Types of Quality of Service Contracts for Component-Based Systems

Steffen Zschaler and Simone Röttger
Institute for Software and Multimedia Technology
Department of Computer Science
Dresden University of Technology
Dresden, Germany
email{Steffen.Zschaler, Simone.Roettger}@inf.tu-dresden.de

ABSTRACT
In this paper, we identify the different roles and contract types which are important in providing Quality of Service (QoS) properties of component-based systems. A surprising result of our work is that direct contracts between components are not necessary and even insufficient to handle non-functional properties of component-based systems.

KEY WORDS
Software Engineering, Quality of Service, Component-Based Software, Design by Contract

1 Introduction

In [1] four different levels of contracts are defined for component-based software: the syntactic, behavioural, synchronization and quality of service (QoS) level. In this paper we take a closer look at the QoS contract level, examining the various roles that are involved and the different types of contracts which are needed to provide QoS properties for component-based software. We also sketch out how these contracts can be used to provide QoS-properties for component-based software.

The work presented in this paper is restricted to QoS in the narrower sense. Properties like security, transactionality, or maintainability are left out of consideration. Also, we restrict ourselves to components as per the definition in [2]. This research is part of the COMQUAD[1] project.

2 Overview of Contracts and Roles

Figure 1 shows a component-based system and the roles and contract types which we have identified. Components are executed by a runtime environment – the component container – which provides components with various services, most notably:

- *Platform Abstraction*: The components only interoperate with the container, they do not need to know about the underlying operating system or hardware – the platform.
- *Component Communication*: The container ensures that component instances can communicate with other component instances. To this end, the container creates and maintains connector instances between the component instances.

The container itself runs on a platform, consisting of an operating system which provides resources like cpu-usage, memory, persistent memory (hard disk space) etc. Together, components, container, and platform form the system.

Our notion of a component is similar to EJB [3] session beans, but with the possibility to specify used and provided interfaces. Also, in addition to the normal operational and message based interfaces, components can define streaming interfaces which are used for continuous data delivery, such as video or audio streams. For each component, a QoS-descriptor [4] specifies the provided QoS under the condition, that the container provides certain resources and that other components provide their services with a certain QoS.

There are two roles that actively interoperate with the system. In the development process there are more roles which are of interest, but we do not consider these in this paper. Once the system has been built and is running, there are mainly two kinds of stakeholders with an interest in QoS:

1. *Operators* or *Service Providers* who use the system to provide certain services to their clients, and
2. *Clients* who are interested in the services offered by the system.

For example, an organization that provides video-on-demand services via the internet would be an operator or service provider in our model. The individuals who access the service in order to watch specific videos at a specific time are the clients of the video-on-demand service.

While there can be many individuals who are clients of a system, there usually is only one entity who is the operator of the system. This entity can be an organization and need not be an individual.

Between these roles and the system, as well as between different parts of the system, there exist QoS contracts, which constitute the QoS specification of the sys-

[1]COMponents with QUantitative properties and ADaptivity at Technische Universität Dresden and Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany; supported by German Research Council; see also www.comquad.org

