of 246 entities. These entities are ordinary connected entities that receive data and signals from clients.
- Testing entities: this group consisted of the remaining 10 entities (5 wired and 5 wireless entities). This group is used to collect latency data.

Theoretically, the number of clients cannot exceed the number of entities. Therefore, the number of clients was increased exponentially from 1 up to 256, thus having up to 65,536 simultaneous transmissions. Similar to categorizing the entities, each run of the experiment had one testing client and a number of standard clients. The standard clients continuously transmit a random mix of data and signals to the standard entities, while the testing client is transmitting to the testing entities.

5.3.1 Raw Data Stream

In this experiment, the testing client is sending a data batch to each testing entity every 500 milliseconds. It was observed that as the number of clients increased from 128, some of the connected entities and clients started to timeout. At 256 clients, several entities lost connection to the server. Table 5.3.1 summarizes the obtained results. Figure 5.3 illustrates a plot of the average latency against the number of clients.

	Average Latency (ms)	
Number of Clients	LAN	WLAN
1	14.0	25.8
2	43.0	106.3
4	70.9	78.8
8	89.6	54.4
16	344.8	445.0
32	671.5	1374.2
64	643.6	665.1
128	860.8	711.6
256	1365.2	2215.8

Table 5.3: Multiple Clients - Multiple Entities Data Stream Average Latency.

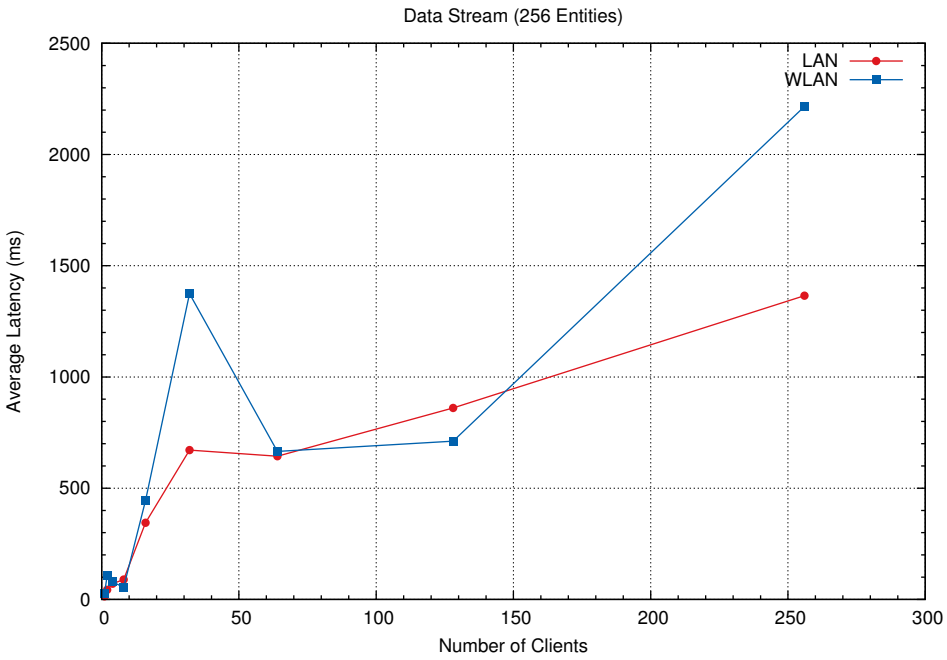


Figure 5.3: Multiple Clients - Multiple Entities Data Stream Average Latency Plot.

5.3.2 Stop Signal

In this experiment, the testing client sends a stop signal to each entity in the testing group every 500 milliseconds. Table 5.3.2 summarizes the obtained results. Figure 5.4 illustrates a plot of the average latency against the number of clients. Similar to the previous experiment, occasional disconnections were observed. The results demonstrate that even with a large number of simultaneous transmissions, a stop signal takes much less than 1 second to reach an entity.

	Average Latency (ms)	
Number of Clients	LAN	WLAN
1	0.3	0.9
2	0.3	0.6
4	0.3	0.8
8	0.3	1.0
16	2.0	3.2
32	3.5	5.7
64	3.9	4.8
128	10.6	10.7
256	8.3	9.5

Table 5.4: Multiple Clients - Multiple Entities Stop Signal Average Latency.

