Enhancing QoS Support for CCM Based Middleware

COSMINA IVAN
Department of Computer Science
Technical University of Cluj-Napoca,
26, Baritiu Str.
ROMANIA
cosmina.ivan@cs.utcluj.ro
www.cs.utcluj.ro/~civan

VASILE DADARLAT
Department of Computer Science
Technical University of Cluj-Napoca,
26, Baritiu Str.
ROMANIA
vasile.dadarlat@cs.utcluj.ro
www.cs.utcluj.ro/~dadarlat

Abstract. Developers of distributed multimedia applications face a diversity of multimedia formats, streaming platforms and protocols; furthermore, support for end-to-end Quality-of-Service (QoS) is a crucial factor for the development of future distributed multimedia and real time systems. Middleware is gaining wide acceptance as a generic software infrastructure for distributed applications, a growing number of applications are designed and implemented as a set of collaborating components using object middleware, such as CORBA, EJB and (D)COM(+), as a software infrastructure that facilitates distribution transparent interactions. However, quality aspects of interactions between objects cannot be specified nor enforced by current component based middleware, resulting only in a best-effort QoS support in middleware. Next generation middleware should offer abstractions for management and control of the system level QoS mechanisms, while maintaining the advantages of the distribution transparencies. The paper presents the QComp framework, which represent our approach for a CCM based implementation, and discusses the design and implementation of a QoS monitoring and control mechanism integrated in the framework. The integration of the QoS support results in a QoS-enabled middleware which is representation, location and QoS transparent.

Key-Words: Middleware, CCM, CORBA, QoS control and monitoring

1. Introduction

Component based software development becomes the most important design pattern for large-scale distributed systems as usually needed in the Telecom domain. "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties" [10].

One of the recent component based frameworks for software development is the CORBA Component Model (CCM) [5]. Although the CCM is a major step forward to support the design and implementation of component based distributed telecommunication systems, it has not sufficiently addressed the non-functional, i.e. QoS related aspects of distributed systems. This paper will present a framework approach that enhances the CCM by generic QoS support.

Our approach enables CCM based software components to negotiate a certain set of QoS...
properties (QoS contract) before they start to interact. This negotiation mechanism can be used in every application domain. In contrast, the concrete QoS contracts that are subject of the negotiation are domain specific. To implement the QoS architecture some changes in the container architecture and language mapping of CCM are needed. The design of the proposed architecture implies only minor changes to the standard. Furthermore it defines additional features (mainly portable interceptors) that are to be plugged into an existing CCM Implementation. It is also possible to switch the QoS support on or off for a given component implementation.

The paper is structured as follows. Section 2 presents the Corba Component model. Section 3 introduces our approach to extend the CCM classic architecture through Container Portable Interceptors in order to integrate QoS support. Section 4 presents a simple architecture in order to make use of defined concepts. Section 5 discuss some related works and section 6 we conclude the paper and we point to some future works.

2. The CORBA Component Model
2.1 The Abstract Model
The CCM specification defines an abstract model and an execution model for the CORBA components as well as a deployment model of components into a distributed environment. In the following, we introduce each of these CCM models.

The CORBA Component Model (CCM) is an open standard which introduces the flexibility and robustness of CORBA in the software components domain, along Enterprise Java Beans (EJB) or Distributed Component Object Model (DCOM). There aren't too many implementations of CCM, but each of them focuses on a certain domain, like clustering, QoS monitoring or real time processing.

A Corba component is the implementation of business logic described by a component designer in Interface Description Language (IDL). The component life cycle (creating, connecting, communicating and destroying) is handled by a Component Container. This way, the developer can focus on behavior, and compatibility in connectivity is enforced through the standard. Each component presents to the environment its functionality through interfaces, attributes and a set of communication ports. Components can connect on functionality through facets and receptacles, when one component uses an interface the other component provides. Another connection type is publishing and consuming events. In CCM a component is managed by a home which behaves like a factory: it creates, configures and finds components it manages.

2.2 The Execution Model
CCM uses a container-based programming model. Component / Container organization has proven to be a good architectural pattern by the separation of concerns it provides. Containers provide the runtime execution environment for CORBA components. A container is a framework for integrating non-functional aspects, such as transactions, security, events, and persistence into a component’s behavior at runtime. All component instances are created and managed at runtime by its container.

CORBA based Middleware is a well-tested and wide spread concept. There are also a lot of approaches to integrate QoS into CORBA based middleware.[7 ] Quality of Service of component based middleware is in the context of telecommunications not investigated sufficiently yet. A generic technology to access QoS categories and parameters for telecommunication services is not yet available. As a relatively new technology the ongoing development of CCM may still be easily influenced by the needs of the Telecom world.

