QuA: Platform-Managed QoS for Component Architectures

Sten Amundsen, Ketil Lund, Frank Eliassen, Richard Staehli

Simula Research Labratory

{stena, ketillu, frank, richard @ simula.no}

Abstract: State-of-the-art middleware and component technologies force application developers to code platform-specific knowledge into the application components. This means that components for applications that are sensitive to Quality of Service (QoS) in practice become dependent on a specific platform, which is contrary to the idea of component architectures. To properly support such applications, component architectures must be extended to enable exchange of QoS-related information and to manage computing resources effectively to meet those requirements.

In this paper, we describe the QuA component architecture. This architecture allows clients to specify logical function (type) and quality requirements for a service and then the platform is responsible for discovering and configuring component implementations that satisfy the requirements, a process we refer to as *service planning*. To demonstrate the feasibility of QuA, we describe an example scenario where video is streamed to mobile terminals. We explain how QuA performs service planning both during initial configuration of the application and then again during a dynamic reconfiguration, to adapt the application to varying resource availability.

1 Introduction

Component technologies promise to enable application development through logical service specification, i.e., by selecting and interconnecting component types. The application developer does not have to consider the physical locations of the components, since access to remote components is hidden by the underlying middleware. The results are improved reuse of existing software, reduced software development cost, higher reliability due to load-balancing and fail-over features, and built-in system management. For these reasons, component technologies and middleware platforms are extensively used by the industry in both large computational systems, like banking, finance, and on-line content; and small applications for hand-held devices, like games, video players and picture editors.

Three component technologies are widely known, both in industry and the research community: CORBA component model (CCM), Enterprise Java Beans (EJB), and Common Object Model Plus (COM+). For each component technology there are corresponding groups of middleware platforms; Common Object Request Broker Architecture (CORBA), Java 2 Enterprise Edition (J2EE) [1] and .NET Framework. These existing solutions have a number of limitations, which are being addressed by the research community [2][3][4][9]. For our research we focus on two such limitations:

- The application components cannot communicate its assumptions and requirements about its environment to the component platform. This means that the application cannot receive any QoS guarantees from the middleware platform.
- Middleware system services (communication, transaction, security, etc.) are hidden inside the platform, making it impossible for application developers to redesign the system services.

The proposed solution is an open, reflective component architecture, called QuA (Quality of service-aware component Architecture). In addition to provide an execution environment for components, it offers hooks where QoS-management components can be attached in order to meet committed QoS-levels during run-time. We refer to this feature as *platform-managed QoS* and the concepts employed to provide this are: 1) reflection and 2) QoS-driven service planning. Reflection is an established solution for adapting the application during run-time [2][8][9], while QoS-driven service planning is a concept that has been introduced with QuA. The architecture also has three properties to achieve the desired flexibility:

- *Component-based*; to simplify the development and maintenance of the middleware itself.
- *Openness*; middleware system services and application components can be deployed and replaced on a need-basis.
- *Small core*; to be able to use the middleware on devices with limited memory and processing power, the core is very small. It has hooks for QuA service plug-ins and can download middleware system service when needed.

This paper is structured as follows: Section 2 gives an overview of the QuA architecture and the service planning concept. Section 3 discusses how a video streaming application can be designed, deployed on QuA, and how QoS guarantees are provided and maintained during run-time. Related work is discussed in Section 4, and finally in Section 5 we present our conclusions.

2 QuA Architecture

At the highest level, QuA is a middleware platform supporting creation and composition of *components* to implement computational services (applications). Thus, an application designer specifies which component types to include, how these components should be interconnected (*bindings*), and the required QoS. An important feature is that components are specified by their logical types (QuA type). The QuA platform is responsible for locating and instantiating implementations of the specified QuA types, as well as for interconnecting them. In addition, QuA is a distributed platform, so the components included in an application may reside on remote platforms.

2.1 QuA Platform

Existing component technologies tend to use the term “component” independently of the context the component is in; design, deployment, or run-time. In QuA, on the other hand, we use a more strict definition of this term, as illustrated in Figure 1. At design-time we have *component type*[1]; and when deployed on the QuA platform, the component type is represented by a *blueprint*; a persistent, immutable value encoding how to implement a component type. Finally, during run-time, *components* are created from blueprints. Application developers design an application only at the logical level, specifying component types and bindings between them.

QuA [5][6] is designed for distributed applications, and provides execution of and communication between QuA components. To support platform-managed QoS, the architecture implements the reflection concept [8], which allows for both inspection and reconfiguration of the application and the middleware platform itself. This reflection concept is realized through a Meta-Object Protocol (MOP) called QuAMOP,

[1] All entities in QuA have a *QuA type*, which represents a contract between the author of the type and the client for its interfaces and functional semantics. Thus, a component type is also a QuA type.

implemented by one of the QuA core objects and reachable from all objects and components in the capsule.

