Quartz: A QoS Architecture for Open Systems[*]

Frank Siqueira[♦] *and Vinny Cahill*

Distributed Systems Group, Department of Computer Science, Trinity College Dublin, Ireland
E-Mail: {Frank.Siqueira, Vinny.Cahill}@cs.tcd.ie

Abstract

This paper describes an architecture that provides support for quality of service (QoS) specification and enforcement in heterogeneous distributed computing systems. The Quartz QoS architecture has been designed to overcome various limitations of previous QoS architectures that have constrained their use in heterogeneous systems. These limitations include dependencies on specific platforms and the fact that their functionality is often limited by design to one particular area of application. Quartz is able to accommodate differences among diverse computing platforms and areas of application by adopting a flexible and extensible platform-independent design, which allows its internal components to be rearranged dynamically in order to adapt the architecture to the surrounding environment. Further significant problems found in other QoS architectures, such as the lack of flexibility and expressiveness in the specification of QoS requirements and limited support for resource adaptation, are also addressed by Quartz. This paper describes the motivations for and design of Quartz in detail, presents a prototype implementation of Quartz, evaluates its design based on experience with a number of applications that use this prototype, and finally compares Quartz to a number of other QoS architectures.

1. Introduction

Despite the evolution of computing platforms, computational resources such as network bandwidth, processing time and memory are still scarce due to the increasing complexity of computer applications. Moreover, there is a category of application that cannot tolerate uncertainty concerning access to computational resources, demanding that the availability of resources be predictable. These applications can have different levels of dependence on the resources provided by the system, ranging from the strong resource availability guarantees required by real-time embedded control systems to the best-effort nature of non-critical Internet-based multimedia applications. With the migration of real-time systems from specialised architectures to more ‘open’ environments, predictable services and guaranteed response times with very low (or even null) error rates are required to support consistent real-time behaviour. On the other hand, multimedia applications such as multi-party conferencing, audio and video broadcast, and distributed co-operative applications are becoming common despite existing limitations of bandwidth for media transfer and processing power to perform tasks such as media compression and decompression. The requirements imposed on the behaviour of the services being provided to an application by the system support are known as quality of service (QoS) requirements.

The main problem faced by applications with QoS requirements is to guarantee that system services will be performed while respecting all of the QoS requirements imposed by

[*] A slightly reduced version of this paper has been published in the Proceedings of the 20th IEEE International Conference on Distributed Computing Systems (ICDCS), Taipei, Taiwan, April 2000.

[♦] This work was performed while the first author was pursuing his Ph.D. in Computer Science at Trinity College.

the application. A myriad of resources may have to be provided by the underlying system to perform a service, ranging from local resources such as memory and CPU to network bandwidth and other remotely located resources. Modern networks and operating systems provide predictable behaviour through the use of resource reservation mechanisms. However, most applications do not benefit from these mechanisms because distributed computing middleware is still being adapted to make use of them.

QoS architectures describe middleware that provides applications with mechanisms for QoS specification and enforcement. These architectures organise the resources provided by the system with the intent of fulfilling the QoS requirements imposed by their client. Many different types of hardware, operating system and network infrastructures and protocols coexist in open systems, and multiple resource reservation protocols populate this complex environment. Nevertheless, applications with QoS constraints expect similar behaviour from the underlying system support independently of the particular characteristics of the hardware, operating system and network support present in the lower-level platform. Consequently, allowing applications to reserve resources via a middleware layer implies that the differences between reservation protocols have to be masked by the middleware itself.

Substantial work on QoS architectures can be found in the literature (see [1] for a survey). However, the architectures proposed so far consider only part of the overall problem of QoS specification and enforcement [2].

Our focus in the study of QoS architectures is on the provision of QoS-constrained services in open, heterogeneous and distributed computing systems. The QoS architectures proposed so far typically have a strong dependency on a particular computing platform. Real-time operating systems combined with ATM are the most popular platforms for the development of QoS architectures because of their suitability for the implementation of QoS mechanisms for resource reservation. Examples of such architectures are QoS-A [3] and Xbind [4]. This tight dependency on a specific platform constrains their application in open environments, where heterogeneity is an intrinsic characteristic. Some architectures are also targeted at particular application areas, with distributed multimedia being the one where the technology is most mature because of several research projects that have explored this topic (see [5] for a review of QoS in distributed multimedia systems).

In addition, other important problems can be identified in the QoS architectures presented in the literature. Some architectures constrain the expressiveness of the user in the specification of QoS requirements and lack transparency from the lower level, forcing the user to deal with a notion of QoS that is not familiar for him. In some cases, due to the tight integration of the architecture with the lower-level platform, the user must know the characteristics of the available reservation mechanisms in order to make use of the architecture, while a higher level of transparency would be more appropriate. Furthermore, in most architectures support for resource adaptation is very limited, if not completely absent.

In this paper we present Quartz, a generic QoS architecture that addresses the limitations of previous proposals in this area [6]. This is achieved by adopting a highly flexible, extensible, component-based platform-independent design, which supports user transparency from the underlying system and at the same time is suitable for open distributed systems.

The remainder of this paper is organised as follows. Section 2 surveys this area of research. Section 3 explains in detail the proposed QoS architecture. Section 4 presents a prototype implementation, describes a number of applications that have been built on top of this prototype, analyses the obtained results and compares Quartz to other QoS architectures. Finally, section 5 presents some conclusions and plans for future work.

2. Quality of Service

'Quality of Service', or QoS for short, is the keyword used to represent the set of requirements imposed by a user (human being or software component) on the behaviour of the services being provided to an application by the underlying system support.

