The COMQUAD Component Container Architecture

Steffen Göbel Christoph Pohl
Ronald Aigner Martin Pohlack
Institute for System Architecture
TU Dresden, Germany
{goebel|pohl}@rn.inf.tu-dresden.de
{aigner|pohlack}@os.inf.tu-dresden.de

Simone Röttger Steffen Zschaler
Institute for Software Engineering
TU Dresden, Germany
{Simone.Roettger|Steffen.Zschaler}
@inf.tu-dresden.de

Abstract

Component-based applications require runtime support to be able to guarantee non-functional properties. This paper proposes an architecture for a real-time-capable, component-based runtime environment, which allows to separate non-functional and functional concerns in component-based software development. The architecture is presented with particular focus on the real-time–non-real-time split of the runtime environment and the communication issues of respective component types and container parts.

1. Introduction

Considering non-functional properties of a system, such as Quality of Service (QoS) or security aspects, is crucial for reliable software systems. Apart from explicit specification at design time, this also includes implicit consideration at implementation level and adequate runtime support. In this paper, we introduce the COMQUAD container architecture, which provides a runtime environment for QoS-capable, component-based software applications.

Component models are typically implemented by *application servers* containing all necessary infrastructural services of the component runtime environment. The term *container* is often used to refer only to the immediate execution shell of component instances. In contrast, our notion of a *container* comprises all major parts of component management. Our components are black-box elements, which implement business logic and cooperate with other components to solve an application requirement. One of the main features of our component model is that—corresponding to ideas presented by Cheesman and Daniels [1]—more than one implementation can be provided for one functional specification. This allows to provide the same functionality with different non-functional properties, and thus serves to separate functional and non-functional concerns.

The internal architecture of component-based software applications is usually captured by descriptive means of *architecture description languages* (ADL, [10]). We use *assembly descriptors* as a part of our COMQUAD component model [4] for this purpose. Classic ADLs describe connections between components at the level of instances of specific component implementations whereas our assembly descriptors leave implementation selection to the container. This is done by defining only how instances of certain functional component specifications are to be connected without referencing any implementations. Thus, the container can select the appropriate implementations for these component specifications based on their non-functional properties and according to its current client's needs.

For many applications where QoS is an issue it is natural to separate mission-critical real-time (RT) code from lower priority non-real-time (NRT) code. A Video on Demand (VoD) application is a typical example. Only the actual delivery of movies needs to provide guarantees of non-functional properties. The much larger part of the application dealing with management of customers and movies, selection of, and payment for, movies by customers, advertisement, bonus actions, etc. is typically not so critical in terms of its non-functional properties. This realization was the driving force behind our decision for a split container architecture that handles both aspects separately. Such a separation, however, brings about additional challenges—in particular with respect to the communication between the two parts. This paper focuses on describing these challenges and our solutions for them.

The remainder of this paper is organized as follows: Section 2 provides a high-level view on our container architecture as well as the reasons for the RT–NRT split. Section 3 describes the necessary communication primitives between both parts. Finally, we give an overview of related work and conclude with an outlook. An extended version of this pa-

per is available as a technical report [3].

2. Split Architecture

Typically, applications that give guarantees on non-functional properties are split into two parts: *(i)* a small part of code that performs the actions for which guarantees of non-functional properties are essential and *(ii)* a usually much larger part for which non-functional guarantees are not important.

This distinction can be applied to the architecture of a component runtime environment. Figure 1 gives an overview of our architecture consisting of two layers: the component manager and the resource management subsystem. The component manager itself consists of three subsystems: implementation manager, QoS repository, and contract manager. The implementation manager supports the life cycle of component implementations and the QoS repository stores the non-functional specification of each component implementation. The parts that need to give guarantees (RT) and the parts that do not (NRT) have been clearly separated. Most of the work is done in the NRT part of the component environment. It manages the available component implementations and application specifications, handles requests for creation of component networks, negotiates contracts between components, and maintains a model of the instantiated components and the networks formed by them. The RT part is essentially restricted to actually instantiating components, reserving resources, and executing the instantiated components in response to user requests. The contract manager instantiates and connects the components required to service a particular client's requests. In order to select the correct components and component implementations, the contract manager uses the functional and non-functional component specifications provided in XML-based descriptors [4, 3].

