TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik

Technische Berichte

Technical Reports

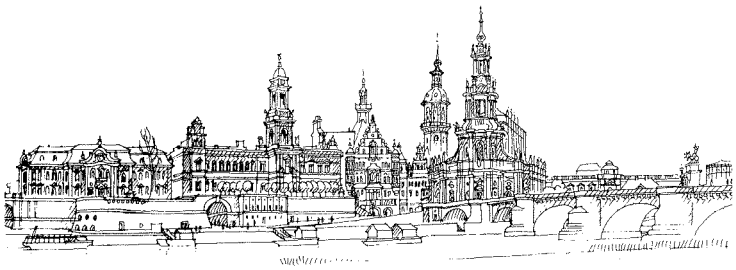
ISSN 1430-211X

TUD-FI04-04 - April 2004

Steffen Göbel, Christoph Pohl, Ronald Aigner,
Martin Pohlack, Simone Röttger, Steffen Zschaler

Institut für Systemarchitektur,
Institut für Softwaretechnik

The COMQUAD Component Container Architecture and Contract Negotiation



Technische Universität Dresden
Fakultät Informatik
D-01062 Dresden
Germany

URL: `http://www.inf.tu-dresden.de/`

The COMQUAD Component Container Architecture and Contract Negotiation*

Steffen Göbel Christoph Pohl Ronald Aigner Martin Pohlack
Simone Röttger Steffen Zschaler

Abstract

Component-based applications require runtime support to be able to guarantee non-functional properties. This report 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 three key issues: the conceptual architecture, an approach including implementation issues for splitting the runtime environment into a real-time-capable and a real-time-incapable part, and details of contract negotiation. The latter includes selecting component implementations for instantiantion based on their non-functional properties.

*A shorter version of this technical report appears in the proceedings of the WICSA-4 [10]

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 report we introduce the COMQUAD container architecture, which provides a runtime environment for QoS-capable, component-based software applications.

Component models like Sun's Enterprise JavaBeans (EJB) [4] or the OMG's CORBA Component Model (CCM) [27] 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—for instance, in [7]. In contrast, our notion of a *container* comprises all major parts of component management, as we will explain in Sect. 2.

One of the key benefits of component-oriented application servers is that new applications can be assembled from existing, commercial off-the-shelf (COTS) components in a plug-and-play fashion. The internal architecture of such component-based software applications is usually captured by descriptive means of *architecture description languages* (ADL, [25]). We use *assembly descriptors* as a part of our COMQUAD component model [11] for this purpose.

For brevity, we will use the terms real-time (RT) and non-real-time (NRT) in the context of this report. RT refers to QoS-capable parts that can guarantee certain non-functional properties of their service, whereas the latter term (NRT) shall be used for QoS-incapable parts providing best-effort services only.

We argue that the separation of mission-critical real-time code on one hand and lower priority non-real-time code on the other hand is natural for many applications where QoS is an issue. This realization was the driving force behind our decision for a split architecture that handles both aspects separately.

The remainder of this report is organized as follows: Section 2 provides a high-level view on our container architecture and the underlying component model [11]. The reasons for the real-time–non-real-time split are explained afterwards. Section 3 describes the implementation of this split architecture, the necessary communication primitives between both parts, and the details of contract negotiation necessary to guarantee a requested QoS for servicing client requests. Finally, we give an overview of related work and conclude with an outlook.

2 Approach

In this section we first give an overview of the conceptual elements of our architecture before explaining in more detail our approach of splitting the runtime environment into two parts: a real-time and a non-real-time part.

2.1 Conceptual Architecture

Our components are black-box elements, which implement business logic and cooperate with other components to solve an application requirement. The component concept is based on Szyperski's definition [38]. Components exist in a runtime environment—the container—which hides the system logic and details of the underlying platform. One of the main features of our component model is that—corresponding to ideas presented by Cheesman and Daniels [2]—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.

To support non-functional properties, it is essential to extend the container by a resource management providing services for admission and reservation of system resources, such as CPU time or memory. Figure 1 gives

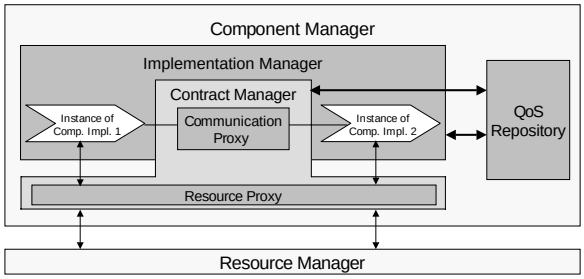


Figure 1: Conceptual COMQUAD container architecture [5]

an overview of our architecture consisting of two layers: component manager and resource manager. The component manager itself consists of three subsystems: implementation manager, QoS repository, and contract manager. The implementation manager supports the lifecycle of component implementations and the QoS repository stores the non-functional specification of each component implementation. We have developed CQML$^+$ [33] as our specification language for non-functional properties.

