A Microkernel Architecture for a Highly Scalable Real-Time Middleware

U. Brinkschulte, C. Krakowski, J. Riemschneider
Institute of Process Control,
Automation and Robotics
University of Karlsruhe
D-76128 Karlsruhe, Germany

J. Kreuzinger, M. Pfeffer, T. Ungerer
Institute of Computer Design
and Fault Tolerance
University of Karlsruhe
D-76128 Karlsruhe, Germany

Abstract

Middleware systems support the development of distributed real-time systems in particular in heterogeneous environments. We propose OSA+, a scalable service-oriented middleware architecture. OSA+ offers support not only for communication but also for the execution of tasks. It adapts to different hardware, operating and communication systems by use of basic services. These services can be plugged in a microkernel platform. Thus, OSA+ can be used on PCs, workstations, but also in lower-priced embedded systems with scarce memory and CPU equipment. As case study for an OSA+ application, we present the Komodo project which consists of a multithreaded Java microcontroller, a real-time Java virtual machine, an adapted Java API, and a Java version of the OSA+ middleware.

1 Introduction

The complexity of real-time systems in the field of system automation is steadily increasing. Complex embedded real-time systems consist of several computing nodes ranging from DSPs and microcontrollers to PCs and workstations. Using middleware for such real-time systems offers several advantages: it eases the software development, increases the portability of the software and improves the system maintenance and reliability. Furthermore, the middleware can be used to monitor timing constraints in a distributed real-time system.

The design of such a middleware must reflect the special needs of real-time systems, e.g. predictability, definition of timing constraints or resource allocation. Embedded real-time systems have additional needs: many of the computation nodes in such systems are strictly limited in memory and processor power, because they are realized by cheap microcontrollers. So the overhead produced by the middleware must be minimized for such nodes. To solve this problem, we propose a scalable middleware architecture for embedded real-time systems called OSA+[1]. This architecture adapts the microkernel concept to middleware. Scalability is achieved by a very small system core and additional functionality, which can be added on demand. To deal with the variety of features given by the hardware and software environment, we rely on a quality of service (QoS) concept.

2 Basic OSA+ architecture

OSA+ provides a service oriented architecture. Services are plugged into a platform and communicate with each other through jobs (Figure 1). A job consists of an order and a result.

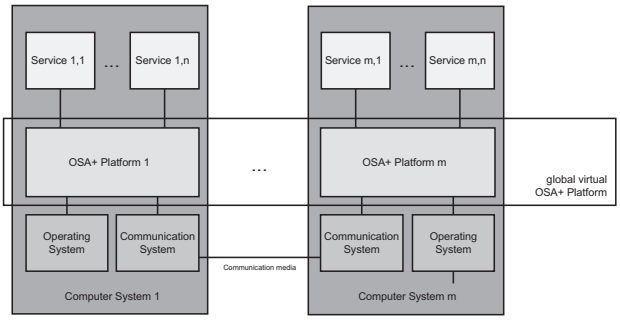

Figure 1. OSA+ basic architecture

One main goal of OSA+ is the realization of a highly scalable architecture, which can easily be adapted to different hardware and software environments. 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 contains no hardware or operating system dependent parts. The core platform uses special services to

[1] OSA+: Open System Architecture - Platform for Universal Services

extend its own functionality. These special services are *basic services* which are used for the adaptation to a specific hardware and operating system environment, and *extension services* which extend the core functionality of the platform (Figure 2). These services are plugged in like user services.