QoS is defined by the ISO OSI/ODP group as 'a set of qualities related to the collective behaviour of one or more objects' [7]. Other authors try to clarify this definition. For example, Vogel et al. [5] state that QoS 'represents the set of quantitative and qualitative characteristics of a distributed multimedia system necessary to achieve the required functionality of an application'. We adopt a very similar definition, except that we do not constrain the application of QoS to distributed multimedia systems, but also extend the application of QoS to any system with constraints related to response time, performance, and/or output quality. This includes, besides distributed multimedia, other areas such as real-time systems, cooperative work and high capacity storage servers.

ISO, along with the concept of QoS, defines a complete terminology for dealing with QoS. Their concern is mainly with the application of QoS to the specification of communication services at network level. We prefer to adopt their terminology slightly modified to encompass diverse areas of application.

2.1. Resource Reservation

The concept of resource reservation provides the predictable system behaviour necessary for applications with QoS constraints. Reservation mechanisms have to keep track of the use of the limited set of resources provided by the system, and receive requests from new users interested in using these resources. New requests are subject to admission tests based on current resource usage and the guarantee levels requested by the user. Reservations are then accepted, if enough resources are available, or rejected if not. The problem of allocating limited resources becomes even more complex if we consider that current computational systems are basically heterogeneous, subject to mobility and constant reconfiguration, but still have to provide a dependable and accurate service in a limited response time.

Mechanisms for resource reservation are being incorporated into networks and operating systems in order to guarantee the availability of resources for applications. The concept of resource reservation, as well as QoS, originated in work on communication networks and was subsequently extended to other components of computational systems. In the area of computer networks, the development of ATM [8] represented a significant advance towards the provision of QoS-constrained communication services. Aiming to provide similar behaviour, but working at the logical network level, the IETF is adding reservation capabilities to its suite of protocols, including the resource reservation protocol (RSVP) [9], which handles QoS at the network level, and the real-time transport protocol (RTP), which works at the transport level. At the operating system level, some work has been developed to extend operating systems to provide more predictable behaviour suitable for applications with QoS constraints. Real-time operating systems, such as QNX [10] and Chorus [11], have mechanisms that provide time-constrained services. Following the same direction, desktop operating systems such as Linux [12] and Windows NT [13] are being adapted to provide behaviour suitable for applications with QoS constraints.

Despite providing an important contribution towards the provision of QoS for applications, resource reservation protocols are situated at a low level of abstraction, which is not suitable for the application programmer to deal with.

2.2. QoS Architectures

QoS architectures are responsible for integrating QoS mechanisms in computational systems in order to organise the resources provided by the system in a consistent manner with the intent of fulfilling the QoS requirements imposed by the user. In other words, QoS architectures aim to fill the gap between resource reservation protocols, situated at a low level of abstraction, and the application level.

To allow the utilisation of the mechanisms provided by networks and operating systems with resource reservation capabilities at user level, several QoS architectures have been defined in the literature [1]. However, most of these architectures have limitations in the way they allow QoS to be specified, or related to the way they enforce QoS using the resources provided by the underlying system support. These architectures typically target only a specific configuration of processing and communication hardware, constraining their utilisation in open, heterogeneous systems. Furthermore, support for dynamic resource adaptation is typically limited or completely absent. These drawbacks, and the strategies adopted by us with the aim of solving them, are discussed in more detail in the next section.

3. The Quartz Architecture

We have designed and implemented a QoS architecture with the intent of addressing the limitations of previous proposals in the area. The Quartz architecture is based on a highly flexible, extensible, and platform-independent design that allows it to be used in different application areas and in conjunction with a variety of different resource reservation protocols.

3.1. Handling Heterogeneity at Application and System Level

The main goal considered in the development of Quartz was to provide support for heterogeneous environments. This implies that the architecture should be able to handle the different protocols and hardware that can coexist in an open, distributed and heterogeneous platform. Similarly, the architecture is expected to provide support for very diverse applications, which may have different ways to express and handle QoS requirements.

Figure 1 illustrates the use of the Quartz QoS architecture in a heterogeneous environment. Applications requiring QoS enforcement use the mechanisms provided by Quartz to specify their requirements. In order to enforce the required QoS, Quartz employs the resource reservation protocols available in the target network and operating system.

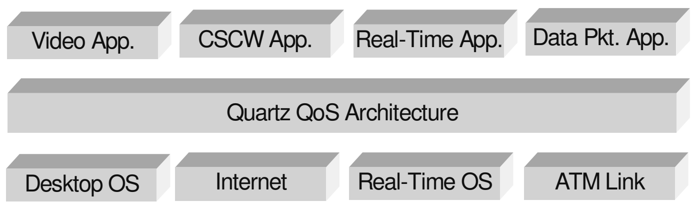

Figure 1– Quartz in a Heterogeneous Environment

In order to handle heterogeneity, Quartz must not only be capable of being ported to different platforms, but it also has to be capable of handling QoS for an application when the lower-level resource reservation protocol changes without requiring recompilation. For example, if the application is able to transfer data using both ATM and TCP/IP, the QoS architecture has to be able to perform QoS reservations for both protocols by adapting itself internally instead of requiring a new port of the architecture to be linked to the application.

This level of flexibility is achieved by Quartz by adopting an architectural design based on interchangeable components, in which components able to handle QoS for different reservation mechanisms can be plugged into the architecture dynamically. In addition, support for new reservation protocols can be added to the architecture without the necessity of porting the whole infrastructure. Instead, a new component that interacts with the new reservation protocol can be written by the programmer.