The key part of the container is the contract manager, which instantiates and connects the components required to service each client request. 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 [11]. The non-functional descriptors essentially contain CQML$^+$ specifications for each component implementation. The key information in the functional descriptors is contained in *component net templates*, which describe the component assemblies forming an application. The term *component assembly* refers to component instances whose used and provided ports have been connected to form a network of cooperating instances handling a request. Note that, although the component net templates are very similar in purpose to classic ADLs [25] there is one major difference: Classic ADLs describe connections between components at the level of instances of specific component implementations whereas our component net templates 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.

Once the contract manager has selected implementations and created component instances, it uses communication proxies to connect these instances according to the specification, and resource proxies to manage the component instances' usage of the reserved resources. It is important to realize that these proxies are primarily conceptual in nature. They serve to represent various mechanisms in the conceptual architecture, but are not necessarily present as actual elements of the running container. We usually implement communication proxies either implicitly by directly exchanging references between instances or explicitly by placing a chain of interceptors [35] between the instances. Resource proxies are very often implemented implicitly only, such that the resource management layer itself does all the work.

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 Video-on-Demand (VoD) service. We have enhanced the component model to allow specification of stream-based communication between components. At runtime, such communication is implemented by a container-provided buffer component, which uses normal request-response–based communication to exchange data packets with the participating components. More details can be found in [11].

2.2 Real-Time and Non-Real-Time Parts

In the previous section we described the conceptual architecture of our component run-

time environment. 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. Again, a VoD application is a typical example. Only the actual delivery of the 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 distinction can be applied to the architecture of a component runtime environment. Figure 2 shows an extension of the conceptual architecture with the parts that need to give guarantees (real-time) and the parts that do not (non-real-time) clearly separated. Most of the work is done in the non-real-time 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 real-time part is essentially restricted to actually instantiating components, reserving resources, and executing the instantiated components in response to user requests.

This concept is closer to the reality of applications with non-functional properties than the approach of monolithic real-time applications. Additionally, it also allows us to make use of advanced component technology (namely JBoss [7]) on the non-real-time side while still being able to make full use of the real-time and resource-reservation features of the underlying platform on the real-time side. Thus, we can leverage the best of both worlds and move towards a running prototype very efficiently.

The most noteworthy consequence of this split in the architecture can be seen in the communication proxies. Because now there can occur

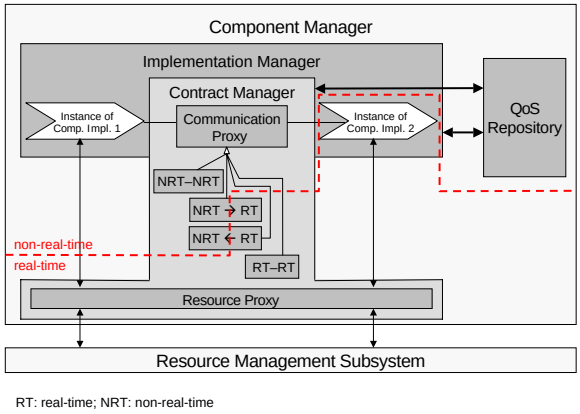


Figure 2: Detailed COMQUAD container architecture

four different types of communication, either in one of the container parts or between them, we need four different specializations of communication proxies. It is important to understand the differences between these communication proxies. Namely, the proxy for communicating from the real-time part to the non-real-time part needs to provide asynchronous communication whereas all other proxies only have to provide synchronous communication.

3 Implementation

3.1 Split Architecture

Based on our experience with the development of DROPS (Dresden Real-time OPerating System) [16], we conclude that real-time capabilities are often not necessary for large applications, but just for small parts of them. In DROPS we have small real-time capable server processes for dedicated tasks—for example, a window manager [6], a SCSI disk driver [32], and network drivers [1, 3, 24]. These servers run directly on our real-time capable microkernel (Fiasco, [19]). Complex legacy software without real-time requirements runs concurrently on a large off-the-shelf Linux server (L^{4}Linux, [18]), which gives us a fairly large code base.

