

Flexible QoS management and real-time in OSA+ middleware

F. Picioroaga, U. Brinkschulte
Institute for Process and Robotics
University of Karlsruhe (TH)
D-76131 Karlsruhe, Germany

Abstract—A more and more obvious trend in applied computer science is to move from applications designed in a best-effort spirit to applications that prove to be deterministic, predictable and thus more user controllable and tunable. For these applications it is not only important that a number of services are provided, but these services have to be performed with a certain quality of service. While quality of service can have different meanings - depending on the context of each service, a middleware must be able to adapt to this variety. Moreover, several quality of service requests can be combined in so called multi-dimensional quality of service. We will show in this article: (i) how OSA+ provides a flexible mechanism to cope with the variety of custom QoS requests and multi-dimensional QoS and (ii) how we bring real-time support in our service oriented middleware.

Keywords: multi-dimensional QoS, real-time middleware

I. Introduction

The research domain of distributed systems experiences actually a growing interest for the sub-domain of distributed real-time and embedded systems (DRE). As technology advances and hardware is available at lower costs and tiny dimensions, more devices are becoming interestingly to be enhanced and to seemingly integrate in the network. Currently, these devices are mostly centrally controlled. However, important steps are lately done towards a more autonomous behavior of these small devices, that can interact and cooperate in order to do some tasks. Nevertheless, this implies considerable effort from the side of the design and developing teams. Some examples of applications in this matter are: hospital area networks (networks of monitoring sensors placed on the human body, of medical devices and computers), intelligent houses (a system manages heating, ventilation, energy, etc. in a house), sensor networks (used mostly to monitor: temperature, concentration of chemical material, vibrations, etc).

For some of these applications is not enough to successfully perform a task, but additional demands must be satisfied in order to really use the results produced. Here we can name the applications that need real-time capabilities. For example, in the hospital area network

application the values measured by sensors, which represent the heart pulse rate, must be available within a certain period of time (the deadline) to the monitoring system. But, other kinds of demands can be thought for each specific application. Multimedia applications need to transfer the data for the media streams with a minimum bandwidth or, controlling a robot that does surgical operations needs a high accuracy that require fine timing reactions from the system. For all these, there is other aspect that plays a crucial role - the *quality of service*.

Designing such applications is a complex task and can become heavily time consuming and error prone as more problems arise when working with the hardware available on these devices. For these Moreover, solutions which are available for “standard” hardware (e.g. PCs) are not anymore suitable in this case, e.g. application libraries, communication protocols, etc. Therefore, the user has to build from scratch the application.

Here, we are coming with a solution which eases the development of such applications by managing the interactions between different computing nodes in a heterogeneous environment. Our solution, the OSA+ ¹ middleware, proves to be easily adaptable to different hardware, as we will see further, and is designed having in mind the special requirements of those computing nodes with low resources, real-time needs and custom QoS attributes support.

In section I-A we will enumerate the requirements which a middleware for DRE systems has to fulfill. Then, in section II we will place our approach in the context of other existing solutions. Section III gives an overview for our middleware and discusses the main design principles. Sections IV and V consist the main focus of this article and describe the real-time and quality of service aspects in OSA+. Finally, we will present and discuss some evaluation results in section VI, and we will conclude in section VII.

A. Middleware requirements

Adapting to a variety of hardware devices and providing real support for many application types, it's a

¹Open Service Architecture Platform for Universal Services

difficult task and require a carefully chosen software design for the middleware. Before going to present diverse details about our approach or other solutions, we will summarize here the main requirements that a middleware for *distributed real-time and embedded systems* has to fulfill. These are:

- running in a distributed environment, the middleware must facilitate message exchange and be *available on heterogeneous computing nodes*, e.g. different hardware, using different communication mechanisms.
- the middleware must be able to scale down to systems with *limited computing and memory resources* as a consequence of the size, weight, cost, and power constraints which must be satisfied by the embedded systems.
- real-time capabilities require *deterministic behavior* and *control* over end-to-end system resources, such as memory, CPU, network bandwidth, energy, etc. This control can be achieved by using QoS attributes between the counterpart systems.

