N° d'ordre:

École Doctorale Sciences Pour l'Ingénieur

ULP - INSA - ENGEES - URS

THÈSE

présentée pour obtenir le grade de

Docteur de l'Université Louis Pasteur - Strasbourg

Discipline: Sciences pour l'Ingénieur

(spécialité Informatique)

par

Florentin Picioroagă

SCALABLE AND EFFICIENT MIDDLEWARE FOR REAL-TIME EMBEDDED SYSTEMS. A UNIFORM OPEN SERVICE ORIENTED, MICROKERNEL BASED ARCHITECTURE

Soutenue publiquement le 3 décembre 2004

Membres du jury

Directeur de thèse:	M. Uwe Brinkschulte, professeur, Université de Karlsruhe
Co-Directeur de thèse:	M. Bernard Keith, professeur, INSA Strasbourg
Rapporteur interne:	M. Pierre Colin, professeur, ENSPS Strasbourg
Rapporteur externe:	M. Andreas Polze, professeur, HPI - Université de Potsdam
Rapporteur externe:	M. Gabriel Ciobanu, professeur, Académie Roumaine de Iași
Examinateur:	Mme. Aurélie Bechina, maître de conférence, Université d'Oslo

LIIA N° de l'Unité: ERT N°9

To my beloved wife - Carmen, my daughter - Ana Maria,
my parents and my grandmother.

ACKNOWLEDGMENTS

I would like to thank Prof. Uwe Brinkschulte, for the valuable discussions and clear directions which he showed me along the entire period at IPR Institute. He introduced me in the research domain of real-time systems and gave me the very basic conception, trying to use a microkernel architecture and services for embedded middleware, as a premise and basic idea for my thesis. Based on that, I have developed the detailed fine grain architecture, implemented and evaluated it. Without his help on the last four years, I wouldn't finish this work.

Many thanks to Mrs. Aurelie Bechina and Etienne Schneider for the interesting discussions and their support.

Many thanks to Prof. Bernard Keith who encouraged me and accelerated the process of writing the thesis.

I would like also to thank the reviewers: Mrs. Aurelie Bechina, Mr. Gabriel Ciobanu, Mr. Pierre Colin and Mr. Andreas Polze for their constructive comments.

ABSTRACT

The task to design and implement a middleware for embedded real-time systems has, near the normal challenges for building a usual middleware, two additional difficult demands: 1) it must maintain the real-time features of the underlying environment and, 2) it must adapt in the same time to powerful systems and, moreover, fit on small systems (embedded systems). This thesis presents the research done in providing a solution for these problems - OSA+ middleware. The proposed solution is using a well known concept from operating systems - the microkernel architecture. Adapting this concept to a middleware brings a serie of advantages and disadvatages which are analyzed in this thesis.

Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

1.1 Motivation

In the early days of computing, most solutions were equipped with monolithic software designs specific to the problem domain. APIs (Application Programming Interfaces) were not yet existent for many common functions and standard communication protocols were not used. The applications had to be adapted again and again to run on a different system. Entire systems (hardware/software) could only be designed and programmed by a single engineer unless a big software design project with a large overhead was started. The problem was that programming at a low abstraction level was and remains a cumbersome task. Many platform and programming system specific details have to be taken care of, e.g. storage management, exception handling, proper initialization of service routines and networking.

In our days, the goals have been shifted from delivering problem specific end-solutions, to delivering open-solutions. With other words, the sintagm *a solution is good when it solves my problem* has been transformed in *a solution is good when it solves my problem and can be used in other contexts as well.* So, the problem solution takes in account further possibilities of reusing the software in other environments where they can enlarge the functionality of existing applications. Hence, the integration and functionality of application programs has greatly increased over the past years. And what is interesting and more challenging is that, not only the desktop, server, or super computers are directly affected by this trend, but also systems with less computational resources: small microcontrollers with low clock frequency (a few MHz) and limited memory resources (64-512 KB). They are usually highly specialized for tasks like: controlling devices, actuators, gathering data from sensors and reacting to critical situations. In most of these systems, *the right answer delivered too late becomes the wrong answer*, that is why achieving of end-to-end quality of service(QoS) is essential [52]. Such systems are known as real-time and embedded systems.

These systems are increasingly being connected via wireless and wireline networks to create large-scale distributed real-time and embedded (DRE) systems. They have become critical in domains such as avionics (e.g., flight mission computers), telecommunications (e.g., wireless phone services), tele-medicine (e.g., robotic surgery), nuclear reactors, oil refineries and industrial process automation.

Distributed embedded and real-time systems, or more correctly the problems which they pose to the software designer, programmer and system maintainer, represent one of the main focuses of this thesis. These problems are shortly enumerated in the following paragraph and they will be further analysed in chapter 2. They are the driving forces for the research done in this thesis and for the decisions which were made in order to reach the envisioned goals.

DRE systems include many interdependent levels, such as communication (e.g., network, bus communication), many local and remote systems which must be coordinated, and often multiple layers of software. In such systems, there are situations when it is hard to enumerate, or often even approximate, all possible physical system configurations or workload mixes a priori [52]. Together all these derive the following challenges:

- As distributed systems, DRE systems require capabilities to manage connections and message exchange between (possibly heterogeneous) networked computing devices.
- As real-time systems, DRE systems require predictable and efficient control over end-to-end system resources, such as memory, CPU, and network bandwidth.
- As embedded systems, DRE systems have size, weight, cost, and power constraints that often limit their computing and memory re-

sources. For example, embedded systems often cannot use conventional virtual and automatic memory techniques because of space or timing constraints. In this case, the software must fit on low-capacity storage media, such as EEPROM or NVRAM.

- As open systems which operate in a dynamic environemnt, DRE systems require controllability, and adaptability of operating characteristics for applications with respect to such features as time, quantity of information, accuracy, and synchronization.

While it is possible *in theory* to develop these types of complex systems from scratch, contemporary economic and organizational constraints, as well as increasingly complex requirements and competitive pressures, make it infeasible to do so *in practice*. To address all the competing design forces and run-time QoS demands, sustained efforts are made for creating comprehensive software methodologies, design-/run-time environments, and hardware/software co-design.

The presented thesis contributes to these efforts and is foccusing on the software side of the problem.

1.2 Background

After seeing the main challenges for DRE systems, an overview about the solution which conciliates these challenges is given in this section. Later the solution will be further analyzed, so the reader can become familiar step by step with the research theme of the thesis.

Over the past decade, various technologies have been intended to alleviate many complexities associated with developing software for DRE systems. Their successes have added a new category of systems, programming languages, and networking offerings to the previous generation. In particular, some of the most successful of emerging technologies have centered on *middleware*, which is system software that resides between applications and the underlying operating systems, network protocol stacks, and hardware. The middleware provides general purpose services which have standard programming interfaces and use standard protocols. The management of handling presentation, computation, information storage, communication, control and system resources are all highly popular examples of middleware services [5].

Typically, middleware packages take care of the details and offer some functionality at a higher level of abstraction. The oldest pieces of middleware packages in the history of computing were possibly the scientific libraries written in Fortran. For example, LINPACK [18] raises the level of abstraction from floating point numbers and basic arithmetic operations to matrices and their operations.

Most middleware-oriented systems impose specific structure formats on the applications built on them. Their API software layers are responsible for inter-operability and are well defined [64]. For example, in high performance distributed computing, the data is distributed to a large number of computers so they can work in parallel and achieve speeds that are much higher than on a single computer. Message passing libraries such as PVM [59] and MPI [24] evolved into middleware packages offering many more services than simple send and receive calls for data communication.

This thesis focuses on middleware which eases the development of a distributed system by hiding the distribution from the user. As can be seen in figure 1.1, the user is not concerned on two important aspects regarding the distribution:

- the locality, the middleware hides the location of the user applications. The user makes no distinction between different physical platforms when using the middleware, but perceives only one virtual platform offered by the middleware.

- the platform specific implementation, the user application has to implement the same middleware APIs wherever the application will be deployed (e.g., Unix, VxWorks, Windows). The platform specific APIs are hidden by the middleware layer.

Another important aspect of middleware is the availability on different architectures and languages. Even the middleware provides a standard API and the possibility to work with high level abstraction notions, it will be a great disadvantage if it is ported only on few combinations of architectures and languages. A study made by the Standish's Group's CHAOS

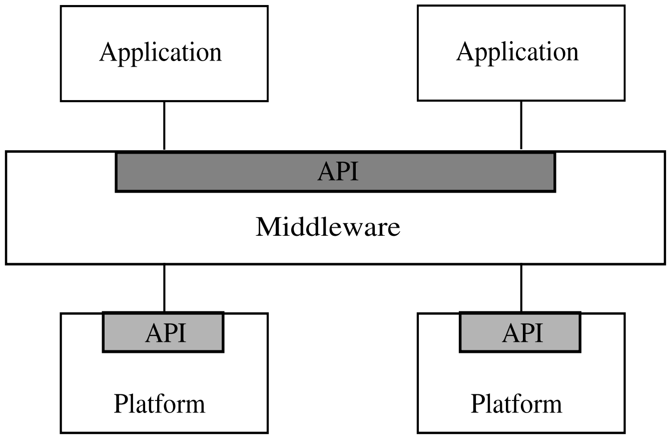


Figure 1.1: Middleware, hiding the distribution

shows the importance of choosing the right infrastructure in order to have a solid platform which also enables the application to easily adapt to new demands. They state that 70% of all software projects involve the creation and recreation of the infrastructure [1]. Considering also the special requirements regarding real-time aspects imposed by the application the potential user has less options for choosing a whole tool chain for developing the application.

Figure 1.2 shows the possibilities to build a real-time distributed application in an embedded system environment using the middleware approach. A distributed application is real-time if all the components have real-time capabilities. Using a top-down approach it can be observed in figure 1.2, that the middleware, which in fact realizes the distribution of the application, has to be real-time. Under the middleware layer the following system configurations can exist:

- *System 1*: in this case the chosen programming language can be transformed directly in execution code for the general purpose (GP) used processor(C, C++, Ada). Here the middleware runs directly on top of the operating system (OS) and is using the network communication provided by the OS. Both have to be real-time.

- *System 2*: the programming language is interpreted and is executed by a virtual machine. Java is the most used interpreted language

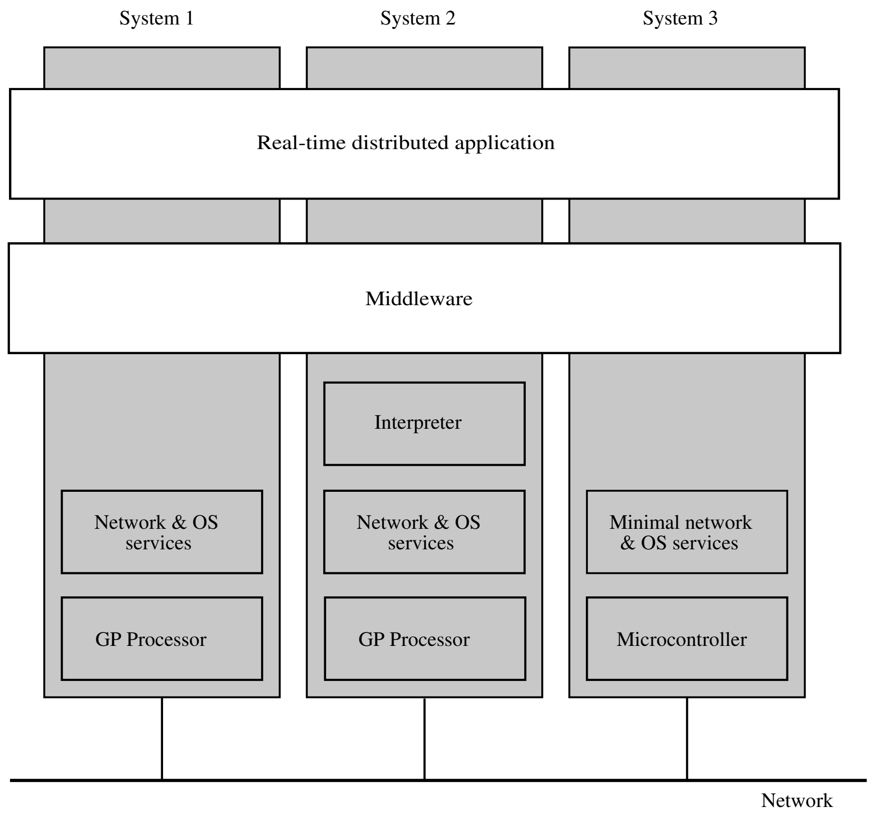


Figure 1.2: Main components for building a real-time distributed application for embedded systems

in the area of embedded systems. In this system configuration, in addition, the Java Virtual Machine has to be real-time.

- *System 3*: this system configuration is highly specialized for embedded real-time applications. OS and network services are minimal, special functionalities specific for real-time systems are provided by the microcontroller.

When chosing the infrastructure of the application(Fig. 1.2) the following criteria have to be considered:

- Platform availability
  - programming language for development (e.g., C, C++, Java, Ada)

  - compiler or virtual machine in case of choosing an interpreted language (e.g., Java)
  - operating system which for real-time applications must be a Real-time Operating System (RTOS)
  - processor

- Costs
  - for development, which includes licence prices for the used components: compiler, operating system, middleware and maybe training courses for the developers
  - deployment
  - maintenance

- Time to market. It can be short if the developers become rapidly familiar with the choosed components, and/or there is comprehensive documentation and support.

The previously enumerated criteria concern the infrastructure of the application which restricts already the user possibilities or gives the important paths to be followed. But there are also other important aspects to consider, which become critical in case of real-time embedded systems. These are:

- Performance
  - speed, efficiency of the chosen components
  - footprint, especially for the embedded systems

- Predictability, without predictability, performance is useless

- the need for loading and executing dynamic code at run-time. This is important when a system must be reconfigured, updated during run-time without stopping it (e.g., for air traffic control systems)

- Power consumption and techniques for power management. This is important for wireless interconnected systems.

Regarding the programming language for real-time distributed systems, for small embedded applications, sequential languages like C and C++ reign supreme. For the larger real-time high integrity systems, Ada still dominates. In the telecommunications market, CHILL is popular. In Germany, Pearl is widely used for process control and other industrial automation applications [22].

Although there is little doubt that the Java language has been immensely successful in a wide range of application areas, it has yet to establish itself completely in the real-time and embedded markets. Even that a preliminary version of the Real-Time Specification for Java exists, the introduction of the final approved specification could dramatically alter the status quo. In the future, if Microsoft's C# programming language starts to gain momentum, extensions will inevitably be considered to make it more appropriate for real-time systems.

1.3 Objectives

After having an overview of the research domain, distributed real-time and embedded systems, and the problems which appear designing and developing applications for such systems, it is very important to point out now our goals. The main focus of our middleware is to ease the development of DRE systems. Therefore, the special requirements for these systems must be satisfied:

1. Footprint: the middleware should be able to run on top of systems with low resources (embedded systems).

2. Scalability: The middleware should perform well on low resource systems, but also it should be able to scale up to more complex applications where a richer set of features are needed.

3. Real-time:

    - the middleware should first provide bounded worst case execution times (WCET) for all its components.

- quality of service (QoS) information must be supported in order to deal with different real-time aspects and to reflect different environments.
- end-to-end priorities should be preserved over different platforms to avoid priority inversion [1].

4. Overhead minimization: middleware introduces overhead in terms of memory and performance degradation which have to be minimized as much as possible.

Choosing a microkernel approach to solve at least part of these requirements seems to be a natural approach. This concept is well-known from operating systems and it looks suitable to be applied when designing a middleware which has to meet the first two mentioned requirements for DRE systems.

So this leads to one of the main research goals of this thesis: *to investigate the suitability of the microkernel concept when designing middleware for DRE systems.*

The middleware designed and presented within this thesis is implemented in two programming languages: C and Java. However, it will be discussed the Java reference implementation, as it is more updated, more complete regarding the implementation of our mentioned goals and better evaluated, especially in a real-time environment.

1.4 Thesis organization

In the previous sections the reader got familiar with the research area of the thesis and, more important, with the goals that are aimed. A roadmap of the thesis is presented here which can guide the reader further:

In **Chapter 2**, the research domain - middleware for distributed real-time and embedded systems - is thoroughly studied. This permits us to have a deeply view of all the issues that have to be tackled for reaching our goals and assures the foundation for the following chapters.

In **Chapter 3**, existing solutions provided by other people: research groups, companies, are discussed and compared to the aimed goals.

[1]priority inversion appears when higher priority tasks are blocked by lower priority tasks

Chapter 4 presents our approach. First, a description of the abstract concepts is made, which is continued with a deep view of all the important insides related to the goals.

In **Chapter 5**, an evaluation of our middleware approach is presented in order to verify if and in which extent the aimed goals have been reached.

Chapter 6 summarizes the research contribution presented in this thesis and suggests directions for future work.

A logical roadmap of the thesis is presented in figure 1.3.

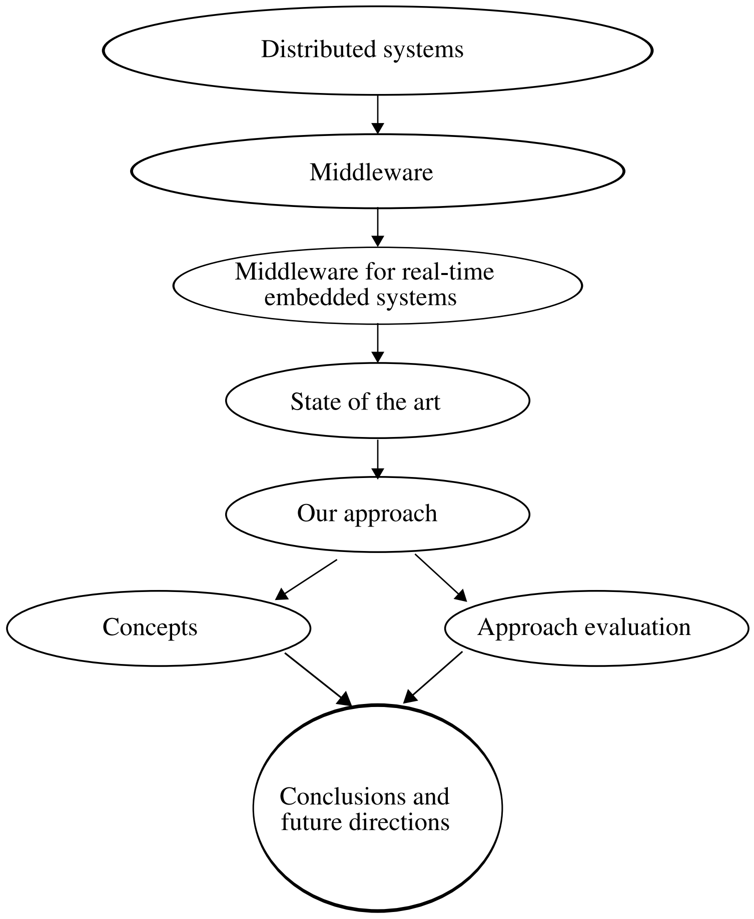


Figure 1.3: Thesis logical roadmap

Chapter 2

MIDDLEWARE FOR REAL-TIME EMBEDDED SYSTEMS

2.1 Distributed systems

2.1.1 Definition of distributed systems

An important step in computer systems history was the appearance of *Local-area networks* or *LANs* which allow hundreds of computers to be connected within a building. Later the appearance of *Wide-area networks* or *WANs* allowed more LANs to be interconnected. As a consequence of these technologies, it became very easy to put together large numbers of computers in so called *computer networks.* They enabled at the beginning exchange of data between computers located in different geographical places in a faster, and easier way than previous communication technologies. This evolved in more complex interactions between computers than basic data communication and finally resulted in the appearance of applications, logically spread on different computers and with them the appearance of the *distributed systems.*

There are different definitions of a distributed system in the literature. One of the first definitions was given already in 1978 by Enslow [21]. More recent definitions are given by Bal [4], Tanenbaum [60], Schroeder [54] and Couloris [16]. We will give here only the later one:

A distributed system is that composed of a collection of autonomous computers connected by a communication network, and equipped with software enabling them to coordinate their activities and share resources.

As an example of a distributed system, we consider a network of workstations in a university or company department. In addition to each user's personal workstation, there might be a pool of processors in the machine room that are not assigned to specific users but are allocated dynamically as needed. Such a system might have a single file system, with all files accessible from all machines in the same way and using the same path name. Furthermore, when a user types a command, the system could look for the best place to execute that command, possibly on the user's own workstation, possibly on an idle workstation belonging to someone else, and possibly on one of the unassigned processors in the machine room. If the system as a whole looks and acts like a classical single-processor timesharing system (i.e., multi-user), it qualifies as a distributed system [60].

2.1.2 Goals of distributed systems

Building a distributed system should solve some problems that arise and should serve some goals in order to worth the effort.

The main important goals for building a distributed system are: connect easily users or applications to resources, hide the fact that resources are distributed across a network, increase performace (e.g., adding new resources in the system), be open, be scalable. These aspects will be discussed further.

Connecting users to resources

A main goal of a distributed system is to make it easy for users to access remote resources, and to share them with other users in a controlled way. Some examples of resources are: printers, computers, storage facilities, data, files, networks, etc. One of the problems which appear here is that sharing resources can lead to unwanted accesses to the resources, security mechanisms have to be considered. Furthermore, synchronization becomes an issue. The resources have to be accessed in a consistent way, it can happen that they are accessed by more users or tasks at the same time.

Transparency

Transparency is an aspect of the distributed system that is hidden from the user (programmer, system developer, user or application program). Transparency is provided by including a set of mechanisms in the distributed system at a layer below the interface where the transparency is required. A distributed system that is able to present itself to users and applications as if it were only a single computer system is said to be *transparent*.

The International Standards Organization study on Open Distributed Processing [46] identified eight dimensions of transparency. These areas are interrelated and build upon each other. They describe properties of distributed systems of varying complexity and importance. We can briefly state the characteristics of each type of transparency [19]:

- *Access.* Software can interact in the same manner (i.e. exchange information), regardless of where it is (local or remote) or how it is implemented.
- *Location.* Software components can find other components regardless of location.
- *Migration.* A component can be moved from one host to another without affecting interaction.
- *Replication.* Multiple copies of the destination component may exist on different networked machines.
- *Concurrency.* Users need not be aware of ongoing interactions between other software and the destination component.
- *Scalability.* Systems can grow substantially (more connections, more components), yet maintain the same basic interaction mechanism and architecture.
- *Performance.* Mechanisms by which performance is obtained is hidden from users. This may include dynamic load balancing.
- *Failure.* Destination components may fail without affecting the consistency of the entire system.

Openness

An Open Distributed System is made up of components that may be obtained from a number of different sources, which together work as a single distributed system. It offers services according to standard rules that describes the syntax and semantics of those services. For example, in computer networks, standard rules govern the format, contents, and meaning of messages sent and received. Such rules are formalized in protocols. In distributed systems, services are generally specified through *interfaces*, which are often described in an *Interface Definition Language (IDL)*. Interface definitions written in an IDL nearly always capture only the syntax of services. In other words, they specify parameters, return values, possible exceptions that can be raised, and so on. Proper specifications are complete and neutral. Complete means that everything that is necessary to make an implementation has indeed been specified and neutral means that the specification should not prescribe how an implementation should look like. Completeness and neutrality are important for interoperability and portability [8]. *Interoperability* characterizes the extent by which two implementations of systems or components from different manufacturers can co-exist and work together by relying on each other's services as specified by a common standard. *Portability* characterizes to what extent an application developed for a distributed system A can be executed, without modification, on a different distributed system B that implements the same interfaces as A [60].

Another important goal for an open distributed system is that it should be flexible, this means that it can use easily different components from different developers. To achieve flexibility in an open distributed system, it is crucial that the system is organized as a collection of relatively small and easily replaceable or adaptable components. This implies that we should not only describe the API seen by the user applications, but also the interfaces of the internal components and their interactions. This differs from monolithic approaches where the components are mixed together without a clear separation. Therefore, in monolithic systems adding, modifying or removing components is more difficult to be realized and make them to be rather closed instead of open.

Scalability

Many distributed systems must be scalable. Typical present and future applications include web-based applications, e-commerce, multimedia news services, distance learning, remote medicine, enterprise management, and network management. They should be deployable in a wide range of scales, in terms of numbers of users and services, quantities of data stored and manipulated, rates of processing, numbers of nodes, geographical coverage, and sizes of networks and storage devices. Small scales may be just as important as large scales (e.g., adapting to small systems, embedded systems). Scalability means not just the ability to operate, but to operate efficiently and with adequate quality of service, over the given range of configurations. Let's discuss now some problems that can appear when a system needs to scale.

Scalability in respect to size is a frequent problem when dealing with distributed systems. It appears when a varying amount of resources or users has to be supported by the system. In this case we are confronted with limitations of centralized services, data, and algorithms.

For example if only one service exists which has to support a large number of requests, than this service becomes a bottleneck in the whole system. The same thing is happening in case of centralized data. As example we can think of an on-line phone book. Even if more database services permit the users to query the on-line book, there is still a bottleneck created due to communication load. All the requests at some point access the same back-end database.

Another problem is to have a centralized algorithm. Here a very good example is the routing problem. In wide networks (e.g. WANs) a large number of messages are transported from one node to another. An optimal path can be calculated if all the information about the topology of the network is known by the routing algorithm. In this case an algorithm from graph theory can be executed (e.g. for determining maximum flux through the network, or the fastest connection) in order to find the desired path. But, this implies that information about the network should be collected, information that can change very often. However, only to maintain this information up-to-date, a lot of traffic will be generated. That's way a more natural approach for this problem is to have a decentralized algorithm, which has the following characteristics:

1. No machine has information about the complete system state
2. Machines make decisions based only on local, partial information
3. If a machine fails, the algorithm can still provide a solution
4. There is no time synchronization

Another issue is the geographical scalability. A WAN network suffers when compared to a LAN network in terms of communication performance and reliability. Also, the connections in a WAN network are point-to-point, so that no broadcast protocol is possible.

To solve these scalability problems there are basically three techniques: hiding communication latencies, distribution, and replication.

Communication latencies can be hidden when using *asynchronous communication.* In this way the system can do other tasks so long as some information is not yet available from the network.

Distribution involves that a component is splitted in smaller parts which are spread over the network. An example of distribution is the Internet Domain Name System (DNS). The DNS name space is hierarchically organize into a tree of domains, which are divided into nonoverlaping zones. The names in each zone are handled by a single name server.

Replication of a component in a distributed system can help to balance the load of the system and also provides a higher degree of fault tolerance. This is the case of web site mirrors, where the same data is available in different places of the Internet. If a mirror is faulty from some reason the user can choose another mirror of the same site.

2.2 Middleware

In the previous section we reviewed the characteristics of distributed systems. Also, we could see the problems that occur when building such systems. A possible solution to cope with these problems is to use a *middleware.*

The term middleware first appeared in the late 1980s to describe network connection management software, but did not come into widespread use until the mid 1990s, when network technology had achieved sufficient penetration and visibility [3]. By that time middleware had evolved into a much richer set of paradigms and services offered to make it easier and more manageable to build distributed applications. Concepts similar to today's middleware previously went under the names of network operating systems, distributed operating systems and distributed computing environments.

2.2.1 Middleware definition

In the literature, there are different middleware definitions. One which is more complete and covers more aspects of the term is: "*Middleware* is the software that assists an application to interact or communicate with other applications, networks, hardware, and/or operating systems. This software assists programmers by relieving them of complex connections needed in a distributed system. It provides tools for improving quality of service (QoS), security, message passing, directory services, file services, etc. that can be invisible to the user" [7].

Middleware provides a higher-level abstraction layer for programmers than Application Programming Interfaces (APIs) such as sockets that are provided by the operating system. This significantly reduces the burden on application programmers by relieving them of this kind of tedious and error-prone programming.

Middleware is designed to mask some of the kinds of heterogeneity that programmers of distributed systems must deal with [3]. They always mask heterogeneity of networks and hardware. Most middleware frameworks also mask heterogeneity of operating systems or programming languages, or both. Few of them, such as CORBA, also mask heterogeneity among vendor implementations of the same middleware standard. Finally, programming abstractions offered by middleware can provide transparency with respect to distribution in one or more of the following dimensions: location, concurrency, replication, failures, and mobility. The classical definition of an operating system is the software that makes the hardware useable. Similarly, middleware can be considered to be the software that makes a distributed system programmable.