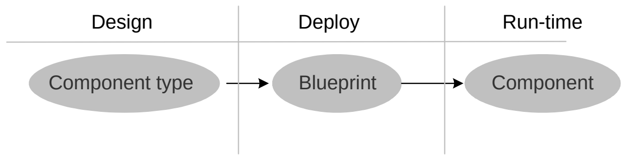

Figure 1: Context-dependent terms

Figure 2 illustrates a run-time view as well as a conceptual view of the QuA architecture. The run-time view in Figure 2a, shows how all QuA objects execute in a *capsule* that provides the essential services of the QuA platform, and manages the address space for both QuA core objects and QuA components. The capsule contains the following entities: the core objects; a set of helper objects implementing platform functionality beyond what is provided by the core; meta-objects to allow reflection on applications; and the objects implementing the application components.

From a conceptual point of view has QuA two levels: meta-level and base-level (see Figure 2b). The base level represents the space in which the QuA components execute, while the meta-level enables reflection. The QuAMOP provides the interfaces for accessing the meta-objects, and through the reflection principle, changes made to meta-objects are implemented on the base level.

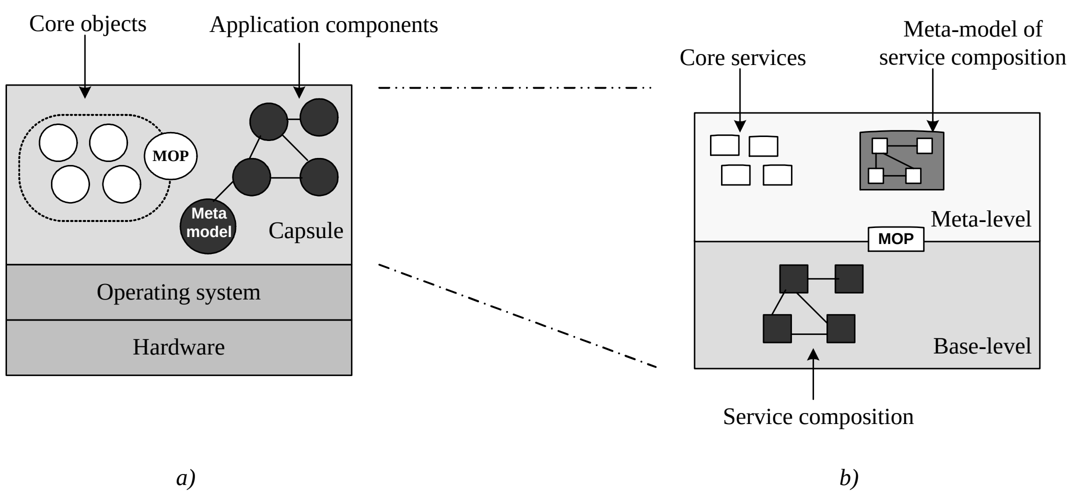

Figure 2: QuA architecture from a) run-time view and b) conceptual view

QuA is designed with a minimal core, which enables us to deploy it on a wide range of computers. Default QuA core services includes life-cycle management and adaptation of service compositions, and the architecture has hooks for inserting QoS-mechanisms. The idea is that application and middleware developers can insert QoS-management mechanisms when available and required. In addition to being reflective and minimal, QuA is an open architecture. Through the QuAMOP, the structure of both the middleware platform and applications are made visible, enabling middleware developers to add middleware system services (like transaction control and security) for a particular installation.

The QuA core is instantiated from a set of object classes. The *QuA* class provides access to the core by implementing the QuAMOP, and the *ServiceContext* class has a default service planner for platform-managed QoS, which may be replaced with specialized planners to optimize QoS-management for a particular application. The *Capsule* class is responsible for run-time platform-specific services for publishing and for interpreting blueprints of components and the application composition.

When a user wants to locate an instance of an application, the *ServicePlanner* first identifies a service composition that meets the user QoS-requirements by requesting the *ImplementationBroker* to retrieve suitable implementation plans. When a suitable plan is found, it is executed. First, all dependencies on other QuA types are resolved, by instantiating the corresponding components, and then the service is created. There are three ways of creating a service: re-use existing instances, create instances from blueprints, or use a factory to create instances. Finally, the plan uses the bind-operation, provided by the QuAMOP, to connect components together. If one object is remote, the local object is bound to a local proxy, which in turn is responsible for communicating with the remote object.