3.2. QoS Specification and Translation

QoS parameters have to be translated between different levels of abstraction to be meaningful for the mechanisms present at a particular level. Two main levels of abstraction can be identified: the application level and the system level. Requirements specified at different levels are related, but differ strongly in their interpretation. An application parameter is generally related to an idea present only at this level, for example the number of frames of video shown per second in a video broadcast application. At system level this corresponds to requirements on the network bandwidth needed to transfer data, the processing time needed to handle the information, the amount of memory used by the application, etc.

For the user it is easier to abstract from the system level and concentrate on his own view of quality. However, many QoS architectures do not provide mechanisms for mapping QoS requirements between different levels of abstraction, forcing the user to deal with a system-level notion of quality that may not be clear for him. Furthermore, the application area in which a QoS architecture can be utilised varies enormously. For example, a QoS parameter such as 'frequency range' for an audio application would be completely meaningless for an application based on data transfer. The same could be said about a parameter such as 'window size', useful for video but not meaningful for audio. Therefore, a balance must be achieved between the needs of different application fields regarding the manner in which QoS requirements are expressed and the generalisation necessary for the architecture to be deployed over heterogeneous platforms. Any attempt to define a common set of parameters for QoS specification to be employed by the application to specify QoS requirements would constrain its expressiveness in a very damaging manner. Consequently, the mechanisms for QoS specification provided by Quartz must be flexible enough to accept different formats of QoS parameters and extensible in order to recognise a potentially infinite set of parameters.

The QoS parameters specified by the application must be interpreted appropriately by Quartz in order to perform the reservation of resources at the lower level. This implies translating the parameters from their original format into parameters that are understood internally by the architecture. In order to translate parameters, a mapping must be established between parameters at different levels. Mappings are not usually one-to-one between parameters, but may be one-to-many, many-to-one or many-to-many. This implies that resources might be interchangeable, and that balancing requirements and resources is another task that has to be performed by the architecture. Although the whole mapping may be complex, the process of translation typically consists in simple arithmetic operations over a limited set of variables.

For the particular case in which several different application areas and reservation protocols must be supported, the translation process has to deal with different sets of parameters appropriate for the environment into which it is inserted. Since the creation of direct (one-step) translators for X application fields deployed on top of Y resource reservation protocols would need the definition of X * Y translators, this solution seems to be unacceptable. The addition of support for a new kind of application should not imply writing one translator for every reservation protocol supported, and vice-versa. Quartz addresses these issues through the use of an extensible, multi-step translation unit.

In order to avoid having a translator for each combination of application field and reservation protocol, Quartz adopts a three-step translation process. Applications specify their application-specific QoS parameters, which are first translated into a set of generic application-level parameters defined by Quartz. These parameters are further translated into a set of generic system-level parameters and balanced between the network and the operating system. Finally, generic system-level parameters are translated into the system-specific parameters understood by each of the reservation protocols present in the underlying system.

The sets of generic application-level and generic system-level parameters recognised by Quartz during the translation process are listed in Table 1 and Table 2 respectively. Parameter names are suffixed by a tag that identifies the corresponding abstraction level. Threshold values can be specified by suffixing parameter names with '`Max`' for specifying maximum values and '`Min`' for minimum values.

Parameter Name	**Description**
`App::DataUnitSize`	Size of data units produced by the application (in bytes)
`App::DataUnitRate`	Rate of data units produced by the application (in units/s)
`App::EndToEndDelay`	Time between data production and consumption (in µsec.)
`App::ErrorRatio`	Acceptable error (in bits per million)
`App::Guarantee`	Level of service guarantee (best-effort, deterministic, etc.)
`App::Cost`	Financial cost (currency per data unit or per second)
`App::SecurityLevel`	Security mechanism (none, encrypted, etc.)

Table 1 – Generic Application-Level QoS Parameters

Parameter Name	**Description**
`Net::Bandwidth`	Bandwidth provided by the network (in bits/s)
`Net::PacketSize`	Size of data packets (in bytes)
`Net::Delay` and `OS::Delay`	Transmission and processing delays (in µsec.)
`Net::ErrorRatio`	Acceptable transmission error (bits per million)
`Sys::Guarantee`	Levels of service guarantee for both network and O.S. (best-effort, unloaded, deterministic, etc.)
`Net::Cost` and `OS::Cost`	Financial cost (currency per connection or time)
`Sys::SecurityLevel`	Security mechanism (none, encrypted, etc.)

Table 2 – Generic System-Level QoS Parameters

These sets of generic parameters have been chosen based on the generic notion of QoS present at the corresponding abstraction level. Despite the generalisation necessary for the middleware to be able to handle these parameters, the power of expression of the application is not affected because requirements are expressed by using application-specific parameters. Since the generic parameters are close to the notion of QoS present at each level of abstraction, it is easy to establish an efficient mapping and perform a low-complexity translation process between the generic parameters and the application and system-specific sets of parameters.

Table 3 illustrates the transformation undergone by a parameter at the different levels of the translation process (in this case, audio quality is translated into a set of RSVP parameters). Table 4 illustrates the case of a parameter (in this example, the overall delay) that must be balanced between the network and the operating system.