Middleware is quite a general term which covers a variety of software. In the literature there are several ways to classify middleware. We avoid to delve into detail here, but we will shortly enumerate the most important middleware so that we can have a clear separation of what is the targeted research area for our middleware and what is not intended to cover:

- *Transactional middleware.* Supports transactions involving components that run on distributed hosts. Transactions oriented middleware uses the two-phase commit protocols [6] to implement distributed transactions. Examples of transaction middleware: IBM's CICS [31], BEA Tuxedo [28].

- *Message oriented midlleware (MOM).* Supports the communication between distributed system components through message queues across the network. As characteristics that differenciate MOMs from other types of middleware can be mentioned: asynchronous communication, group communication, storing of messages on persistent storage. Some examples of MOMs: IBM's MQSeries [26] and Sun's Java Message Queue [29].

- *Procedural middleware.* Extends the procedure call interface to offer the abstraction of being able to invoke a procedure whose body is across a network. Remote procedure call (RPC) systems are usually synchronous, and thus offer no potential for parallelism without using multiple threads, and they typically have limited exception handling facilities. RPC was devised by Sun Microsystems in the early 1980s.

- *Object middleware.* Makes object-oriented principles, such as object identification through references, inheritance, polymorphism, available for development of distributed systems. The communication between objects (client object and server object) can be synchronous, deferred synchronous, and asynchronous using threading policies. The Common Object Request Broker Architecture [13] is a standard for distributed object computing. It is part of the Object Management Architecture (OMA), developed by the Object Management Group

(OMG), and is the broadest distributed object middleware available in terms of scope. Another examples of object middleware are: Microsoft's DCOM [10] and Sun's Java RMI [45].

- *Component middleware.* Components are third-party deployable software modules that can be composed in ways unforeseen by their developers to produce desired behaviour [22]. A component middleware is a configuration of components which are selected either at build-time or at run-time. As examples we can mention: OMG's CORBA Component Model [14], Microsoft's .NET [47] and Sun's Enterprise Java Beans [20].

- *Publish-subscribe middleware.* A special type of message oriented middleware is the publish-subscribe middleware. Publish-subscribe systems differ from point-to-point systems in the way that the communication between the end points is anonymous, asynchronous and loosely coupled. In a publish-subscribe communication model applications use named topics rather than network addresses to distribute data. Publishers simply create a publication and give it a topic name. Then, they can send issues (data) for the already created topic. Subscribers simply create a subscription for a topic name and they instruct the middleware to take specific actions when a new issue arrives. Examples of publish-subscribe middleware are: Elvin [55], Tibco Rendezvous [32], NDDS [33].

- *Service oriented middleware.* A service oriented middleware is essentially a collection of services. A service realizes some functionalities and has the following properties:

    1. The interface to the service is platform-independent.
    2. The service can be dynamically located and invoked.
    3. The service is self-contained. That is, the service maintains its own state.

  The idea of service oriented architecture departs significantly from that of object oriented programming, which strongly suggests that you should bind data and its processing together. Therefore, the service oriented architecture has as a main goal the achieving of loose coupling between the software components, which are services. Examples of service oriented architecture are Web services like SOAP [62]. Our middleware approach uses *service oriented* principles too, more details will be presented in section 4.1.

While the existing middleware covers a broad range of application domains, they still have several shortcomings like:

- inflexibility, they do not respond well to changing requirements.
- unscalability, even that they perform good in local-area networks, they are not able to scale to wide-area networks and at the other extreme to systems with low resources, embedded systems.

As a result new middleware appeared that addresses these requirements, such as:

- *Adaptive and reflective middleware.* Adaptive middleware is software whose functional and QoS-related properties can be modified either:
  - *statically*, e.g., to reduce footprint, leverage capabilities that exist in specific platforms, enable functional subsetting, and minimize hardware/software infrastructure dependencies.
  - *dynamically*, e.g., to optimize system responses to changing environments or requirements, such as changing component interconnections, power levels, CPU/network bandwidth.

  Reflective middleware makes the internal organization of systems as well as the mechanisms used in their construction both visible and adjustable at run-time. It permits automated examination of the capabilities it offers, and automated adjustment to optimize those capabilities [52]. Examples of reflective middleware are: OpenCORBA [38], FlexiNet [30], Quarterware [56].

- *Middleware for Mobile Computing.* They address problems like network unreachability, low bandwidth connection (9600 baud), peer-to-peer connections, etc.

- *Real-time middleware.* Addresses systems which need real-time properties and will be discussed in section 2.2.3 as it makes the subject of the present thesis.

Before going to discuss about the section for our research domain in the middleware area, we will discuss about what we understand through the real-time notion, while this is an important term for the research described in the thesis.

2.2.2 Notion of real-time

Real-time systems are defined as those systems in which the correctness of the system depends not only on the logical result of the computation, but also on the time at which the results are produced [57]. As a completion to this definition we can add that "guaranteeing timing behavior requires that the system be predictable. It is also desirable that the system attain a high degree of utilization while satisfying the timing constraints of the system". In achieving predictability, a system must have bounded worst case execution times for all its internal components. Timing constraints for activities to be done in a real-time system can be more complex, but usually there are *periodic* or *aperiodic*. An activity is aperiodic when a deadline can be attached for the start time or end time of the activity or even for both. In the case of a periodic activity, a period might mean "once per time interval T" or "exactly T units apart" [58].

Depending of the correctness of the system to respect the deadlines the real-time systems can be divided in three categories:

- *Soft real-time*, when the system can tolerate some degree of latency, can miss some deadlines without compromising the whole system.

- *Hard real-time*, when a missed deadline result in a failure of the system.

- *Firm real-time*, when the result of an operation becomes worthless after the missed deadline.

Another important point when discussing about real-time systems is the moment when the activities has to be scheduled for execution, otherwise stated: the scheduling used by the real-time system. Here we can distinguish:

- *static scheduling*, priorities are assigned to the activities to be scheduled a priori and do not change during run-time. Static scheduling policies allow complex analyzes to be made before execution. Every situation can be tested and information like: deadlines, execution times, activity inter-dependencies which affect the timing constraints, order of executed activities, resources needed by each activity, can be determined during the tests. Static scheduling policies algorithms introduce almost no overhead in the system at run-time. However, they realize usually a lower utilization of the processor, because the priorities are set in concordance with the peak load for each activity. Example of static scheduling policies are: Rate monotonic (RM) [36], Fixed Priority (FP) [39].

- *dynamic scheduling*, priorities for each activity to be scheduled can be changed during run-time. Dynamic scheduling policies are more flexible, adapt better to the actual demands of the system which result in a better utilization of the processor for low or average loads of activities. The drawbacks are that they need extensible testing and they introduce a higher overhead in order to be implemented. Examples of dynamic scheduling policies are: Earliest Deadline First (EDF) [58], Least Laxity First (LLF) [41].

As a distinction from normal systems, real-time systems do not have the *fairness* characteristic. Thus, the activity which has the highest priority gets all the needed resources like: CPU, memory, etc. In non real-time systems the demand of all activities are tried to be solved on a more or less fair basis. On the other hand, non real-time systems try to maximize the overall performance of the system, as in real-time systems predictability is the main goal.

2.2.3 Middleware for real-time embedded systems

Using traditional middleware solutions can not satisfy the requirements of special distributed applications which need to react to events in a determined time. In fact traditional middleware does not consider notions such as time, priorities, in general quality of service. It performs the task in a “best effort” fashion. Furthermore, resources are limited in embedded systems. Therefore a new type of middleware emerged where issues like efficiency, predictability, and scalability are considered stringent.

Conventional middleware is developed for general purpose applications. In such systems, memory is often abundant. Thus, middleware can have at least several megabytes without affecting system performance or cost. In contrast many embedded systems have tight constraints regarding memory footprint due to cost, power consumption or weight restrictions. For example space systems, even in the last decade, frequently have less than a megabyte of on-board memory [63]. Therefore a middleware designed for embedded systems has to be able to scale down and to respect low memory footprint constraints.

As important as the memory footprint, is the real-time characteristic of the middleware. With this respect, the middleware has to offer means to user applications for controlling timing constraints, priorities, resource allocation like: CPU, memory, network bandwidth. All these require mechanisms at the middleware level to assure that the user application is able to keep the deadlines.

Some of these requirements can be specified by the user application through quality of service information. But, of course the middleware should provide necesary API functions to accept QoS information from the user side. The enforcement of the QoS information can be made in two ways:

- *passive*, the middleware can just map these information to the underlying operating system and communication system and provide the user information about the success of the enforcement.

- *active*, the middleware tries to use the resources of the underlying systems together with its own mechansims in order to realize the enforcement of the QoS information.

Nevertheless the middleware should introduce as little as possible performance overhead. While the targeted systems usually have low power CPUs the middleware should spare the precious clock cycles, especially if these can bring missed deadlines to the application. The performance overhead can be a result of: internal message buffering strategies which can produce non-uniform behavior of the middleware for different sizes of the messages, excessive data copying, long chains of method calls, inefficient data marshaling, and lack of integration with underlying real-time OS and network QoS mechanisms.

2.3 Refining the problems and the objectives

Following the introduction chapter, where an overview of the research theme of the presented thesis was given, this chapter took a step further to enlarge and deepen this view. This should make the reader more familiar with the research area. Moreover, it should make him/her discover the multitude of perspectives when looking at the research subject of this thesis. On one hand this is good because the reader can find out all the important issues that have to be tackled when discussing about the thesis subject. On the other hand, due to the fact that the number of issues is quite large, in the reminding of this section will be given only the issues which we are focusing on.

First of all it should be mentioned that the main focus of the research is to ease the development for **distributed real-time embedded systems**. As a solution to deal with the distribution we choosed to use a middleware. More precisely our middleware addresses, regarding the **distribution** problem, the following issues:

- *networks, hardware, operating system and programming language heterogeneity*. A distributed system contains many computation nodes and inherently different hardware, operating systems and programming languages over which they have to be implemented. Also, the nodes can be interconnected through different networks. Our middleware guarantees that the user doesn't have to deal with all these kinds of heterogeneity, but to focus on the logic of the application.

- *access transparency*. The user of our middleware can interact in the same manner with middleware components (services in our case), regardless of where there are: local or remote, or how there are implemented.

- *location transparency.* Middleware components can find other components regardless of location.

Regarding **embedded systems**, our middleware approach aims towards:

- high scalability
- minimize the memory footprint
- minimize the performance overhead.

A *micro-kernel* architecture concept in combination with a service architecture is proposed and evaluated in this work.

Finally, our middleware provides **real-time** features using the following techniques:

- guarantying predictability for all the internal middleware components
- giving control to the user over end-to-end system resources (e.g., memory, CPU, network bandwidth). This control can be achieved directly through middleware API or, indirectly using corresponding QoS parameters.

Chapter 3

STATE OF THE ART

This thesis explores the development of middleware for real-time embedded systems. In this chapter will be presented existing solutions in this domain which can be categorized from the point of view of standards in:

- CORBA compliant middleware
- Non CORBA middleware suitable for DRE

3.1 CORBA

The Common Object Request Broker Architecture (CORBA) is a standard for distributed object computing. It is part of the Object Management Architecture (OMA), developed by the Object Management Group (OMG). The CORBA specification [13] is supported by more than 800 organizations. CORBA specification is object-oriented and is based on a client-server model for distributed computing. It makes possible that objects are available for remote invocations regardless of the language in which they are written or the system environment (e.g., operating system, CPU type) in which they exist. The Object Request Broker (ORB) is responsible for all the mechanisms required to find the object's implementation, prepare it to receive the request, communicate the request to it, and carry the replay (if any) back to the client. Figure 3.1 shows the structure of a typical ORB. To invoke operations on a remote distributed object, a client must know the interface of the object. This interface contains methods of the object, types of data to be passed as parameters for the methods and is defined in the OMG Interface Definition Language (IDL). These interfaces can be further translated to different programming languages like C++, C, Java, Smalltalk and Ada.

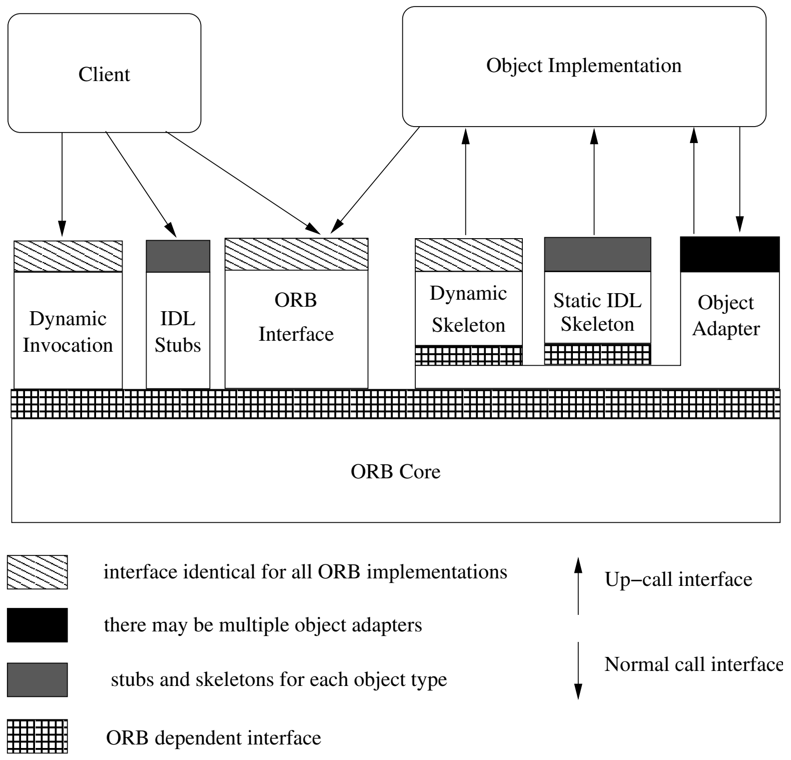


Figure 3.1: Main components of ORB architecture

In order to enable various ORBs from different vendors to communicate to each other, OMG specified a general ORB interoperability architecture named General Inter-ORB Protocol (GIOP).

3.1.1 Minimum CORBA

Minimum CORBA is a sub set of CORBA which appeared as a necessity to support systems with limited memory resources, embedded systems. Minimum CORBA has the same goals as CORBA, portability and interoperability, and represent a trade-off between usability and resource conservation. Features omitted from CORBA could still be implemented in applications as long as they are needed and system resources permits.

3.1.2 Real-time CORBA

Real-time CORBA (RT CORBA) is a set of extensions to CORBA which enable an ORB to be used in a real-time system. The goals of RT CORBA specification is to support the developer in building predictable distributed systems, by providing mechanisms to control resources like: CPU, memory, and network. The current version of RT CORBA supports only fixed priority scheduling. A Request For Proposal (RFP) for dynamic scheduling which should be addressed in next versions of RT CORBA is started. RT CORBA does not guarantee that deadlines will always meet, but aims to provide the developer a middleware which behaves in a deterministic manner. For realizing this objective it addresses a series of issues like:

- *Processor resource management.* In order to use the processor in a predictable way, RT CORBA assigns Real-time CORBA Priorities to all operations of CORBA objects. These priorities are global on all the platforms where RT CORBA is implemented and assure an uniform priority model.

- *Memory management.* RT CORBA enables the user to specify the number of threads that are used by the middleware, the amount of memory that each thread should have and the amount of memory reserved for queuing CORBA requests.

- *Network resource management.* Network resources are controlled in the following ways:

  - the application can select and configure the available network protocols. For the moment only TCP/IP has been defined by OMG (which is not a real-time communication mechanism), but other vendors can implement other network protocols, e.g., VME/PCI, IP Multicast, 1394, etc.
  - the client can obtain a private connection to a server, which will be not shared with other connections
  - the client can request to use multiple connections to the server. The way that RT CORBA deals with this connections is transparent to the application.

As described earlier, OMG addresses real-time distributed applications through a separate extension to the CORBA specification which is called CORBA Real-time. Also, the embedded systems have different requirements than general purpose middleware for distributed systems and these are reflected by OMG in a subset of the CORBA specification named Minimum CORBA. First, available implementations which comply with one or both of these tailored specifications of CORBA, are discussed. Later, non CORBA compliant approaches are presented.

An overview will be given afterwards which can help the reader to observe the differences between the discussed middleware in relation with: memory footprint, real-time, scalability and architecture design techniques.

3.2 CORBA compliant middleware

The most significant CORBA implementations for the domain of embedded real-time distributed applications are: TAO/ZEN, ORBExpress RT, e*ORB RT, Visibroker RT, ROFES. They will be described in the following section.

3.2.1 ORBExpress RT

ORBExpress RT is one of the few CORBA implementations which addresses real-time embedded applications. The product is part of the ORBExpress family developed by Objective Interface Systems since 1997. The latency and memory overhead are significantly reduced compared with other real-time middleware solutions as shown in a study conducted by Boeing [12]. It provides Distributed Priority Inheritance and propagation which are essential features for maintaining end-to-end predictability. Focusing on embedded systems, it has the ability to adapt to different transport media which are used by these systems. Here can be enumerated: ATM, 1394, IP Multicast, VME/PCI bus, shared memory or even a proprietary one. The proprietary transports can be implemented and plugged in the platform by the developer.

From the information provided by the company, the minimum footprint which the Orb can reach, using a highly customized configuration (processor, operating system, network, etc.), is 93k. However, this configuration does not provide real-time capabilities. On the other side, when using the Orb in more common operating conditions, like on an embedded real-time operating system, VxWorks, and a PowerPc processor, the footprint is around 168 KB. Also, in this configuration of the Orb, real-time functionalities are not included.

3.2.2 TAO

TAO is an open source Orb developed by Douglas Schmidt and his research group from Universities of Washington and California, Irvine. It is developed on top of a framework called Adaptive Communication Environment (ACE), which is a widely used, freely-available object-oriented framework that contains a rich set of components implementing design patterns for high-performance and real-time communication systems. One of the strengths of the TAO middleware is the use of designed paterns [48] [51]. They describe solutions to common problems which can appear quite often in application development, together with advantages, disadvantages and practical examples for these solutions. Because of using these high level principles in development, the TAO middleware is easy to be understandable by the developers. As a disadvantage of applying these general solutions for special applications, like those executed on embedded systems, is exactly its generality. And this, because more specific, more optimized (e.g., for memory footprint) solutions should be considered for the same problems. This was one of the reasons why the research group started a new development process for the RTZen.

It is important here to mention that the TAO research group played an important role in influencing the OMG's Real-time CORBA specification. They have used TAO or ACE in projects with important companies like: Boeing, Motorola, BBN, Lucent, Nokia, Lockheed, Microsoft, Cisco, DARPA, etc. and the open community around these projects count more than 100 persons.

TAO development was started with the aim to realize a CORBA compliant real-time Orb which should provide high quality reusable software that can be used in mission critical applications. Later, the goal to adapt

TAO to embedded systems made the research group to find techniques for tailoring the Orb to the memory constraints imposed by the new targeted applications domain. With a memory footprint for the complete C++ libraries of about 1.4 MB(for Linux), TAO is still not appropriate for the use on embedded systems. However, in the last period, commercial support is provided by a series of companies as OCI, Riverace, PrismTech which includes consulting regarding the techniques used to tailor the Orb to the specific targeted embedded system.

3.2.3 RTZen

RTZen is developed by the same team of Douglas Schmidt and is based on the ACE framework too. Even that the techniques used are quite the same - applying design patterns for solving the problems - the focus is to develop a Java real-time Orb for embedded systems. The Orb is based on a micro-kernel architecture to minimize the footprint and is implemented using the Real-time Specification of Java (RTSJ) in order to acheive predictability of the middeware in the Java world. Another important aspect is that RTZen can run on top of a conventional Java Virtual Machine (JVM) and also on interpreted and ahead-of-time compiled RTSJ platforms. For using RTZen on an ahead-of-time compiled RTSJ platform, the research team developed an additional tool, jRate (Java Real-Time Extension), which is an extension of the GNU GCJ compiler and compiles java class files in target executable files. The footprint of the actual implementation of RTZen is about 896 KB, and implements partially the RT CORBA standard.

3.2.4 OpenFusion e*ORB

OpenFusion e*ORB is one of the most important Orbs in the domain of embedded real-time systems. Together with ORBExpress, they were the only ones which focused from the begining toward systems with low resources and later, providing also real-time capabilities. This brings to the OpenFusion e*ORB the advantage of a bottom-up evolving in contrast with the top-down strategie used by other Orbs to adapt from normal systems to small resource systems. It was in 1998 the only implementation of a C++ CORBA ORB running on handheld device Operating Systems.

The design of the Orb is based on a micro-kernel architecture, making it possible to adapt to small systems. What is remarkable for this Orb, is the high control of the user over the components of the Orb. The micro-kernel architecture provides to the user the ability to easily add Orb components as modules or libraries, and configure them according with the application needs. Moreover, the Orb provides a set of defined points where the user can intercept even the interactions between the Orb components, providing a deeper information about the actions performed by the Orb. This can substantially help the user to understand and eventually to optimize its application in order to meet the required real-time properties. The Orb defined points are called interception pluggable boundaries and can be observed in the Fig. 3.2.

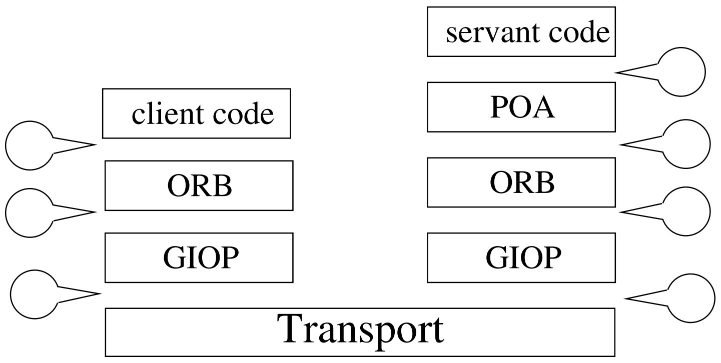


Figure 3.2: OpenFusion e*ORB's Open Pluggable Boundaries

Some of the componenets which the user can configure are:

- communication: the wire protocol, wire transport can be user implemented, so protocols which are not supported by the company can be added.

- POA: the size of the Portable Object Adapter(POA) can be adapted to the user application. If the user needs a more complex adapter, having so called child POAs, new code is plugged to the Orb which provides this functionality.

- different thread pools strategies can be plugged in by the user.
- different queue dispatching techniques are available as plug-in modules.

Its reduced footprint permited to adapt even to Digital Signaling Processing (DSP) environments, which have severe requirements regarding the processing power and the amount of memory. The footprint for a minimum real-time server for e*Orb 2.3 in C++, using shared libraries, without debugging information, on a Linux OS with a 8086 compatible processor is about 1.1 MB. On the other side the company states that a minimum client can reach a size from 100 KB.

3.2.5 Visibroker RT

Visibroker RT is developed by Borland and is used by some important companies like: Cisco Systems, Deutsche Bank, Ericsson, Hitachi, Nokia, Sun Microsystems, etc. Like the other Orbs designed for embedded systems it provides a mechanism to plug in, beside the TCP/IP network protocol, other custom network protocols needed by the embedded applications. Notable issues which has to be mentioned for Visibroker RT are:

- communication is optimized for in-process communication between clients and servers. The performance obtained is comparable to direct function call.
- the Orb can be tested without target hardware, using tools delivered together with Visibroker which simulate the hardware.
- like OpenFusion e*ORB, the Orb exposes to the user an interface which can be modified in order to debug the Orb and get more information about the Orb execution

The footprint of Visibroker's RT 6 shared libraries for C++, without debug information and not using exceptions, is on a Linux OS with a Pentium 4 processor, approximatively 5 MB for the Minimum RT CORBA 2.6 specification. As an important goal for Visibroker is to speed up the development process, a console visualization tool is available which helps the user to inspect different aspects of the Orb itself or of the distributed objects managed by the Orb.

3.2.6 ROFES

ROFES is a middleware developed at the Faculty for Electrical Engineering and Information Technology Aachen, Germany [50]. The actual implementation of ROFES, version 0.4, implements partly the RT CORBA specification. One of the aims of the development team is to realize in the near future a fully compliant RT CORBA Orb. ROFES provides support for other network mediums like: CAN Bus (Controller Area Network) [9] and Scalable Coherent Interface (SCI) [27]. It uses a microkernel architecture which permits that different components can be dynamically loaded into the platform. It is notable to mention that the size of the C++ library, without debug information, exceptions and run time type identification is about 324 KB.

3.3 Non CORBA middleware suitable for DRE

3.3.1 NDDS

NDDS (Network Data Delivery Service) is a publish-subscribe middleware for distributed real-time applications developed by Real-Time Innovations [33].

NDDS provides real-time by allowing to set a number of parameters for subscription and publication which give the programmer control over:

- *Subscription deadline.* This deadline lets the developer to set how long to wait for the next publication issue.
- *Subscription minimum separation.* Minimum separation lets the developer set the highest rate at which the application receives data.
- *Publication strength.* Strength lets the developer establish hierarchies among applications publishing the same topic. The idea is similar with having priorities for issues for the same topic.

- *Publication persistence.* Persistence lets the developer define how long each issue is valid.

It is important to note here that Real-Time Innovations was the lead author for the Data Distribution Service Standard Specification [17] adopted by OMG. The specification establishes a standard interface for publish-subscribe communications for real-time systems.

3.3.2 MidART

MidART stands for Middleware and network Architecture for distributed Real-Time systems. MidART was developed starting with 1995 at Misubishi Electrical Research Laboratories. The main focus is to enable easy communication over different networks. Thus, they have introduced some new concepts: RT-CRM (Real-Time Channel based Reflective Memory) and Selective Real-time Channels. RT-CRM is a software-based unidirectional reflective memory. It provides data reflection with guaranteed timeliness, while allowing applications to specify how and when data are reflected. Data reflection can be done per period or upon a write. The communication model is similar with publish-subscribe paradigm. The server application creates a reflective memory area and all new data written in this area are sent to the remote clients which register to this area. The communications created in order to enable the data transfer are unidirectional. A distinction of the MidART is the admission control support, which guarantees that the QOS parameters specified by user for timeliness and transfer rates can be achieved by the middleware.

3.4 Conclusions

This section presented existing solutions in the area of distributed real-time embedded systems. All of them address the requirements for DRE systems, requirements which get different weights in the overall end solution. For example, interoperability is one aspect which becomes important specially for large distributed applications which spread over a highly heterogeneous environment and use, probably, different specialized middleware. For this, CORBA solutions are superior to the others, because they implement the same standard which permit them to easily inter-communicate. On the other side, non CORBA solutions are not restricted to a standard and have more freedom when choosing necessary means to reach the solution.

For each described middleware there were presented some of the defining ideas which delimit them from the others. Table 3.1 realizes a summary of these solutions. We will also include in the comparision our middleware approach which is described in the next two chapters. Our middleware is named OSA+, which stands for **O**pen **S**ystem **A**rchitecture, **P**latform for **U**niversal **S**ervices. The comparison criteria are described below:

- *Footprint*, the three sizes have the following meaning:
  - total size of the middleware
  - size of an unoptimezed minimum application
  - size of an optimized and highly configured minimum application

  For the cases where only the total size of the middleware is present, there are included also additional features which can be excluded when a minimum application should be built. To reduce the size of an end product, we can use the following techniques:
  - remove debug information from libraries, or classes in case of Java
  - make custom libraries for the specific application and remove from them, the symbols which are not used in the application
  - rename classes, methods, variables with shorter names (there are tools which automatically make this)
- *Scalability*:
  - - : not enough information is available
  - 0 : the middleware is able to scale down, but not for critical systems where resources are minimum
  - + : the middleware is able to scale down to low resource embedded systems

Middleware	Footprint	Scalability	Architecture	Middleware category	Standards	Programming language
e*Orb	- 7,9 MB sh. libs - 1,1 MB sh. libs -> 100KB	+	compositional microkernel	object oriented	- RT CORBA - Minimum CORBA	C, C++
OrbExpress RT	- - 168 KB - 93 KB	+	-	object oriented	- RT CORBA	C++, Ada
TAO	- 1,5 MB sh. libs - -	0	design patterns	object oriented	- RT CORBA - Minimum CORBA	C++
RTZen	- 896 KB - -	+	compositional microkernel	object oriented	- part. RT CORBA	Java
Visibroker RT	- 5 MB sh. libs - -	-	-	object oriented	- RT CORBA - Minimum CORBA	C++
ROFES	- 324 KB - -	+	compositional microkernel	object oriented	- part. RT CORBA	C++
NDDS	- - -	-	-	publish/ subscribe	- DSS from CORBA - RTPS	C
MidART	- - -	-	-	publish/ subscribe	-	C++
Our approach (OSA+)	- 75 KB - 53 KB - 29 KB	++	full microkernel	service oriented	-	Java, C

Table 3.1: State of the art, middleware for DRE

  - ++ : the middleware is able to scale down to very low resource embedded systems

- *Architecture*:

  - compositional microkernel: the middleware uses a microkernel concept only for composing a desired middleware configuration based on some building blocks which extend the features of the middleware. The building blocks are used as libraries which are added to the application and are tightly coupled through linking or compiling dependencies. Therefore, the middleware components can not be plugged in the middleware if these dependencies are not satisfied. For example using e*Orb is not possible to build a real-time application without using the real-time mutex library due to linking dependencies. Another example would be that a client needs at compile time the interface definition of the server when using the CORBA static invocation method for the objects. This can be avoided using the dynamic invocation method. But, this introduces much more degradation in terms of code size and performance and is not suitable for applications running on embedded systems. Another important issue is that the compositional microkernel doesn't provide a *real medium for inter-communication* between the components, which normally introduces a relaxing of the coupling between the components. Further, each component has its individual interface.

