# Quartz:
# A QoS Architecture for Open Systems

**Frank Siqueira**

Ph.D. Thesis

Department of Computer Science

Trinity College, University of Dublin

December 1999

# Declaration

I hereby declare that:

(a) This thesis has not been submitted as an exercise for a degree at this or any other University.

(b) This thesis is entirely the work of the author, except where otherwise stated.

(c) The Trinity College Library may lend and copy this thesis upon request.

Frank Siqueira

October 1999

# Acknowledgements

I would like to express my earnest gratitude to my supervisor, Dr. Vinny Cahill, for always providing useful advice with patience, care and interest. I will always be grateful to him for believing in my abilities and for providing me with freedom to explore and investigate the research topics that I found interested.  I would also like to thank him for arranging for my funding and for heading the research group skilfully.

I would like to thank all my colleagues in the Distributed Systems Group for their direct and indirect contributions to this Thesis, and for the good moments that we spent together in the last three years. Many thanks to the staff of the Department of Computer Science for the efficient support and friendship provided by them. I would also like to thank David O'Flanagan and John Segrave-Daly for the contribution given to this research project in the form of application tests.

Finally, I would like to thank my family and friends that, despite the distance, never let me feel alone and always provided me with extremely valuable emotional support.

# Summary

The term ‘QoS architecture’ is used to describe middleware that provides applications with mechanisms for specification and enforcement of quality of service (QoS) requirements. These architectures administer the resources provided by the underlying system with the intent of fulfilling the QoS requirements imposed by applications. Substantial work on QoS architectures can be found in the literature. However, the architectures proposed so far consider only part of the overall problem of delivering QoS in open systems.

The QoS architectures currently available are able to provide applications in a particular application area with the guarantees supported by the specific network architecture and the operating system to which they are bound. Most of these architectures, moreover, require QoS to be specified by using a low-level format that is not appropriate for applications that have a more high-level notion of QoS. Due also to their close integration with the underlying system, these architectures cannot be deployed over multiple computing platforms. Since they target a specific area of application, the area in which these architectures can be used is limited. In most cases the underlying system is not made completely transparent for the application, which still has to deal with low-level issues. In addition, most QoS architectures ignore dynamic resource adaptation, which can occur due to factors such as resource failure or system reconfiguration.

This thesis describes the design and implementation of Quartz, a QoS architecture for QoS specification and enforcement in open systems, designed with the aim of solving

the limitations of previous proposals in this area. Quartz is able to handle the differences between the multiple platforms that may be present in open systems due to its flexible and extensible structure based on replaceable components. Quartz can also adapt itself by rearranging its internal components, in order to provide mechanisms for QoS specification and enforcement suitable for different areas of application. These requirements are interpreted by Quartz and used to enforce QoS by employing the resource reservation mechanisms provided by the network and operating system. By adopting this strategy, Quartz makes the underlying resource reservation mechanisms transparent to applications requiring QoS enforcement. Furthermore, Quartz provides support for QoS adaptation by tracking resource adaptations that occur at lower levels and initiating QoS adaptation whenever needed.

This thesis presents a prototype implementation of the Quartz architecture and a series of applications built on top of it, including a complete framework for the development of multimedia applications based on the CORBA architecture. The experience gained by writing these applications shows the usefulness and efficiency of Quartz as a tool for supporting QoS specification and enforcement for diverse areas of application in open, heterogeneous environments.

# Table of Contents

# List of Figures

# List of Tables

# 1

# Introduction

This thesis is concerned with the study of mechanisms for the provision of quality of service (QoS) in open, heterogeneous and distributed environments. In order to overcome a series of limitations that were identified in similar work previously developed in this area of research, we have designed and implemented the Quartz QoS architecture. In essence, Quartz describes middleware that supports the specification and enforcement of QoS in open systems.

Quartz provides support for applications that require QoS-constrained services from the underlying system by interacting with resource reservation protocols present at the lower level. Applications supported by Quartz do not necessarily come from a single application domain. Moreover, the architecture does not depend on the use of a specific underlying operating system or network infrastructure. By supporting heterogeneity and distribution, the Quartz architecture is made suitable for open systems.

This chapter begins by introducing the problem of supporting QoS in open systems and describing the issues that should be addressed by middleware that is responsible for providing mechanisms for the specification and enforcement of QoS. In sequence, we briefly describe our proposed QoS architecture and outline its contribution to this field of research in comparison with other work in the same area. Finally, a roadmap of the remainder of this thesis is given.

## 1.1 Delivering QoS in Open Systems

With the popularisation of the Internet, the demand for distributed applications has increased considerably. These applications can be very complex to implement due to the necessity of dealing with network protocols, addresses, system heterogeneity and any other factor incurred from the topology of the computing platform. Distributed applications are often built on top of distributed computing middleware, which handles the complexity that originates from the distributed nature of the application on behalf of the programmer, makes applications easier to implement, increases their portability, and allows them to interoperate with other applications.

Some distributed applications can function properly on the currently available networking platforms and operating systems. However, there exists a category of computer applications that is not satisfied by the best-effort resource management policies provided by the majority of the computing platforms that are currently available. For these applications, the availability of resources provided by the underlying system is required to be predictable. These applications are said to have quality of service (QoS) requirements, and include applications varying from real-time control systems to distributed multimedia systems.

Applications that require a certain level of QoS must specify their requirements in a clear and accurate manner by using QoS parameters. The values of these parameters reflect the requirements imposed by the application, and can be stored in pre-defined user profiles containing the QoS constraints imposed on the behaviour of the application or can be obtained by the application through direct interaction with the user.

The achievement of the specified level of QoS is typically made possible through the reservation of the resources managed by the underlying system that are necessary to provide the network and operating system services used by the application with the requested level of quality. These resources include network bandwidth, processing time, physical memory, and access to multimedia hardware.

Several operating systems and network protocols incorporate mechanisms that allow applications to retain resources for their exclusive use. These mechanisms, called resource reservation protocols, are the key elements that support the provision of QoS guarantees. Unfortunately, most applications do not benefit from these mechanisms because the distributed computing middleware on which they rely is still being adapted to make use of such mechanisms. Furthermore, multiple resource reservation protocols coexist in open systems. Consequently, allowing applications to reserve resources via a middleware layer implies that the differences between reservation protocols have to be handled by the middleware.

Middleware components, usually referred as QoS architectures, are responsible for providing mechanisms for specification and enforcement of QoS that make use of the resource reservation protocols provided by the underlying system. QoS architectures deal with issues such as the translation of QoS parameters comprehensible at the application-level into the parameters understood by the underlying reservation protocols that control access to the resources provided by the system. Without the services provided by a QoS architecture, these issues would have to be dealt with by the application.

Research on QoS architectures has resulted in several proposals that can be found in the literature (see [4] for a survey and [139] for an analysis of the open issues in this area of research). During an extensive surveying of the area, four main limitations that prevent the use of the existing QoS architectures in open systems have been identified by us. These limitations are:

- most architectures require QoS to be specified by using a low-level format that is not appropriate for applications with a more high-level notion of QoS, or use a format appropriate for a specific area of application, limiting the use of the architecture;
- QoS enforcement often occurs at either network or operating system level, instead of both, and in some cases the underlying system is not made completely transparent for the application, which still has to deal with low-level issues;
- the use of most architectures in open, heterogeneous systems is prevented due to their close integration with the underlying system; and
- most QoS architectures ignore the possibility of dynamic resource adaptation, which can occur due to factors such as resource failure or system reconfiguration.

By solving the limitations described above we would have a QoS architecture that, besides handling the complexity that originates from the necessity of obtaining QoS-constrained services from the underlying platform on behalf of applications, would make applications easier to implement and would increase their portability between different underlying platforms. In addition, this QoS architecture would allow application programmers to use the same high-level interface for specifying QoS requirements for applications inserted in different application contexts and making use of different networking infrastructures and operating systems.

## 1.2 The Quartz QoS Architecture

In this thesis we propose a generic architecture for the specification and enforcement of QoS in open systems. This architecture provides mechanisms necessary for building applications with QoS requirements in open systems, since the QoS architectures currently available are typically dependent on a specific underlying system and are useful only in limited application contexts. Besides, they often expose the application to lower-level details and ignore the need of support for resource adaptation. These characteristics tend to limit the application portability, make them more difficult to implement, constrain the way applications express their QoS requirements, and prevent their use in open, heterogeneous systems.

The Quartz QoS architecture provides support for different application fields and is able to interact with the multiple resource reservation protocols that may be available in an open environment. This is achieved by adopting a flexible and extensible component-based design. The design of Quartz is based on a common core in which components are plugged in order to handle the particular characteristics of different applications and resource reservation protocols.

The Quartz QoS architecture allows users to specify their QoS requirements according to the notion of QoS present at the application level (e.g. requesting audio with CD quality instead of specifying the bandwidth needed to transmit the encoded audio over the network). This information is interpreted by Quartz, allowing it to perform resource reservations at a lower level of abstraction in order to fulfil the QoS requirements imposed by the application. No strong assumptions are made about the characteristics of the lower-level system, therefore allowing the deployment of the architecture on top of heterogeneous systems with different resource reservation capabilities.

The architecture, furthermore, supports resource reservation protocols that allow the availability of resources to changes dynamically due to factors such as resource failure, hardware reconfiguration or mobility. This is achieved through the use of resource adaptation techniques. Adaptation may occur at system level and be masked from the application, or may require the application to be notified in order to adjust its QoS requirements.

QoS enforcement and QoS adaptation at system level occur in a transparent manner from the application's point of view. Consequently, the application need not be aware of the resource reservation protocol used at the lower level. Despite providing transparency, Quartz allows the application to control the behaviour of resource reservation protocols whenever necessary, and notifications are sent to the application when the requested QoS requirements cannot be fulfilled.

## 1.3 Contribution

The Quartz QoS architecture provides a combination of features that are not present in other proposals of QoS architectures that are found in the literature.

One of the main features provided by Quartz is the possibility for applications to specify their QoS requirements according to their own notion of QoS. Being generic, Quartz does not require QoS to be specified in a standardised format, which would likely be unsuitable for a wide range of applications. Instead, Quartz allows the application to specify its QoS requirements by using its own notion of QoS, and employs a component that is able to interpret this information and convert it into the format that is used internally by Quartz. The internal format of QoS parameters used by Quartz is completely transparent to the application.

Quartz is also able to carry forward the translation of the QoS requirements specified by the application into a format that is suitable for performing resource reservation using the protocols provided by both the network and the operating system. During this process, QoS requirements are mapped into resources provided by the underlying system, which are then reserved in order to enforce the requirements of the application. Since network and operating system resources may be interchangeable, a balancing process has to be performed by the architecture at this point.

Quartz provides support for resource adaptation to occur at the lower level. Adaptation is handled by rearranging the balancing of resources internally or, when this is not feasible, by informing the application that its QoS requirements cannot be fulfilled and have to be adapted if they are to be satisfied with the resources that are currently available.

Due to its component-based design, Quartz is able to support multiple resource reservation protocols and application areas. The differences found in the surrounding environment are handled by plugging new components into the architectural core. These components can be selected from a library or provided by application developers, who are, therefore, able to extend the architecture according to their particular needs.

Furthermore, application simplicity is increased by the use of Quartz. By handling the QoS-related issues on behalf of the application, Quartz allows the programmer to concentrate on the functional aspects of the application. Due to its capacity to handle the differences between multiple resource reservation protocols present in heterogeneous platforms, Quartz also increases the portability of applications.

## 1.4 Analysis

We have implemented a prototype of the Quartz QoS architecture in order to evaluate its behaviour and validate its applicability in open systems and in different application scenarios. Some test applications that make use of Quartz have been written, including a complete framework for the development of multimedia applications that integrates Quartz into a CORBA-based environment.

The Quartz prototype and the applications built on top of it allowed us to conclude that the architecture fulfils the goals imposed on it during the design process. In order to evaluate the achievement of these goals, we have:

- Implemented applications inserted in different application contexts and that consequently have different ways of expressing their QoS requirements; Quartz has been shown to be able to interpret this information and identify the resources that have to be allocated in order to fulfil the requirements of the application.

- Observed through the application examples implemented by us that the QoS requirements of the application are enforced by Quartz through reserving resources provided by both the network and the operating system; in addition, regardless of the resource reservation protocols in use, the process of resource reservation was shown to become transparent to applications built on top of Quartz.

- Used multiple resource reservation protocols for reserving resources for the same application, without the necessity of rewriting the application code that specifies its QoS requirements, showing that Quartz is able to handle the differences between different underlying systems without any side-effects for the application.
- Simulated the occurrence of dynamic resource adaptation and observed that Quartz is able to perform QoS adaptation in order to reflect the new distribution of resources; this process occurs either transparently at system level by rearranging the balance of interchangeable resources, or by notifying the application to perform adaptation of its QoS requirements when adaptation at system level is not feasible.

In addition, with the experience obtained by writing applications that make use of Quartz, we have noticed an increase in their portability and the simplicity of their code, since the differences between underlying platforms are handled by the architecture.

Performance measurements have shown that the impact caused by Quartz on the overall performance of applications that use it to specify and enforce their QoS requirements is very small. We believe that this small overhead will be accepted by application programmers that can benefit from the services provided by the Quartz architecture.

We have also analysed the characteristics of Quartz in the light of other work in the area of QoS architectures and frameworks. To the best of our knowledge, there is no QoS architecture that combines a similar set of features. Most architectures found in the literature are able to provide a similar set of features but are highly dependent on a particular platform. In general, their mechanisms for specification of QoS are not flexible enough to support a wide variety of applications, forcing them to deal with a notion of QoS that is not suitable for their needs. Quartz, on the other hand, is able to provide efficient tools for specification and enforcement of QoS in open systems, which are intrinsically distributed and heterogeneous, and still fulfil the needs of diverse areas of application due to its extensibility and flexibility.

## 1.5 Roadmap

The remainder of this thesis is organised as follows. Chapter 2 provides some necessary background to the thesis and presents the technology currently available in the area. Chapter 3 describes the state-of-the-art work developed in the areas of middleware for distributed multimedia and QoS architectures, analysing their characteristics and limitations. The Quartz architecture is introduced and described in detail in Chapter 4. Chapter 5 describes a prototype implementation of Quartz, and Chapter 6 shows how this prototype can be used together with CORBA in order to provide a framework for the development of multimedia applications. Chapter 7 presents a complete validation and evaluation study of the Quartz architecture, introducing examples of applications that demonstrate the suitability of the architecture as a tool for obtaining QoS-aware services in open systems. Finally, Chapter 8 presents some conclusions, summarises the contributions of this thesis and unveils some opportunities for future work in this area.

# 2

# Overview of the Research Area

This chapter provides some background on the area of research with which this thesis is concerned. The following sections introduce the key concepts and technologies present in the areas of multimedia and real-time applications, open systems, distributed object computing, quality of service and resource reservation protocols.

The technologies used by us for implementing a QoS architecture and a multimedia framework, which will be presented in chapters 5 and 6 respectively, are also described in detail. These technologies include the Common Object Request Broker Architecture (CORBA), Asynchronous Transfer Mode (ATM), the new generation of Internet Protocols, with special attention to the Resource Reservation Protocol (RSVP) and the real-time capabilities provided by Windows NT.

The technologies described in this chapter are seen as important tools for building an environment for QoS specification and enforcement in open systems such as the one targeted by this thesis.

## 2.1 Multimedia Applications

Interactions between computers and other processor-based equipment are based on the exchange of data represented by electrical signals. This data follows a well-defined encoding scheme that represents some kind of information (e.g. a text file might be encoded using the ASCII character set, or a compressed file might be encoded in zip format). However, computer systems are evolving towards becoming not only responsible for handling computer data, but also media such as audio, video and animated graphics.

Media data is represented using digital encoding schemes as is other data, although the characteristics of multimedia data distinguish it clearly from regular computer data. Multimedia typically requires a higher amount of data to be processed and transmitted, presents a higher complexity to encode and decode, and often requires that time and synchronisation rules are obeyed during reproduction [53].

Multimedia systems are able to handle multiple kinds of media data in order to support multimedia applications, i.e. programs that, often employing special hardware, are able to encode, transfer, decode and reproduce multimedia data. During the last few years a rapid growth in the demand for multimedia applications has been observed. Nevertheless, the performance of these applications is still poor due to limitations in processing power and, in the distributed case, due to limited network bandwidth.

Hardware support for multimedia is increasing fast. Almost every desktop computer sold nowadays comes with some sort of support for multimedia. Devices such as sound cards, 3D graphics accelerators and digital cameras are becoming more common every day. The latest generations of processors include special instruction subsets designed to improve the performance of media processing (such as Intel's MMX) and for rendering 3D graphics (such as AMD's 3D Now!) [120].

Moreover, a fact that has to be taken into account when studying multimedia applications is the increasing popularity of portable devices, inexpensive gadgets that are small enough to fit in a pocket. Devices such as laptop computers, cellular phones, pagers, and video games are now converging into a single unit, called a personal digital assistant (PDA) or palmtop computer. Users are likely to expect to have the same functionality on these devices as they have on their desktop computers. This leads to greater demand for computing power, but at the same time, the size, weight, and power consumption of these devices must remain limited. For example, these devices are typically the size of a cigarette pack, weigh from 150 to 300 grams (4 to 8 ounces), have only 2 to 8 Mb of memory, and are expected to run on the same set of batteries for about a week [85]. Complete operating systems (such as Palm Computing's PalmOS [117] and Microsoft's Windows CE [17]) provide a fully functional platform for the execution of common software applications such as text editors, e-mail readers, web browsers and personal information managers (PIM). A complete networking infrastructure based on modems and infrared communications is also provided for such devices, but complete support for multimedia is still being introduced.

Another important factor that influences multimedia applications is network performance. The networking infrastructure currently available for the typical end-user (i.e. local area networks such as Ethernet and personal computers with modems interconnected by the Internet) satisfies the common needs of most network applications.

Applications such as Web browsing and chatting require low bandwidth and can even tolerate some degree of network congestion. On the other hand, multimedia applications demand higher bandwidth and behaviour in the distribution of network resources that is not provided by the networking infrastructure in place at the present. Solutions for this problem have started to emerge in the form of reservation protocols (discussed in Section 2.6) and middleware with support for QoS specification and enforcement (such as the proposals introduced in Chapter 3), but the application of these solutions is yet to reach most users [141].

### 2.1.1 Distributed Multimedia

Distribution is yet another fact that has to be considered in the study of multimedia applications. Due to the interactive nature of some multimedia applications, information has to be exchanged through the network, characterising them as distributed multimedia applications.

Distributed multimedia applications have well-defined requirements based on the characteristics of the kind of media that is to be transmitted. Besides, the technique used for transferring media has to be taken into account. Some kinds of pre-produced media, such as short movies, songs or news reports, have less strict timing requirements than other categories of media. This is because pre-produced media can be buffered after transmission and reproduced only when available locally. On the other hand, live media and long video streams that are reproduced while they are still being transferred through the network require the media transfer and rendering process to be more predictable than that obtained in best-effort systems. This last class of media, which is called continuous media, is discussed in detail in the next section.

### 2.1.2 Continuous Media

In the face of increasing interest in the transmission of digital media over computer networks, the usual communication mechanisms adopted for data transmission have been shown to be unsuitable in some particular cases. This unsuitability derives from the distinct characteristics of continuous media.

Continuous media can be defined as media where there is a timing relationship that must be observed during the rendering of the media in order to reproduce it correctly. This timing relationship must be followed otherwise the integrity of the media is destroyed, i.e. the information that was intended to be presented can be lost or misunderstood [132]. In the particular case where media is transmitted over the network and reproduced at a different location, this implies that the destination must reproduce the timing relationship that existed at the source. This contrasts with the usual approach adopted for data transmission in which information is usually retrieved and delivered upon request. This kind of media does not have an implicit temporal dimension, and is also termed discrete media. The required bandwidth for discrete media depends on the nature of the task to be accomplished, ranging from a couple of bits per second in case of simple sensors, to very large volumes of data if we consider a world-wide banking system or a massively distributed file system. The acceptable delay is also related to the application, being for example very small in manufacturing and control systems and more relaxed in an airline reservation system or an FTP session. Traffic is usually bursty and the timing relation present at the origin typically does not need to be reconstructed at the destination. The reliability of data is generally a critical factor in all of these systems.

The requirements of different kinds of continuous media can be classified according to the following parameters:

- Bandwidth required;
- Acceptable end-to-end transmission delay and delay variation (jitter);
- Transmission reliability (data loss);
- Distribution of traffic during time (burstiness or bit-rate distribution);
- Importance of keeping the original timing relation during the reproduction of media.

In the following paragraphs we analyse different classes of continuous media according to these parameters. In sequence, Table 1 provides a comparison of the bandwidth required by typical encoding schemes of different classes of continuous media.

**Audio.** Audio is usually encoded as a constant bit-rate (CBR) signal in the majority of communication systems currently available. However, digital techniques have been adopted to allow compression of audio with very low loss of quality, resulting in variable bit-rate (VBR) traffic. Compression techniques based on data redundancy and forward prediction are difficult to apply for audio, and as a consequence compression algorithms for audio usually have a lower efficiency if compared to compression algorithms for video. The reproduction of audio with reasonable quality is basically related to the sampling rate and the number of bits used to represent the signal in a digital stream. The frequency range is limited by the capacity of the human ear, limiting the number of bits needed to encode the signal and the bandwidth necessary to transfer it through the network. Furthermore, the original timing relationship observed during the generation of the sound must be reproduced at the destination. Some data loss can be tolerated by audio applications, and delay and jitter can be compensated by using buffering techniques.

**Voice.** The human voice, despite being a particular category of audio, must be studied separately due to its importance in communication systems. Voice requires low bandwidth when compared with full audio signals because of its limited frequency range and the low sampling rate necessary for its comprehension. In addition, voice media is sensitive to delay and delay variations, but a delayed sample may be discarded without serious loss of comprehension by the listener. Apart from the lower bandwidth necessary for voice media, its traffic characteristics are very similar to those of audio signals. Voice can also be encoded as CBR or VBR signals, and compression is usually more efficient than for other categories of audio because of the presence of silent periods.

**Video.** A video signal can be encoded digitally as CBR or VBR data, depending on the method used to encode it. The bandwidth required is very high when compared with other sorts of media, and increases considerably when slightly better resolution is desired. If we consider the capacity of the human eye, a rate of 30 frames per second is sufficient to allow a comfortable sensation of movement. The loss of one frame in a long sequence is usually not perceptible, and delayed frames need not be exhibited. A low delay variation is tolerable, and the reconstruction of the timing relation present at the source time during rendering at the destination must be carried out. Like for audio, delay and jitter can be compensated by using buffering techniques.

| Media | Description | Bandwidth |
|---|---|---|
| Voice | Low quality (4 bits/sample; 8 kHz sample frequency) | 32 Kbps |
| | Phone quality (8 bits/sample; 8 kHz sample frequency) | 64 Kbps |
| Audio | CD quality (16 bits/sample; 44.1 kHz sample frequency) | 1.4 Mbps |
| | MPEG audio with approx. CD quality | 200 to 300 Kbps |
| Video | VHS quality (352x240pix;24bits/pix; 30 frames/s) | 60 Mbps |
| | MPEG video with approx. VHS quality | 1.5 to 2 Mbps |
| | Pal/Secam quality (544x480pix; 24bits/pix; 30fr/s) | 188 Mbps |
| | MPEG video with approx. Pal/Secam quality | 4 to 6 Mbps |
| | HDTV quality (~1200 lines; 24bits/pix; 30fr/s) | 1 Gbps |
| | MPEG video with approx. HTDV quality | 20 to 60 Mbps |

**Table 1 – Typical Bandwidth of Different Classes of Continuous Media**

### 2.1.3 Multimedia Technology

There is currently a major research effort being carried out on the development of distributed multimedia systems and platforms. Unfortunately, most proposals do not adopt an approach that allows them to handle media in open systems.

Standards for distributed multimedia such as the Multimedia System Services (MSS) [67] that was proposed by the International Multimedia Association (IMA) have started to emerge. MMS has as its main purpose to provide support for interaction between multimedia applications in open environments. IMA members include the main software and hardware suppliers and consumers around the world. MSS, as a reference model for design and programming of distributed multimedia applications, is targeted at complex, scaleable, extendible and interoperable applications running in an open environment. Being just a reference model, MSS does not specify how the multimedia services are obtained from the system.

Another proposal in this area is the MHEG standard [95], defined by the Multimedia and Hypermedia Experts Group (MHEG) of the International Organisation for Standardisation (ISO). Like MSS, MHEG still does not provide a complete solution for supporting multimedia applications with QoS requirements in open systems since it is basically a reference model.

Some software development kits such as Microsoft's DirectX [98] and the Apple QuickTime architecture [2] consist in useful solutions for the development of multimedia applications. However, these technologies lack mechanisms for QoS specification and enforcement, and have limited platform coverage.

A series of state-of-the art proposals of support for multimedia programming will be presented in chapter 3, but these frameworks and architectures still lack complete support for QoS specification and enforcement in open, heterogeneous systems.

This thesis aims to provide mechanisms in order to overcome the limitations found in supports for the development of multimedia applications. Further in this thesis we will introduce a QoS architecture that is used in a media streaming framework. This framework provides support for the development of multimedia applications with QoS requirements in open systems.

## 2.2 Real-Time Applications

Real-time systems include those responsible for the control of manufacturing plants, aircraft, nuclear plants and so on. Most of these control applications must avoid critical failures that can lead to catastrophic errors, in which the occurrence of an error can be much more damaging than the benefit provided by the system. For example, a plane crash is much more damaging for an airline than the completion of a flight safely.

Furthermore, real-time data is not considered valid only by being semantically correct. A delay in the provision of a data item can be much more damaging than the provision of an inaccurate value. So, real-time data must not only be semantically correct, but also timely correct (i.e., the data must be available when it is required by the system).

Like multimedia applications, real-time applications are not satisfied by best-effort networking and processing policies. Real-Time applications are time critical by definition, and cannot tolerate unpredictable variations in the behaviour of the system without the possibility of causing catastrophic errors.

So far, real-time systems have been isolated from external influence by using special software and hardware designed for this purpose. However, aiming to reduce costs and simplify maintenance, real-time systems are evolving towards a more open approach, in which real-time data would flow side by side with non-real-time traffic. In this case, a complex approach for avoiding interference between the two classes of traffic must be adopted.

Real-time applications typically require less bandwidth than multimedia applications. The transfer of the encoded data read by a temperature sensor may account for just a few bytes, for example. However, this data may be required to activate alarms that will prevent a fire, and is therefore of more importance for network transmission and processing than other information that is being transferred through the network and processed by the same computational system.

Research on real-time systems has focused mainly on two areas of research:

- Real-time operating systems, languages and middleware [23][122], which provide mechanisms for implementing real-time applications; and
- Real-time scheduling algorithms [3][83], which select the order of execution of the activities performed by the system based on priorities and deadlines associated to them.

Despite the solid knowledge obtained with intense research in the area of real-time systems, the techniques employed for supporting real-time computing do not integrate gracefully with the technology currently available for best-effort applications. The integration of these technologies has emerged as an important research topic, and constitutes a challenge for scientists in this area of research.

### 2.2.1 Real-Time Technology

Current real-time platforms are mainly focused on embedded hardware and specialised software. Several successful real-time environments for embedded systems can be found in the literature and commercial, off-the-shelf solutions can be found in the market.

One of the most influential experimental real-time environments originated from the MARS project [84], developed at the University of Vienna. MARS provides tools for the analysis and deployment of hard real-time systems. Its architecture employs redundancy with self-checking procedures for enabling fault tolerance, including a redundant network bus. At the software level, the MARS real-time operating system (O.S.) provides tools for the analysis of worst-case execution times and static (prior to execution) real-time scheduling. Communication is based on the time-triggered protocol (TTP), which provides a predictable, fault-tolerant communication system.

In the field of commercial products for real-time systems two products share the status of market leaders: QNX Software Systems' QNX real-time O.S. [62] and WindRiver's VxWorks [151]. Both are microkernel-based, extensible, and POSIX-compatible operating systems, and can be scaled up or down according to the necessities of the target system. Another product with a strong presence in the market is the Chorus O.S. [142], developed by Chorus Microsystems, now a division of Sun Microsystems. Chorus OS provides real-time guarantees such as QNX and VxWorks, and offers a very rich set of tools for developing and testing real-time applications.

Despite the common use of real-time operating systems in embedded environments, the possibility of their integration with open systems is somewhat limited. Research in this field has been divided into two directions: the integration of real-time capabilities into commercial operating systems and the provision of communication paths between real-time and non-real-time operating systems.

Operating systems such as Sun's Solaris 2.X and Microsoft's Windows NT provide a set of mechanisms that allow real-time activities to be performed. The approach adopted in both cases encompasses a real-time scheduling class which has priority over the regular activities performed by the system. In addition, these operating systems provide mechanisms that support higher predictability in the execution of code by reducing interference caused by memory paging, I/O operations and interrupts.

On the other hand, software vendors are starting to provide communication paths between embedded real-time software and applications running on non-real-time operating systems. Iona Technologies provides implementations of its CORBA-compatible Orbix ORB for VxWorks and QNX. Chorus has its own implementation of CORBA, called COOL (Chorus Object-Oriented Layer), which provides a communication path between non-real-time ORBs and applications running on top of the Chorus O.S..

The technology available in the area of real-time systems is still very low-level for the application programmer, and its close integration with the computing platform limits its use in open, heterogeneous systems. One of our goals in this thesis is to provide middleware with mechanisms for the specification of real-time requirements. These mechanisms would hide the complexity present at the low-level from the programmer of real-time applications, and would allow these applications to interact with different low-level supports through the middleware. This middleware, which will be described further in this thesis, allows not only real-time applications but also other applications with timing constraints (such as multimedia applications) to specify their constraints in the form of QoS parameters, which are then enforced by the QoS mechanisms provided by the middleware.

## 2.3 Open Systems

Computational systems are said to be "open" when they allow external components to take part of the computation by using a well-known interface. By specifying their interfaces, software components are allowed to interoperate with their peers, independently of the vendor that provides the component [38].

The concept of open systems implies that distribution and heterogeneity are key characteristics in these systems. Distribution originates from the fact that a program is not constrained to a single machine or address space, but can communicate through the network with other software elements. Heterogeneity is present because programs can be written in different languages, and run on different operating systems built on top of different computer architectures, but can still interoperate via standardised interfaces to other software.

The heterogeneity of hardware and software and the complexity of developing distributed applications to run on top of systems with the characteristics mentioned previously require the use of open computing architectures.

A key requirement in open architectures is to provide the programmer with mechanisms to support distribution and heterogeneity transparency so that the handling of these issues is simplified. These mechanisms include standard services that are needed by applications in an open environment, such as naming and location services, and a standardised communication mechanism that allows software to communicate with each other. In addition, mechanisms for describing, storing and retrieving the interface of a component must be provided in order to allow software to know how to contact their peers.

### 2.3.1 Open Systems Technology

Middleware providing support for open systems, which make the intrinsic distribution and heterogeneity present in open systems transparent to the application, is widely available and used by programmers nowadays.

Reference models such as the ISO Open Systems Interconnection [68] and Open Distributed Processing [69] models represent an important contribution to this area of research at a theoretical level. However, the most important step for the proliferation of support for open systems was the wide adoption of products based on standards such as OSF DCE [114], Microsoft's DCOM [97], and OMG's CORBA [108]. These products allow distributed programs to interact in open environments in order to carry out cooperative tasks. This is done by providing a standard way of specifying the interfaces supported by programs, describing the routines (or methods in object-oriented systems such as CORBA and DCOM) that can be called remotely and their respective parameters. In addition, a communication protocol that allows client programs to remotely invoke the routines supported by other programs acting as servers is defined by each of these standards.

Despite the wide availability of standards for open distributed environments, and despite the intense research efforts focused in this area, the integration of multimedia and real-time support into middleware for open distributed systems is a problem that still lacks a satisfactory solution [31][141]. We intend to overcome this deficiency by proposing an architecture that provides support for the specification and enforcement of the requirements present in multimedia and real-time applications built in open, heterogeneous and distributed systems. This architecture will be described in detail further in this thesis.

## 2.4 Distributed Object Computing

In the last decade the development of software, which was typically based on the use of structured design and procedural languages, has rapidly migrated towards the adoption of object-oriented design and programming techniques. Procedural programming languages such as C and Fortran are being replaced by object-oriented languages such as C++ and Java. Meanwhile, modelling tools that employ object-oriented design, especially the tools that are based on the Unified Modelling Language (UML) [22], have been widely employed by programmers for the analysis and design of object-oriented software.

The adoption of object-based approaches to distributed computing that occurred subsequently was a natural evolution. The use of object orientation in distributed systems was further reinforced by the introduction of standards for interaction between distributed objects.

In a distributed object environment, objects are used to represent parts of the computation that may be running on different machines dispersed over the computer network. These objects interact with each other in order to execute cooperative computation. Interaction between objects occurs in the way of remote object invocations, i.e. by calling methods on remotely located objects. To allow remote invocation, a remote object has to know the interface of the target object and call one of its methods, passing the parameters of the call and possibly receiving a return value. Moreover, remote objects can be written in different languages and run on different platforms. In order to allow interaction between different remote objects a common invocation model must be agreed upon.

The interfaces of an object can be specified at different phases of its lifetime:

- Statically: i.e. at compile time, by describing the interfaces supported by the object using an Interface Definition Language (IDL); or
- Dynamically: i.e. at run-time, by availing of metadata obtained directly from the object or through an interface repository.

Invocation models have to specify communication protocols and message formats to be used for calling methods remotely. An invocation model can support one or more calling semantics. Three well-known types of calls classified according to calling semantics are:

- Synchronous calls: block waiting for the target object to conclude the processing of the call and send the result of the operation;
- Asynchronous calls: non-blocking, with the execution of the caller continuing right after the call is dispatched through the network; and
- Deferred synchronous calls: asynchronous call with the possibility of a posterior synchronisation, in which the result of the operation can be retrieved.

Distributed object computing models are supported by middleware, which hides the complexity resulting from network communication, handling of network addresses, conversion between data formats and so on. Distributed object middleware provides the programmer with powerful tools for the development of distributed applications.

### 2.4.1 CORBA

With the widespread adoption of object-oriented design and programming techniques for application development, it was natural that this approach reached the area of distributed computing. The adoption of object-based approaches in distributed systems was reinforced by the availability of standards proposed by the Object Management Group (OMG), a consortium created by software vendors, developers and end-users to promote object technology for the development of distributed computing systems.

The OMG has proposed a set of standards for distributed object computing, which compose the Object Management Architecture [105]. The OMA is a framework of standards and concepts for open systems, centred in the *Object Request Broker* (ORB). In this architecture, methods of remote objects can be invoked transparently in a distributed and heterogeneous environment using the services provided by an ORB.