The local *Repository* provides a mechanism for mapping between QuA Names and local volatile objects, and finally, *ResourceManagers* are responsible for admission control, negotiation, monitoring, and reservation.

2.2 QoS-Driven Service Planning

The service planner is an essential part of the QuA architecture, since it, together with the resource manager creates the notion of platform-managed QoS. The service planner is performing what we refer to as QoS-driven service planning: *A process that identifies components and resources that form a correct service composition, which meets a set of QoS-requirements*. The process must have alternative service compositions to choose between, and its criteria for choosing between these are QoS-driven.

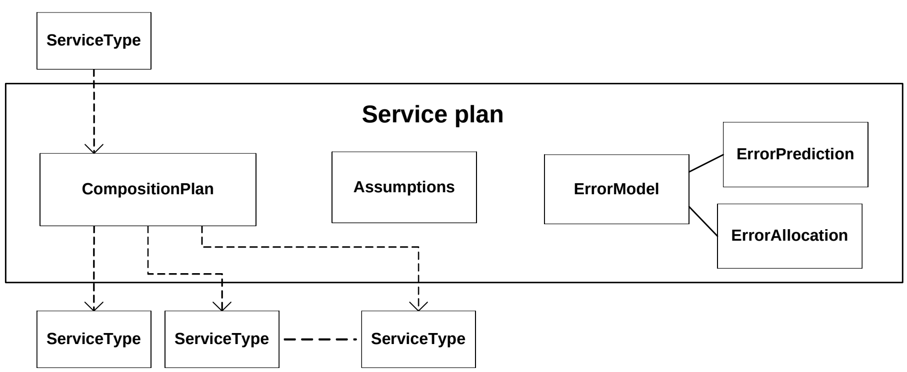

Figure 3: Contents of a service plan and associations to service types

QuA introduces the *service plan* to encapsulate the QoS-properties. Each plan is associated with a QuA type, and application developers may design alternative implementations of the service type, where each implementation has different QoS-properties. Figure 3 illustrates the content of a service plan, how a service type refers to the plan, and how the plan in turn refers to other service types.

The service plan contains five information elements:

- *Assumptions;* A list containing configuration data for the components, including requirements to the execution environment and static dependencies to front-end or back-end systems.

- *Composition plan;* A graph specifying the construction of the service, i.e., composition of service types and the bindings between them.
- *Error model;* The error model expresses quality in a way that differs from the traditional approach, using its own QoS semantics [6][7]. It measures quality relative to perfect quality, i.e. the difference between perfect quality and achieved quality. This difference is called *error*. Formally, this is the difference between ideal and actual output trace, where a trace is the output messages from a service [6]. The ideal output trace, *I*, is defined to be the service output when system resource availability is infinite. The actual output trace, *A*, is defined as the service output with finite and shared system resources. The difference between ideal and actual is a series of error vectors. Figure 4 illustrates the two output traces and the difference vectors, *C*, between them, for *n* output messages. In the figure, the error model is two-dimensional; time and message quality. For some applications, a two-dimensional error model is sufficient, while other systems have a multi-dimensional error model. As an example, a video conference system may have error-dimensions like delay (sec), image resolution (pixels), frame rate (fps), security, etc. The number of ways in which an actual trace may differ from the ideal trace may explode as the complexity of the message structure increases. Fortunately, we are frequently concerned only with an overall measure of distance from the ideal. We therefore define an error model as a set of functions over the series of error vectors. Each error function is typically defined as an aggregating statistical measure such as maximum, mean value or variance. This approach makes error functions useful and suitable for computations.

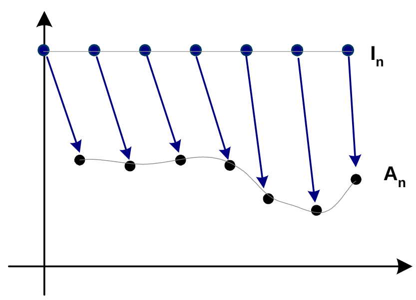

Figure 4: Ideal and actual output trace for a service

- *Error prediction function*; The functions encode the application developer’s knowledge about the service, and predict the error level for the service as a function of error in the sub-services of the service composition and system resource availability (CPU, disk, network, etc).
- *Error allocation function;* The function budgets error for each sub-service in the service composition. The input represents the error-limit for the error-dimension of interest, and the allocation function at the service level map down to error-limits for sub-services. At leaf-level the error allocation function defines resource requirements. These must be expressed in a form that can be understood by resource managers.