Parameter Set	**Parameter Values**
Application-specific Parameters	`Audio::Quality` = CD (44KHz, 16 bits per sample)
Generic Application-level Parameters	`App::DataUnitSize` = 2 bytes; `App::DataUnitRate` = 44k/s
Generic System-level Parameters	`Net::Bandwidth` = 88 Kb/s
System-specific Parameters	`RSVP::TokenRate` = 88Kb/s; `RSVP::BucketSize` = 88Kb; …

Table 3 – Example of Parameter Translation

Parameter Set	**Parameter Values**
Generic Application-level Parameters	`App::EndToEndDelay` = 500 ms
Generic System-level Parameters	`Net::Delay` = 300 ms; `OS::Delay` = 200 ms

Table 4 – Example of Parameter Balancing

Quartz is also required to allow dynamic changes in the distribution of resources to be performed by the system. This must occur without causing loss of service consistency at application level. Any change in the reservation of resources at lower-level must be reported to the application by using QoS parameters that are understood at high level. This implies that the QoS architecture has to perform a reverse translation of parameters before informing the application that QoS has changed.

3.3. QoS Enforcement and Resource Reservation

Quartz must provide transparency of QoS and reservation mechanisms from the application's point of view. This implies that the interaction with the reservation protocols present in the underlying system, which is necessary to guarantee the QoS to be provided to the application, must be performed by Quartz. However, different resource reservation protocols may be present in an open environment, and each of the existing reservation protocols has its own interface and its own mechanisms for resources allocation.

Quartz is able to interact with different reservation protocols by defining, for each reservation protocol, a component that encapsulates all the mechanisms necessary for interacting with it. By adopting this strategy, we hide from the application the differences between the way different protocols allow resources to be reserved. This has the important effect of increasing the portability of applications across different platforms, and makes it easier to extend the architecture in order to support new resource reservation protocols. The components defined by the Quartz architecture will be described in detail in section 3.5.

3.4. QoS Adaptation

One important trend in the area of resource reservation protocols is the provision of support for resource adaptation [10]. Initial studies in this area defended the provision of deterministic guarantees in the allocation of resources, which would be valid for the entire lifetime of the application that requested the resource reservation. However, several drawbacks appear in efforts to provide completely guaranteed resource reservation due to the impossibility of guaranteeing the availability of resources in computer systems subject to hardware reconfiguration or failure. Aiming to overcome this problem, another school of thought proposed the development of adaptive applications to deal with the changes in resource availability during the provision of service. However, pure adaptation does not solve the problems faced by applications with strong QoS requirements, which are not satisfied by the best-effort systems currently available.

A third idea based on resource adaptation, which mixes both approaches mentioned previously, has been considered as a viable and necessary alternative to both. Resource reservation combined with adaptation yields a more flexible approach for providing QoS to applications. In this approach, resources are seen by applications as guaranteed during some time, but their availability can vary over long periods. This technique allows resources to become unavailable due to reasons such as hardware failure, system reconfiguration, or because they are required by an application with higher priority. Applications are responsible for estimating their initial resource requirements and for negotiating their reservations with Quartz. In addition, applications have to be able to adapt their behaviour at run time based on feedback received from Quartz.

Quartz provides support for QoS adaptation at both system and application levels. In the Quartz architecture, some QoS requirements such as cost and delay are defined by the sum of resources provided by both the operating system and the network. Consequently, losing resources from one source may be compensated by requesting more resources from another source. When this is possible, the adaptation occurs only at the system level, completely transparent from the application's point of view, and the quality seen by the application is not affected. If adaptation at system level fails, Quartz notifies the application, which has to adapt its requirements in order to decrease the consumption of resources. This can be done for example by reducing the quality of a video stream or changing the compression method used for data transfer.

The notification message sent by Quartz to the application carries QoS parameters understood at application level, which reflect the changes in resources reserved at system level. During this process, a set of system-level QoS parameters is translated into application-level QoS parameters by using the reverse translation path provided by the translation components.

3.5. Architectural Components

Each component defined by Quartz encapsulates a particular task in the overall problem of QoS specification and enforcement in an open, heterogeneous environment.

The *QoS agent*, the central component of the Quartz architecture, is responsible for implementing the QoS mechanisms necessary for the provision of services with the quality requested by the user. This involves two main tasks: the translation of QoS parameters between different levels of abstraction, and the interaction with the underlying reservation mechanisms provided by the resource reservation protocols present in the system. The QoS agent, as illustrated by Figure 2, is composed of a *translation unit* and multiple *system agents* associated with the reservation protocols responsible for administering the use of resources.

The translation unit contains a *QoS interpreter* and *QoS filters*. QoS filters can be subdivided into *application* and *system* filters, which are responsible for translating their respective sets of QoS parameters to and from the generic set of parameters at the same abstraction level. The QoS interpreter establishes the mapping between the two sets of generic parameters defined by Quartz. During this process, the *balancing agent*, which is basically a resource trader encapsulated by the interpreter, balances the usage of resources between the network and the operating system. When either the operating system or network reduces the resources allocated to the application due to resource adaptation, the balancing agent tries to compensate for the loss of resources on one side by requesting more resources from the other. If this process succeeds, nothing changes from the application point of view, but when it fails, the application must be notified and asked to adapt its requirements.