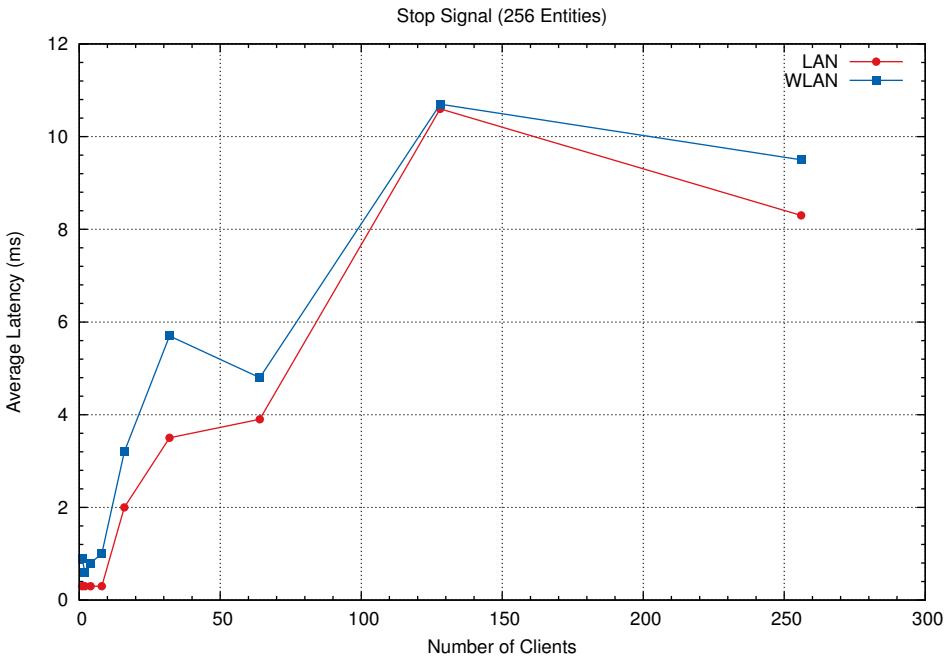


Figure 5.4: Multiple Clients - Multiple Entities Stop Signal Average Latency Plot.

5.4 Chapter Summary

This chapter presented a preliminary evaluation of the middleware. A series of experiments were performed to measure the latency between clients and entities under different conditions and setups. At the same time, these experiments simulated an overloaded environment to test the robustness and scalability of the server. The results clearly demonstrated that the average transmission latency between clients and entities is relatively small. The server proved to be quite robust and scalable, able to handle a large number of clients and entities.

Chapter 6

Conclusion and Future Work

Robotic middleware is becoming an important research topic in the robotics field. The great advances in the hardware, software and network technologies resulted in a broad diversity in robotic hardware and software. The rapidly growing interest in multi-robot research and development has brought a great need for tools that could enable seamless integration of multiple robots.

The goal of any robotic middleware is to provide the "glue" to attach multiple robotic applications in an integrated framework. Robotic middleware provides an abstraction layer that hides the inherent heterogeneity of hardware and software components of the underlying robotic entities. This greatly reduces complexity, simplifies software design, reduces development time, and promotes reusability and extensibility of applications. Using a robotic middleware, the developer needs only to develop his own custom algorithm or application as a component that can be plugged in to the environment along with other components.

This work tackles the problem of robotic middleware from a different angle. For a robot, instead of attempting to provide a comprehensive abstraction layer for its software and hardware components, the middleware provides a simple software abstraction layer atop its vendor provided software libraries. The rationale behind this design is that the vendor provided software is the best tool to make full use of the robot's capabilities. A developer would use the provided robotic software libraries in developing his application, and use the middleware to seamlessly plug his application in to a multi-robot environment. The middleware is composed of two main components: the EM and the server. The EM runs on the robot's onboard machine. It hides all networking and concurrency details from developers, and presents a simple and extensible interface for robotic applications

development. The current version of the EM supports `C++` applications, but can easily be extended to support other programming languages. The server component keeps track of all connected entities, and provides efficient information and data routing between them. It is based on the ICE technology, a highly efficient and robust network middleware. The GPB technology was employed to have a structured lightweight information exchange in the middleware. The notion of lightweight efficient information and data exchange was greatly emphasized in this work, and introduced as a substantial issue in robotic middleware research.