The *Common Object Request Broker Architecture* is the standard ORB defined by the OMG [108]. The CORBA specification establishes the role of each component of the ORB and defines their interfaces. By introducing a common architecture, the OMG makes transparent for applications the differences between distinct CORBA implementations and lower-level systems. The components of the CORBA architecture are shown in Figure 1.

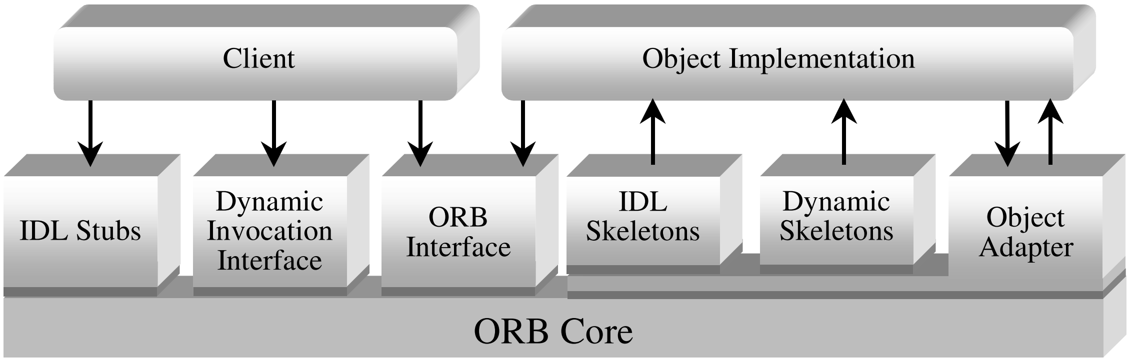


**Figure 1 – The CORBA Architecture**

In the CORBA environment, each object implementation has its interface specified in IDL (Interface Definition Language). Remote method invocations are issued by clients calling methods in local stub objects that are generated by the IDL compiler and have the same interface as the corresponding remote objects. Alternatively, the client can build the request using the Dynamic Invocation Interface (DII). The ORB Core transmits the request and transfers the control to the object implementation (i.e., the server) using an IDL skeleton and the corresponding object adapter. The object implementation is unaware of the invocation method used by the client to issue the request.

Dynamic skeletons allow servers to receive calls on methods that are not specified in their IDL interfaces, and therefore are not accepted by the IDL skeleton supported by the object. The dynamic skeleton interface (DSI) is usually adopted to build gateways.

The ORB Interface offers some general-purpose services to CORBA objects through a well-defined interface. Object implementations can also use services provided by their corresponding object adapters. OMG specifies the interfaces of a general-purpose Basic Object Adapter (BOA) which covers the requirements of a wide range of object systems. However, this does not prevent object adapters with more specialised capabilities being added to the support in order to fulfil the needs of applications with requirements that are not covered by the BOA.

When the client and the object implementation are located in ORBs that use different protocols for remote communication, a communication protocol called IIOP (Internet Inter-ORB Protocol) is used to allow interoperability. In addition, the OMG has approved mappings of the IDL interfaces for the main programming languages in order to allow objects written in different languages to interoperate.

CORBA has become a *de facto* standard for distributed object computing. Several legacy applications in sensitive areas such as banking and telecommunications have been ported to CORBA platforms, and new applications have been built based on CORBA in order to obtain easy interoperability with other applications.

### 2.4.2 CORBAservices

The CORBA Services Specification [107] intends to provide the OMA architecture and CORBA with a set of standard mechanisms that are often necessary in a distributed environment.

In particular, the following services of major importance to distributed systems were proposed by the OMG:

- a Naming Service;
- a Life Cycle Service;
- an Event Service;
- a Transaction Service; and
- a Security Service.

**Naming Service.** A naming service allows components to discover other components in the distributed environment with which they can interact and execute cooperative tasks. It is basically a location service that associates identifiers to handles (i.e. references) that provide a way to contact the desired component in the distributed system.

The CORBA Naming Service binds names, inserted into hierarchically organised name contexts, to CORBA objects.

**Life Cycle Service.** A life cycle service is basically a configuration service. It defines services and conventions for creating, deleting, copying and moving components in a distributed system.

The CORBA Life Cycle Service defines the interface to a ‘Factory’ object that allows the creation of objects using factories, as well as a ‘LifeCycle’ interface with remove, copy, and move operations.

**Event Service.** An event service implements a decoupled communication model, based on the publish/subscribe paradigm. In this communication model, one or more publishers (or suppliers) can send messages related to a specific topic, while a group of subscribers (or consumers) receive the messages asynchronously. Through this mechanism, suppliers can generate events without knowing the identities of consumers, and consumers can receive events without knowing who supplied them.

The CORBA event service is used by applications whose interaction model can be more easily modelled as events than as method invocations. Despite providing a useful alternative to the traditional invocation mechanism implemented by CORBA, the event service cannot be used by applications with constraints on the latency of events. In order to overcome this limitation, the OMG has introduced the notification service [11], an extension of the event service that allows QoS constraints to be associated to events.

**Transaction Service.** A transaction service organises interactions between distributed objects establishing delimiting transactions and commitment points. The CORBA transaction service not only supports multiple transaction models, but it also allows interoperability between different transaction models.

**Security Service.** A security service controls identification of human or software entities in order to verify that they have authorisation to execute a requested action. The CORBA security service allows identification and authentication of distributed objects and implements access control, in order to verify if a particular entity has authorisation for calling a specific operation (method) of an object. Furthermore, it allows secure communication over insecure network links, with protection of the integrity and confidentiality of messages transferred between objects.

**Additional Services.** In fact, a complete set of services useful in some specific applications was defined by OMG. Other important services are:

- Time Service: supplies time information and allows the definition of periodic (i.e. recurrent time-triggered) calls.
- Licensing Service: controls the utilisation of specific objects in the distributed system, inhibiting improper use of intellectual property.
- Properties Service: provides the ability to dynamically associate properties (any IDL value) with objects.
- Relationship Service: allows the definition of relationships between CORBA objects.
- Query Service: allows users and objects to invoke query operations on collections of objects.
- Persistency Service: provides mechanisms used for retaining and maintaining the state of persistent objects.
- Concurrency Control: allows the co-ordination of multiple access to shared resources through the provision of several lock modes, and permits flexible resolution of access conflicts.
- Externalisation Service: defines conventions and protocols for representing state information related to an object in the form of a stream of data, allowing this information to be stored or transferred to other locations.

### 2.4.3 CORBA A/V Streams

Despite its adequacy for transmission of best-effort data, CORBA, like its counterparts, originally did not have support for streaming continuous media. The immediate solution for the implementation of multimedia applications would be the use of CORBA for transmission of control data, while the stream data would have to be transferred using a lower-level network protocol that provides mechanisms for QoS specification or resource reservation. However, it is not reasonable to use high-level middleware to provide protocol and distribution transparency for transmitting best-effort data, and on the other hand expose the programmer to the lower-level protocols used for transmission of stream data.

It is highly desirable that distributed object middleware provide the same kind of high-level abstraction, which is available for best-effort data, to deal with continuous media. To make CORBA suitable for continuous media, OMG has defined streaming mechanisms to be introduced at application level in the CORBA architecture [109].

The CORBA streaming mechanism defines a set of abstractions to deal with stream data in multimedia systems. These abstractions are illustrated by Figure 2.

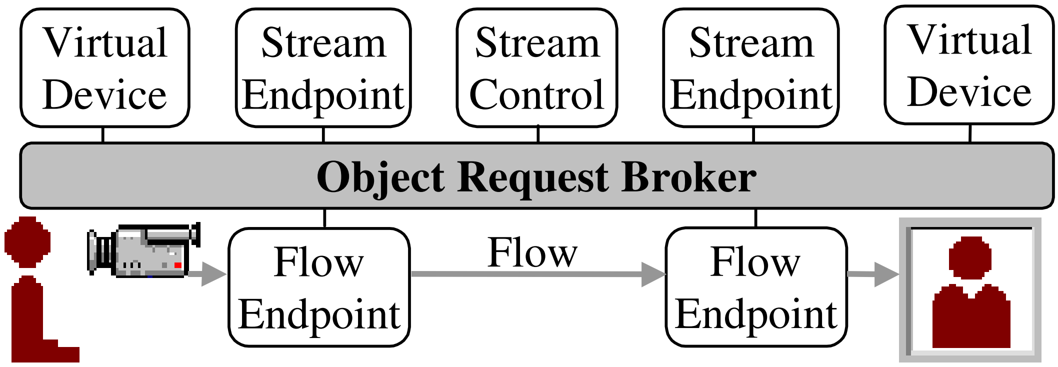


**Figure 2 – Overview of the CORBA A/V Streams Mechanism**

Virtual device objects abstract multimedia devices (i.e. cameras, speakers, etc) used by multimedia applications. A stream control object allows the user to control the media flow (i.e., start and stop it) as well as add and remove parties from a multi-party connection.

Stream endpoints transfer stream data through the network, getting data from and delivering data to virtual devices. Each endpoint has one or more media flows, which are abstracted as flow endpoint objects located at each end of the flow. Flow endpoints are classified as producers and consumers according to the role performed by them in the transfer of data.

A binding operation establishes a stream between two virtual devices and defines the QoS parameters associated to the stream. Two binding semantics are allowed. The simple local binding uses the virtual device and returns a stream control object. Besides, in the third-party binding mechanism, a stream control object can bind two remote virtual devices and keep control over the data flow.

A set of QoS parameters for audio and video streams is defined by the specification (for example, the number of bits per sample and the number of channels for audio, or the resolution and the colour depth for video). The QoS parameters associated with a stream can be modified through the stream control object, through the stream endpoint or through individual flow endpoints.

Figure 2 shows an application scenario in which a video streaming application is built using the CORBA streaming mechanism. The image generated by a video camera is fed by a virtual device to a flow producer, which establishes a connection with the flow consumer and then feeds the virtual device associated to the video screen with the image produced by the camera. The characteristics of the media stream, i.e. the number of flows of audio and video and the QoS parameters associated to them, are specified when the devices are bound. The stream endpoint handles the network connection between the two or more partners, and the control of the media flow and of the membership of the connection is done through the stream control object.

With the introduction of the stream mechanism, the CORBA architecture becomes suitable for the provision of the communication services necessary for multimedia applications, and can be adopted as the common middleware for control *and* transfer of media. However, the mechanisms used for QoS enforcement are not defined by the CORBA A/V streams specification. In order to solve this deficiency, this thesis proposes middleware that provides QoS mechanisms that, when integrated with CORBA, can provide a full framework for the development of distributed multimedia applications.

## 2.5 Quality of Service

Despite the evolution of computer hardware, resources such as network bandwidth, processing time and memory are becoming scarce because of the complexity of a special category of applications. We are experiencing the proliferation of applications with very different sets of requirements, ranging from real-time embedded control systems to multimedia applications running on desktop computers connected to the Internet. With the migration of real-time systems from specialised architectures to open environments, guaranteed services and limited response times with very low (or even zero) error rates are required to allow consistent real-time behaviour. On the other hand, multimedia applications such as multi-party conferencing, audio and video broadcast, and distributed cooperative applications are becoming common for the end-user. This class of application can tolerate some error, but requires limited response times and specific throughput. At the moment, the quality of the output generated by these applications is inadequate due to the existing limitations of bandwidth and processing power experienced by the networks and computing systems currently available.

The main problem faced by this category of applications is to guarantee that the services executed by the underlying system on behalf of the application will be performed respecting all the necessary requirements. A myriad of resources may have to be provided by the system to allow these requirements to be fulfilled, ranging from local resources such as memory and CPU to network bandwidth and other remotely located resources.

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 performance of the services provided to an application.

### 2.5.1 Definitions

*QoS* is defined by the ISO OSI/ODP group as 'a set of qualities related to the collective behaviour of one or more objects' [70]. Other authors try to clarify this definition. For example, Vogel et al. [146] 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'.

In this thesis we adopt a very similar definition, with the main difference residing in the fact that we do not constrain the application of QoS to distributed multimedia systems, but also extend it to incorporate any system with constraints related to response time, performance, and output quality.

ISO, along with the concept of QoS, defines a complete terminology for dealing with QoS. However, they consider the application of QoS only for the specification of communication services at network level. We adopt their terminology slightly modified to encompass diverse application fields.

According to ISO, a *QoS characteristic* is 'a quantifiable aspect of QoS' such as bandwidth or memory usage. A *QoS measure* is defined as 'one or more observed values relating to a QoS characteristic' such as the measured value of bandwidth or memory being used by an application at an instant in time.

A *QoS information* is any 'information related to QoS'. QoS information can be classified as a *QoS requirement*, when expressing 'a requirement to manage one or more QoS characteristics', or as *QoS data*, if the information is a measured value, a warning, or any QoS information used by the system in an enquiry. QoS information is further classified as *QoS parameters*, when exchanged between entities, or *QoS context*, when retained in an entity.

A *QoS operating target* is 'QoS information that represents the target values of a set of QoS characteristics, derived from QoS requirements', such as the necessary value of bandwidth to be provided to an application to allow it reach its requirements. 'A group of user requirements that leads to the selection of a set of QoS requirements' is called a *QoS category*. One example of *QoS category* is a user requirement such as 'VHS quality' for a video session, leading to the selection of QoS requirements for frame size, number of frames per second, colour scheme, and so on.

*QoS mechanisms* are responsible for the establishment, maintenance, enquiry and management of QoS. *QoS establishment* creates the conditions to attain the desired set of QoS characteristics for an activity before it occurs; *QoS maintenance* keeps the set of QoS characteristics at the desired values while the activity is in progress; and *QoS enquiry* consists in 'the use of QoS mechanisms to determine properties of the environment related to QoS'.

*QoS management* is a set of activities performed by the system to support QoS monitoring, control and administration. *QoS monitoring* is the use of QoS measures to estimate the values of QoS characteristics, while *QoS control and administration* (usually referred as 'QoS control' for short) are responsible for providing conditions so that a desired set of QoS characteristics is attained.

*QoS management functions* allow a specific QoS requirement to be met. *QoS alerts* are messages issued when limits or thresholds are crossed.

Finally, a *QoS policy* is defined as 'a set of rules that determine the QoS characteristics and QoS management functions' to be used by the system in a particular case.

The concepts defined by ISO are widely accepted, but are commonly used more freely. QoS characteristics, parameters, data and information are commonly used as synonyms, while the study of QoS mechanisms is mainly focused on QoS establishment, with maintenance, enquiry and management often being relegated to a second plan despite their important role in the achievement of QoS.

### 2.5.2 QoS Specification

The desired quality of service has to be conveyed to QoS mechanisms in the form of QoS parameters. Parameters can be expressed in very different formats, varying according to the application field, the corresponding abstraction level, or even personal taste.

QoS mechanisms should be flexible enough to accept different formats of QoS parameters and interpret them correctly. 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', which is useful for video but unintelligible for audio.

Two main abstraction levels can be identified in systems subject to QoS specification: the application level and the system level. Parameters specified at different levels are related, but differ strongly in meaning. 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 network bandwidth to transfer data, maximum error and delay, processing time to compress and decompress the information, memory, access to specialised hardware, etc.

The application fields and the system components in which QoS mechanisms can be utilised vary enormously. Any attempt to define a common set of parameters for QoS specification to be employed by the user to specify its QoS requirements would likely constrain his expressiveness and make the process of QoS specification more difficult. Therefore, a compromise must be achieved between the needs of different application fields regarding the manner in which QoS requirements are expressed and the generalisation necessary for specifying QoS parameters to be enforced by QoS mechanisms.

### 2.5.3 QoS Mapping and Translation

QoS parameters have to be translated between different levels of abstraction to be meaningful for the mechanisms present at that level. Translation implies that a mapping exists between parameters at different levels. Mappings are not always one-to-one between parameters, but may be one-to-many, many-to-one or many-to-many.

| One-to-One | One-to-Many | Many-to-One | Many-to-Many |
|---|---|---|---|
| Error Rate<br>⇓<br>ATM Cell Loss Ratio | Deterministic Guarantee<br>⇓<br>Deterministic Scheduling;<br>Deterministic Transport;<br>Memory Paging Blocked | Video Resolution;<br>Colour Depth<br>⇓<br>Network Bandwidth | Network Bandwidth;<br>Packet Size<br>⇓<br>Token Bucket Size;<br>Token Bucket Rate |

**Table 2 – Examples of Parameter Mapping**

Table 2 shows some examples of parameter mappings. Although the mapping may be complex, the process of translation usually consists in simple arithmetic operations over a limited set of variables.

Mappings and translation mechanisms have to be bi-directional in order to allow the transfer of QoS information from the bottom layers of the system to the top ones, reporting QoS measures to the application using QoS parameters comprehensible at this level of abstraction.

For the particular case in which several different application fields and lower-level components must be supported, the translation process has to deal with different sets of parameters appropriate for the environment in which it is being used. Since the creation of direct (one-step) translators for N application fields deployed on top of M low-level components would need the definition of N * M translators, this solution seems to be unacceptable.

To allow support for diverse application fields and resource reservation protocols, a way of extending the mapping and the translation mechanisms must be provided. The addition of support for a new application field or reservation protocol should be as easy as possible. In other words, the addition of support for a new kind of application should not imply writing one translator for every reservation protocol supported by the architecture, and vice-versa.

### 2.5.4 QoS Enforcement

After obtaining the QoS requirements of the application in the form of QoS parameters and translating them into a format that is appropriate at a lower level of abstraction, mechanisms for QoS enforcement have to be used in order to fulfil these QoS requirements.

QoS enforcement is necessary in order to guarantee that the QoS requirements imposed by the application will be fulfilled. This can be achieved by controlling and monitoring the use of the resources provided by the underlying system, either directly by the QoS mechanism or indirectly by means of a resource reservation protocol [103]. Since the handling of resources would add even more complexity to QoS mechanisms, the second solution is the most widely adopted. Resource reservation protocols are discussed in more detail in item 2.6.

The allocation of system resources is even more complex if we consider that multiple resource reservation protocols can be present in the platform. Since one of our main goals in this thesis is to provide support for heterogeneous systems, the presence of multiple reservation protocols must be taken into account during the design of the mechanisms for QoS enforcement being proposed by this thesis.

### 2.5.5 QoS Architectures

QoS architectures consist in middleware components that are responsible for integrating mechanisms for QoS specification and enforcement in computational systems. The mechanisms provided by QoS architectures are used by applications in order to have their QoS requirements observed by the underlying system.

To allow the utilisation of the features provided by networks and operating systems with reservation capabilities at user level, several QoS architectures have been defined in the literature [4][24][27][102][156]. However, most of the QoS architectures proposed so far target only a specific configuration of operating system software and communication infrastructure. This tight dependency on a specific environment constrains their application in open systems, where heterogeneity is always present. ATM-based systems combined with real-time operating systems are the most popular platforms for the development of QoS architectures, because of their suitability for the implementation of QoS mechanisms based on the reservation of system resources. Some architectures are also targeted at very specific application fields, with video conferencing being the one where the technology is more mature because of several research projects that explored this topic.

### 2.5.6 ISO QoS Framework

The ISO QoS framework [70] consists of a model that "defines the architectural principles, the concepts and the structures that underlie the provision of QoS" in the basic reference model of Open Systems Interconnection (OSI).

The OSI basic reference model is provided with QoS entities that take part in QoS management. Two classes of entities are defined within the ISO QoS framework:

- System QoS entities: entities which have a system-wide role, i.e. are cross-layer; and
- Layer QoS entities: entities associated with the operation of a particular subsystem present in a specific layer.

System QoS entities interact with layer QoS entities in order to monitor and control the performance of the system.

The following layer QoS entities are defined by ISO:

- N-Layer Policy Control Function (N-PCF): receives the QoS requirements and determines the policy that is to apply to the operation of the N subsystem;
- N-Layer QoS Control Function (N-QCF): selects the entities that will participate in the provision of communication services by taking into account the QoS requirements received from N-PCF; and
- N-Layer Protocol Entity (N-PE): checks if the QoS requirements imposed on a communication service can be supported, possibly renegotiating them with peer N-PEs or indicating a failure when a requirement cannot be accepted.

At the system (cross-layer) level, the ISO QoS framework defines two functions and a set of entities. They are:

- a System Quality Control Function: a system-wide capability for tuning the performance of the various protocol entities that are in operation;
- a System Policy Control Function: a function that ensures system-wide consistency to the policies implemented by each individual N-PCF in a layer; and

- Management Entities: responsible for supporting OSI system management and general management operations and notifications necessary for QoS management (such as QoS data metrics and time management, for example).

As can be easily seen, the ISO QoS framework only defines a reference model for QoS specification and management at the network level. This follows the approach undertaken by ISO for the definition of the OSI reference model. Despite being a good starting point for understanding the structure of middleware that provides support for QoS specification and enforcement, the ISO QoS framework covers QoS only at the network level and lacks a more detailed architectural specification.

## 2.6 Resource Reservation

The deployment of a distributed application on fast machines interconnected by a network with sufficient bandwidth does not necessarily guarantee that an application will behave as expected.

The services available at the network and operating system level have to take into account the requirements of the application that is using these services during their execution. This does not mean that low-level details have to be accessible at application level, introducing more complexity to be dealt with by the programmer. Instead, we consider that resource reservation mechanisms will have to be provided to allow applications to easily control and manage the use of network and operating system resources. This strategy allows the communication support to guarantee the availability of bandwidth for the application and to impose an upper limit on the communication delays, for example. In the same way, the operating system is able to guarantee the availability of resources such as processing power and memory for the use of the application.

In order to achieve the desired system performance, QoS mechanisms have to guarantee the availability of the shared resources needed to perform the services requested by users. The concept of resource reservation provides the predictable system behaviour necessary for applications with QoS constraints.

Aiming to provide more guaranteed availability of resources for applications, networks and operating systems are incorporating mechanisms for resource reservation.

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 the usage of the resources and the guarantee levels that satisfy the user. Reservations are then accepted, if there are resources 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.

The concept of resource reservation, like QoS, originated from work on communication networks and was subsequently extended to other components of computational systems.

In the area of computer networks, we can mention the development of ATM as 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 proposing a new suite of protocols for the Internet. The suite of protocols is based on IP version 6 (IPv6) in conjunction with a reservation protocol (RSVP) and a new transport protocol named RTP. Both ATM and the new Internet protocol suite will be described in detail later.

At the operating system level, some work has been developed to extend conventional operating systems in order to provide more predictable behaviour suitable for applications with QoS constraints. Microkernel-based operating systems such as Mach [145] have been extended to provide time-constrained services. Conventional commercial versions of UNIX, such as Sun's Solaris and IBM's AIX, are also being adapted to provide behaviour suitable for applications with QoS constraints. In addition, some experimental operating systems [59][89][123] provide mechanisms for resource reservation that can be used by applications that need to have guaranteed access to system resources.

Each existing reservation protocol has its own interface and its own set of parameters or reservation messages for the specification of the resources to be reserved for use by the application. The handling of differences between multiple reservation protocols by QoS mechanisms then becomes a complex task that has to be considered carefully in the design of middleware that is intended to provide QoS constrained services.

### 2.6.1 Resource Adaptation

One important trend in research into resource reservation protocols is resource adaptation [52].

Some authors consider resource reservation as a guaranteed method of delivering a specified QoS to an application during its entire lifetime [36][88][101]. Another school of thought proposes the development of adaptive applications capable of dealing with changes in resource availability during the provision of service [25][26][30]. A third idea based on resource adaptation, which mixes both approaches mentioned previously, has been considered as a viable and necessary alternative to either.

Several drawbacks appear in efforts to provide guaranteed resource reservation services for the following reasons:

- the physical structure of the whole computer system may change due to hardware reconfiguration and computer mobility, and changes in the distribution of resources may become necessary;
- monitoring services may detect that the QoS requested by an application is not being achieved with the resources that were allocated for it, and then decide to allocate new resources;
- the system may reclaim resources that are not being utilised or that are necessary for more important activities.

On the other hand, adaptation is very limited for services with strong requirements, and does not solve the problems that applications with QoS requirements face when they try to use the best-effort systems currently available.

Resource reservation combined with adaptation entails a more flexible approach for providing QoS to applications [52]. In this approach, resources can be seen by applications as being guaranteed during some time, but their availability can vary over long periods. Applications are responsible for estimating their initial resource requirements and for negotiating their reservation with resource providers. During run-time the application has to be able to adapt its behaviour based on feedback received from the system.

QoS mechanisms have to be aware of the possibility of resource adaptation, making it transparent to the application whenever possible. When the agreed QoS is not reachable with the resources available, the application has to be informed that the previously agreed QoS has to be renegotiated. Applications have to be ready to handle this kind of situation without disruptions in the service being provided to the user. In other words, applications holding resources that are subject to changes in their availability because of resource adaptation have to be able to degrade gracefully when it occurs.

### 2.6.2 ATM

Asynchronous Transfer Mode (ATM) [16][113] is a communications standard proposed by the Telecommunication Standardisation Section of the International Telecommunications Union (ITU-T) as the transport mode for the Broadband Integrated Services Digital Network (B-ISDN) [71].

ATM networks aim to integrate traffic with different flow characteristics and service requirements. The type of traffic to be supported by ATM networks ranges from real-time data such as voice and high-resolution video, which can tolerate loss of data but not delay, to non-real-time traffic such as computer data and file transfer, which can tolerate delay but not loss of information.

The flow can vary from CBR traffic as required by non-compressed audio or video, to the bursts of data typical of computer systems where the average arrival time between bursts may be quite large and randomly distributed.

The main idea behind ATM is to transmit small data packets (ATM cells) asynchronously without error recovery at the network level. Each ATM cell consists of 53 bytes of data, of which 5 bytes are used by the ATM header. The adoption of an asynchronous approach allows better performance in the presence of bursts of data with low latency. Error correction and data recovery can be implemented at higher levels if necessary, according to the characteristics of the data being transmitted.

The transmission of data is completely connection-oriented, with data flowing through ATM channels. Each channel is identified by a channel number, which is composed of a Virtual Channel Identifier (VCI) and a Virtual Path Identifier (VPI). The channel number appears in the header of each ATM cell.

ATM supports point-to-multipoint and multipoint-to-multipoint connections in addition to the usual point-to-point communication. New members – either senders or receivers – can be added dynamically to multipoint connections by using the ATM signalling protocol. It is important to notice that communication is always unidirectional in multicast connections (i.e., data flows from senders to receivers only), and traffic characteristics and service requirements (QoS) must be the same for every member. Some manufacturers of ATM equipment still do not provide full support for multipoint communication.

There is no defined mapping between ATM and the seven layers of the OSI Reference Model [68]. The ATM Reference Model goes beyond the OSI model, introducing the concept of planes subdivided in layers in a three-dimensional structure. The use of planes allows the representation of features such as control and management in parallel with data transfer. The structure of the ATM architecture is shown in Figure 3 and described below.

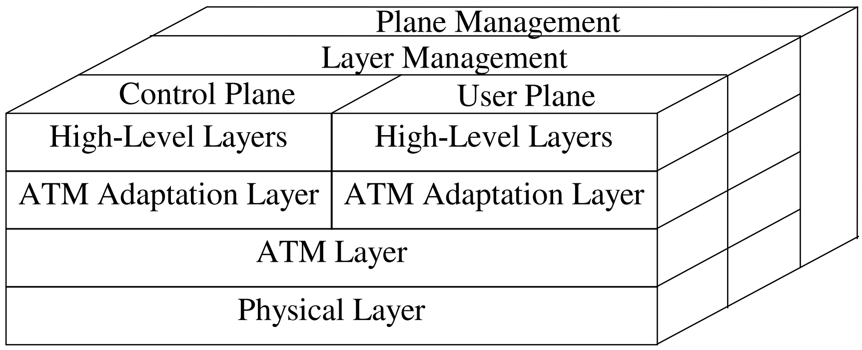


**Figure 3 – The ATM Architecture**

The three planes defined by the ATM architecture are:

- User Plane: for transporting user information.
- Control Plane: deals with channel control, connection control, and signalling information.
- Management Plane: implements layer management and plane management.

The layers are divided as follows:

- Physical Layer: performs bit-level functions, adapting the cell-based approach to the transmission medium. The physical layer is subdivided into:

    - Physical Medium (PM): contains only the medium dependent functions. It provides bit transmission and alignment, performs line coding and also electrical/optical conversion when necessary. PM layers for optical fibre, and coaxial and twisted pair cables are defined in the reference model. This layer also includes bit-timing functions such as insertion and extraction of timing information.

    - Transmission Convergence (TC): takes care of generation, recovery and adaptation of transmission frames, adapting the cell flow according to the transmission system. TC also performs the cell delineation function, enabling the receiver to recover the cell boundaries, and calculates and verifies the header checksum (HEC), allowing correction of header errors. TC also inserts idle cells in the medium in order to adapt the rate of the ATM cells to the capacity of the transmission system, and suppresses all idle cells received.

- ATM Layer: is the core of the ATM network architecture. This layer performs the following functions:

    - Cell header generation/extraction: adds the appropriate ATM cell header (except for the HEC value) to the cell received from the ATM adaptation layer (AAL) in the transmit direction, and does the opposite (i.e., removes the cell header) in the receive direction. Only cell data fields are passed to the AAL.

    - Cell multiplexing and demultiplexing: multiplexes cells from individual virtual paths (VPs) and virtual channels (VCs) into a single cell stream in the transmit direction. It divides the arriving cell stream into individual cell flows (VC or VP) in the receive direction.

    - VPI and VCI translation: performed at the ATM switching and/or cross-connect nodes. The values of VPI and VCI fields of each incoming cell are translated into new VPI and VCI values in the outgoing cell according to the characteristics and the state of the ATM switch.

    - Generic Flow Control (GFC): supports control of the ATM traffic flow in a customer network. This is defined only at the user-to-network interface (UNI); GFC does not exist in the network-to-network interface (NNI) cell.

- ATM Adaptation Layer (AAL): performs the adaptation of higher layer protocols. AAL is divided into two sub-layers:
  - Segmentation and Reassembly (SAR): performs segmentation of the higher layer packets of data into a suitable form for the payload of the ATM cells of a virtual connection; at the receive side, it reassembles the contents of the cells of a virtual connection into data units to be delivered to the higher layers.
  - Convergence Sublayer (CS): performs message identification and time/clock recovery. This layer is further divided into a Common Part Convergence Sublayer (CPCS) and a Service Specific Convergence Sublayer (SSCS).

Four classes of service (namely A, B, C and D) were originally identified by the ITU-T, each one providing a data flow appropriate to a different set of applications with similar traffic characteristics. For each class of service a corresponding adaptation layer (AAL1 for class A traffic, AAL2 for class B, and so on) was defined. Table 3 summarises the original relationship between classes and AAL types.

| Class of Service | A | B | C | D |
|---|---|---|---|---|
| Original AAL | AAL1 | AAL2 | AAL3 | AAL4 |
| Timing Relation | Preserved | | Not Preserved | |
| Bit Rate | Constant | Variable | | |
| Connection Mode | Connection Oriented | | | Connectionless |
| Original Target Applications | Circuit emulation, non-compressed audio and video. | Compressed audio and video. | Virtual circuit services (X.25, Frame Relay, etc.). | Datagram service, IP over ATM, LAN emulation. |

**Table 3 – Original ATM Classes of Service and Adaptation Layers**

However, during the specification process it was recognised that this approach needed to be modified. As a result, new AAL types and classes of traffic were defined, and different combinations between them were allowed when appropriate. Table 4 shows the resulting classes of services and AAL types.

AALs 3 and 4 were merged because of their similarities, giving rise to AAL3/4. Nevertheless, the 4-byte per cell overhead required by the headers and trailers necessary to provide connectionless service multiplexed over a single ATM channel constrained AAL3/4 adoption to class D services, and AAL5 was created to provide a more efficient interface for class C services. AAL5 is also used by the ATM signalling protocol (ITU-

T Q.2931) and by a new class Y, known as available bit-rate (ABR), which is suitable for delay-tolerant applications.

| Class | A | B | C | Y | Q.2931 | X | D |
|---|---|---|---|---|---|---|---|
| AAL | AAL1 | AAL2 | AAL5 | | | AAL0 | AAL3/4 |
| Time | Preserved | | Not Preserved | | | | |
| Bit Rate | Constant | Variable | | | | | |
| Conn. Mode | Connection Oriented | | | | | | Conn-less |
| Examples of Application | Circuit emulation | Compressed audio/video | Conn-oriented services | Available Bit Rate | Signalling | Cell-relay | Conn-less services |

**Table 4 – Resulting ATM Classes of Service and Adaptation Layers**

A class X, or unassigned bit-rate (UBR), was also introduced for provision of cell-relay service for applications with minimum QoS requirements, i.e., delay-tolerant and subject to cell loss. Class X employs a null AAL (also known as AAL0) to access the cell-relay service provided by the ATM layer.

Class B service is not currently available, because there is no standard or vendor-specific proposal for AAL2.

ATM is a technology under development. Despite not being completely standardised, this technology is evolving rapidly and being widely accepted in the computer network market, with applications in both local and wide area networks. A large number of vendors are providing ATM hardware, with a reasonable degree of interoperability between equipment from different vendors.

**QoS and Resource Reservation**. During the connection establishment process, the application can specify the desired QoS for an ATM channel based on parameters such as maximum number of cells per second, statistical variation of the volume of data being transmitted, maximum end-to-end delay, and maximum jitter allowed. A connection is refused when the network cannot support the required QoS level. The acceptance or refusal of a QoS specification is determined during an admission process, and rejection may occur due to lack of network resources.

The API used for requesting QoS varies from vendor to vendor, and also depends on the abstraction layer at which the specification is taking place. The format of QoS parameters can also vary considerably. A common approach for QoS specification uses extensions to the sockets API based on `ioctl()` calls.

### 2.6.3 Internet Protocols

The next version of the Internet suite of protocols, known as Internet Next Generation, provides resource reservation and guaranteed end-to-end QoS at application level.

The work carried out by the IETF (Internet Engineering Task Force) is orthogonal to the subnetwork-specific solutions for broadband integrated services networks (such as ATM) because it targets the logical network.

All this effort to adapt the Internet to a new communication framework is being developed in parallel with the new version of the IP network protocol, known as IPv6 (version 6) [34][40][64]. IPv6 is designed to support a larger set of addresses (128 bits instead of 32) while maintaining compatibility with the current version. Additionally, IPv6 has the following characteristics:

- IP packets can be associated to a flow through the use of a field in the IP header;
- multicast communication is improved through the addition of a scope definition field in the IP header, limiting the geographical amplitude of multicast messages;
- a new communication paradigm, called anycast, has been introduced; anycast supports groups of receivers to which messages are sent only to the nearest member of the group;
- packet options are implemented in a more flexible and extensible way, using multiple option fields outside the area of the IP header; and
- support for authentication and QoS specification is allowed through the definition of traffic flows with particular requirements and characteristics.