Consequently, we designed our component

container as a split architecture where we have a large non-real-time container, which is based on JBoss [7], and a small real-time container, which runs directly on DROPS . Thus, we could both simplify the development by reusing existing software and minimize the amount of real-time–capable program code.

The split architecture also motivates the introduction and support of both QoS-capable and QoS-incapable components. This allows developers to divide applications into a real-time and a non-real-time part as well. Currently, NRT components must be developed with Java and RT components with C/C++, respectively. In the future we plan to loosen this restriction.

For the NRT container we use the infrastructure of a stripped down JBoss container and add support for the new COMQUAD component model [11] 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 (cf. Sect. 3.4) 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. 3.2), resource managers, and a small framework for components, consisting of interfaces and base classes for component instances, and helper functions.

When communicating with real-time–aware cients, the RT container acts as a proxy for the NRT container with respect to its management functionality for connecting component instances, intercepting communication, and starting and stopping components. The real-time component instances have the RT container reserve their required resources with the respective resource managers. DROPS resource managers as described in [17] are organized in a resource reservation systems, which is governed by a *QoS manager*. The QoS manager accepts the reservation requests of the container and tries to place the reservation with the respective resource managers. If all requested resources can be reserved, it replies to the container with a message containing handles to the reservations. These handles are later used to access the reserved resources. 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 reservation. 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 of all components using the considered resource.

In real-time environments we have to differentiate between active and passive components. A passive component is only executed when a user request arrives or another component calls its methods. In contrast, an active component is processing data in its own thread of control, that is, without an explicit user request. An example is a video decoder that decodes a frame every 40 ms without another component calling the decode method. To be able to use such an active component, the container must be able to reserve periodic execution time for this component. The component has to interact with the execution infrastructure to be invoked whenever its execution time starts. This could, for instance, be done by explicitly waiting for the beginning of the next period.

3.2 Communication between RT and NRT Container Parts

To maximize software reuse and to create a really small and fast RT container, we tried to find a minimal set of functionality the RT container has to support in order to be manageable by the NRT container. Most of these

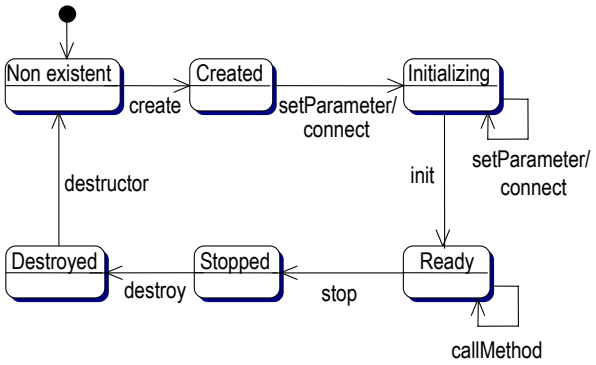


Figure 3: Life cycle of a COMQUAD component

functions also correspond to state transitions in the life cycle of a COMQUAD component as depicted in Fig. 3.

create/destroy. The RT container must be able to create new and remove existing component instances.

init. Components must be initialized before they can be started.

stop. This functions stops the request delivery to a component. Buffered requests will still be processed, but no new requests will be accepted.

connect. The container must be capable of connecting component interfaces with other instances.

setParameter. The container must be able to set components' properties. This method is meant to be used for modifying configuration parameters upon initialization.

reserveResources. The container must be able to make resource reservations on behalf of components.

callMethod. The container must be able to forward calls to arbitrary component methods. We discuss the communication details of this function in Sect. 3.3.

install/uninstall Specification/Implementation. The container must be able to accept and remove component specifications and implementations.

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. Other services, such as instance creation, resource reservation, and connecting instances do not have RT requirements.

3.3 Communication between RT and NRT Components

The architecture split described in Sect. 3.1 implies two types of components: real-time and non-real-time. As a consequence, four constellations of communication can occur:

Non-real-time to non-real-time. In this constellation a NRT component in the NRT component container invokes another NRT component. This communication uses the infrastructure provided by the JBoss-based container only.

Non-real-time to real-time. The details for this communication are depicted in Fig. 4. A NRT component invokes a RT component in the RT container. 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 through a generated bridge, which marshals the message into an octet stream and transfers it to the RT container using native microkernel IPC mechanisms. In the RT container a generated demultiplexing container skeleton delivers the message to the addressed interface skeleton, which itself unmarshals the message from the octet stream. On the RT side we do not use a Dynamic Invocation Interface but generated code instead, which is larger but faster [12, 14, 15].