The middleware was rigorously evaluated in terms of its capability to handle heterogenous robots, and its performance. The middleware was deployed in a real laboratory environment and was used to integrate two different types of robots in a seamless manner, and to handle collaboration between them. In terms of performance, several experiments were carried out to test and measure the middleware's ability to handle exchange of data between a large number of entities and clients. The results illustrated that the middleware prototype is fit enough to be used right away.

The modular and professional design of the system allows for easy expansion and upgrade, and given that the current version is only a prototype, this opens the door to numerous ideas for upgrading and enhancing the system. Much more work can be addressed in related future work as follows:

- Based on performance evaluations, the system has a decent performance when handling multiple entities and clients. However, this cannot replace the need for real-time support. Adding real-time capabilities in the middleware is an obvious upgrade, especially given that real-time constraints are a recurring requirement in robotics.
- The modular design of the EM and EI allows for easy upgrade and expansion. The simple API provided in the EI can be easily expanded.
- The modular design of the server component allows for deployment on multiple machines to increase scalability. The middleware can grow to support multiple robotics laboratories and coordinate among them.
- The primitive notion of "global services" and collaboration currently implemented opens the door to develop a plugin framework that can support complex experiments and scenarios where the developer can specify his experiment and plug it in for execution in the environment.

- Support for entities other than robots can be easily included. In robotics research, researchers can make use of complex algorithms, which can be computationally expensive. A developer can deploy his/her algorithms on dedicated machines and plug them in to the environment using the middleware, where the robots can make use of them.

Needless to say, much more designing, testing, and development are required to make the middleware design a mature technology that can be used by a wide variety of developers and end users.

Bibliography

[1] AWARE-project.net. http://www.aware-project.net/. Accessed: June 27, 2012.

[2] Free CORBA downloads. http://www.omg.org/technology/corba/corbadownloads.htm. Accessed: June 16, 2012.

[3] Ice Performance White Paper. http://www.zeroc.com/articles/IcePerformanceWhitePaper.pdf. Accessed: May 24, 2012.

[4] Intelligent Mobile Robotic Platforms for Service pobots, Research and Rapid Prototyping. http://www.mobilerobots.com/Mobile_Robots.aspx. Accessed: June 18, 2012.

[5] Microsoft Robotics Developer Studio. http://www.microsoft.com/robotics/. Accessed: August 6, 2012.

[6] Mobilerobots Research Development Software. http://www.mobilerobots.com/Software.aspx. Accessed: June 18, 2012.

[7] Object Management Group (OMG). http://www.omg.org/. Accessed: June 2, 2012.

[8] ObjectWeb - what's Middleware. http://middleware.objectweb.org/. Accessed: June 21, 2012.

[9] OpenRTM-aist. http://www.openrtm.org/. Accessed: June 20, 2012.

[10] Oracle Berkeley DB. http://www.oracle.com/technetwork/products/berkeleydb/overview/index.html?origref=http://www.zeroc.com/freeze/index.html. Accessed: May 25, 2012.

[11] Peoplebot Robot Makes Human-Robot Interaction Research Affordable. http://www.mobilerobots.com/ResearchRobots/PeopleBot.aspx. Accessed: June 18, 2012.

[12] Protocol Buffers – Google Developers. https://developers.google.com/protocol-buffers/. Accessed: May 21, 2012.

[13] Rise and fall of CORBA. http://www.zeroc.com/documents/riseAndFallOfCorba.pdf. Accessed: June 16, 2012.

[14] Robai - Powerful Affordable Robots. http://www.robai.com/. Accessed: June 19, 2012.

[15] SOAP Specifications. http://www.w3.org/TR/soap/. Accessed: June 16, 2012.

[16] ZeroC - Our Customers. http://www.zeroc.com/customers.html. Accessed: May 24, 2012.

[17] ZeroC - the Internet Communications Engine. http://zeroc.com/ice.html. Accessed: May 24, 2012.

[18] N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku, and W. Yoon. RT-Middleware: Distributed Component Middleware for RT (Robot Technology). In *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3933–3938, 2005.

[19] D. E. Bakken, P. Dasgupta, and J. Urban. Middleware. *Encyclopedia of Distributed Computing*, 2001.

[20] H. Bruyninckx. Open robot control software: the OROCOS project. In *2001 IEEE International Conference on Robotics and Automation (ICRA)*, volume 3, pages 2523–2528, 2001.

[21] C. Côté, Y. Brosseau, D. Létourneau, C. Raevsky, and F. Michaud. Robotic Software Integration Using MARIE. *International Journal of Advanced Robotic Systems*, 3(4):55–60, March 2006.