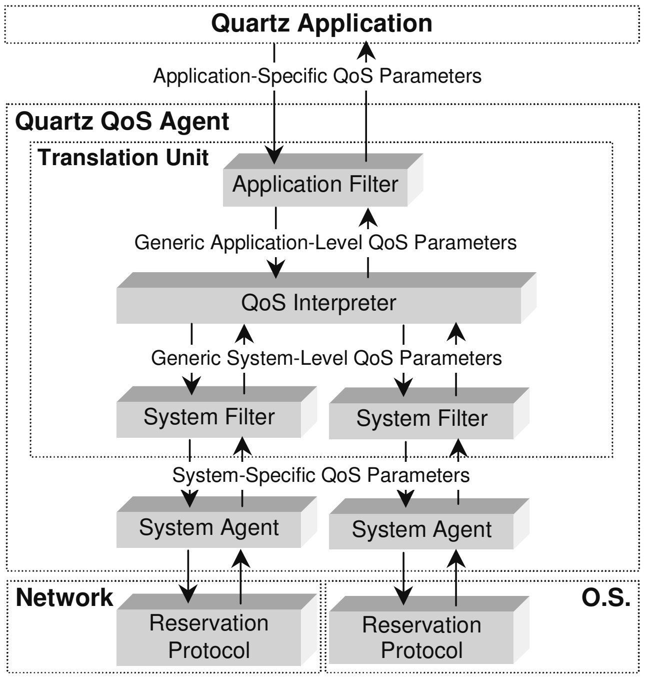

Figure 2 – Detailed Structure of the QoS Agent

Finally, the *system agents* use the values of the QoS parameters provided by the translation unit to perform the necessary reservation of resources using the corresponding reservation protocol. Each system agent is familiar with the public interface of the corresponding reservation protocol, being able not only to request reservations but also to monitor the usage of the resources allocated to it and to receive notifications from the protocol informing it of the occurrence of resource adaptation.

The Quartz architecture supports heterogeneity by encapsulating the QoS mechanisms necessary for interacting with a specific resource reservation protocol or application area into a replaceable component with a standardised interface, which are plugged into the architecture whenever the associated protocol or application area is in use. Changes at application level can be accommodated by replacing the application filter. Similarly, changes at system level imply the replacement of system filters and agents. These components may be selected from a component library provided by Quartz or implemented by the user.

4. Implementation,Validation and Evaluation

We have developed a functional prototype of the Quartz architecture in order to analyse its behaviour when supporting applications with QoS requirements. In this section we present this prototype and a number of applications built on top of it for validation purposes. Finally, we evaluate Quartz in face of the requirements imposed on it and compare it to the other architectures presented in the literature.

4.1. The Quartz Prototype

The prototype is composed of a set of fixed components that form the common core of the architecture and a series of replaceable components that can be plugged into this core whenever necessary. The prototype was written in C++ on top of Windows NT.

The common core of the implementation consists of the components of the architecture that are independent of the lower-level platform and of the application area. Two categories of components make up the Quartz core: components responsible for translation of QoS parameters, such as filters and the QoS interpreter, and the agents, such as network and operating system agents and the QoS agent. Two interfaces are defined by Quartz, one for each categories of components. In addition, one interface is defined for allowing interaction between the components of the architecture and their clients.

Two standard translation components make up the Quartz core: the default QoS interpreter and the bypass filter (the default filter, which simply forwards QoS parameters untouched).The interface supported by translation components, called `QzTranslation`, is described in Pseudo-IDL in Figure 3. This class defines the methods that perform translation of QoS parameters in both directions of translation, handling parameters stored in a data structure called `QzQoS`.

```
struct QzQoSParam {
  string            param_name;
  unsigned long     param_value;
};

typedef sequence <QzQoSParam> QzQoS;

interface QzTranslation {
  QzQoS TranslateQoSDownwards ( in QzQoS upper_qos );
  QzQoS TranslateQoSUpwards   ( in QzQoS lower_qos );
};
```

Figure 3 – Pseudo-IDL of Translation Components

The standard interface of a system agent is called `QzSystemAgent`. This interface defines the operation `QoSRequest`, which is used by the application to specify QoS requirements. This interface is supported by the network and operating system agents and by the QoS Agent. The QoS Agent is a skeleton in which other components are plugged in and out in order to reflect changes in the surrounding environment. The replacement of components encapsulated by the QoS Agent does not affect the interfacing with the user, who still sees the interface of the QoS Agent as the standard means of interaction with Quartz.

The application has to support the upcalls defined by class `QzUpcall` which are used for reporting changes in the QoS seen at application level due to resource adaptation. Applications using Quartz must support this interface by inheritance or encapsulation. The QoS Agent also inherits from `QzUpcall` in order to receive notifications issued by network and operating system agents, which are handled and forwarded to the application if needed. Figure 4 shows the Pseudo-IDL of the system agent and the upcall interface.

```
interface QzSystemAgent {
  void QoSRequest   ( in QzQoS requested_qos );
};

interface QzUpcall {
  void QoSReport    ( in QzQoS adapted_qos );
};
```

Figure 4 – Pseudo-IDL of the System Agent and the Upcall Interface

The Quartz prototype has also system agents and filters for RSVP, ATM, and for the real-time mechanisms provided by Windows NT, which are presented in the following sections.

4.2. The RSVP Sub-System

We have implemented a system filter and a system agent for the RSVP protocol. In addition to the network support provided by the operating system, we use the implementation of RSVP developed by Intel, called PC-RSVP, currently in beta version.

The RSVP filter is responsible for translating QoS parameters understood by the RSVP protocol, and supports the public interface `QzTranslation`. The translation process occurs in two ways, downwards from the generic set of system-level QoS parameters defined by Quartz into RSVP-specific QoS parameters, and upwards from RSVP-specific parameters into generic system-level parameters. The QoS parameters defined for RSVP use a token bucket to model the data traffic. The parameters recognised by the RSVP filter are listed and described by Table 5.