Communication with RT components requires a reservation and the invoking component's communication pattern has

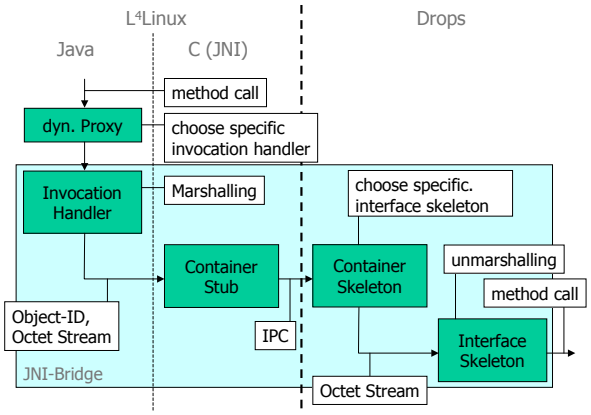


Figure 4: NRT to RT component communication

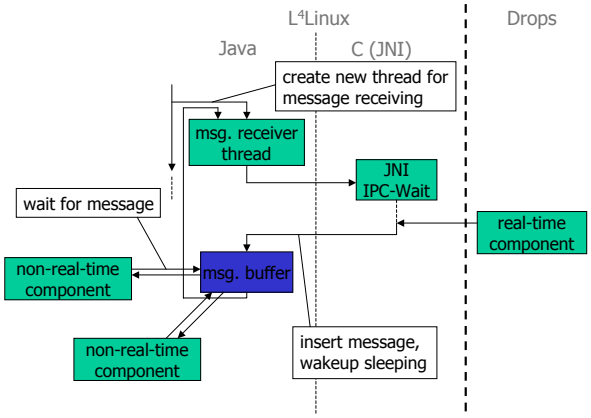


Figure 5: RT to NRT component communication

to conform with the reservation. Non-conforming communication can be delayed or dropped by the RT container.

Real-time to real-time. Communication between RT components does not cross container boundaries and can be scheduled entirely by the RT container, because their specification contains information about the load generated.

Real-time to non-real-time. 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 will be discussed in detail below.

Asynchronous communication requires message buffers. A very simple solution for this approach is depicted in Fig. 5, with the buffer located in the NRT container. 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 [30], where we propose a generic buffer component, which is responsible for managing requests sent from RT components to NRT components. This component handles buffer replacement strategies and request delivery on behalf of the sender. Using this generic buffer component it is possible to transparently connect RT and NRT components.

3.4 Component Contract Negotiation

The negotiation of component–component as well as component–platform contracts [41] in the contract manager is crucial for supporting non-functional properties in our architecture. Negotiation is responsible to find valid component implementations for component nets fulfilling the required properties of a particular client.

We use the terminology of CQML$^+$ [33] for contract negotiation, meaning that a contract as well as QoS requirements and offers consist of a set of *quality statements*. Each quality statement formulates a Boolean expression using *quality characteristics*. Finally, quality characteristics represent the current value of some system state—for example, response time.

According to Cheesman and Daniels [2], a (functional) component specification supports a non-empty set of interfaces and it can be realized by a number of implementations. In turn, we require each QoS-aware component implementation to provide at least one *QoS profile*. It represents a particular operating range of the implementation and consists of offered and required quality statements as well as the resource demand. All this information spawns a search space from which a component implementation and profile must be selected by

the contract negotiation algorithm.

Contract negotiation is initiated when a client wants to create a specified component instance via the home interface of this component. The client transmits QoS requirements together with the `create` request to the container. The actual contract negotiation is then performed in three steps within the NRT part of the container:

1. *Component net computation:* A request is typically not handled by a single component, but by a network of cooperating components. In this step, the contract manager recursively derives a representation of this component net from the application assembly. The component net representation exists entirely at the level of (functional) component specifications, no implementations have been selected so far. Because it uses functional information only, the result can already be determined at component deployment time.

2. *Component implementation and profile selection:* Based on the initial QoS requirement of the client, only component implementations and profiles are selected that mutually fulfill the required and offered quality statements for each connection in the component net.

3. *Resource reservation:* The contract manager transmits the resource demand of the concrete component net found in the previous step to the resource manager. The resource manager can either successfully reserve the required resources or report an error in case of insufficient system resources. In the latter case the contract manager must return to the previous step and try to find another solution with a smaller footprint. If the resource reservation has been completed successfully, the contract manager returns a reference to the requested component instance to the client.

The component implementation and profile selection step is the most important part of the contract negotiation algorithm. We first present a naïve approach together with a discussion of its problems. Later we investigate optimization strategies to overcome the identified challenges.