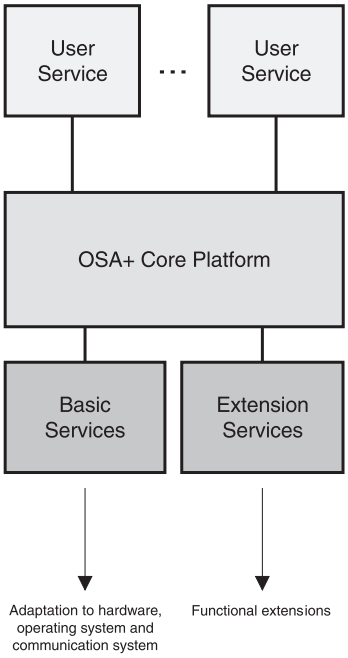

Figure 2. OSA+ microkernel architecture

2.1 The core platform

The core platform is the component seen by the users of the OSA+ middleware. It offers functionality which allows to plug in and remove services as well as to deliver jobs. The platform consists of a service management, a job management and a user interface. The task of the service management is to install, remove and locate services. Services can be local or global. Local services are only known to their platform. Global services are known to the whole system. The job management allows to deliver jobs between services. It offers the following functionality: establishing connections between services, creation, parameterization and destruction of jobs, synchronous and asynchronous sending and receiving of orders and results, scheduling of jobs and keeping track of the job's state and timing. A simple user interface (about 14 basic functions) provides access to the functionality of the service and job management.

2.2 Basic and extension services

The core platform is able to work standalone on a very small system. The functionality is than restricted to: only local jobs and services, a limited number and size of jobs and services, strictly sequential job management (no multithreading or multitasking) and no real-time monitoring (see section 3). To adapt and scale the platform to a target environment, the basic and extension services are used. These are services with a predefined interface, which are plugged in like user services. The following basic service interfaces are currently defined: the *memory service* gives access to the OS memory management and removes the restriction of limited numbers and sizes of jobs and services. The *process service* gives access to the OS process management and allows the use of multithreading and multitasking to realize the services as threads or heavy weight processes. The *communication service* is the link to the available communication systems and allows global jobs and services. The *event service* manages timer and other hardware events. It allows the monitoring of real-time constraints (see section 3) and event triggered job execution (see section 4). The aim of the extension services is to add new functionality to the platform, that might be useful for certain applications. Typical examples are a *protocol service* and a *security service*. The former allows the platform (and all other services) to protocol internal activities (e.g. job delivery, plug in of services). The latter can limit access to services and encipher data for secure transmission. So, with basic and extension services, a tradeoff can be made between size and functionality of the OSA+ platform.

3 Real-time issues

This chapter first describes the real-time job scheduling, as it is seen by the application. Then we explain, how this is realized by the core platform and the basic services.

3.1 Job scheduling

The job management presented in section 2.1 must guarantee that all deadlines are met in hard real-time systems. To achieve this, the user can specify real-time constraints for every job defined. In this way the application programmer has better control over the execution of jobs. As mentioned in section 2 a job consists of two parts: the order and the result. The job timing constraints apply to both parts. All possible job states, the actions which can initiate a transition from one state to another and the relevant timing constraints are depicted in Figure 3.

Note that all timing specifications are optional. The user can specify a subset of these constraints which are interesting for his application. Instead of the 'latest end of execution' time a fixed priority or a *guaranteed percentage* rate for every job can be specified. Guaranteed percentage scheduling is supported by the Komodo microcontroller (see section 4). Here each job gets a guaranteed percent-