Alternative implementations of a service type are specified in different service plans, and deployed onto QuA. When a user wants to instantiate a service, the service planner is invoked and starts the *initial service composition* phase. The service planner takes service type and QoS-requirements as input. Figure 5a shows the interaction the between the service planner object, the surrounding QuA core objects, and a resource manager service type. From the implementation broker, the service planner receives alternative service plans for the service types. Based on the system resource availability, the service planner predicts error along each error-dimension in the error model, and chooses the service composition that best meets the user QoS-requirements. Then, the system resource requirements are estimated by using the error-allocation function. The result is forwarded to appropriate resource managers for reservation and monitoring. Finally, the service composition is instantiated.

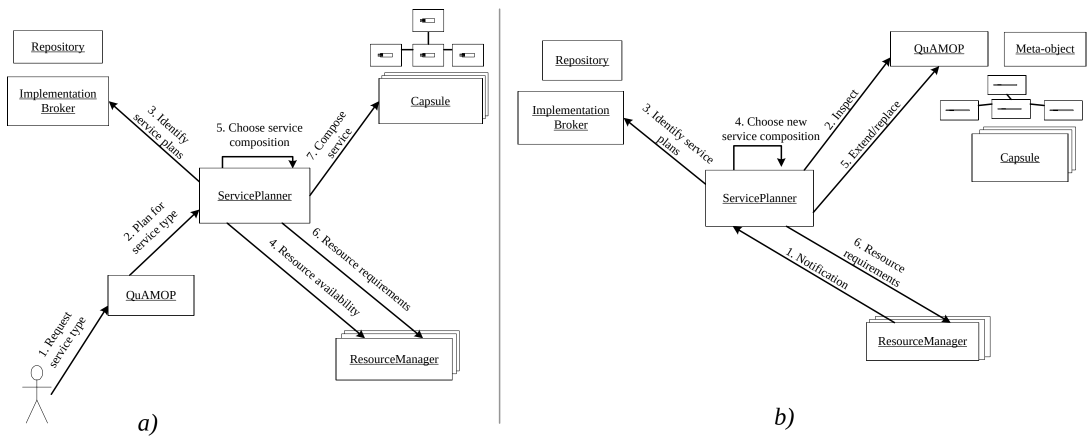

Figure 5: Main interactions during a) initial service composition, and b) service re-composition

The load on the system resources vary over time, since the execution environment is shared with other applications. Resource managers detect these changes and check if the resource requirements are broken. If this is the case, the service planner is notified, which then starts the *service re-composition* phase, illustrated in Figure 5b.

The meta-level in QuA enables the service planner to inspect the running composition. By combining this information with alternative service compositions, derived from the service plans in the repository, the service planner identifies alternative compositions and makes a decision. Using the meta-level, the service planner replaces component types or extends the composition with additional component types.

2.3 Implementation Status

The dominant research method of QuA is using a reference architecture and prototyping. We port applications from areas like multimedia and mobile computing, and by evaluating the prototype and the applications, we investigate how the QuA architecture supports the application domain in question. Our goal is to examine our architecture definition, and to clarify how the architecture compares with the existing alternatives. The result of these investigations may yield revisions to the architecture.

QuA has been documented in a platform independent model (PIM). In addition, we have developed one working prototype in an open-source dialect of Smalltalk and another in Java. The Java version is the reference implementation of the architecture, while Smalltalk has allowed for rapid prototyping, in addition to being a test of our goal of language neutral design. The QuA core has reached a stable state in both versions, and some small QoS-sensitive applications have been successfully ported, both to the Smalltalk and the Java platform. Currently, we are focusing on developing binding type blueprints, as well as other infrastructure for adaptive QoS management, and during June 2004 we also expect that the first open source release of the QuA Java prototype will take place. Finally, a third prototype written in Python is being developed by the University of Tromsø.

3 Video Streaming to Mobile Terminals

A video streaming system is used to describe how QuA provides platform-managed QoS. The system is intended for mobile users and three types of terminals: laptops, personal digital assistants (PDA), and smart phones (see Figure 6). All the terminals as well as the server has a QuA middleware platform installed. The video server is located on the Internet, accessible from local area network (LAN), wireless LAN (WLAN), or cellular network with General Packet Radio Service (GPRS).