First, a list of available implementations is created together with their QoS profiles for each component specification of the component net. Next, a particular implementation and profile are chosen for each component of the net from the previously created list. We call such a selection a *component net configuration*. Then, the required and provided quality statements for each connection in the component net are compared in order to find valid contracts. Iff (if and only if) a valid contract can be obtained for all connections in the component net configuration, the current component net configuration represents a possible solution for the client request. In this case the resource reservation step can be executed. If reservation fails, another iteration is started and the next possible configuration is chosen.

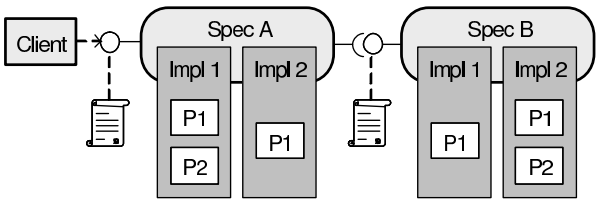


Figure 6: Contract negotiation in component nets

Figure 6 depicts a simple example scenario where a client creates a component net consisting of two components. Each of the two connected component specifications A and B has two available implementations. Depending on the client's QoS requirements, the contract manager has to select one out of nine possible configurations (see Tab. 1). The example is deliberately kept simple to allow a concise illustration of our different approaches.

Table 1 shows all possible configurations together with a short explanation whether they are valid. The first configuration selects implementation 1 with profile 1 for $Spec\ A$ and implementation 1 with profile 1 for $Spec\ B$ (abbr. $A.1.1$ resp. $B.1.1$). The configurations 2–9 are

Table 1: Possible component net configurations for the example in Fig. 6

No.	Configuration	Contract negotiation result
1	A.1.1 – B.1.1	Provided QoS of A.1.1 < required QoS of client
2	A.1.1 – B.2.1	Provided QoS of A.1.1 < required QoS of client provided QoS of impl B.2 is not compatible with impl. A.1
3	A.1.1 – B.2.2	Provided QoS of A.1.1 < required QoS of client provided QoS of impl B.2 is not compatible with impl. A.1
4	A.1.2 – B.1.1	Provided QoS of A.1.2 < required QoS of client
5	A.1.2 – B.2.1	Provided QoS of A.1.2 < required QoS of client provided QoS of impl B.2 is not compatible with impl. A.1
6	A.1.2 – B.2.2	Provided QoS of A.1.2 < required QoS of client provided QoS of impl B.2 is not compatible with impl. A.1
7	A.2.1 – B.1.1	Valid configuration
8	A.2.1 – B.2.1	Negotiation ok, but insufficient resources
9	A.2.1 – B.2.2	Valid configuration

created accordingly.

The naïve contract negotiation approach applied to the example successively verifies the component net configurations—for example, in the order depicted in Tab. 1. The two component connections ($Client$ to $Spec\ A$ and $Spec\ A$ to $Spec\ B$) are tested for each configuration to find valid contracts but the order of these connection comparisons is chosen randomly. An invalid contract for one connection stops the test and the algorithm continues with the next configuration. Finally, configuration 7 is found to be valid and the resources are reserved successfully. This is a stop criteria and the remaining configurations 8 and 9 are not tested anymore.

Unfortunately, this algorithm has several drawbacks:

Complexity. To illustrate the complexity consider the following example: A component net with only four components and two available implementations for each component with two profiles each already results in $4^4 = 256$ different configurations that need to be searched. Obviously, we need to limit the search space to avoid this combinatorial explosion.

Negotiation and Reservation are unrelated. The negotiation of contracts between components in a component net is performed independently from the ensuing resource reservation. This is only a problem if resource reservation fails. In this case, the next iteration may find *any* other component net configuration even if that configuration required more resources. In fact, resource demand is completely ignored during the contract negotiation phase.

Finds any solution, but not the best. Although the algorithm is guaranteed to find a possible solution if any exists, it does not necessarily find the *best* solution. This means that it may not provide the best QoS possible for the currently available system resources. Of course, it would be possible to always look for all solutions and then pick the best one out of them. But this would even increase complexity, rendering it impracticable for large-scale applications.

To reduce complexity, we need to find stop criteria for the implementation and profile search. We also need to ensure that the best possible solution is selected instead of an arbitrary one. Currently, we do not have a comprehensive strategy to solve these challenges, but we have a couple of independent approaches that, combined, lead to a good heuristic.