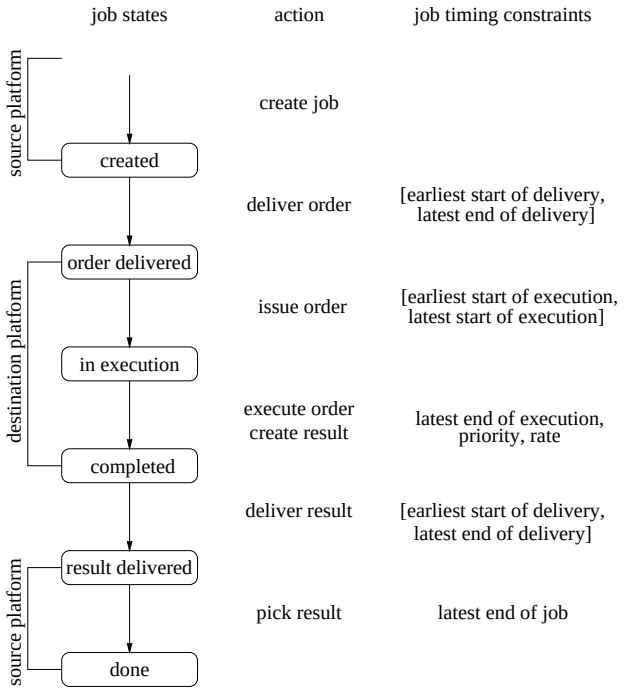

Figure 3. Job timing constraints: deliver and execution phase

age rate of the totally available processor cycles [1]. Because many real-time applications use periodic jobs (e.g. for the input of sensor data) a *period* for the repeated delivery of jobs can also be specified. So the priorities, deadlines or rates of a service (thread, process) which executes a job, are directly controlled by this job. The various timing constraints allow a concise definition of scheduling needs, monitoring and load transfer between client and server. A description of the use of these constraints can be found in a long version of this paper [11].

3.2 Platform mechanisms

The real-time characteristics of the middleware platform depend on the features of the underlying hardware and OS. So the realization of the various timing constraints given in section 3.1 is spread over the core platform and the basic services, which form the connection to hardware and OS.

The *core platform* offers the following basic real-time mechanisms: first, the user functions are divided into initialization functions and working function. All working functions use static resources pre-allocated by the initialization functions and offer strictly limited worst case execution times (WCETs). Second, the core platform uses the 'latest times' or 'priorities' given in job scheduling to order the jobs in the internal queues. So an urgent job overtakes a job with more laxity. All the other real-time mechanisms depend on hardware and OS features and are therefore placed in the basic services. Each basic service offers a QoS function, which reports the supported real-time features to the core-platform.

The *memory service* can offer the real-time features 'locking' and 'real-time allocation'. If none of them is present, the memory service isn't used for real-time jobs. If 'locking' is available, the buffers of the core platform can be extended to increase the number and size of real-time jobs and services. If 'real-time allocation' is offered, dynamic-sized real-time jobs are allowed (no pre-allocation is needed).

The *process service* offers real-time scheduling policies available from the OS for processes and threads, which realize the services. These policies are used in the execution phase of the job scheduling. Currently 'fixed priority', 'EDF' and 'guaranteed percentage' scheduling can be supported. If no real-time policy is offered, we allow no real-time jobs.

The *communication service* can offer real-time protocols, which allow guaranteed worst case transmission times and prioritized transmissions between platform nodes. Explicit support for asynchronous communication is given by the OSA+ user interface. If no real-time communication is available, only local real-time jobs are allowed.

The *event service* is the basis for real-time monitoring and release scheduling. The timer events offered by the event service are used to trigger the jobs based on the 'earliest times' given in job scheduling. Furthermore, all 'latest times' are monitored by timer events. As shown in the next section, the event service can also be used to trigger real-time jobs based on external hardware events.

4 A case study: the Komodo project

We are currently implementing a first prototype of OSA+ on top of Windows NT and CMX. The following case study exemplifies the adaptation of the OSA+ middleware architecture to an underlying non-standard hardware and software environment.

The Komodo project explores the suitability of Java and a multithreaded processor in embedded real-time systems.

A multithreaded processor is characterized by the ability to simultaneously execute instructions of different threads within the processor pipeline [2], which allows an extremely fast context switch. Multithreading techniques are proposed in processor architecture research to mask latencies of instructions. The Komodo project applies multithreading for event handling by taking advantage of its fast context switching ability. Contemporary microcontrollers activate Interrupt-Service-Routines (ISRs) for event handling. Assuming a multithreaded processor core in a microcontroller allows to trigger so-called Interrupt-Service-

Threads (ISTs) instead of ISRs for event handling. In this case an occurring event activates an assigned thread. This has several advantages, like extremely fast thread activation, non-blocking parallel thread execution, hiding of latencies, flexible context switches and flexible priority schemes for interrupt handling. A detailed description of the IST concept can be found in [10].