The new suite of Internet protocols being developed by the IETF is illustrated by Figure 4.

At the network layer, the IETF has two parallel activities underway. The first is addressing the provision of a Stream Transport Protocol (ST-II). The alternative activity is based on a Resource Reservation Protocol (RSVP) running over IP. Both solutions support multicast communication, resource reservation, bandwidth management and QoS specification, but the approach adopted in their design differs in several fundamental points.

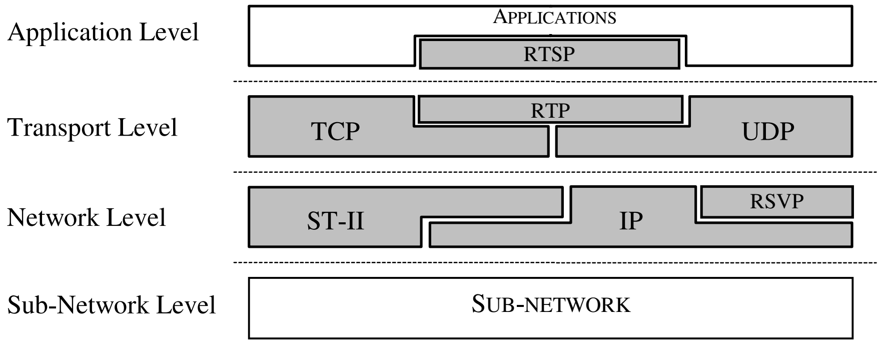


**Figure 4 – Internet Next Generation Protocols**

ST-II [41] is a connection-oriented network protocol, designed to coexist with IP, which allows resource reservation determined by the origin of the flow. New receivers interested in participating in an ongoing connection have to contact the origin to request their inclusion. An associated control message protocol, SCMP, is responsible for initiating and modifying connections, allowing resource specifications to be changed to fulfil the requirements of a specific receiver. The main disadvantage of the approach adopted by ST-II is related to the bottleneck imposed by the excess of tasks assigned to the source of the media, which constrains the scaling of receiver groups.

On the other hand, RSVP [19] is based on the establishment of flows over a connectionless network service (IP version 4 or 6), allowing reservation of resources for a specific flow when necessary. The multipoint communication service is designed to scale to very large groups through the use of flow specifications defined by receivers.

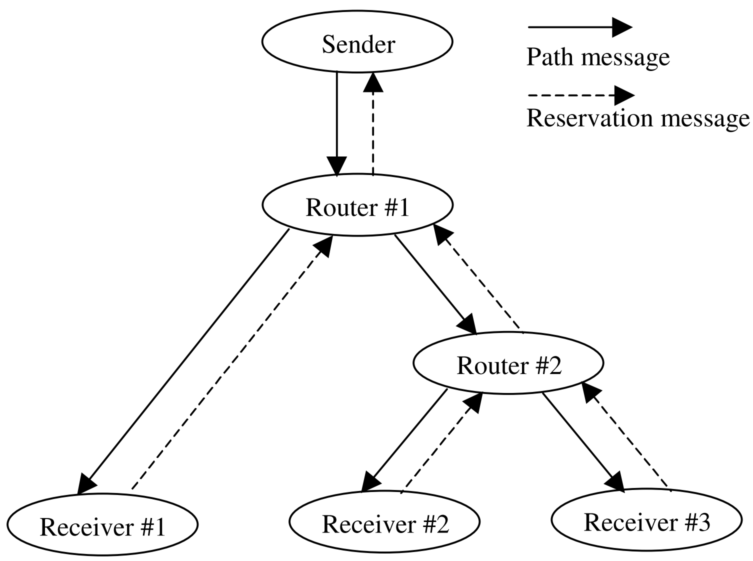


**Figure 5 – Propagation of RSVP Messages**

Figure 5 shows the exchange of RSVP messages in a multicast flow. Each receiver interested in a communication flow enters the corresponding multicast group and waits for a path message that describes the composition of the flow. The receiver then decides in which part of the flow it is interested (e.g. only the black and white component of a coloured video flow, only the audio of a film, etc.) and generates a filter. A flow specification (*flowspec*) describing the required characteristics of the flow is also generated by the receiver. It then arranges both the filter and the *flowspec* in a reservation message, which is propagated periodically in the direction of each individual origin of the flow. Routers along the path are responsible for reserving the resources needed for the flow described by the reservation message. Reservation messages are short-lived, having to be periodically refreshed by the receivers. This results in soft state information being stored by the routers along the path. In the absence of refresh messages, the reservation expires. The storing of soft state is an appropriate approach for implementing reservation of resources in the Internet, which was originally designed as a stateless infrastructure in order to tolerate failure and to scale gracefully.

At transport level, the usual Internet transport layers, TCP and UDP, are being adapted to take advantage of the new characteristics of IPv6. Furthermore, a new protocol called Real-Time Protocol (RTP) has been proposed.

RTP [131] provides higher level services such as encapsulation, multiplexing and demultiplexing, encoding, synchronisation, error detection and encryption. RTP can make use of the other transport protocols (TCP or UDP) or access the network layer directly (IP or ST-II). A transport control protocol named RTCP is responsible for group management, connection control, and QoS monitoring.

An application protocol developed by Netscape Corporation and Real Networks was accepted by IETF as a draft standard. The Real-Time Streaming Protocol (RTSP) offers a “WWW-based and HTTP-friendly” interface to allow on-demand delivery of real-time media over the Internet [132]. RTSP is built on top of the currently available transport protocols (RTP, TCP and UDP). The protocol provides mechanisms to request delivery of real-time data, specify the transport layer to be used and the destination for the delivery, request information about the encoding format, control the flow of data through the stream, and to access portions of data when applicable (i.e., in case of recorded media).

With the introduction of these protocols, the IETF intends to provide a complete suite of mechanisms for the transmission of continuous media and real-time data through the Internet.

**QoS and Resource Reservation**. The most common method used by Internet applications for specifying their requirements is by using RSVP, and it implies defining a token bucket model [19] for the data traffic generated by the application. According to this model, the transmission capacity of the network is modelled by a bucket that is filled with tokens in a constant rate; for each data unit transferred through the network, a token must be taken from the bucket; the size of the bucket limits the occurrence of network bursts.

A series of calls to the sockets API is necessary to set up RSVP and then perform reservations of network resources by specifying parameters such as the size of the token bucket, the token rate, packet size, etc. Like ATM, RSVP performs an admission process that determines whether the requirements can be met by the network or not. In addition to the token bucket style of reservation, RSVP also provides other methods for specifying resource reservations. However, these new methods are still in the process of being defined by the IETF and added to the RSVP API.

### 2.6.4 Real-Time Capabilities of Windows NT

The Microsoft Windows NT operating system provides a set of mechanisms that make it possible to be used by applications with timing constraints. By definition, Windows NT is a general-purpose O.S. with soft real-time capabilities and not a hard real-time O.S. that provides deterministic response times [96]. Despite lacking support for hard real-time computing, Windows NT still consists of a useful platform for the deployment of multimedia applications and non-critical real-time applications [144].

The real-time mechanisms provided by Windows NT consist of:

- A real-time scheduling class, which isolates real-time tasks from the ones satisfied by the best-effort scheduling policy that is used by default by the system scheduler;
- Mechanisms for blocking memory paging, which makes the access times to the physical memory more predictable;

- Asynchronous I/O, which allows applications to queue an I/O operation and continue processing without having to either wait or respond to an end-of-I/O event;
- Bounded latency for handling interrupts; and
- Synchronisation mechanisms such as timers, events, mutexes and semaphores.

These mechanisms are accessible for the application programmer either through the Windows 32-bit (Win32) API or through the Windows Sockets (WinSock) API. In particular, WinSock extends the sockets API by providing alternative versions of the original API calls with additional parameters, which allow the user to execute asynchronous I/O and to specify QoS requirements in the form of a traffic descriptor, for example.

Later in this thesis we will make use of the real-time mechanisms provided by Windows NT in order to build middleware that provides support for applications with timing constraints imposed on their computation.

## 2.7 Summary

This chapter has situated the area of research in which this thesis is inserted. Concepts in the fields of multimedia and real-time applications, open systems, distributed object computing, QoS and reservation protocols have been presented.

The immediate goal of this thesis is, by using the knowledge gathered in these area of research, to provide an environment in which QoS can be achieved in the deployment of critical applications (such as multimedia and real-time applications) in open systems. Technologies such as resource reservation and distributed object computing have a key role in providing a solution for this problem.

In this chapter we have also presented a series of well-established technologies which can be used for building a QoS architecture suitable for open systems. On the one hand we have reference models such as MSS and the ISO QoS Framework that provide a series of insights that can be used for designing the QoS Architecture. On the other hand, technologies such as CORBA, real-time operating systems, ATM and the new Internet protocols indicate the sort of support that a QoS architecture can rely upon.

# 3

# State of the Art

The following sections present a number of frameworks and architectures that provide QoS mechanisms for applications with QoS requirements. During this surveying study, we have identified a series of limitations that prevent the use of state-of-the-art technology to provide support for QoS specification and enforcement in open systems. These limitations are targeted by a new proposal of QoS architecture, which will be described in the next chapter of this thesis.

## 3.1 XRM and Xbind (Univ. of Columbia)

The eXtended Reference Model (XRM) and the Xbind platform [88] have been designed and implemented by the Comet group at the University of Columbia.

XRM models the communication architecture of networking and multimedia computing platforms using three components: the broadband network, the multimedia network, and the services and applications network.

The broadband network is defined as the physical network. Upon this physical infrastructure resides the multimedia network, whose primary function is to provide support for services with end-to-end QoS guarantees. The user has contact only with the service abstractions provided by the services and applications network.

The services visible at the multimedia network level are provided by Xbind, a programming platform for creating, deploying and managing multimedia services. Xbind is basically a conceptual framework for creation, deployment and management of multimedia services on ATM networks, which allows the specification and control of end-to-end QoS.

Xbind adopts an open signalling concept called "hardware independent signalling". The main idea entails creating a separation between switching software and hardware. This allows, for example, the installation of signalling software on an ATM switching platform from a third party vendor, simplifying the service creation, deployment and management.

Functionally, Xbind consists of a kernel that provides end-to-end QoS over broadband networks, including software components for implementing mechanisms for distributed network resource allocation, broadband signalling, real-time switch control, multimedia transport and device management.

The associated binding architecture presents the service provider with a set of open binding interfaces with multimedia and QoS capabilities. Additionally, the Binding Interface Base (BIB) allows an easy integration of control (connection establishment and resource allocation), network management and service management.

The Xbind kernel has been ported and tested on several operating systems and different types of ATM equipment. The CORBA architecture was adopted as the lower level for implementing the binding architecture, providing interoperation between multiple platforms.

Xbind is an example of a multimedia programming framework that provides network-level QoS mechanisms. Some degree of system heterogeneity is provided by the use of CORBA and by the different versions of the Xbind kernel that were built on different platforms. However, it lacks some features that are required by applications with QoS requirements deployed in open systems. Heterogeneity at the resource reservation protocol level is not supported due to the tight integration of the system with the network protocol (i.e. ATM). Low-level details regarding the network transport are not hidden completely from the programmer. In addition, Xbind does not provide support for QoS adaptation, and there is no mechanism for extending the programming framework.

## 3.2 ReTINA Project (ACTS)

Real-Time TINA [18] is a cooperative project sponsored by the European ACTS program that was carried out by Chorus Systems, Alcatel, Siemens, HP, CSELT, France Telecom, British Telecom, Telenor, APM, O2 Technology and Broadcom.

The main goal of the ReTINA project was to develop a distributed processing environment (DPE) with support for interactive real-time and multimedia services, following standards such as CORBA [108] and ODP [39]. Support for real-time and multimedia traffic is obtained with the encapsulation of multiple communication protocols within an ORB, providing flexible bindings and interaction models, and with fine-grained control of resources in order to provide QoS guarantees.

ReTINA adopts a binding protocol based on the ODP reference model for establishing connections (i.e. bindings) between objects. This protocol introduces binding factory objects responsible for the provision of binding models appropriate for real-time and multimedia applications, such as third-party and multiple bindings, and with specification of QoS for bindings. In order to provide QoS guarantees for bindings, the resources necessary to achieve the desired QoS must be reserved for the use of the 'ODP capsule' (address space) of the corresponding object.

In addition to the core of the DPE, which encompasses the ReTINA ORB, several peripheral services make up the ReTINA environment. These services are based on the CORBA services defined by the OMG, including trading, notification and events, security, replication and transaction services as well as telecommunication-specific services such as connection managers and QoS providers.

Stream abstractions are supported by means of extensions to the IDL language that allow stream interfaces to be specified. Stream interfaces have operations responsible for transferring frames of media and do not have return values or argument directionality specification. Arguments specify the format of the frames transmitted through the stream. Stream interfaces are referenced via regular CORBA object references that may be passed as arguments of operations or stored by the CORBA name service.

The ReTINA environment is based on a multi-platform ORB, which runs with a real-time 'profile' over the Chorus micro-kernel [142] and with a non-real-time 'profile' over generic operating systems. Furthermore, ReTINA provides a set of application development, configuration and management tools aimed at facilitating the process of deploying applications on top of the ReTINA environment.

ReTINA combines support for real-time and multimedia applications, but QoS specification mechanisms still are not user-extensible. Like Xbind, it enforces QoS only at network level, and important features that are necessary in an open environment such as support for QoS adaptation are still missing.

## 3.3 DIMMA (APM-ANSA)

The DIMMA (Distributed Interactive MultiMedia Architecture) project [116] proposes to extend currently available distributed computing architectures in order to incorporate real-time and multimedia capabilities. This research project was carried out by the APM-ANSA Laboratories.

The main objectives of the DIMMA project were:

- to extend distributed computing object models such as CORBA for supporting real-time and interactive multimedia applications;
- to incorporate support for new paradigms of communication best suited for multimedia and real-time systems such as streams (e.g. audio/video communication) and signals (real-time processing) in distributed systems;
- to identify extensions needed to support distributed services management, fine-grained control and monitoring of resource usage and management (e.g. explicit binding);
- to define an architecture for QoS negotiation, management and control; and
- to investigate of the interworking of distributed object models (e.g. CORBA vs. ODP) at the computational level (e.g. IDL translations) and at the engineering level (e.g. transport protocols).

The adopted approach consists in using a set of distributed computing environments (ANSAware [91], DCE [114], and various ORBs) and operating systems (ranging from desktop to real-time operating systems) as a basis for constructing a higher-level API.

DIMMA adopts several partial models for structuring the distributed environment. An object model incorporates features appropriate to real-time and multimedia environments. An interaction model adds the concept of streams and group communication to the traditional client-server model. In addition, a control model regulates both synchronous and asynchronous communication. Implicit and explicit bindings are provided through the adoption of binding managers at both client and server sides. The QoS model addresses the requirements present in multimedia and real-time environments, allowing specification and control of end-to-end quality of service. Finally, the scheduling model provides resource allocation, priority scheduling mechanisms and deadline control.

DIMMA provides a stub generator toolset that supports several kinds of IDLs and the generation of client and server stubs for multiple platforms (e.g. ODP, CORBA). The toolset is built around the Abstract Syntax Tree concept (AST), as a syntax-free way to capture the semantics of interfaces. The stubs generated are, whenever possible, independent from the engineering details such as transport used, and marshalling algorithms. The AST also serves as a data interchange format between the tools and services in the DIMMA development and runtime environment (type and scope checker, interface repository, and trader).

The AMBER project [86], also carried out at APM-ANSA Laboratories, provides a CORBA-compliant environment suitable for distributed multimedia and telecommunication applications by applying solutions proposed by APM in the DIMMA project and also from TINA and ODP. The approach adopted is based on the incorporation of the concept of stream interfaces to the CORBA IDL and the extension of object adapters in order to support streams.

DIMMA and AMBER, like the other systems presented before in this chapter, still have limited support for QoS specification and enforcement. Despite their wide platform coverage, these architectures do not take into account the possibility of having multiple reservation protocols in open systems. In addition, mechanisms for the application programmer to extend the QoS mechanisms are not provided by both architectures, which also lack support for QoS adaptation.

## 3.4 SUMO (Univ. of Lancaster)

The SUMO Project (SUpport for Multimedia in Operating systems) [35] was developed by the Distributed Multimedia Group at Lancaster University and by CNET at France Telecom. SUMO is a microkernel-based system that aims to provide facilities to support distributed real-time and multimedia applications in ODP-based platforms.

The SUMO project provides the application programmer with a QoS-driven API, a connection-oriented communication system with connection-dedicated resources and facilities for monitoring and controlling the QoS of multimedia applications.

A set of key architectural principles was adopted in the design of SUMO. They are:

- upcall-driven application structuring, whereby communications events are initiated by the system rather than by the application;
- split-level system structuring, which means that key system functions are carried out cooperatively between kernel and user level components; and
- decoupling of control and data transfer, i.e. the control is carried out asynchronously with respect to the transfer of data.

The Chorus microkernel [142] was used as a vehicle for providing the operating system level services as well as basic real-time support. The design of the SUMO distributed real-time/multimedia support system was implemented partly in kernel space and partly as a user-level library that is linked to native Chorus applications.

In parallel with the development of SUMO over Chorus, the necessary requirements to support synchronisation of continuous media in open distributed processing environments were investigated by the creators of SUMO [36]. Sophisticated support for synchronisation was proposed as a result of this work. The coordination of multimedia presentations, and the maintenance of timing relationships between distinct data transmissions and of event-based synchronisation points is allowed through the cooperation of invocation mechanisms with new services introduced at system level, called streams and synchronisation managers. These services allow the verification of synchronisation constraints imposed on the exhibition of distributed media.

The use of the SUMO system as a support infrastructure for CORBA was also investigated [37]. The developers of SUMO have proposed extensions to the CORBA computational model in order to incorporate interfaces for multimedia streaming, and QoS ‘annotations’ (i.e. specifications) on interfaces along the lines suggested by the ISO Reference Model for Open Distributed Processing (RM-ODP) [69].

SUMO concentrates on providing QoS at operating system level only. Its use is limited in open systems due to the lack of support for multiple reservation protocols; its area of application is limited to multimedia; mechanisms for extending the run-time support are not provided; and it lacks support for QoS adaptation.

Other research initiatives related to multimedia and group communication, as well as end-to-end QoS over integrated services networks, are also being undertaken by the Distributed Multimedia Group at Lancaster University.

The experience gained with the SUMO project was employed by the research group for developing the QoS-A architecture, which is described in section 3.8. Conclusions of this project also contributed for the MMN project, described in section 3.12.

## 3.5 TAO (Washington University at St. Louis)

The TAO project intends to optimise the performance of CORBA implementations for performance-sensitive applications such as hard real-time systems (e.g. avionics) and constrained latency systems (e.g. teleconferencing).

Earlier studies and performance measurements [127][128] identified severe bottlenecks in the currently available CORBA implementations, which limit their utilisation for building applications with real-time requirements. To allow the development of a CORBA-compliant platform suitable for this kind of application, several changes in the way that the ORB processes remote method invocations were proposed in order to take real-time constraints into account. In addition, support for high-speed networks (e.g. ATM) and mechanisms for QoS specification and enforcement were identified as aspects that have to be added to the ORB.

Aiming to overcome the limitations that constrain the use of CORBA technology in performance-sensitive systems, a real-time ORB called TAO (also known as “the ACE ORB”) has been designed and implemented [129][130].

The proposed approach to solving the observed performance problems introduces a real-time object adapter, responsible for processing real-time requests, and an ORB scheduler which implements both dynamic and offline real-time scheduling algorithms. Changes are introduced at protocol level in order to allow application programmers to plug customised communication protocols into the ORB.

Other improvements at the protocol level include the introduction of scheduling queues for processing requests and optimisations in data copying, marshalling operations, and in the multiplexing of requests over communication links. At the system level, TAO employs a highly optimised real-time I/O subsystem and a high-speed networking infrastructure.

TAO is an example of real-time middleware that provides real-time applications with mechanisms for imposing QoS constraints at operating system level. Being compatible with CORBA, TAO allows applications to interact in an open environment Nevertheless, it does not have support for enforcing end-to-end network-level QoS constraints, and depends heavily on an optimised O.S. kernel. In addition, TAO lacks important features such as platform transparency, extensibility and support for adaptation.

## 3.6 Arcade (CNET and ENST)

The Arcade project [42] was proposed by CNET (France Telecom Research Centre) and the Department of Informatics at the *École Nationale Supérieure des Télécomunications* (i.e. the French National Superior School of Telecommunications). Arcade is conceptually a platform that aims to provide support for distributed multimedia applications. Its main concern is the area of system-level QoS.

Arcade provides a scheduling framework that automatically derives scheduling information from temporal QoS constraints. This framework allows the description of high-level temporal QoS constraints in the form of arithmetic equations using a language called QL (QoS Language). These equations are translated automatically into scheduling information used by a user-level scheduler. This scheduler works together with the operating system scheduler, to make a best effort at allowing applications to meet their deadlines.

In this framework, both schedulers follow the Earliest Deadline First (EDF) policy. Temporal QoS equations are translated into deadlines and used as parameters for the scheduling of system threads. An application is implemented as a multithreaded process with one or more objects, and each invocation of a method of an object creates or activates a thread.

A library of lightweight user-level threads is employed for implementing the framework, allowing that switches between threads of the same application occur without interaction with the system kernel. Arcade is based on a variation of the Chorus kernel with extensions to support the EDF policy.

Like TAO and SUMO, Arcade is not able to enforce QoS at network level, since it concentrates on providing QoS at the O.S. level. Its main difference is the definition of more complex specification and translation mechanisms based on the QL language. Nevertheless, Arcade still suffers from the same limitations that prevent the use of the systems described previously for providing QoS in open systems.

## 3.7 OMEGA and The QoS Broker (Univ. of Pennsylvania)

The OMEGA Architecture [101] is a QoS architecture for provision of real-time guarantees in distributed multimedia systems. In order to provide service guarantees, the research effort focused on local and global resource management in distributed systems.

The QoS Broker [102] is the main component of the OMEGA Architecture. It is basically a middleware component responsible for the negotiation of QoS levels to be delivered to the application by the underlying system. The requirements specified by the application are translated by the QoS Broker and result in the negotiation of resource allocations with the operating system and the network.

Resource reservation is based on QoS parameters associated to the specific media device, and handled internally by the QoS Broker. A translator is employed to obtain lower-level QoS requirements from the application-level parameters specified by the user, adopting a fixed set of translation relations for each media type. The translation is bi-directional, allowing dynamic changes in resource reservations to be reported to the user as application-level QoS parameters. The obtained parameters result in the reservation of network and operation system resources, both local and remote. Consequently, a local QoS Broker aiming to perform reservations of remote resources, called a ‘buyer’, is responsible for interacting with other QoS Brokers located remotely, known as 'sellers' of resources. A 'buyer/seller protocol' is provided by the architecture to allow sellers to advertise their services and buyers to contact sellers and therefore avail of their resources.

The QoS Broker employs orchestration services (i.e. mechanisms for balancing resource usage) and relies on information stored in resource databases to achieve a balance among the resources taken from multimedia devices, operating systems and the network. The underlying operating system is assumed to have real-time capabilities, allowing that its temporal behaviour be predicted by the QoS Broker and used to perform the orchestration of resources.

In addition to the QoS Broker, the OMEGA Architecture provides a communication model that consists of two protocols at distinct abstraction levels. The Real-Time Application Protocol (RTAP) is responsible for tasks such as call management, device management, synchronisation, and media delivery at application level; and the Real-Time Network Protocol (RTNP) implements functions for connection management, error correction, rate control, and network access at the transport level. These protocols provide guaranteed communication services over specified communication channels, which are obtained by applications and configured by them through the QoS Broker.

The OMEGA architecture and the QoS Broker were validated through two practical experiments [104]. In the first application, a robot interconnected to a dedicated ATM network was operated and monitored remotely by workstations running the AIX operating system with real-time extensions. In the second example, a video-on-demand application with lip synchronisation was implemented on a platform composed by SGI Indy machines running the IRIX operating system, interconnected by a 10Mbps Ethernet LAN.

OMEGA has more complete QoS enforcement mechanism than the architectures described in the previous sections. Besides being able to perform reservations at both operating system and network levels, OMEGA provides a complete QoS translation mechanism that makes the underlying platform transparent for the application. However, OMEGA does not allow the application programmer to extend the architecture, does not offer provision for multiple reservation protocols, and its support for adaptation consists of simple notification mechanisms.

## 3.8 QoS-A (Univ. of Lancaster)

The Distributed Multimedia Research Group at Lancaster University have developed a QoS Architecture, called QoS-A [24], which offers a framework for specifying and implementing the required performance properties of multimedia applications over high-performance ATM-based networks.

The architecture incorporates the following notions to an environment composed of machines interconnected by an ATM network:

- flow: single media streams that represent the production, transmission and consumption of media data;
- service contract: agreement between users and service providers regarding the QoS level to be achieved; and
- flow management: monitoring and maintenance of the contracted QoS levels.

QoS-A makes possible the provision of data flows with an associated level of QoS through the tight integration between devices, end-system and network. QoS-A provides 'system-wide' QoS, including end-systems, communications systems and networks, through the use of QoS mechanisms that span across all architectural layers.

A structure of layers and planes integrated with the layers and planes of the ATM architecture (see Figure 3 in Section 2.6.2) provides a QoS-configurable communication mechanism, while thread scheduling algorithms based on QoS constraints provide the desired behaviour at system level. In addition, devices are built with the intent of taking advantage of both scheduling and communication with QoS capabilities.

Three planes compose the QoS-A architecture:

- the Protocol Plane is responsible for data transfer. It is subdivided into a user plane, for transmission of media data, and a control plane, responsible for control data, because of the essentially different requirements of these distinct categories of data.
- the QoS Maintenance Plane is responsible for monitoring and maintaining the QoS levels specified by the user in the service contracts that are accepted by the architecture. Based on the information monitored from the system, this layer

configures the available resources aiming to provide the QoS levels agreed with the user.

- the Flow Management Plane executes flow establishment procedures (including admission control and resource reservation), QoS renegotiation, QoS mapping and translation, and QoS adaptation.

The first two planes are subdivided into layers as follows:

- the distributed systems platform provides services for multimedia communication and QoS specification.
- the orchestration layer is responsible for media synchronisation and jitter correction.
- the transport layer provides a basic QoS-configurable communication service.
- the lower layers – network, data link and physical layer – provide the basis for the communication service.

QoS-A is implemented over an enhanced Chorus micro-kernel and enhanced protocols for multimedia built on a local ATM network. Like OMEGA, it is able to translate QoS requirements and reserve resources provided by both the network and the operating system. Its QoS adaptation mechanism is more complex than the one provided by OMEGA, since it is able to perform transparent QoS adaptation (i.e. without involving the application) and not only notify the application when QoS changes. However, QoS-A still has limitations similar to the ones present in OMEGA and that make both architectures unsuitable for open environments. QoS-A is not able to interact with different resource reservation protocols due to its tight integration with the underlying system. In addition, its area of application is constrained to multimedia systems, and its QoS mechanisms cannot be extended by the application programmer.

## 3.9 QuO (BBN Systems and Technologies)

The Quality of Service for CORBA Objects (QuO) architecture [156] provides QoS abstractions that can be used by CORBA objects distributed in a wide area network. QuO extends CORBA's Interface Description Language (IDL) with a QoS Description language (QDL).

The QDL language allows QoS to be described at object level (i.e. methods per second) instead of at communication level (i.e. bits per second). By using QDL, users describe:

- QoS contracts between a client and a server object in terms of usage of the server and the QoS requirements imposed on services provided by the server;
- the internal structure of objects and the amount of resources they require; and
- the resources available in the system and their status.

QoS regions can also be described with QDL, allowing the application to adapt to changing conditions by changing from one QoS region to another.

QuO defines architectural components that are responsible for dealing with QoS enforcement, measurement and adaptation on behalf of the application. These architectural components are regular CORBA objects that are generated based on the IDL and QDL descriptions.

QuO provides a complete framework for application development that reduces the programming effort required for writing applications with QoS requirements. However, it still has limitations similar to the ones present in OMEGA and QoS-A. Despite being compatible with CORBA, its use in open systems is limited by the fact that there is no support for multiple reservation protocols. In addition, its efficiency as a programming support is limited by the fact that the CORBA object model is not suitable for most real-time and multimedia applications.

## 3.10 QUANTA (Old Dominion University)

The Quality of service Architecture for Native TCP/IP over ATM networks (QUANTA) [43] is an architecture that provides mechanisms for QoS specification and enforcement for applications by using communication services based on Internet protocols running over ATM.

Basically, it allows applications that need a network service with predictable performance to specify a region of operation (ROP) in terms of user-level QoS parameters. Applications are characterised according to classes of service, such as the classes defined for ATM. Each class has a translator that converts parameters specific to that class into generic network QoS parameters, such as throughput, delay, loss, etc.

These generic parameters are then translated into end-system protocol-dependent control parameters by a protocol-dependent translator. These parameters are used to configure the lower-level network service.

QUANTA supports TCP/IP and UDP/IP over AAL5/ATM, and AAL5/ATM directly. The communication protocols are optimised in order to apply scheduling algorithms on the delivery of packets, according to the class of service of the application.

Feedback is received from the network by a resource manager, which uses this information for performing adaptation at network and application levels. The resource manager is a global component that maintains information about all the resources in the system. Whenever necessary, the resource manager modifies the parameters of the packet scheduling algorithm in order to fulfil the QoS requirements of the application. Ultimately, it is capable of degrading the performance of applications by changing their region of operation.

QUANTA was implemented on an infrastructure consisting of Sun workstations with ATM cards connected to two Fore Systems ASX-100 switches, which provide a maximum bandwidth of 100 Mbps. The necessary alterations at protocol level are made possible by using the Streams package provided by Sun Microsystems.

Despite providing support for QoS enforcement only at network level, QUANTA offers a solid platform for the development of applications with QoS requirements situated in different application areas and running on diverse network and reservation protocols. Nevertheless, its support for adaptation is limited, and the architecture cannot be extended by the application programmer in order to use different resource reservation protocols.

## 3.11 ERDoS (SRI International)

ERDoS (End-to-End Resource Management of Distributed Systems) [26][27][143] is a middleware infrastructure capable of providing QoS support for applications with soft QoS requirements. By using the support provided by ERDoS, applications with soft QoS requirements such as multimedia applications are able to run on top of non real-time systems. This is achieved by using QoS adaptation techniques that minimise the adverse effects of the non real-time behaviour of the system on the application.

ERDoS is a three-layer architecture that separates applications from the computing, storage and communication resources that constitute the system. At the higher layer, application agents translate application-specific QoS requirements into end-to-end QoS metrics understood by ERDoS. The middle layer is composed by the system manager, which houses the adaptive resource management algorithms. These algorithms, based on the global view of the system and on the requirements of the applications, decide which applications to degrade in case of resource shortage. At the bottom layer, resource agents monitor the behaviour of the resources present in the system and notify the system manager when the QoS of an application is missed. Whenever the occurrence of resource shortage is detected, the system manager selects one or more applications that are requested to adapt their requirements, allowing other applications to meet their QoS requirements.

ERDoS provides a development toolkit together with an instantiation environment and a run-time support. This infrastructure helps the programmer to write new applications on top of ERDoS and to instantiate applications stored in an application repository. Resource agents have been implemented on Solaris, QNX and Real-Time Mach.

ERDoS distinguishes itself from the other QoS architectures presented before in this chapter by the use of purely adaptive techniques instead of relying on QoS enforcement mechanisms. Purely adaptive techniques provide only best-effort QoS guarantees, since they do not use admission control policies. Consequently, ERDoS is unsuitable for applications with more severe QoS constraints, such as hard real-time applications and some commercial multimedia applications (e.g. pay-per-view). ERDoS also has better support for open systems than the architectures described previously in this chapter due to the use of multiple protocols and due to the capacity of extending its QoS specification and adaptation mechanisms. However, it is unclear if the application programmer can write new application and resource agents in order to extend the architecture or if only the developers are able to do it.

## 3.12 MMN Project (BT-URI)

The Management of Multiservice Networks (MMN) project [147] is a British Telecom University Research Initiative (BT-URI) concerned with the provision of support for QoS management within distributed multimedia middleware environments using integrated services networks (i.e. ATM). The project comprises a five-year plan for six research institutions in the United Kingdom (University College London, University of Cambridge, Imperial College, University of Lancaster, Loughborough University and Oxford Brookes University).

Applications with QoS requirements are supported by MMN through the adoption of a CORBA-based object model together with mechanisms for the streaming of continuous media. Extensions to CORBA are proposed in the form of an enhanced object model [37]. Streams are implemented as binding objects (inspired on the binding objects used by the Xbind architecture) and defined by using augmented interface definitions.

At the highest level of abstraction, a Component Definition Language (CDL) is used by MMN to define the overall composition and configuration of objects within the distributed environment. CDL encapsulates the IDL interfaces and incorporates the features introduced by MMN, such as QoS specifications and streams.

At the lower level, mechanisms for resource management are provided by the Distributed Resource Management Architecture (DRMA) [150]. DRMA uses resource capacity regions to describe the resources that have to be allocated by the architecture in order to guarantee the QoS requirements imposed by applications. In order to control the use of these resources, DRMA provides a set of resource adaptation mechanisms. These mechanisms are responsible for monitoring the use of the resources provided by both the network and the operating system, and for initiating resource adaptation whenever the boundary of a resource capacity region is reached.

A framework for building communication protocols from reusable components is also part of the MMN project. Components may be primitive or constructed from other components. The flexible structure of the communication system allows protocol layers to be added to or removed dynamically from the protocol stack during the lifetime of a communication session in order to react to changing levels of QoS.

A set of graphical tools, textual notations and design methods for configuring systems is also proposed for the management of the network services, including QoS mechanisms. Configuration management tools provided by the framework aim to simplify the specification of distributed services, and are able to instantiate components, bind interfaces, and allocate components to nodes. Dynamic configuration allows protocol developers to redefine the component structure of the communication system and to recover from component failures. In addition, developers can group objects in domains and divide configuration management responsibility.

The different experiments that make up the MMN project are being performed using a number of different distributed programming platforms, including Darwin/Regis, AnsaWare, Xerox Parc ILU and Orbix. The final objective of MMN is to fully integrate all the proposed features into a CORBA-compatible architecture suitable for multimedia applications.

MMN is an ongoing project, and most of its results still remain to be validated. Nevertheless, the design principles adopted in the MMN project make us believe that, despite providing support for heterogeneity through the configurable and extensible component-based communication system, its applicability is constrained to a limited area of application. This occurs due to the adoption of resource adaptation techniques in detriment of QoS enforcement mechanisms together with QoS specification mechanisms and an object model that is suitable only for applications with soft QoS requirements.