3. The framework
Our framework QComp represent an implementation of the CCM open standard, using C++ programming language. The framework addresses a set of matters specific to the telecommunications domain: the ability to stream media – voice and video, the ability to specify QoS parameters, use and monitor them throughout the component life cycle.
3.1. Container Portable Interceptors
The CORBA specification defines a mechanism which allows the programmer to intercept the communication between two components and even modify it. It is defined as a portable interceptor and it works as a separate class with methods invoked at different points in the communication process: sending, receiving, polling, requesting, etc. Interceptors are very useful for inserting a control level between connected objects. The CCM doesn't benefit from interceptors as they have to be registered directly in the ORB when the distributed application starts and the component container hides the ORB from the components.

In our framework, the interceptors are defined as Container Portable Interceptors (COPIs), which are registered in the component server by the component at activation. COPIs extend the CORBA interceptor functionality and can modify the in and out parameters, the method return value, and even the context, the object representing the container state.

3.2. The extended Component Container
One of the major drawbacks of software developing is that sometimes you have to rewrite everything to provide an extra functionality, because the application wasn't modular enough or well designed. This is also true with CCM. Functionality in the component container is provided by services (transaction, persistence, trading) which are built in the container, and to add a service you would have to come up with your own container implementation. [2].

The component container is extended with an interface that allows services to be registered and accessed by components. Access and registration of COPIs is also done through the extended container.

3.3. QoS support
Communication and service oriented architecture should work on the basis of contracts: the service provider offers several levels of quality or performance which he guarantees to maintain constant throughout the use of the service. QoS is thus used first at the beginning, for contract negotiation where a set of parameters and values have to be agreed upon; then, throughout the life time of the application the parameters have to be constantly monitored in order to respect the contract.

The QComp framework adds to CCM the ability to specify QoS parameters and to negotiate a contract before the connection. Also, an extension is provided that can be used for the monitoring of the parameters.

Accurate modeling of QoS parameters has been the interest of many studies [1]. We propose for the beginning in our framework a simplified version of the OMG UML model for QoS [3]. A ContractDescription contains several QoS dimensions, which have a name and value. This information is stored in Component Property descriptors (an XML-like file accompanying component implementation). Negotiation is implemented as an interface with methods for requesting and accepting connections. The client initiates the negotiation and if it succeeds it receives a reference to the contract so it can renegotiate or stop it.

Fig.2. The QComp Component Container Architecture
The management of QoS parameters is achieved independent of the component implementation through an extension called QoSEnablers. These are components that run in the extended container and implement the negotiation interface. QoSEnablers can register COPIs in the container and control the communication flow in and out of the container or they can be used to monitor parameters. QoSEnablers can use callback functions to control component behavior and they provide an interface for introspection, but they are components and it is up to the developer to implement their behavior and usage.
4. The proposed architecture

After several case studies we have decided that the most appropriate use for QComp's capabilities would be a component based video streaming application with QoS support. This would take advantage of the QoSEnabler and negotiation interface and can also use streaming. We consider QoS to be more important in a video on demand application than in a video conference, because of the high quality the user is expecting for the video stream.

4.1. Enhancements to QComp framework

Quality demands for a high definition video stream are close to real-time: frame rate has to be very high and constant otherwise delays or jumps appear in the video. The main protocol for live or real-time streaming is RTP (real-time transfer protocol). We have decided to integrate RTP support in the streaming module of QComp. Using a specialized protocol for streaming (as RTP) in a component based telecom application is transparent and it only has to be specified at the time of component deployment and connection.

In RTP packets are not immediately sent but first added to a queue and marked with a timestamp. The timestamp can be synchronized with a system clock or through NTP and it indicates the exact time when the packet was created. The server can create the timestamps by itself to reflect a sequence of events, like the equally spaced frames of a movie.

4.2. Streaming policy and system state

Different QoS parameters are connected and influenced by the performance characteristics for the streaming server. As long as the server can hold full control of the processor and keep all the data in memory (so no swapping necessary) we consider that it can maintain constant the QoS parameters and deliver a right level of quality. If the server is at a critical state and a streaming request is received, the policy is to accept that request but only for a lower quality of the video signal.

We monitor the system load using specialized QoSEnabler components. These insert probes into the system which measure how much of the total time is spent by the processor executing user code or what percentage of the memory is being used.

The streaming quality is determined by the following characteristics: the video resolution (the width and height of video frames), the color depth (number of colors), the frame rate.