Additionally, an important advantage for such middleware is to be *open*. That is, to have clearly specified the interfaces for the provided services and to easily integrate components coming from different sources. An open middleware can be easier adapted for specific applications and ported to different hardware as its components are clearly delimited. Moreover, new components can be easily added, as soon as the middleware framework is understood. We will see in section III, how our middleware proves to be open and which are the possibilities to easily extend it with new functionalities.

II. Related work

The de-facto standard for middleware for real-time distributed systems is real-time CORBA [1] which is part of the CORBA 3 standard. RT CORBA addresses requirements like: fixed priority scheduling, control over ORB resources for end-to-end predictability, a time type, transmission of real-time method invocation, global priority, real-time events and exceptions, documented execution times, etc. However, implementing the full specification for RT CORBA leads to implementations with a big and inadequate footprint for embedded systems. Therefore, the existing implementations choose to implement a part from the specification, e.g. ROFES [2] that has a footprint of 324 KB. Another option is to implement the Minimum CORBA which is a subset of the full CORBA specifications that targets systems with low resources. However, in this case no or less real-time support is available. RT CORBA or Minimum CORBA implementations that target the embedded systems are: OpenFusion e*ORB, ORBExpress RT, RTZen, ROFES. An extended comparison between these middleware and OSA+ can be found in [3]. Nevertheless, we have to

remark here that the footprint of these middleware start from approximately 100 KB up to several hundreds.

Regarding QoS, CORBA addresses QoS information, but lacks a unified framework for supporting Quality of Service (QoS) guarantees. This resulted in a tight coupling to the implementation and specification as opposed with our approach. Moreover, it does not provide the ability to implement custom combinations of different QoS attributes.

Similar approaches with ours for dealing with QoS information are found in CQoS, QBox and FRIENDS described in [4]. They all emphasize a clear separation between mechanisms and the application of QoS enhancements. They use multiple components which can be customarily composed in order to offer support for different QoS. However, the FRIENDS architecture is based on reflection and needs a programming language that supports it. The first two, are realized on middleware level and intercepts middleware messages or requests. They are both based on Cactus [5], a system that supports construction of highly-configurable network protocols and services. While CQoS components are fixed at session creation time and applied to all messages sent in that session, the QBox processes QoS requests on a message-basis. As we will show in section V, OSA+ supports session and a message-basis QoS processing.

III. OSA+ real-time middleware

In this section we will point out the main ideas about the OSA+ real-time middleware (see [3] for a more complete description) and introduce further new aspects.

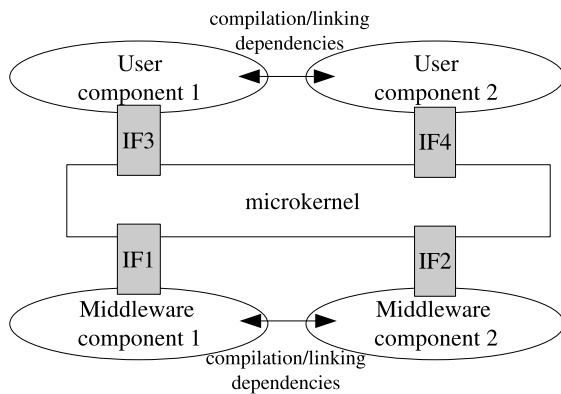
A. Service oriented middleware

OSA+ is a *service oriented* middleware for DRE systems. We are exploring the feasibility of the service concept on embedded systems. To our knowledge, OSA+ is the only service oriented middleware for DRE systems. In OSA+, the active communication parts are services. A service realizes 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 messages that we name *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 with what type of QoS requirements. 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.