## 3.13 Analysis of the State of the Art Technology

Our study of the state-of-the-art revealed that several partial solutions for the provision of services at the middleware level for applications with QoS constraints are available in the literature. However, a generic middleware for QoS-aware applications in open systems still remains to be proposed. Table 5 shows the features desired to be present in QoS architectures in four different areas: QoS specification, QoS enforcement, support for heterogeneity and support for QoS adaptation. These features are then compared with the ones provided by state-of-the-art QoS architectures and frameworks.

We have identified middleware that efficiently targets limited application areas such as distributed multimedia (e.g. Xbind) and real-time processing (e.g. TAO), and are typically concerned only with constraints found at either network (e.g. DIMMA and QUANTA) or operating system levels (e.g. SUMO and Arcade). In addition, most of the work in this area of research limits its target platform by adopting architectural designs highly dependent on a particular platform, with the most common case being ATM networks.

| | | Xbind | ReTINA | DIMMA | SUMO | TAO | Arcade | OMEGA | QoS–A | QuO | QUANTA | ERDoS | MMN |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| QoS Specification | Multimedia Support | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | Real-Time Support | | ✓ | ✓ | ✓ | ✓ | | ✓ | | ✓ | ✓ | | |
| | Extensible | | | | | | | | | | ✓ | ✓ | |
| QoS Enforcement | Network Reservation | ✓ | ✓ | ✓ | | | | ✓ | ✓ | ✓ | ✓ | | |
| | O.S. Reservation | | | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | | |
| | Transparent | | | | | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Support for Heterogeneity | Multiple Protocols | | | | | | | | | | ✓ | ✓ | ✓ |
| | Extensible | | | | | | | | | | | ✓ | ✓ |
| Support for Adaptation | Notification | | | | | | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | Transparent | | | | | | | | ✓ | | | | ✓ |

**Table 5 – Features Provided by State-of-the-art QoS Architectures**

The lack of support for multiple reservation protocols and of support for extending the QoS mechanisms present in architectures such as QoS-A and OMEGA constrain their use in open systems. Furthermore, adaptation mechanisms can be limited (e.g. in QuO) or completely absent (e.g. in ReTINA). When adaptation mechanisms are available, they often try to replace the use of QoS enforcement mechanisms, such as in ERDoS and MMN, resulting in a support suitable only for applications with soft (i.e. best-effort) QoS requirements.

The features necessary for providing support for diverse applications with QoS requirements in open systems were taken into account during the development of a new QoS architecture, called Quartz, which is presented in the next chapter of this thesis.

# 4

# Design of the Quartz QoS Architecture

This chapter introduces Quartz, a generic QoS architecture designed specifically to address the requirements of supporting diverse applications with QoS constraints in open systems. We begin by describing the main requirements imposed on the architecture. We then describe in detail the architectural design adopted to address these requirements. The approach adopted in the design of Quartz results in a very flexible and extensible architecture, capable of supporting multiple areas of application and different underlying systems as required to provide QoS-constrained services in an open, distributed and heterogeneous environment.

## 4.1 Requirements

A complex combination of requirements has to be addressed by a QoS architecture in order to provide support for applications that require QoS-constrained services in an open environment. The fulfilment of these requirements represents an important challenge in the development of QoS architectures.

### 4.1.1 Requirements on the Specification of QoS

As discussed in section 2.5.2, a QoS architecture must allow high expressive power in the specification of QoS requirements. QoS requirements should be specified according to the notion of QoS understood by the user at his level of abstraction. Any limitation constraining the expressiveness of the user would force him to deal with a notion of QoS with which he is not familiar, and would reduce the applicability of the architecture.

Furthermore, since different notions of QoS might be present at the application level because of the multiplicity of application areas that are targeted by the architecture, Quartz must be able to interpret a potentially infinite set of QoS parameters. This might be achieved by adopting mechanisms that would allow the architecture to be extended in order to understand new QoS parameters.

The parameters specified by the user must be interpreted accordingly by the architecture in order to perform the reservation of resources at the lower level of abstraction. This implies translating the parameters from their original format into parameters that are understood internally by the architecture. The architecture has also to map these parameters into resources available at the lower level and then interact with a suitable reservation protocol in order to allocate the corresponding resources for use by the application. This mapping process might not always be one-to-one (i.e. one QoS parameter resulting in one kind resource to be reserved) but can be one-to-many, many-to-one or many-to-many (Table 2 in section 2.5.3 shows examples of parameter mappings). This implies that resources might be interchangeable, and that finding an appropriate balance between requirements and resources is another task that has to be performed by the architecture.

These requirements imply that a sophisticated mechanism for specification of QoS parameters and interpretation of this information must be adopted by the Quartz architecture in order to fulfil the QoS requirements imposed by applications.

### 4.1.2 Requirements on the Enforcement of QoS

After feeding the QoS architecture with information to identify itself together with its QoS requirements, applications are not obliged to use directly any QoS mechanism available at the lower-level. Applications can rely on the QoS architecture, which will enforce the required QoS by interpreting the QoS requirements specified by the application and then use the reservation protocols provided by the network and the operating system in order to reserve resources.

In addition, the QoS architecture must provide transparency of reservation mechanisms from the application's point of view. Transparency is necessary in order to reduce the amount of work needed for applications to impose requirements on services that they avail from the underlying system.

Adopting this strategy for QoS enforcement, we hide from the application the differences between the way different lower-level systems allow resources to be reserved in order to fulfil QoS requirements. This has the important effect of increasing the portability of applications across different platforms.

### 4.1.3 Support for Heterogeneity

The QoS architecture is expected to provide features that make it appropriate to be used in open, distributed and heterogeneous environments. This implies that the architecture is required to be able to use the different protocols and hardware that coexist in an open environment.

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. This should be done by the application by adapting itself internally instead of requiring a different implementation of the architecture to be linked to the application code. In order to achieve this goal, the Quartz architecture is not only required to be portable to different platforms, but it has also to able to handle QoS for an application when the underlying reservation system changes (for example, in case of hardware reconfiguration or failure) without requiring a recompilation. This level of flexibility might be achieved by using 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, by concentrating the task of dealing with a reservation protocol within a single component, support for new reservation protocols can be added to the architecture without the necessity of porting the whole infrastructure. Instead, this is achieved by providing a new component that is able to handle the new reservation protocol. This adds to the architecture the capacity of interacting with the new reservation protocol, and providing the resources made available through this protocol to applications built on top of the architecture.

### 4.1.4 Support for QoS Adaptation

The QoS architecture is required to allow dynamic changes in the distribution of resources to be performed by the system in order to interact properly with any adaptive protocols that might be present at lower level. This must occur without causing loss of service consistency at application level.

As discussed in section 2.6.1, the reservation of resources might change because of dynamic changes in the structure of the system, resource failure or the usage of prioritised policies for the distribution of resources. Resources might also be claimed back by the system when an irregular resource usage pattern is detected by monitoring mechanisms.

Any change in the reservation of resources at lower-level must be made known to the application by using QoS parameters that are understood at its level. This implies that the QoS architecture has to perform a reverse translation of parameters and then inform the application that its QoS has changed.

Since in some cases resources are interchangeable, dynamic changes may be overcome at lower level by requesting additional resources managed by a different resource reservation protocol. In this case, all the adaptation is masked at the lower level and the QoS seen by the application remains the same.

## 4.2 The Quartz Architecture

This section describes the QoS architecture proposed in this thesis. The following sub-sections identify the main architectural components and define their role in the provision of services subject to QoS requirements. The interfaces of the components of the architecture and the messages exchanged between them are also described.

### 4.2.1 Overview of the Architecture

The Quartz QoS architecture is composed of a QoS Agent running on top of the several resource reservation protocols available in the host system. Applications use the services provided by the QoS Agent to obtain the desired level of QoS from the system.

Figure 6 situates the Quartz architecture in a computational system. At the application level, applications with QoS requirements (such as multimedia applications, real-time applications, or any other application with QoS requirements) interact with Quartz through interfaces that are specialised for the corresponding area of application.

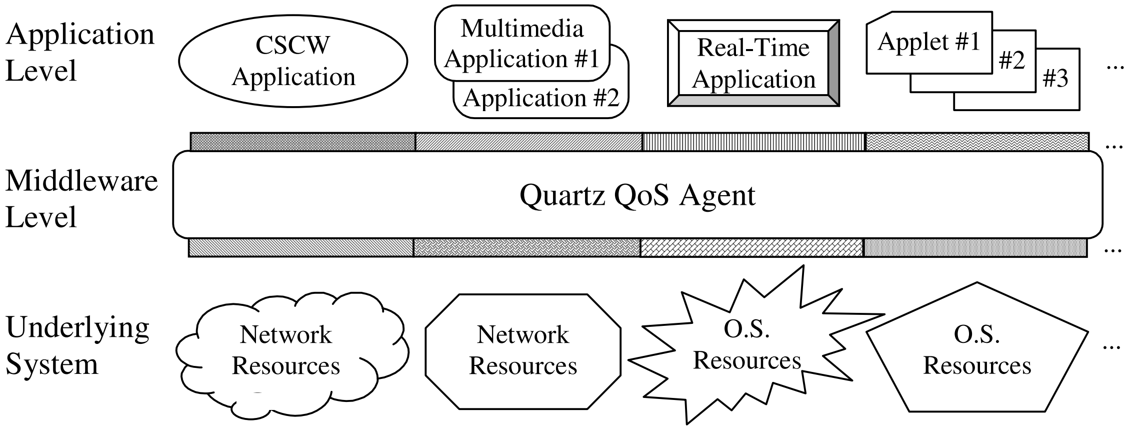


**Figure 6 – Overview of the Quartz QoS Architecture**

The *QoS Agent*, which is the major component of the Quartz architecture, receives the QoS requirements specified by applications and interacts with the underlying system in order to enforce these requirements.

The location of the QoS Agent in the distributed environment is an implementation issue. In most implementations of Quartz, the QoS Agent is likely to be located on the same machine as the application, and may be either linked to the application (i.e. one QoS Agent per application) or running as a system service (i.e. one QoS Agent per node). The mechanism used by the application to interface with the QoS Agent is also implementation-dependent, and can be one of sockets, local or remote method calls, RPC, etc. Communication between different QoS Agents is not necessary, since the tasks that require a deeper knowledge of the capabilities of the (possibly distributed) application or that require a global view of the underlying system are solved at their corresponding levels of abstraction, and are outside of the scope of the architecture. Consequently, issues such as QoS negotiation between peers (e.g. in a client-service style interaction between applications) and service degradation due to resource adaptation are handled directly by the application, while other issues such as interaction with other resource providers (e.g. along a network path) are dealt with by the reservation protocol.

The underlying system contains resources that are used by the application. It is typically composed by several sub-systems, which can be grouped in two main areas: the network infrastructure (e.g. ATM or Internet protocols) and the operating system (e.g. a real-time or desktop operating system with reservation capabilities). Networks and operating systems are seen as resource providers by the QoS Agent. The resources themselves are not represented in the figure because the manner in which they are structured varies significantly, and this would be impossible to represent. Each resource provider allows its resources to be allocated through a resource reservation protocol. The QoS Agent must know how to interact with each of the resource reservation protocols present in the system, using their interfaces in order to allocate resources on behalf of applications.

There are no severe assumptions on the behaviour of the reservation mechanisms provided by the resource providers present in the underlying system. In case a particular resource provider does not supply a resource reservation protocol but provides simple mechanisms for accessing resources, similar behaviour might be achieved by placing a resource management layer between the QoS Agent and the resource provider ([92] is an example of how this can be done). This layer would be responsible for controlling and administering the resources made available by the corresponding resource provider. In addition, it would monitor the availability of resources and issue notifications when thresholds that prevent the QoS requirements of the application to be fulfilled are reached. Similarly, any lack of functionality in a particular resource reservation protocol can be complemented or masked by a management layer placed on top of the reservation protocol.

### 4.2.2 Internal Structure

The QoS Agent is a placeholder into which other components can be plugged in order to interact with the surrounding environment. In general, the QoS Agent is responsible for giving access to the underlying QoS mechanisms necessary for providing services with the quality requested by the application. The QoS requirements imposed by the application are enforced by reserving resources provided by the underlying system. The sub-systems present in the underlying system provide the actual means of organising the access to its resources by offering a resource reservation protocol. Because of the intrinsically open nature of the target environment, several component-specific reservation protocols may be available.

The QoS Agent is responsible for two main tasks:

- Interpreting the QoS requirements specified by the application in the form of QoS parameters, translating them from the format understood by the application into a format suitable for performing resource reservations; and
- Interacting with the underlying resource reservation mechanisms in order to allocate the resources necessary to perform the service subject to QoS requirements.

Being a placeholder for other components, the QoS Agent does not perform these tasks directly. Instead, it relies on other components that are specialised for translation of QoS parameters and interaction with resource reservation protocols. Since the translation process depends directly on the application and on the resource being used, and since the interaction with the reservation protocol depends on the interface provided by this particular protocol, specialised components will be used for performing each of these tasks. These components will be plugged into the QoS Agent whenever necessary. The resulting internal structure of the QoS Agent is represented in Figure 7.

The translation of QoS parameters between the different formats used at different levels in the architecture is performed by the *translation unit*. In addition, the translation unit balances the usage of interchangeable resources provided by the different sub-systems (i.e. the network and the operating system) present in the underlying system.

The translation unit contains *QoS Filters* and a *QoS Interpreter*. QoS Filters can be subdivided into *application* and *system filters*, one for each application field and for each sub-system present at the lower-level, respectively. Filters are responsible for translating between QoS parameters understood internally by Quartz and any external format. The QoS Interpreter handles parameter formats used internally by Quartz at different abstraction levels and balances requirements between sub-systems present at the lower level.

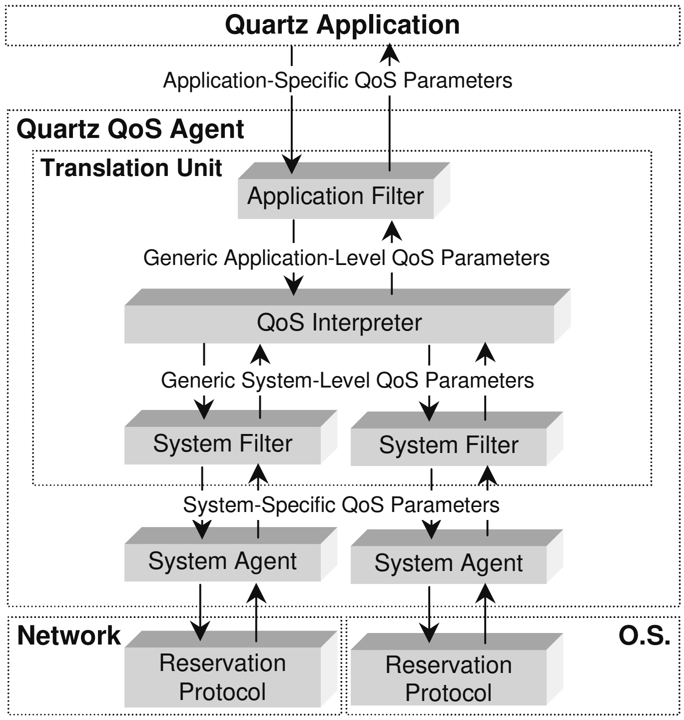


**Figure 7 – Detailed Structure of the QoS Agent**

The QoS Agent also encapsulates multiple *system agents*, which are responsible for interacting with reservation protocols administering the use of the resources provided by the underlying system. The system-specific agents get the values of QoS parameters determined by the translation unit and perform the reservation of the resources provided by the corresponding sub-system (i.e. the network or the operating system) by using the associated reservation protocol.

The following sections detail the role played by each component of the architecture and specify their external interfaces.

### 4.2.3 The Translation Unit

A translation mechanism based on the use of a set of generic parameters is adopted by Quartz to encompass the problem of specification of QoS requirements in an environment composed of multiple application areas and different reservation protocols.

In order to avoid having a translator for each combination of application field and reservation protocol, we have adopted a three-step translation process. Users 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 *system-specific QoS parameters* understood by each of the reservation protocols used by the application.

Despite the definition of two standard sets of generic parameters, the expressiveness of users is not affected because they deal only with QoS parameters understood at they level of abstraction and meaningful for his application field. This is made possible by the adoption of application-specific filters that handle QoS parameters understood at application level. The same occurs at the system level, where system-specific QoS parameters are used in order to provide a bridge between the generic parameters used internally by Quartz and the parameters understood by the reservation protocols present at the lower level.

The translation unit – and the components that form it – implement two paths of translation:

- the direct path of translation, which goes from the application to the lower layers of the architecture and that is used when the application requests the provision of QoS-constrained services; and
- the reverse path of translation, which goes from the lower layers towards the application and is used to inform the application of changes in QoS caused by dynamic adaptation of the resources allocated by the underlying system.

The translation paths are illustrated in Figure 8.

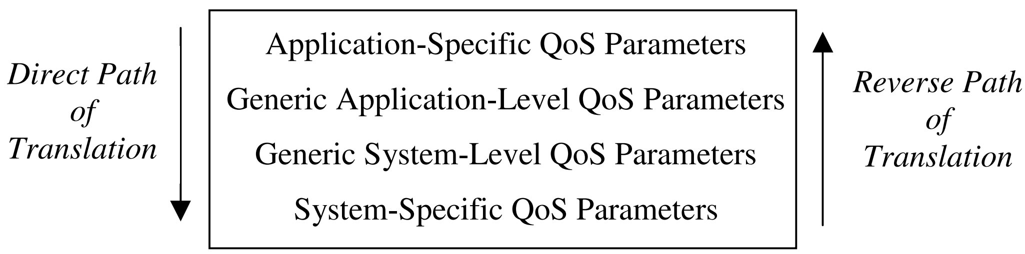


**Figure 8 – Translation Paths**

The components that form the translation unit – i.e. application and system filters and the QoS interpreter – accept two types of message through their public interfaces, illustrated by Figure 9. The first one, `TranslateQoSDownwards`, results in upper-level parameters being translated into lower-level ones. This message is issued by a component situated a level of abstraction above the component receiving the message, following through the direct path of translation. The second type of message, `TranslateQoSUpwards`, implements the reverse path of translation and is used when Quartz needs to inform the application that the QoS parameters have changed due to resource adaptation. This message is issued by a component located a layer below the component that receives the message.

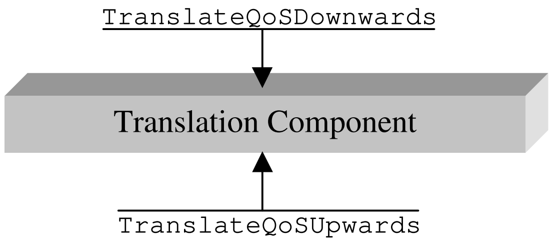


**Figure 9 – Interface of Translation Components**

### 4.2.4 Application Filters

An application filter is associated with a particular area of application. Applications that have similar notions of QoS are expected to employ the same application filter, specifying QoS requirements by using the same set of application-level QoS parameters. On the other hand, applications that have different ways of expressing QoS requirements use different sets of application-level QoS parameters, and consequently require different application filters.

Application filters obtain QoS parameters understood at application-level for the particular area of application, interpret them, and determine the values of a pre-defined set of generic application-level QoS parameters, which are used internally by Quartz. The reverse translation process – i.e. from generic application-level QoS parameters into application-specific QoS parameters – is also implemented by application filters.

For the application filter, receiving a message of type `TranslateQoSDownwards` results in translating the application-specific QoS parameters carried by the message into generic application-level QoS parameters. The second type of message, `TranslateQoSUpwards`, performs the reverse process of translation.

### 4.2.5 The QoS Interpreter

The translation between parameters related to different levels of abstraction is executed by the QoS Interpreter provided by the architecture. By using a fixed (but still user-replaceable) component for the main translation step, we simplify the work done by application and system filters, making it easier for the application programmer to write replacement filters whenever they become necessary.

The QoS interpreter also balances parameters between the sub-systems present at the lower level, i.e. the network and the operating system. This is performed by a *balancing unit* encapsulated within the QoS interpreter. The balancing process is necessary because resources present at lower-level may be interchangeable, and a trade-off has to be reached between using more or less network resources in comparison to operating system resources. Resource balancing at this level is made possible because of the definition of generic parameters that are used by Quartz internally. The trade-off between requirements that are interchangeable for a particular application (e.g. the decision of performing data compression and then require less bandwidth and more processing power) has to be handled directly by the application. Alternatively, this decision can be made by the application filter. Similarly, the trade-off between interchangeable O.S. resources and between interchangeable network resources has to be decided by either the resource reservation protocol or by the corresponding system filter, since only them have the knowledge necessary to perform balancing at this level.

The interface of the QoS interpreter is the same as the other translation components. Whenever it receives a `TranslateQoSDownwards` message, the QoS interpreter translates generic application-level QoS parameters into generic system-level QoS parameters, balancing them between the network and the operating system. On the other hand, a `TranslateQoSUpwards` message results in generic system-level parameters being translated into generic application-level parameters. The QoS parameters carried by the `TranslateQoSUpwards` message do not necessarily reach the application because transparent QoS adaptation can be performed by Quartz, compensating the resources lost during adaptation with resources obtained from another resource provider. The process of QoS adaptation is described in detail in item 4.4.1.

### 4.2.6 System Filters

System filters are associated with sub-systems present in the underlying system. These filters can be subdivided into network filters and operating system filters, according to the sub-system with which they interact. Since sub-systems that can be present in the underlying system use different forms of representation for QoS parameters, a different system filter is provided in order to interpret these parameters.

In order to obtain the value of the system-level QoS parameters understood by the underlying reservation mechanisms, system filters translate the generic system-level parameters used internally by Quartz. In case of a dynamic change of system-level QoS parameters caused by resource adaptation, these parameters are translated upwards by the system filter.

Direct and reverse translation are performed by system filters upon receipt of the `TranslateQoSDownwards` and `TranslateQoSUpwards` messages respectively.

### 4.2.7 System Agents

Quartz is able to handle the differences between multiple reservation protocols by using system agents.

Different resource reservation protocols provide distinct interfaces for applications using them. Consequently, applications targeting heterogeneous systems have to be able to handle these different interfaces in order to reserve the resources provided by the system. In Quartz, each different reservation protocol is associated with a corresponding system agent, which hides differences among protocols by providing a common interface to the rest of the architecture.

A system agent allocates the resources necessary for the application to fulfil its QoS requirements by using the resource reservation protocol associated with it. The QoS requirements specified by the application in the form of QoS parameters meaningful at application level are interpreted by the translation unit and passed on to the system agents corresponding to the reservation protocols present in the underlying system. The system agent then uses the interface provided by the associated resource reservation protocol in order to allocate the resources that are necessary to fulfil the requirements specified by the application.

In addition to enforcing QoS requirements by means of resource reservation, system agents receive notifications from the lower-level system whenever a resource fails or is claimed back by the system. This notification is carried towards the upper layers of the architecture, passing by the reverse translation path provided by the translation unit until potentially reaching the application.

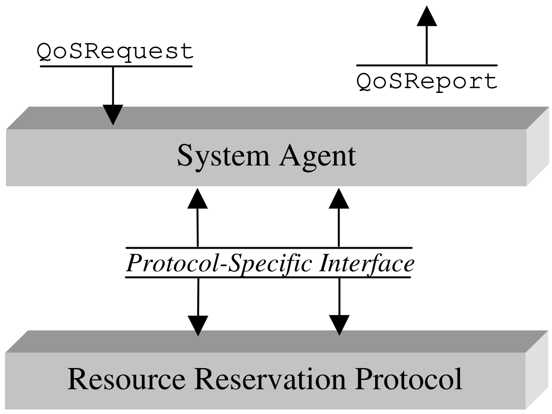


**Figure 10 – Interface of Component Agents**

The interface of the system agent is illustrated in Figure 10. Interfacing with system agents takes place via two different types of message: `QoSRequest` and `QoSReport`. The first message is sent to the system agent in order to request the provision of QoS-constrained services, and carries the QoS parameters corresponding to the requirements imposed by the application translated into the parameter format understood by the system agent. The second type of message is produced by the system agent in order to indicate that QoS has changed at lower-level, and carries the new level of QoS supported by the system after the changes.

### 4.2.8 The User Interface

The QoS Agent provides a very simple interface to the QoS Application. The messages exchanged between the QoS Agent and the application and the interface between the QoS Agent and the reservation protocols provided by the underlying system are represented in Figure 11.

To request a service with a specific level of QoS to be provided by the environment, the application issues a `QoSRequest` message. This message carries a list of variables identifying the QoS parameters and setting their respective values.

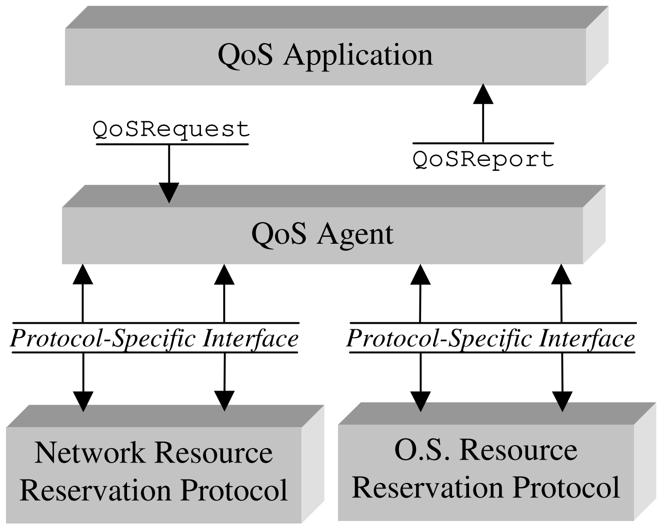


**Figure 11 – User Interface of the QoS Agent**

The application is notified about changes in the QoS provided by the system through `QoSReport` messages. This message is originated by the system agent when it receives a QoS notification from the reservation protocol through the protocol-specific interface, informing that resource adaptation has occurred. The QoS parameters resulting from the adaptation are translated upwards and delivered in the form of a `QoSReport` message to the application. The application is then responsible for adapting the service to the new QoS provided by the system, allowing a graceful degradation of the service provided to the user.

It is important to notice that the same format of messages is accepted by the individual system agents. The main difference is that these messages carry parameters comprehensible at application level, obtained directly from the application when QoS is requested or after the translation process performed by the translation unit when a dynamic adaptation occurs. Despite the differences in parameter formats, the user can interact directly with an individual system agent by using the same message syntax.

## 4.3 QoS Parameters

As described before, the Quartz QoS Architecture deals with four distinct classes of QoS parameters depending on the level abstraction. Two of them are generic classes and are standardised by the architecture: the generic application-level QoS parameters and the generic system-level QoS parameters. The other two classes of parameters are defined according to the application area or the resource reservation protocol.

### 4.3.1 Parameter Formats

Parameters are described as <name,value> pairs. To allow the easy identification of the corresponding abstraction level and consequently identify the translation process to which the parameter has to be submitted, we have adopted a scoped namespace. Parameters are scoped with the name of the abstraction level in which they are meaningful.

Generic QoS parameters are scoped with `App::` and `Sys::` for application- and system-level parameters respectively. System-level parameters are further subdivided into two sets: a set of QoS parameters scoped with `Net::` related to the network, and another scoped with `OS::` for parameters related to the resources which are provided by the operating system. This is necessary to identify when the network and the operating system share the responsibility for the fulfilment of a requirement (parameters scoped with `Sys::`) or is related to only one of them (and scoped with `Net::` or `OS::`). For example, a certain delay seen at application level is usually a global value which results from the composition of two partial delays: one derived from the network and the other from the operating system. A bandwidth requirement is applicable only to the network, while a requirement such as the class of service guarantee provided by the system (e.g. best-effort, unloaded or deterministic) is valid for both the network and the O.S..

A similar strategy is adopted for application- and system-specific parameters. For example, parameters for an MPEG rendering application could be scoped with `Mpeg::`, subsequently translated into generic application parameters scoped with `App::`, then into system parameters scoped with `Sys::`, and finally into system-specific parameters scoped with `Rsvp::` (for resource reservation protocol) and `Posix::` (for POSIX-compliant operating systems) for example.

In general, users specify parameters at their own abstraction level. In cases where they want to specify the QoS to be provided by the system at a lower level, parameters can be specified using the scoped name corresponding to the appropriate abstraction level. These parameters will be ignored by the first translation steps and then caught by the corresponding translation component.

When a parameter is described without a scope, the architecture assumes that it corresponds to the current abstraction level.

In addition to scopes, parameter names can also be suffixed with the modifiers `Min` and `Max` for specifying limit values.

### 4.3.2 Application-Specific QoS Parameters

These parameters are specific to the application area in which Quartz is being used. For example, Quartz can be used in a multimedia environment in which QoS parameters such as 'video frame rate' and 'resolution' are adequate and require an application filter able to understand them and translate them into the format used internally by Quartz (i.e. generic application-level QoS parameters).

However, in a different application scenario, such as in real-time systems, an application would be more likely to express its QoS requirements in the form of deadlines for the processing and for the transmission of data. In this case, a different application filter, which understands different application parameters and translates them into generic application-level QoS parameters, is adopted by Quartz.

The different sets of application-specific QoS parameters that are used by the application examples built on top of Quartz will be described in section 7.1.

### 4.3.3 Generic Application-Level QoS Parameters

Table 6 describes the generic application-level QoS parameters defined by the QoS Architecture and recognised by the translation unit of the QoS Agent.

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

**Table 6 – Generic Application-Level QoS Parameters**

Generic application-level QoS parameters represent the overall QoS requirements of the application as they are seen at application level. At this level there is no distinction between the sub-systems that are providing resources to the application.

The set of application-level QoS parameters has been chosen by observing the notion of QoS present at the application level in different areas of application and then finding a common way of representing QoS requirements at this level. As a result, we have obtained a small set of QoS parameters that can be easily mapped into the different forms of specifying QoS that are used in different areas of application.

### 4.3.4 Generic System-Level QoS Parameters

System-level parameters are subdivided into two groups: network and operating system parameters. Table 7 lists and describes these parameters.

| **Parameter Name** | **Modifiers** | **Description** |
|---|---|---|
| `Net::Bandwidth` | `Min,Max` | Bandwidth provided by the network (in bytes/s) |
| `Net::PacketSize` | `Min,Max` | Size of data packets (in bytes) |
| `Net::Delay` and `OS::Delay` | `Min,Max` | Transmission and processing delays (in seconds) |
| `Net::ErrorRatio` | `Max` | Acceptable transmission error (bits per million) |
| `Sys::Guarantee` | – | Levels of service guarantee provided (best-effort, unloaded, deterministic, etc.) |
| `Net::Cost` and `OS::Cost` | `Max` | Financial cost (currency per second) |
| `Net::SecurityLevel` | – | Security mechanism (none, encrypted, etc.) |

**Table 7 – Generic System-Level QoS Parameters**

At the system level, the division of requirements between sub-systems (i.e. the network and the operating system) becomes clearly represented by the definition of network and operating system parameters, reflecting what is present in the underlying system. The set of generic system-level parameters has been defined based on the parameters used by resource reservation protocols present at lower level. The generic system-level parameters are easily mapped into the QoS parameters used by a large number of reservation protocols that may be present at network and operating system levels in an open, heterogeneous environment.

### 4.3.5 System-Specific QoS Parameters

Different resource reservation protocols accept reservations based on different formats of QoS parameters. For example, despite both being network reservation protocols with similar capabilities, RSVP uses token bucket specifications compared to the bandwidth-driven approach used by ATM. A similar situation occurs with reservation protocols provided by operating systems. Different scheduling algorithms and resource allocation schemes result in different ways to express QoS requirements at this level.

Due to the different ways in which resource reservation protocols are used to express QoS requirements, different formats of QoS parameters have to be used at the lower layers of the architecture. In order to accommodate these differences, Quartz employs multiple sets of QoS parameters, which are specific for a particular reservation protocol. These parameters have to be derived from the generic system-level QoS parameters generated by the QoS interpreter. System filters are responsible for this translation step.

The different sets of system-specific QoS parameters defined by us to be used with the resource reservation protocols that are supported by the prototype of Quartz will be described in Chapter 5.

### 4.3.6 Translation Rules Between Generic Parameters

There is a clear mapping between the two generic sets of QoS parameters defined previously. Table 8 describes the relationship between them. The mapping between generic parameters also takes into account the limit values (i.e. maximum and minimum values) specified by the application.

| At Application Level | | At System Level |
|---|---|---|
| `App::DataUnitSize * App::DataUnitRate` | ⇒ | `Net::Bandwidth` |
| `App::DataUnitSize` | ⇒ | `Net::PacketSize` |
| `App::EndToEndDelay` | ⇒ | `Net::Delay + OS::Delay` |
| `App::ErrorRatio` | ⇒ | `Net::ErrorRatio` |
| `App::Guarantee` | ⇒ | `Sys::Guarantee` |
| `App::Cost` | ⇒ | `Net::Cost + OS::Cost` |
| `App::SecurityLevel` | ⇒ | `Net::SecurityLevel` |

**Table 8 – Relationship between Generic Parameters**

The division of cost and delay between the network and the operating system is calculated by the balancing agent encapsulated by the QoS interpreter. This calculation is based on data structures kept by the balancing agent describing the state of the network and the operating system in terms of load, cost, and any other data useful to obtain a balance in the use of resources from the network and the operating system. By analysing this information, the QoS interpreter obtains the split factors that are used for dividing QoS requirements between the network and the operating system.

## 4.4 Additional Features

The following items detail two further important characteristics of the Quartz architecture: its support for dynamic resource adaptation and rebalancing of resources; and the capacity of the architecture to be reconfigured in order to support different application areas and resource reservation protocols.

Dynamic resource adaptation is made possible by the monitoring of QoS notifications sent by the resource reservation protocols and by translating QoS parameters from the form used by each of the resource reservation protocols into the high-level format understood by the application.

The reconfiguration of the architecture can take place by allowing the application programmer to select the appropriate components from a library or, in case they are not available, by allowing him to write new components that can be plugged into the QoS Agent.

### 4.4.1 Dynamic Resource Adaptation and QoS Adaptation

A common feature generally neglected by available QoS architectures is the provision of support for dynamic resource adaptation.

Resource adaptation occurring at lower levels results in adjustments in the QoS provided by the system to the application. Consequently, in addition to performing reservations, a QoS architecture has to catch notifications issued by the resource reservation protocol informing it of dynamic resource adaptation, and then proceed with QoS adaptation at a higher level.

The QoS Agent catches notifications issued by resource reservation protocols through the corresponding system-specific agent and performs QoS adaptation when needed. QoS adaptation can be performed by Quartz at both system and application level.

In some cases, resource adaptation can be accommodated by performing QoS adaptation at system level, without interfering with the QoS seen by the application. In the Quartz architecture, some QoS requirements such as cost and delay are fulfilled by the sum of resources provided by both the operating system and the network. The component responsible for dividing QoS requirements between sub-systems is the balancing agent, which is basically a resource trader that is encapsulated by the interpreter.