Parameter Name	Description
`RSVP::TokenRate`	Rate in which tokens are produced (in bytes/s)
`RSVP::BucketSize`	Size of the bucket (in bytes)
`RSVP::PeakRate`	Maximum data rate (in bytes/s)
`RSVP::MinPoliced`	Minimum amount of data subject to the policy (in bytes)
`RSVP::MaxPktSize`	Maximum packet size (in bytes)
`RSVP::Rate`	Rate (in bytes/s; only for deterministic service)
`RSVP::SlackTerm`	Slack (in microseconds; only for deterministic service)
`RSVP::FlowType`	Type of data flow (deterministic, best-effort, etc.)
`RSVP::DataTTL`	Time to live (in hops; local area by default)
`RSVP::ReservationStyle`	Style of reservation filter (fixed filter by default)
`RSVP::Policy`	Policy to be used by the policy control component

Table 5 – RSVP QoS Parameters

The RSVP agent performs resource reservations upon receipt of a call to `QoSRequest` and calls method `QoSReport` on the upcall interface when application-level adaptation is necessary.

4.3. The ATM Sub-System

A system filter and system agent for ATM networks have also been implemented. ForeRunner LE 155 Mbps PC cards and a Fore Systems ASX 100 switch have been used for this purpose. We also rely on the WinSock2 service provider that is supplied by Fore Systems together with the hardware.

The parameters defined for the ATM sub-system are listed and described by Table 6. These parameters correspond to the fields of the data structure used for performing resource reservations using Fore Systems' ATM Service Provider for WinSock2. Consequently, the ATM Agent just has to collect this information, fill in a data structure and call the appropriate routine provided by WinSock2 in order to perform a reservation.

Parameter Name	Description
`ATM::PeakCellRate`	Max. rate in which cells are produced (in cells/s)
`ATM::SustainableCellRate`	Long-term cell rate (in cells/s)
`ATM::MaxBurstSize`	Maximum cell burst (in ms)
`ATM::QoSClass`	Type of data flow (CBR, VBR, best-effort, etc)
`ATM::Tagging`	Tag non-compliant cells as subject to be discarded

Table 6 – ATM QoS Parameters

The ATM filter translates ATM parameters understood by the ATM Agent and, like other filters, supports interface `QzTranslation`. The translation process occurs in two ways, downwards from the generic set of system-level QoS parameters defined by Quartz into ATM-specific QoS parameters, and upwards from ATM-specific parameters into generic system-level parameters.

The ATM agent gets the ATM QoS parameters and performs reservation of bandwidth by interacting with the ATM service provider for Winsock2. In addition, it reports any change in QoS occurred in the network by issuing upcalls through the `QzUpcall` interface.

4.4. The Windows NT Sub-System

At the operating system level we have adopted Windows NT as the platform for the deployment of this prototype of Quartz. As a result, a system agent and a filter have been developed for this operating system.

The provision of QoS in Windows NT is limited. We make use of the real-time priority class and of mechanisms for memory locking to provide a more predictable service, which is still non-deterministic. Consequently, only two QoS parameters are defined for Windows NT:

- `WinNT::PriorityLevel`: defines the priority level of a process; used by the operating system to schedule access to the processor.
- `WinNT::MemoryPaging`: determines if the memory allocated by the process will be subject to paging operations, which introduce unpredictable delays and may degrade performance.

The Windows NT filter translates between Windows NT parameters and the generic system-level parameters defined by Quartz in both ways. It inherits from class `QzTranslation` and implements the translation methods defined by this class.

The system agent for Windows NT sets the priority level of the client application and controls the execution of paging operations. In addition, it issues upcalls when the requested guarantees (e.g. deterministic guarantees) cannot be provided.

4.5. The RCP Application

A remote copy daemon and client, equivalent to the UNIX '`rcp`' daemon and command, have been implemented using the prototype. This application is able to use either TCP, UDP (including multicast) or ATM for data transfer, and a graphical interface allows the user to select the required protocol and the desired QoS parameters. Quartz was used as a means of reserving resources for the multiple network supports without adding complexity to the application. According to the network reservation protocol being used, a suitable pair of system agent and filter is plugged into the QoS agent. The RSVP agent and filter are used for TCP and UDP, while ATM requires its own filter and agent.

Parameter Name	**Description**
`DPkt::PacketSize`	Size of packets (in bytes)
`DPkt::DelayBetweenPackets`	Time between production of two packets (in μs)
`DPkt::EndToEndDelay`	Time between production and consumption of a packet (in μs)
`DPkt::ErrorRatio`	Acceptable error ratio (in bits per million)
`DPkt::Guarantee`	Guarantee level (best-effort, deterministic, etc.)
`DPkt::SecurityLevel`	Security level (none, encrypted, etc.)

Table 7 – Parameters Recognised by the Data Packet Filter

In order to handle the notion of QoS understood at application level, we have implemented a data packet application filter, which interprets QoS as understood by applications transmitting data packets. The QoS parameters understood by this filter are described by Table 7. A clear mapping may be noticed between these parameters and the generic application-level parameters presented in Table 1. This mapping is implemented by the data packet application filter.

The remote copy application allows the user to specify QoS requirements by providing values for packet size and packet rate as well as service guarantee (i.e. best-effort, unloaded or deterministic) through a graphical interface. These values are passed to the QoS agent and then processed by the data packet filter, by the QoS interpreter, which translates and balances requirements, and by the corresponding component filters. Finally, the corresponding component agents reserve the necessary lower-level resources by interacting with the reservation protocols supported by the network and the operating system.