[22] G. G. de Rivera, R. Ribalda, J. Cols, and J. Garrido. A generic software platform for controlling collaborative robotic system using XML-RPC. In *2005 International Conference on Advanced Intelligent Mechatronics (IEEE/ASME)*, pages 1336–1341, 2005.

[23] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, USA, 1994.

[24] B. P. Gerkey, R. T. Vaughan, and A. Howard. The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems. In *Proceedings of the International Conference on Advanced Robotics (ICAR)*, pages 317–323, 2003.

[25] P. Gil, I. Maza, A. Ollero, and P. J. Marrón. Data centric middleware for the integration of wireless sensor networks and mobile robots. In *Proceedings of 7th Conference on Mobile Robots and Competitions, ROBOTICA*, pages 1–6, April 2007.

[26] F. Guorui and W. Jian. Research of heterogeneous robots system based on CORBA. In *2011 International Conference on Consumer Electronics, Communications and Networks (CECNet)*, pages 569–573, 2011.

[27] T. H. Harrison, D. L. Levine, and D. C. Schmidt. The design and performance of a real-time CORBA event service. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, volume 32, October 1997.

[28] D. Hernández-Sosa, A. C. Domínguez-Brito, C. Guerra-Artal, and J. Cabrera-Gámez. Runtime Self-Adaptation in a Component-Based Robotic Framework. In *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2700–2705, 2005.

[29] W. Hongxing, L. Shiyi, Z. Ying, Y. Liang, and W. Tianmiao. A Middleware Based Control Architecture for Modular Robot Systems. In *2008 IEEE/ASME International Conference on Mechtronic and Embedded Systems and Applications*, pages 327–332, 2008.

[30] M. Y. Jung, A. Deguet, and P. Kazanzides. A Component-based Architecture for Flexible Integration of Robotic Systems. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, October 2010.

[31] S. Knoop, S. Vacek, R. Zollner, C. Au, and R. Dillmann. A CORBA-based distributed software architecture for control of service robots. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 4, pages 3656–3661, 2004.

[32] T. Lee, H. Seo, B. Lee, and D. Shin. A Software Component Model and Middleware Architecture for Intelligent Mobile Robot. In *2nd International Conference on Computer and Automation Engineering (ICCAE)*, volume 4, pages 453–456, 2010.

[33] S. Magnenat, V. Longchamp, and F. Mondada. ASEBA, an event-based middleware for distributed robot control. In *Workshops DVD of International Conference on Intelligent Robots and Systems (IROS)*, pages 1–6, October 2007.

[34] M. R. Majedi, K. A. Osman, and M. Boyd. A Generic Service Oriented Architectural Model for Pervasive Applications: A Case Study in Internet-based Multiple Robot Control. In *Third International Conference on Pervasive Computing and Applications*, volume 1, pages 54–59, 2008.

[35] A. Makarenko, A. Brooks, and T. Kaupp. Orca: Components for Robotics. In *International Conference on Intelligent Robots and Systems (IROS)*, 2006.

[36] M. Mizukawa, H. Matsuka T. Koyama T. Inukai, A. Nodad, H. Tezuka, Y. Noguchi, and N. Otera. ORiN: open robot interface for the network - the standard and unified network interface for industrial robot applications. In *41st SICE Annual Conference*, volume 2, pages 925–928, 2002.

[37] N. Mohamed, J. Al-Jaroodi, and I. Jawhar. A Review of Middleware for Networked Robots. *International Journal of Computer Science and Network Security (IJCSNS)*, 9(5):139–148, 2009.

[38] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng. ROS: an open-source Robot Operating System. In *ICRA 2009 Workshop on Open Source Software in Robotics*, pages 1–6, 2009.

[39] F. Santos, L. Almeida, P. Pedreiras, and L. S. Lopes. A real-time distributed software infrastructure for cooperating mobile autonomous robots. In *International Conference on Advanced Robotics (ICAR)*, 2009.

[40] D. C. Schmidt. The ADAPTIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Communication Software. pages 214–225, 1993.

[41] W. D. Smart. Is a Common Middleware for Robotics Possible? In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'07)*, 2007.

[42] B. Song, S. Jung, C. Jang, and S. Kim. An Introduction to Robot Component Model for OPRoS(Open Platform for Robotic Services). In *Proceedings of the International Conference Simulation, Modeling Programming for Autonomous Robots Workshop*, pages 592–603, 2008.