When one of the operating system or the network reduces the resources allocated for the application, the balancing agent tries to execute a process called 'rebalancing of resources'. This process tries to compensate for the loss of resources from one side by requesting more resources from the other. When this is possible, the resource adaptation is compensated by rebalancing generic system-level parameters, and the QoS seen by the application is not affected. In this case, the adaptation is completely transparent from the application's point of view. However, rebalancing of resources is possible only with interchangeable resources, and it might fail in case additional interchangeable resources are not available and cannot be allocated to compensate the loss of resources that occurred due to adaptation.

If QoS adaptation at system level does not succeed, Quartz must start QoS adaptation at application level. In this case, the QoS Agent tells the application to adapt its requirements in order to decrease the consumption of resources. It sends a message to the application indicating that its QoS requirements will not be fulfilled because the amount of resources originally allocated for it has been reduced. Consequently, it is necessary for the application to adapt its QoS requirements in order to adjust its behaviour to the resources available in the computing platform. This can be done by reducing the quality of a video stream or by changing the compression method used for data transfer for example. Alternatively, the degrading strategy can be implemented within the application filter, which then becomes responsible to decide how to degrade quality on behalf of the application.

When the need to perform QoS adaptation at application level is reported to the application, the QoS Agent must indicate the new QoS provided to the application after the dynamic change in the allocation of resources. Since the application is unaware of low-level details regarding the allocation of resources, the new QoS must be expressed in the form of application-level QoS parameters. This is made possible by using the reverse translation path provided by the translation unit. In this case, a set of system-specific QoS parameters is translated into application-specific QoS parameters.

Figure 12 illustrates how QoS adaptation occurs at both system and application level. In this example, a user is in his office and has his laptop connected to the local area network. The user is taking part in a collaborative work session, and requests the application to establish a connection with a maximum delay of 70 milliseconds between partners. This is achieved by the application by obtaining a 40 ms delay from the network and allocating the remaining 30 ms for the operating system to perform data processing. When the user leaves his office and prepares to leave the building, his machine connects to the infrared network. However, this medium is unable to guarantee the same delay, which has to be raised to 50 ms.

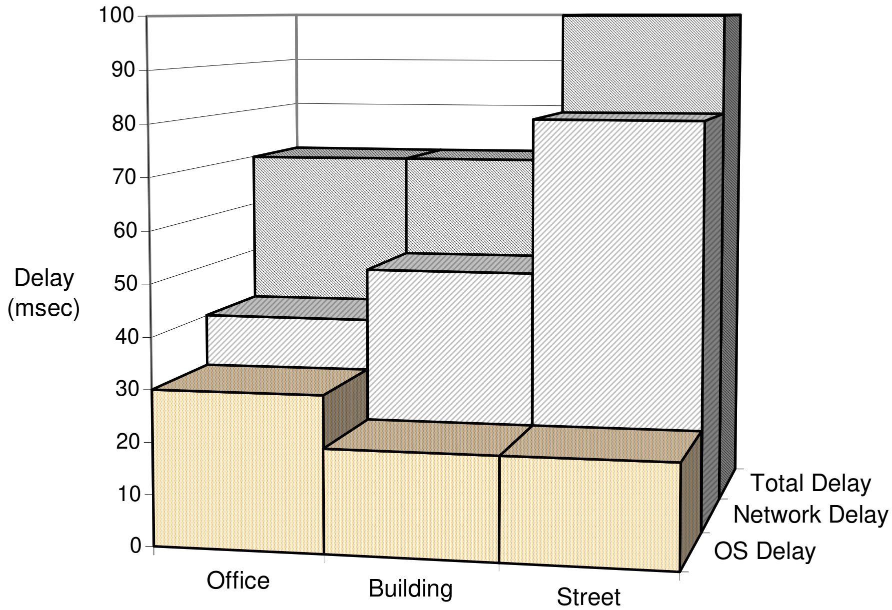


**Figure 12 – Example of QoS Adaptation**

Quartz is able to compensate for this loss by raising the usage of operating system resources, resulting in an O.S. delay of 20 ms. Consequently, the global delay is maintained and the user doesn't notice any change in QoS. When the user leaves the building, the computer starts using the cellular network. In this case the network delay rises to a value (80 ms) that can no longer be compensated by the operating system, and then Quartz asks the user to adapt its requirements in order to accommodate a higher delay.

### 4.4.2 Extending the Architecture

The structure of the QoS Agent is highly portable, reusable and extensible, because the particular needs of application fields and resource reservation protocols are encapsulated by application filters, and system filters and agents respectively. Changes at application or system level imply the replacement of filters and system-specific agents, or the plugging of new ones into the QoS Agent. Since these components have a standardised interface, they can be replaced in order to reflect changes in the environment without requiring any modification of the QoS Agent's code or in the application.

The component-based internal structure of the QoS Agent allows the architecture to be easily extended by the application programmer. This can be done by writing new components that are plugged into the QoS Agent when needed. Predefined and user-defined components can be stored in a library and then used when the corresponding resource reservation protocol is in use, in case the component in question is a system filter or agent, or when the application can be classified into the application area corresponding to the application filter. The components can be combined freely, since there is no interdependence between them.

The addition of support for a new resource reservation protocol requires that a new system agent and a new system filter be written by the programmer. The system agent is responsible for the interaction with the new reservation protocol, while the system filter is responsible for translating QoS parameters between the set of QoS parameters understood by this new protocol and the generic parameters used internally by Quartz. Adding support for a different area of application requires only that the component responsible for interpreting QoS parameters meaningful in this context (i.e. the corresponding application filter) be provided by the application programmer.

Filters are simple and easy to implement because they deal with parameters described at the same level of abstraction, i.e., application filters translate user parameters specified at application level into generic application parameters, while system filters translate generic system parameters into system specific ones. The most complex translation step, which involves mapping between different abstraction levels and balancing requirements between components, is executed by the QoS interpreter provided by Quartz. Since system agents are restricted to dealing with a single reservation protocol, the knowledge necessary to implement them is limited.

## 4.5 Summary

This chapter has described in detail the main contribution provided by this thesis, which is the Quartz QoS Architecture.

After listing the requirements imposed on the architecture, we have introduced the component-based structure of the architecture. We have established the role of each component in the overall task of providing mechanisms for QoS specification and enforcement in open environments transparently from the resource reservation protocols present in the computing platform. In addition, we have specified the interfaces of the internal components of the Quartz architecture and the messages used by these components to interact with their peers.

We have also described the sets of generic QoS parameters that are used internally by Quartz to represent the application- and system-specific QoS parameters.

Furthermore, two important characteristics of Quartz have been described: its capacity to handle resource adaptation and its potential to be reconfigured by replacing components in order to adapt to changes in the surrounding environment.

# 5

# Implementation

This chapter describes a functional prototype of the Quartz architecture that has been implemented as part of this thesis. The following sections describe the main core of the implementation (i.e. the components that are not system- or application-specific) and later introduce the components that are plugged into the core in order to interface with the resource reservation protocols supported by the target environment and to provide QoS mechanisms suitable for the application areas to be supported by Quartz.

## 5.1 The Quartz Prototype

A functional prototype of Quartz has been implemented in order to analyse its behaviour and validate the applicability of Quartz in the provision of QoS-constrained services for applications with QoS requirements. The computing platform supported by this prototype and the software model used for its implementation are described in the following sections.

### 5.1.1 Supported Platforms

The platform on which Quartz was deployed consisted of a network of PCs running the Windows NT operating system. This platform was chosen because of the fact that it is recognised as an industry *de facto* standard, in addition to its soft real-time capabilities

[96]. The real-time capabilities provided by Windows NT, despite being still limited [144], are similar to those provided by other commercial operating systems such as Sun's Solaris. We have also taken into account for choosing the operating system that resource reservation protocols for Windows NT, including preliminary versions, are easy to obtain and experiment with, in most cases at no cost.

Two different network-level resource reservation protocols are supported by the prototype: ATM and RSVP. These protocols were described in items 2.6.2 and 2.6.3 respectively. We have opted for using ATM and RSVP due to the fact that they are the two most widely used network-level resource reservation protocols.

We use the implementation of RSVP developed by Intel, called PC-RSVP, currently in beta version. For ATM, we use ForeRunner LE 155 Mbps PC cards and a Fore Systems ASX 100 switch. We also rely on the device driver and the Winsock2 service provider that are supplied by Fore Systems together with the hardware.

The programming environment used for the implementation was Microsoft Visual C++ 5.0. It was chosen because of its close integration with the computing platform in which the development took place.

### 5.1.2 Software Model

In the prototype that we have implemented, the components of the Quartz architecture and the messages exchanged between them are represented as C++ classes [46].

Each component that makes up part of the Quartz prototype is implemented as a C++ object. All components are linked with the application within the same address space. As a result, the Quartz prototype runs only at user level and does not require any change to the O.S. kernel, making the architecture more portable and flexible.

Messages are exchanged between components by means of method invocations. Consequently, each message recognised by a component corresponds to a public method that can be invoked by the other components. The contents of the message are passed as parameters to the method.

Resource reservation protocols are typically implemented as Windows NT services. The reservation mechanisms provided by the operating system are integrated with the kernel.

## 5.2 The Quartz Core

The Quartz Core consists of the components of the prototype that are independent of the underlying system and of the application area. These components are: the data structure used for specification of QoS parameters, the QoS Agent, the QoS Interpreter, and a default filter called the bypass filter. In addition, the core provides base classes for system agents, which are specialised as operating system agents and network agents.

### 5.2.1 Specification of QoS Parameters

The mechanisms for specification of QoS provided by Quartz are based on class `QzQoS`, which stores parameter names and their corresponding values. This class provides methods to set and retrieve the values of these parameters from its internal data store.

Figure 13 shows the representation of class `QzQoS` in the Unified Modelling Language (UML) [22].

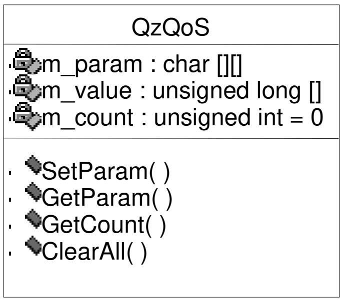


**Figure 13 – UML of Class `QzQoS`**

Class `QzQoS` stores a list of parameters, which are represented by a parameter name and the corresponding value, and a parameter counter. An unsigned long field is used for storing the value of the parameter or the address of a data structure where more complex parameter values can be stored.

Methods `SetParam()`, `GetParam()`, `GetCount()` and `ClearAll()` are provided for manipulating the list of parameters. These methods are used for setting a parameter, getting the value of a parameter, get the number of parameters stored by the class, and to clear all parameters respectively.

### 5.2.2 Basic Translation Components

The base class for translation components is called `QzTranslation`. This class defines the methods that perform translation of QoS parameters in both directions of translation. These methods are `TranslateQoSDownwards()`, which implements the direct path of translation, and `TranslateQoSUpwards()`, which implements the reverse path.

Two translation components derived from `QzTranslation` are included in the Quartz Core: the bypass filter and the QoS interpreter. The UML of the translation components is shown by Figure 14.

The bypass filter is a default QoS filter that simply forwards QoS parameters without modifying them. By using this filter as the application filter, it is possible for the application to specify QoS requirements in the form of generic system-level QoS parameters.

The QoS interpreter, implemented by class `QzInterpreter`, translates between the generic sets of QoS parameters defined by Quartz and balances requirements between the network and the operating system, performing adaptation when necessary.

The execution of balancing and adaptation is delegated by the interpreter to the balancing unit, which is implemented by class `QzBalancingUnit`. This component stores split factors that are used to divide QoS requirements between the network and the operating system. Split factors have the same values for agents running on the same machine and reflect the current capacity of the network and the operating system to provide interchangeable resources. The values of split factors are defined according to the global state of the system and are modified every time adaptation occurs.

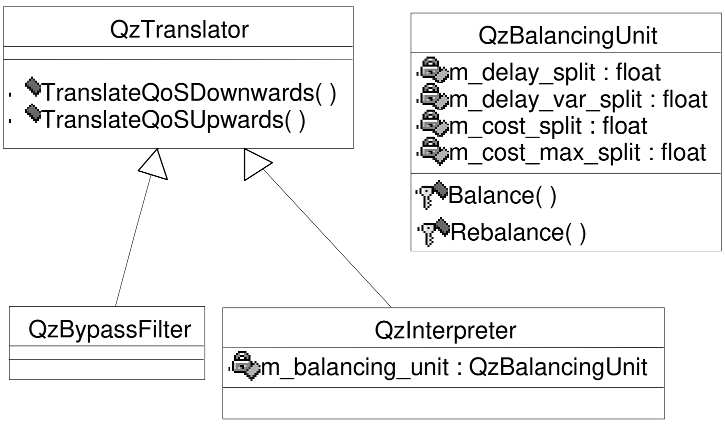


**Figure 14 – UML of Basic Translation Components**

### 5.2.3 Basic System Agents

Class `QzSystemAgent` defines the standard interface of a system agent and implements method `QoSRequest()` for specifying QoS and some additional methods needed for initialisation (method `Init()`) and for close control of the notification and adaptation process to be used by the application when necessary. These methods are `QoSNotify()` for notifying the application of resource adaptation and `QoSAdapt()` for initiating adaptation. In addition, agents contain pointers to the corresponding QoS filter, to the interface used to upcall the application and to the structure with the current QoS parameters. They have also a flag to identify the state of the QoS reservation (i.e. to indicate whether the QoS parameters are currently being enforced by the associated resource reservation protocol or not).

Class `QzSystemAgent` is specialised by `QzNetworkAgent` and `QzOSAgent`, which are the base classes for network and operating system agents respectively.

The application interfaces with only one component of the architecture: the QoS Agent, implemented by class `QzQoSAgent`, which like system agents is also derived from class `QzSystemAgent`. The `QzQoSAgent` class has a pointer to the network and operating system agents and for the interface used for upcalling the application.

Figure 15 shows the UML representation of the basic system agents, the QoS Agent and the upcall interface.

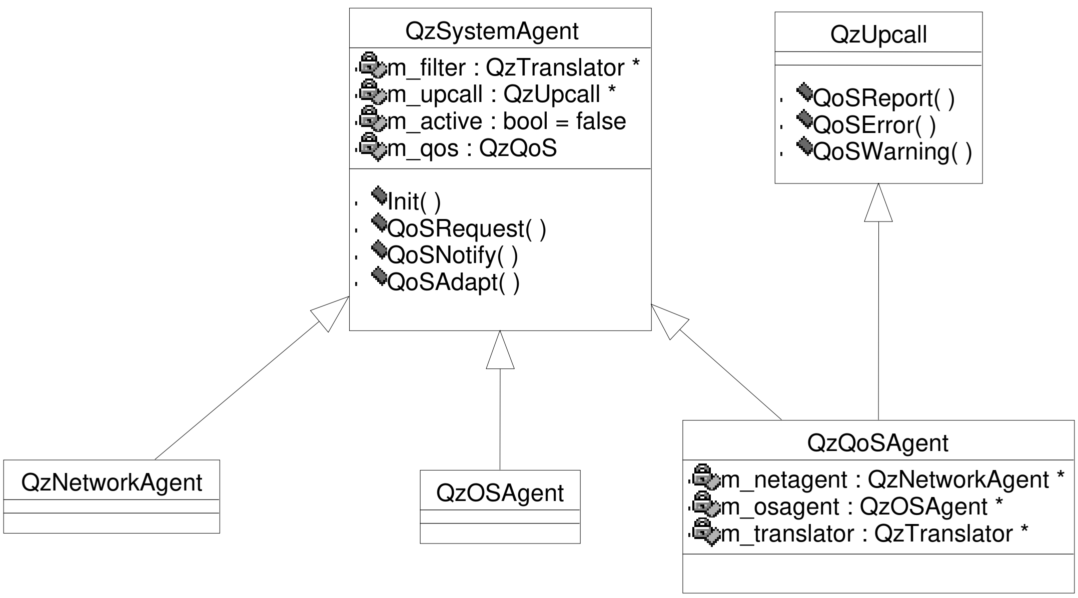


**Figure 15 – UML of Basic System Agents and Upcall Interface**

The QoS Agent is a skeleton in which the components of the Quartz core are put together with the components that depend on the application area and the underlying system. These components can be plugged into and removed from the QoS Agent in order to reflect changes in the surrounding environment. The replacement of components encapsulated by the QoS Agent does not affect the interface with the user, which 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 errors, warnings and dynamic changes of QoS parameters. These are implemented by methods `QoSReport()`, `QoSError()` and `QoSWarning()` respectively. Applications using Quartz must support this interface by inheritance or encapsulation. The QoS Agent also inherits from `QzUpcall` in order to receive notifications, errors and warnings from network and operating system agents, which are then forwarded to the application.

### 5.2.4 Interaction Between Components

The Quartz prototype is ready to be used by an application after the application initialises the QoS Agent and identifies itself. The QoS Agent is responsible for initialising the other components of the architecture.

Figure 16 illustrates the interaction between the components of the Quartz prototype when QoS is requested (action [1]) and when adaptation takes place (action [2]).

First, the application calls method `QoSRequest()` on the QoS Agent, passing its QoS requirements in the form of a `QzQoS` structure with application-level QoS parameters. Then, the QoS Agent calls the application filter and the interpreter, which translate and balance the QoS parameters, and requests QoS from both the O.S. agent and from the network agent. These agents ask their corresponding filters to translate the QoS parameters and, in the possession of system-specific QoS parameters understood by them, perform resource reservations by using their corresponding reservation protocols. With the reservations accepted by the reservation protocols, the resources become available for the application.

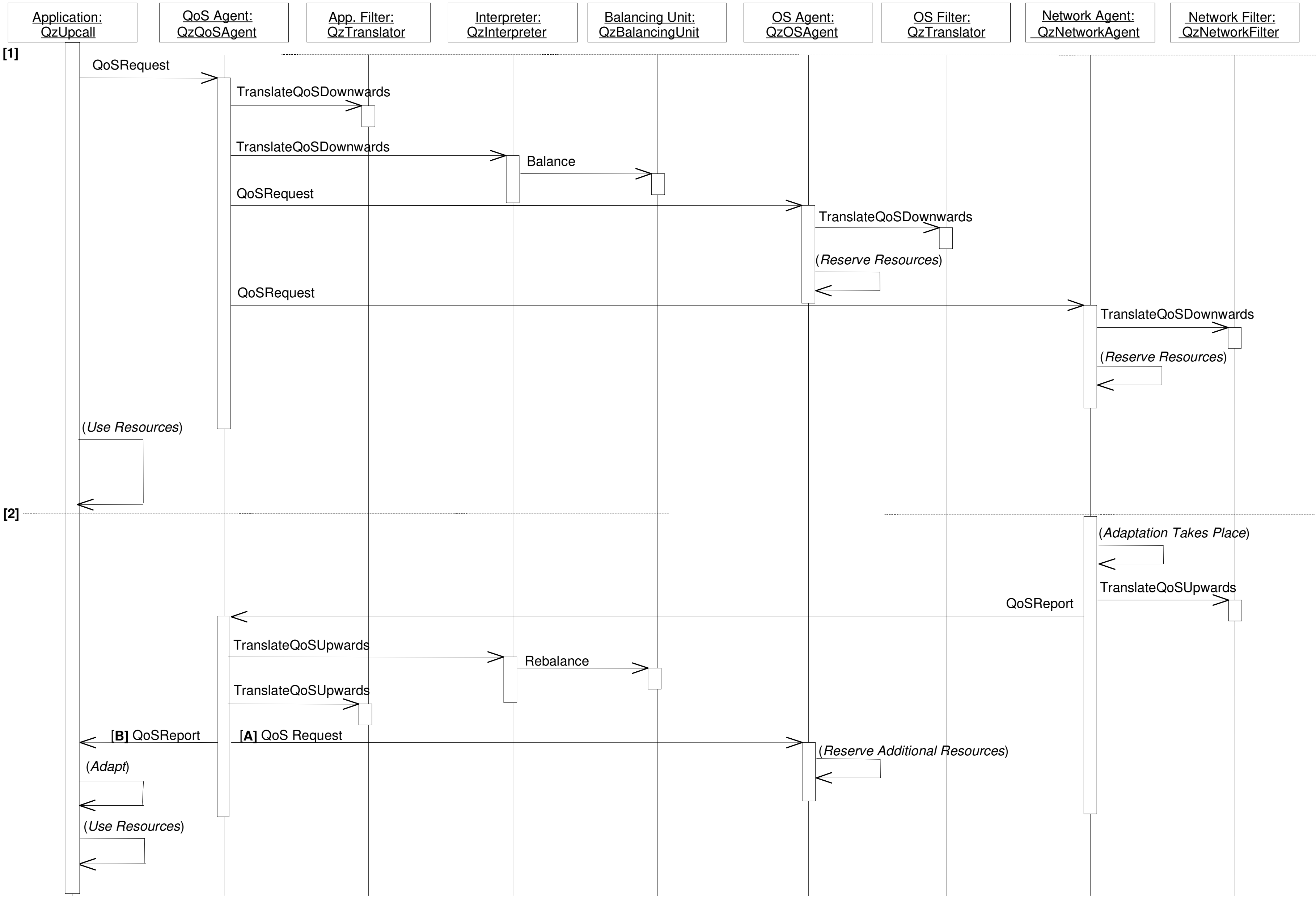


**Figure 16 – Interaction Between Components**

At some stage in the computation, the network or the O.S. might start resource adaptation (in the example, this is performed by the network). The new QoS parameters are then translated upwards by the corresponding system filter and reported to the QoS Agent in the form of a call to the `QoSReport()` method. The QoS Agent passes these parameters to the interpreter, who asks the balancing agent to try to rebalance them. If the rebalance succeeds (option [A] in the figure) the QoS Agent tries to get additional resources from the O.S.. If the rebalance fails (option [B]) the QoS Agent informs the application of the new QoS by calling `QoSReport()` on the application, which then has to adapt its requirements.

## 5.3 The RSVP Sub-System

The following sections present the system agent and the system filter for the RSVP protocol, together with the QoS parameters recognised by them.

### 5.3.1 RSVP Parameters

The QoS parameters defined for RSVP use a token bucket to model the data traffic. The parameters recognised and translated by the RSVP filter are listed in Table 9. In addition, three other parameters are also supported by RSVP. These parameters are very system-specific and do not have a form of representation at the application level. Consequently, they are not translated by the RSVP filter, but can be specified by the user to fine-tune the behaviour of the network support. These parameters are listed in Table 10.

| Parameter Name | Description |
|---|---|
| `RSVP::TokenRate` | Rate at 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) |

**Table 9 – RSVP QoS Parameters**

| Parameter Name | Description |
|---|---|
| `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 10 – Additional Parameters**

The meaning of the RSVP parameters is exactly the same as defined by the RSVP specification. Consequently, after the QoS parameters are translated into RSVP-specific parameters, the RSVP agent just has to arrange them in a suitable data structure and pass this structure to the RSVP protocol in order to perform a resource reservation.

### 5.3.2 The RSVP Filter

The system filter for RSVP is implemented by class `QzRSVPFilter`, which is responsible for translating QoS parameters understood by RSVP. It inherits from class `QzTranslation` and implements the translation methods defined by this class. The translation process occurs in two ways, downwards from the generic set of system-specific QoS parameters defined by Quartz into RSVP-specific QoS parameters, and upwards from RSVP-specific parameters into generic system-level parameters.

The translation between generic system-level parameters and RSVP parameters implies creating a token bucket in order to describe the traffic requirements specified by the application. The rules described below execute the mapping between generic system-level parameters and RSVP parameters:

- `RSVP::TokenRate` is equal to `Net::Bandwidth`.
- `RSVP::PeakRate` and `RSVP::Rate` have the same value as `Net::BandwidthMax`.
- If `Net::BandwidthMax` is not specified, it is considered to be equal to `Net::Bandwidth`.
- `RSVP::BucketSize` is equal to `Net::Bandwith` or the difference between `Net::BandwidthMax` and `Net::Bandwidth` if higher.
- The value of `RSVP::MaxPktSize` is the same as `Net::PacketSizeMax`, and `RSVP::MinPoliced` is equal to `Net::PacketSizeMin`.

- `RSVP::FlowType` is derived from `Net::Guarantee`.
- The value of `RSVP::SlackTerm` is the same as `Net::DelayVariation`.

Parameters `Net::PacketSize`, `Net::Delay`, `Net::ErrorRatio`, `Net::Cost`, and `Net::SecurityLevel` have no equivalent at the RSVP level. In order to allow the reverse translation from parameters at RSVP level into parameters at application level, these parameters are forwarded to the RSVP agent, which stores them and passes them upwards when a reverse translation occurs.

### 5.3.3 The RSVP Agent

The system agent for RSVP has been implemented as a class called `QzRSVPAgent`.

Upon initialisation, details about the link that is subject to QoS have to be specified by the application (or by middleware working on behalf of it) by calling the `Init()` method of the QoS Agent before requesting the provision of QoS-constrained service. Information such as the communication protocol (TCP or UDP), the host name or its IP address, the port, and the communication role (i.e. sender, receiver or both) is passed to the RSVP agent by using the `Init()` method. In addition, a pointer to the upcall interface that is to be called in order to report QoS adaptation, errors or warnings has to be passed to the RSVP agent upon initialisation.

After initialisation, the RSVP agent performs resource reservations upon the receipt of a `QoSRequest` message (i.e. whenever method `QoSRequest()` is called) by interacting with the interface that it shares with the RSVP protocol. In addition, the RSVP agent issues `QoSReport` messages when adaptation occurs at lower-level (i.e. calls method `QoSReport()` on the upcall interface).

## 5.4 The ATM Sub-System

A system agent and a system filter for ATM have also been implemented in order to allow Quartz to perform reservations in ATM networks.

### 5.4.1 ATM Parameters

The parameters defined for the ATM sub-system are listed in Table 11. 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 has just 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::QoSClass` | Type of data flow (CBR, VBR, best-effort, etc) |
| `ATM::MaxBurstSize` | Maximum cell burst (in ms) |
| `ATM::Tagging` | Tag non-compliant cells as subject to be discarded |

**Table 11 – ATM QoS Parameters**

It is important to notice that these parameters differ significantly from those used for the RSVP protocol, but are similar to the generic system-level QoS parameters defined by Quartz. This happens because RSVP uses a token bucket model for QoS specification, while both Quartz and ATM adopt bandwidth-oriented QoS parameters.

### 5.4.2 The ATM Filter

The ATM filter is implemented by class `QzATMFilter`, which translates ATM parameters understood by the ATM Agent. Like other filters, the ATM Filter inherits from class `QzTranslation`. The translation process occurs in two ways, downwards from the generic set of generic system-level QoS parameters defined by Quartz into ATM-specific QoS parameters, and upwards from ATM-specific parameters into generic system-level parameters.

The translation from generic system-level QoS parameters into ATM-specific QoS parameters is mainly concerned with bandwidth. Whenever the provision of QoS-constrained services is requested, the reservation of the necessary bandwidth is performed. The value of `ATM::PeakCellRate` is taken from `Net::BandwidthMax`, while the value of `ATM::SustainableCellRate` corresponds to `Net::Bandwidth`. The QoS class is defined based on the characteristics of the bandwidth, i.e. if it is constant or variable. `ATM::MaxBurstSize` and `ATM::Tagging` are specified directly by the application when it finds it necessary. The additional network parameters that have no equivalent at ATM level (i.e. cost, packet size, etc.) are forwarded to the ATM agent, which stores them and sends them back upwards when needed in order to perform the reverse translation.

### 5.4.3 The ATM Agent

The system agent for ATM is implemented by class `QzATMAgent`, which specialises class `QzNetworkAgent`. During the initialisation of the agent, the identity of the socket through which the QoS-constrained data will flow has to be specified.

The ATM Agent gets the ATM QoS parameters from the translation unit and performs reservation of bandwidth by interacting with the ATM service provider for Winsock2. In addition, it reports any change in QoS occurring in the network to the application by translating QoS parameters upwards and then issuing upcalls through the `QzUpcall` interface provided by the application.

## 5.5 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. By using these mechanisms, we can guarantee unloaded behaviour to applications (i.e. shield the application from overload conditions) making use of the reservation mechanisms provided by operating system.

### 5.5.1 Windows NT Parameters

Only two QoS parameters are defined for Windows NT. They are:

- `WinNT::PriorityLevel`
- `WinNT::MemoryPaging`

`WinNT::PriorityLevel` defines the priority level at which the process executes. This is used by the operating system to schedule access to the processor. Processes in Windows NT are divided into two priority classes. These classes are the `NORMAL_PRIORITY_CLASS`, which is subject to the usual time-sharing policy implemented by Windows, and the `REALTIME_PRIORITY_CLASS`, which has precedence over the normal class and over system services. The NT scheduler handles threads directly, taking into account not only the priority class but also the thread priority, which is left for the user to assign.

`WinNT::MemoryPaging` determines if the memory allocated by the process will be subject to paging operations, which introduce unpredictable delays and may degrade performance. Values allowed are true (i.e. 1) or false (0).

### 5.5.2 The Windows NT Filter

`QzWinNTFilter` translates QoS parameters understood to Windows NT. It inherits from class `QzTranslation` and implements the translation methods defined by this class. The translation process occurs in two ways, downwards from the generic set of system-level QoS parameters defined by Quartz into NT-specific QoS parameters, and upwards from NT-specific parameters into generic system-level parameters.

Translation between generic system-level parameters and Windows NT parameters is quite simple, since only the parameter `OS::Guarantee` is taken into account. If best-effort service guarantee is requested, the value of `WinNT::PriorityClass` is set to `NORMAL_PRIORITY_CLASS` and `WinNT::MemoryPaging` to true. If unloaded service guarantee is requested, `WinNT::PriorityClass` is set to `REALTIME_PRIORITY_CLASS` and `WinNT::MemoryPaging` to false. Deterministic behaviour is not supported by Windows NT, and unloaded behaviour is assumed if deterministic service is specified.

Parameters `OS::Delay` and `OS::Cost` have no equivalent at Windows NT level. In order to allow the reverse translation, these parameters are forwarded to the Windows NT agent, which stores them and sends them back upwards when needed.

### 5.5.3 The Windows NT Agent

The system agent for Windows NT has been implemented in C++ as a class called `QzWinNTAgent`. `QzWinNTAgent` inherits from class `QzOSAgent`, which is a subclass of `QzSystemAgent`.

During initialisation of the QoS Agent, the process and the thread for which operating system resources will be allocated has to be identified by specifying the corresponding process and thread identifiers through the initialisation (`Init()`) method. By default, these values are assumed as being the current process and its main thread.

Whenever it receives a request for enforcing QoS, the Windows NT agent sets the priority class of the application and the locking state of memory pages according to the value of the QoS parameters obtained after the translation process.

## 5.6 Component Testing

Before testing the whole prototype of the architecture, which will be done by using application examples and will be described in section 7.1, the translation components have been tested individually.

The translation components have been tested with the assistance of an application that allows us to select application and system filters, and then perform the translation of parameters listed in a file. The translation can be performed in both directions (i.e. downwards or upwards). Filters and the direction of the translation are defined based on arguments passed to the application when it is called from the command line. The output of the application shows the transformation suffered by the QoS parameters contained in the input file during the three translation steps performed by the translation unit. Several test cases have been studied with the help of this application, and we have observed that the mapping between parameters at different levels is correctly executed by the translation components.

Table 12 shows an example of A/V streams parameters being translated into ATM and Windows NT parameters (Note: A/V streams parameters will be described in section 6.4.2; for the moment, it is enough to know that they represent the horizontal and vertical resolution of a video screen, the number of bits used to represent the colour of each pixel, and the number of video frames generated per second).

Tests have shown that the translation unit is able not only to translate the parameters listed in Table 12 downwards, but it can also translate them upwards without any loss of information.

| Parameter Set | Parameter Values |
|---|---|
| Application-Specific QoS Parameters (A/V Streams App.) | `A/V::VideoResolutionHorz = 354 pixels/line`<br>`A/V::VideoResolutionVert = 240 pixels/column`<br>`A/V::VideoColourDepth = 24 bits/pixel`<br>`A/V::VideoFrameRate = 30 frames/sec` |
| Generic Application-Level QoS Parameters | `App::DataUnitSize = 253440 bytes/unit`<br>`App::DataUnitRate = 30 units/sec`<br>`App::Guarantee = Unloaded` |
| Generic System-Level QoS Parameters | `Net::Bandwidth = 7603200 bytes/sec`<br>`Net::PacketSize = 253440 bytes`<br>`Sys::Guarantee = Unloaded` |
| System-Specific QoS Parameters (ATM & Windows NT) | `ATM::PeakCellRate = 5280 cells`<br>`ATM::SustainableCellRate = 5280 cells`<br>`ATM::QoSClass = CBR`<br>`WinNT::ProrityLevel = REALTIME_PRIORITY_CLASS`<br>`WinNT::MemoryPaging = TRUE` |

**Table 12 – Example of Translation of QoS Parameters**

| Parameter Set | Parameter Values |
|---|---|
| Generic Application-Level QoS Parameters | `App::EndToEndDelay = 500 msec`<br>`App::EndToEndDelayMin = 450 msec`<br>`App::EndToEndDelayMax = 550 msec`<br>`App::Cost = 6 pence/sec`<br>`App::CostMax = 9 pence/sec` |
| Generic System-Level QoS Parameters | `Net::Delay = 300 msec`<br>`Net::DelayMin = 270 msec`<br>`Net::DelayMax = 330 msec`<br>`OS::Delay = 200 msec`<br>`OS::DelayMin = 180 msec`<br>`OS::DelayMax = 220 msec`<br>`Net::Cost = 4 pence/sec`<br>`Net::CostMax = 6 pence/sec`<br>`OS::Cost = 2 pence/sec`<br>`OS::CostMax = 3 pence/sec` |

**Table 13 – Example of Balancing of QoS Parameters**

| State | Parameter Values | Notes |
|---|---|---|
| Normal Working State | `Net::Delay = 300 msec`<br>`OS::Delay = 200 msec` | `Total Delay = 500 msec`<br>`Split Factor = 0.6` |
| After Resource Adaptation | `Net::Delay = 400 msec`<br>`OS::Delay = 200 msec` | `Total Delay = 600 msec`<br>`Split Factor = 0.6` |
| After QoS Adaptation at System-Level | `Net::Delay = 400 msec`<br>`OS::Delay = 100 msec` | `Total Delay = 500 msec`<br>`Split Factor = 0.8` |

**Table 14 – Example of Adaptation of QoS Parameters**

The test cases also include simulations of QoS balancing and of QoS adaptation. In the first case, which is illustrated by Table 13, parameters delay and cost are balanced between the network and the O.S. supposing a split factor of 0.6 for delay (i.e. 60% of the delay is caused by the network and 40% by the O.S.) and of 0.66 for cost. In a second moment, illustrated by Table 14, we simulate the occurrence of resource adaptation, making the network increase its delay from 300 to 400 milliseconds. The translation unit then performs a rebalancing process, increasing the split factor to 0.8 and causing the O.S. delay to decrease to 100 milliseconds. If this new delay is accepted by the O.S., the original total delay of 500 milliseconds is restored and the process of resource adaptation is contained at system level without involving the application; otherwise, the application is asked by Quartz to accept a higher delay.