4.6. Evaluation and Analysis

Important conclusions can be reached based on the observation of the remote copy example and on the results of performance measurements executed with this example.

The remote copy example shows that the resource provider can be changed without interfering with the application code. Independently from the reservation protocol used at the network level – i.e. RSVP or ATM – equivalent behaviour was observed from the application's point of view in regard to the provision of QoS. This shows that, by using Quartz, the reservation mechanism became transparent for the application despite the different characteristics of the lower-level reservation protocols. Consequently, applications using Quartz are highly portable, since the code necessary for requesting QoS behaviour is kept unchanged independently of the underlying system that is providing resources for the application. The use of different component agents and filters in this example shows that Quartz can be used in different platforms, and that the filters can be combined freely in order to reflect the characteristics of the underlying system.

Performance tests have shown that the overhead added by Quartz to the application is very small. Table 8 shows typical values of the overhead imposed by Quartz for the remote copy application. This data was obtained on a Pentium Pro 200 MHz by using the profiling tools that accompany Microsoft Visual C++ 5.0.

	ATM	RSVP
Initialisation of Reservation Protocols	N/A	24.859 ms
Initialisation of Quartz	346 μs	9.272 ms
Total Overhead per QoS Reservation	1.177 ms	1.220 ms
composed of: QoS Specification	93 μs	113 μs
QoS Translation	759 μs	991μs
Resource Reservation	325 μs	116 μs

Table 8 – Overhead Imposed by Quartz

The total overhead caused by Quartz in a single reservation (i.e. the time taken to specify, translate and interact with the resource reservation protocols) is of about 1.2 millisecond for both ATM and RSVP. This value is considerably less that it takes to open a socket (about 10 ms) or to obtain the host (which in our testbed varied from 5 to 40 ms). The initialisation of Quartz is also considerably fast even for RSVP, which takes relatively long to initialise; since in ATM the reservation mechanism is integrated with the transport, no extra time is taken to initialise it. The overhead caused by the initialisation of Quartz occurs only once, while the request overhead occurs every time the application requests a new set of QoS requirements to be enforced. There is no overhead imposed on the data transmission, which depends only on the networking infrastructure and on the resources reserved for the communication channel.

4.7. Other Applications

In addition to being used in heterogeneous environments, Quartz can be used in different application areas. Besides the use of Quartz for data transfer applications (such as the remote copy example) other applications are being implemented on top of Quartz in the areas of distributed multimedia (the Quartz/CORBA Framework and the Distributed Music Rehearsal Studio) and real-time telecommunications (the telephone switch application).

The Quartz/CORBA Framework allows distributed multimedia applications to transfer audio and video over the network. Media control can be performed remotely by interacting with regular CORBA objects [15], while the flow of media uses the CORBA A/V streaming mechanism [16]. The enforcement of QoS requirements specified by the application is performed transparently by Quartz. The QoS parameters specified through the CORBA A/V streaming mechanism are interpreted by Quartz by using a new application filter, the A/V streams filter. After being translated by this filter, the QoS parameters are further processed by the translation unit and result in reservations at system level.

The Distributed Music Rehearsal Studio [17], which is an application that is being built on top of the Quartz/CORBA Framework, allows musicians to play together despite being geographically separated. The music rehearsal is done by plugging musical instruments to computers interconnected by a network. The sound produced by each partner is multicast to the others, mixed, and reproduced in order to provide a feedback to the musician. The QoS requirements incurring from the bandwidth necessary for transmitting the audio and from the performing of encoding and decoding operations are interpreted and enforced by Quartz.

The telephone switch application simulates the allocation and connection of phone trunks and the process of routing phone calls. A circuit switch filter interprets the requirements that are present at application level, which are basically the limited switching times of phone calls established by telecommunication regulators. This results in deadlines imposed on the switching process, which are set through a deadline scheduling filter and agent. At the network level, a phone circuit filter and agent allow phone trunks to be allocated for the call.

The application examples built on top of Quartz show the adequacy of the mechanisms for specification of QoS provided by Quartz and its suitability for enforcement of QoS in open systems. In each of the examples, the QoS parameters seen by the application, either when it specifies its QoS requirements or when it receives a QoS notification, are in the form of application-specific parameters which are suitable for the particular application area. The resulting parameters at system level allow the reservation of resources to be performed by using the reservation protocols available in the underlying system.

4.8. Comparison with Other QoS Architectures

One of the most important characteristics of the proposed QoS architecture is its capacity to handle QoS at both operating system and network levels. This contrasts with other systems described in the literature that are either network-oriented (and mostly ATM-oriented) such as Quanta [18] or that are purely system-level QoS architectures such as Arcade [19]. In addition, independence of operating system and network support provides a common way to handle QoS in open systems, which are naturally distributed and heterogeneous.

Translation and mapping have a special role in our architecture, which covers a wide range of QoS information specified at a high level of abstraction. In Quanta and Arcade, the mapping and translation of parameters is simplified because of their limited platform coverage. Quanta adopts a translation mechanism similar to the one proposed by us, but there are many differences that can be observed. Quanta translators, equivalent to our filters, involve a more complex translation procedure. In addition, they are not accessible to the user, and the addition of new classes of applications or network protocols implies that changes have to be made to the middleware itself. Arcade defines a QoS language for specification of constraints. Arcade could be emulated by Quartz by using an application-level QoS filter that interprets its QoS specification language and by providing a system filter and agent that interact with the Chorus kernel. In addition, we still provide support for establishing network QoS constraints.