We base our system on DROPS (Dresden Real-time OPerating System, [6]) where we have small real-time capable server processes for dedicated tasks—for example, a window manager, a SCSI disk driver, and network drivers. These servers run directly on our real-time capable microkernel. Complex legacy software without real-time requirements runs concurrently on a large off-the-shelf Linux server (L^4Linux, [7]).

For the NRT container we use the infrastructure of a stripped down JBoss container [2] and add support for the new COMQUAD component model [4] together with necessary services, such as contract negotiation, administration, as well as implementation and component management. The NRT container exclusively handles the deployment of component archives including integrity checks and the initialization phase of components. Whenever a client wants to create a component instance, the NRT container receives and processes the required `create` call of the component's home interface. It also starts the contract negotiation phase, which is discussed in more detail in [3], and sends control commands to the RT container. Afterwards, the client directly talks to the RT container.

For the RT container we have identified a minimal set of necessary services. It contains a simple instance repository, communication infrastructure (cf. Sect. 3), resource managers, and a small framework for components, consisting of interfaces and base classes for component instances, and helper functions. The RT container must be able to manage the component life cycle (`create`, `destroy`, `initialize`, `connect`, `stop`, `setParameter`, `reserveResources`, and `install/uninstall Specification/Implementation`). Additionally, the RT container must allow to invoke operations (`callMethod`) on component instances. Fortunately, it is not necessary to implement all of these functions with RT guarantees. We focus on RT communication between connected components, not the establishing of communication structures and setup in RT. Thus, only the `callMethod` operation must be carried out in RT.

DROPS resource managers as described in [8] are organized in a resource management subsystem (see Fig. 1), which is governed by a *QoS manager*. To be able to manage different kinds of resources, the resource managers implement a generic *admission interface*, which is used by the QoS manager to make reservations. In case a reservation is violated—for instance, because a higher priority reservation has been made—the container is notified via its *notification interface*. The container then initiates the adaptation

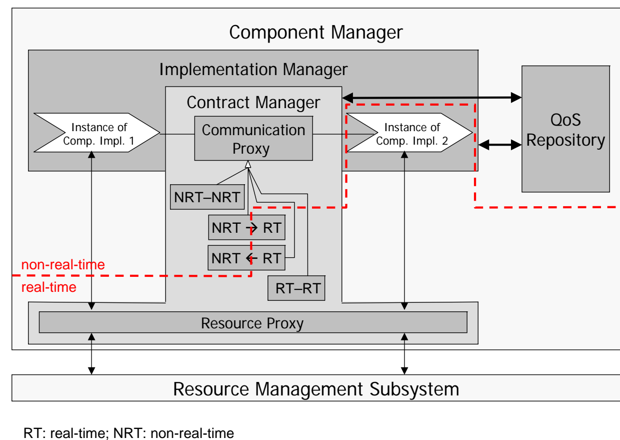

Figure 1: COMQUAD container architecture

of all components using the considered resource.

The most noteworthy consequence of this split in the architecture can be seen in the communication proxies. This is the subject of the next section.

3. Component Communication

The architecture split described in Sect. 2 implies two types of components: real-time and non-real-time. As a consequence, four constellations of communication can occur:

NRT to NRT. This communication uses the infrastructure provided by the JBoss-based container only.

NRT to RT. The communication crosses container boundaries. Therefore, we use a dynamic proxy to intercept and delegate the message to a specialized invocation handler. The message is transferred to the RT container via a generic bridge. In the RT container a generated demultiplexing container skeleton delivers the message to the addressed interface skeleton. On the RT side we do not use a Dynamic Invocation Interface but generated code instead, which is larger but faster [5]. Communication with RT components requires a reservation and the invoking component's communication pattern has to conform to the reservation. Nonconforming communication can be delayed or dropped by the RT container.