[43] I. Song, F. Guedea, and F. Karray. CONCORD: A Control Framework for Distributed Real-time Systems. *IEEE Sensors Journal*, 7(7):1078–1090, 2007.

[44] I. Song, F. Karray, and F. Guedea. A Distributed Real-time System Framework Design for Multi-Robot Cooperative Systems using Real-Time CORBA. In *2003 IEEE International Symposium on Intelligent Control*, pages 793–798, 2003.

[45] A. S. Tanenbaum and M. V. Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, USA, 2006.

[46] H. Utz, S. Sablatnög, S. Enderle, and G. Kraetzschmar. Miro - Middleware for Mobile Robot Applications. *IEEE Transaction on Robotics and Automation*, 18(4):493–497, November 2002.

[47] A. Whitbrook. *Programming Mobile Robots with Aria and Player: A Guide to C++ Object-Oriented Control*. Springer, London, 2010.

[48] J. Yoo, S. Kim, and S. Hong. The Robot Software Communications Architecture (RSCA): QoS-Aware Middleware for Networked Service Robots. In *International Joint Conference, SICE-ICASE*, pages 330–335, 2006.

Appendix A

Installing and Running the Server

This appendix describes the procedure that the developer needs to follow in order to install the server component on a machine. The current version of the server supports x86 GNU/Linux platforms.

A.1 Prerequisites

The only prerequisite for installation is the ICE software package v.3.4.2 or later. This package can be downloaded from this link: http://zeroc.com/ice.html, or from the package manager of the Linux distribution on the target machine.

A.2 Installation

The server component is provided in the `axon_backend.tar` archive file. A developer must perform the following steps in order to properly compile and install the server:

1. Open a terminal.
2. Untar the `axon_backend.tar` archive.
3. Go into the "`environment/`" directory by typing: "`cd axon_backend/environment/`".

4. The default installation directory is "`/opt/axon_backend/environment/`". This can be changed as follows:
   (a) Go to the "`config`" directory.
   (b) Open the `Make.rules` file using a text editor.
   (c) Edit the *prefix* parameter by typing the desired installation directory.
   (d) Save and close.
5. Inside the "`environment/`" directory, type "`make`", followed by "`make install`". This will compile and install the server into the target installation directory.

A.3 Configurations and Running

Before running, the server must be assigned to port numbers and the host machine IP address in order for external entities and clients to be able to connect to it. Two configuration files in the "`config/`" sub-directory of the installation directory must be edited as follows:

- `entityserver.cfg`, for the entity server. In this file, the parameter *EntityServer.Endpoints* takes a port number and two IP address values for SSL and TCP types of connections. The IP addresses must be the same for both types. On the other hand, port numbers must be distinct.
- `frontendserver.cfg`, for the front-end server. In this file, the parameter *FrontEndServer.Endpoints* must be edited in the same way as the *EntityServer.Endpoints* of `entityserver.cfg`.

In order to run the server, go to directory to the "`bin/`" directory and execute the `run_server.sh` script by typing in the terminal: "`sh run_server.sh`". If initialization is success, the server will display success messages. It is now ready to receive connections.

A.4 Global Services

As mentioned earlier, a simple profile called the "global services" may be created to define simple collaborative services between entities. This profile has to be kept on the server machine in order to make it available to all clients.

This profile must be included in the "`globalservices/`" directory inside the server's installation directory. This will ensure that the profile will be parsed by the server and made available to all connected clients, on request.