In [20], Waddington affirms that single QoS managers such as the QoS broker adopted by the OMEGA architecture [21] 'require a huge amount of mapping and management knowledge to support large-scale distributed applications, and the service management through a single entity is too centralised and severely inflexible'. We don't incur this problem because QoS agents encapsulate only the support necessary for specification of QoS capabilities for the corresponding application field and interaction with the reservation protocols supported by the end-system. Flexibility and extensibility are guaranteed by the use of filters and component-specific QoS agents, instead of a monolithic structure such as that adopted by the QoS broker. Issues regarding resource reservation are handled by the corresponding protocol associated to the component that provides the resources, with the component-specific agent being responsible only for the interface with these protocols. Scalability is an intrinsic characteristic that results from the lightweight and distributed nature of the architecture.

Flexibility and extensibility are favoured by the design of the proposed architecture. These characteristics are especially important for supporting QoS in open, heterogeneous systems. QoS architectures such as QoS-A [3] and XRM/xbind [4] are tightly integrated with the network infrastructure, limiting their use in open systems. Finally, it is important to make clear that Quartz is targeted at a wide range of platforms, a matter that is not considered by the other architectures with QoS capabilities described in the literature so far.

5. Conclusions and Future Work

In this paper we have introduced a QoS architecture that deals with QoS constraints present in distributed applications. Quartz makes the lower-level aspects of resource reservation transparent for the application, although allowing the necessary control through notification in the case of resource adaptation. Quartz was designed to allow its use in heterogeneous platforms, enabling its integration into frameworks for the development of distributed computing applications with QoS requirements. The design of the architecture allows its easy extension to support new classes of applications, operating systems and communication infrastructures by adding components written by the application programmer.

We have developed a prototype of the Quartz architecture that has been used to provide mechanisms for QoS specification and enforcement to applications with QoS requirements. Applications built on top of this prototype show that Quartz handles heterogeneity at both system and application level efficiently, without incurring severe performance penalties. In the future, we intend to extend the platform coverage of the architecture by implementing new components that would provide support for a wide range of network protocols and operating systems.

Acknowledgements

This work has been supported by grants from CAPES (http://www.capes.gov.br) and from Iona Technologies (http://www.iona.com). The authors would like to thank David O'Flanagan and John Segrave-Daly for the development of the CORBA A/V Streams Mechanism and the Distributed Music Rehearsal Studio respectively.

References

[1] C. Aurrecoechea, A. Campbell and L. Hauw "A Survey of Quality of Service Architectures", MPG Group, University of Lancaster, Internal report MPG-95-18, 1995.

[2] R. Steinmetz and L.C. Wolf "Quality of Service: Where are We?", IWQoS'97 Proceedings, May 1997.

[3] A. Campbell, G. Coulson and D. Hutchison, "A Quality of Service Architecture", ACM Computer Communications Review, Vol. 24(2), April 1994.

[4] A. Lazar, K-S. Lim and F. Marcocini "Realizing a Foundation for Programmability of ATM Networks with the Binding Architecture", IEEE Journal of Selected Areas in Communication, Vol. 7, September 1996.

[5] A. Vogel et al. "Distributed Multimedia & QoS: A Survey", IEEE Multimedia, Vol. 2(3), Summer 1995.

[6] F. Siqueira and V. Cahill "Delivering QoS in Open Distributed Systems", FTDCS'99 Proceedings, December 1999.

[7] International Organization for Standardization "Information Technology: Quality of Service Framework", ISO/IEC JTC1/SC21 DIS 13236 ICS 35.020, 1995.

[8] International Telecommunication Union "B-ISDN Protocol Reference Model and Its Application", ITU-T Recommendation I.321, April 1991.

[9] R. Braden et al. "Resource Reservation Protocol (RSVP)". IETF RFC 2205, September 1997.

[10] D. Hildebrand "An Architectural Overview of QNX", QNX White Paper, 1999.

[11] Sun Microsystems "Sun Embedded Telecom Platform – Combining the Power of Solaris with the Real-time Performance of ChorusOS", White Paper, 1999.

[12] V. Yodaiken "The RT-Linux Approach to Hard Real-time", White Paper, October 1997.

[13] Microsoft Corporation "Real-time Systems and Microsoft Windows NT", White Paper, June 1995.

[14] J. Gecsel "Adaptation in Distributed Multimedia Systems", IEEE Multimedia, Vol. 4(2), April 1997.

[15] Object Management Group "The Common Object Request Broker: Architecture and Specification (Revision 2.0)", OMG Document PTC/96–03–04, March 1996.

[16] Iona Technologies, Lucent Technologies, Siemens-Nixdorf "Control and Management of Audio/Video Streams", OMG Document Telecom/97–05–07, July 1997.

[17] J. Segrave-Daly "The Design and Implementation of a Distributed Rehearsal Studio Application", Final Year Project, Comp. Science, Trinity College Dublin, June 1999.

[18] S. Dharanikota and K. Maly "Quanta: Quality of Service Architecture for Native TCP/IP over ATM Networks", HPDC'96 Proceedings, February 1996.

[19] I. Demeure, J. Farhat and F. Gasperoni "A Scheduling Framework for the Automatic Support of Temporal QoS Constraints", IWQoS'96 Proceedings, March 1996.

[20] D. Waddington, C. Edwards and D. Hutchison "Resource Management for Distributed Multimedia Applications", ECMAST'97 Proceedings, May 1997.

[21] K. Nahrstedt, and J. M. Smith "The QoS Broker", IEEE Multimedia, Vol. 2(1), April 1995.