- full microkernel: additionally to the compositional microkernel concept, the full microkernel architecture provides a basic mechanism of communication between components. Another important element is that components do not have compilation or linking dependencies, but only semantic dependencies, so they can exist and provide reduced functionality if other dependencies are not present on the platform. The components are entities which are self-contained and share the same interface. This significantly increase the flexibility and scalability. Figure 3.3 shows the difference.
- design patterns: for reaching the goals, the middleware uses clearly stated solutions for a class of recurring problems. These solutions are analyzed and exposed to the user in a standard way.

As it can be seen the memory footprint for all presented middleware is not suitable for a system with less than 100 KBytes. Furthermore, non of them have realized a full microkernel. To delimit the work presented in this thesis from the other presented middleware, the last line of the table contains as a preview of chapters 4 and 5 the comparison values for our approach named OSA+:

- footprint: OSA+ has the smallest footprint which makes it easily adapt to very low resource embedded systems.
- full microkernel architecture: apart from the facts presented in the previous paragraph, OSA+ provides a uniform interface for its building components (services). This means that all the components, regardless if they are middleware components or user components, can be included in the platform in the same manner. No difference is made. This leads to a maximum flexibility and scalability.

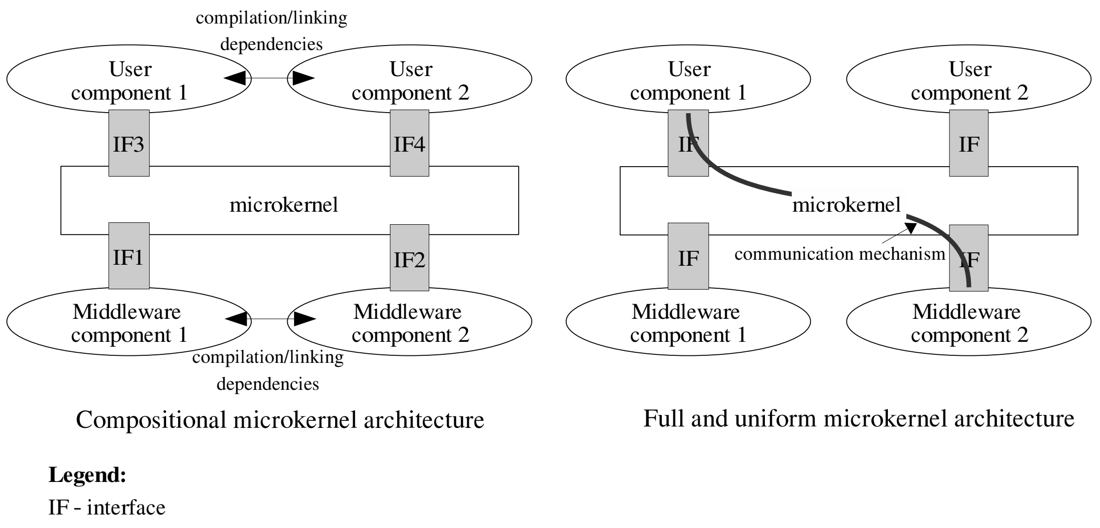


Figure 3.3: Microkernel architectures comparison

- service orientation: OSA+ is a service oriented architecture. As we discussed in section 2.2, services are defined by their platform and language independent interface. They are defined at a higher abstraction level than the objects, which use a finer level of granularity. Thus, the management of services is more simple than in the case of objects, because they are simply less in number than objects. Another advantage, when comparing with object oriented middleware, is that services are self-contained. Once they are activated they have their own execution flow and life cycle. On the other side, objects get the execution flow of some thread. Therefore, additional techniques to manage thread pools have to be implemented on object oriented middleware.

Chapter 4

THESIS APPROACH

In the previous chapters we have presented, in the following order:

1. research goals of the thesis

2. research domain, which helps the reader to understand the research background and the problems which appear when trying to reach the goals

3. and finally, existing solutions, which inform us about what is already done and give a first outline of what is defining this research from the other solutions.

As a natural follow-up, in this chapter, we will present our approach to ease and improve the applications development for distributed real-time embedded systems. As a new concept for embedded real-time middleware, OSA+ is combining a purely service oriented concept with the full microkernel architecture. To explain the design and the design motivation, this chapter is structered in three sections, which describe the following issues:

- the general service oriented architecture of OSA+ and how it solves the distribution problem

- how OSA+ adapts to embedded systems due to the microkernel principles

- what the means to provide real-time are.

4.1 OSA+ general architecture

OSA+ is a service oriented middleware for DRE systems. It facilitates the applications to be easily deployed and to communicate over a heterogenous environment. In OSA+, the active communication parts are *services*. A service, as already mentioned in section 2.2, realizes some functionalities which are made public to the execution environment through an interface. This interface can be accessed in a platform and a language independent manner. In our case, the service interface is accessed through *jobs*. A job consists of an order and a result. The order is sent from one service to another to state what functionality the service should do, for which data, and when the action should be performed. After the service executes the order, a result is sent back. The communication of jobs is accomplished by a platform. The platform facilitates the plugging of services which can communicate with each other. Figure 4.1 shows the service oriented architecture of OSA+.

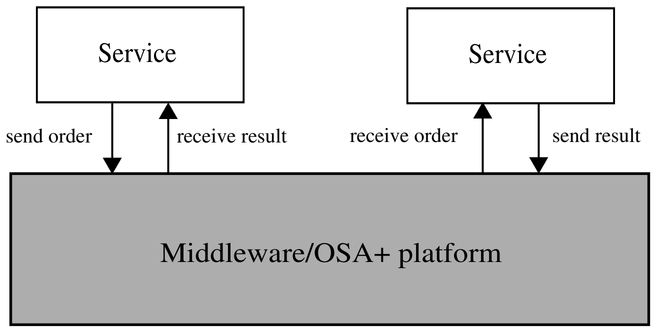


Figure 4.1: OSA+: service oriented architecture

A natural question arrises now: why did we choose, in contrast to common real-time middleware approaches, to have a service oriented architecture for our middleware?

The motivation is based on the following reasons:

- higher abstraction level concept independent of the programming language
- uniform approach

- loose coupling
- platform and programming language independent access to the service
- and as a consequence of all these, flexibility and efficieny

Let's explain these ideas.

A service concept is defined at a higher abstraction level and, is independent of programming language. The essence of a service is given by the functionality of the service, by what it does. It can be argued that a service concept is equal, when considering the abstraction level, with an object concept used in OO (Object oriented) programming. In fact, they both expose their functionality which is one of the main issues for both concepts, but the significant difference comes in place when considering granularity. Objects are finer in granularity than services. A service can be in fact mapped to an object in an OO programming language, which would be composed inherently form many other objects to realize its functionality. To see the difference of abstraction level an example will be helpful. Let's consider the functionality of moving from one town to the other. This can be done (not in all the cases) by a car. In a service oriented architecture, the relevant issues are that the car can realize the moving service and the quality of this service, e.g., car speed and comfort. In an object oriented programming there are additional issues which have to be considered, like:

- granularity: each other piece of the car is also an object
- how the car is built: the car object is composed using a certain arrangement from other thousands of objects which are the pieces of the car.

Therefore, the management of services is simpler, because they are not syntactically inter-dependent, but only semantically and they are normally less in number due to the higher level of abstraction.

With a service oriented architecture, an uniform middleware approach si possible. As we will see in the next chapters, services are the only necessary instances in our middleware to perform application and system tasks. In contrast with other architectures, no libraries, shared objects, etc. are needed. Middleware configuration, middleware extensions and the applications are realized by the same service structure.

Service oriented architectures realize a loose coupling between data and its processing. This, again contrasts with object oriented programming. To get a better picture about this, let's consider another example. A CD player realizes the functionality of playing CDs. This functionality can be provided by other CD players: a portable player, the player in the car, etc. So, you can use your CD on all this players, getting a different quality of service when listening the CD. This is exactly what is happening in a service oriented architecture, there is no binding between the data, in our case the CD, and the processing, in our case the CD player. But in an object oriented programming, using this example, every CD would come with its own CD player which is able to play it.

In our middleware, the access to the service is realized through jobs which are, in fact, messages. These messages have some special format which are recognized by the platform and by the services which exchange them. Nevertheless, they are platform and programming language independent. Thus, the job communication makes the access to the services to be platform and programming language independent. In fact the access is dependent in our case only by the semantic interface of the service which dictates the content of the messages.

CORBA succeeded in some extent to raise the level of abstraction for objects and make it less dependent of the execution environment and the programming language (a CORBA object abstraction can be mapped to the C language as well). But still there are a number of dependencies, which bind the object from the CORBA processing model, like: object activation techniques through thread pools, and more important the interface to the object, which is programming language dependent and generates mandatory changes in the application in the case it is modified.

These properties provide flexibility and efficiency. Due to the uniform service approach, the middleware can be adapted and scaled in a very flexible way, as it is shown in the following section. Since service management is simpler than object management, the management overhead can be reduced and the efficiency can be increased.

4.1.1 OSA+ as middleware

First of all, OSA+ is a middleware and it must solve the distribution problem. In section 2.3 we enumerated the issues which we tackle regarding distribution, and these are:

- networks, hardware, operating system and programming language heterogeneity.
- access transparency
- location transparency

Solving heterogeneity problem

To solve the programming language heterogeneity we have to port OSA+ concepts to different languages. At this moment, OSA+ is ported to C and Java.

To deal with the other mentioned heterogeneities, we divided our middleware in an environment independent part which is the *OSA+ core platform*, and a part which adapts the middleware to the environment.

The OSA+ core platform contains no hardware nor operating system dependent parts. It is maintained as small as possible and provides basic functionalities which gives the posibility to the user to extend the platform and configure for its needs (a microkernel, see section 4.2). Because the core platform is environment independent and contains the user interface of the middleware, the applications developed using OSA+ middleware will implement the same interface on every platform where OSA+ is ported. This shields the applications from the environment changes, and help the user to focus more on the logic part of the application.

The adaptation part consists from services which are implemented by the middleware developers. It adapts to every operating system where OSA+ is ported and also to different communication mediums. These services are part from the *basic services* and they are: Process Service, Event Service, Memory Service, and Communication Services. They will be described in section 4.1.3. Currently, OSA+ is ported on Windows NT 4.0, VxWorks, Linux RT, Komodo micro-controller and due to the fact that is available in

Java, OSA+ is running on all the operating systems where a java virtual machine is implemented. In figure 4.2 we can see how OSA+ addresses platform heterogeneity.

As mentioned before, the service is the uniform instance. There is no difference between application services and basic services. They share the same API, the same structure and communication principles.

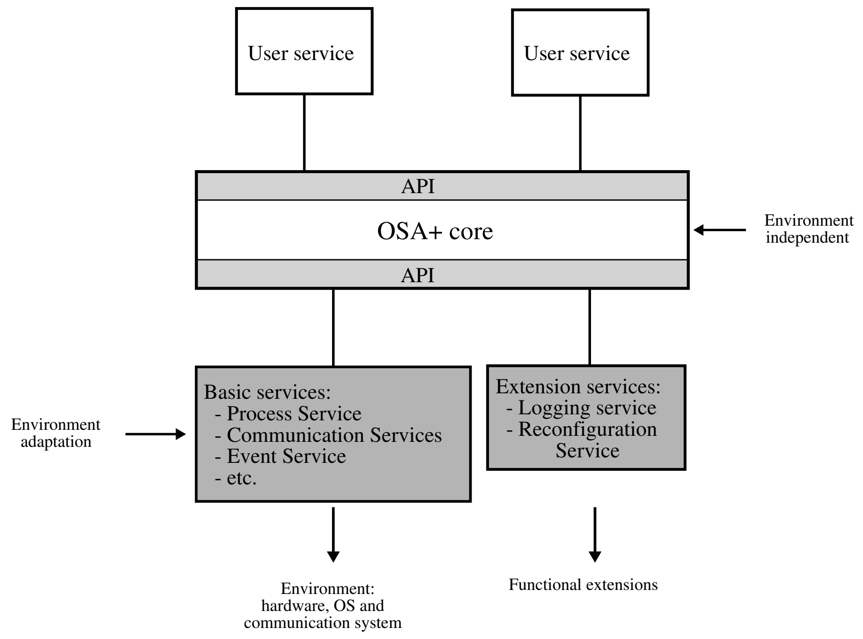


Figure 4.2: OSA+: adapting to different environments

Besides the services which adapt the middleware to the environment, the basic services contain an additional service called ARS (Address Resolution Service). This service permits the user to locate remote services and local services in the same manner, providing location transparency to the middleware. More about this service will be presented in section 4.1.3.

Once a service is located through the ARS service, the user can send jobs to the service. The middleware will deliver the jobs to the service whether the service is local or remote. The access to the service, which in our case is the action to send jobs to the service, is transparent to the user. All OSA+ platforms involved in the communication between two services will work together to deliver the jobs to the corresponding services.

Having location and access transparency the user is not anymore concerned about the issues which occur when having the application on different physical platforms, but perceives the environment presented by our middleware as a overall virtual platform, figure 4.3.

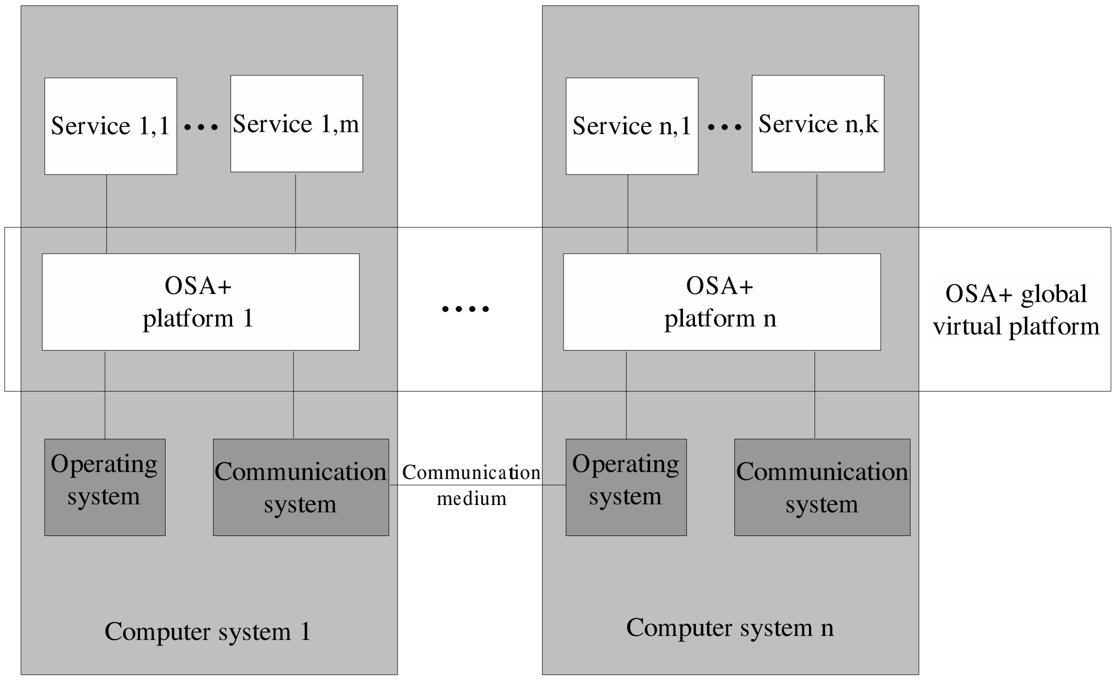


Figure 4.3: OSA+ virtual platform

Additional services can be provided by the middleware which extend the functionality of the middleware, these are *extension services*. An example of extension service is the Reconfiguration Service which can be added on the platforms where there is a need to replace, remove or add services at run-time [53]. Again, the service is used as the uniform concept to extend the functionality.

4.1.2 OSA+ core platform

The core platform, as already mentioned, provides basic functionalities in order to extend the core and to adapt to the demands of the application and the environment. The basic functionalities are:

- the middleware user interface: the API
- service management: services can be plugged into the platform

- local job management: jobs can be exchanged between local services
- basic support for quality of service.

They will be described in the following sections.

User interface

The user is able to interact and use the middleware through the interface presented by the middleware, so called API (Application Programming Interface). When discussing about the API for a middleware we have to address two points:

1. what functionalities offers the middleware to the user, through the API
2. and, how the API looks for different programming languages.

For the first point, we already mentioned that the core platform contains the necesarry functions to manage services, jobs, and quality of service information. These functions will be described in the next sections.

The second point is related to the portability. Applications which are not using a middleware and must be ported on different operating systems have to solve the portability problem on their own. A good solution is to base on APIs which are the same on every different platform where they will be executed. For example, applications written in C which are based on standards like ANSI C and POSIX, can be easily adapted to run on all the operating systems where these standards are ported. This solution can be applied when the application contains all the components written in the same programming language. While middleware tends to solve the portability for programming languages too, the problem is solved by specifying their API in an interface definition language (IDL). This specification is then mapped to a programming language using a middleware tool. This tool generates the necessary code and frames which are simply filled by the user with the code desired for each function.

In our case we didn't spent effort to realise such tools for the following reasons:

- our middleware API is quite simple and it contains a relative reduced number of functions. This eases considerable the use of the middleware without the help of extra tools which would map the IDL interface to C or Java for our case.

- the time spent developing such tools, require additional human resources which were not available.

Services and service management

Services are the active entities which can be added to the platform in order to extend its functionality. A service can exist in our middleware in three forms:

- as a procedural service

- as a lightweight service

- and as a heavyweight service.

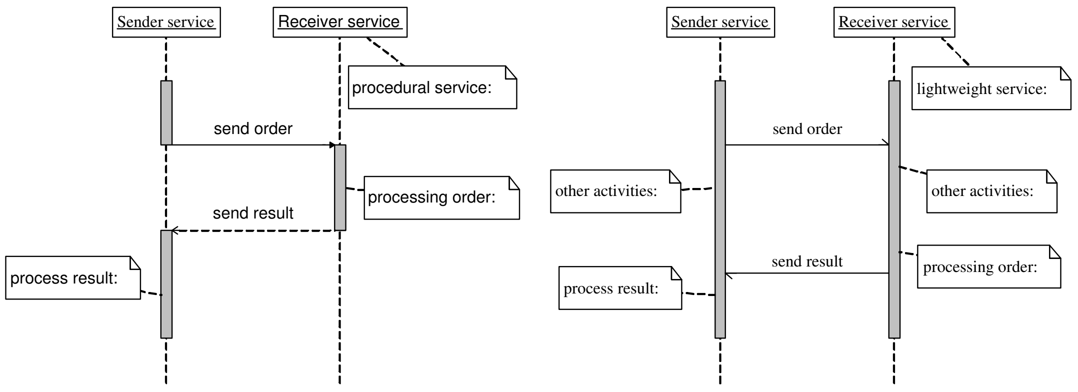
A *procedural service* does not have its own control flow, but takes the control flow of the service which sent the order, when processing the received orders. In this case only a synchronous communication is possible, so the sender of the order will be able to do other activities after the procedural service is finishing to process the order, figure 4.4(a).

A *lightweight service* has its own control flow and has the same address space as the platform. In this case it is possible to have asynchronous communication. The middleware will make sure that the order will be sent to the service receiver, while the service sender can do other activities, figure 4.4(b). The platform is able to create lightweight services only if the Process Service is present on the platform. This service adapts the middleware to the task management of the operating system (see section 4.1.3) and creates threads which will host a lightweight service. For each lightweight service, a priority can be specified. This separates the services by their importance. Priorities are important for real-time and help for realizing the predictability property for our middleware. More about service priorities will be discussed in the real-time section 4.4.

A *heavyweight service* has its own control flow, but has a different address space than the platform. To communicate with such a service, the platform needs to have inter-process communication functionality included in the Process Service.

The main reason to introduce procedural services on one side and lightweight and heavyweight services on the other side is again efficiency: many tasks in the application don't need a control flow of its own. Thus, overhead and complexity can be reduced by using procedural services. Furthermore, the concept of the procedural service is vital for the entire uniform service architecture: since light- and heavyweight services are only possible when the Process Service is present to adapt to the operating system, thus Process Service necessarily must be a procedural service. Otherwise, this service could never be plugged in.

To distinguish between lightweight and heavyweight services is for efficiency and flexibility reasons, too. Heavyweight services offer more protection while lightweight services run more efficiently due to the same address space.



(a) Procedural service (b) Lightweight service

Legend: bars indicate control flow

Figure 4.4: Procedural and lightweight services

The user can enable a service on the platform by building first the service and then make it known to the platform. The platform has a `Service` class which has to be derived by the user to implement a custom service. This class provides functions which a service can do, like: `sendOrder`, `sendResult`, `waitOrder`, `waitResult`, etc.

On the other side, the core platform realizes a basic service management. It offers two functions which add or remove a service from the platform: `registerService` and `unregisterService`. By adding a service to the platform, the user must provide: the name and the version of the service, the service type (ex. lightweight) and a priority. The function returns an ID for the service which identifies uniquely the service on the local platform. The core platform realizes the service management by applying the Service Configurator design pattern [34].

Patterns have roots in Cristopher Alexander's work on urban planning and building architecture [2]. Software design patterns first became popular with the object-oriented *Design Patterns* book [25].

A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems. It describes the problem, the solution, when to apply the solution, and its consequences. It also gives implementation hints and examples. The solution is a general arrangement of objects and classes that solve the problem. The solution is customized and implemented to solve the problem in a particular context. [25]

A very important advantage of design patterns is the fact that they speed up the development process by providing an almost ready made solution that has been used earlier and proved to be efficient. Commonly used design patterns also have the potential of being revised and improved over time, and thus are more likely to perform better than home made designs.

The Service Configurator pattern decouples the implementation of services from the time at which the services are configured into an application or a system. This decoupling improves modularity of the services and allows the services to evolve over time independently of configuration issues, such as whether or not two services must be co-located or what concurrency model will be used to execute the services. In addition, the Service Configurator pattern provides centralized administration of all the services it configures. This facilitates automatic initialization and termination of the services and can optimize performance by performing common service initialization and termination activities [34].

The Service Configurator pattern requires all services to have an uniform interface for configuration and control. This allows the services to be treated as building blocks that can be easily integrated as components in our platform. The uniform interface which must be present by all the services is given by the `Service` class of the core and has two parts:

- a mandatory syntactic interface, which realizes the physical binding (due to the programming language) to the platform
- and a mandatory semantic interface, which presents the functionalities provided by the service. This interface is accessed through the orders that can be executed by the service.

Furthermore, the service can present its semantic interface through an optional extension of the syntactic interface, which is called *procedural interface*. The procedural interface contains functions which can be called directly to execute the service functionalities. The procedural interface of the service can be obtained by executting the `getInterface` function of the service. The advantages and motivation of having this optional interface are described in section 4.2.2. Figure 4.5 shows the Service Configurator pattern used by our middleware.

The mandatory syntactic interface of a service contains the following functions:

- `getInterface`, returns the optional procedural interface of the service
- `serviceLoop`, represents the execution body of the service and must be overwritten by the user. For lightweight services, this function is the start function for the thread which will host the service. For procedural services, the function will be executed by the platform every time when a new order is received.
- `orderSignaled`, executed every time an order is received for the service. This function has to be as simple as possible and used only to signal that an order is waiting to be processed by the service.

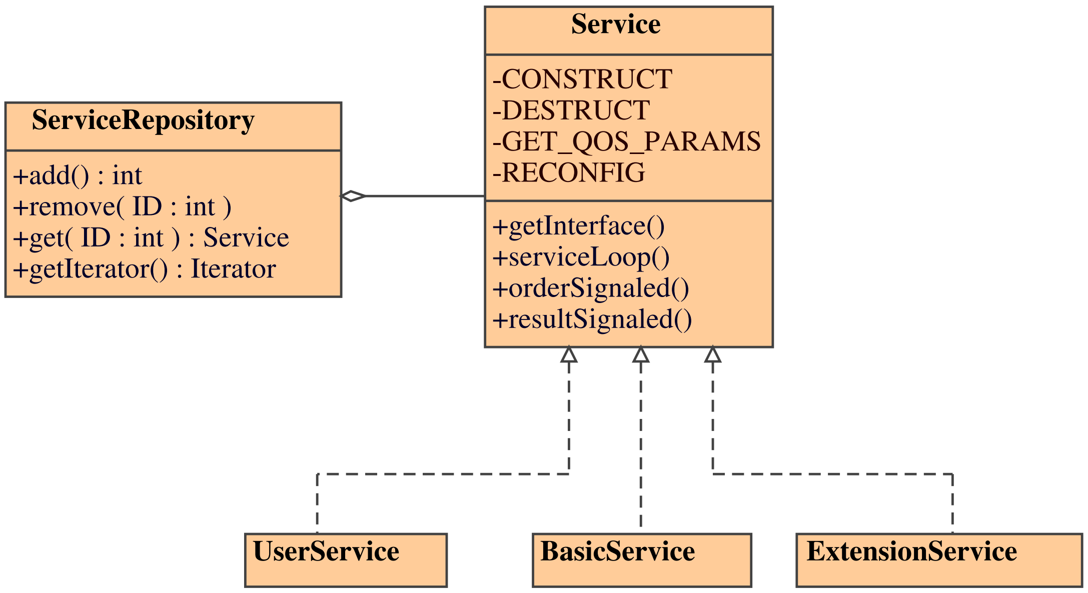


Figure 4.5: Service Configurator design pattern

- `resultSignaled`, executed every time a result is received for the service. The same discussion from orderSignaled is valid here.

The mandatory semantic interface of a service consists of four platform predefined orders which a service has to recognize. These are:

- `CONSTRUCT`, this is an order which is sent by the platform once the service is registered on the platform. This is the first order received by a service. The service can use it to allocate resources needed for its life time.

- `DESTRUCT`, this order is sent by the platform when the service is unregistered. The service can use this order to deallocate platform resources or even to reject its unregistration.

- `GET_QOS_PARAMS`, when receiving this order the service should respond with the QoS parameters supported by the service, if there are any. More about QoS will be discussed in section 4.1.2.

- `RECONFIG`, this order is sent by the Dynamic Reconfiguration Service to announce that this service should be reconfigured with another one. The service to be reconfigured can act accordingly and do special actions like: prepare the current state of the service to be transfered to the new service, reduce at minimum the number of activities made by the service, etc. This is part of another PhD thesis [53].

Beside the mandatory orders, any user defined and application specific order can be used to realize applications.

The `ServiceRepository` is a container which maintains all services registered on the platform. It has to use as less memory as possible, but in the same time it should adapt for cases when a large number of services are present on the platform. For static applications where the number of services is known and will not vary, the ServiceRepository can be configured as a simple array. For more dynamic applications the ServiceRepository is implemented as a `BucketContainer` which is explained in section 4.4.1. The ServiceRepository has four basic functions:

- `add`, which adds a service to the repository and returns an unique ID
- `remove`, which removes a service from the repository. The ID of the service to be removed is provided by the user.
- `get`, which returns the reference to a service with the ID specified by the user
- `getIterator`, returns an Iterator which can be used to enumerate all services actually maintained by the ServiceRepository. This iterator is used for example by the platform when searching for a service knowing its name.

Communication principles

Now that we know how the services can be built and included in the platform, the next step will be to define how they exchange information. Our middleware approach provides three ways to exchange information between services which are shown in figure 4.6 and are discussed further.