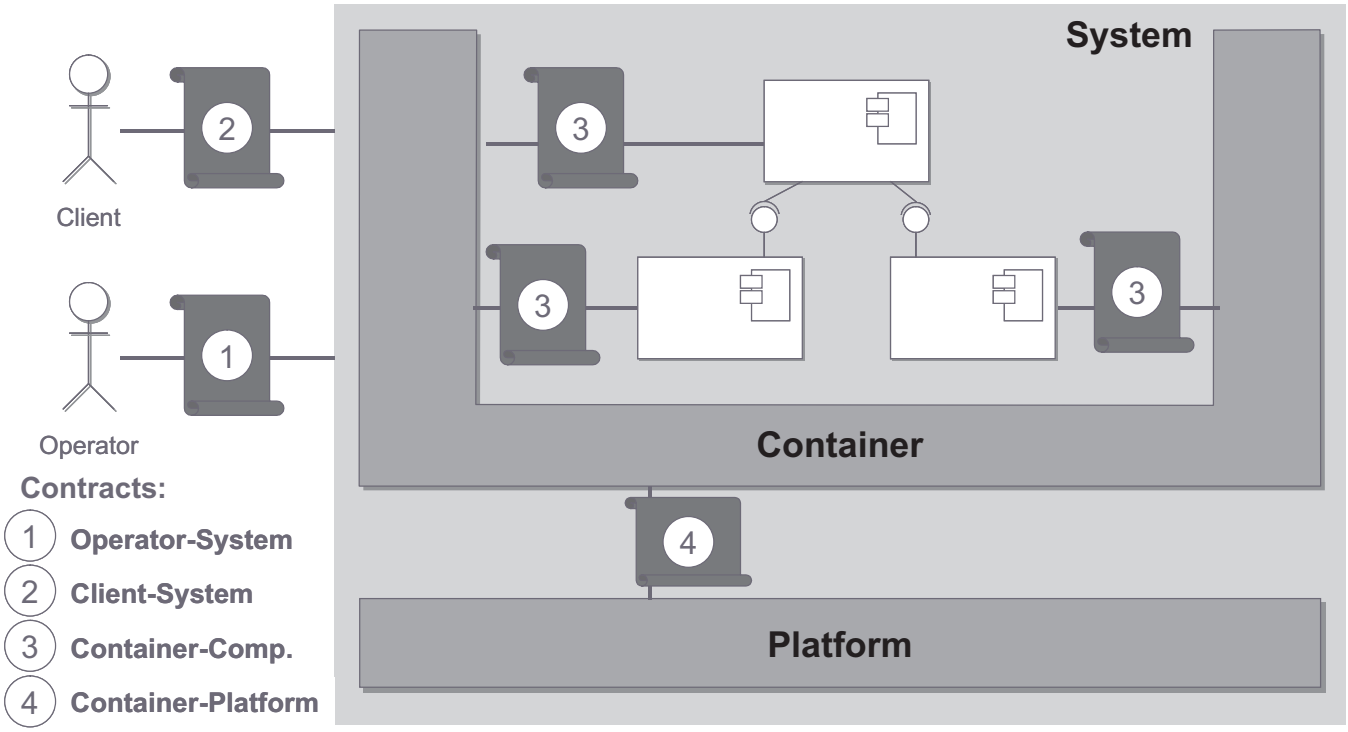

Figure 1. Roles and contract types in our model.

tem. Some of these contracts are *static*, in that they can be negotiated off-line, before the system is running, and some are *dynamic* contracts which can only be negotiated in a running system. Beugnard et al state in [1]: "A contract carries mutual obligations and benefits…", i.e., there are always two parties in a contract, both of which assert something about their own behaviour, or behaviour that is under their control. If one of the parties does not fulfill its part of the contract, the other party also is no longer bound by the contract. In order to support its assertions in one contract, a party may need to rely on other contracts with other parties. If such a contract is broken, the contract it supports may also be broken. For example, the container needs to conclude with the platform a contract about required resources before it can conclude a contract with a component. If the platform breaks the container-platform contract, the container may not be able to fulfill the container-component contract.

Consequently, there are four types of contracts in our model: operator-system, client-system, container-component, and container-platform contracts. Some of these contracts exist in support of other contracts. All contracts will be described in more detail – including a classification as static or dynamic – in the following sections.

3 The Operator-System Contract

The contract between operator and system is a contract which can be completely negotiated off-line. The operator asserts to the system a certain load – e.g., a maximum number of client requests per second or a specific distribution of the interarrival times of client requests – and in exchange, the system guarantees to provide a particular QoS – e.g., a maximum response time or a minimum frame rate – for at least a certain percentage of the client requests. In the latter case, the operator-system contract is a probabilistic description of the system's behaviour.

For example, the operator of our video-on-demand example may assert that there will be no more than 15 client requests per second, and that nine out of ten clients will watch "The Matrix" and one out of ten clients will watch "Much ado about nothing". In exchange, the system asserts that it will be able to deliver videos with at least 15 frames per second at 400×300 pixels for 80% of all requests.

Under which circumstances can the two parties confidently "sign" such a contract? The operator needs to have statistic data from previous runs of a similar system. If such data is not available, she could estimate these values – e.g., based on relevant market research – and correct the contracts after some experience with the running system. Notice that the system may fail to provide the QoS asserted if client requests come in at a higher rate than estimated. The container can only conclude an operator-system con-

tract if it can create a plan for component usage that fulfills the following conditions:

- It allows the container to provide the QoS properties required by the operator and to serve the load specified, and
- The container can conclude supporting contracts with the components and with the platform.

The container uses Container-Based Scheduling [5] to determine the number of component instances to create and the size of request buffers to allocate. Based on this plan it attempts to conclude container-platform contracts to reserve the required resources and container-component platforms to ensure the required QoS will be provided by the component instances.