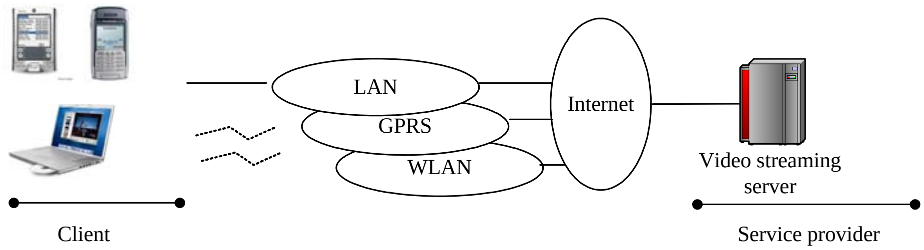

Figure 6: System overview

We assume that videos are stored in MPEG-2 format, and that the available video transcoding algorithms are H.263 and H.26L. The H.263 algorithm is specified for video transportation over low-bandwidth networks and for presentation on devices with small to medium sized screens. H.26L extends H.263 with forward error correction (FEC) to improve the resilience towards bit-errors in the network. Lastly, the MPEG-2 format can be used directly, but requires much network bandwidth.

3.1 Design and Implementation

The application designer specifies that an instance of the component type *VideoBinding* reads an MPEG stream from an *MPEGServer* and writes uncompressed video frames to a *VideoRenderer*. Given that the *MPEGServer* and the *VideoRenderer* are existing components on remote devices; the service planner discovers that there are three alternative implementations for the *VideoBinding* type, as illustrated in Figure 7. Composition 1, specified in the service plan *H263Simple*, employs a H.263 transcoder on the server side to prepare the video frames, and a H.263 decoder on the client side. Composition 2, specified in the service plan *H263Advanced*, has the same transcoder as composition 1, but now configured to apply FEC on the video frames, use a higher resolution, and use a H.26L decoder. Composition 3, specified in service plan *MPEG2*, does not use a transcoder. Instead the *MPEGServer* is bound directly to an MPEG-2 decoder on the client side.

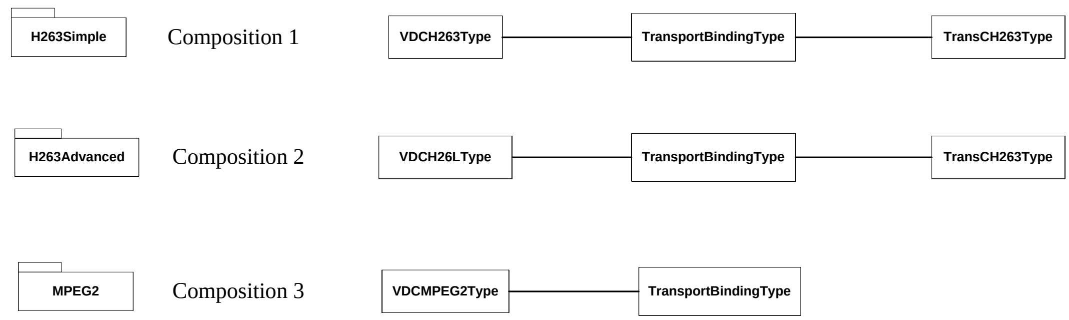

Figure 7: Alternative implementations of the VideoBinding service type

Each component type is developed for a video format with different image resolutions and frame rates. These are assumptions that the error model and error mapping functions are based upon. To keep the example simple, we consider only a small set of parameters when defining the error model and the prediction functions. In a real system there may be additional parameters, and more complex prediction functions.

The error model for the service type *VideoBinding* defines three error-dimensions: delay, frame rate, and correctness. For each error-dimension there is one error function, which defines the difference between the ideal and actual output message traces. For all service types one must relate the service output to the service input, as illustrated in Figure 8. The ideal along the defined error-dimensions are: 1) zero delay, 2) output message rate equals input message rate, and 3) content of each output message is a correct representation of the input messages. The actual output trace along the three error-dimension, are: 1) delay measured from zero to a maximum acceptable delay, 2) frame rate reduction measured from zero to a maximum acceptable reduction, and 3) correctness measure from zero to maximum accepted reduction in correctness. For composition 1 (in Figure 7) the error limits for a useful service have been defined to be: 1) delay from 0 to 1 second, 2) reduction in frame rate from 0 to 19 frame per second (fps), and 3) probability of a reduced correctness from 0.0 to 0.1.