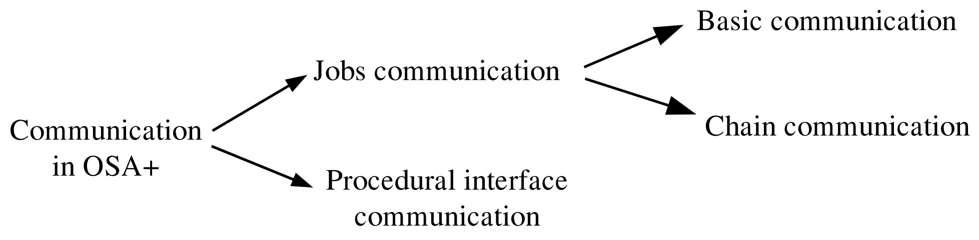


Figure 4.6: Communication methods in OSA+

Basic job communication

The basic job communication mechanism realizes the communication between services through messages. These messages are divided in orders and results, where a pair (order, result) forms a job. The basic job mechanism will be described step by step as it is used in a real application. These steps are:

1. identifying the destination service for the orders
2. building the order which has to be sent to the destination service
3. filling the order with a message
4. choosing between connection-oriented or connectionless communication
5. sending the order and getting the result.

1. Identifying the destination service A communication is needed, for example, when the user wants to transport some data over the network or use some functionalities of a service - process the data and get back some results. In both cases the user needs to specify the ending point of the communication, the destination for its data. Our middleware presents the services to the user as name and version pairs. In fact, to identify uniquely a service on the global platform, the middleware needs more information, like:

- information to uniquely identify the platform. This can be IP address when communicating using internet protocol as network protocol or serial port when using serial communication as network

- information to uniquely identify the service on the host platform. This is the local service ID given by the platform once the service is registered.

But, the middleware should be location transparent and should shield the user from these details. Therefore, the core offers a `lookUpService` function which searches a service by its name and/or version and returns back a `lookUpInfo` object which contains information about the found service. This object can be further used to send orders to the located service. The `lookUpService` function will look first for services on the local platform, and if no service is found, will use the ARS service to search on the global platform.
For identifying a service, the user has also the possibility to use directly the local ID of the service when a service is on the local platform and the ID is known.

2. Building the order Orders can be built by the registered services using the `buildOrder` function of the `Service` class. This function will return back an order which can be transmitted to the destination service, but it doesn't contain any message. For the moment, the order represents only the envelope for the message. An `Order` object has the following relevant members regarding the basic communication mechanism (figure 4.7):

- `serviceDestId`, the destination service for this order. This parameter can be empty in the case that a `lookUpInfo` object is used further for the communication.
- `serviceSourceId`, the service which built and sent the order. This service will get back the result in case it exists.
- `orderId`, this identifies the functionality of the destination service requested through this order.
- `priority`, this member is important for real-time applications where different priority levels have to exist in order to indicate the importance of each operation. In case that more orders have to be executed by a service, the order with a higher priority will be executed first.

This priority is inherited by the result and becomes the priority of the whole job.

- `resultSize`, in case that the processing of this order will produce a result, the size of the result is needed by the platform for reserving necessary resources.
- `orderData`, this contains the real message which has to reach the destination service of the order. The message can contain different parameters for the service functionality selected by this order. These parameters can be filled before sending the order using platform functions. On the destination service, the parameters will be read and processed.
- `endian`, this indicates the byte order format for the message parameters stored in the `orderData` and in `qosData`.
- `qosData`, this contains QoS information requested by sending this order. More details about handling QoS information will be discussed in section 4.1.2.

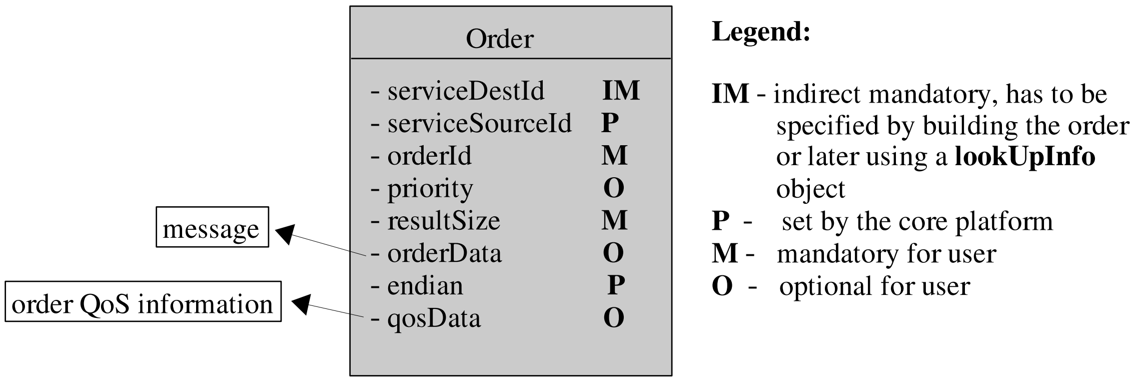


Figure 4.7: Order

3. Filling the order with a message Once the envelope for the message is created, i.e. the order, the user has to fill it out with a message and eventually to attach some quality of service demands for processing the order. The `orderData` and `qosData` members are of type `ByteArray`. This class contains an array of bytes and has functions to write and read data into it. The service which sends the order uses write functions for writting the message into the order, and the service which gets the order will use read functions for reading the message. Our middleware defines a series of types which can be written and read from a `ByteArray` object, like: `OSAShort`, `OSAChar`, etc. All the OSA+ types and their mapping to the C and Java programming languages can be seen in table 4.1.

It can be observed from the table that the middleware introduces an overhead of at least one byte for each data type compared with the native programming language type. This is due to an additional byte which defines the middleware data type. Because messages represent the main mean of communication in our middleware, we decided to introduce this additional byte in order to provide the type safety characteristic for the data transported over messages. In this way no misinterpretation can be made for the message bytes.

One of the aims of our middleware is to be portable over programming languages, and this must be valid for the data types used by the middleware, too. In our conception, a middleware data type is portable over different programming languages, if it maintains the same value on the mapped programming language type. As example, an `OSAUShort` has the same value range when mapped to an `unsigned short` in C and to an `int` in Java. This is different from the CORBA standard, where the user has to take care to use the appropriate CORBA data type that fits the programming language dependent value range when sending a value to an application in a different programming language.

4. Connection-oriented or connectionless communication When a service is willing to send an order to another service, it can choose between two possibilities. It can create a long term connection and send orders over it or it can send the orders directly without establishing any connection. These possibilities are known as: *connection-oriented* or *connectionless communication.*

The connection-oriented communication has the advantage that all resources which are needed for transmitting the jobs will be reserved before the communication starts. This will result in:

OSA+ types mapping											
OSA+				ANSI C				JAVA			
Type	**Min**	**Max**	**Bytes**	**Type**	**Min**	**Max**	**Bytes**	**Type**	**Min**	**Max**	**Bytes**
OSAChar	0	2^{16}-1	3	unsigned short	0	2^{16}-1	2	char	\u0000	\uFFFF	2
OSAShort	-2^{15}	2^{15}-1	3	signed short	-2^{15}	2^{15}-1	2	short	-2^{15}	2^{15}-1	2
OSAUShort	0	2^{16}-1	3	unsigned short	0	2^{16}-1	2	int	-2^{31}	2^{31}-1	4
OSALong	-2^{31}	2^{31}-1	5	signed long	-2^{31}	2^{31}-1	4	int	-2^{31}	2^{31}-1	4
OSAULong	0	2^{32}-1	5	unsigned long	0	2^{32}-1	4	long	-2^{63}	2^{63}-1	8
OSADouble	IEEE754	IEEE754	9	double	IEEE754	IEEE754	8	double	IEEE754	IEEE754	8
OSAByte	*	*	2	unsigned char	*	*	1	byte	*	*	1
OSAWord	*	*	3	unsigned short	*	*	2	short	-	-	2
OSADWord	*	*	5	unsigned long	*	*	4	int	-	-	4
OSABool	0	1	2	unsigned char	0	1	1	boolean	TRUE	FALSE	-
OSAString	string of characters		3+2*n	char*	\0 terminated string		n+1	String	string of characters		2*n
OSAByteArray	array of bytes		3+n	unsigned char*	*	*	n	byte[]	*	*	n

Legend:	
*	: not applicable or relevant
-	: not present
Bytes column	: nb. of bytes when the type is serialized

Table 4.1: OSA+ types

- efficient communication, no time is spent anymore during the communication for reserving and freeing resources

- safe communication, once the resources are reserved there is no more the danger to be accidently out of resources. This situation can happen in connectionless communication where resources are allocated each time a communication is realized.

These advantages are important especially for real-time processing where the guaranteeing of resource reservation is made more easier when using connection-oriented communication.

The core platform has the `createConnection` function which returns to the user a `Connection` object that can be further used to send orders. As parameters for this function, the user has to give a template order, a number of orders that can be active over the connection and information to locate the destination service. The template order and the number of orders to be reserved indicate to the platform the memory needed by the user for the communication. This information is used also to allocate resources in the case that the destination service is on a remote platform.

The connectionless communication is suited when there are no strict timing demands and only sporadic orders will be sent to the service.

5. Sending the order and getting the result The communication can be synchronous or asynchronous. The synchronous communication is realized by the following functions: `sendOrder, waitOrder, sendResult` and `waitResult`. The asynchronous communication is realized through the functions: `sendOrder, existOrder, sendResult` and `existResult`. The `exist` functions check if there is something available to be processed for the service: order or result. If something is available, it can be requested from the platform through a `wait` function. Figure 4.8 depicts the methods used when communicating with jobs.

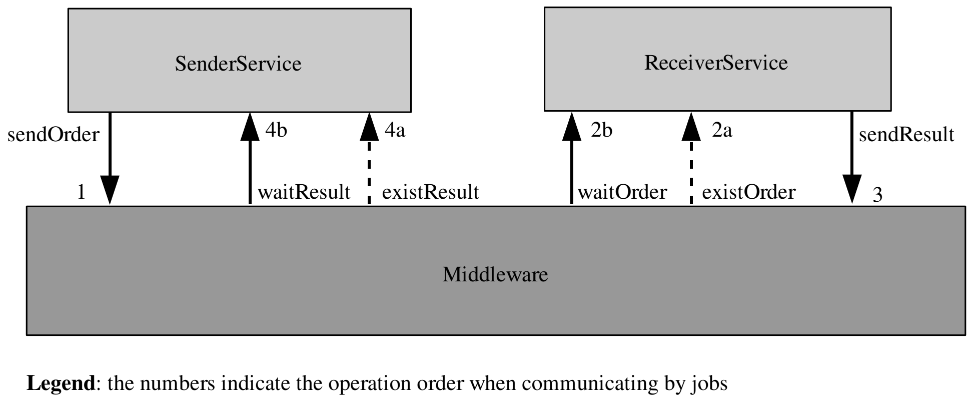


Figure 4.8: Synchronous/asynchronous communication with jobs

Note, there is a major difference between message oriented middleware like e.g. JMS and the service oriented OSA+ architecture. OSA+ not only delivers orders and results (jobs), but it is as well responsible for the execution of these jobs. This affects for e.g. that the middleware changes the priority of a service when this service receives an order with a different priority. Therefore, the OSA+ middleware performs job and service scheduling.

If the services are on the local platform, the core is able to realize the communication. The modality how this is accomplished will be described further. But, when a service is on a remote platform, the core uses the Communication Services to send the order to the remote service, again following the uniform service approach. This services will be described in section 4.1.3.

Now we will describe the connection-oriented communication realized by the core platform. The connectionless communication is realized also through connections which are hidden for the user. In this case, the platform creates the necessary connections to realize the communication.

As already mentioned, the connection-oriented communication reserves the resources once a connection is created. These resources are, for the local communication, mainly the jobs that can be simultaneously active and are not yet processed. These jobs will be reserved inside the `Connection` object and their amount is specified by the user when the `Connection` is created.

Each service has two heaps sorted on priorities: one for unprocessed orders - `availableOrders`, and one for unprocessed results - `availableResults`. The core platform realizes the local communication between two services through the `Connection` object created by the service which initiated the communication - the source service of the orders. The steps which are made by the core to realize the communication, are shown in figure 4.9 and described here:

1. the core gets a free (unused) job from the `Connection`

2. the order is copied from the source service to the `Connection` object. To realize a faster communication, we distinguish here two cases:

   a) the content of the order is really copied byte by byte to the `Connection` object

   b) the `Connection` will use only a pointer to the order which has to be sent. This method is faster, because it uses only pointers, but has the drawback that the user must not modify the content of the order until it is processed by the destination service. This requires user collaboration. This case is used inside the middleware whenever it is possible.

3. the core adds the orders to the priority sorted order heap of the destination service and acknowledges the destination service that a new order has arrived

4. the order is processed by the destination service according to the order's priority and other QoS requests (see section 4.1.2 and the

corresponding result will be written by the service directly inside the `Connection`

5. the core adds the result to the priority sorted result heap of the order source service, and acknowledges the order source service that a result arrived.

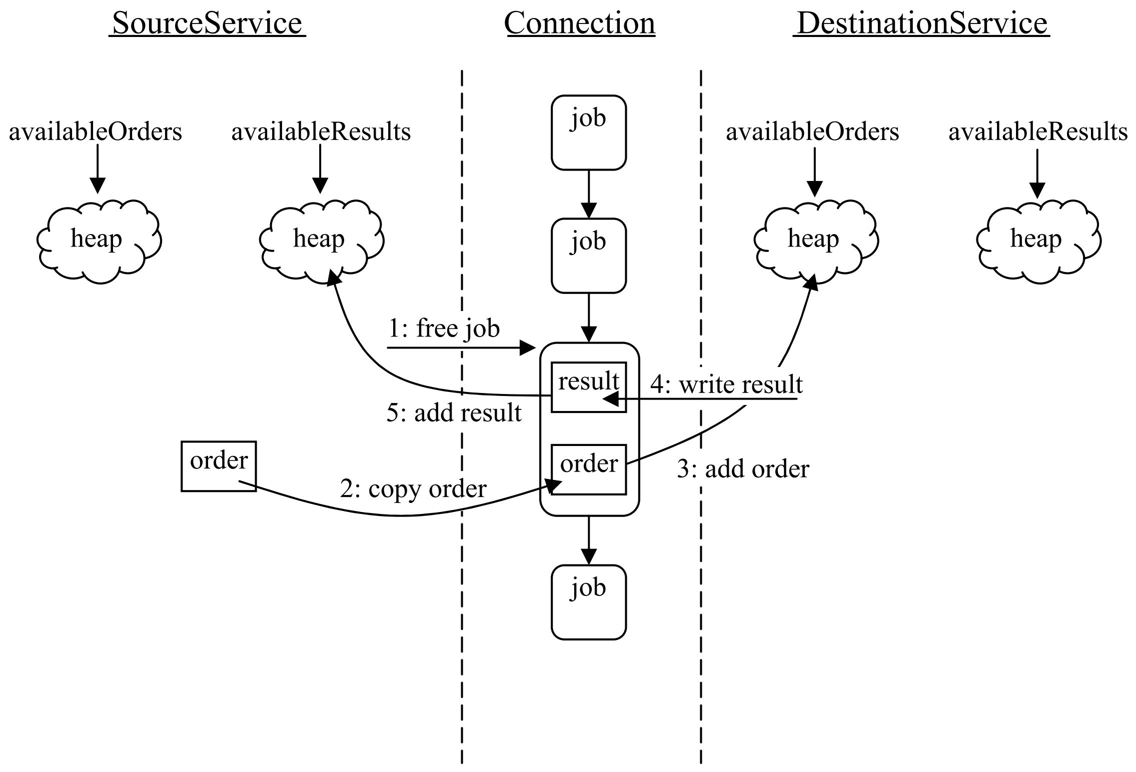


Figure 4.9: Local connection-oriented communication

Procedural interface communication

The procedural interface communication is introduced in our middleware to obtain a faster communication than the communication with jobs. It is realized by normal method call, and the data is transmitted over the method parameters. Using this mechanism to communicate to a service introduces the following consequencies:

- avoiding of the job communication mechanism which results in a faster communication. In this case the priority used to execute the service functionality is the caller service priority.

- the communication will be synchronous, the service which uses the procedural interface of another service has to wait until the called function finishes to execute

- the communication can be only local, on the same platform

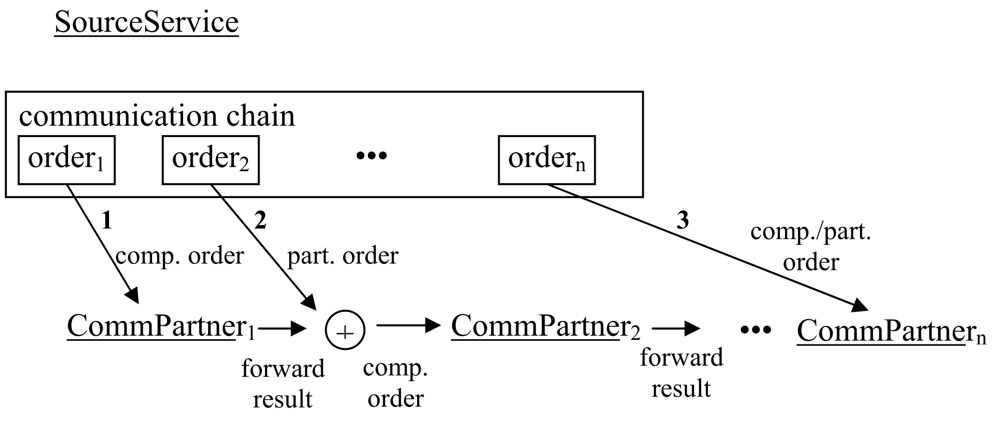
- in this case a tight coupling is made between both services.

The procedural interface is optional, so a service can ommit this. In fact, the procedural interface communication is used by our middleware to access frequently used functionalities of the basic services (e.g. the Process Service and the Communication Services) thus having an important impact to the overall speed and real-time characteristic of the middleware. This issue is addressed in more detail in section 4.2.2.

Chain communication mechanism

In the domain of the real-time systems, the application demands regarding resources (CPU, communication needs, bounded time for memory usage) are always important and considered. In order to fulfill these demands, any additional information can be used to optimize the resource allocation. Therefore, many applications, especially from the real-time domain, can provide information in advance about a sequence of communications that will be established with other applications. This information can be used to minimize the overall resource usage, e.g. fewer inter-platform connections will be established, which will save in the same time: memory, CPU and communication time. Once the middleware has this information, it can optimize the flow of communication data and improve the overall performance of the application. This can produce better results regarding the real-time constraints imposed by the application.

The chain communication mechanism is an extension of the basic job communication mechanism. Using the basic communication mechansim, our middleware allows one-to-one communication through jobs. Using the chain communication mechanism, a service can build a sequence of one-to-one basic communications through jobs, which we call a *chain*. The chain describes the communication partners of the service - the *chain nodes*, and the orders to be transmitted to them. Furthermore, part of the results from a chain node can be supplied as input data for the next node in the chain, see figure 4.10. Regarding the chain concept, we define a *multijob* as a sequence of jobs. A multijob contains a *multiorder* - the sequence of orders to be sent to the communication partners, and the results obtained by processing these orders.



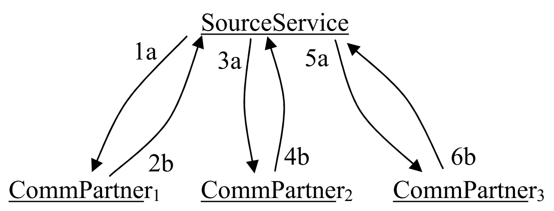
Legend:
- the numbers represent the chain communication order
- CommPartner$_i$: communication partner service
- comp. : complete, part. : partial

Figure 4.10: Communication chain

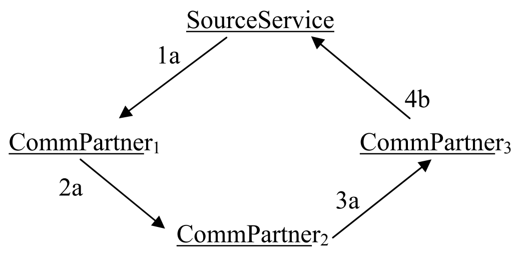
Analyzing the chain, the Communication Service knows the platforms involved in communication and can obtain the values for the communication throughput (the amount of data transported between two platforms in a given time period) between all these platforms. With this information the Communication Service can choose the optimum solution for each node communication. It can choose between the following choices, figure 4.11:

- standard mechanism of establishing one-to-one connections to the other communication partners
- establishing fewer connections and forwarding the data between the

nodes of the chain. In this case the Communication Service can apply an algorithm from graph theory which maximizes the data flow between the nodes of the chain.



(a) Establishing a connection for every communication



(b) Establishing a connection between successive partners and forwarding the data

Legend:
- the numbers represent chain communication order
- a: sending an order, b: sending a result
- $CommPartner_i$ **: communication partner service**

Figure 4.11: Optimizing communication with chains

For supporting the chain communication mechanism, the `Order` class contains the following members:

- `mOrderPolicy` defines different policies that can be applied inside the chain, when processing this order. These policies are:
  - `MO_POLICY_BREAK_AT_ERROR`, if the destination service reports an error when processing this order, the processing of the chain must be interrupted
  - `MO_POLICY_CONTINUE_AT_ERROR`, if the destination service reports an error when processing this order, continue to process the chain
  - `MO_POLICY_IGNORE_RESULT`, do not transmit back the result produced when processing this order

  - `MO_POLICY_BREAK_SEQUENTIALITY`, if there is no data to be forwarded between chain nodes, then make a multicast and send the orders in the same time to the corresponding services

- `fwdTable`, this member contains information about which output data from the result obtained by processing this order, should consist an input data for the next order in the chain.

Inplementing the chain communication concept, two other classes were added: `MultiOrder` and `MultiJob` which are linked lists of `Order` respectively, `Job` objects.

Chain processing As in the basic communication mechanism, there are two steps when using the chain communication:

1. establishing connections and reserving resources
2. sending data over the connections.

The most complex step is the first one, especially when the chain is spread over different platforms. This step is realized by the HLCService together with the corresponding low level communication services. The HLCService will do the following (see also figure 4.12):

1. analyze the whole chain and group the communication partners which reside on the same platform
2. analyze the inter-dependencies between successive communication partners. If information will be forwarded we have an inter-dependency.
3. analyze the communication performance of the inter-platform connections. This information is received from the low level communication service.
4. create on the remote platforms chains containing the groups created at step 1 and create inter-platform connections. These connections are created according with the information obtained at steps 2 and 3.

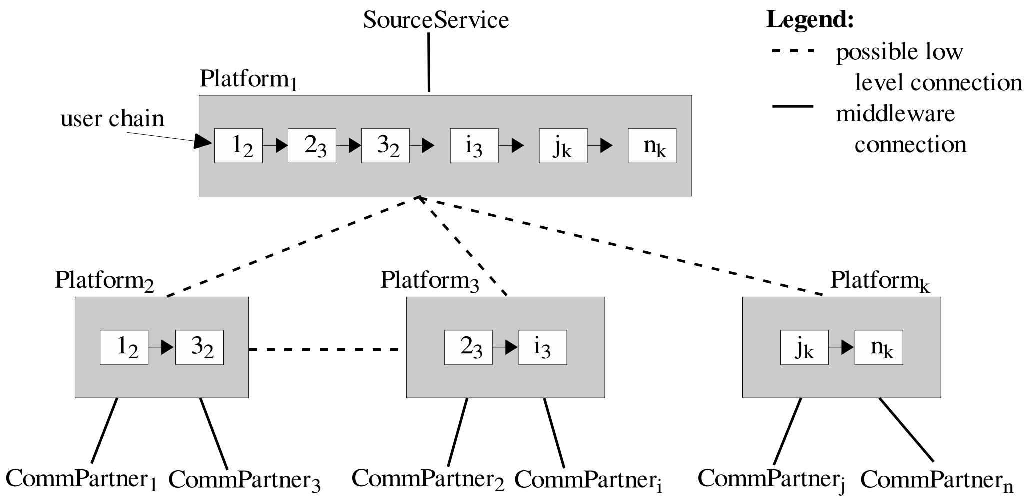


Figure 4.12: Chain processing

Once that connections are established and the user provided chain is split in smaller chains on each remote platform, the OSA+ Core is able to process the chain. The processing of the chain will happen sequentially on each platform.

One way to extend the chain communication is to allow parallel processing. If between two consecutive services there is no data to be forwarded, then the platform could send the orders in parallel to both services.

QoS support

In order to present the real-time features of the underlying environment (hardware, operating and communication systems), OSA+ uses QoS information. The user can request desired QoS properties for its actions: deadlines, priorities, minimum band width for an Internet connection, etc. Our middleware distinguishes between two kinds of QoS:

1. *general QoS* which are globally recognized by the middleware, like: deadlines, band width, etc.

2. *service specific QoS* which can be defined by the service and can be directly requested by the other services. The platform will not interfere when processing this QoS information.

The QoS information is divided in classes: time related, communication related, etc. Our middleware provides a simple mechanism to realize and extend the general QoS processed by the platform. Each class of QoS information is managed by a service. This service will register a `QoSHandler` to the platform, which will be executed every time an order contains QoS information belonging to the class. As an example, the Event Service is responsible for initiating and monitoring all time triggered actions. These are mainly release times (earliest start of execution, earliest start of order delivery, etc.) and deadlines (latest end of execution, latest end of order delivery, etc.). The service will register to the platform a `QoSHandler` using the following core function:

```
addQoSHandler(RTEventClass, RTEventQoSHandler)
```

Where `RTEventClass` is an integer representing the QoS class of parameters regarding real-time events and `RTEventQoSHandler` is the handler implemented by the Event Service. With this mechanism, new classes of QoS information can be easily added to the platform.

The core platform differentiates two points in time when processing the QoS information:

1. initialization time, when resources are reserved. As example for the communication related QoS class, this is the time when connections are created.

2. execution time, when orders are exchanged between services. As example for the time related QoS class, this is the time when deadlines for the orders are set and monitored.

The `QoSHandler` interface provides two methods for supporting these QoS processing times: `initializationTime` and `executionTime`. They are executed by the core platform at the corresponding processing time.

For the moment, our middleware considers only three QoS information classes: time related, communication related and task scheduling related. These are managed by the Event Service, Communication Services, respectively Process Service.

The service specific QoS information will be delivered by the platform along with the order. It depends of the service how this will be processed.

An example of service specific QoS is a database service which might define: database access priority or rate of read or write transactions per second.

As the efficiency is a key factor for OSA+, the QoS information is organized in classes of simple (key-value) pairs, where the key is a unique code for a QoS parameter and value is the corresponding value for it. As example an order can contain as QoS parameters the key-value pair (10 - 100), where 10 represents the QoS code parameter for the order execution deadline and 100 represents the time in milliseconds. This mode to represent QoS information is very simple, efficient and flexible. On top of the actual representation of QoS parameters, a layer can be easily added which describes them in more general formats, e.g. the XML format. This layer can be added on more powerful platforms with enough resources to support it.

After discussing about the main aspects concerning the OSA+ core platform, we will further describe the basic services.

4.1.3 Basic services

The basic services allow the middleware to use the underlying operating system and/or hardware. They adapt and scale OSA+ to the heterogeneous environments. The middleware is using a fixed interface for each specific task which realizes the adaptation. These tasks are:

- using real-time memory management
- using processes and threads
- using communication hardware and protocols
- using timer events

In case of using OSA+ on an embedded system whith less resources, the basic services can contain only the basic functionalities which are needed for this environment.

Process Service

The Process Service allows OSA+ to use lightweight (threads) and/or heavyweight tasks (processes) to run light- and/or heavyweight services.

The Process Service adapts to different process models and scheduling policies and use them for scheduling the platform lightwieght and heavyweight services.

OSA+ requires that a Process Service can create, start, terminate and destroy tasks. Additionally, it can also take advantage of the possibility to change priorities and deadlines or to suspend and resume tasks. Especially the priority changing feature is mandatory for inheriting the order priority to the service which executes the order. More about this will be discussed in the real-time section 4.4. The Process Service is a procedural service and has to be registered on the platform before registering other lightweight or heavyweight services.

Currently the service adapts OSA+ to three processing environments:

1. Komodo real-time micro-controller. Here the service can use the FPP and GP real-time scheduling schemes.

2. real-time specifications for java. The service is able to use the `javax.realtime` package which gives the possibility to manage real-time threads.

3. standard java. Only priorities are available to schedule the threads.

Communication Services

Communication services allow the platform to use communication hardware and protocols to communicate with other platforms. This not only includes sending and receiving data, but also includes routing data to services.