In order to complete the evaluation of the behaviour of individual components, we have also tested the correctness of each one of the system agents. The system agents have been tested individually by feeding them with sets of QoS parameters understood at this level (i.e. without interference from the translation unit). The agents have shown to be able to interpret these parameters and perform the corresponding resource reservations by using the associated reservation protocol.

Adaptation has also been tested by simulating the receipt of notification messages from the resource reservation protocol. These messages are issued by the reservation protocol at run-time in order to indicate that the allocation of resources has changed. The system agents have been shown to recognise these messages and have performed the appropriate upcalls in order to trigger adaptation at the higher layers.

## 5.7 Summary

This chapter has presented a prototype implementation of the Quartz QoS architecture. This prototype provides support for Windows NT, RSVP and ATM, and can be easily extended in order to support other network infrastructures and operating systems.

The prototype presented in this chapter was used as the basis for a framework for the development of multimedia applications and for a series of evaluation tests, which are described in the next two chapters.

# 6

# The Quartz/CORBA Framework

In order to provide a complete support for multimedia applications, we have integrated Quartz into a framework based on the CORBA architecture. The Quartz/CORBA framework allows continuous media, which has intrinsic QoS requirements, to be transmitted between applications running in open systems. In this framework, the CORBA architecture provides mechanisms for the exchange of control and synchronisation messages, as well as for audio and video streaming. Quartz is responsible for enforcing the QoS requirements imposed by the application on the transmission of audio and video streaming data.

A more detailed description of the Quartz/CORBA framework is presented in the next section. Further sections describe application scenarios in which the framework is used for transferring media streams with QoS constraints; the implementation of the Quartz/CORBA framework; and also the development of an application for music rehearsal over the Internet built on top of the framework. An analysis of the characteristics of the framework and a comparison with similar programming supports for multimedia conclude this chapter.

## 6.1 Motivation

The use of multimedia applications for transmission of audio and video over the Internet has become a reality due to the accessible prices of multimedia hardware (such as audio cards and video cameras) and the increasing speed of network connections. Users of this sort of application, however, still face quality limitations in both sound and image, since the infrastructure on top of which the application is built (i.e., the Internet and desktop operating systems) provides only best-effort services. These limitations have constrained the effective use of these multimedia applications in areas such as business and telecommunications, where quality must be ensured in order to enable the service provider to charge for the service. Multimedia applications, in addition, suffer from limited compatibility with heterogeneous software and hardware platforms.

Two distinct categories of data are handled by distributed multimedia applications. First we have the control data, which can tolerate some delay but has to be reliable. This behaviour can be provided by using best-effort logical networks such as TCP/IP directly, or by adopting middleware for distributed computing such as CORBA. On the other hand, continuous media are subject to QoS requirements, which are not possible to enforce by using middleware built on top of a best-effort networking infrastructure. The provision of QoS-constrained services for multimedia applications requires the use of QoS-aware middleware that use the QoS mechanisms present in the underlying system in order to enforce the QoS requirements of the application.

Figure 17 shows a generic multimedia application with control and media data. The remote communication in an application like this may use the available network protocols directly, or may adopt a middleware platform that allows transparency regarding issues like remote communication and physical location of the partners.

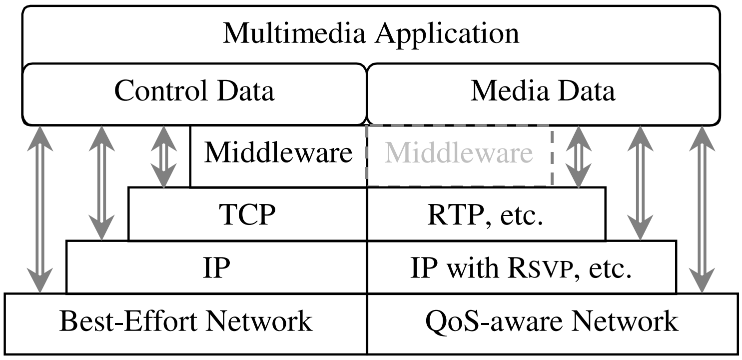


**Figure 17 – Available Technology for Distributed Multimedia Applications**

Since we do not want multimedia applications to have to interact directly with lower-level network protocols, extending current middleware is required in order to allow distributed multimedia applications to impose QoS constraints on the transmission and processing of audio and video data.

Another aspect that has to be taken into account by multimedia applications is media synchronisation [35]. The synchronisation mechanisms necessary in a multimedia system can be implemented at application level by using:

- information carried in-band together with the encoded media (i.e. timestamps) that allows the reconstruction of the media sequence by the receiver (intra-stream synchronisation); and
- events issued in order to define synchronisation points between distinct video streams (inter-stream synchronisation and event synchronisation).

The application code is responsible for handling this information and rendering the media according to the corresponding synchronisation requirements. The synchronisation cannot be executed at support level because the differences between the processing and rendering times of different media at application level have to be taken into account.

## 6.2 Description of the Quartz/CORBA Framework

The CORBA architecture has been extended by the OMG in order to make it more suitable for application areas such as multimedia.

The CORBA streaming mechanism, which was described in section 2.4.3, adds mechanisms for the transmission of continuous media and for the specification of QoS requirements to the CORBA architecture, making it suitable for handling both control and media data.

The notification service [11] is an extension to the CORBA event service that allows objects to issue events with delivery constraints associated to them. It can be used by multimedia applications to issue events in order to define synchronisation points, which are then caught by the destination of the media to reconstruct the timing relation present at the source of the media.

However, despite having been extended in order to provide new communication mechanisms that make it suitable for multimedia applications, CORBA does not specify how QoS is enforced at lower level by these new communication mechanisms. Aiming to overcome this limitation, we have integrated Quartz into the CORBA environment in order to provide a framework for the development of multimedia applications with QoS requirements.

The Quartz/CORBA framework is composed by the objects defined by the CORBA A/V Streams mechanism – virtual devices, stream control objects, stream and flow endpoints – put together with the QoS Agent, the ORB and the CORBA notification service.

Virtual device objects abstract multimedia devices (i.e. cameras, speakers, etc) used by multimedia applications. A stream control object allows the user to control the media flow (i.e., start and stop it) as well as add and remove parties from a multi-party connection. Stream endpoints transfer stream data through the network, getting data from and delivering data to virtual devices. Each endpoint has one or more media flows, which are abstracted as flow endpoint objects (subdivided into flow producers and consumers) located at each end of the flow.

Flow endpoints also issue synchronisation events through the notification service. The notification service is employed in the context of the Quartz/CORBA framework as the basis of a mechanism for imposing inter-stream synchronisation constraints. In this context, events issued by flow producers in order to define synchronisation points are caught by flow consumers to perform media synchronisation.

The data delivery is subject to QoS requirements, which are agreed on by the devices during a negotiation process and then requested during the creation of the stream, i.e. during the binding of two or more virtual devices. After the binding takes place, the QoS parameters associated with a stream (such as video resolution, number of audio channels, etc.) can be modified through the stream control object.

In the framework, the process of QoS translation and enforcement of QoS is delegated to the QoS Agent. This makes the middleware more portable and simplifies significantly the way in which the media streaming mechanisms are built due to the separation between QoS enforcement and media transfer.

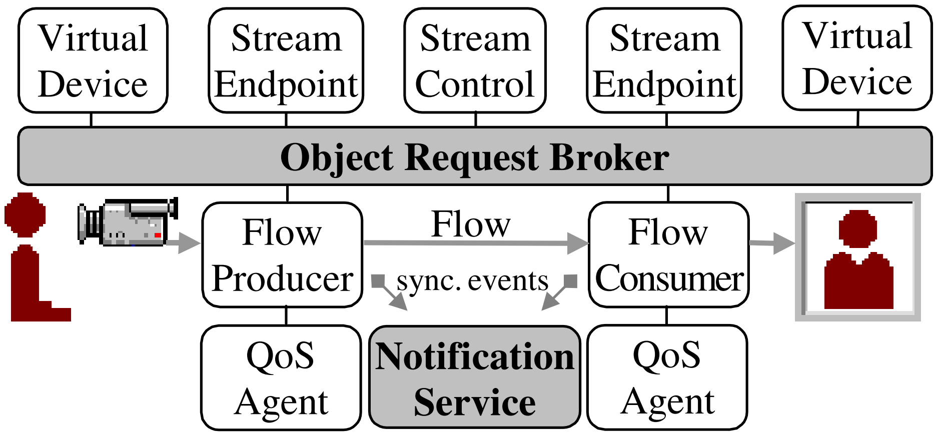


**Figure 18 – The Quartz/CORBA Media Streaming Framework**

Figure 18 shows an application scenario in which a video streaming application is built using the Quartz/CORBA framework. The application specifies the characteristics of the media stream, i.e. the number of flows of audio and video and their expected quality, which are obtained either from a user profile or through interaction with the user. The CORBA streaming mechanism handles the transmission of media between partners, and also controls the flow of media and the membership of the conference. The QoS Agent is informed of the QoS requirements associated to the media flows and handles them accordingly.

The Quartz/CORBA framework simplifies the development of multimedia applications by offering a common middleware for the exchange of control and media data transparently through the network. The application programmer does not have to deal with details related to the kind of network protocol being used, the location of sources and destinations of data, or even the QoS requirements at network level. Instead, the programmer uses high-level abstractions that hide the complexity incurred from distribution and resource reservation, although having the possibility of interacting with the lower-level components if it becomes necessary.

The framework allows the programmer to construct a multimedia application by using built-in classes of virtual devices and configuring them as necessary to provide the desired output quality (i.e., the QoS required by the application). Virtual devices have to be implemented according to the target platform based on the general structure provided by the A/V streams mechanism.

The platform-independent nature of the Quartz architecture makes the framework able to provide QoS to a wide range of applications running over several different reservation protocols. The framework also provides means to specify and enforce the synchronisation constraints that are necessary in some media presentations by using the CORBA notification service.

## 6.3 Application Scenarios

In the following sections we describe several application scenarios in the area of distributed multimedia, showing how the Quartz/CORBA framework could be employed as the basis for building multimedia applications.

### 6.3.1 Media Broadcast

One common application scenario in the area of distributed multimedia is the broadcast of audio and/or video. Figure 19 illustrates the use of the Quartz/CORBA framework to implement a broadcast application. In this scenario, a media source broadcasts a video signal through the network, to one (or more) receivers (i.e. media sinks).

In the video broadcast application scenario, the media flow is built by using the abstractions provided by the CORBA A/V streams mechanism. The virtual device, the stream endpoint and the flow endpoint are present at both ends, while the control, made through the stream control object, is located at the media source. Provision of QoS is delegated to the QoS Agent, which is attached to both flow endpoints (i.e. the flow producer and the flow consumer).

In a broadcast session, the quality of the broadcast is derived primarily from the media source. The source defines the maximum quality of audio or video that will be possible through the transmission, and the quality of the presentation at the receiver will be specified within the limits imposed by the media source.

In a similar way, the flow of media is controlled by the source. The destination cannot influence the flow of media (e.g. stop or pause the broadcast signal).

In order to be added to the stream and receive the broadcast signal, the user has to get a reference to the stream control object in the source and use this reference to call a bind operation.

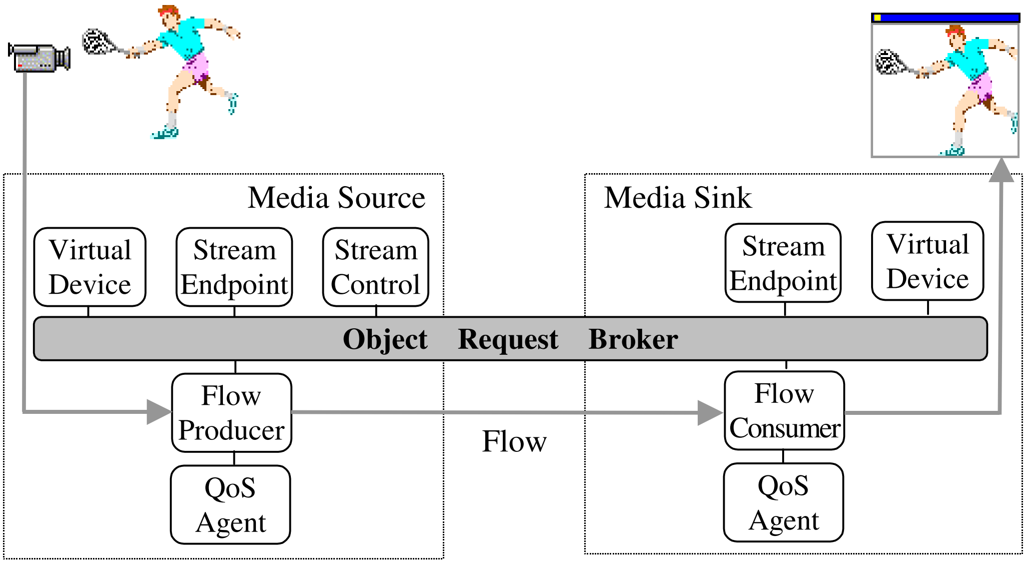


**Figure 19 – Video Broadcast**

An analogy between this situation and a TV or radio broadcast is applicable. In both cases the quality of the broadcast signal depends only on the source, but the quality of the rendered audio or video can be degraded according to the characteristics of the receiver equipment. In addition, the signal is continuously broadcast, independently of factors such as the number or receivers connected to it.

### 6.3.2 On-Demand Media

Another application scenario involving media transfer is the transmission of media on-demand. In this case the receiver specifies its will to receive a specific audio or video signal from the source.

The media sink should be able to control the QoS of the transmission, directly influencing the media source. The requested QoS may influence the billing of the service provided by the media source, in case of a private service subject to copyright. The media sink is also able to control the flow of data (e.g. stop or pause the flow) in the same way it does with locally stored media such as a videotape or an audio CD.

In this category of application, the media sink creates a stream between the input and output devices. The corresponding stream control object, located at the sink, is used to specify the required QoS for the transmission and to control the flow of data.

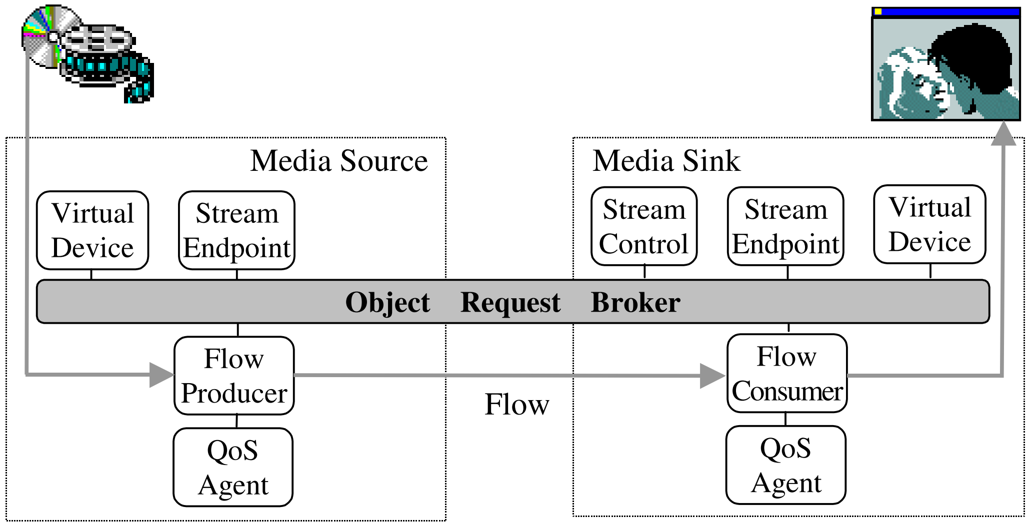


**Figure 20 – On-Demand Video**

Figure 20 shows the transmission of video on-demand from a media source (a video server). This scenario distinguishes itself from the previous one due to the location of the stream control object, which is now located at the media sink.

### 6.3.3 Videoconference

The names 'media flow' and 'media sink' are relative. A component of a multimedia application can be source and sink at the same time, when more than one flow is being transferred through the network.

The most common example of this category of application is the videoconference. In this scenario, each partner has the components necessary to send and receive media (usually audio and video). Each user locally controls the quality of the signal being transmitted, and can also degrade the received signal during the rendering process.

One particular case of videoconference is one in which only two partners are involved, often called digital video telephony. Figure 21 shows a video telephony system designed according to the Quartz/CORBA framework proposed by us. In this scenario, each partner has an input and an output flow, which are represented by a flow producer and a flow consumer respectively. As shown in the figure, the input and output flows can be modelled as a single stream composed by two flows, one in each direction, and a single virtual device can represent the whole video phone. Alternatively, the media flow could be represented as two streams each composed by a unidirectional flow, and the input and output devices could be represented separately.

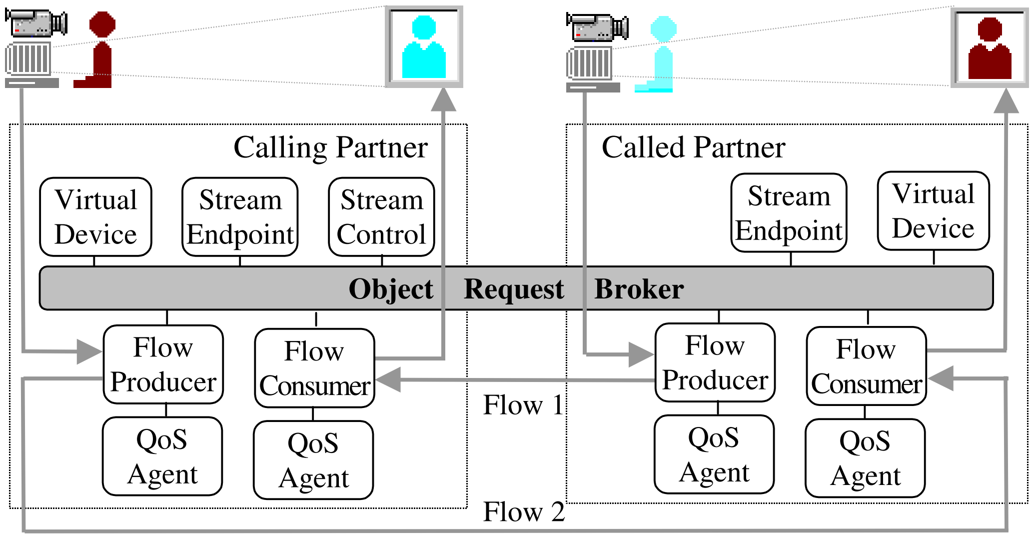


**Figure 21 – Video Telephony**

The control over the stream (represented by the stream control object) is held by the calling partner. This is justifiable if we make an analogy of this scenario with a telephone call. In a telephone call, the caller usually controls the connection and disconnection. This happens mainly because he is the one who is paying for the call.

In the particular case of the video telephony application, the caller also controls and is charged according to the QoS requirements associated to the call. The privacy of the called partner is preserved because he has total control over his local devices, and can choose not to answer the call or to redirect it to a media recording device that behaves like an answering machine (or voice/video mail).

In this application scenario, QoS is provided by the QoS Agents attached to each of the flow endpoint objects. Since two flow endpoints are present at each side of the call, one for input and another for output, two QoS Agents are needed in order to establish the QoS on both the input and the output flows.

This scenario can easily evolve in order to become a multi-party conference by adding new partners to the scenario and connecting them to the flows. In this case, flow endpoints with support for multipoint communication should be employed.

### 6.3.4 Media Session with Synchronisation

In a complex application, in which multiple flows of media related to each other are being received by one particular destination, synchronisation between the media during the exhibition may be necessary. In this situation, output devices have to rebuild the synchronisation patterns originally present when the media was produced.

One common scenario in which synchronisation of media is necessary is the composition of multimedia documents with media received from different sources [140]. Figure 22 shows a scenario in which a movie is retrieved from a server. The image and the soundtrack are stored separately to allow the receiver to select between different languages, but without the necessity of duplicating the video. However, encoding video and audio separately obligates the receiver to execute the synchronisation between the audio and the video signals received from different sources possibly with different generation and transmission rates.

In this scenario, each of the sources has a flow producer, while the sink has a flow consumer for each media. As occurred in the examples of on-demand video, the media sink controls the exhibition of the content of the multimedia document through the stream control object. This object interacts with the stream endpoints in order to control the flow of media data through the stream.

In order to enable synchronisation, the media sources issue synchronisation events that are caught by the media sink in order to rebuild the original synchronisation pattern. These events are produced by the flow producers, transmitted by the notification service, and caught by the flow consumers.

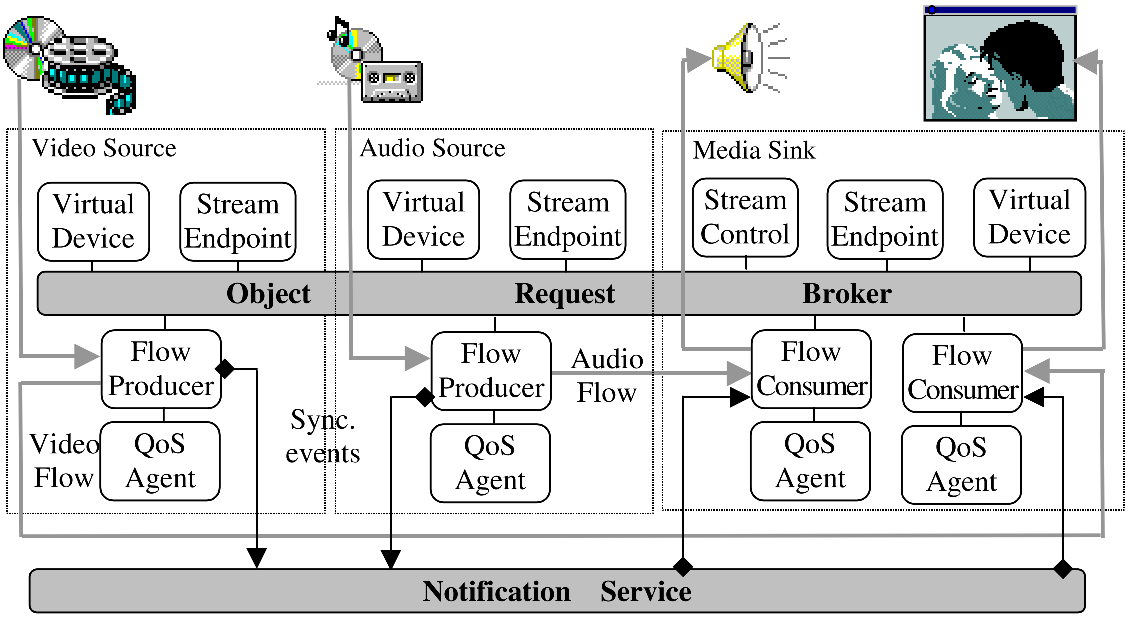


**Figure 22 – Multimedia Document with Synchronisation**

## 6.4 Implementation of the Quartz/CORBA Framework

The Quartz/CORBA framework has been implemented by integrating an existing implementation of the A/V streaming mechanism with the prototype of Quartz described in Chapter 5. In order to simplify the implementation, we have not included support for synchronisation in this prototype and in the application example that will be presented in section 6.5.

### 6.4.1 Implementation of the CORBA A/V Streams Mechanism

The CORBA A/V Streams mechanism adds to CORBA the ability to transfer continuous media between distributed objects. The specification of the CORBA A/V streams mechanism was described in detail in section 2.4.3.

We have an in-house implementation of the A/V streams mechanism that supports the light profile of the standard defined by the OMG. This implementation was developed by David O'Flanagan and described in his M.Sc. dissertation [111]. The support was written in C++ using Iona's Orbix 2.3 [73]. The implementation was carried out on top of Windows NT 4.0 using Microsoft's Visual Development Studio 5.0. The implementations of the TCP/IP and UDP/IP protocols (including UDP multicast) provided by WinSock 2.0 [137] were used for the transmission of continuous media data.

Figure 23 shows the UML of the light profile of the CORBA A/V Streams mechanism. This figure illustrates each of the interfaces described in IDL by the A/V streams specification.

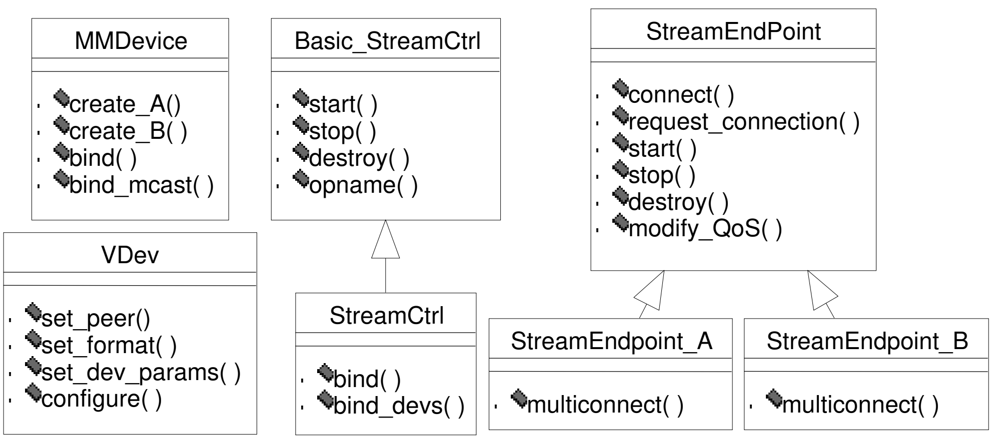


**Figure 23 – UML of the CORBA A/V Streams Mechanism**

The `MMDevice` is the interface supported by the object responsible for initialising the multimedia application, creating the media partners (the ‘A’ partner, which initialises the connection, and the ‘B’ partner) and binding them. The `VDev` is the interface of the virtual multimedia device, which is extended according to the particular characteristics of the device. It provides mechanisms for configuration of the device and for setting the media format. The stream control object supports the interface `StreamCtrl`, which inherits from `Basic_StreamCtrl`. It provides operations for controlling the flow of data through the stream and additional binding operations. The stream endpoint is responsible for providing operations for connecting the flows, for controlling the flow of data through the stream, and for setting QoS parameters. The stream endpoint is specialised into interfaces for the ‘A’ and ‘B’ partners for providing operations that allow the creation of multipoint connections.

A stream is composed by one or more media flows. The light profile of the CORBA A/V streams specification does not standardise how the individual flows are implemented (this is covered by the full profile, though). In this implementation of the standard, the structure described by the UML diagram shown in Figure 24 has been adopted for implementing individual flows. According to this structure, the flow endpoint interface provides operations for controlling the flow of data through the individual stream, for setting a routine that will interact with the virtual device (i.e. to feed or to be fed with data), and for transferring data through the network channel.

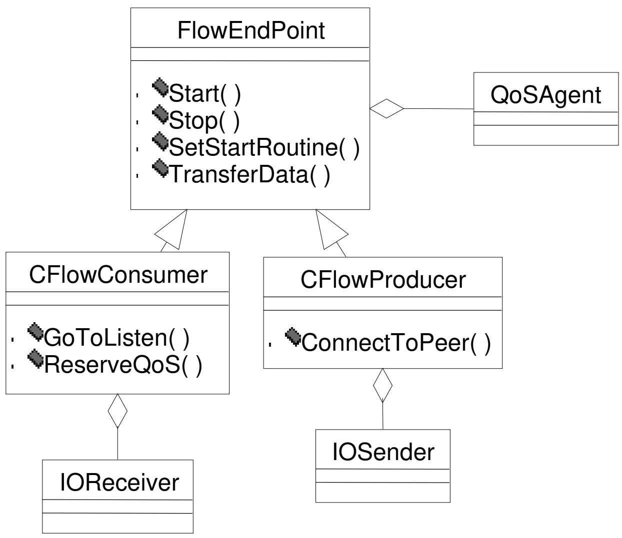


**Figure 24 – UML of the Flow EndPoint**

The flow endpoint is specialised as flow consumers and flow producers, which have operations specific to their corresponding communication roles and are associated to an I/O object that provides the actual communication channel (i.e. an `IOSender` for the flow producer and an `IOReceiver` for the consumer). In particular, the I/O objects provided by this implementation use TCP/IP for point-to-point communication and UDP/IP multicast for point-to-multipoint channels.

The figure also shows that each flow endpoint is associated to a QoS Agent, which during this phase of the implementation of the A/V streams mechanism corresponded to a 'dummy' object. The next item explains how this object was replaced with the Quartz QoS Agent, leading to the integration of the A/V streams implementation with the Quartz architecture.

### 6.4.2 Integration of A/V Streams and Quartz

The implementation of the CORBA A/V streams mechanism described in the previous item was adapted in order to use Quartz for provision of QoS. This is done through the flow endpoint, which interacts with Quartz in order to specify the QoS requirements imposed by the application. By separating the support for QoS from the communication support (i.e., from the implementation of the A/V streams mechanism) we have made the middleware more portable and easier to implement, test and maintain.

The interaction between flow endpoints and Quartz comprises the following tasks:

- initialisation of Quartz, when are identified the communication port and the process that will avail of services on which QoS requirements will be imposed;
- request for enforcement of QoS requirements, which is done by calling the `QoSRequest()` method of the QoS Agent during the establishment of a connection between flow endpoints; and
- receipt of upcalls from the QoS Agent, which are issued in order to report QoS adaptation, errors or warnings.

By performing these tasks, flow endpoints make the use of Quartz completely transparent for the application, which only have to specify their QoS requirements through the QoS specification mechanisms that are provided by CORBA A/V streams.

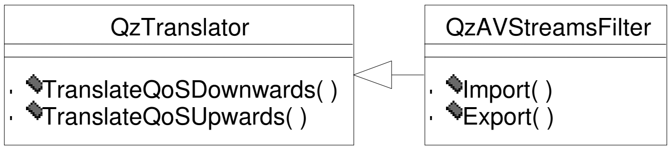


**Figure 25 – UML of the A/V Streams Filter**

Since the CORBA A/V streams specification defines its own mechanisms for specification of QoS, a filter had to be written in order to recognise the parameters defined by A/V streams. The Quartz A/V streams filter is implemented by class `QzAVStreamsFilter`, which is shown in Figure 25. This filter has additional methods to import and export from the format used by Quartz to store QoS parameters (i.e. the `QzQoS` structure) to the format used by the A/V streams mechanism (i.e. the `AVStreams::QoS` class).

The A/V streams filter imports parameters and values from the `AVStreams::QoS` class defined by the A/V streams specification and translates them into generic application QoS parameters. Reverse translation is also executed by the filter in case of QoS adaptation. The A/V streams QoS parameters are listed by Table 15.

The mapping from A/V streams parameters into generic application-level QoS parameters is shown by Table 16. Parameters `A/V::AudioQuantisation` and `A/V::VideoColourModel`, which define the encoding format used by the application, have no influence on the values of QoS parameters at lower abstraction levels.

| **Parameter Name** | **Description** |
|---|---|
| `A/V::AudioSampleSize` | Size of audio sample (in bytes) |
| `A/V::AudioSampleRate` | Number of audio samples (in samples/second) |
| `A/V::AudioNumChannels` | Number of audio channels |
| `A/V::AudioQuantisation` | Quantisation (linear, u-law, A-law, GSM, etc.) |
| `A/V::VideoFrameRate` | Number of video frames (in frames/second) |
| `A/V::VideoColourDepth` | Number of bits representing colour for each pixel |
| `A/V::VideoColourModel` | Colour encoding (RGB, CMY, HSV, YIQ, etc.) |
| `A/V::VideoResolutionHorz` | Horizontal resolution of the image (in pixels) |
| `A/V::VideoResolutionVert` | Vertical resolution of the image (in pixels) |

**Table 15 – A/V Streams QoS Parameters**

| CORBA Audio Streams Parameters | CORBA Video Streams Parameters | Generic Application-Level Parameters |
|---|---|---|
| `A/V::AudioSampleSize *` `A/V::AudioNumChannels` | `A/V::VideoColourDepth *` `A/V::VideoResolutionHorz *` `A/V::VideoResolutionVert` | `App::DataUnitSize` |
| `A/V::AudioSampleRate` | `A/V::VideoFrameRate` | `App::DataUnitRate` |

**Table 16 – Mapping between A/V Streams Parameters and Generic Parameters**

The A/V streams QoS parameters, after being translated by the A/V streams filter, are further translated by the Quartz translation unit and handed to the system agents corresponding to the reservation mechanisms present in the system. At this point, the RSVP protocol is used by the corresponding system agent in order to enforce QoS.

## 6.5 Application: The Distributed Music Rehearsal Studio

An application for music rehearsal over the network was written by John Segrave-Daly for his final year project [133]. The distributed music rehearsal studio allows musicians to play together by using instruments connected to networked computers. This application has strong QoS requirements, since users expect that factors such as network or computing delays or lack of network bandwidth do not interfere with their musical performance.

The human ear is able to notice differences of more than about 30 milliseconds in the synchronisation of audio events. A difference higher than this amount between two audio sources is enough to allow a listener to notice that the music is out-of-sync. As a result, the distributed music rehearsal studio must keep the delay between the production and reproduction of audio signals within 30ms, otherwise the rehearsal session will lose integrity [133]. Consequently, packets with a lifetime exceeding this limit should be discarded. This phenomenon sets the application apart from traditional audio streaming and audio conference tools, which can tolerate much higher delays.

Figure 26 shows a usage scenario of the distributed music rehearsal studio, in which three musicians participate in a rehearsal. Each user has an audio source object, responsible for broadcasting the audio produced by the instrument being played by him, and audio sinks for each musician participating in the rehearsal.