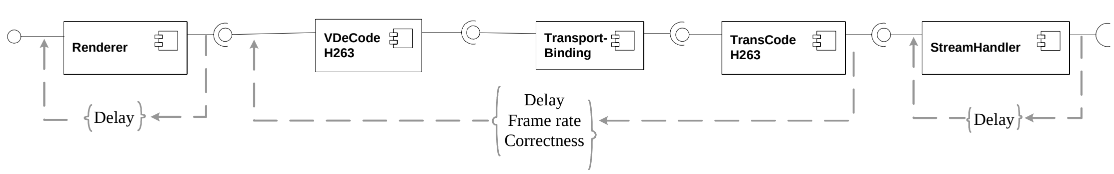

Figure 8: Error-dimensions for composition 1 and the two mandatory component types

Error prediction functions map the availability of system resources to the expected error along each error-dimension. System resources considered relevant for the video streaming system are: memory, CPU, and network capacity. Application developers are free to decide their own format and the complexity of the error prediction functions. Condition statements or a large table with measured errors are two fairly easy approaches to implement error functions. For composition 1, the error prediction function takes available memory, CPU, and network capacity as input, and returns either result set *a* or *b*.

$$RS = (condition)\,?\,[a],[b]$$

$$\begin{bmatrix} Delay \\ FrameRateReduction \\ Correctness \end{bmatrix} = \begin{bmatrix} MEM_{client} > 20kB \wedge \\ MEM_{server} > 1MB \wedge \\ CPU_{client} > 200\ MIPS \wedge \\ CPU_{server} > 700\ MIPS \wedge \\ NET > 9kbit/s \end{bmatrix} ? \begin{bmatrix} 140\ ms \\ 0\ fps \\ 0{,}015 \end{bmatrix}, \begin{bmatrix} 2\ s \\ 19\ fps \\ 0{,}015 \end{bmatrix} \qquad \textbf{(Eq. 1)}$$

Alternative implementations of the *VideoBinding* service type can then be specified in full. Table 1 shows the assumptions, error limits for useful service, and error prediction functions for three alternative compositions; *H263Simple*, *H263Advanced*, and *MPEG2*. For each implementation of a service type, including alternatives, we have specified the service plans and deployed them together with the blueprints in the repository. Service types are published in the implementation broker, and then the service *MVideoStream* is in production and can be requested by the end-users.

Plan / Properties	H263Simple	H263Advanced	MPEG2
Assumptions: - File format: - Image resolution: - Frame rate - Platform:	 H.263 [128*96, ..., 352*288] [20 – 50] Java	 H.263 [128*96, ..., 352*288] [20 – 50] Java	 MPEG-2 [352*288, .., 1400*1050] [20 - 50] Java
Error limits: - Delay - Frame rate reduc. - Correctness	 0 – 2 s 0 – 19 fps 0.0 – 0.1	 0 - 2 s 0 – 19 fps 0.0 – 0.1	 0 – 2 s 0 - 19 fps 0.0 – 0.1
Error prediction: - Delay (ms) - Frame rate reduction - Resilience	IF (MEMclient ≥ 20 kB AND MEMserver ≥ 1 MB AND CPUclient ≥ 200 MIPS AND CPUclient ≥ 700 MIPS AND NET ≥ 9 kbit/s) THEN (EDelay = 140 AND EFrameRate = 0 AND ECorrectness = 0.015) ELSE (EDelay = 2000 AND EFrameRate = 19 AND ECorrectness = 0.015)	IF (MEMclient ≥ 40 kB AND MEMserver ≥ 1 MB AND CPUclient ≥ 400 MIPS AND CPUserver ≥ 700 MIPS AND NET ≥ 35 kbit/s) THEN (EDelay = 200 AND EFrameRate = 0 AND ECorrectness = 0.005) ELSE (EDelay = 2000 AND EFrameRate = 19 AND ECorrectness = 0.005)	IF (MEMclient ≥ 40 kB AND MEMserver ≥ 0.5 MB AND CPUclient ≥ 400 MIPS AND CPUserver ≥ 500 MIPS AND NET ≥ 200 kbit/s) THEN (EDelay = 170 AND EFrameRate = 0 AND ECorrectness = 0.02) ELSE (EDelay = 2000 AND EFrameRate = 19 AND ECorrectness = 0.02)

Table 1: Assumptions, error limits and error prediction functions for service type VideoBinding

3.2 Service Instantiation

When a user wants to view a video, QuA will identify, choose, instantiate, and configure a service composition that meets the end-user's QoS-requirements. The service planner is responsible for identifying and choosing a suitable service composition, while instantiation (and destruction) of component types are managed by the *Capsule*.

The user invokes the local QuA object, specifying the service type *MVideoStream* and QoS-requirements. Initially, we assume that the user is accessing this service from a PDA over an enterprise WLAN. The resource model in QuA is updated with this information, although mechanisms for handling this inside QuA is part of future research. The service planner receives the request for a video and starts the initial service composition phase, as described in Section 2.2. The service planner resolves the service type *MVideoType* and receives the service plan *MVideoPlan* from the implementation broker. This plan specifies the overall composition at a logical level. We assume that the user QoS-requirements have been specified as error limits as shown in Table 1. For each service type specified in the *MVideoPlan*, the service planner searches for corresponding service plans and blueprints.