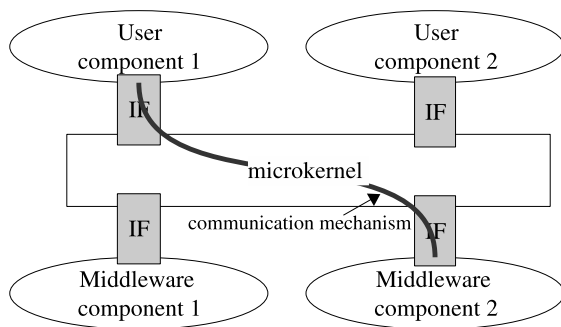
One of the main aspects that differentiate our middleware to the other ones, is that we are using an *uniform approach* for the middleware components (see fig. 1). Thus, a service is the only unit in our middleware to

perform application and system tasks. The same interface is available for configuring the middleware itself or the user application. Moreover, between middleware services there are no programming language specific dependencies, e.g. member accesses of other middleware service. This introduces a clear delimitation for the middleware components - the services and enhances considerably the middleware's openness. Additionally, the API available to each service is kept minimal but powerful enough to permit a good service inter-operability (see section III-B.1).

We will see in the next section that we combine the service orientation with a micro-kernel architecture. However, other middleware approaches are using the micro-kernel architecture only with a compositional purpose, i.e. to configure the middleware with certain components.



a) Compositional microkernel architecture



b) Full and uniform microkernel architecture

Legend:

IF - interface

Fig. 1. Uniform approach for middleware components

For service discovery, each OSA+ platform points to a central service repository where services are published. Actually, we are using a (*service name, service version*) pair to identify a service. However, we are analyzing new decentralized methods for discovering services that

are based on a approximative matching of the service interface for accepted messages. These methods we are researching in the framework of wireless sensor networks, where nodes may discover dynamically at run-time possible communication partners.

B. Micro-kernel architecture

Another main aspect of our middleware is that we are using a *micro-kernel architecture*. We are building the middleware by starting with a minimum foundation of functionalities - the micro-kernel which is the core platform, and enhancing it by adding new components - the services (see fig. 2). The core platform provides the means to add new services and to have local communication between them.

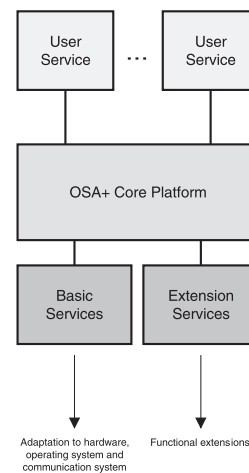


Fig. 2. Micro-kernel architecture

1) **The core platform:** 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 possibility to the user to configure and extend the platform according to its needs. The core platform has three main tasks:

- 1) service management, services can be added and removed on the platform.
- 2) local job management, jobs can be exchanged between the local services.
- 3) user API.

Regarding the user API, the middleware has functions for:

- adding and removing services from the platform
- building, sending and receiving orders and results
- requesting QoS features by attaching the corresponding QoS attributes to the jobs (see section V).

The middleware is able to realize synchronous communication using functions like: `sendOrder`, `sendResult`, `waitOrder`, `waitResult` and asynchronous communication through the non-blocking functions: `existOrder` and `existResult`.

2) **Extending the core functionality:** The core platform is extended through services. These are classified in: *basic services* and *extension services*. The middleware is adapted to different environments by the basic services. They link to the processing environment: Process Service, Event Service, Memory Service and also to the communication environment: Communication Services. The Process Service introduces in the middleware the ability to have services realized as threads and processes. The Event Service allows the platform to use environment based events. Of great interest for us are the time events which are used for measuring time and indirectly for monitoring activities on the platform. The Memory Service is intended for real-time memory allocation during runtime. The Communication Services make use of different communication protocols to send data remotely to other platforms.