As a first approach, we try to minimize the search space by selectively removing QoS profiles or even component implementations from the list of available profiles for a particular component specification. This can be done if a profile does not match any profile of the partner component—for example, if it uses quality characteristics that no other profile provides. The complexity of this step depends linearly on the number of profiles.

Applied to our example, we can take out configurations 2, 3, 5, and 6 because the profiles of implementation $B.2$ are not compatible with the implementation $A.1$.

The second approach optimizes the negotiation between two components as well as between components and clients. All available profiles for a particular component specification are ordered by descending quality. Given a quality requirement, profiles' offers are now checked in this order. The comparison can be stopped as soon as the current profile's of-

fer does not fulfill the requirement. Because the profiles have been ordered by their quality, none of the remaining profiles can fulfill the requirements. A lot of unneeded comparison steps are avoided this way. To order the profiles, we combine two different orderings: *(i)* The order of the profiles for one implementation is given in this implementation's non-functional specification (using CQML$^+$'s `precedence`-clause). *(ii)* Profiles of different implementations are ordered using the quality semantics from the definition of the constrained characteristics, extended by a client-defined precedence rule for resolving conflicts. We are currently investigating how conflicts between these two order relationships can be resolved best when merging them.

In our example, profile $A.1.1$ does not fulfill the client's QoS requirements. Thus, if the profiles for one particular implementation are ordered by descending quality, profile $A.1.2$ cannot fulfill the requirements either, because it offers even less QoS than profile $A.1.1$. Consequently, configurations 1–6 can be removed from the search space.

To extend this approach to complete component nets, we need to define an order in which the contract manager negotiates bilateral component–component contracts. Based on their connections, we can define a partial order on component specifications in component nets as follows: A component C_1 is "smaller than" a component C_2 iff C_1 directly or indirectly uses C_2. Two components are incomparable iff they have no direct or indirect usage relationship. By this order relation the client (or the component directly used by the client, respectively) is always the minimal component. We can now use standard algorithms for sequencing partially ordered sets to determine a total order in which contracts are negotiated. If the bilateral contract negotiations are performed in this order, potential conflicts can be detected early and many—then unnecessary—comparisons can be skipped.

The two component connections in our example (cf. Fig. 6) define two order relations ($Client < Spec\ A$ and $Spec\ A < Spec\ B$) leading to the total order ($Client < Spec\ A < Spec\ B$). Bilateral contracts are negotiated in this order—first, the connection between $Client$ and $Spec\ A$ and afterwards between $Spec\ A$ and $Spec\ B$. The negotiation of contracts between $Client$ and $Spec\ A$ fails for the configurations 1–6 because $Spec\ A$ cannot fulfill the required QoS of the client. Since this connection is negotiated first according to the determined order relation, the second component connection between $Spec\ A$ and $Spec\ B$ does not need to be compared. Several unnecessary negotiations are avoided this way.

The third approach deals with the problem that contract negotiation and resource reservation are unrelated. This issue is more difficult to tackle. Currently, the resource reservation returns only a simple Boolean response. We believe a more detailed response would be more appropriate in case of reservation failure. This information could then be used to find another configuration for the component net, requiring less resources.

We are currently working on these issues, investigating the various options for optimization of the contract negotiation algorithm in more detail.

4 Related Work

The OMG's CORBA Component Model (CCM) [27] 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 [4], 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 [7]. It features a slightly different variant of Interceptors as described in Schmidt's Pattern-

oriented Software Architecture [35]. This variant is the foundation for the resource and communication proxies in our component platform architecture. We have already shown in previous publications how this concept can be exploited to support arbitrary middleware functionality in such component platforms [29, 9].

The project QuA [37, 36] aims at precisely defining an abstract component architecture, including the semantics for general QoS specifications. The proof of concept is provided by implementing an open framework for platform-managed QoS. 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 [16]. This allows us to fully leverage the virtues of this platform—for instance, its clean microkernel architecture.

CIAO [40, 13], another related project, builds a QoS-enabled CCM implementation on top of TAO [34]. The project's philosophy is a strong adherence to existing OMG specifications such as RT/CORBA [28] 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 considerable overhead of implementing or extending a fully compliant CCM infrastructure would have been counterproductive to a prompt realization of our main targets.

The Real-Time Specification for Java (RTSJ) [31] 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. The reference implementation of Real-Time Java is based on TimeSys Linux [39], which ensures dependability of real-time applications by running critical sections in kernel mode. In contrast, our platform DROPS [16] follows the philosophy of running as much code as possible in user mode, thus increasing system stability and safety.