State-of-the-art embedded systems are programmed in assembly or C/C++ languages. Java features several advantages with respect to real-time systems: the object orientation of the Java language supports easy programming, reusability, and robustness. Java bytecode is portable, of small size, and secure due to the bytecode verifier in the Java-Virtual-Machine (JVM). However, the use of Java is hardly possible in embedded real-time systems because of its high hardware requirements and unpredictable real-time behavior. Both problems are solved with the design of a real-time, multithreaded Java-microcontroller.

The layers of the Komodo project are depicted in Figure 4 (for more details see [3] [1]).

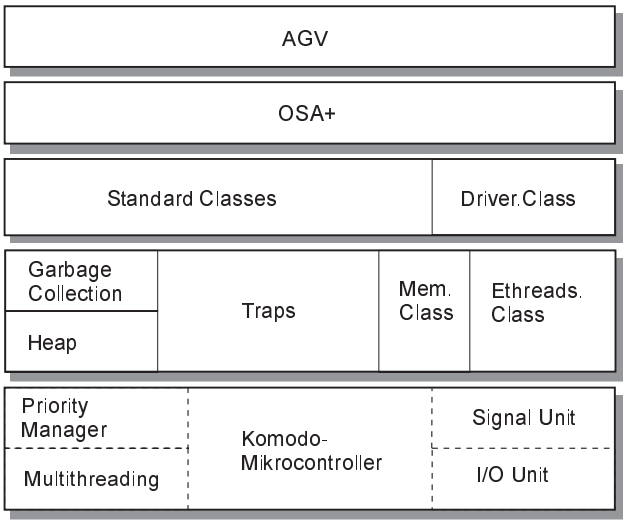

Figure 4. The Komodo project layers

The first layer—the core of the project—is a Java microcontroller with the following features: a Java-microcontroller executes Java bytecode instructions directly in hardware raising the performance of Java applications and decreasing the hardware requirements compared to a Java interpreter or Just-In-Time (JIT) compiler. The proposed Komodo microcontroller supports multiple ISTs with zero-cycle context switching overhead. It holds the contexts of up to four hardware threads in different register sets. Several thread priority schemes can be managed in hardware, mainly EDF and guaranteed percentage. The guaranteed percentage scheme offers the possibility to assign a rate of the full processing power to each thread of a system of threads. Herewith response times and data rates can be guaranteed even for several concurrent events independent of other processor activities, as long as there is no overload condition.

The second layer consists of the Komodo Virtual Machine (VM) which is a real-time virtual machine used by the Komodo API. Most of the third layer, the Komodo API, is derived from the Java API which is adapted to the Komodo VM and extended by special classes for real-time thread handling and the system I/O.

The fourth layer of the project is the OSA+ middleware. This has two reasons: first, the Komodo microcontroller supports up to four hardware real-time threads. If more real-time threads are needed, several Komodo microcontrollers can be interconnected by the middleware. So the middleware scales the Komodo system. The second reason for using OSA+ is that we want to interconnect the Komodo controller with other non-Java microcontrollers and microprocessors.

To adapt the middleware to the Komodo system, the basic services must be provided and combined with the standard Java version of the microkernel core platform: the *process service* maps the OSA+ service model to the real-time hardware threads of the Komodo microcontroller. Installing an OSA+ service is mapped to the creation of a Komodo hardware thread. The hardware EDF and guaranteed percentage scheduling policies are made available to the core platform. The *event service* is used to map the hardware event handling mechanism of the Komodo microcontroller to the OSA+ service model. A hardware event is connected to an OSA+ job, which is directed to an OSA+ service. This service is mapped to an IST and placed in one of the four hardware thread slots of the Komodo microcontroller. If an event occurs, it activates the IST and executes the predefined job concurrently to other real-time or non-real-time threads. After the execution the IST is suspended and waits for the next event. The *communication service* currently simply maps the protocol layer of the communication service to an serial peripheral interface (SPI) similar to those available in Motorola microcontrollers. The *memory service* is currently omitted, because there is only automatic memory allocation in Java.

The fifth layer is the real-time application layer—currently an automated guided vehicle (AGV) system is envisioned as pilot application and described in a longer version of this paper [11].