C. Scaling OSA+ to low resource systems

By the micro-kernel 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 micro-kernel can be used stand-alone. 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. This configuration can be used on a simple embedded device that do not communicate remotely.
- adding the Process Service establishes the connection to the operating system. Thus, local procedural, lightweight and heavyweight services can be used. The configuration can be used on more complex embedded devices that need more controlling flows, but still no remote communication.
- combining the micro-kernel with the Process Service and the Communication Services allows local and global services to be executed. Devices using this configuration can be accessed remotely.
- combining the micro-kernel with the Event Service enables local procedural time-triggered services. This configuration can be used on simple devices that have time dependent tasks to perform.
- etc. Other configurations are possible, too.

It can be seen that using the micro-kernel architecture, the middleware can be configured with only the necessary parts. This is very important especially for embedded systems, where any additional overhead may hinder the application (see footprint evaluation results in section VI-A).

IV. Real-time in OSA+

As the middleware runs on top of other layers: hardware, operating system, communication system, a necessary condition for offering real-time capabilities to the user application is that all these layers must offer real-time capabilities. Once the condition is valid, the middleware has to manage three things:

- 1) the interface to the underlying environment, i.e. it must be able to make use of the real-time capabilities offered by it.
- 2) the middleware layer, i.e. the middleware algorithms must maintain the real-time characteristics, e.g. must be deterministic, with good worst case execution times (WCET).
- 3) the interface to the user application. The middleware is free to choose the abstractions to offer the real-time capabilities, abstractions that are available on all platforms where the middleware is ported.

Regarding the first point, OSA+ is gathering the real-time capabilities from the environment by each specialized service. The Process Service is able to use real-time scheduling schemes available on the operating system: Earliest Deadline First (EDF), Fixed Priority Preemptive (FPP), Guaranteed Percentage (GP), etc. The Memory Service is relying on the memory allocation techniques in real-time operating systems (RTOS), or can use the memory locking feature of the OS. The Communication Services can adjust the protocol communication buffers according with the messages that are sent remotely or it can use priorities when communicating over real-time communication protocols e.g. CAN [6].

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 and implementing the middleware. In OSA+ these are represented by:

- strict separation between initialization and operational functions. Initialization functions are responsible for resource allocation. They can be unpredictable and might not obey any timing constraints, e.g. `createConnection`, `releaseConnection`, `lookUpService`, `registerService`, etc. In contrast, the operational functions use the resources allocated by the initialization functions and must offer a constant and bound time behavior, e.g. `sendOrder`, `sendResult`, `existOrder`, `existResult`, etc.
- choosing algorithms and data structures which have a bound time and a good WCET. We are using priority heaps and general trees with constant and logarithmic complexities for the operations.
- sacrificing high-level programming techniques for efficiency. For example: reusing objects to avoid

run-time allocation or using list pointers instead of iterators.

Additionally, we are using a number of techniques with resolution especially for low resource systems, where every byte and clock cycle must be utilized (see [7]). Some of them we remind here:

- small interfaces [8] and strong design [9] principles, which affirms 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 handling QoS information (see section V).
- 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. The core platform shares information with the basic services. 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.

Currently, we offer real-time capabilities to user application in the form of priorities: fixed and time-based. However, we offer an easy mechanism to extend these capabilities as we will see in section V. The user can attach to every job a fixed priority which is used to order the job in internal priority queues. In this way jobs with higher priorities are processed first by the service. Additionally, we have also implemented a priority inheritance mechanism to avoid priority inversion. A lightweight service (a service implemented as thread) that receives a higher priority job than the one that is processing, has automatically its thread priority increased. Timing constraints can be present too, as release times and deadlines for job delivery and job completion only if the `EventService` is configured on the platform.

V. Quality of service

Quality of service support is mandatory for real-time applications. It enables the user to request desired QoS properties for its actions: deadlines, priorities, bandwidth for Internet connections, etc. As we are aiming to have an open middleware which can be easily adapted to different environments and different application requests, we realized a flexible support for the quality of service. Therefore, not only the typical real-time related capabilities can be requested by the user, but support for new QoS information can be easily added as we will see further.