The video is a simple file and streaming it means that the server reads data from the file and sends it to the client. We could control streaming quality by adjusting resolution and color depth but this would only be affecting the client, not the server, which would need additional processing and further increase the system load. So we only control streaming quality through frame rate. As said before we can use RTP timestamps to suggest the sequence and delay between the frames of a movie. Whenever one of the monitoring probes indicates critical values the quality policy goes down and the streaming server will broadcast at lower quality. Thus the server adapts dynamically to the system state and delivers good quality to the clients.

4.3. System Components

The application is modeled using CCM so the client and server components are deployed on separate component servers. Clients requesting video signal are different hosts so for each streaming connection opened there will be a new component server on the client side. On the server side there is only one component server where the streaming components are dynamically loaded and configured as requests come in. The overall system architecture is illustrated in the collaboration diagram in Figure 3.

![Figure 3. Negotiation component](image)

When the server side of the application is started it will install the monitoring components (QoSEnablers), a negotiation component, a policy component which stores overall streaming
policy and a *loader* component which loads and configures the streaming components.

The monitoring and negotiation components are deployed into the extended QComp container where they can have access to the COPI interface and thus intercept all traffic entering or leaving the component server. When the client requests a video stream it interacts with the loader which eventually spawns the new streaming component and binds it to the client.

The monitoring and negotiation components are deployed into the extended QComp container where they can have access to the COPI interface and thus intercept all traffic entering or leaving the component server. When the client requests a video stream it interacts with the loader which eventually spawns the new streaming component and binds it to the client.

The client side of the application is also designed as a component. The client component receives the RTP video stream from the server, decodes the stream into video frames and outputs them on the screen. This implies that the client has a video decoder incorporated into the component, which we implemented using an external library. Apart from being a simple video receiver and decoder, the client also has to inform the server of the quality of the received data to allow the server to adapt and deliver the right quality.

From the client's viewpoint, video quality means constant frame rate. If delay or variation appears in the frame rate the video will look slow or unsteady and jump from a frame to another. We measure the jitter in the frame rate using the video decoder incorporated in the client component. If the jitter rises over a critical value we tell the server to adjust it's parameters accordingly using a *changePolicy* interface which the server exposes and the client uses.

To connect the client to the server the latter will have to expose internal elements to the client using Naming Service references. The client must call the *startStreaming* interface on the *Loader* component so a reference to the *provides startStreaming* port on the Loader is published at deployment time in the Naming Service.

When the client application starts it must receive the address of the server side Naming Service where it can retrieve the *Loader* interface. It then spawns a component server, a container and the client component home which is used to create the client component. The client component, using the *startStreaming* interface call, requests a new streaming channel from the server. From here on it is the *Loader's* job to install and configure the server component, bind it to the client through a streaming channel and connect them through the *changePolicy* interface.

### 5. Related Work

A lot of work has already been undertaken in the area of QoS for CORBA. Most of these approaches emphasize on the implementation of a couple of concrete QoS categories (e.g. bandwidth reservation) and are not targeted on a generic QoS architecture.

Quality Objects (QuO) [2] is one of the most advanced concepts to integrate QoS into distributed application based on CORBA. QuO stresses also that business code and other code (i.e. QoS supporting code) shall be independent. On the other side the QuO toolkit is not aligned with the CCM and is not directly applicable on CCM based systems.

There is one project that deals with QoS for CCM based middleware. This project is the TAO CCM Project [11]. This project is going to integrate some of the achievements of QuO into their container implementation. The TAO CCM project is specialized for DRE application and is
strongly connected with the TAO ORB. Whereas our approach focuses on the metamodeling approach and the negotiation architecture.

6. Conclusions and Further work

Using the quality of service support implemented in the QComp framework we propose an architecture that allows dynamic creation and adaptation of components. The application for the architecture is a video on demand where quality of service is very important.

We added RTP support to the QComp framework to provide a quality video streaming. Through RTP we control the quality of the video stream using the timestamps. There is a constant communication between client and server, the client providing feedback to the server for the received video stream.

To further improve our architecture we would need to use the information provided by the client to the server, through the RTCP protocol. This provides more low-level information about the stream received by the client and could be used at the RTP server transport endpoint to adapt the quality of the stream in a transparent way. The system could change the source when the quality has to be raised or lowered; changing the frame rate has a serious impact on watching a movie and can even render it unwatchable, as said before.

The QComp QoS framework was developed to support an easier implementation and integration of new QoS providers that support new QoS categories. We also expect that other domains will require other contract types and therefore other QoS providers. To increase the acceptance of our approach a fast and easy way for adding new QoS categories is needed. We have to check whether one of the already existing QoS approaches for plain CORBA based middleware could be easily integrated into the CCM container to enable more advanced QoS category implementations.

References