4 The Client-System Contract

Each individual client who accesses the system negotiates an individual contract for his usage of the system. Therefore, client-system contracts can only be negotiated online.

There must be an operator-system contract for each client-system interaction. A client-system contract is said to be *in accordance to* an operator-system contract, iff the system concludes it to fulfill the operator-system contract. Cases in which a client-system contract may be construed not to be in accordance to an operator-system contract include:

- If the number of clients wishing to be served exceeds the maximum number of concurrent requests asserted in the operator-system contract.
- If declining the contract would still allow the system to provide the QoS asserted in the operator-system contract at the percentage asserted.

The client asserts the time when she wants to access the system – e.g., "now" or "in five minutes" – and other parameters the system needs to know to schedule her request. For example, if the application uses active components to stream a video to the client, the system may need to know whether or not the client plans to make pauses while watching the video.

For instance, client Caroline and the system may negotiate a contract that Caroline can watch "The Matrix" with at least 15 frames per second at 400×300 pixels starting "now" – i.e., at the moment the contract is concluded.

5 The Container-Component Contract

The container uses contracts with the components in an application to support its assertions in operator-system and client-system contracts. Depending on which type of contract it supports, a container-component contract can either be static or dynamic. The container negotiates and concludes such a contract with each component. The component asserts to provide a specific QoS. In exchange, the container guarantees to provide the resources the component needs as well as the QoS needed from other components. The container finds information on QoS requirements and resource demand of a component in that component's QoS-descriptor.

A contract between the container and one component may depend on contracts between the container and other components in the application, if the former component needs services (with a certain QoS) from the latter components. In such cases, the container can only conclude the contract if it has previously concluded contracts with all of the required components.

6 The Container-Platform Contract

Resources are not managed by the container directly, but by the platform. If the container wants to guarantee required resources to components, it needs a guarantee about these resources from the platform. This is the contents of a container-platform contract. Container-platform contracts can be both static or dynamic.

In such a contract, the platform asserts that the container may use a certain amount of the resources managed by the platform. It guarantees that these resources will be available when needed. On the other hand, the container asserts that it will not use more than the contracted amount of resources. Container-platform contracts need not be restricted to one-time usage of resources, but may also cover periodic usage of resources, or even probabilistic descriptions of resource usage. The platform uses resource-specific admission algorithms to determine whether or not to "sign" a contract. The container uses container-based scheduling [5] and the container-component contracts to determine the amount of resources it needs to support an application.

7 Contracts between Components?

When we first started to think about quality of service contracts for components, the main contract type we were aware of was contracts between components. We were surprised to find that contracts of this type not only can, but *must* be, replaced by contracts between the container and the components[2]. The reason is, that to provide the service of component communication (as described in Section 2), the container potentially needs to interpose management code between communicating components. This implies, that component-component contracts cannot be negotiated without knowledge of the container. We have solved this problem by making the container the central point of negotiation: Now, all contracts are mediated by the container,

[2]Notice that for *functional* contracts the possibility exists too, but not the *necessity*.

which can thus use its knowledge about effects caused by management code when negotiating contracts.

Having said this, an application developer may still find it helpful to consider component-component contracts at an early stage of development. He will then need to refine his model to use container-component contracts, before he can implement and deploy the system.

8 Conclusions

In this paper, we have identified two roles with an interest in the QoS properties of a system: the operator and the client. We have also described the various types of contracts that are necessary to provide QoS properties of a component-based system, and how they fit together. Surprisingly, component-component contracts are not only not necessary, but even incomplete due to QoS effects caused by the container. They may, hoewever, be helpful in the early stages of high-level design.

This paper presents work in progess. We plan to implement a component container with QoS support, which will make use of the contract types outlined. The next steps in our research will be to formalize the precise contents of contracts as well as contract negotiation rules. Also, we need to better understand what happens when contracts are violated; in particular when a contract is violated because a supporting contract was violated before. Another point for future research is to analyse whether we need to consider other stakeholders (e.g., EJB roles [3]) in our approach.

References

[1] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. *IEEE Computer*, 32(7):38–45, July 1999.

[2] Clemens Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley Publishing Company, November 1997.

[3] Sun Microsystems. Enterprise JavaBeans Specification, version 2.0. Final Release, August 2001.

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

[5] Ronald Aigner, Martin Pohlack, Simone Röttger, and Steffen Zschaler. Towards pervasive treatment of non-functional properties at design and run-time. In *Proc. 16th International Conference on Software and Systems Engineering and their Applications (ICSSEA)*, Paris, December 2003.