A. Decoupling middleware implementation from QoS specification

One of our main design principles is to realize a loose coupling between the middleware components. This increases its flexibility, extendability and openness. This principle we followed for implementing the QoS management, too. As result, the middleware can receive QoS requests independently of its implementation, e.g. it can receive QoS requests even there is no support in the middleware to process them. This differs from other approaches where QoS support is strongly connected to the implementation, i.e. the user can request that through specific API existing in the middleware. In our case, this is done by simply attaching to each job a stream filled with QoS requests (see fig. 3). This enables the user to request multi-dimensional quality of service. This stream is further processed by the middleware and does not introduce any dependency to the implementation.

As the efficiency is a key factor for OSA+, the streamed QoS requests are simple $(key, value)$ pairs, where the *key* is a unique code for the QoS request and *value* is the corresponding value for it. As example, an order can contain as QoS request the key-value pair (10, 100). 10 is the QoS request code representing order execution deadline, and 100 is the value which is interpreted as time in milliseconds. That means that the respective order that contains this QoS request has to be processed within the deadline of 100 milliseconds. This mode to represent QoS information is very simple, efficient and flexible. However, on top of the actual representation of the QoS parameters, a layer can be easily added that could describe the QoS requests in more descriptive formats, e.g. the XML format. This layer can be added on more powerful platforms with enough resources to support it.

Another possible and easy to implement extension for the actual handling of the QoS requests, is to introduce simple support for logical and arithmetical operations, like: , e.g. $(2 < QoSvalue)AND(QoSvalue < 5)$. This allows a certain tolerance when requesting QoS.

B. QoS processing

The middleware uses for QoS processing, again, an *uniform approach*. It makes no difference between middleware directly supported QoS requests and the custom ones supported by the user. It provides the same means and access points (hooks) which can be used by middleware services or user application in order to add QoS support to the platform.

Thus, we divide QoS information in classes: time related, communication related, etc. Each class of QoS information is managed by a service, is identified by a unique integer and contains a range of QoS codes. For efficiency reasons, each class can manage a constant number of QoS codes, `QOS_CLASS_SIZE` (we choosed

100). A QoS parameter code belongs to the class with the identifier (QoS code / QOS_CLASS_SIZE) + 1. The service responsible to process a QoS class, will register a QoSHandler to the platform. This handler intermediates between the middleware and the service and it will be executed by the middleware every time a job containing QoS information belonging to its class is received (see fig. 3). The middleware will pass to the QoSHandler the job received and the requested QoS parameters. As example, the Event Service is respon-

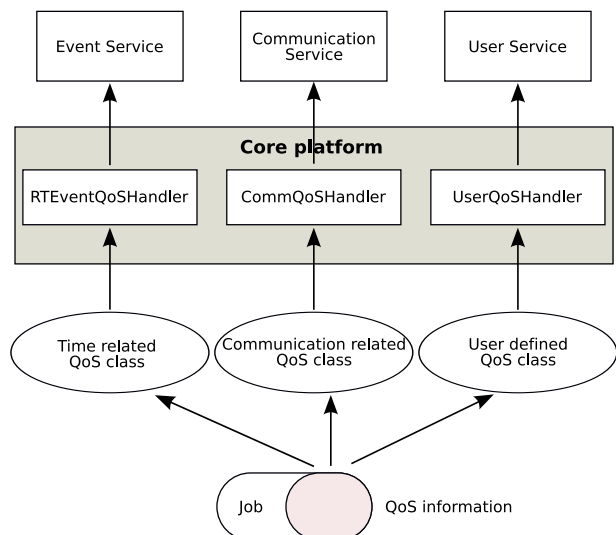


Fig. 3. Flexible QoS management

sible for the time related QoS information class. It initiates and monitors 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 registers to the platform a QoSHandler using the core function: `addQoSHandler(RTEventClass, RTEventQoSHandler)`. Where, `RTEventClass` is an integer representing the QoS class of parameters for real-time events (we defined it 1) and `RTEventQoSHandler` is the handler implemented by the Event Service. Regarding the previous example with the QoS key-value pair request (10, 100), the 10 QoS parameter code belongs to the QoS class with identifier 1 (is in the range of the first 100 codes), which is the `RTEventClass`.