5 Related work and conclusions

Middleware systems like CORBA [4], DCOM [5], RMI [6] and JMS [7] are increasingly popular in the non-real-time world. But these systems are not suitable for real-time applications. They require a large amount of memory and do not take advantage of most of the operating system's real-time features [8].

For these reasons the Real-Time Special Interest Group (RT-SIG) of the Object Management Group (OMG) started the definition of a real-time CORBA standard [9]. RT CORBA eliminates the above mentioned disadvantages and makes CORBA suitable for the application in real-time systems. Some requirements of RT CORBA are: 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. Several RT CORBA realizations from stripped-down, faster versions of existing ORBs to more advanced solutions like MITRE's RT CORBA, TAO, CHORUS/COOL RT CORBA or NRaD/URI RT CORBA are described in [8].

There are several differences between our proposed middleware system and RT CORBA. OSA+ is a service oriented system while RT CORBA is object oriented. However, OSA+ is more than a pure message-oriented system like JMS, because it offers support not only for communication but also for the execution of tasks. RT CORBA is only a specification. It says nothing about the implementation and not much about the architecture of the ORB. In contrast, the OSA+ proposes a specific broker architecture, which is based on the microkernel concept and scales from small embedded systems to larger systems. OSA+ was designed for small embedded systems from scratch while RT CORBA is the extension of a standard which was not designed for real-time embedded systems initially. OSA+ can be seen as a set of components for constructing ORBs of different sizes. Internal studies showed that OSA+ can be used to build a RT CORBA ORB.

We are currently implementing OSA+ on top of Windows NT and CMX. So we hope to get soon first evaluation information about performance and overhead. We are further integrating OSA+ into the Komodo project based on the multithreaded Komodo microcontroller, which currently exists as software simulation. A complete Komodo system with a FPGA based controller prototype is under construction.

References

[1] U. Brinkschulte, C. Krakowski, J. Kreuzinger, T. Ungerer. A Multithreaded Java Microcontroller for Thread-oriented Real-time Event-Handling. 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT '99), Newport Beach, Ca., 12.-16.10.1999, pp. 34-39.

[2] J. Silc, B. Robic, and T. Ungerer. Processor Architecture: From Dataflow to Superscalar and Beyond. Springer-Verlag, Heidelberg, 1999.

[3] U. Brinkschulte, C. Krakowski, J. Kreuzinger, R. Marston, T. Ungerer. The Komodo Project: Thread-Based Event Handling Supported by a Multithreaded Java Microcontroller. Proceedings of the 25th EUROMICRO Conference, Volume 1, Milan, Italy, September 8-10, 1999.

[4] Object Management Group. The Common Object Request Broker: Architecture and Specification. 2.2 ed., Feb. 1998.

[5] Microsoft Corporation. DCOM Architecture. http://microsoft.com/com/comPapers.asp.

[6] T. B. Downing. Java RMI: Remote Method Invocation. IDG Books Worldwide, February, 1998.

[7] Sun Microsystems. The Java Message Service (JMS) API. http://java.sun.com/products/jms/index.html.

[8] V. F. Wolfe, L. C. DiPippo, R. Ginis, M. Squadrito, S. Wohlever, I. Zykh. Real-Time CORBA. Proceedings of the Real-Time Technology and Applications Symposium, June 9-11, 1997, Montreal, Canada, pp. 148-157.

[9] Object Management Group. Realtime CORBA 1.0 Joint Submission. OMG Document orbos/98-11-03, 1998.

[10] U. Brinkschulte, C. Krakowski, J. Kreuzinger, T. Ungerer. Interrupt Service Threads - A New Approach to Handle Multiple Hard Real-Time Events on a Multithreaded Microcontroller. Real-Time Systems Symposium, WIP Proceedings, December 1-3, 1999, Phoenix, Arizona, pp. 11-15.

[11] U. Brinkschulte, C. Krakowski, J.Riemschneider, J. Kreuzinger, M. Pfeffer, T. Ungerer. The OSA+ Middleware Architecture. IPR Technical Paper, 1999, http://www-ima.ira.uka.de/mitarbeiter/krakow/osa+arch.ps.