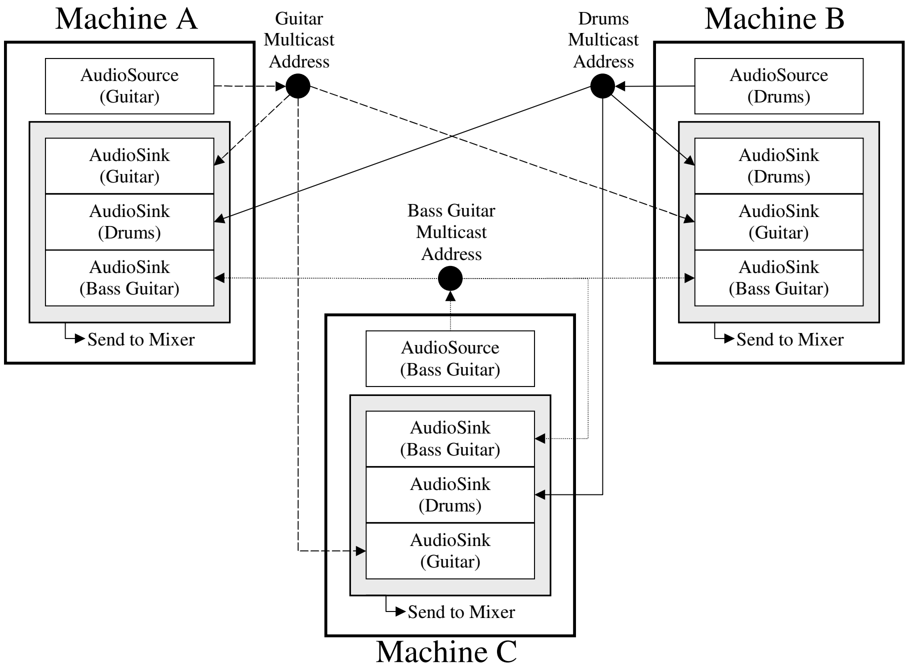


**Figure 26 – Usage Scenario of the Distributed Music Rehearsal Studio**

Audio sinks are connected to a mixer object, which is similar to a mixing desk. The mixer object mixes the audio sent by each musician and reproduces the resulting audio, which is reproduced by the local machine and heard by the musician. This audio signal gives feedback to the musician, who must play his instrument in accordance with what the other musicians are playing.

The distributed rehearsal studio was built on top of the Quartz/CORBA framework, which is responsible for enforcing the requirements imposed by the application. Audio sources and sinks are virtual devices in the context of the A/V streams mechanism. Sources and sinks are connected to flow producers and consumers respectively, which allow them to transfer the audio signal through the network. In addition, Quartz enforces the QoS constraints imposed by the applications in order to avoid audio packets being delayed by a possible lack of resources at network or operating system levels. Audio sampling and reproduction are performed by using, respectively, the waveIn API and the Microsoft DirectSound API, an integrated part of DirectX 3.0 [98].

Figure 27 shows the user interface of the distributed rehearsal studio. Through this interface, the musician can create audio sources and sinks, and can select the quality with which the audio is encoded. Audio controls such as volume and panning (balance) are also provided.

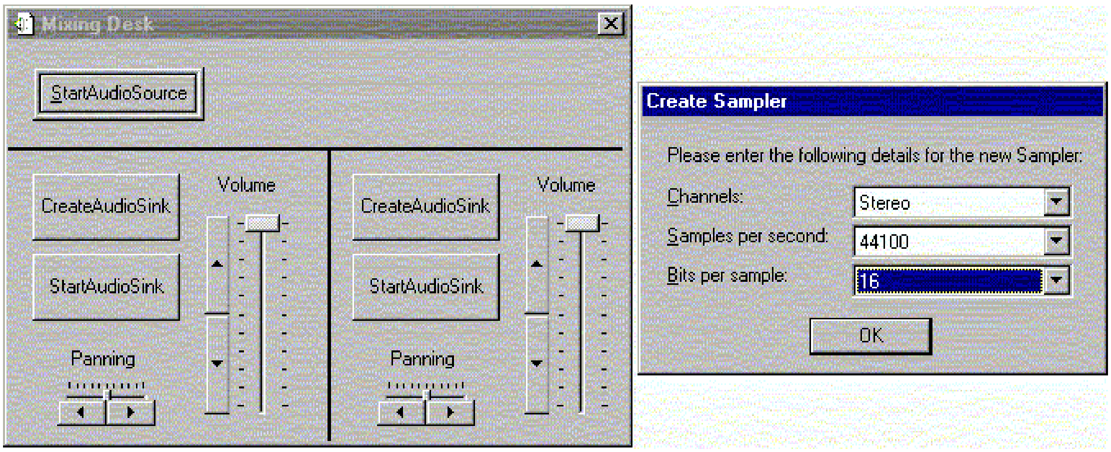


**Figure 27 – Interface of the Distributed Music Rehearsal Studio**

Tests with the prototype implementation of the distributed music rehearsal studio have shown that delays still largely exceed the desired limit of 30 milliseconds. This is due to the unpredictable latency introduced by the waveIn API and by the audio drivers provided by the operating system. Unfortunately, the API and the drivers do not provide resource reservation mechanisms that allow us to limit the latency. This does not affect the role of the Quartz/CORBA framework, which is able to enforce QoS by reserving bandwidth for the application and by shielding the application from any interference caused by excessive CPU usage.

## 6.6 Analysis of the Quartz/CORBA Framework

With the adoption of the CORBA architecture support – both the client-server and stream interaction models – as the core of the distributed multimedia framework, multimedia applications are given the ability to handle control data and stream data uniformly using high-level abstractions. The object-oriented nature of CORBA provides a large set of benefits to the application development process (e.g. reusability, encapsulation, use of software engineering techniques, etc.). In addition, the adoption of the CORBA streaming mechanism together with network protocols with multicast capabilities makes efficient and scalable group communication feasible. Furthermore, the framework supports the provision of QoS requirements described as A/V Streams QoS parameters through the use of Quartz for translation and enforcement of QoS requirements imposed by applications.

Several other proposals of programming supports for the development of multimedia applications in open systems, which were presented in Chapter 3, aim to achieve similar results by adopting different approaches.

The ReTINA project [18] has basically the same goals as our proposal, but the adoption of a binding model inherited from ODP introduces a new level of complexity that has to be handled by the application programmer. Our approach maintains the simplicity and transparency of the CORBA binding model, incorporating a binding paradigm appropriate for multimedia applications through the introduction of a stream abstraction, which allows connections with associated QoS specifications and multi-party connections.

Furthermore, ReTINA proposes extensions to the CORBA IDL for the introduction of stream interfaces. Major changes in the IDL are not likely to be well accepted by the OMG and by the CORBA community, and may reduce the compatibility between different ORBs. On the other hand, the Quartz/CORBA framework for distributed multimedia applications that was proposed by us is entirely compatible with the current OMG specifications for ORBs and for the IDL language.

The same considerations apply to the DIMMA [116] and Amber [86] projects developed by APM that had a strong influence on the ReTINA project.

The XRM [88] project is a proposal for a framework that provides multimedia applications with QoS specification mechanisms together with an appropriate binding architecture, called Xbind, which is built upon CORBA. XRM and Xbind, however, do not provide some features that are provided by Quartz such as QoS enforcement at the operating system level and resource adaptation.

The TAO project [129] proposes optimisations on the communication support to improve the performance achieved in method calls between clients and servers. Despite recognising that a better performance has to be provided by the CORBA communication protocols, we consider that this is not sufficient for the provision of transfer of media data, since high performance does not necessarily guarantee the QoS requirements imposed by distributed multimedia applications.

Several other projects in the field of distributed multimedia, such as SUMO [35] and DAVE [49], do not adopt an approach based on standards for distributed object computing, constraining their utilisation in open systems and limiting their acceptance by computer software developers. On the other hand, our multimedia framework adopts an approach based on standard solutions (i.e. CORBA) in order to provide and environment for development and execution of multimedia applications.

## 6.7 Summary

This chapter presented a framework for building multimedia applications. This framework shows how Quartz can be used by middleware platforms such as CORBA in order to allow applications to impose QoS constraints on the services provided by the underlying system. An implementation of the Quartz/CORBA framework was presented, together with a multimedia application that was built using the framework. We have also analysed the achievements of the framework and compared it to similar support for the development of multimedia applications.

# 7

# Validation and Evaluation

This chapter presents a series of validation tests, which were executed in the form of application examples that were implemented using the Quartz prototype. In addition, we analyse the characteristics of Quartz, taking into account the requirements that were imposed on it during the design phase. Finally, we compare Quartz to other supports for QoS specification and enforcement that are described in the literature.

## 7.1 Validation Tests

We have implemented some applications on top of the prototype of Quartz in order to analyse whether or not Quartz meets the requirements imposed during the design phase. These applications – the remote copy application, the telephone exchange application, and a new version of the distributed music rehearsal studio that is built directly on top of the sockets API – are described in the following sections. Further in this chapter, we present some conclusions based on the experience obtained with these applications, and provide some performance measurements that quantify the overhead imposed by Quartz in the form of processing time.

### 7.1.1 The Remote Copy Application

In order to evaluate the behaviour of Quartz in an environment with multiple network-level reservation protocols, we have implemented a Windows-based version of the Unix remote copy command. The "`rcp`" command allows users to transfer files between machines interconnected by a network. Our "`rcp`" application goes beyond the Unix command by providing a graphical interface that allows the user to choose between multiple network protocols and to specify QoS requirements that are to be observed in the file transfer operation. This application uses Quartz to enforce its QoS requirements.

By using the graphical interface, which is shown by Figure 28, the user is able to specify the name of the file to be transferred and select the protocol to be used for data transfer. The user can choose between three different communication protocols: TCP, UDP and ATM. In the first two cases, RSVP is used for resource reservation, while ATM is used directly in the latter case.

The role played by the application (i.e. client or daemon/server) must be specified on startup. Information needed to establish the connection also has to be provided by clients. This information consists of the host name, IP or ATM address and the communication port of the destination. IP multicast is also supported. In this case, the user has to specify an IP multicast address in the 'target address' field for the (possibly multiple) clients and the daemons.

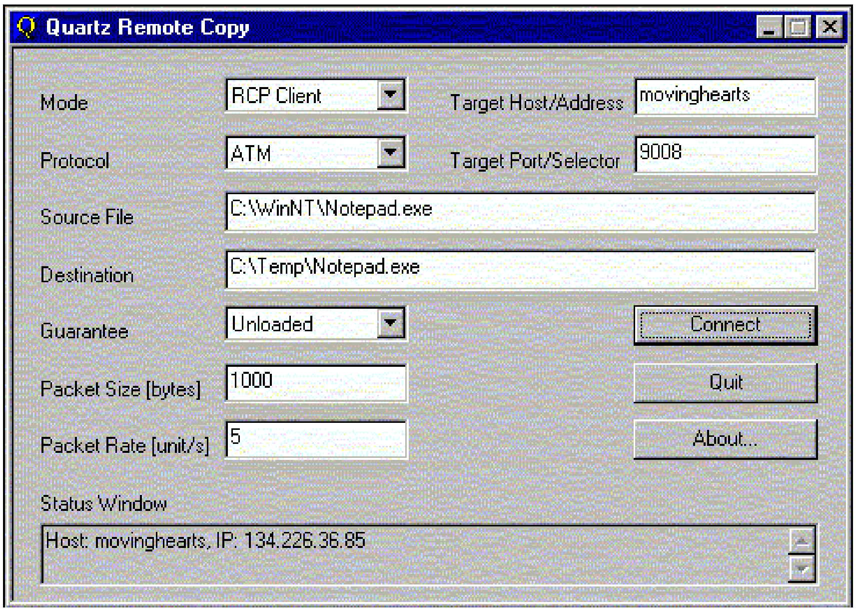


**Figure 28 – The Remote Copy Application**

In addition, the graphical interface allows the user to specify QoS requirements such as the kind of guarantee (best-effort, unloaded or deterministic), the size of data packets, and the packet rate. These requirements are interpreted by the application filter for data packet transfer, which is implemented by class `QzDataPacketFilter`. The QoS parameters understood by this filter are listed by Table 17.

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

**Table 17 – Parameters Recognised by the Data Packet Filter**

The data packet filter performs the translation of these parameters into the generic set of application parameters defined by Quartz. This translation step is very straightforward, since these parameters are very similar to the generic application parameters used by Quartz. The parameters resulting from this first translation step are translated further into system parameters and passed to the system agents that perform the necessary reservation of resources at lower level. The application provides an interface for Quartz so that it can perform upcalls in order to report errors, warnings or dynamic changes in QoS.

This application was based on the sample provided together with Intel's implementation of RSVP. The addition of Quartz as a means for reserving resources simplified the application and allowed us to incorporate support for ATM without increasing complexity. According to the protocol selected by the user, the corresponding system filter and agent are plugged into the QoS Agent, and from this point the translation of parameters and the reservation of resources are handled directly by Quartz according to the selected protocol.

The remote copy application shows us that Quartz is able to perform QoS reservations by using both ATM and RSVP without interfering with the way the application specifies QoS and requests QoS to be enforced. In addition, the application code is simplified and made more portable by the use of Quartz due to the separation between data transfer, which is performed by the application directly, and QoS enforcement, which is executed by Quartz.

### 7.1.2 The Telephone Exchange Application

In order to show the use of Quartz in an environment distinct from the traditional computing environment – i.e. desktop machines interconnected by a computer network such as the Internet – we have implemented a software model of a telephone exchange.

Telecommunications systems are subject to QoS requirements stipulated by government regulatory agencies (such as the FCC in the US, OFTEL in the UK and the ODTR in Ireland), international organisations (such as the ITU-T) and internal standards defined by telecom operators. Very strict limits on the time the system takes to process a phone call are defined by the companies and regulatory bodies. Consequently, real-time software has to be run on the telephone exchange so that the call set-up process can be time bound. When these requirements are not fulfilled, besides the degradation of the service provided to the consumer, it may result in hefty fines imposed by regulatory bodies and, ultimately, in the suspension of the license to operate the telecommunications service.

The processing of a telephone call passes through well-defined stages [48], which are illustrated in the state transition diagram in Figure 29. Two of these stages, which are performed by a computerised telephone exchange and are subject to QoS requirements, are of particular interest for this study.

The exchange is responsible, after the customer takes the phone off the hook, for reserving a path from the customer’s equipment to the exchange. After that the exchange has to perform an initialisation procedure. First, it has to identify the customer by mapping the phone line through which the off-hook signal was received to a directory number and a customer name, to whom the call will be charged afterwards. The exchange is also responsible for identifying the class of service to which the customer is subject. The class of service defines the way in which the call will be handled, including charging methods and rates, authorisation for long-distance calls and for using premium services, etc. After this is accomplished, a dial tone is sent to the customer's terminal. The time taken by this whole process is expected not to exceed the limits imposed by regulatory bodies or by the company's own quality standards, characterising the first QoS requirement imposed on the call initialisation process. We refer to it as the "dial-tone deadline" from now on. In the figure, it is represented by the time taken to change from state ⓪ to state ①.

In a second phase, after the customer dials the phone number to which he wants to be connected, the exchange is responsible for routing the call to the destination, reserving lines along the way. When the lines are reserved, a ring tone is sent to the caller and a ringing signal is sent to the called phone. If an available route is not found because there are no lines available or because the called party is busy, a busy tone is sent to the caller. The time taken to route the call and send a calling tone (or busy tone) to the caller is also subject to limits, resulting in the second QoS requirement imposed on the call set-up process. We refer to this as the "ring tone deadline". It corresponds to the time spent to change from state ① to state ② in Figure 29.

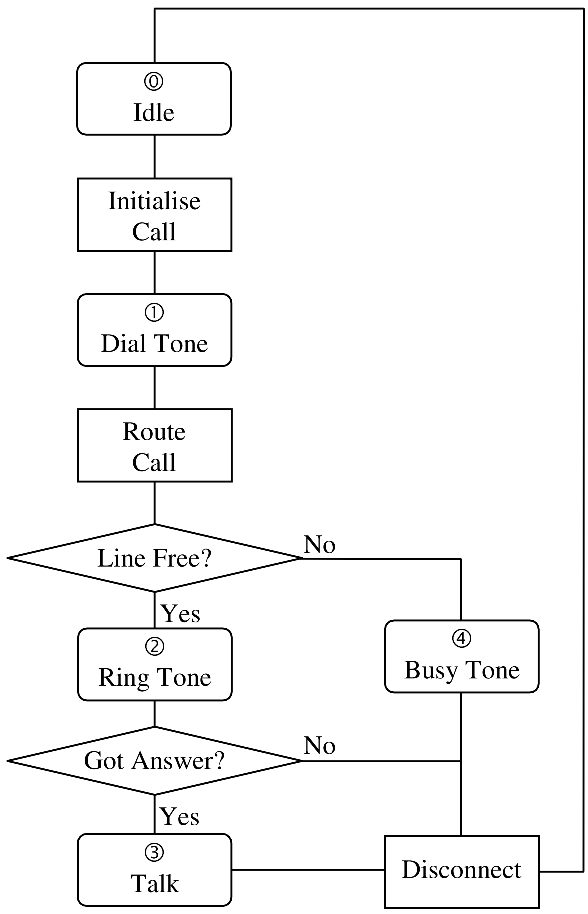


**Figure 29 – State Diagram of a Telephone Call**

The call clean-up process is not subject to QoS requirements from the exchange's point of view. The quality of the phone line through which the conversation takes place is also subjected to requirements, but this does not interfere directly with the call initialisation and routing process.

We have implemented a simplified software model of a telephone switch in order to evaluate the capacity of Quartz as a mechanism for enforcing the QoS requirements present in the call switching process, i.e. the dial tone and ring tone deadlines. In addition, Quartz also performs the allocation of telephone circuits.

This application example is built on top of Windows NT and is totally written in C++. An application filter for circuit switching, a system filter and agent for deadline-based scheduling, and a system filter and agent for reservation of telephone circuits have been written by us.

Class `QzCircuitSwitchFilter` implements the application filter for circuit switching. It recognises the parameters listed by Table 18, which are proper for applications such our telephone exchange simulator, and translates them into generic QoS parameters.

| **Parameter Name** | **Description** |
|---|---|
| `CSw::CircuitType` | Type of circuit (POTS, ISDN, T1, E1, etc.) |
| `CSw::CircuitDelay [Min,Max]` | Total transmission delay (in milliseconds) |
| `CSw::SwichingDelay [Min,Max]` | Total switching delay (in milliseconds) |
| `CSw::ErrorRatio [Max]` | Transmission error (in %) |
| `CSw::Guarantee` | Guarantee level (best-effort, unloaded, etc.) |
| `CSw::CircuitCost [Max]` | Financial cost (in local currency per second) |
| `CSw::SecurityLevel` | Security level (none, encrypted, etc.) |

**Table 18 – Parameters Recognised by the Circuit Switch Filter**

In order to simulate the allocation of phone lines, we provide a system agent for allocation of phone circuits implemented by classes `QzPhoneCircuitAgent`.

The phone circuit agent is used together with a system filter for phone circuits, implemented by class `QzPhoneCircuitFilter`. This filter translates the system-specific parameters listed in Table 19 into the generic QoS parameters defined by Quartz.

| Parameter Name | Description |
|---|---|
| `Phone::CircuitType` | Type of circuit (POTS, ISDN, T1, E1, etc.) |
| `Phone::CircuitDelay [Min,Max]` | Total transmission delay (in milliseconds) |
| `Phone::ErrorRatio [Max]` | Transmission error (in %) |
| `Phone::Guarantee` | Guarantee level (best-effort, unloaded, etc.) |
| `Phone::CircuitCost [Max]` | Financial cost (in local currency per second) |
| `Phone::SecurityLevel` | Security level (none, encrypted, etc.) |

**Table 19 – Parameters Recognised by the Phone Circuit Filter**

Initially, the same operating system agent for Windows NT that was used in the other examples was used by this application. Later, a system agent and a system filter for deadline-based process schedulers were written in order to take into account the establishment of deadlines for the processing of switching tasks. The system agent and the system filter for deadline schedulers are implemented by classes `QzDeadlineSchedAgent` and `QzDeadlineSchedFilter` respectively. The system-specific QoS parameters understood by these components are listed by Table 20.

| Parameter Name | Description |
|---|---|
| `DlSched::Deadline [Min,Max]` | Processing deadline (in milliseconds) |
| `DlSched::Guarantee` | Guarantee level (best-effort, unloaded, etc.) |

**Table 20 – Parameters Recognised by the Deadline Scheduling Filter**

Our model of a telephone exchange is able to process 10 phone calls concurrently. For each phone call, the exchange reserves the circuit from the customer's equipment to the exchange and sets the dial tone deadline for the call initialisation process. After the customer dials the number of the called terminal, a new reservation is performed, which allocates a circuit from the exchange to the called terminal and sets the calling tone deadline for the routing process.

Reservation of resources is simulated by the phone circuit agent, because the actual allocation of resources would require the presence of actual switching hardware. Similarly, the deadline-based scheduling agent only simulates the scheduling of the switching tasks, since the platform in which the application was implemented does not provide deadline-based scheduling mechanisms.

The telephone exchange application shows how QoS requirements very different from those present in traditional computer applications can be expressed using the specification mechanisms provided by Quartz. This example also shows that, despite the different nature of the platform that provides resources for the application, Quartz is able to interpret the high-level requirements and perform resource reservations at the low level.

### 7.1.3 The Distributed Music Rehearsal Studio

A new version of the distributed music rehearsal studio has been built in order to evaluate the use of Quartz as a mechanism for QoS enforcement that is used directly by a multimedia application, instead of being used indirectly by applications through multimedia middleware. This new version of the rehearsal studio uses sockets directly instead of the CORBA A/V streaming mechanism for transferring media data.

Compared to the version described in section 6.5, the functionality of the application and the user interface remained the same. The only difference can be found at networking level, where the dependencies on CORBA have been removed. This made the application more lightweight, reducing the executable by approximately 10 times the original size, and less memory-hungry, reducing the memory allocated by the application in about 30%. Despite these advantages, there was no noticeable improvement in the overall performance of the audio transmission, since the problems with the latency originated from the waveIn API and from DirectSound.

This example shows that multimedia applications, such as the music rehearsal studio, can avail of the QoS specification and enforcement mechanisms that are provided by Quartz in order to have their QoS requirements fulfilled.

### 7.1.4 Results Achieved with the Validation Tests

Table 21 summarises the applications built using Quartz for test purposes. These applications have been selected in order to demonstrate the fulfilment of the requirements imposed on the Quartz architecture during the design process, which were described in section 4.1 of this thesis.

| Application | App. Filter | OS Agent & Filter | Network Agent & Filter |
|---|---|---|---|
| Remote Copy | Data Packet | WinNT | RSVP, ATM |
| Telephone Exchange | Circuit Switch | Deadline Scheduler, WinNT | Phone Circuit |
| Rehearsal Studio | A/V Streams | WinNT | RSVP |

**Table 21 – Applications Built Using Quartz**

An initial observation of the nature of the tests shows that Quartz can be used in different areas of application besides multimedia transfer, demonstrated previously by the Quartz/CORBA framework and the rehearsal studio application. The test applications show that Quartz is also suitable for specification and enforcement of QoS for data transfer applications (such as the remote copy example) and for real-time applications (such as the telephone exchange model).

In addition, the tests described previously, including the rehearsal studio, show the adequacy of the mechanisms for specification of QoS provided by Quartz. 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 Quartz/CORBA Framework (and consequently the rehearsal studio) uses parameters describing the quality of audio and video streams. The format of these parameters is defined by the A/V streams specification and is translated by the corresponding filter. On the other hand, the remote copy example uses a data packet filter that recognises parameters for applications transferring data packets, while the telephone switch application uses a different set of parameters appropriate for circuit switched lines and for deadline scheduling. An application filter translates application parameters into Quartz-defined generic parameters in each case.

The remote copy example also shows that the resource provider can be changed without interfering with the application code. The remote copy application shows that even using a different network resource provider, which in this case could be either RSVP or ATM, the user can obtain similar QoS-constrained services. This shows that the lower-level reservation protocols supported by the networking infrastructure are made transparent for the application. 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.

On the other hand, the telephone exchange application shows that the component responsible for the reservation of resources provided by the operating system can be exchanged in a similar way. In this case, we use either a simulator for deadline scheduling or the same Windows NT agent used for the other examples for processing and routing phone calls.

The multiplicity of operating system and network agents and filters used by the examples 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.

### 7.1.5 Performance

Performance tests have shown that the overhead added by Quartz to the application is very small. Table 22 shows typical values for 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. The complete results of the tests are shown in the appendix of this thesis.

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

**Table 22 – Overhead Imposed by Quartz**

The total overhead caused by Quartz in a single request (i.e. the time taken to specify, translate and interact with the resource reservation protocols) is 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 name (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 due to the amount of state information that has to be gathered by both WinSock and Quartz; since in ATM the reservation mechanism is integrated with the transport, no extra time is taken to initialise it.

During the tests we have used the real-time scheduling class provided by Windows NT in order to reduce the influence of other processes in the results. The overhead, however, can still vary about ±4% for both RSVP and ATM under similar load conditions due to unpredictability introduced by the system services that are used by Quartz (i.e. the values shown in Table 22 have an error of ±4%).

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 transmission of data, which depends only on the networking infrastructure and on the resources reserved for the session.

## 7.2 Fulfilment of Requirements

The following items explain how the requirements imposed during the design phase, which were listed in section 4.1, are fulfilled by the Quartz architecture.

### 7.2.1 QoS Specification

Quartz allows users to specify QoS requirements according to the notion of QoS understood by them. This is achieved by adopting a multi-step, extensible translation mechanism.

The translation unit understands parameters specified at application level by using application filters, which can be selected from a library or written by the user. The examples presented show that these filters can handle a wide variety of formats of QoS parameters. These different sets of parameters, each one representing a different notion of QoS present at application level, can be easily translated into a generic format used internally by the translation unit as shown by the applications implemented using Quartz.

Similarly, the system filters were shown to be able to perform the translation of generic parameters adopted by Quartz into QoS parameters meaningful at the lower level. These filters were implemented for different reservation protocols with different notions of QoS, such as ATM and RSVP at the network level, and for the Windows NT real-time support and for deadline-based schedulers at the operating system level, showing the suitability of Quartz for handling QoS in heterogeneous systems.

At the same time, the application and system filters are made easy to implement since the main translation step is performed by the interpreter. The interpreter handles the translation between different abstraction levels (i.e. translates parameters from the application level into the system level), balances requirements between the network and the operating system, and performs QoS adaptation whenever necessary. Consequently, these aspects do not have to be handled directly by the application or systems filters.

A typical filter (such as the data packet filter and the RSVP filter) has less than 300 lines of code, and should take an average programmer less than one working day to define the mapping between parameters, and then write and test the filter.

### 7.2.2 QoS Enforcement

The user interface defined by the Quartz QoS architecture hides from the user the details regarding the way in which resources provided by both the network and the operating system are reserved. This allows the application to have its QoS requirements enforced independently of the characteristics of the sub-system that is providing the resources.

Quartz can be seen as a common interface to the diverse resource providers present in a computing system. It has a common form of specification and notification of QoS, despite the fact that the resources have to be reserved by using different specification methods. Consequently, the resource provider is made transparent for the user, which interacts only with Quartz in order to obtain resources from the system.

Applications built on top of Quartz are able to have their QoS requirements enforced by using different supports without having to change the way their QoS requirements are specified. This is demonstrated by the remote copy example, which uses multiple reservation protocols at network level (i.e. ATM and RSVP), and by the telephone exchange application, which shows that the component responsible for the reservation of resources provided by the operating system can be exchanged in a similar way. This occurs because Quartz provides full transparency from the lower-level reservation protocols present at both network and operating system level in regard to the way they handle QoS reservations.

### 7.2.3 Heterogeneity

The application examples show that Quartz can be used in conjunction with different reservation protocols on different platforms. The differences in the lower-level support are handled by Quartz without requiring the application to get involved. The application code, in addition, does not have to be modified when the resource reservation protocol present in the underlying system is replaced by a different one.

Despite the limited range of lower-level resource reservation mechanisms covered by the prototype, we have deliberately chosen resource reservation mechanisms that are very different in nature so that we can assume that other system filters and agents can be written in order to support other mechanisms for resource reservation.

Extension of the set of platforms supported is not a very complex task. Support for new reservation mechanisms can be added to Quartz by writing a system agent and filter. Like filters, simple system agents such as the ATM agent and the Windows NT agent have about 300 lines of code and can be implemented and tested in no more than one working day by a programmer that is familiar with the reservation mechanisms provided by the reservation protocol. Protocols with more complex APIs, however, can take longer to implement and test. The RSVP agent, for example, has about 1000 lines of code and we estimate that it would take about a week for a programmer to rewrite it.

Similarly, different application areas have been covered by the examples implemented on top of Quartz. Support for different notions of QoS present at different areas of application was added by writing new application filters and then plugging these filters into the QoS Agent. As a result, we can assume that other different areas of application can be supported by Quartz without involving a high degree of complexity.

### 7.2.4 Support for Adaptation

Adaptation is supported by Quartz at two levels:

- Adaptation at application level; and
- Transparent adaptation at system level.

Adaptation is initiated every time the allocation of resources changes at a lower level. However, depending on the level at which it is performed by Quartz, adaptation has a different effect from the application's point of view.

Adaptation at system level occurs by rearranging the allocation of resources used by the application, so that the same QoS requirements are enforced by using a different set of resources. This process is transparent for the application, since the QoS seen at application level remains the same. This process is successful only when it is possible to rearrange the balance of resources so that the QoS requirements specified by the user are fulfilled.

Adaptation at application level is initiated when adaptation at system level fails. In this case, Quartz requests the application to perform adaptation by sending a notification message to the application in the form of an upcall. The notification message carries parameters understood at the application level, which are obtained by using the reverse path of translation provided by the translation unit. Whenever an application receives a request for adaptation, it has to adapt its QoS requirements and issue a new request for QoS.

By incorporating these two mechanisms for adaptation, Quartz provides a robust support for applications running on platforms in which the state of the resources might change dynamically. Some of the resource reservation protocols available at the moment allow changes in the state of resources to happen at run-time, and consequently it is important to make the QoS architecture able to handle this situation.

## 7.3 Comparison with other QoS Architectures

Most of the QoS architectures found in the literature and presented in Chapter 3, including Xbind [88] and QUANTA [43], are basically network-oriented (and mostly ATM-oriented) architectures, i.e., they do not cover system-level QoS constraints.

The matter of translation and mapping is very simplified in Xbind and QUANTA because QoS is handled at network level only. In addition, the tight integration of the architecture with the communication system that occurs in Xbind and also in QoS-A [24] constrains the flexibility and limits the extensibility of the architecture. On the other hand, translation and mapping have a special role in Quartz, which is able to cover a wider range of QoS requirements specified at a higher level of abstraction.

QUANTA and ERDoS [26][27] adopt translation mechanisms similar to the one proposed in this thesis, but there are a few differences that can be observed. First, the introduction of two sets of generic QoS parameters with a translator from one generic set into the other reduces the work done by QoS filters, which can be added by the application programmer and are easy to implement due to their low complexity. On the other hand, QUANTA translators, equivalent to our system filters, involve a more complex translation procedure and are not accessible to the user. As a consequence, support for new areas of application or network protocols would require changes in the code of the middleware. ERDoS incorporates its translation mechanisms into the resource and application agents, which can be written by the user as in Quartz. However, due to the adoption of an approach for provision of QoS based on monitoring and adaptation, QoS enforcement through resource reservation is not provided by ERDoS.

Like ERDoS, MMN [147] also uses adaptation mechanisms instead of resource reservation. This approach prevents the use of these architectures for implementing applications with strong QoS requirements due to the lack of support for service guarantees other than best-effort [52].

Researchers at Lancaster say that a single QoS manager such as the QoS Broker in the OMEGA architecture [101][102] 'requires a huge amount of mapping and management knowledge to support large-scale distributed applications, and that service management through a single entity is too centralised and severely inflexible' [148]. We don't incur this problem because per-application 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 system-specific agents, instead of a monolithic structure such as adopted by the QoS Broker. Issues regarding resource reservation are handled by the corresponding protocol associated to the sub-system that provides the resources, with system agents being responsible only for interfacing with these protocols.

QuO [156] and Arcade [42] are architectures that provide QoS languages for the specification of QoS requirements. We could simulate the same functionality by implementing application-level QoS filters that interpret the QoS languages used by QuO and Arcade and translate the QoS requirements expressed with this language into the generic application-level QoS parameters used by Quartz.

QuO is a network-oriented architecture that adds mechanisms for QoS specification to the client-server interaction model adopted by CORBA. Based on this approach, QoS requirements are specified in relation to method invocations issued between CORBA objects. However, the client-server interaction model is inappropriate for applications that use continuous media due to the necessity of modelling streams as a series of method invocations [37]. We believe that this approach is likely to impose limitations to the use of the QuO architecture.

Arcade is an example of a purely system-level QoS architecture based on a specific platform, in this case the Chorus kernel. Quartz could be easily made able to interact with this operating system by writing a system agent and a system filter that interface with the QoS mechanisms provided by Chorus. In addition, Quartz allows the user to establish network QoS constraints.

Table 23 is a new version of the table showed during the analysis of the state-of-the-art in the area of QoS architectures, which was presented in section 3.13. This new version of the table adds Quartz to the list of QoS architectures, and shows that our architecture has a number of features that is not matched by the other architectures found in the literature.

| | | Xbind | ReTINA | DIMMA | SUMO | TAO | Arcade | OMEGA | QoS – A | QuO | QUANTA | ERDoS | MMN | Quartz |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| QoS Specification | Multimedia Support | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | Real-Time Support | | ✓ | ✓ | ✓ | ✓ | | ✓ | | ✓ | ✓ | | | ✓ |
| | Extensible | | | | | | | | | | ✓ | ✓ | | ✓ |
| QoS Enforcement | Network Reservation | ✓ | ✓ | ✓ | | | | ✓ | ✓ | ✓ | ✓ | | | ✓ |
| | O.S. Reservation | | | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | | | ✓ |
| | Transparent | | | | | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Support for Heterogeneity | Multiple Protocols | | | | | | | | | | ✓ | ✓ | ✓ | ✓ |
| | Extensible | | | | | | | | | | | ✓ | ✓ | ✓ |
| Support for Adaptation | Notification | | | | | | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | Transparent | | | | | | | | ✓ | | | | ✓ | ✓ |

**Table 23 – Comparison of Quartz with other QoS Architectures**

Quartz is suitable for open systems due to its capacity to interact with multiple reservation protocols at both network and operating system level. In addition, Quartz provides support for QoS specification and enforcement to applications situated in different application areas, and is able to handle resource adaptation either transparently or by notifying the application. Furthermore, Quartz can be easily extended by the application programmer in order to use other resource reservation protocols and to provide QoS specification mechanisms appropriate for different application areas.

## 7.4 Summary

This chapter showed that Quartz provides an efficient service for applications requiring QoS specification and enforcement in open distributed systems. Different areas of application and underlying systems are covered by Quartz, as shown by the validation tests implemented on top of the prototype of the architecture.

This chapter has also shown that Quartz provides a number of improvements when compared to other architectures and frameworks for distributed multimedia that are found in the literature.

# 8

# Conclusions

The research project presented in this thesis has resulted in the design of an innovative QoS architecture whose properties make it useful in an open environment where applications require QoS-aware services to be provided by the underlying system.