With this simple and flexible mechanism, new classes of QoS information can be easily added to the platform. The process implies implementing the services that will process them and using the middleware hook (the QoSHandler) to get informed when jobs that contain the respective QoS arrive in the middleware.

Additionally, the middleware distinguishes between QoS requests that have resolution on:

- a session basis, applied at initialization time when the session is established. They are not anymore

processed and introduce no overhead once the messages start to be exchanged, e.g. communication buffer sizes.

- a message basis, applied at execution time when orders are exchanged between services. The QoS requests are processed for each message.

To allow this separation between the QoS processing times, the QoSHandler interface provides two corresponding methods pairs: `onSession(QoSCode)`, `onMessage(QoSCode)` and `onSession(Job)`, `onMessage(Job)`. The first methods pair informs the middleware for the processing time of a QoS code, and the second pair is used by the middleware to pass the Job and the QoS information to the handler.

VI. Evaluation

We have realized implementations of OSA+ middleware in C and Java. However, we will present here only evaluation results for the Java implementation as this is more complete.

A. Small 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 I, 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 using an additional tool [10] on the classes generated at the previous step. This tool makes code size optimizations for Java class files.

| | Minimum | Normal |
|-----------------|---------|--------|
| OSA+ Core | 28628 | 52705 |
| Process Service | 3316 | 5084 |
| HLC Service | 5695 | 9561 |
| TCPIP Service | 5992 | 7573 |
| Init Service | 880 | 1046 |
| Total | 44511 | 75969 |

TABLE I
OSA+ FOOTPRINT(IN BYTES)

Where, HLC Service and TCPIP Service are part of the Communication Services and are used for remote communication.

As it can be seen from the table, the minimum version has a footprint of about 44 kBytes including the TCP/IP communication, which is, to our knowledge, the smallest footprint for a fully configurable real-time middleware. Furthermore, for simple embedded devices that do not need remote communication (only the core and the Init Service are needed) the footprint is 29 kBytes.

B. Real-time

For evaluating the real-time capabilities, we have explored the determinism of our middleware. The idea is to measure the time for sending a single order and to analyze if there are peaks in the measured values. If they exist in a real-time environment, it means that our middleware behaves not deterministic under given circumstances. These tests were made on a local platform, as we do not have yet implemented a service for a real-time communication protocol. The hardware platform was a PC with a PentiumM 4 at 1800 MHz processor and 512 MB RAM. All the evaluations were running in console mode, without any other processes on the system except the application. Two services were exchanging 1000 orders of 32 bytes sizes. The test was executed 10 times and we computed the following statistics: minimum, maximum, average and standard deviation(stdev) for each test.

We have made the test on several real-time operating systems (RTOS): TimeSys Linux GPL 4.1 based on the Linux kernel 2.4.21 (see [11]), TimeSys Linux GPL 5.0 based on the Linux kernel 2.6.0 and MontaVista 3.1 based on the Linux kernel 2.4.20 (see [12]). For comparison we have used also a non RTOS: RedHat 9.0 based on the Linux kernel 2.4.20 [13].

The JVM used was the reference implementation of the real-time specification for java, version 1.1 (see [14]).