From the communication's point of view, the middleware has a layered architecture that is built by four layers (Figure 4.13):

- Job oriented communication between services (*General Service Layer, GSL*)

- Job oriented communication between platforms (*Platform Layer, PL*)

- Simple data oriented communication across network boundaries (*Protocol Spanning Layer, PRSL*)

- Simple data oriented communication inside of networks (*Protocol Layer, PRL*)

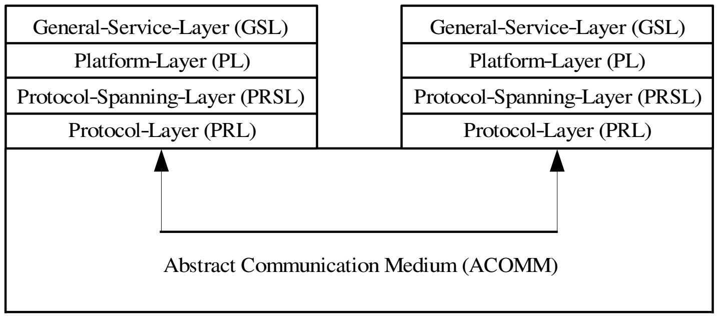

General-Service-Layer (GSL)
Platform-Layer (PL)
Protocol-Spanning-Layer (PRSL)
Protocol-Layer (PRL)
General-Service-Layer (GSL)
Platform-Layer (PL)
Protocol-Spanning-Layer (PRSL)
Protocol-Layer (PRL)
Abstract Communication Medium (ACOMM)

Figure 4.13: OSA+ communication layers

The Protocol Layer consists of *Low Level Communication services* (LLCs) that provide a simple communication with well-known protocols such as TCP/IP, CAN or ISDN. To switch these protocols and to route data between services that are located in separate networks, the Protocol Spanning Layer offers a *High Level Communication service* (HLC). This service will send and receive data across the boundaries of networks. To keep the layers lean and to reduce the overhead, a very simple protocol is used. This protocol only contains functions to establish and release connections and to transmit data over them. OSA+ Platforms use the protocol spanning layer to deliver remote orders and results. As the users of the HLC they form the Platform Layer. Finally, the application services are grouped in the General Service Layer.

The LLC services are designed to be synchronous regarding the communication. They are always waiting for remote data or commands received from the HLC Service. In this way, no processor time is spent when no remote communication exists.

TCPIPService The TCPIPService is a low level communication service which is able to use the TCP/IP protocol. The service is using the `java.nio` package appeared in JDK version 1.4. This package provides the following advantages compared with the standard `java.net` package:

- multiplexing/demultiplexing I/O operations. Using the `java.net` package, the programmers would have to deal with multiple socket connections by starting a thread for each connection. Inevitably, they would encounter issues such as operating system limits, deadlocks, or thread safety violations. With the `java.nio` package, multiple simultaneous socket connections can be managed by a single thread. This leads to a scalable approach which saves considerable memory and processor resources.

- unblocking blocked read, write operations. The new read, write blocking operations available in the `java.nio` package can be interrupted by other threads. This feature is used by the TCPIPService when it listens, in a blocking mode, from the network for incoming data or connections. At the same time, the HLCService is willing to use the TCPIPService for executing some actions, e.g. sending an order on a remote platform. To solve this, the TCPIPService overrides the `signalOrder` method of the `Service` class which is executed by the platform when a new order arrives. In this method, the service unblocks itself from the blocking mode in order to act on the received order.

- non-blocking operations. Using blocking operations from `java.net` package, a thread will block on a read or a write until the operation is completely finished. If during a read, data has not completely arrived at the socket, the thread will block on the read operation until all the data is available. Using non-blocking operations, the thread will read whatever amount of data is available and return to perform other tasks.

A similar library as the `java.nio` package, is the NBIO library [43]. This can be used for earlier versions of the JDK standard libraries.

In figure 4.14 the simple protocol between all the communication layers can be seen. It is interesting to mention here the `createRemoteConnection` function of the HLCService which is executed in two steps:

1. first, a low level connection is established, and

2. secondly the information about the resources to be reserved for this connection is sent to the remote platform. This information contains the number and size of active jobs that can exist on this connection.

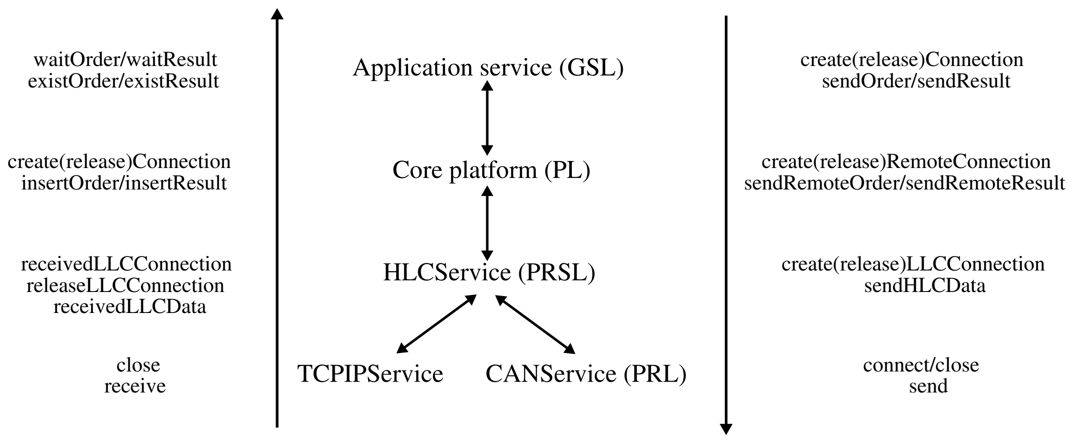


Figure 4.14: Simple protocol between the communication layers

Event Service

The event service allows the platform to use environment based events. These can be:

- hardware based events, e.g. hardware timer events and other interrupts. These events require the Event Service to work directly with the hardware.

- operating system events, e.g. OS timers and interrupts. In this case the Event Service uses the operating system capabilities.

An important event class for real-time are timer events. They give to the platform the possibility to attach different timings to the jobs and to monitor if they are respected. Once that the Event Service is registered on the platform, it will also register a `QoSHandler` which is able to deal with all timing related QoS information. It will initiate the release times of the orders and will monitor their deadlines.

Real-time Memory Service

Contemporary and future real-time systems are more dynamic, so not all resources can be pre-allocated. OSA+ provides support for these dynamic systems by definition of a basic service, the Real-time Memory Service, which can be plugged into the platform. The Real-time Memory Service allows the allocation and freeing of memory under guaranteed timing constraints. The platform can use this functionality to deliver jobs in real-time, for connectionless communication where the needed buffers are not pre-allocated. For real-time operating systems, the Real-time Memory Service can rely on their capabilities to dynamically allocate and free memory. In environments without that support, the approach is to demand in advance an important amount of memory when the service is registered on the platform. In that case, the Real-time Memory Service realizes a predictable memory management itself. This task is challenging especially in the case of embedded systems, where the memory is limited and has to be used as efficient as possible.

ARS Service

The Address Resolution Service is responsible for locating services on the global virtual platform, the middleware level. In a usual configuration an ARS Service maintains location service information for several platforms. This information consists of:

- service identification information: name and version
- service location: the network type used to reach the platform where the service is located and the network address of the remote platform.

An example for this information, which is mapped to the middleware class `LookUpInfo`, will be: {*UserService, 0.7, TCPIP, 127.0.0.1:5000*}. Having this information, the communication services can locate and reach all the services which are maintained by the ARS Service. The core platform uses the ARS Service to provide location transparency to user applications. The user has to give only the name and/or version of the service to communicate, and the core platform will:

- contact the ARS Service and obtain information about the service location
- use location information and the communication services to send orders and results.

In order to use the ARS Service, the user has to configure first the core platform with its location using the core function `configureARS`. Once that the core is configured, each new service which will be registered on the platform will be also made public to the ARS Service. An exception is made for the basic services which are known only on the local platform. However, other implementations of the ARS are possible too. Services can be located as well on a peer-to-peer basis, when the ARS just asks the neighbour platforms if a service is unknown. Broadcasts would be another possibility to locate remote services.

4.2 OSA+ microkernel architecture

One main goal of OSA+ is the realization of a highly scalable architecture, which can easily be adapted to different hardware and software environments. Especially we want to be able to scale down to embedded systems. To reach this goal, a microkernel architecture well known from operating systems is used. The OSA+ platform consists of a very small core platform, which offers a basic functionality. This core platform can be extended and adapted to the environment through the basic services. And finally user services can be added on the platform. If we compare OSA+ to a microkernel operating systems, we can see that:

- the microkernel role is played by the core platform
- the operating system services are in our case the basic services
- the inter-process communication mechanism corresponds to basic job communication mechanism in our middleware, and
- the user applications correspond for our middleware to the user services.

In the following section the motivation of choosing to use a microkernel concept for our middleware will be discussed.

4.2.1 Microkernel principals and ideas

The microkernel concept will be discussed in the context of the operating systems first, because it is more easily and intuitive to be presented on that level instead of using only abstract concepts. Once we map the microkernel concepts to our middleware, the principles and ideas can be easily seen for the middleware, too.

In operating systems, we can distinguish the kernel approach and the microkernel approach. The kernel approach included all the OS services in the kernel. The main idea is, that all the services are safe and absolutely trustworthy. The advantage for this approach is that there is no restriction to implement the OS services which makes the kernel approches to obtain a good performance. The disadvantage is that the OS does not control the execution of these services, so if an OS service is failing, the entire OS can fail.

The main idea for the microkernel approach is to keep the kernel minimal. Therefore, the OS services are kept outside the kernel. This reduces the kernel's size and protects the OS services from each other and the users. Now, OS services can be introduced that are not necessarily totally trusted by the other services. The idea for this approach is to separate the OS services by each other. This is realized for OSes by having different address spaces. Since they are isolated and protected by each other, the microkernel has to supply user and system services with a cross-address-space communication facility which is usually called inter-process communication (IPC).
The advantages of using the microkernel approach are:

- first of all *flexibility and extensibility*. The system can be easily adapted to new hardware or software. Only selected services need to be modified or added to the system. The modifications can be made and tested on line.
- *coexistence of different APIs*, multiple services which provide the same facility can exist in the OS.

- *isolation*, OS services malfunctions are isolated as normal applications. They can be shut down and replaced.
- modular system structure
- easy maintenance, less error prone

After having presented the microkernel principles, it can be seen that these principals are extremely close to our goals:

- focus for embedded systems - the central idea is minimality
- as middleware, scalability and adaptation to different environments - flexibility and extensibility are inherent advantages of the microkernel concept which permit to microkernel OSes to easily adapt to new hardware or software.

Therefore, we decided to investigate this concept for building our middleware and to work further for the real-time aspect of the middleware.

4.2.2 Avoiding the microkernel drawbacks

The microkernel concept seems to be a perfect approach for reaching the above mentioned goals. However, the disadvantages must also be considered. The strict separation of the services introduces two drawbacks when using the microkernel approach:

1. less performance, the services can not share common information but have to use the IPC to exchange this information
2. increased communication overhead, by raising the modularization of the entire system, the communication between the services is raised too.

To overcome these drawbacks in our middleware approach, we introduced the following techniques:

- the basic services can share common information with the core platform (and only with the core platform) on a read-only basis. They are not allowed to modify this information. Otherwise, a faulty service can affect the entire platform.

- the procedural interface communication, see section 4.1.2. A procedural interface of a service represents the public access points for the other services. These access points can be used natively in the implementation programming language, with the restrictions already discussed. The main advantage using this technique is to reduce the communication time by a more direct connection. The amount of communication remains the same.

Figure 4.15 shows how the procedural interface communication avoids the job communication which is more time consuming (message based communication).

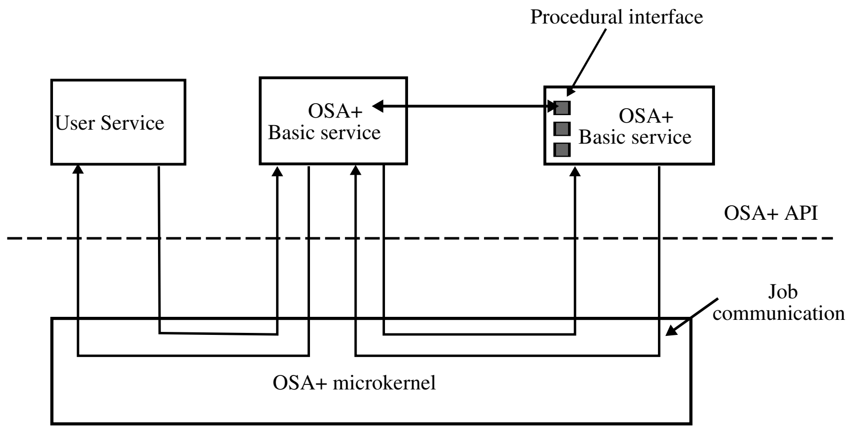


Figure 4.15: Overcoming microkernel performance loose through procedural interface communication

4.3 Scaling OSA+ by the microkernel and the basic services

By the microkernel approach and the basic services, the OSA+ middleware can be scaled to different environments and application requirements. Here are some examples on how OSA+ can be configured:

- the microkernel can be used standalone. Since it contains no operating system or communication system dependant parts, it can only provide the use of local procedural services with a single control flow. This results in the lowest possible functionality and memory footprint.

- adding the Process Service establishes the connection to the operating system. Thus, local procedural, lightweight and heavyweight services can be used.

- combining the microkernel with the Process Service and the Communication Services allows local and global services to be executed.

- combining the microkernel with the Event Service enables local procedural time-triggered services.

- etc.

As it can be seen, the full microkernel approach in combination with a purely service oriented architecture provides a great flexibility. Furthermore, different types and qualities of basic services can be used without the need to change the application or the core platform. As mentioned in the previous sections, different implementations for e.g. the ARS, the Real-time Memory Service or the Communication Services are possible. Finally, extension services can be used to increase the core platform functionality, e.g. by encryption or error logging.

4.4 OSA+ and real-time

The OSA+ middleware can guarantee real-time performance of its operations only if all the underlying layers are able to carry out their functions in real-time (see figure 1.2). First of all, the hardware must guarantee worst case execution times for its operations. Simple microcontrollers or processors without caches and branch prediction are more suitable than highly speculative complex microprocessors. Second, the operating system used should be a real-time one. It should support at least one of the common real-time scheduling policies like fixed priority preemptive or earliest deadline first (EDF). Another possibility is to omit the operating system layer at all. On a small microcontroller, the Process Service described in section 4.1.3 can operate directly on the hardware and exploit the real-time features of the processor. This is very important in case of the embedded systems. Real-time memory management is a useful feature which can be used by

OSA+ as it was discussed in section 4.1.3. In case of no real-time memory management is available, the operating system should support memory locking. And third, the physical communication medium and the protocols used should guarantee worst case execution times for the transmission of messages between the nodes of the distributed system.

These assumptions regarding the environment are necessary conditions in order to provide real-time to the user applications. Furthermore, the middleware layer itself, OSA+, has to maintain the real-time features of the underlying environment and to present them to the user applications. In the following sections the modalities chosen to maintain and present these features to the user applications are described.

4.4.1 Separate functionality

An important aspect when designing real-time applications is the resource allocation issue. This affects the predictability of the application. In most of the cases resource allocation implies complex techniques which do not respect strict timing constraints, e.g. complex algorithms for memory allocation, parameter negotiation for protocol communication, task creation implies memory allocation, etc. Therefore, our middleware makes a strict separation between:

- initialization functions, and
- operational functions.

Initialization functions are responsible for resource allocation. They can be unpredictabile and might not obey any timing constraints. Some examples of initialization functions of our middleware are:

- `createConnection` and `releaseConnection`. The `createConnection` function allocates all the operating system and middleware needed resources regarding communication, e.g. sockets, buffers, jobs.
- `lookUpService`. This function is used to locate a service before starting the real communication. It realizes a search operation (searches a service having the name and/or version values) whith no constant time behaviour.

- `registerService` and `unregisterService`. Calling the `registerService` function, the core platform has first to add the service to the Service Repository, an operation which might not be performed in constant time. Then, in case the service is lightweight or heavyweight the core will use the Process Service to create an operating system thread or process to host the service. Again, this operation implies memory allocation and might not be necessarily executed in constant time.

Operational functions use the resources allocated by the initialization functions and must offer a constant and bound time behaviour. As middleware function examples, we can mention:

- `sendOrder` and `sendResult`. The functions use the resources already allocated by the `createConnection` function. A special issue here is the processing of the job priorities. The operations to maintain the job queues sorted on priorities have to perform in a bound time. This aspect is discussed further in section 4.4.3.

- `waitOrder`, `waitResult`. These functions have to fetch in bound time the job with the highest order from the job priority queue. More about this is described in section 4.4.3.

- `existOrder` and `existResult`. These functions simply check if the job priority queue is not empty. It is not very difficult to realize this in bound time.

Techniques for realizing predictablity behaviour

In order to maintain the determinism and predictability of the underlying environment, a series of principles and implementation techniques must be strictly applied when designing the middleware. In OSA+ these are represented by:

- separation between initialization and operational functions. This principle has been already explained in the previous section.

- using priorities: fixed or time-based priorites. They will be explained in sections 4.4.3 and 4.4.4.

- choosing algorithms and data structures which have a bound time and a good WCET(worst case execution time). We are using priority heaps (see 4.4.3) and general trees (see the description of bucket containers below).

- avoiding search operations for the operational functions

- sacrifice high-level programming techniques for efficiency. Some cases to mention here are:

  - using list pointers instead of iterators. To obtain an iterator implies in most of the cases the allocation of a new iterator object even if this is not obvious for the programmer. The allocation introduces unbounded performance overhead.
  - reusing objects, which again avoids allocations.

- and, realize a close loop between development and evaluation. This can show the weak points of the performance from already earlier stages of development.

Applying these techniques becomes more difficult especially when the middleware is confronted with additional requirements, like:

- modularization, which means in OSA+ strict separation between the basic services of the middleware basic services. This introduces more communication overhead, but the procedural interface communication introduced by us can overcome this problem.

- strict restrictions regarding memory consumption, OSA+ is focusing on embedded systems. To cope with these restrictions, OSA+ uses a number of techniques described in the Noble's book [44], like:

  - "small interfaces" [40] and "strong design" [35] principles, which say that an interface should present only the minimum data and behavior to its clients.
  - hooks, which dynamically modify the behavior of the core platform. This is the case of `QoSHandler` objects, which change dynamically the behavior of the platform when treating QoS information.

- packages. A large program with lots of optional pieces can be split in packages and loaded when there are needed. The micro-kernel architecture concept uses this principle. The packages in our case are the services.
- sharing. Multiple copies of the same data can be avoided when the information is shared everywhere it is needed. We apply this principle when the core platform shares information with the basic service. However, this principle has to be applied carefully due to security and fail safe reasons. Thus, the shared data is read-only for the basic services.
- multiple representations. To support several different implementations of an object, each implementation must satisfy a common interface. For example in OSA+, the HLC Service can have different implementations:
    - a basic one, with a small footprint, which is able only to route messages to the low level communication services
    - an enhanced version which supports the common basic functionalities and other extra features like: applying special algorithms to find the optimum solution when establishing low level communication connections in case of using the chain communication mechanism (see figure 4.11).

An example for a data structure with a good WCET used in OSA+ is the Bucket Container. The Bucket Container is a container (data structure) that maintains objects in buckets. A bucket is represented by a vector of pointers to the objects managed by the container, see figure 4.16. This structure represents a special type of general incomplete tree, where each non leaf node has exactly *bs* siblings (*bs* is the bucket size) and the object pointers are stored only on the tree leaves. Additionally, the structure contains a linked list with buckets which have at least one free slot for storing an object. This list is used when adding a new object to the container. From now on, objects will be used for convenience in place of pointers to objects.
The ideas for building such a structure are:

- using the memory space in an optimal way when managing a variant number of objects. This is e.g. needed when managing connections or services registered on the platform. A simple linked list would introduce a lot of overhead. A simple array lacks the necessary dynamics. The tree-based Bucket Container is able to manage a number of $n = bs^{tl}$ objects, where tl is the number of tree levels. At the beginning, the container will be able to store objects in a preallocated number of buckets. Once that these buckets are filled with objects, new buckets are created. In this way, the storage capacity can be exponentially extended by keeping the overhead low at the same time.

- the operations to manage objects have low bound complexity values:

  - `add` is adding an object to the container and returns an ID. The operation complexity is $max(O(bs), O(1)+O(al(log_{bs}n)))$, if exists already a free slot among the existing buckets, respectively a new leaf bucket has to be created. $al(m)$ is the allocation time for m buckets.
  - `get`, returns the reference of an object knowing its ID. The complexity of the operation is $O(log_{bs}n + bs)$.
  - `remove`, removes the object with the specified ID from the container. The complexity is: $O(log_{bs}n + bs)$.

4.4.2 QoS information

OSA+ allows every service to provide *quality of service* (QoS) information for the platform and all other services which want to use it. QoS parameters depend highly on the individual service, so they belong to the service specification. However, for the middleware basic and the extension services all QoS parameters are predifined by the OSA+ middleware and they can be used by the user. These QoS parameters are important because they reflect the characteristics of the underlying hardware and operating system. Therefore, the real-time characteristic of the middleware depends upon them.
The QoS parameters predifined by the OSA+ middleware are:

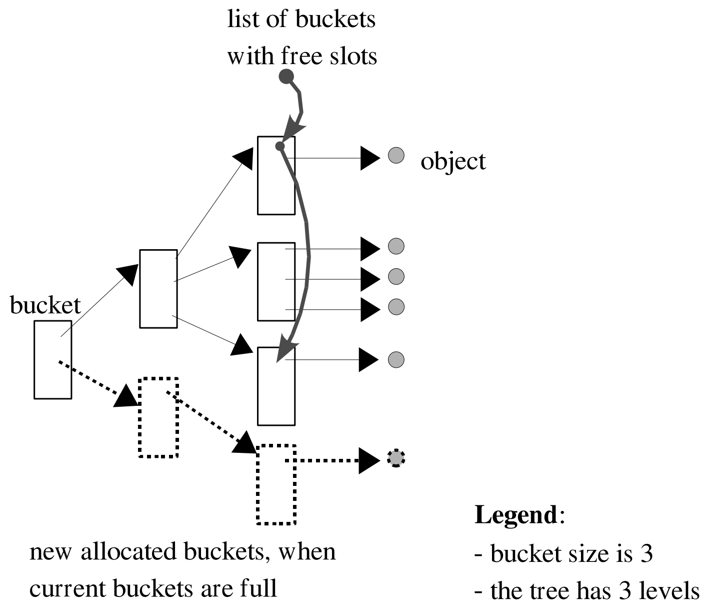


Figure 4.16: Bucket container structure

- for low level communication services:
  - RT, if a predictibale communication is possible at all
  - bandwidth, the amount of data that can be transferred on a connection in a defined amount of time
  - priority, if the connection allows priority based communication
  - send/receive buffer sizes
  - timeout, if a timeout value can be set for the blocking operations: accept, read
- Process Service:
  - the process/thread stack size
  - the available RT scheduling schemes
  - the RT scheduling scheme specific parameters: priority, deadline, processor percentage utilization, etc.
  - the possibility to modify task scheduling parameters during runtime

- Event Service:
  - deadlines set for order processing

Using the QoS parameters the middleware core and the application services can evaluate the real-time capabilities based on the current environment (operating system, communication system, etc.). Here are some examples:

- if the Process Service offers no real-time scheduling scheme (non RT-OS present), no real-time operations are possible at all.
- if RT scheduling schemes are available but the low level communication services report no real-time features (bandwidth, priorities, etc.), only local jobs can be processed in real-time.
- if RT scheduling schemes are available and the low level communication services report real-time features, e.g. priority based communication, remote jobs can be processed in real-time too, because priorities can be preserved end-to-end from on platform to the other thus avoiding priority inversions.

4.4.3 Job scheduling

The main communication mechanism in OSA+ is realized through jobs. In order to have a predictibale communication the core platform introduces priorities for jobs. Each service processes the arrived jobs in the priority order. The core platform uses priority quees to maintain the service jobs which are waiting to be processed, see 4.4.3. Moreover, the core platform will modify the priority of the job's destination service according with the priority of the job. This action is performed when the job is picked up by the service and is possible only if the Process Service realizes a priority based task scheduling.

Regarding priorities, the core platform realizes a priority inheritance mechansim in order to avoid priority inversion for job processing. Priority inversion can happen when a service is curently processing a job with a low priority and a high priority job is waiting to be processed in the priority queue.

So, the core platform is able only to realize a priority based scheduling of the jobs.

When the user needs a time based scheduling, the Event Serivce has to be plugged to the platform. The Event Service will manage all the jobs which contain time related QoS information: release times and deadlines for job processing. For the release times, the Event Service will:

1. send the job to the job's destination service

2. modify the deadline for the job's destination service according to the deadline of the job. In this way, the job scheduling parameters are inherited by the service which should process the job. This can be realized only if the Process Service supports modification of scheduling parameters for the tasks at run-time.

Priority queues

The jobs are scheduled by the core platform using priority queues. These queues are used by the core when sending orders and results. Because these operations are categorized by the middleware in operational operations (see 4.4.1) and are determinant for the predictability of the middleware, they have to provide bound time for the enqueue and dequeue operations. Therefore, we chose to implement them using binary heaps.

A *binary heap* is a heap-ordered binary tree which has a very special shape called a complete tree. The order in the heap is given by the rule: each node value of the tree is greater than the values of its siblings. As a result of its special shape, a binary heap can be implemented using an array as the underlying basic data structure. Thus, the implementation is based on array subscript calculations rather than pointer manipulations. And since an array is used, the storage overhead associated with the pointers contained in the nodes of the trees is eliminated.

The enqueue and dequeue operations for the binary heap have $O(log\ n)$ complexities which make them appropriate for using as priority queues.

4.4.4 Service scheduling

Besides job scheduling, our middleware realizes service scheduling, too. As already discussed in section 4.4.3, services are scheduled in OSA+ according to the job scheduling parameters. The priority (or deadline) of a service is set to the priority (or deadline) of the job it is currently processing. Services are scheduled by the core platform through the Process Service which directly adapts to the scheduling scheme of the hardware or operating system. The adaptation of the service priority according to the currently processed job is the default behaviour applied to a registered service on the platform, and is known as `ORDER_PROPAGATED` policy. The user can otherwise change this behaviour and use the `SERVICE_DECLARED` policy. This allows the user to:

- request specific scheduling parameters for the service (e.g. priority)
- not inherit the scheduling parameters from the jobs received by the service.

Note, only the lightweight and heavyweight services can be scheduled because they have their own control flow.

Chapter 5

APPROACH EVALUATION

In the previous chapter we described the principals and realization of our middleware approach. We presented the concepts that drove us toward a solution for the development of applications for distributed real-time and embedded systems. In this chapter, we will evaluate the implementation of these concepts and analyze if they cope with our expectations.

5.1 OSA+ for embedded systems

One of our main objectives is to adapt to embedded systems. Thus, we have to address the following requirements:

- high scalability. Our middleware has to scale down to these systems and maintain only the necessary components.
- minimize the memory footprint. Even if we scale down and maintain the necessary set of functionality, we can only benefit from this functionality if we match the stringent memory requirements. Our software has to fit on a system with low memory resources.
- minimize the performance overhead. If we succeeded to physically adapt our middleware to these embedded systems, we must also consider the performance degradation which our middleware introduces to a distributed application. The performance overhead introduced must not hinder the requirements of the application which uses our middleware, especially in the case of real-time applications.

5.1.1 High scalability

Our middleware is highly scalable due to the pure microkernel architecture and service orientation. These concepts offers two characteristics regarding the scalability:

- platform configurability: the microkernel architecture permits us to select for the middleware only the desired functionalities. These functionalities are realized by the microkernel and additional services.

- fine adjustment: each component selected to be part from the platform, can be further configured and adjusted to the needs of application and environment requirements. For example, the HLC Service can provide basic functionalities, or can implement more complex algorithms in order to realize an optimum communication with chains of orders, see 4.1.2.