PDAs have screens with limited resolution, typically around 320 * 240 pixels. Hence, the service planner excludes the *MPEG2* plan based on its assumptions about resolution. The service planner then predicts the error using the prediction function specified in the two remaining plans, *H263Simple* and *H263Advanced*. The predicted error is lowest along the error-dimension frame rate and correctness for *H263Advanced*. The error-dimension delay is worse, but acceptable. So the service planner chooses the *H263Advanced* plan, and resolves all the blueprints from the composition plan (in *H263Advanced*). Finally the service planner uses the error allocation functions to calculate the system resource requirements; *[MEMclient=1MB, MEMserver=3MB, CPUclient=410MIPS, CPUserver= 800MIPS, NET=35kbit/s]*. This resource vector is forwarded to the resource manager for reservation and monitoring. QuA then dynamically loads the blueprints, instantiate the components, and configure them to form the service.

3.3 Service Adaptation

After some time, the user moves out of the WLAN coverage area. This is detected by the application, which instructs the PDA operating system to use the GSM GPRS card. We assume that network connection and the video streaming session are successfully re-established on the new network. Shortly after this, the system resource monitors detects the reduction in network capacity, and notify the service planner about the changes. The service planner then starts the service re-composition phase, as described in Section 2.2.

The service planner resolves the service type *MVideoType* and receives the service plan *MVideoPlan* from the implementation broker. For each service type in the *MVideoPlan*, the service planner resolves all the way down to the blueprints, giving a hierarchy of service types and service plans. The planner compares the service plans for the *VideoBinding* service type, as it did when choosing the initial service composition. The *MPEG2* plan is discarded due to its screen resolution requirement. Predicting the error for the *H263Simple* and *H263Advanced* shows that, the low transfer rate in GPRS gives high error along all error-dimensions for the *H263Advanced* plan. Thus, the service planner selects the *H263Simple* plan. Using the meta-level of QuA, the service planner stops the component instance of type *VDeCodeH26L* and replaces it with *VDeCodeH263*. Then, via the capsule, the component instance of the *TransCodeH263* type is reconfigured to use ordinary H.263 (no FEC) and a 128*96 image resolution. Finally, the service planner uses the error allocation functions to calculate the new system resource requirements; *[MEMclient=1MB, MEMserver=3MB, CPUclient=210MIPS, CPUserver=800MIPS, NET=9kbit/s]*.

4 Related Work

In this section we discuss related work in three areas: component architectures, reflective middleware, and service planning.

Component technologies/architecture used in commercial products, like EJB [10], lack APIs for adding QoS-management mechanisms. The component architecture OpenORB v2 [2] addresses this by introducing component frameworks (CF) as building blocks. Each CF has a set of policies and rules that provides QoS-support. QuA also support CF, but here QoS-specification and service planning are integral parts of QuA, and not solely the responsibility of the application code. Another approach [11] is to extend the EJB container with QoS support in the form of new container components and interfaces for QoS negotiation and adaptation. This work also involves adding network reservation to Java RMI. QuA, on the other hand, is based on a different philosophy, aiming for a technology neutral QoS-aware architecture.

OpenORB employs reflection for run-time reconfiguration of CFs [2]. Meta-models of the underlying structure (at the base-level) are causally connected to components, i.e. changes made on the meta-model causes corresponding changes at the base-level. QuA has adopted the reflection concepts for the same purpose. One OpenORB implementation [14], introduces dynamic QoS-management components that adapt the application according to predefined strategies. These strategies are described in the formal language of timed automata. An automaton can be wrapped in a component type, and is considered a possible technology for QuA resource monitors. DynamicTAO adds reflection to CORBA, allowing inspection and reconfiguration of the ORB [15]. There are hooks for strategies that the ORB uses to implement middleware services, and these may be replaced at run-time. To use the QoS-support in the ORB, the application must be programmed towards a set of APIs. Furthermore, application developers are limited to one particular technology for invoking remote objects/component. Reflective middleware has also been applied to mobile computing, such as CARISMA [9]. Policies coded in eXtensible Markup Language (XML) define application profiles, which deliver different QoS and consume different amounts of system resources. When an application detects changes in the context, it can, via reflection, add or change policies in its profile. In case of a conflict, the middleware performs a “closed-bid” action. CARISMA, like other reflective middleware prototypes, requires application code for QoS handling. QuA, on the other hand, uses service plans to capture the QoS-properties of the applications, enabling the platform itself to perform QoS-driven service planning.