RT to RT. Communication between RT components can be scheduled entirely by the RT container, because their specification contains information about the load generated.

RT to NRT. There are cases when RT components use complex services provided by NRT components. However, synchronous communication is not suitable in this case, because it could delay the RT components for an unbounded amount of time. The alternative of asynchronous communication requires message buffers. It is noteworthy that it is basically impossible to communicate from the RT side to the NRT side without the possibility of message loss. The thorough discussion of this communication constellation is subject of another publication [13], where we propose a generic buffer component, which implements replacement strategies and handles request delivery on behalf of the sender. Using this generic buffer component it is possible to transparently connect RT and NRT components.

4. Related Work

The OMG's CORBA Component Model (CCM) forms the basis for many functional concepts of our component model, but it does not address special problems related to non-functional properties—for instance, dynamic selection of implementations at runtime. Just like Sun's Enterprise JavaBeans (EJB) component model, CCM supports only a limited, fixed set of non-functional aspects like persistence, access control, transactions, etc.

A fundamental building block of our component container implementation is formed by the extensible JBoss application server [2]. It features a variant of Interceptors that forms the foundation for the resource and communication proxies in our component platform architecture.

The project QuA [15] aims at precisely defining an abstract component architecture, including the semantics for general QoS specifications. There are some differences to our approach: First of all, we do not only consider QoS in terms of timeliness and accuracy of output but also with respect to other non-functional properties such as security aspects. While the abstract QuA architecture could theoretically be implemented on top of any real-time-capable combination of operating system and middleware, our approach is closely tied to DROPS [6]. This allows us to fully leverage the virtues of this platform—for instance, its clean microkernel architecture.

CIAO [16], another related project, builds a QoS-enabled CCM implementation. The project's philosophy is a strong adherence to existing OMG specifications such as RT/CORBA [12] and CCM, and the extension of those. In contrast, we decided to focus on the challenges of supporting non-functional properties. Hence, we have tried to keep the functional part of our component model as lean as possible while still adopting tried and tested concepts.

The Real-Time Specification for Java (RTSJ) [14] introduces the concepts of timeliness, schedulability, and real-time synchronization to Java-based applications. One of the biggest challenges in this connection is to prevent the garbage collector of Java's memory management to interfere with real-time task scheduling. However, resource reservation is not addressed by this specification, which would prevent an implementation of our concepts on top of this platform.

Requirements for real-time extensions for Java were defined in the NIST report [11]. The NIST group proposes partitioning the execution environment into a real-time core providing the basic real-time functionality and a traditional JVM, which services normal Java applications. Based on these requirements, the J Consortium defined the Real-Time Core Extensions for Java (RTCE) [9], which follow the idea of a separate core for real-time services. In contrast, in RTSJ all services are provided in one JVM, as such containing the real-time and the non-real-time applications. The architectural RTCE approach is similar to the design of DROPS, in that both run large and complex parts in a classic non-real-time environment and only small, predictable parts in a real-time environment.

5. Conclusions and Outlook

Our paper proposed an architecture for a real-time-capable, component-based runtime environment, building on concepts for separating non-functional and functional concerns in component-based system development [4]. We furthermore introduced a prototypical implementation of this architecture, and explained various special issues thereof.

In detail, we presented the conceptual split of our COMQUAD component container into a lean real-time part and a larger non-real-time part. The former is capable of giving guarantees by enforcing resource reservations, whereas the latter part has been built for running less time-critical code, including contract negotiation for real-time components. Thus, our container is an application server that acts as a contract manager selecting component implementations to be instantiated for application assembly at runtime.

We are currently investigating different implementations of contract negotiation algorithms. These are needed by the contract manager to select component implementations and profiles based on non-functional requirements of clients. We refer to this selection process as *Container-Managed QoS* [4].