Figure A.1 illustrates a sample global services profile.

```
<?xml version="1.0" encoding="UTF-8"?>

<GlobalProfile count = "3">
  <Service name = "Collaboration Demo">
    <entity category = "ROBOT" type = "MOBILE" id = "0" service_global_id = "cruiser_zone" preced = "1">
        <key_value param_name = "Destination" param_value = "6" />
    </entity>

    <entity category = "ROBOT" type = "STATIONARY" id = "1" service_global_id = "cyton_pick" preced = "1" />

    <entity category = "ROBOT" type = "MOBILE" id = "0" service_global_id = "peoplebot_blobpick" preced = "2" />

    <entity category = "ROBOT" type = "MOBILE" id = "0" service_global_id = "cruiser_dest" preced = "3">
        <key_value param_name = "Destination" param_value = "1" />
    </entity>

    <entity category = "ROBOT" type = "MOBILE" id = "0" service_global_id = "gripper" preced = "4">
      <key_value param_name = "Open/Close" param_value = "true" />
      <key_value param_name = "Direction" param_value = "true" />
    </entity>

  </Service>

  <Service name = "Pick and Place Demo">
    <entity category = "ROBOT" type = "MOBILE" id = "0" service_global_id = "cruiser_zone" preced = "1">
      <key_value param_name = "Destination" param_value = "6" />
    </entity>
    <entity category = "ROBOT" type = "MOBILE" id = "1" service_global_id = "peoplebot_blobpick" preced = "2">
      <key_value param_name = "Color" param_value = "1" />
    </entity>
    <entity category = "ROBOT" type = "MOBILE" id = "0" service_global_id = "cruiser_dest" preced = "3" />
    <entity category = "ROBOT" type = "MOBILE" id = "0" service_global_id = "peoplebot_grip" preced = "4">

      <key_value param_name = "Open/Close" param_value = "true" />
    </entity>
  </Service>

  <!-- ............... -->

</GlobalProfile>
```

Figure A.1: "Global services" Profile.

Appendix B

Adding an Entity to the Environment

This appendix describes the procedure that the developer needs to follow in order to install the entity component of the middleware and to link a robotic application to it.

B.1 Prerequisites

There are two software packages that need to be installed prior to installing and using the entity component. They are listed as follows:

- The ICE software package v.3.4.2 or later. This package can be downloaded from this link: `http://zeroc.com/ice.html`, or from the package manager of the Linux distribution on the target machine.
- The GPB package v.2.4.1 or later. This package can be downloaded from this link: `https://developers.google.com/protocol-buffers/`, or from the package manager of the Linux distribution on the target machine.

B.2 Installation

The entity component is provided in the `axon_backend.tar` archive file. A developer must perform the following steps in order to properly compile and install the entity component.

B.2.1 Entity Manager

1. Open a terminal.
2. Untar the `axon_backend.tar` archive.
3. Go into the "`entity/`" directory by typing: "`cd axon_backend/entity/`".
4. The default installation directory is "`/opt/axon_backend/entity/`". This can be changed as follows:
   (a) Go to the "`config/`" directory.
   (b) Open the `Make.rules` file using a text editor.
   (c) Edit the *prefix* parameter by typing the desired installation directory.
   (d) Save and close.
5. Inside the "`entity/`" directory, type "`make`", followed by "`make install`". This will compile and install the EM into the target installation directory.

B.2.2 Entity Interfacer

1. Open a terminal.
2. Untar the `axon_backend.tar` archive.
3. Go into the "`entityinterfacer/`" directory by typing:
   "`cd axon_backend/entityinterfacer/`".
4. The default installation directory is "`/opt/axon_backend/entityinterfacer/`". This can be changed as follows:
   (a) Go to the "`config/`" directory.
   (b) Open the `Make.rules` file using a text editor.
   (c) Edit the *prefix* parameter by typing the desired installation directory.
   (d) Save and close.
5. Inside the "`entityinterfacer/`" directory, type "`make`", followed by "`make install`". This will compile and install the EI into the target installation directory.

B.3 Configurations and Running

This section provides details about configuring and running the entity component of the middleware on a robot.

B.3.1 Entity Interfacer

In order to use the the EI in an application, the application must make use of two items in the installation directory:

1. The "`entityinterfacer.h`" header file found in the "`include/`" sub-directory. This header file must be included in the application source code.
2. The "`libentityinterfacer.so`" shared library found in the "`lib/`" sub-directory. The application must link against this shared library.

The EI and the EM communicate through a TCP socket. The socket port number can be configured by editing the "`ports.cfg`" configuration files found in the "`config/`" sub-folder. A range of port numbers can be entered. This range must be between `5500` and `8000` inclusive.

B.3.2 Entity Manager

The EM must be configured before running. Two configuration files must be edited as follows:

- The "`entitygl2client.cfg`" Glacier2 client configurations file. The *Ice.Default.Router* parameter takes two IP address values. To ensure that the EM can locate the server, the two IP addresses must match the IP address of the server machine.
- The "`ports.cfg`" configuration file. The specified port range must match that in the "`ports.cfg`" of the EI. As with EI, the range must fall between `5500` and `8000` inclusive.