Requirements for real-time extensions for Java were defined in the NIST report [26]. 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) [21], 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.

The 2K Operating System [22] implements a resource management system targeted at distributed applications. It focuses on load balancing through global knowledge about resource utilization on local nodes [23]. All local resources are monitored by a local resource manager (LRM), which is also responsible for admission, resource negotiation, reservation, and scheduling of jobs. Resources are described using a name-value pair: The name identifies the resource, the value contains a description of the resource properties. One reservation can contain several resource descriptions for different resources. In contrast, COMQUAD uses heterogenous resource managers and a generic description for resources and reservation interfaces.

5 Conclusions and Outlook

Our report 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 [11]. 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.

The container uses the information from assembly, specification, implementation, and quality (CQML$^+$) descriptors (cf. [11]) to instantiate and connect component implementations in such a way that non-functional properties required by clients of the system can be guaranteed. We refer to this as *Container-Managed QoS*. In the most simple, currently implemented case of contract negotiation—which we described in Sect. 3.4—the container compares client requests with all possible component net configuration offers and if they match (i.e., the component net offers more than, or at least the same amount as, the client requests), the container instantiates the component net configuration. Various suggestions for improvement have already been made in Sect. 3.4—for example, to selectively reduce the search space of potential implementations, to apply standard graph serialization algorithms after defining a partial order on required and provided quality statements, or to augment the return type of resource reservation. These enhancements are currently being implemented and evaluated. Related activities also include the unified treatment of security and other non-functional properties [8], where user-defined weighing of different non-functional measures (e.g., confidentiality vs. throughput) becomes an important issue. This aspect forms an integral part of our contract negotiation scheme.

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] Martin Borriss and Hermann Härtig. Design and implementation of a real-time ATM-based protocol server. In *19th Real-Time Systems Symposium (RTSS)*, Madrid, Spain, December 1998. IEEE.

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

[3] U. Dannowski and H. Härtig. Policing offloaded. In *6th Real-Time Technology and Application Symposium*, Washington D.C., USA, May 2000. IEEE.

[4] Linda G. DeMichiel. *Enterprise JavaBeans Specification, Version 2.1*. Sun Microsystems, final release edition, 12 November 2003.

[5] Brit Engel. Entwicklung einer Laufzeitumgebung für Komponenten mit Ressourcenanforderungen. Diplomarbeit, Technische Universität Dresden, 30 April 2003. In German. English title: Development of a runtime environment for components with requirements for resources.

[6] Norman Feske and Hermann Härtig. Demonstration of DOpE — a window

server for real-time and embedded systems. In *24th Real-Time Systems Symposium (RTSS)* [20], pages 74–77.

[7] Marc Fleury and Francisco Reverbel. The JBoss extensible server. In Markus Endler and Douglas 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.

[8] Elke Franz and Christoph Pohl. Towards unified treatment of security and other non-functional properties. In *Workshop on AOSD Technology for Application-Level Security (AOSDSEC'04)*, Lancaster, UK, 23 March 2004.

[9] Steffen Göbel and Michael Nestler. Composite components support for EJB. In *Winter International Symposium on Information and Communication Technologies (WISICT'04)*, Cancun, Mexico, 5–8 January 2004. ACM.

[10] Steffen Göbel, Christoph Pohl, Ronald Aigner, Martin Pohlack, Simone Röttger, and Steffen Zschaler. The COMQUAD component container architecture. In Clemens Szyperski, editor, *4th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, Oslo, Norway, 12–15 June 2004. IEEE.

[11] Steffen Göbel, Christoph Pohl, Simone Röttger, and Steffen Zschaler. The COMQUAD component model – enabling dynamic selection of implementations by weaving non-functional aspects. In Karl Lieberherr, editor, *3rd International Conference on Aspect-Oriented Software Development (AOSD'04)*, Lancaster, UK, 22–26 March 2004. ACM Press.

[12] 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, November 1996. IEEE.

[13] Aniruddha Gokhale, Douglas C. Schmidt, Balachandran Natarajan, and Nanbor Wang. Applying model-integrated computing to component middleware and enterprise applications. *Communications of the ACM*, 45, October 2002. Special Issue on Enterprise Components, Service and Business Rules.

[14] A. Haeberlen, J. Liedtke, Y. Park, L. Reuther, and V. Uhlig. Stub-code performance is becoming important. In *1st Workshop on Industrial Experiences with Systems Software (WIESS)*, pages 357–363, San Diego, CA, USA, 22 October 2000. USENIX.