In [11], a QoS compiler is presented that translates user-perceived QoS levels to a run-time ”script” corresponding to our service planning concept. It is assumed that application developers can provide a QoS-specification with knowledge of system-level QoS representations. This approach appears to yield an application that can only be deployed with the particular set of QoS-management services understood by the application developer. Another approach [13] addresses QoS-awareness by capturing non-functional properties in a QoS-model, specifying service compositions as task graphs. It selects services based on QoS criteria, and employs an adaptive execution engine that re-plans the composition of services. Their ontologies and implementations of local and global service planning using integer programming, are designed for Web-services. Hence, system resource management is merely assumed, while QuA uses this as the foundation for predicting and maintaining QoS.

5 Conclusions

In this paper, we have presented the QuA component architecture. Where most programming languages support instantiation using the name of a class or other implementation name, QuA allows programmers to separate QoS requirements from the code implementing the service. The application programmer only needs to specify which component types to use, how to bind them together, and the QoS properties. Components implementing the types have been designed by specialized programmers, and the QuA platform itself is responsible for selecting components that meet the requirements of the application programmer. The QuA architecture adopts an open-reflective approach that allows dynamic reconfiguration of platform services and applications. Pluggable service planners facilitate adaptation by allowing the platform QoS management capabilities to grow, as more sophisticated service planners are developed.

We have illustrated how the QuA platform can be used in a context of mobile terminals and QoS-sensitive applications. As the terminal moves between different networks, the available bandwidth varies considerably, and the application must be adapted accordingly. Our example shows that QuA provides the functionality required, by being able to dynamically replacing components in the application as the available resources varies.

6 References

[1] Sun Micro Systems, *Java 2 platform enterprise edition specification*, v1.4, 2003

[2] Coulson, G., Blair, G., Clarke, M., Parlavanzas, N., *The design of a configurable and reconfigurable middleware platform*, Distributed Computing Journal, 15(2), 2002, pp. 109-126

[3] Wichadakul, D., Nahrstedt, K., Gu, X., Xu, D., *2K^{Q+} An Integrated Approach of QoS Compilation and Reconfigurable, Component-Based Run-Time Middleware for the Unified QoS Management Framework*, In Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms (Midddleware 2001), Heidelberg, Germany, November 2001, pp. 373-394

[4] Trofin, M., Murphy, J., *A Self-Optimizing Container Design for Enterprise Java Beans Applications (Position Paper)*, In 8th international Workshop on Component-Oriented Programming, Darmstadt, Germany, July 2003, pp. 396-397

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

[6] Staehli, R., Eliassen, F., Aagedal, J. Ø., Blair, G., *Quality of Service Semantics for component based systems*, In proceedings for 2nd International Workshop on reflective and adaptive middleware systems, Rio de Janeiro, Brazil, June 2003, pp. 153-157

[7] Object Management Group, *UML profile for Schedulability, Performance & Time*, Final adapted specification, In ptc/2003-03-02, 2003

[8] Maes, P., *Concepts and Experiments in Computational Reflection*. Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA’87), Orlando, FL, USA, October 1987, pp. 147-155

[9] Capra, L., Emmerich, W., Mascolo, C., *CARISMA: Context-Aware Reflective Middleware System for Mobile Applications*, IEEE Transactions on software engineering, 29(10), 2003, pp. 929-945

[10] Sun Microsystems, *Enterprise JavaBeans ™ Specification*, Version 2.1. Sun Microsystems Inc (2002). http://java.sun.com/products/ejb

[11] Miguel, M.A., Ruiz, J.F., Garcia, M., *QoS-Aware Component Frameworks*, Proc. 10th International Workshop on Quality of Service (IWQoS’02), Miami Beach, FL, USA, May 2002, pp. 161-169

[12] Nahrstedt, K., Xu, D., Wichadakul D., Li, B., *QoS-Aware Middleware for Ubiquitous Computing*. IEEE Communications Magazine, 39(11), 2001, pp. 140-148

[13] Zeng, L., Benatallah, B., Ngu, A.H.H., Dumas, M., Kalagnanam, J., Chang, H., *QoS-aware middleware for Web Service Composition*, IEEE Transactions on software engineering, 30(5), 2004, pp. 311-327

[14] Andersen, A., *OOPP: Reflective Middleware Platform including Quality of Service Management*, PhD Thesis, Department of Computer Science, University of Tromsø, 2002

[15] Kon, F., Roman, M., Liu, P., Mao, J., Yamane, T., Magalhaes, L.C., Campbell, R.H., *Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB*, Proc. Middleware 2000, New York, USA, April 2000, pp. 121-143