5.1.2 Memory footprint

In order to measure the memory footprint for our middleware, we used the Sun java compiler for Linux, version 1.4.2_05. In table 5.1, there are two sets of measurements:

- a normal set, obtained by compiling the source files and removing debug information from the resulted classes

- a minimum set which is obtained by executing an additional tool [37] on the classes generated at the previous step. This tool makes optimizations for code size, like:
  - removes the code which is not used by the application, eg. classes, methods, variables.
  - renames long with short names

As it can be seen from the table, for the minimum version the overall footprint including TCP/IP communication is about 44 kBytes, which is, to our knowledge, the smallest footprint for a fully configurable real-time middleware. Furthermore, e.g. for local sequential applications, only the core and the InitService is necessary leading to a footprint of 29 kBytes.

	Minimum	Normal
OSA+ Core	28628	52705
ProcessService	3316	5084
HLCService	5695	9561
TCPIPService	5992	7573
InitService	880	1046
Total	44511	75969

Table 5.1: OSA+ footprint(in bytes)

5.2 OSA+ and real-time

Our approach aims toward a middleware which maintains the real-time features of the underlying environment. Therefore, the predictability of the middleware has to be evaluated and analyzed.

The main idea when evaluating the real-time features of our middleware, is to obtain an environment which is proved to be real-time. In this way, the results generated by an application using OSA+, are not influenced by the enviornment but only by our middleware. The influence in this case is introduced in terms of predictability and determinism.

5.2.1 Considerations about the test environment

As it can be seen in the picture presented at the beginning of the thesis 1.2, in order to have an environment with real-time features, we choosed two configurations:

1. a microcontroller with real-time features

2. a general processor with a:

    - real-time operating system, and
    - a real-time interpreter, which in our case is represented by a java virtual machine.

From both configurations, we removed the remote communication, because at this moment we do not have implemented a communication service which adapts to a real-time network communication medium, like: CAN Bus [9] or, SCI [27]. The only remote communication which we can do is based on the TCP/IP protocol which does not have support for real-time.

For the first configuration we used the Komodo Simulator which is simulating the Komodo microcontroller. The Komodo project explores the suitability of Java and a multithreaded processor in embedded real-time systems [11]. A multithreaded processor is characterized by the ability to simultaneously execute instructions of different threads within the processor pipeline, which allows an extremely fast context switch. Multithreaded techniques are proposed in processor architecture research to mask latencies of instructions. The Komodo microcontroller has the following features which make it appropriate for real-time applications:

- supports four real-time scheduling schemes:
  - earliest deadline first (EDF), where the thread with the closest deadline gets the highest priority,
  - guaranteed percentage (GP), where each thread gets a guaranteed percentage of the available processor power
  - fixed priority preemptive (FPP), where each thread gets a fixed priority, and
  - Least Laxity First (LLF), where the thread with the lowest laxity to the deadline gets the highest priority.
- has zero latencies for thread context switches. A context switch appears when a different thread is scheduled for execution.
- can execute maximum six real-time threads
- has a java virtual machine implemented in hardware

For the second configuration we use for evaluation as real-time operating systems:

- TimeSys Linux GPL 4.1 based on the Linux kernel 2.4.21
- TimeSys Linux GPL 5.0 based on Linux kernel 2.6.0
- MontaVista 3.1 based on Linux kernel 2.4.20

As comparision we use also a non real-time operating system: RedHat 9.0 based on Linux kernel 2.4.20 [49].

The TimeSys [61] and MontaVista [42] RTOS apply different modifications to the original Linux kernel in order to improve the real-time performance of basic mechanisms, such as:

- improvements to the scheduler and drivers in order to have a preemptible kernel and to improve the responsiveness of user processes.
- high-resolution timers, with a resolution below one millisecond (eg. 100 microseconds or even less depending on the hardware). The Linux kernel provides timers with a 10 milliseconds resoultion.
- mutexes with priority inheritance, so that a higher priority process can continue with the execution as soon as possible.

These RTOSes use the standard Linux API and the POSIX standard. Therefore, existing Linux applications can be simply executted in a real-time operating system without changes.

The java virtual machines used for evaluation were:

- Sun JVM 1.4.2_06 which is a standard java virtual machine, and
- the reference implementation of the real-time specification for java. This is realized by TimeSys and we will refer it as RTSJ. The version used in our evaluations is 1.1. The real-time specification for java enable Java programs to be used for real-time applications, see [23].

The test environment for these two configurations had the following characteristics:

- first configuration, Komodo microcontroller: 5.5 MHz frequency, 2Mb RAM
- second configuration, general purpose processor: Mobile Intel Pentium 4, 1800MHz, 512 MB RAM. All the evaluations were running in console mode, without any other processes on the system except the application. In this way we tried to avoid as much as possible influences from other sources.

For evaluating the predictability of our middleware not only the average performance for a bunch of operations is important, but also the peak performance for a single operation. For hard real-time applications, the worst case execution time of an operation is more important than the overall performance. Thus, we used a simple Client/Server application. There were implemented as lightweight services and their pseudocode looks like this:

```
   Server                                 Client
while (not test_finished) {        test_finished=false;
  waitOrder();                     createConnection(Server);
  measure time after receiving;    order = createOrder(order_size);
  sendResult();                    fill_order_with_data;
}                                  for (1000 times) {
                                     measure time before sending;
                                     sendOrder(order);
                                     waitResult();
                                   }
                                   test_finished = true;
```

We use the basic communication mechanism with order sizes of 32 bytes. On the Komodo we collected 10 samples of this test and on the Pentium 100 samples. Furthermore, we recorded the following computed values: minimum, maximum, average and standard deviation for each test. Moreover, on the Komodo microcontroller we used four different configurations:

- FPP scheduling scheme, no garbage collector, each service had the same priority

- GP scheduling scheme, no garbage collector, Client/Server CPU shares of 60/30 percent

- GP scheduling scheme, no garbage collector, Client/Server CPU shares of 50/50 percent

- GP scheduling scheme, with garbage collector, Client/Server/GC CPU shares of 60/30/10 percent.

The real-time garbage collector of Komodo can be started as an additional thread and, as for the other real-time threads, custom scheduling parameters can be assigned. In this way, the user can control the garbage collector. On a standard java virtual machine, the garbage collector introduces unpredictable behavior for an application.

On the Pentium processor, we tested the Sun JVM in interpreted mode, avoiding the execution of the Just In Time compiler (JIT). The JIT introduces unpredictable behavior, by compiling the java bytecode of the hot spots in an application into machine specific code. This results in longer times when this operation is performed, but the performance is increased for next execution of the same part of the application.

For both configurations we were looking for patterns when sending these 1000 orders. Eventual patterns, like peeks which appear in all configurations at a certain time, can show that our middleware encounters a situation where it doesn't behave predictably.

The time was measured on the Pentium processor by executing the **rdtsc** instruction, which returns a 64 bit quantity indicating the number of processor cycles since the machine was booted. In order to use the **rtdsc** instruction, we used a C code function which was accessed by our Java application through the Java Native Interface (JNI). The JNI interface allows java applications to access libraries written in C or C++. As observed in our tests, the first measurement has every time a greater value than the average. This can be explained by the fact that the java virtual machine is loading the library code used for our fine measurements in its address space. This is confirmed on the Komodo Simulator, where the first measurement does not have any specific pattern. Here, no library is needed and loaded by the microcontroller, but a Komodo API function is used to get the processor cycles since the Komodo Simulator was started. This value still appears on the graphics, but is not considered in statistics.

All time measurements were recorded in memory for each test, and saved in a file when the test was finished. In this way we tried to avoid as much as possible I/O operations to the harddrive. However, when collecting the statistics for the 100 samples, they were saved after completing a sample in a file. This could result in I/O operations in between the tests, which may interleave with our time measurements for the next test.

5.2.2 OSA+ predictability evaluations

Before starting to discuss the evaluations regarding our middleware predictability, a short roadmap is given which hopefully will help the reader to easily follow the ideas behind these evaluations:

1. firstly, we will analyze our approach in a full real-time environment: the Komodo microcontroller. This evaluations set we will refer it as *Full RT* and should show clearly if OSA+ performs in a predictibale manner.

2. then, OSA+ will be studied on top of a real-time JVM - RTSJ and all selected operating systems. We expect to observe here, that the configuration RTOS + RTSJ is comparable with the *Full RT* evaluations set. Furthermore, when combining RTSJ with a non real-time operating system, we expect the real-time java virtual machine can not support the predictability of the application without the help of a RTOS. This evaluations set we will name *RTSJ*.

3. in the third set, we will analyze the predictability of OSA+ on top of a standard java virtual machine. Here we are interested if the real-time operating system can support by itself the predictability of the application when the java virtual machine is not real-time. We should expect here worst results regarding the predictability compared with the *RTSJ* evaluations set. This set we will name it *Sun* evaluation set.

When evaluating the results obtained, we will qualify them using two criteriums:

- the predictability: as uniform the values are, as predictable they are evaluated. A statistical value which indicates the uniformity of the values, is the standard deviation.

- and their value: as small the value is, as better is the performance for executing a single operation.

The Full RT evaluations set

On the Komodo microprocessor we observed that all 10 samples for each configuration produced the same succession of time measurements. This confirm us the high level of predictability for this environment.

The ideal results were obtained when using the FPP scheduling scheme, figure 5.1. Here, all the 1000 measurements made have the same value, 4308 microseconds. This is due to two reasons:

- the Garbage Collector does not influence the application.
- as soon as one service is blocked waiting for an order or result, the other thread gets the processor in a deterministic context switch time, which for Komodo is zero cycles.

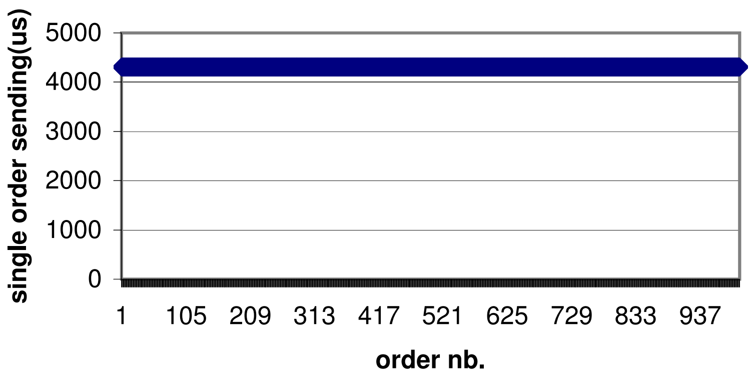


Figure 5.1: OSA+ on Komodo, FPP, no GC, same priority

The values obtained here, 4308 microseconds, are the smallest obtained from all configurations. This is due to the scheduling scheme, which suits best here. This can not be obtained using the GP scheduling scheme, figures 5.2 and 5.4. However, without the garbage collector we still get a uniform distribution, all measurements show the same value.

This set of results confirm us clearly that OSA+ maintains the predictability of the environment. No peaks are appearing in the measured results once that the resources are reserved by the middleware.

From our evaluation, a standard and often configuration for a normal application is missing: OSA+ using the FPP scheduling scheme in combination with the Garbage Collector. This is hapenning due to a weakness of

Komodo: since the garbage collector is realized as a thread, under FPP it will get a fixed priority like the other threads. Possible priority assignments are:

1. GC < Server = Client. In this configuration the GC could not get the processor.

2. GC = Server = Client. The Server and Client reach both a blocking state (waiting for an order, respectively the result) which makes the GC to take and not realease back the processor.

3. GC > Server = Client. Only the GC will run.

In the configurations were the GP scheduling scheme was used, we can make the following considerations regarding the results obtained:

- the GC influences the results obtained: the distribution is no longer uniform and there is a performance degradation (compare figures 5.2 and 5.3). This indicates that the execution of the application threads depends on the actions made by the GC when collecting the unused application objects. The GC may aqcuire some locks on objects used by the application threads, which results in blocking periods for the affected thread. In this situation, a similar problem like in the FPP scheduling scheme appears: priority inversion. The affected thread will wait for the GC to release the locks, but this one has only a small share of the processor. A possible solution here is to realize a priority inheritance mechanism, and to assign to the GC the additional share from the affected thread as long as the lock is still owned by the GC. In this way, the blocking periods will be shorter which will increase the predictability and performance of the application.

- the results depend of the allocated percentages for each thread. For our application, the threads become active in a succesive order and depend one from the other. Moreover, the time needed by each thread to perform its action until the next blocking state is equal. This leads to better results in the configurations with equal shares for each thread than the ones with inequal processor shares. This can be observed in figures 5.4 and 5.2. Nevertheless, it can be observed that in both configurations we have a very good distribution for a real-time application. This confirms once more that OSA+ maintains the predictability of the underlying environment.

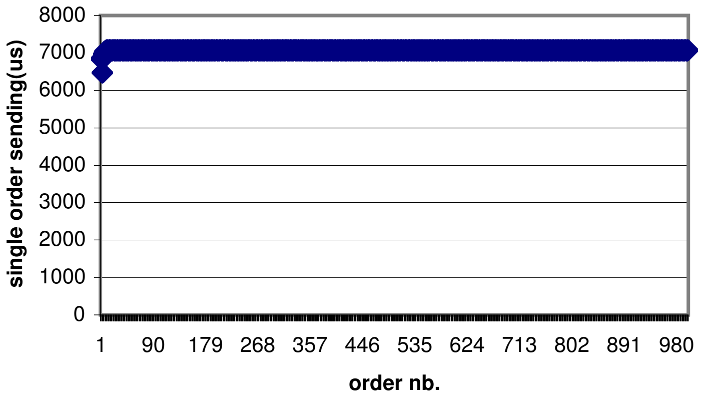


Figure 5.2: OSA+ on Komodo, GP, no GC, 60/30 CPU

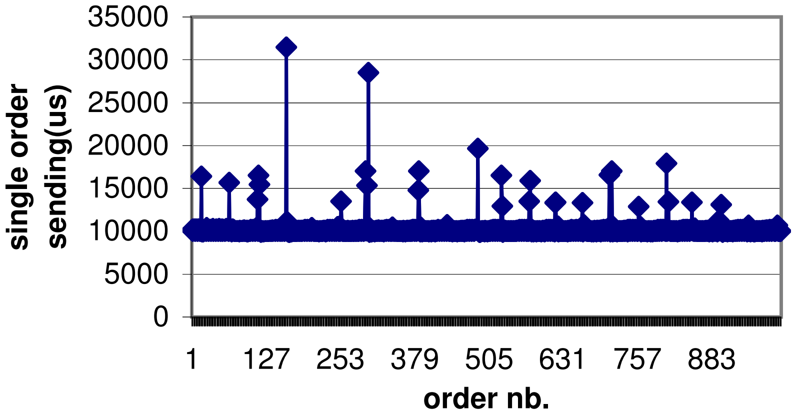


Figure 5.3: OSA+ on Komodo, GP, with GC, 60/30/10 CPU

The previous considerations are summarized in figure 5.5. Here we can see regarding the predictability and performance that:

FPP > GP, no GC, equal CPU shares > GP, no GC, inequal CPU shares > GP, with GC

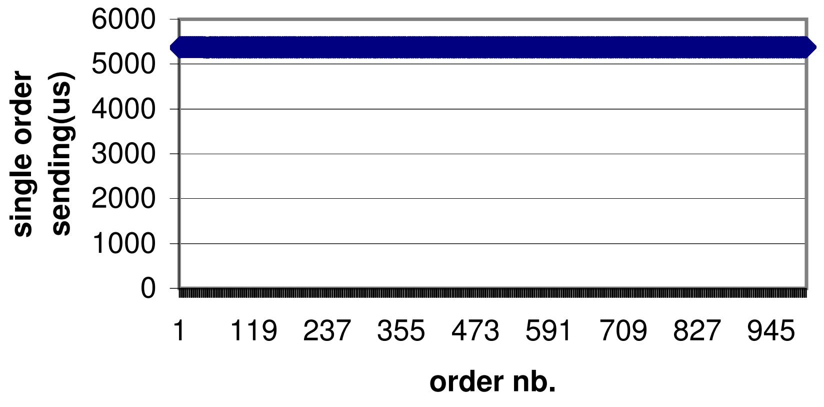
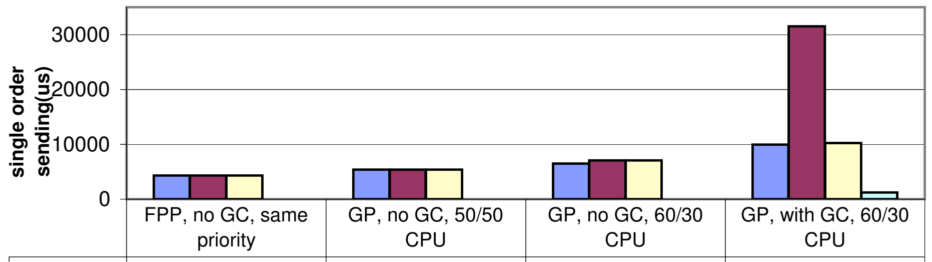


Figure 5.4: OSA+ on Komodo, GP, no GC, 50/50 CPU



	FPP, no GC, same priority	GP, no GC, 50/50 CPU	GP, no GC, 60/30 CPU	GP, with GC, 60/30 CPU
🟦 minimum	4308	5367	6468	9913
🟥 maximum	4308	5376	7072	31467
🟨 average	4308	5373.72	7066.881	10248.111
⬜ stdev	0	2.95864834	24.0663009	1226.389314

Figure 5.5: OSA+ on Komodo, statistics.

The RTSJ evaluations set

Before starting to discuss about the *RTSJ* set, we will discuss some issues which are valid for the *Sun* evaluations set, too.

In order to check the consistency of our evaluations, we recorded a number of 100 samples of the same test. For each operating system we present two diagrams:

- one diagram which is selected from the 100 samples. This diagram shows a single sample execution and time evolution for our application.

- one diagram which shows the collected statistics from all samples.

As we can observe from all the diagrams (5.6(b), 5.7(b), 5.8(b) and 5.9(b)) which show the collected statistics of the 100 samples, their representation is rather liniar and does not have major variances. This assures us that the measured results are consistent and representative.

Another important observation which has to be stated before going to present the evaluations is about the garbage collector on the java virtual machines. As it can be seen from our pseudocode for the Client/Server services, the resources are allocated before the communication starts. This has an important impact on the overall performance of the application. When no more resources are allocated during run-time by the application, the garbage collector does not have to make any action in order to collect the unused application objects. Moreover, before starting each test we called the **System.gc()** method, which should collect all the unused objects up to that moment. We have monitored also the activity of the garbage collector on Sun JVM and did not registered any collecting actions.

The inactivity of the garbage collector shows us that we have succeded to separate the initialization phase, when the resources are allocated by the middleware, from the operational phase, when no resources are more allocated. However, for the Komodo microcontroller this behaviour of inactivity from the side of the garbage collector is not encounted. In fact, we could observe on the Komodo Simulator that the system itself was allocating some memory for executing some optimizations for the java byte code of our application. This was resulting in a continue activity of the GC which affected the bahaviour of the OSA+ application.

With the *RTSJ* evaluations set we wanted to observe how OSA+ performs on top of an real-time java virtual machine. It can be seen from the diagrams which show the time evolution of a single sample, that our Client/Server application produces almost the same pattern on all the operating systems. This pattern is identified through regular peaks which are with approximatively 10 microseconds greater than the average. However, on the RedHat operating system, this peak has a higher value. This can be explained by the fact that the real-time java virtual machine has no real-time support from the operating system. These peaks can be accepted for a real-time application, if the following conditions are respected:

- the peaks must be bound, and
- they must be acceptable for the timing constraints of the application.

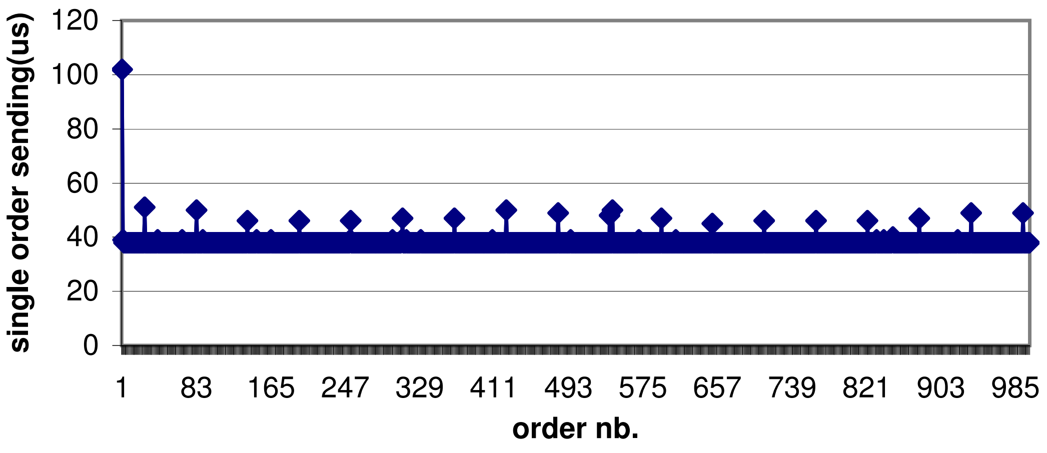
However, for RedHat 9 their is no guarantee that in other situations, when the system is loaded, the value of the peaks will not be even higher. In case of TimeSys 4.1, the company provided for the Linux GPL version a bounded time of 1000 microseconds maximum latency, see TimeSys white paper [15]. In the Linux preemptible kernel, whenever a kernel thread needs exclusive access to a resource, all other kernel threads must wait, resulting in maximum latency that is as long as the duration of the longest non-preemptible interval. For the comercial version, TimeSys Linux/Real Time, the maximum latency is even better, 10 microseconds. Furthermore, in the same paper they estimate for a standard Linux (e.g. RedHat) a maximum latency of 100000 microseconds, much more higher than both TimeSys real-time operating systems.

From the collected sample statistics diagrams, it can be observed that TimeSys 4.1 produces the smaller peaks, less than 250 microseconds. All other operating systems produced peaks around 4500 microseconds. In fact, these peaks are due to the state of the operating system at that time, and like already mentioned they could appear as result of our bash script which started each test and saved the results in a file. We have registered samples with major peaks appearing grouped in small numbers. This could result when our application interrupted the I/O operations for the results file, so they appeared again when sending the next orders.

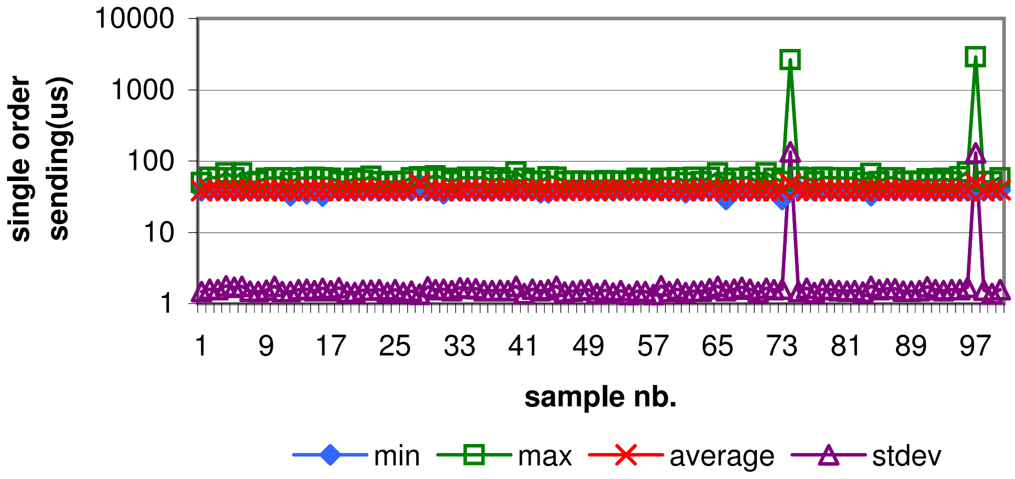
From the statistics presented in figure 5.10, we can observe that OSA+ performs predictably on all operating systems. However, no guaranties exist that the same will happen in case of the RedHat operating system under different load.

The Sun evaluations set

After having evaluated OSA+ on top of a real-time java virtual machine running on standard and real-time operating systems, we would like to observe if the same predictable behaviour is maintained when using a non real-time java virtual machine.