[15] Timothy H. Harrison, David L. Levine, and Douglas C. Schmidt. The design and performance of a real-time CORBA event service. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '97)*, pages 184–200, Atlanta, GA, USA, 5–9 October 1997. ACM SIGPLAN.

[16] H. Härtig, R. Baumgartl, M. Borriss, Cl.-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, September 1998. ACM SIGOPS.

[17] H. Härtig, L. Reuther, J. Wolter, M. Borriss, and T. Paul. Cooperating resource managers.

[18] Hermann Härtig, Michael Hohmuth, and Jean Wolter. Taming Linux. In *5th Annual Australasian Conference on Parallel And Real-Time Systems (PART '98)*, Adelaide, Australia, September 1998.

[19] Michael Hohmuth. *Pragmatic nonblocking synchronization for real-time systems*. PhD thesis, Technische Universität Dresden, Fakultät Informatik, September 2002.

[20] IEEE. *24th Real-Time Systems Symposium (RTSS)*, Cancun, Mexico.

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

[22] F. Kon, R. H. Campbell, F. J. Ballesteros, M. D. Mickunas, and K. Nahrstedt. 2K: A distributed operating system for dynamic heterogeneous environments. In *9th Intl. Symposium on High Performance Distributed Computing*, Pittsburgh, USA, August 2000. IEEE.

[23] F. Kon, T. Yamane, C. K. Hess, R. H. Campbell, and M. D. Mickunas. Dynamic resource management and automatic configuration of distributed component systems. In *6th Conference on Object-Oriented Technologies and Systems (COOTS '01)*, San Antonio, USA, January 2001. USENIX.

[24] Jork Löser and Hermann Härtig. Real time on ethernet using off-the-shelf hardware. In *1st Intl. Workshop on Real-Time LANs in the Internet Age*, Vienna, Austria, June 2002.

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

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

[27] Object Management Group. *CORBA Components*, 2001. ptc/01-11-03.

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

[29] Christoph Pohl and Steffen Göbel. Integrating orthogonal middleware functionality in components using interceptors. In *Kommunikation in Verteilten Systemen (KiVS 2003)*, Informatik Aktuell, Leipzig, Germany, February 2003. VDE/ITG & GI, Springer.

[30] Martin Pohlack, Ronald Aigner, and Hermann Härtig. Connecting real-time and non-real-time components. Technical Report TUD-FI04-01, Technische Universität Dresden, February 2004.

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

[32] Lars Reuther and Martin Pohlack. Rotational-position-aware real-time disk scheduling using a dynamic active subset (DAS). In *24th Real-Time Systems Symposium (RTSS)* [20], pages 374–385.

[33] Simone Röttger and Steffen Zschaler. CQML$^+$: Enhancements to CQML. In Jean-Michel Bruel, editor, *1st Intl. Workshop on Quality of Service in Component-Based Software Engineering*, pages 43–56, Toulouse, France, June 2003. Cépaduès-Éditions.

[34] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee. The design of the TAO real-time object request broker. *Computer Communications*, 21(4), 1998.

[35] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, volume 2 of *Software Design Patterns*. John Wiley and Sons, 2000.

[36] Richard Staehli and Frank Eliassen. QuA: A QoS-aware component architecture. Technical Report Simula 2002-12, Simula Research Laboratory, 2002.

[37] Richard Staehli, Frank Eliassen, Gordon Blair, and Jan Øyvind Aagedal. QuA: A QoS-aware component architecture. In *Middleware2003 Companion*, page 330, Rio de Janeiro, Brazil, June 2003. PUC–Rio.

[38] Clemens Szyperski. *Component Software: Beyond Object-Oriented Program-*

ming. Component Software Series. Addison-Wesley, 2nd edition, 2002.

[39] TimeSys Corp. *TimeSys Linux*. See `http://www.timesys.com/`.

[40] Nanbor Wang, Christopher D. Gill, Douglas C. Schmidt, Aniruddha Gokhale, Balachandran Natarajan, Craig Rodrigues, Joseph P. Loyall, and Richard E. Schantz. Total quality of service provisioning in middleware and applications. *Microprocessors and Microsystems*, 27(2):45–54, March 2003. Special Issue on Middleware Solutions for QoS-enabled Multimedia Provisioning over the Internet.

[41] Steffen Zschaler and Simone Röttger. Types of quality of service contracts for component-based systems. In *Intl. Conference on Software Engineering (IASTED SE 2004)*, Innsbruck, Austria, 17–19 February 2004. IASTED, ACTA Press.