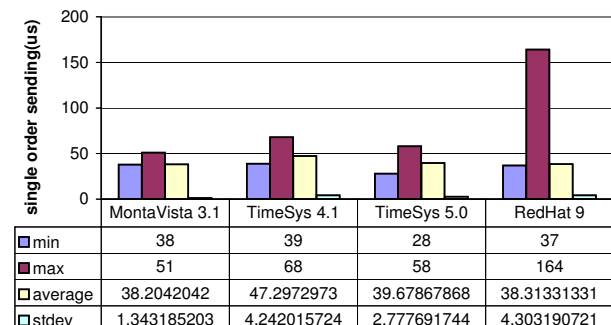


Fig. 4. Real-time evaluation

As it can be seen from figure 4, the results for all RTOSes are very good regarding determinism. There are low values for stdev, i.e. a small variance in the results, and more important the maximum peek (the difference between two measured values) is only 30 microseconds. Even on the non RTOS the results are quite good regarding the stdev, but the peek here has 127 microseconds. This can be explained by the fact that the test was done on an idle system and there were no other influences from other processes. However, there is no guarantee that we will have the same results when the non RTOS is loaded. An interesting observation is that no garbage collection activity took place once the orders started to be exchanged. This explains that our principle, to avoid as much as possible dynamic allocation and

reserve resources only in the initialization functions, was successfully implemented.

As conclusion, the evaluations shows that our middleware approach behaves predictable if it is based on a predictable environment. If only one component fails in providing real-time capabilities, the whole setup fails with this respect.

VII. Conclusions

We have presented in this article a middleware suitable for DRE systems. We have described the principles followed for designing and implementing it. The combination of a micro-kernel architecture with a service based middleware proved to be open (flexible, easy user customizable) and highly adjustable for DRE systems where only the needed functionalities can be selected for the middleware. The evaluation results for the middleware showed that the footprint is quite small, approx. 44 Kb, which is the smallest compared with other middleware. Furthermore, the middleware offers real-time support in terms of fixed priorities and time based priorities if configured with the EventService. It behaves deterministic when it runs in real-time environments.

Regarding QoS information management, we have an uniform framework that decouples the application from the specification and implementation. It allows the QoS attributes of the distributed OSA+ middleware to be customized transparently for the user application. Multi-dimensional quality of service can be easily requested by the user by just simply attaching the desired QoS requests to the messages that are exchanged between the active parts of the middleware - the services.

REFERENCES

- [1] "Common Object Request Broker Architecture: Core Specification 3.0.3," Object Management Group, Tech. Rep. formal/2004-03-01, March 2004.
- [2] "ROFES: Real-Time CORBA for embedded systems," Available: <http://www.lfbs.rwth-aachen.de/content/20> (Accessed: 2004, December 10).
- [3] F. Picioroaga, "Scalable and efficient middleware for real-time embedded systems. a uniform open service oriented, microkernel based architecture," Ph.D. dissertation, University Louis Pasteur Strasbourg, 2004.
- [4] J. He, "Customizable multi-dimensional qos in distributed systems," Ph.D. dissertation, University of Arizona, 2004.
- [5] M. Hiltunen, R. Schlichting, and G. Wong, "The Cactus Project," <http://www.cs.arizona.edu/cactus/index.html>.
- [6] "International Standards Organisation. Road Vehicles - Interchange of Digital Information - Controller Area Network (CAN) for High-Speed Communication," 1993, ISO 11898.
- [7] J. Noble and C. Weir, *Small Memory software*. Pearson Education Limited, 2001.
- [8] B. Meyer, *Object oriented software construction*, 2nd ed. Prentice Hall PTR, March 2000.
- [9] J. Coplien, *Advanced C++ programming styles and idioms*. Addison-Wesley Professional, 1994.
- [10] E. Lafortune, "ProGuard," <http://proguard.sourceforge.net>, 2004.
- [11] "TimeSys Linux GPL," <http://www.timesys.com>.
- [12] "MontaVista Linux," <http://www.mvista.com>.
- [13] "RedHat Linux 9," <http://www.redhat.com>.
- [14] "The Real-Time Specification for Java," <https://rtsj.dev.java.net/rtsj-V1.0.pdf>.