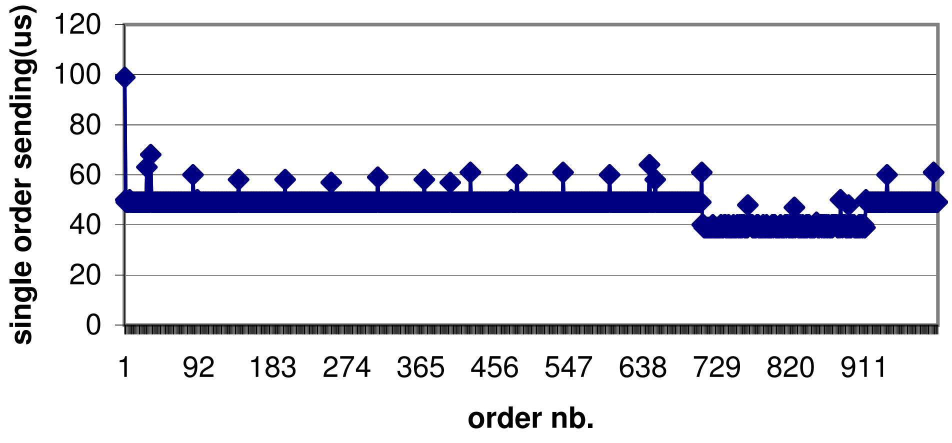


(a) A single sample

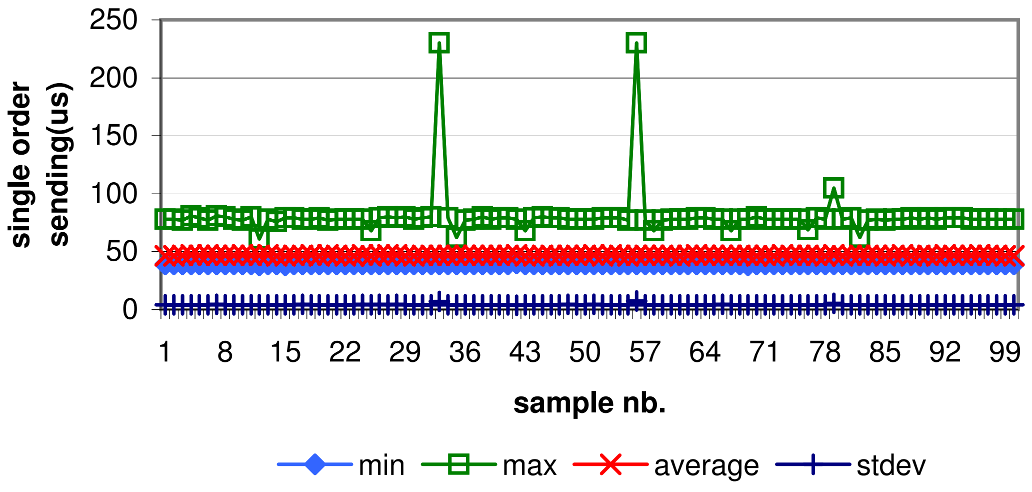


(b) Collected sample statistics

Figure 5.6: OSA+ on MontaVista 3.1 and RTSJ

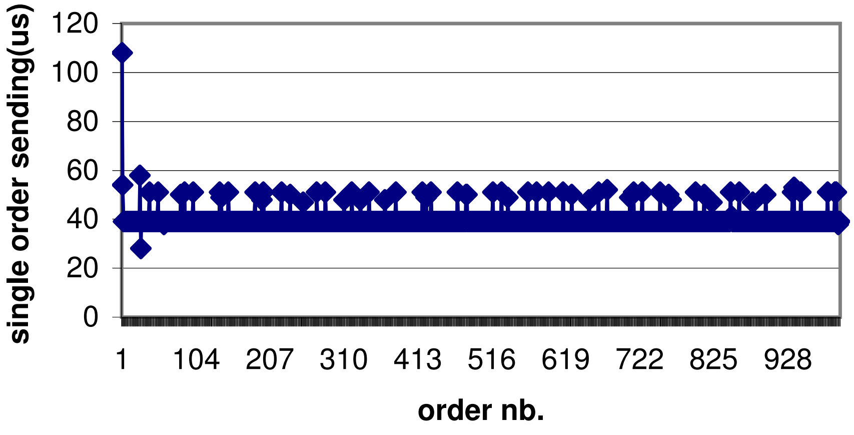


(a) A single sample

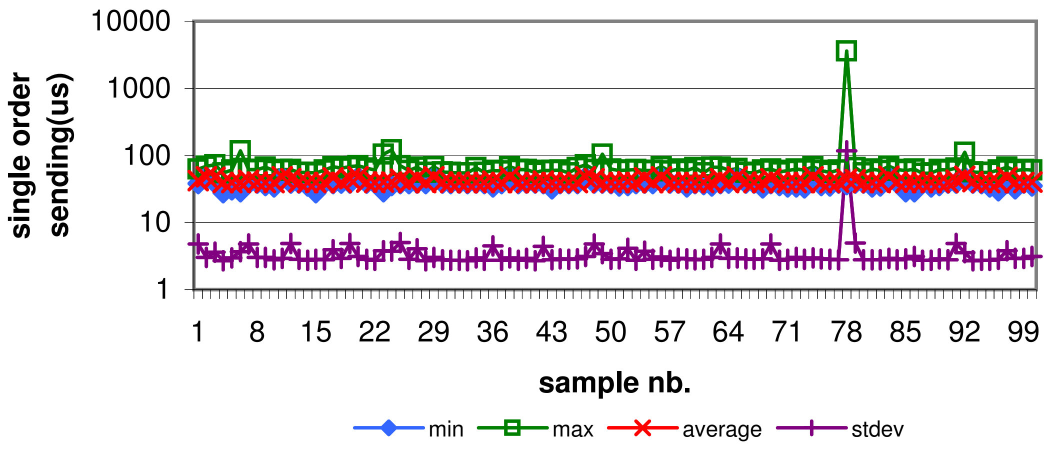


(b) Collected sample statistics

Figure 5.7: OSA+ on TimeSys 4.1 and RTSJ

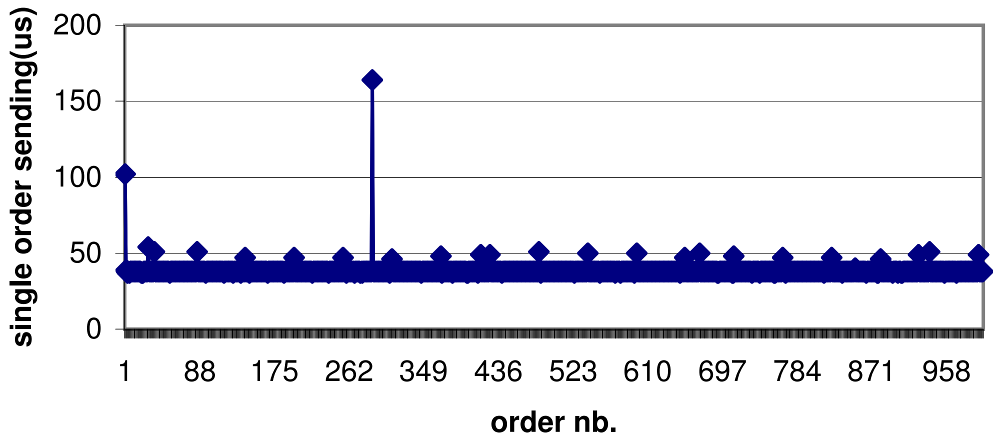


(a) A single sample

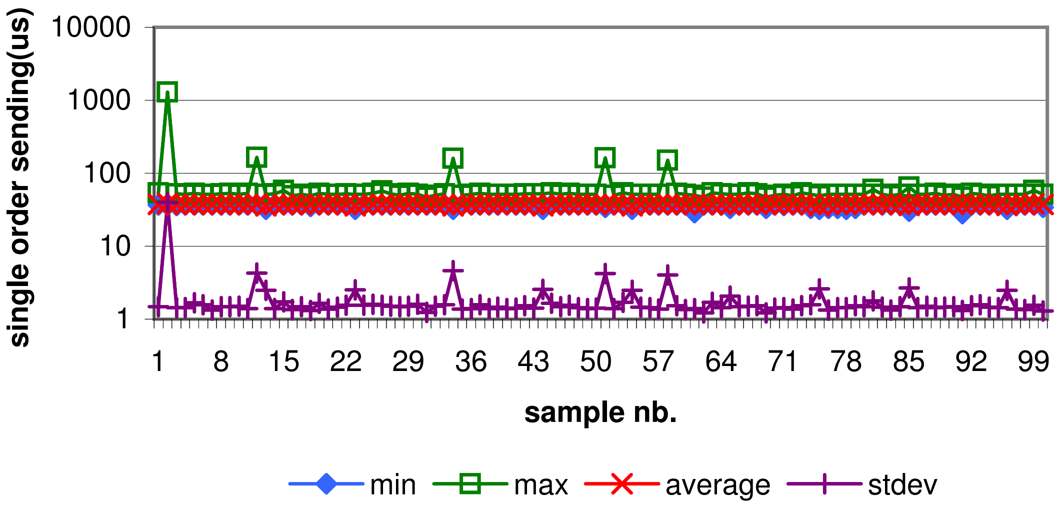


(b) Collected sample statistics

Figure 5.8: OSA+ on TimeSys 5.0 and RTSJ

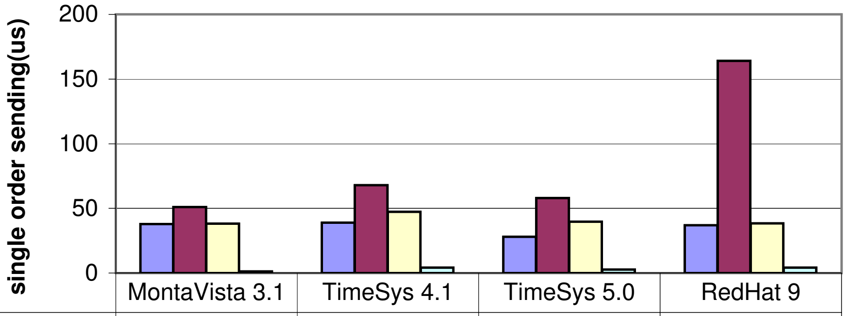


(a) A single sample



(b) Collected sample statistics

Figure 5.9: OSA+ on RedHat 9 and RTSJ



	MontaVista 3.1	TimeSys 4.1	TimeSys 5.0	RedHat 9
■ min	38	39	28	37
■ max	51	68	58	164
□ average	38.2042042	47.2972973	39.67867868	38.31331331
□ stdev	1.343185203	4.242015724	2.777691744	4.303190721

Figure 5.10: OSA+ on RTSJ, statistics.

As expected the real-time distribution is worse (5.11(b) to 5.2.2, regard the logarithmic scale of the y-axis).

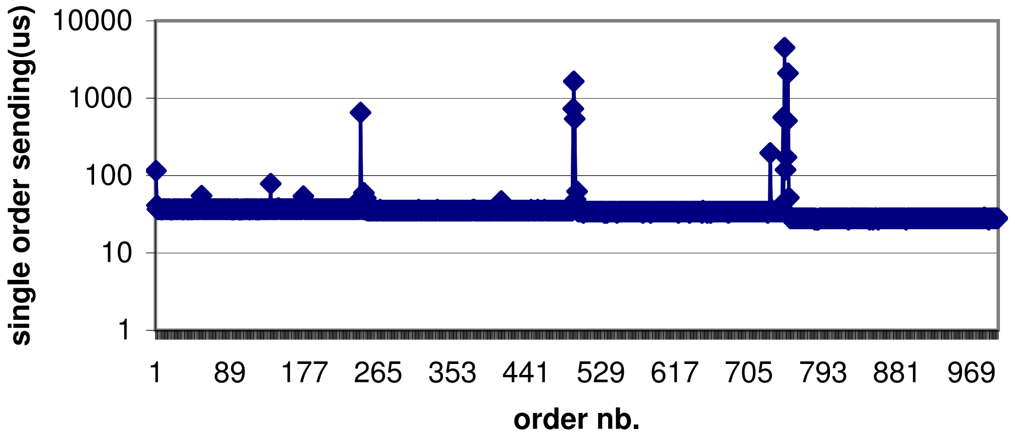
At this evaluations set, like for the *RTSJ* evaluations set, we don't see any notable difference when running in the same java virtual machine, but on different operating systems: real-time and non real-time. This can have the following explanations:

- the OSA+ application reserves resources at the begining and not during operational phase. This results in no activity from the garbage collector, which could influence the application.

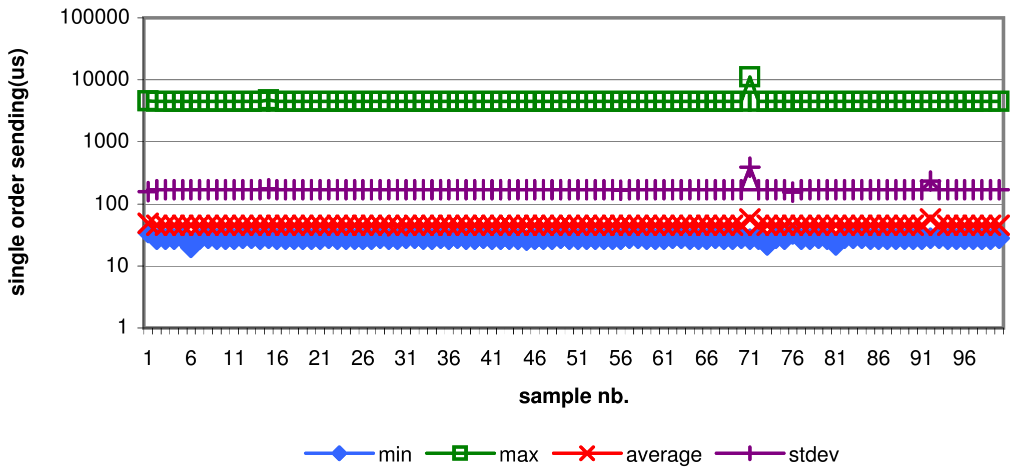
- our tests were made on an idle system. This minimized the influence of other activities on the system, and in fact masked the non predictibale responsiveness of a non real-time operating system in case that more requests exist for the processor.

However, when comparing the *RTSJ* set with the Sun evaluations set, we can clearly see that the first one has a lower variation for the measured values.

As conclusion, the evaluations shows our middleware approach behave predictable if it based on a predictable environment. If only one component is non real-time, the entire configuration looses real-time capabilities, too.

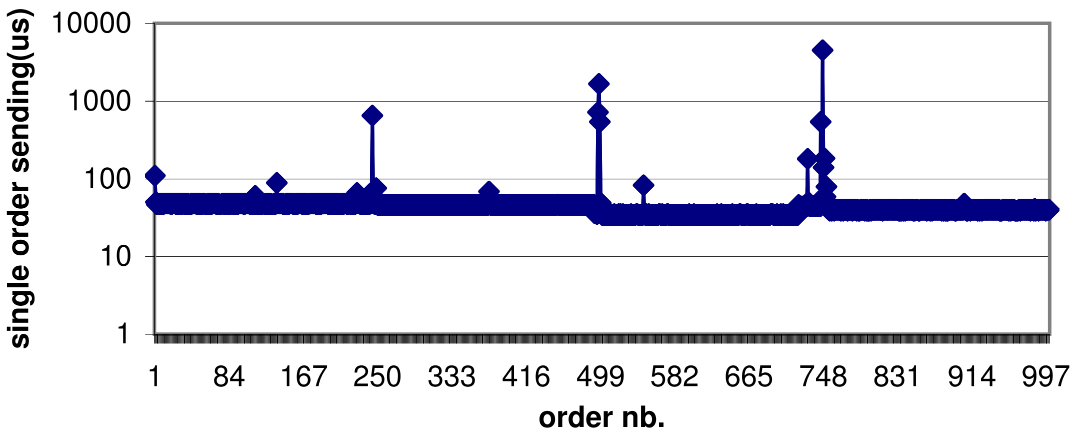


(a) A single sample

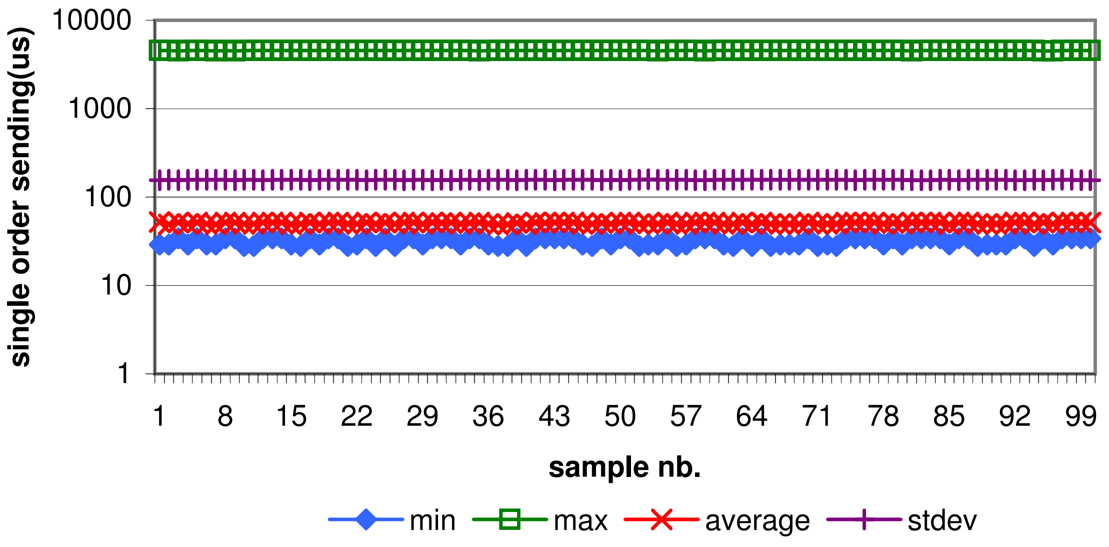


(b) Collected sample statistics

Figure 5.11: OSA+ on MontaVista 3.1 and Sun JVM

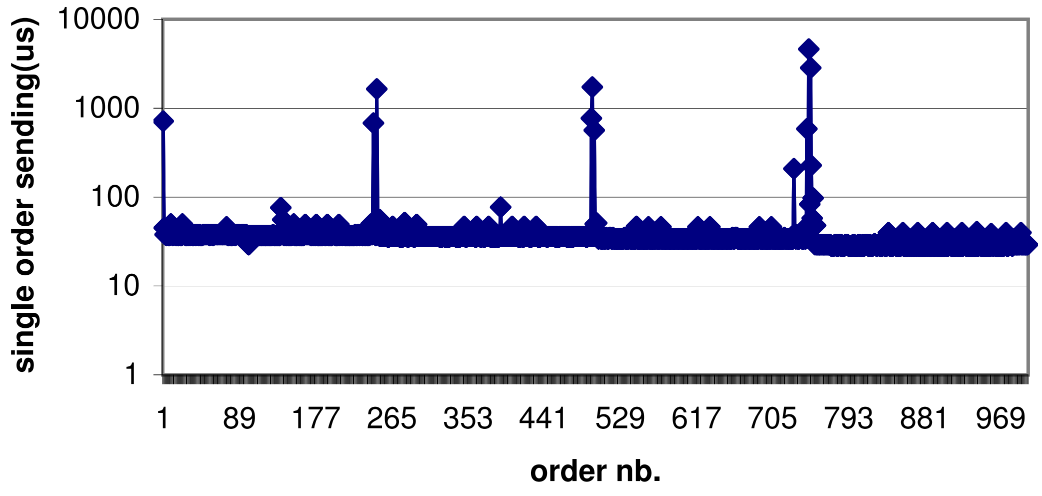


(a) A single sample

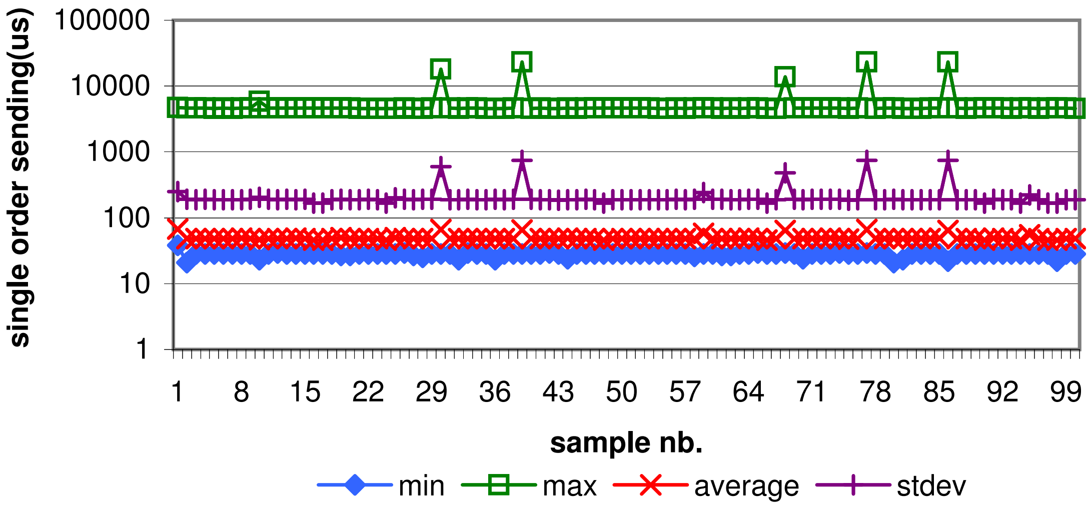


(b) Collected sample statistics

Figure 5.12: OSA+ on TimeSys 4.1 and Sun JVM

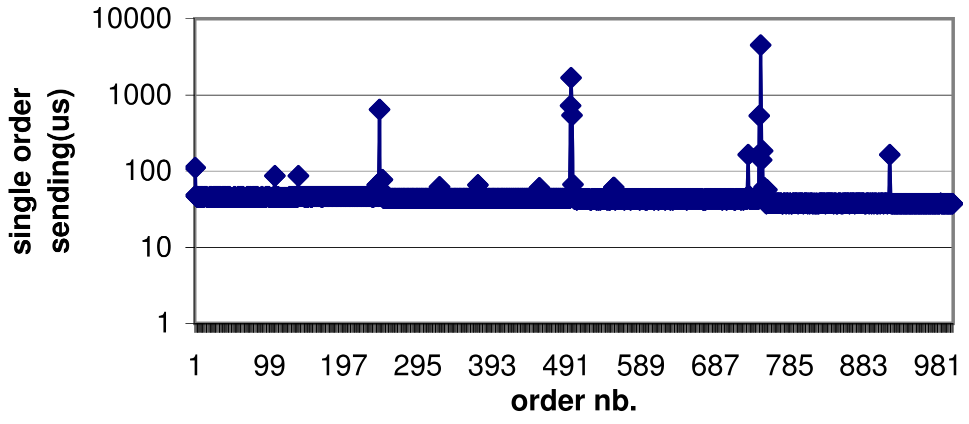


(a) A single sample

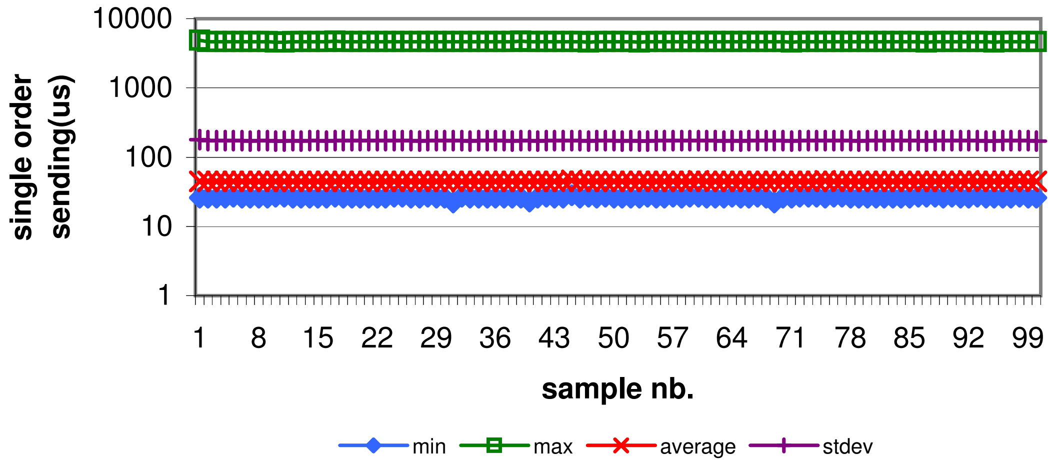


(b) Collected sample statistics

Figure 5.13: OSA+ on TimeSys 5.0 and Sun JVM

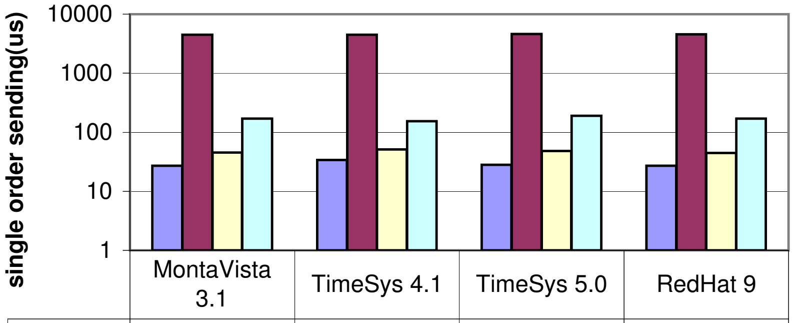


(a) A single sample



(b) Collected sample statistics

Figure 5.14: OSA+ on RedHat 9 and Sun JVM



	MontaVista 3.1	TimeSys 4.1	TimeSys 5.0	RedHat 9
🟦 min	27	34	28	27
🟥 max	4516	4492	4628	4526
🟨 average	45.11711712	51.15815816	48.29229229	44.51751752
🟦 stdev	169.3079221	154.1703263	189.619129	169.8477898

Figure 5.15: OSA+ on Sun JVM, statistics.

5.3 Performance overhead

In the previous section we analyzed and prooved the predictability of our middleware approach. But, this characteristic together with the very good results for the memory footprint could not be enough for using OSA+ in some applications. The reason is, the performance degradation introduced by the middleware might break tight timming requirements of the application. Therefore, we have to analyze the performance overhead introduced by our approach as well. For this, we measured the communication latencies introduced by OSA+ in two cases:

- local communication, services are on the local platform. This measurements are especially important for the middleware itself. Because OSA+ is using a microkernel architecture, the middleware uses platform services for accomplishing some tasks. These services can intercommunicate over the basic communication mechanism. Therefore, these evaluations are important for the performance of OSA+ middleware.

- remote communication, services are on different platforms. As OSA+ is used for distributed systems, the latencies introduced for remote communication must also be measured.

5.3.1 Local communication

For the local communication measurements, we compared the OSA+ Client/Server application with a pure Java Client/Server application. In the pure Java application the Client and Server threads exchange data through a message queue and use the java synchronization mechanims to announce the arrival of a message in this queue. We measured the average time for sending a 32 bytes message for a number of 1000 operations. To compute the OSA+ performance overhead we compared the obtained time with the time for sending a single 32 bytes order using the basic communication mechanism which is already measured, see figures 5.5 and 5.10. The overhead was measured for two real-time configurations:

- Komodo micro-controller using FPP scheduling scheme, no garbage collection with the same priority for both services, and
- TimeSys 4.1 real-time operating system and RTSJ java virtual machine

Additionally, we have simulated a payload (data processing activity) from the side of the server for each message sent. The idea is to simulate a real application which will do some processing with the message, and to observe how the overhead evolves for different payloads.

Table 5.2 presents the measured values. Regarding the timing constraints for the application that is using our middleware, the 22 us overhead introduced by OSA+ in case of the powerfull Pentium processor is almost negligible. For the embedded system, the 4.3 milliseconds communication time has to be considered especially if the application has to react to events which introduce lower timming constraints, e.g. 10 ms. In this case, due to the low computation power of the micro-controller, the time left for other processings is quite narrow. To overcome such tight constraints, the user has different posibilities to further minimize the overhead:

- if it is possible, the service could be registered on the platform as a procedural service. The communication time is approximatively half compared with the case of using a lightweight service, see figure 5.19.
- the procedural interface mechanism can be used. This introduces almost no overhead, see figure 5.19.

Configuration	Java app.	OSA+ app.	OSA+ overhead
TimeSys 4.1 and RTSJ	25 us	47 us	22 us
Komodo, FPP, no GC	1.6 ms	4.3 ms	2.7 ms

Table 5.2: OSA+ performance overhead for sending a single message

Figures 5.16 and 5.17 show the evolution of the performance overhead when sending a single message using our middleware and considering a processing payload for each message. The measured time in the figures is splitted in:

- payload for processing a single message by the server
- time which the Java application needs to send the message
- overhead introduced by OSA+ when sending the same message using the basic communication mechanism

As we can see, already for a 195 microseconds payload (the tile with value 242), the performance overhead introduced by OSA+ reaches a value of only 10% in the configuration which is using the powerfull Pentium processor. On the Komodo micro-controller we need a payload of 25.4 milliseconds to reach the same 10% performance overhead which for an 5.5 MHz micro-controller is quite a good value. The measurements made in figures 5.16

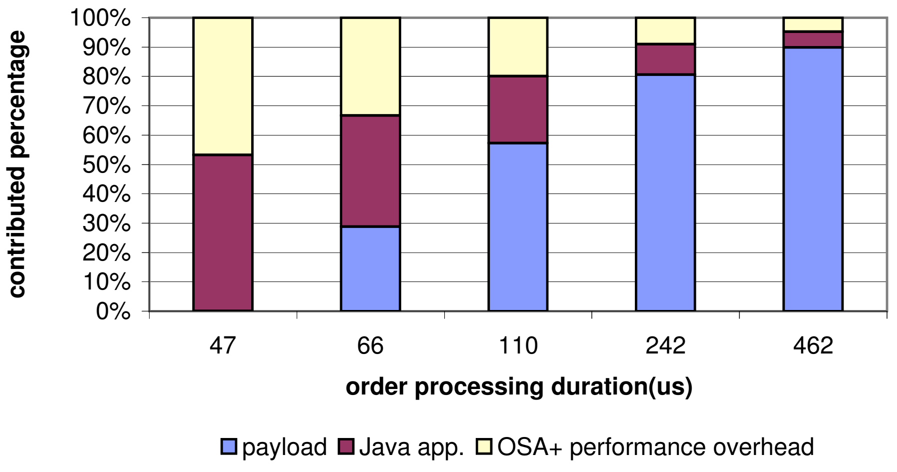


Figure 5.16: OSA+ performance overhead on TimeSys 4.1 and RTSJ

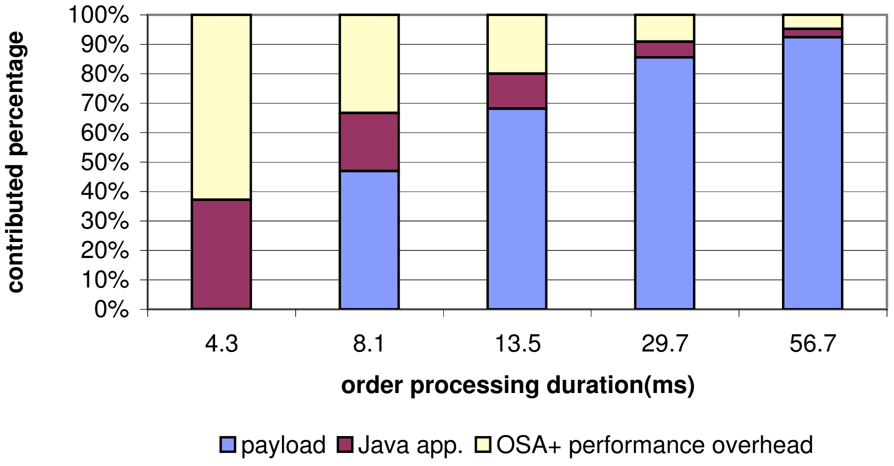


Figure 5.17: OSA+ performance overhead on Komodo, FPP, no GC, same priority

and 5.17 are directly compared in figure 5.18, where the time is normalized and measured in clock cylces. It can be easily observed that on the Komodo embedded system the computation power is more efficient used, to reach a 5% performance overhead (last plotted value) on Komodo, are necessary only 38% of the clock cycles needed by the Pentium processor. This is due to the fact that the Komodo micro-controller is highly specialized for executting Java applications. Otherwise, we are expecting the same evolution as for the Pentium processor on other embedded systems which are using a simillar configuration, see figure 1.2.