A special case of inter-component communication is stream-based communication. This kind of communication frequently has associated non-functional requirements, the classic example being a VoD service. More details on stream support in our component model can be found in [4].

Acknowledgements

COMQUAD—**Com**ponents with **Qu**antitative properties and **Ad**aptivity—is a DFG-funded research group (FOR 428) at Technische Universität Dresden. See `http://www.comquad.org/` for details.

References

[1] J. Cheesman and J. Daniels. *UML Components: A Simple Process for Specifying Component-Based Software*. Addison Wesley Longman, 2001.

[2] M. Fleury and F. Reverbel. The JBoss extensible server. In M. Endler and D. Schmidt, editors, *International Middleware Conference*, volume 2672 of *Lecture Notes in Computer Science*, pages 344–373, Rio de Janeiro, Brazil, 16–20 June 2003. ACM / IFIP / USENIX, Springer.

[3] S. Göbel, C. Pohl, R. Aigner, M. Pohlack, S. Röttger, and S. Zschaler. The COMQUAD component container architecture and contract negotiation. Technical Report TUD-FI04-04, Technische Universität Dresden, Apr. 2004.

[4] S. Göbel, C. Pohl, S. Röttger, and S. Zschaler. The COMQUAD component model – enabling dynamic selection of implementations by weaving non-functional aspects. In K. Lieberherr, editor, *3rd International Conference on Aspect-Oriented Software Development (AOSD'04)*, pages 74–82, Lancaster, UK, 22–26 Mar. 2004. ACM Press.

[5] A. Gokhale and D. Schmidt. The performance of the CORBA dynamic invocation interface and dynamic skeleton interface over high-speed ATM networks. In *Global Telecommunications Conference (GLOBECOM '96)*, pages 50–56, London, England, Nov. 1996. IEEE.

[6] H. Härtig, R. Baumgartl, M. Borriss, C.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, and J. Wolter. DROPS: OS support for distributed multimedia applications. In *8th European Workshop on Support for Composing Distributed Applications*, Sintra, Portugal, Sept. 1998. ACM SIGOPS.

[7] H. Härtig, M. Hohmuth, and J. Wolter. Taming Linux. In *5th Annual Australasian Conference on Parallel And Real-Time Systems (PART '98)*, Adelaide, Australia, Sept. 1998.

[8] H. Härtig, L. Reuther, J. Wolter, M. Borriss, and T. Paul. Cooperating resource managers. In *5th Real-Time Technology and Applications Symposium (RTAS)*, Vancouver, Canada, June 1999. IEEE.

[9] J Consortium. *Real-Time Core Extensions (RTCE)*, Sept. 2000. Available at http://www.j-consortium.org/.

[10] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, Jan. 2000.

[11] National Institute of Standards and Technology. *Requirements for Real-time Extensions for the Java Platform*, Sept. 1999. Available at http://www.nist.gov/rt-java/.

[12] Object Management Group. *Real-Time CORBA Specification*, version 2.0 edition, Nov. 2003. formal/03-11-01, see http://www.omg.org/realtime/.

[13] M. Pohlack, R. Aigner, and H. Härtig. Connecting real-time and non-real-time components. Technical Report TUD-FI04-01, Technische Universität Dresden, Feb. 2004.

[14] The Real-Time for Java Expert Group. *The Real-Time Specification for Java*, v1.0 edition, 12 Nov. 2001. http://www.rtj.org/.

[15] R. Staehli and F. Eliassen. QuA: A QoS-aware component architecture. Technical Report Simula 2002-12, Simula Research Laboratory, 2002.

[16] N. Wang, C. D. Gill, D. C. Schmidt, A. Gokhale, B. Natarajan, C. Rodrigues, J. P. Loyall, and R. E. Schantz. Total quality of service provisioning in middleware and applications. *Microprocessors and Microsystems*, 27(2):45–54, Mar. 2003. Special Issue on Middleware Solutions for QoS-enabled Multimedia Provisioning over the Internet.