5.3.2 Remote communication

For the remote communication we measured the latency introduced only by the OSA+ when sending a 48 bytes order. The latency does not include the network communication and the time needed by the functions used from the `java.nio` package. The test was done in a configuration which includes TimeSys 4.0 RTOS and Sun JVM running in interpreted mode. We choosed Sun JVM because the reference implementation of RTSJ does not implement the `java.nio` package which is used by the TCPIPService. The average for 1000 orders was about 498 microseconds. For realizing the remote communication, the middleware is using the basic communication

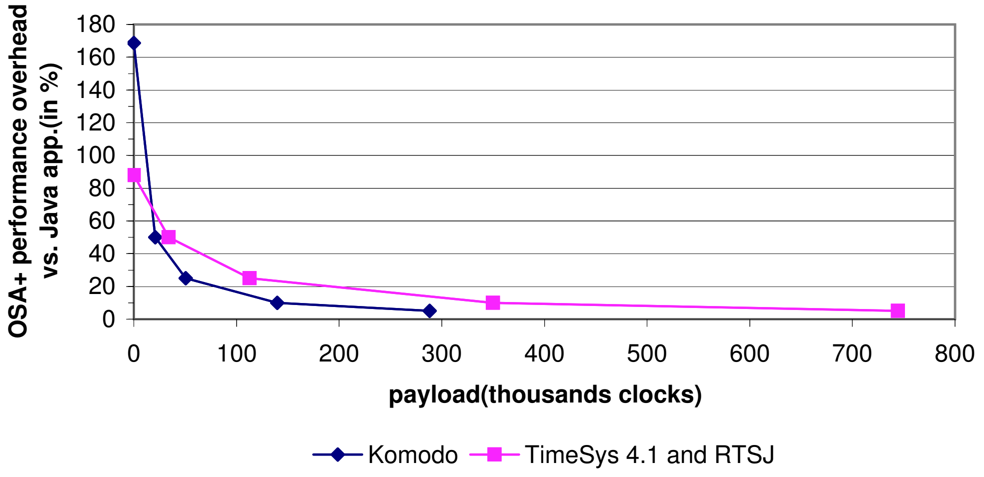


Figure 5.18: OSA+ performance overhead, TimeSys 4.1 and RTSJ vs. Komodo

mechanism where jobs are exchanged between OSA+ Core, HLCService and TCPIPService. The latency can be further minimized if the procedural interface is used.

5.4 Avoiding microkernel communication drawbacks

As discussed in section 4.2.2, the microkernel architecture which we are using for our approach introduces disadvantages in terms of performance degradation and increased communication overhead. We addressed the performance degradation by introducing:

- the procedural interface communication mechanism, and
- the non copy variant of the order communication mechanism, see 4.1.2

Both techniques can be used only on the local platform. To evaluate these communication mechanisms we used the same test described in the previous section 5.3 with a payload of 0 microseconds and for the following configurations:

1. Client: lightweight service, Server: procedural service. With this

configuration we wanted to evaluate order communication for a **procedural service**.

2. Client: lightweight service, Server: lightweight service. The Client is using the **procedural interface** of the Service to exchange data.

3. Client: lightweight service, Server: lightweight service. The **standard order communication** mechanism is used.

4. Client: lightweight service, Server: lightweight service. The **non copy order communication** mechanism is used.

In figure 5.19 can be observed that the latency of the procedural interface mechanism is comparable as to a direct method call invocation and the time to send 1000 orders is below one microsecond. We can conclude that this is a very efficient way to communicate between local services and to improve considerably the performance of our microkernel based middleware. However, this mechanism has to be used as less as possible, because it's introducing strong dependencies between service.

To avoid the strong dependencies introduced by the procedural interface, it can be choosen to implement only basic services which are critical for the performance of the platform as procedural services. If this is possible (no control flow is needed for the service), it can be observed that communicating with orders to a procedural service is more than two times faster than communicating between two lightweight services.

The non copy variant of the order communication mechanism introduces constant overhead independently of the order size. This mechanism, when used on the local platform, can speed up the application especially when large amounts of data must be transported.

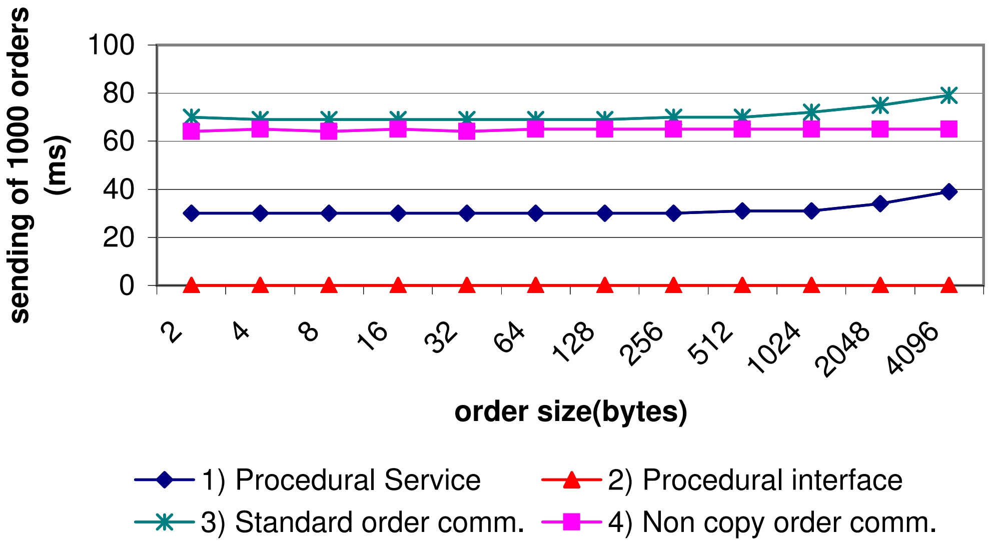


Figure 5.19: Performance of OSA+ communications mechanisms

Chapter 6

CONCLUSIONS AND FUTURE DIRECTIONS

6.1 Conclusions

In this thesis, we presented our approach which addresses the development of applications on distributed real-time embedded systems. In the first two chapters we introduced the research domain and described the requirements and the problems which appear when trying to cope with these requirements. Due to the fact that our research domain represents an intersection of three research areas:

- distributed systems
- real-time systems, and
- embedded systems,

the mission to find a solution which solves the requirements of all these areas was more difficult. We had to address issues regarding distribution, our solution must have strict real-time capabilities and must fit on systems with very low resources. Nevertheless, there are already existing solutions in our research domain. We have summarized them and presented a brief comparision in the third chapter. Here, we already introduced the main aspects which differentiate our approach from the other solutions. The following list summarizes the contributions of this thesis to the research domain:

- **full microkernel architecture**. We introduced a full microkernel middleware architecture which permited us to reach a high adaptability:

  - to the environment. Our middleware can adapt to different processing and communications environments.
  - to low resource systems. Using a microkernel architecture, it becomes more easy to select only the neded components for a specific application. Moreover, these components can be further configured and adapted to the low resource systems in a user transparent manner (e.g. some functionalities can be removed from the components).

  The full microkernel architecture makes the adaptability task to be more easy due to the *uniform and independent* view of each component. This differs from other approaches which use a microkernel architecture, but introduce tight dependecies between the components. The full microkernel approach permits us to reach a memory footprint of 44 KB for a distributed version of the middleware, which is the smallest footprint from the presented approaches.

- **service orientation**. Our approach is the only one in our research domain which is service oriented. The other approaches are object oriented. This has the following important consequences:

  - our approach is working at a higher abstract level. The notion of service is focused more on the offered functionality and less on the implementation and programming language aspects. As an immediate result of this, the management of services introduces less overhead. For example, they can overlap on many other objects from a object oriented programming language.
  - due to the abstraction level, our approach is loosely coupled. It can be more natural and easily implemented in different programming languages. This differs from the other approaches which are tight to the processing environment.

- **chain communication mechanism**. This communication mechanism permits our middleware to establish optimum inter-platform connections. The connections are optimum in regard to communication time or amount of data to transfer. The mechanism can be used when multiple connections has to be established by the application.

In chapter 4 we described the concepts and the realization of our approach and we pointed out the differences to the other approaches. Nevertheless, we had to evaluate if our concepts produce good results when there are applied in practice. Thus, we evaluated our middleware in chapter 5 and proved, that it respects the real-time and embedded systems requirements.

6.2 Future directions

We have realized an implementation reference for our approach, but there are still many things that can be added. As a matter of consequence, in this reference implementation we realized only the basic concepts which permit us to further build on top of what we have alreay realized.

Firstly, we have to implement the Address Resolution Service. This will permit the user to transparently discover services on remote platforms. Currently, the user has to provide specific parameters in order to identify a remote service.

Then we will realize a bridge between OSA+ and CORBA as an additional service. The service will be able to understand and communicate with CORBA applications. CORBA is the most important standard for distributed applications.

We would also like to publish our implementation, so it can be tested by other people and hopefully evolve in more richer and mature middleware.

Finally, organic computing is an upcoming research initiative. Here, computational systems should behave like organic entities and provide self-x features like self-organizing, self-configuring, self-healing, self-protecting, etc. Extending OSA+ to an organic middleware could introduce such features to distributed embedded systems by automatically organizing, configuring, copying or moving services.

Bibliography

[1] Extreme CHAOS. Technical report, The Standish Group International, Inc., 2001.

[2] Cristopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language.* Oxford University Press, 1977.

[3] David Bakken. *Encyclopedia of Distributed Computing*, chapter Middleware. Kluwer Academic Publishers, 2003.

[4] H.E. Bal. *Programming Distributed Systems.* Pretince-Hall, Inc., 1990.

[5] Philip A. Bernstein. Middleware: A Model for Distributed System Services. *Communications of the ACM*, 39(2):86–98, February 1996.

[6] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Professional, 1987.

[7] Toni A. Bishop and Ramesh K. Karne. A survey of middleware. In N. Debnath, editor, *Proceedings of ISCA 18th International Conference Computers and Their Applications*, pages 254–258, Honolulu, Hawaii, USA, March 2003. ISCA.

[8] Gordon Blair and Jean-Bernard Stefani. *Open Distributed Processing and Multimedia.* Addison-Wesley Professional, 1998.

[9] Bosch. Can 2.0b. Available: `http://www.can.bosch.com/content/Literature.html` (Accessed: 2004, November 9), 1991.

[10] Don Box. *Essential COM.* Addison-Wesley Professional, 1998.

[11] U. Brinkschulte, C. Krakowski, J. Kreuzinger, R. Marston, and T. Ungerer. The Komodo Project: Thread-Based Event Handling Supported by a Multithreaded Java Microcontroller. In *Proceedings of the 25th EUROMICRO Conference*, volume 1, Milan, Italy, September 1999.

[12] H. Rebecca Callison and Daniel G. Butler. Real-Time CORBA Trade Study. Technical Report D204-31159-1-5, Boeing Corporation Phantom Works, October 2000.

[13] Common Object Request Broker Architecture: Core Specification 3.0.3. Technical Report formal/2004-03-01, Object Management Group, March 2004.

[14] CORBA Components Specification 3.0. Technical Report formal/02-06-66, Object Management Group, June 2002.

[15] TimeSys Corporation. TimeSys Linux GPL. Available: `http://www.timesys.com/index.cfm?bdy=bsp_downloads.cfm` (Accessed: 2004, November 9).

[16] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems. Concepts and Design.* Addison-Wesley Professional, 2nd edition, 1994.

[17] Data Distribution Service for Real-Time Systems Specifications. Technical Report ptc/03-07-07, Object Management Group, May 2003.

[18] J. Dongarra. Performance of various Computers using Standard Linear Equations software in a Fortran Environment. In *Proceedings of Third Conference on Multiprocessors and Array Processors*, San Diego, California, January 1987.

[19] Wolfgang Emmerich. *Engineering Distributed Objects.* John Wiley and Sons, 2000.

[20] Robert Englander. *Developing Java Beans.* O'Reilly, June 1997.

[21] P.H. Enslow. What is a ‘distributed’ data processing system? *Computer*, 11(1):13–21, January 1978.

[22] Alexander Romanovsky et. al. CaberNet Vision of Research and Technology Development in Distributed and Dependable Systems. Technical report, Network of Excellence in Distributed and Dependable Computing Systems, January 2004.

[23] The Real-Time for Java Expert Group. The Real-Time Specification for Java. Available: `https://rtsj.dev.java.net/rtsj-V1.0.pdf` (Accessed: 2004, November 9).

[24] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, 1994.

[25] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[26] Leonard Gilman and Richard Schreiber. *Distributed Computing with IBM(r) MQSeries*. Wiley, 1996.

[27] IEEE P1596.6 Working Group. SCI/RT - Scalable Coherent Interface for Real-time applications, 1992.

[28] Carl L. Hall. *Building Client/Server Applications Using Tuxedo*. Wiley, 1996.

[29] Mark Hapner, Rich Burridge, Rahul Sharma, Joseph Fialli, and Kate Stout. Java Message Service Specification 1.1. Technical report, Sun Microsystems, Inc., April 12 2002.

[30] Richard Hayton. FlexiNet Open ORB Framework. Technical Report 2047.01.00, APM Ltd, Poseidon House, 1997.

[31] Eugene S. Hudders. *CICS: A Guide to Internal Structure*. Wiley, 1994.

[32] TIBCO Software Inc. TIBCO Rendezvous. Available: `http://www.tibco.com/software/enterprise_backbone/rendezvous.jsp` (Accessed: 2004,
November 9).

[33] Real-Time Innovations. NDDS: The Real-Time Publish- Subscribe Middleware. Available: `http://www.rti.com/products/ndds` (Accessed: 2004,
November 9).

[34] Prashant Jain and Douglas C. Schmidt. Service Configurator A Pattern for Dynamic Configuration of Services. In *Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems*, Portland, Oregon, USA, June 1997. USENIX.

[35] Coplien J.O. *Advanced C++ programming styles and idioms.* Addison-Wesley Professional, 1994.

[36] Mark Klein, Thomas Ralya, Bill Pollak, Ray Obenza, and Michael Gonzalez Harbour. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate-Monotonic Analysis for Real-Time Systems.* Kluwer Academic Publishers, 1993.

[37] Eric Lafortune. ProGuard. Available: `http://proguard.sourceforge.net` (Accessed: 2004, November 9).

[38] Thomas Ledoux. OpenCorba: A Reflective Open Broker. In Pierre Cointe, editor, *Meta-Level Architectures and Reflection, Second International Conference, Reflection'99*, volume 1616 of *Lecture Notes in Computer Science*, pages 197–214. Springer-Verlag, Saint-Malo, France, July 1999.

[39] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *ACM*, 20(1):40–61, 1973.

[40] Bertrand Meyer. *Object oriented software construction.* Pretince Hall PTR, 2nd edition, March 2000.

[41] A. K. Mok and M. L. Detouzos. Multiprocessor scheduling in a hard real-time environment. In *Proceedings of the 7th Texas Conference on Computing Systems*, November 1978.

[42] MontaVista Software. MontaVista Linux. Available: `http://www.mvista.com/previewkit/index.html` (Accessed: 2004, November 9).

[43] NBIO: Nonblocking I/O for Java. Available: `http://www.eecs.harvard.edu/~mdw/proj/java-nbio/` (Accessed: 2004, December 10).

[44] James Noble and Charles Weir. *Small Memory software*. Pearson Education Limited, 2001.

[45] Rickard Oberg. *Mastering RMI: Developing Enterprise Applications in Java and EJB*. Wiley, February 2001.

[46] Information Technology - Open Distributed Processing - Reference Model: Foundations. Technical Report ISO-20746-2, International Standards Organization, 1996.

[47] David S. Platt. *Introducing Microsoft .NET*. Microsoft Press, May 2001.

[48] I. Pyarali, C. O Ryan, D. C. Schmidt, N.Wang, V. Kachroo, and A. Gokhale. Applying optimization patterns to the design of real-time orbs. In *Proceedings of the 5th Conference on Object-Oriented Technologies and Systems*, San Diego, CA, May 1999.

[49] Red Hat Inc. RedHat Linux 9. Available: `ftp://ftp-stud.fht-esslingen.de/pub/Mirrors/ftp.redhat.com/redhat/linux/9/` (Accessed: 2004, November 9).

[50] ROFES: Real-Time CORBA for embedded systems. Available: `http://www.lfbs.rwth-aachen.de/content/20` (Accessed: 2004, December 10).

[51] D. C. Schmidt and C. Cleeland. Applying patterns to develop extensible orb middleware. *IEEE Communications Magazine*, 37, April 1999.

[52] Douglas Schmidt. Middleware R&D Challenges for Distributed Real-time and Embedded Systems. Workshop on New Visions for Software Design and Productivity, December 2002.

[53] Etienne Schneider. *A middleware approach for dynamic real-time software. Reconfiguration on distributed embedded systems*. PhD thesis, University Louis Pasteur Strasbourg, 2004.

[54] Michael D. Schroeder. *Distributed Systems*, chapter 1, pages 1–16. Addison-Wesley Professional, 2nd edition, 1993.

[55] Bill Segall and David Arnold. Elvin has left the building: A publish/-subscribe notification service with quenching. In *In Proceedings of the 1997 Australian UNIX Users Group*, Brisbane, Australia, September 1997.

[56] A. Singhai, A. Sane, and R. H. Campbell. Quarterware for middleware. In *Proceedings of ICDCS'98*, May 1998.

[57] John Stankovic. Misconceptions about real-time computing: A serious problem for next generation systems. *IEEE Computer*, 21(10):10–19, October 1988.

[58] John A. Stankovic, Marco Spuri, Krithi Ramamritham, and Giorgio C. Buttazzo. *Deadline scheduling for real-time systems: EDF and related algorithms.* Kluwer Academic Publishers, October 1998.

[59] V. S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–340, 1990.

[60] Andrew S. Tanenbaum and Maarten van Stenn. *Distributed Systems. Principles and Paradigms.* Pretince-Hall, Inc., 2002.

[61] TimeSys Corporation. A TimeSys Perspective on the Linux Preemptible Kernel. Available: `http://www.timesys.com/_content/media/docs/whitepapers/` (Accessed: 2004, November 9).

[62] XML Protocol Working Group W3C. Soap version 1.2. Available: `http://www.w3.org/TR/soap12-part1/` (Accessed: 2004, November 9), June 2003.

[63] D. Wagner. Spacebourne Processors: Past, Present and Future Satellite Onboard Computers. 49th International Astronautical Congress, Sept. 28 - Oct. 2. 1998.

[64] Uffe Kock Wiil and Peter J. Nurnberg. Evolving hypermedia middleware services: Lessons and observations. In *Proceedings of Selected Areas in Cryptography*, pages 427–436, 1999.

Publications by the author

In reverse chronological order:

1. E. Schneider, F. Picioroaga, U. Brinkschulte: Dynamic Reconfiguration through OSA+, a Real-Time Middleware, Middleware 04, 1st Middleware Doctoral Symposium, 2004, 18th-22nd October 2004, ACM, Toronto, Canada

2. A. Bechina, U. Brinkschulte, F. Picioroaga, E. Schneider: OSA+ Real-Time Middleware. Results and Perspectives. International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2004), Wien, Austria, May 12 - 14, 2004.

3. U. Brinkschulte, A. Bechina, F. Picioroaga, E. Schneider: Open System Architecture for embedded control applications Concepts and results. To be published in ICIT'03 International Conference on Industrial Technology, Maribor, Slovenia, December 10 - 12, 2003, IEEE.

4. A. Bechina, F. Picioroaga, U. Brinkschulte: Towards a Collaborative Engineering Framework. International Conference on Computer Information Systems and Industrial Management Applications, CISIM'03, Elk, Poland, June 26 - 28, 2003.

5. U. Brinkschulte, A. Bechina, B. Keith, F. Picioroaga, E. Schneider: A Middleware Architecture for Ubiquitous Computing Systems with Real-Time needs. 2002 IAR Workshop (Institute for Automation and robotic Research), Grenoble, France, November 23-24, 2002

6. U. Brinkschulte, A. Bechina, F. Picioroaga, E. Schneider: Distributed Real-Time Computing for Microcontrollers - the OSA+ Approach.

International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2002), Washington D.C., 2002, IEEE, p.169-172. ISBN: 0-7695-1558-4

7. U. Brinkschulte, A. Bechina, F. Picioroaga, E. Schneider, Th. Ungerer, J. Kreuzinger, M. Pfeffer: A Micro-kernel Middleware Architecture for Distributed Embedded Real-Time Systems. 20th Symposium on Reliable Distributed Systems, New Orleans, MI, USA, October 28-31, 2001, IEEE, p.218-226, ISBN: 0-7695-1366-2.

8. Bechina, U. Brinkschulte, F. Picioroaga, E. Schneider: Real Time middleware for industrial embedded measurement and control application OSA+. The 2001 International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, Nevada, USA, June 25-28, 2001, CSREA, p.843-849, ISBN: 1-892512-69-6.

Curriculum vitae

Florentin Picioroagă

30 July, 1977	Born in Iaşi, Romania
Oct. 1991 - Jul. 1995	High School
	"Anghel Rugină" High School, Vaslui, Romania
	Specialization: computer science
Oct. 1995 - Jun. 1999	Diploma degree
	Faculty of Computer Science
	"Al. I. Cuza" University of Iaşi, Romania
Sep. 1999 - Jun. 2000	Master degree
	Faculty of Computer Science
	"Al. I. Cuza" University of Iaşi, Romania
	Specialization: distributed and parallel programming
Nov. 2000 - Dec. 2004	Ph.D. student
	Institute for Process Control and Robotics
	University of Karlsruhe, Germany
	&
	LIIA - INSA de Strasboug, France

Appendix

Application example using OSA+

```
import osa.*;

public class ClientService extends osa.Service {

  //This executes the construct order received from the platform.
  public final void construct(final osa.Error osaErr, final Job job) {
    //nothing to do
    sendResult(osaErr, job);
  }

  //This executes the destruct order received from the platform.
  public final void destruct(final osa.Error osaErr, final Job job) {
    //nothing to do
    sendResult(osaErr, job);
  }

  public static void main(String[] args) {
    MicroKernel osaMKernel = new MicroKernel(
      new BucketContainer(3, 2, 3, 3),//Service Repository
      new BucketContainer(3, 2, 3, 3),//Connection Repository
      null);
    osa.Error osaErr = new osa.Error();

    //Initialize the OSA+ platform
    osaMKernel.init(osaErr);

    ProcessService processService = new ProcessService();
    //we add the ProcessService to the platform
    osaMKernel.registerService(
      osaErr, processService, "ProcessService",//service name
```

```
        "1.0", //service version
        PlatformInfo.PRC_SERVICE); //procedural service

      //we add HLC service because we want to have remote access
      HLCService hlcService = new HLCService();
      osaMKernel.registerService(osaErr, hlcService, "HLCService","1.0",
                                 PlatformInfo.LWP_SERVICE);

      //we can have remote access over TCPIP
      TCPIPService tcpipService = new TCPIPService(
        "127.0.0.1", 7777);//IP address of the platform
      osaMKernel.registerService(osaErr, tcpipService, "TCP/IP","1.0",
                                 PlatformInfo.LWP_SERVICE);

      ClientService client = new ClientService();
      //we add the ClientService as a lightweight service
      osaMKernel.registerService(osaErr, client, "HelloWorldService", "0.1",
          PlatformInfo.LWP_SERVICE);
    }

    //This is the entry point of the service for OSA+
    public final void serviceLoop(final osa.Error error) {
      Order receivedOrder, sendOrder;
      osa.Error osaErr = new osa.Error();
      Job job;
      LookUpInfo serverLookUpInfo;
      Connection conn;
      String message;

      while (true) {
        job = waitOrder(osaErr);
        receivedOrder = job.getOrder();
        switch (receivedOrder.getOrderId()) {
          case PlatformInfo.DESTRUCT_FUNCTION_ID:
            destruct(osaErr, job);
            return;
          case PlatformInfo.CONSTRUCT_FUNCTION_ID:
            construct(osaErr, job);
            //discover a SERVER service
            serverLookUpInfo = getMicroKernel().lookUpService("SERVER");
            sendOrder = buildOrder((byte) 100,
          //order id, identifies the functionality of the SERVER service which is
              requested
                                  50); //resultSize
        sendOrder.setOrderData(ByteArray.allocate(60));
        //fill the order with a message
        sendOrder.getOrderData().writeOSAString(
          osaErr, "Hello World!");
        //create a connection to the SERVER service
        conn = getMicroKernel().createConnection(
          osaErr, sendOrder, (short) 1
          //nb. of orders which should be reserved on the connection
          ,serverLookUpInfo);
        //we send the order over the created connection
        sendOrder(osaErr, sendOrder, conn);
        //and we wait the result from the SERVER
        job = waitResult(osaErr);
        //supposing that the SERVER service realizes for orderId 100
        //an echo functionality, we read back the received message
        message = job.getResult().getResultData().readOSAString(osaErr);
        System.out.println("Received: " + message + " from the server!");
        break;
    }
    //we need this in case that this service is registered as
    //procedural service
    if (kernelNeedControlFlow()) {
      return;
    }
  }
 }
}
```

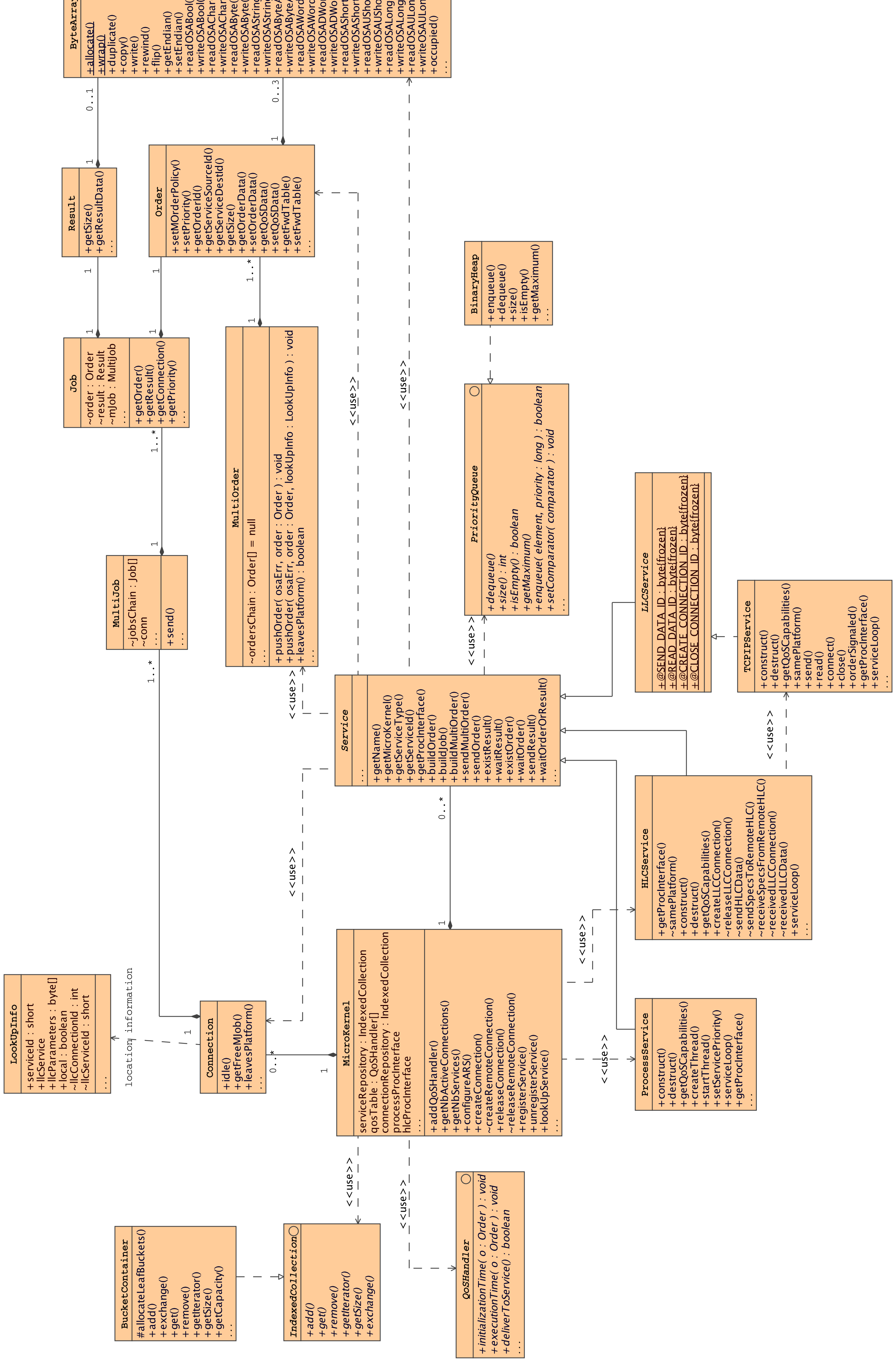


Figure .1: OSA+ UML class diagram