This chapter summarises the characteristics of the Quartz architecture that make it particularly useful in open systems. In addition, it presents the achievements of this research project in the light of other work in this area of research, and suggests some future developments making use of the proposed QoS architecture.

## 8.1 Background

Analysing the requirements of distributed applications, researchers have found that an important class of applications is not satisfied with the best-effort approach to resource management offered by the computing platforms available today. Instead, these applications impose more strict requirements on the services provided by the computing platform, not only from the local machine but also the communication network and other remote services. These requirements are called the quality of service (QoS) required by the application.

The necessity of providing support for QoS enforcement drove the research community to improve the support for resource reservation available in networks and operating systems. In essence, this effort aims to make access to the resources provided to applications behave in a more predictable fashion, avoiding the typical best-effort resource management policies that are provided by most platforms.

Despite the effort spent in the development of resource reservation protocols, there is still a gap between what the application sees at its level – i.e. its QoS requirements – and what is provided by the underlying system – i.e. protocols for reserving resources such as bandwidth, processing power, and so on. This gap has to be filled by a middleware layer in order to avoid application programmers having to deal with low-level concepts necessary for interacting with resource reservation protocols.

Research on middleware responsible for bridging the gap between applications and resource reservation protocols has resulted in proposals for what are now called 'QoS architectures'. Examining the proposals for QoS architectures present in the literature concerning their ability to provide support for the deployment of applications with QoS requirements in open systems, we have identified several significant problems:

- the way in which these architectures allow applications to specify their QoS requirements typically suits only a limited area of application, and often requires QoS to be expressed in a low-level format that is unsuitable for the application;
- requirements are often enforced either at network or operating system level, instead of both, and the underlying system is not always made transparent to the application, forcing it to deal with low-level issues;
- these architectures are usually closely tied to a particular platform, such as a particular networking infrastructure and/or operating system; and
- dynamic resource adaptation is not always taken into account by architectures, which ignore the fact that the state of resources can change at run-time in some platforms.

With these problems in mind, we have designed and implemented a novel architecture for QoS specification and enforcement. This architecture, called Quartz, addresses the limitations of previous proposals in this area.

## 8.2 The Quartz Architecture

Our main objective while developing Quartz was to design an architecture that would provide QoS-constrained services for a wide range of applications running in open, distributed and heterogeneous systems. The solution adopted to deal with this situation was to design Quartz in a way that would make it possible for the architecture to reconfigure itself internally in order to fulfil the particular needs of the surrounding environment. This was achieved by adopting a component-based design, with each component being responsible for a particular task in dealing with a particular area of application or with a specific resource reservation protocol.

In this design, an architectural core is populated with interchangeable components. Different components handle the different notions of QoS present at application level, allowing the architecture to understand the requirements of different applications. Similarly, other components are responsible for dealing with the different resource reservation mechanisms that may be present in the underlying system

Quartz allows applications to specify their QoS requirements by using QoS parameters appropriate for their particular application area. The parameters specified by applications are handled internally by Quartz and translated into a format suitable for performing resource reservations. After translating QoS parameters, Quartz enforces QoS by reserving resources provided by the underlying system. The reservation of resources is done by interacting with resource reservation protocols provided by the network and the operating system.

The implementation of the Quartz QoS architecture, together with a series of test applications that make use of the QoS mechanisms provided by Quartz, demonstrated that the requirements expected from the architecture prior to the design phase have been achieved. Validation tests have shown that Quartz allows QoS to be specified in high-level formats appropriate for different application areas; enforces QoS transparently from the application's point of view and at both network and operating system level; and is able to use multiple reservation protocols in order to enforce the QoS requirements of the application. Quartz also provides support for resource adaptation by either transparently adapting QoS or by notifying the application that it has to reduce its requirements. Moreover, we have verified through performance tests that the overhead caused by Quartz is very small compared to related system activities.

In our understanding, Quartz, due to its unique combination of features, consists in an important step forward in the design of architectures intended to bridge the gap between applications in need of QoS-constrained services and the mechanisms for resource reservation that are provided by underlying systems.

## 8.3 Contribution

The main advantages of Quartz are provided by its component-based architectural design, which makes it flexible and extensible and allows it to handle the different notions of QoS present at application level and the heterogeneity found at system level in open systems.

The interfaces used by the application to specify its QoS requirements can be adapted in order to recognise the different formats of QoS parameters that are appropriate for different application areas. The application interface, however, is independent of the reservation protocols present in the underlying system, making them transparent for the application. The provision of transparency from the lower-level resource reservation protocols simplifies the task of building applications with QoS requirements and significantly increases their portability.

Quartz is flexible enough to adapt itself to dynamic changes in the underlying system by replacing the components used internally by the architectural support. Whenever a new resource provider and an associated resource reservation protocol are added to the underlying system, a new component able to interact with the resource provider can be added to the architecture in order to perform resource reservations on behalf of applications. In this case, a component responsible for translating between the format used by the new reservation protocol to express QoS requirements and the generic QoS parameters used internally by Quartz must also be added to the translation unit of the architecture.

Changes at application level can be also accommodated by Quartz. In order to understand the different ways used by applications to express their QoS requirements, Quartz allows a translation component to be replaced. This component, the application filter, is responsible for translating between the QoS parameters used by the application and the generic QoS parameters used internally by Quartz.

New components can be selected from a library or, in cases where no appropriate component is available, they can be written by the application programmer. Since each component performs a very particular role in the overall architecture, the knowledge necessary for implementing them is limited. Independence between components, which communicate through a set of well-defined interfaces, ensures that components can be combined freely.

Transparent QoS enforcement is another feature provided by Quartz, since the architecture is designed in a way that the application is unaware of the mechanisms used for resource reservation at lower levels. The interaction with resource reservation mechanisms is performed by components of the architecture specialised for interacting with each of the resource reservation protocols present in the underlying system.

Another important feature supported by Quartz is dynamic resource adaptation. Adaptation is supported at both system and application level. The architecture provides mechanisms to receive notifications of any change in the QoS supported by the platform. Whenever changes occur, the architecture tries to perform adaptation at system level, rearranging the balance of resources in order to compensate for any losses resulting from the dynamic adaptation. When adaptation at system level is not feasible due to the impossibility of compensating for the resources lost, the architecture informs the application, which has to degrade its QoS requirements.

In addition, Quartz is able to enforce QoS at both network and system level, in comparison with other QoS architectures presented in the literature, which usually concentrate on only one of them.

## 8.4 Perspectives for Future Work

Since Quartz can be deployed on different platforms and can provide QoS for different areas of application, the possibilities of using the architecture are very diverse.

One of the possibilities for future work that we envisage for Quartz is the porting of the prototype to a wider variety of computing platforms. At operating system level, it would be interesting to add support for real-time operating systems such as Chorus [142], VxWorks [151] or QNX [62]. Support for new reservation protocols at network level, such as the Scalable Reservation Protocol (SRP) that is being developed by the IETF [1], could also be added in order to expand the platform coverage. This would be done by implementing new components of the architecture that interface with these reservation protocols.

Being a component-based and extensible architecture, Quartz can also be modified in order to implement new resource balancing algorithms and adaptation policies. In fact, Quartz can be used as a toolkit for testing new policies and algorithms by writing new components and plugging them into the architecture.

Another possibility for future work involving Quartz is in developing new applications that make use of the architecture for QoS specification and enforcement. We foresee several possibilities for Quartz and the Quartz/CORBA framework, more specifically in the fields of distributed multimedia, real-time applications, and telecommunications.

In the area of distributed multimedia, we envisage the application of the Quartz/CORBA framework in the scenarios described in Chapter 6, i.e. media broadcast, media on demand and videoconference.

In the field of real-time applications, two main areas are candidates to be explored: manufacturing control and avionics.

In the area of telecommunications, it would be interesting to test the behaviour of the telephone exchange example in a real telecommunications platform.

# References

[1] W. Almesberger, T. Ferrari and J.-Y. Le Boudec "Scalable Resource Reservation for the Internet", Internet Draft, November 1997.

[2] Apple Computer Inc. "QuickTime API Documentation", Software Documentation, 1997. Available at developer.apple.com.

[3] N. Audsley and A. Burns "Real Time System Scheduling", Specification and Design for Dependability, PDCS – Predictability Dependable Computer Systems – ESPRIT, 1st Year Report, LAAS, Toulouse, May 1990.

[4] Cristina Aurrecoechea, Andrew Campbell and Linda Hauw "A Survey of Quality of Service Architectures", Distributed Multimedia Research Group (MPG), University of Lancaster, Internal report number MPG-95-18.

[5] O. Babaoglu and A. Schiper "On Group Communication in Large-Scale Distributed Systems", Broadcast (ESPRIT Basic Research Project 6360) Technical Report No. 57, 1994.

[6] F. Baker, R. Guerin and D. Kandlur "Specification of Commited Rate Quality of Service", Internet Draft, June 1996.

[7] Sean Baker, Vinny Cahill and Paddy Nixon "Bridging Boundaries: CORBA in Perspective", IEEE Internet Computing, Vol. 1, No. 5, pp. 52-57, September/October 1997.

[8] Paul Barham “A Fresh Approach to File System Quality of Service”, Proceedings of the 7th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV’97), St. Louis, USA, May 1997.

[9] Tsipora Barzilai, Dilip Kandlur, Ashish Mehra, Debajan Saha and Steve Wise “Design and Implementation of an RSVP-based Quality of Service Architecture for Integrated Services Internet”, Proceedings of the 17th International Conference on Distributed Computer Systems (ICDCS’97), Baltimore, USA, May 1997.

[10] John Bates “CaberNet Report: The State of the Art in Distributed and Dependable Computing”, CaberNet Technical Report, June 1998.

[11] BEA Systems et al. “Notification Service”, OMG Document Telecom/98-01-01, January 1998.

[12] H.W.P. Beadle “Experiments in Multipoint Multimedia Telecommunication”, IEEE Multimedia, Vol. 2 No. 2, Summer 1995.

[13] Luc Bellissard, Slim Ben Atallah, Fabienne Boyer and Michel Riveill “Distributed Application Configuration”, INRIA Rhone-Alpes, Research Report RR-3119, February 1997.

[14] T. E. Bihari and P. Gopinath “Object-Oriented Real-Time Systems: Concepts and Examples”, IEEE Computer, Vol. 25, No. 12, pp. 25-32, December 1992.

[15] Manuel Billot, Valérie Issarny, Isabelle Puaut, Michel Banâtre “A Proposal for Ensuring High Availability of Distributed Multimedia Applications”, Proceedings of the 15th International Symposium on Reliable Distributed Systems, Canada, October 1996.

[16] U. Black “ATM: Foundation for Broadband Networks”, Prentice Hall, 1995.

[17] Douglas Boling “Programming Microsoft Windows CE”, Microsoft Press, 1998.

[18] Pier Giorgio Bosco, Eirik Dahle, Michel Gien, Andrew Grace, Nicolas Mercouroff, Nathalie Perdigues, Jean-Bernard Stefani “The ReTINA Project: an Overview”, Chorus Systemes Technical Document RT/TR-96-15, May 1996.

[19] R. Braden, I. Zhang, S. Berson, S. Herzog, and S. Jamin "Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification", Internet Enginnering Task Force, RFC 2205, September 1997.

[20] Torsten Braun "Internet Protocols for Multimedia Communications", IEEE Multimedia, Vol. 4, No. 3, pp. 85-90, July-September 1997.

[21] C. Brazdziunas "IPng Support for ATM Services", IETF RFC 1680, August 1994.

[22] G. Booch, J. Rumbaugh and I. Jacobson "The Unified Modelling Language User Guide" , Addison-Wesley, 1998.

[23] A. Burns and A. Wellings "Real-Time Systems and Programming Languages", Second Edition, Addison-Wesley, 1997.

[24] Andrew Campbell, Geoff Coulson and David Hutchison "A Quality of Service Architecture", ACM Computer Communications Review, Vol. 24, No. 2, pp. 6-27, April 1994. MPG Internal report number MPG-94-08.

[25] Andrew Campbell and Geoff Coulson "A QoS adaptive transport system: design, implementation and experience", ACM Multimedia'96, pp. 117-127, November 1996.

[26] Saurav Chateerjee, Bikash Sabata and Jaroslav J. Sydir "ERDOS QoS Architecture", SRI International, Technical Report, May 1998.

[27] Sarauv Chatterjee, Michael Brown and Bikash Sabata "Graceful Adaptation of Distributed Soft Real-Time Applications", Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS'98), Work in Progress session, Madrid, Spain, December 1998.

[28] Shigang Chen "Quality of Service in Heterogeneous Environments", M.Sc. Thesis, Engineering College, University of Illinois at Urbana-Champaign, 1997.

[29] S. C. Cheng and J. A. Stankovic "Scheduling Algorithms for Real-Time Systems – A Brief Survey", Hard Real-Time Systems : Tutorial, pp. 150-173, IEEE Computer Society Press, USA, July 1988.

[30] Michael Clarke, Tom Fitzpatrick and Geoff Coulson "Adaptive System Support for Multimedia in Mobile End-Systems", Proceedings of the 3rd Communications Networks Symposium, Manchester, U.K., July 1996. MPG Internal report number MPG-96-24.

[31] CNET – France Télécom R&D Center "Requirements for a Real-time ORB", ReTINA Technical Report 96-08 (also Chorus Systemes Technical Document RT/TR-96-08), June 1996.

[32] R. Cole, D. Shur and C. Villamizar "IP over ATM: A Framework Document", IETF RFC 1932, April 1996.

[33] George Colouris and Jean Dollimore "Distributed Systems – Concepts and Design", Addison-Wesley Pub. Co., USA, 1988.

[34] A. Conta and S. Deering "Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification", Internet Enginnering Task Force, RFC 1885, December 1995.

[35] G. Coulson, and G.S. Blair, "Meeting the Real-time Synchronisation Requirements of Multimedia in Open Distributed Systems", Distributed Systems Engineering Journal (DSEJ), Vol. 1, No 1, pp. 135-144, 1994.

[36] G. Coulson, G.S. Blair, J.B. Stefani, F. Horn, and L. Hazard "Supporting the Real-time Requirements of Continuous Media in Open Distributed Processing", Computer Networks and ISDN Systems, Vol. 27, No. 8, July 1995.

[37] G. Coulson and D. Waddington "A CORBA Compliant Real-Time Multimedia Platform for Broadband Networks", Proceedings of the International Workshop on Trends in Distributed Systems (TreDS), Lecture Notes in Computer Science Volume 1161, Aachen, Germany, October 1996.

[38] Terry A. Critchley and K. C. Batty "Open Systems: The Reality", Prentice-Hall, New York, 1993.

[39] Frederic Dang Tran, Victor Perebaskine, Jean-Bernard Stefani. Ben, Crawford, Andre Kramer and Dave Otway "Binding and Streams: the ReTINA Approach", Chorus Systemes Technical Document RT/TR-96-16, ReTINA Technical Report 96-16, March 1996.

[40] S. Deering and R. Hinden “Internet Protocol, Version 6 (IPv6) Specification”, Internet Enginnering Task Force, RFC 1883, December 1995.

[41] L. Delgrossi and L. Berger “Internet Stream Protocol Version 2 (ST2) Protocol Specification – Version ST2+”, Internet Enginnering Task Force, RFC 1819, August 1995.

[42] Isabelle Demeure, Jocelyne Farhat, and F. Gasperoni “A Scheduling Framework for the Automatic Support of Temporal QoS Constraints”, Proceedings of the Fourth International Workshop on Quality of Service, Paris, March 1996.

[43] Sudheer Dharanikota and Kurt Maly “QUANTA: Quality of Service Architecture for Native TCP/IP over ATM networks”, Old Dominion University, Department of Computer Science, Technical Report TR-96-01, February 1996.

[44] John Matthew Simon Doar “Multicast in the Asynchronous Transfer Mode Environment”, PhD Dissertation, St. John’s College, University of Cambridge, January 1993.

[45] Manek Dubash, Rupert Goodwins and Alan Stevens “The PC Magazine Guide to Voice and Data Integration”, PC Magazine, November 1998.

[46] Margaret A. Ellis and Bjarne Stroustrup “The Annotated C++ Reference Manual”, Addison-Wesley Pub. Co., USA, 1990.

[47] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. “Exokernel: An Operating System Architecture for Application-Level Resource Management”, Proceedings of the Fifteenth Symposium on Operating Systems Principles, December 1995.

[48] John E. Flood “Telecommunications Switching, Traffic and Networks”, Prentice Hall International (UK) Limited, 1995.

[49] J. A. Friesen, C. L. Yang, and R. E. Cline, Jr. “DAVE: A Plug-and-Play Model for Distributed Multimedia Application Development”, IEEE Parallel and Distributed Technology, Summer 1995.

[50] B. Fuhrt “Multimedia Systems – An Overview”, IEEE Multimedia, Vol. 1 No. 1, Spring 1994.

[51] Francisco Garcia, David Hutchison, Andreas Mauthe and Nicholas Yeadon “QoS Support for Distributed Multimedia Communications”, Proceedings of the 1st International Conference on Distributed Platforms, Dresden, Germany, 27 February-1 March 1996. MPG Internal report number MPG-96-08.

[52] Jan Gecsei “Adaptation in Distributed Multimedia Systems”, IEEE Multimedia, Vol. 4, No. 2, April-June 1997.

[53] S. J. Gibbs and D. C. Tsichritzis “Multimedia Programming”, ACM Press – Addison-Wesley, 1995.

[54] Richard Golding and John Wilkes “Persistent Storage for Distributed Applications”, Proceeding of the Eighth ACM SIGOPS European Workshop, Sintra, Portugal, September 1998.

[55] Sreenivas Gollapudi “A Multithreaded Client-Server Architecture for Distributed Multimedia Systems”, M.Sc. Thesis, Department of Computer Science, University of New York at Buffalo, July 1996.

[56] W. J. Goralski “Introduction to ATM Networking”, McGraw-Hill, 1995.

[57] Amit Gupta “Multi-party Real-Time Communication in Computer Networks”, Ph.D. Dissertation, Computer Science Division, University of California at Berkeley, February 1996.

[58] Timothy H. Harrison, David L. Levine and Douglas C. Schmidt “The Design and Performance of a Real-time CORBA Event Service”, Proceedings of the 12th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA’97), Atlanta, USA, October 1997.

[59] H. Hartig et al. “DROPS: OS Support for Distributed Multimedia Applications”, Desdresn University of Technology, Department of Computer Science, Internal Report, February 1998.

[60] Mark Hayter and Derek McAuley “The Desk Area Network”, Cambridge University Computer Laboratory, Technical Report 228, May 1991.

[61] A. Herbert “CORBA Extensions for Real-Time and Interactive Multimedia”, APM/ANSA External Paper 1311.02, Cambridge, UK, October 1994.

[62] Dan Hildebrand "An Architectural Overview of QNX", QNX Software Systems White Paper, available at www.qnx.com.

[63] M. T. Hills "Telecommunications Switching Principles", George Allen & Unwin Limited, 1979.

[64] Robert M. Hinden "IP Next Generation Overview" Sun Microsystems Technical Report, May 1995. Available at playground.sun.com.

[65] Sungjune Hong, Youngjae Kim and Sunyoung Han "Real-Time Inter-ORB Protocol on Distributed Environment", Proceedings of the 1st IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'98), Kyoto, Japan, April 1998.

[66] J.-F. Huard and A.A. Lazar, "On QOS Mapping in Multimedia Networks", Proceedings of the 21th IEEE International Computer Software and Application Conference *(COMPSAC '97)*, Washington, D.C., USA, August 1997.

[67] Interactive Multimedia Association (IMA) "IMA Recommended Practice – Multimedia Systems Service", Second Draft, September 1994.

[68] International Organization for Standardization (ISO) "Information Technology – Open Systems Interconnection – Basic Reference Model: The Basic Model", ISO/IEC 7498-1 | ITU-T Recommendation X.200, July 1994.

[69] International Organization for Standardization (ISO) "Information Technology – Open Distributed Processing – Reference Model: Overview", ISO/IEC 10746-1, 1998.

[70] International Organization for Standardization (ISO) "Information Technology – Quality of Service – Framework", ISO/IEC 13236, 1998.

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

[72] Iona Technologies "OrbixTalk White Paper", April 1996.

[73] Iona Technologies "Orbix Programmer's Guide" and "Orbix Reference Guide", October 1997.

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

[75] Iona Technologies “Orbix MX – A Distributed Object Framework for Telecommunication Service Development and Deployment” Iona Technologies White Paper, April 1998.

[76] Valérie Issarny, Christophe Bidan, Frédéric Leleu, Titos Saridakis “ Towards Specifying QoS-Enabling Software Architectures”, Proceedings of the Fifth IFIP International Workshop on Quality of Service (IWQOS'97), New York, USA, May 1997.

[77] R. Joseph “MHEG-5: An Overview”, GMD Focus Technical Report, December 1995.

[78] Thomas V. Johnson and Aidong Zhang “A Framework for Supporting Quality-Based Presentations of Continuous Multimedia Streams”, Department of Computer Science, State University of New York at Buffalo, 1996.

[79] J.-I. Jung “Quality of Service in Telecommunications (Part I & II)”, IEEE Communications Magazine, Vol 34, No 8, pp-108-117, August 1996.

[80] Krishna Kavi, James C. Browne, Anand Triphati “Computer Systems Research: The Pressure is On”, IEEE Computer, Vol. 32, No. 1, pp. 30-39, January 1999.

[81] T. Kanezuka, and M. Takizawa “Quality-based Flexibility in Distributed Objects”, Proceedings of the 1st IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC’98), Kyoto, Japan, April 1998.

[82] H. Knoche and H. de Meer “Quantitative QoS-Mapping: A Unifying Approach”, in “Building QoS into Distributed Systems – IFIP TC6 WG6.1 Fifth International Workshop on Quality of Service (IWQoS’97)”, Andrew Campbell and Klara Nahrstedt (Ed.), Chapman & Hall, May 1997.

[83] H. Kopetz “Scheduling”, An Advanced Course on Distributed Systems, Estoril, Portugal, 1992.

[84] H. Kopetz, G. Fohler, G. Gruensteidl, H. Kantz, G. Pospischil, P. Puschner, J. Reisinger, R. Schlatterbeck, W. Schuetz, A. Vrchoticky and R. Zainlinger "Real-Time System Development: The Programming Model of MARS", Proceedings of the International Symposium on Autonomous Decentralized Systems, p.190-199, Kawasaki, Japan, March 1993.

[85] Christoforos E. Kozyrakis and David A. Peterson "A New Direction for Computer Architecture Research", IEEE Computer, Vol. 31, No. 11, November 1998.

[86] A. Kramer "ANSAworks: Multi-Media Streams and CORBA", ANSA Technical Report APM.1757.01, April 1996.

[87] M. Laubach "Classical IP and ARP over ATM", Internet Enginnering Task Force, RFC 1577, January 1994.

[88] A. A. Lazar, K. S. Lim, and F. Marconcini "Realizing a Foundation for Programmability of ATM Networks with the Binding Architecture", IEEE Journal on Selected Areas in Communications, No. 7, pp. 1214-1227, September 1996.

[89] I. Leslie et al. "The Design and Implementation of an Operating System to Support Distributed Multimedia Applications", IEEE Journal of Selected Areas in Communcations, September 1996.

[90] G. Li "Supporting Distributed Real-Time Computing", PhD Thesis, University of Cambridge Computer Laboratory, Technical Report 322, August 1993.

[91] G. Li "Distributed Real-Time Objects : the ANSA Approach", 1st IEEE Workshop on Object-Oriented Real-Time Dependable Systems, Dana Point, CA, USA, September 1994.

[92] C. Lin, H. Chu and K. Nahrstedt "A Soft Real-Time Scheduling Server on the Windows NT", 2nd USENIX Windows NT Symposium, Seattle, WA, USA, August, 1998.

[93] Thomas D. C. Little and Dinesh Venkatesh "Prospects for Interactive Video on Demand" IEEE Multimedia, Vol. 1 No. 3, Fall 1994.

[94] S. Lu, K.-W. Lee and V. Bharghavan "Adaptive Service in Mobile Computing Environments", in "Building QoS into Distributed Systems – IFIP TC6 WG6.1 Fifth International Workshop on Quality of Service (IWQoS'97)", Andrew Campbell and Klara Nahrstedt (Ed.), Chapman & Hall, May 1997.

[95] T. Meyer-Boudnik and W. Effelsberg "MHEG Explained", IEEE Multimedia, Vol. 2 No. 1, Spring 1995.

[96] Microsoft Corporation "Real-time Systems and Microsoft Windows NT", White Paper, June 1995. Available at www.microsoft.com.

[97] Microsoft Corporation "DCOM: A Technical Overview", White Paper, 1996. Available at www.microsoft.com.

[98] Microsoft Corporation "DirectX 3 Software Development Kit", Software Documentation, 1996. Available at www.microsoft.com.

[99] R. F. Mines, J. A. Friesen, and C. L. Yang "DAVE: A Plug and Play Model for Distributed Multimedia Application Development", Proceedings of the ACM Multimedia 94 Conference (also Sandia Report SAND94-8233, July 1994), ACM Press, pp. 59-66, New York, 1994.

[100] Scott Mitchell, Hani Naguib, George Colouris and Tim Kindberg "Dinamically Reconfiguring Multimedia Components: A Model-based Approach", Proceeding of the Eighth ACM SIGOPS European Workshop, Sintra, Portugal, September 1998.

[101] Klara Nahrstedt, Jonathan M. Smith, "Design, Implementation and Experiences of the OMEGA Architecture", Internal Report MS-CIS-95-22, University of Pennsylvania, May 1995.

[102] Klara Nahrstedt, Jonathan M. Smith "The QoS Broker", IEEE Multimedia, Vol.2, No.1, pp. 53-67, Spring 1995.

[103] Klara Nahrstedt, Ralf Steinmetz "Resource Management in Multimedia Networked Systems", IEEE Computer, pp. 52-64, May 1995.

[104] Klara Nahrstedt "Experiences with QoS Brokerage and Enforcement", 2nd International Conference on Multimedia Information Systems (ICMIS97), Chicago, IL, April, 1997.

[105] Object Management Group "The Object Management Architecture Guide", OMG Technical Document 92-11-1, September 1992.

[106] Object Management Group "IDL C++ Language Mapping Specification", OMG Document Number 94-9-14, September 1994.

[107] Object Management Group "CORBAservices: Common Object Services Specification", OMG Technical Document, March 1995 (Revised in November 1996).

[108] Object Management Group "CORBA 2.0 Specification", OMG Technical Document 96-03-04, USA, March 1996.

[109] Object Management Group – Telecommunication Domain Task Force "Control and Management of A/V Streams – Request for Proposals", OMG Technical Document Telecom/96-08-01, August 1996.

[110] Object Management Group's Object Request Broker / Object Services Task Force "Realtime Technologies – Request for Information", OMG Technical Document ORBOS/96-09-02, October 1996.

[111] David O'Flanagan "An Implementation of the CORBA Audio/Video Streaming Service using RSVP", M.Sc. Dissertation, University of Dublin – Trinity College, Department of Computer Science, Technical Report TCD-CS-1999-32, October 1998.

[112] H. Okamura, Y. Ishikawa and M. Tokoro "Toward Building the Flexible Systems on the Distributed Environment", Proceedings of the OOPSLA'91 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming, October 1991.

[113] R. O. Onvural "Asynchronous Transfer Mode Networks – Performance Issues", Artech House, Second Edition, 1995.

[114] Open Software Foundation "Distributed Computing Environment Overview", OSF White Paper, OSF-DCE-PD-1090-4, 1992.

[115] Maximilian Ott, Daniel Reininger, Wenjun Luo "Adaptive and Scalable QoS for Multimedia using Hierarchical Contracts", Proceedings of ACM Multimedia'96, Boston, USA, November 1996.

[116] D. Otway "DIMMA Overview", ANSA Tech. Report APM.1439.02, April 1995.

[117] Palm Computing "Palm OS Reference", Software Documentation, 1999. Available at www.palm.com

[118] F. Panzieri and M. Rocetti "Reliable Synchronization Support and Group-Membership Services for Distributed Multimedia Applications", Broadcast (ESPRIT Basic Research Project 6360) Technical Report No. 90, 1995.

[119] Bjorn Pehrson, Per Gunningberg and Stephen Pink "Distributed Multimedia Applications on Gigabit Networks", IEEE Network Magazine, pp. 26-35, January 1992.

[120] Stephen M. Platt "What Makes a Faster Processor?", Byte Magazine, March 1999. Available at www.byte.com.

[121] Ian Pratt "The User-Safe Device I/O Architecture", Ph.D. dissertation, King's College, University of Cambridge, August 1997.

[122] K. Ramamritham and J. A. Stankovic, "Scheduling Algorithms and Operating Systems Support for Real-Time Systems", Proceedings of the IEEE, Vol. 82, No. 1, pp. 55-67, January 1994.

[123] F. Reynolds, R. Clark and F. Travostino "Evolution of a Distributed, Real-Time, Object-Based Operating System", Proceedings of the 1st IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'98), Kyoto, Japan, April 1998.

[124] R. Rooholamini and V. Cherkassky "ATM-Based Multimedia Servers", IEEE Multimedia, Vol. 2 No. 1, Spring 1995.

[125] Debajan Saha "Supporting Distributed Multimedia Applications on ATM Networks", Ph.D. Dissertation, Department of Computer Science, University of Maryland, 1995.

[126] Richard E. Schantz "Quality of Service", in "Encyclopedia of Distributed Computing", Partha Dasgupta and Joseph Urban (Eds.), Kluwer Academic Publishers, 1998.

[127] Douglas C. Schmidt, Tim Harrison and Ehab Al-Shaer "Object-Oriented Components for High-speed Network Programming", Proceedings of the 1st USENIX Conference on Object-Oriented Technologies (COOTS'95), Monterey, USA, June 1995.

[128] Douglas C. Schmidt and A. Gokhale "The Performance of the CORBA Dynamic Invocation Interface and Dynamic Skeleton Interface over High-Speed ATM Networks", Proceedings of the IEEE Globecom'96 Conference, London, England, November 1996.

[129] Douglas C. Schmidt, Aniruddha Gokhale, Timothy H. Harrison and Guru Parulkar "A High Performance Endsystem Architecture for Real-Time CORBA", IEEE Communications Magazine, Vol. 14, No. 2, February 1997.

[130] Douglas C. Schmidt, David Levine and Sumedh Mungee "The Design of the TAO Real-Time Object Request Broker", Computer Communications, Elsevier Science, Vol. 21, No. 4, April, 1998.

[131] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson "RTP: A Transport Protocol for Real-Time Applications", Internet Enginnering Task Force, RFC 1889, January 1996.

[132] H. Schulzrinne, A. Rao and R. Lamphier "Real-Time Streaming Protocol (RTSP)", Internet Enginnering Task Force, Internet Draft, November 1996.

[133] John Segrave-Daly "The Design and Implementation of a Distributed Music Rehearsal Studio Application", University of Dublin – Trinity College, Department of Computer Science, Final Year Project, June 1999.

[134] S. Shenker, C. Partridge and R. Guerin "Specification of Guaranteed Quality of Service", Internet Enginnering Task Force, RFC 2212, September 1997.

[135] S. Shenker, J. Wroclawski "General Characterization Parameters for Integrated Service Network Elements", Internet Enginnering Task Force, RFC 2215, September 1997.

[136] Richard A. Staehli "Quality of Service Specification for Resource Management in Multimedia Systems", Ph.D. Dissertation, Oregon Graduate Institute of Science and Technology, January 1996.

[137] Stardust Technologies “Windows Sockets 2 Application Programming Interface, White Paper, Revision 2.2.0, May 1996.

[138] R. Steinmetz “Analyzing the Multimedia Operating System”, IEEE Multimedia, Vol. 2 No. 1, Spring 1995.

[139] R. Steinmetz and L. C. Wolf “Quality of Service: Where are we?”, in “Building QoS into Distributed Systems – IFIP TC6 WG6.1 Fifth International Workshop on Quality of Service (IWQoS’97)”, Andrew Campbell and Klara Nahrstedt (Ed.), Chapman & Hall, May 1997.

[140] E. Stoica, H. Abdel-Wahab and K. Maly “Synchronization of Multimedia Streams in Distributed Environments”, Old Dominion University, Department of Computer Science, Technical Report 97-19, February 1997.

[141] H.J. Stuttgen “Network Evolution and Multimedia Communication”, IEEE Multimedia, Vol. 2 No. 3, Fall 1995.

[142] Sun Microsystems “Sun Embedded Telecom Platform – Combining the Power of Solaris Computing with the Real-Time Performance of the ChorusOS Operating System”, Sun White Paper, available at www.sun.com.

[143] Jaroslaw J. Sydir, Sarauv Chatterjee, Bikash Sabata “Providing End-to-End QoS Assurances in a CORBA-Based System”, Proceedings of the 1st IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC’98), Kyoto, Japan, April 1998.

[144] Martin Timmerman and Jean-Christophe Monfret “Windows NT as Real-Time OS”, Real-Time Magazine, Issue 1997/2, April 1997.

[145] Hideyuki Tokuda, Tatsuo Nakajima and Prithvi Rao “Real-Time Mach: Towards a Predictable Real-Time System”, Proceedings of USENIX Mach Workshop, October 1990.

[146] A. Vogel, B. Kerherve, G. von Bachmannn, and J. Gecsei “Distributed Multimedia and QoS: A Survey”, IEEE Multimedia, Vol. 2 No. 2, Summer 1995.

[147] D. Waddington, G. Coulson and D. Hutchinson, "Specifying QoS for Multimedia Communications within a Distributed Programming Environment", Proceedings of the 3rd International COST237 Workshop: Multimedia Telecommunications and Applications, Lecture Notes in Computer Science Volume 1185, pp104-130, Barcelona, Spain, November 1996. MPG Internal report number MPG-96-34.

[148] Daniel Waddington, Christopher Edwards and David Hutchison "Resource Management for Distributed Multimedia Applications", Proceedings of the Second European Conference on Multimedia Applications, Services and Techniques; Milan, Italy, May 21-23 1997; MPG Internal report number MPG-97-10.

[149] Daniel Waddington and David Hutchison "A General Model for QoS Adaptation", MPG Internal report number MPG-98-09, May 1998.

[150] Daniel Waddington and David Hutchison "End-to-End QoS Provisioning through Resource Adaptation", MPG Internal report number MPG-98-10, May 1998.

[151] Wind River Systems "The Next Generation of Embedded Development Tools", White Paper, available at www.wrs.com.

[152] J. Wroclawski "The Use of RSVP with IETF Integrated Services", Internet Enginnering Task Force, RFC 2210, September 1997.

[153] J. Wroclawski "Specification of the Controlled-Load Network Element Service", Internet Enginnering Task Force, RFC 2211, September 1997.

[154] Nicholas Yeadon, Andreas Mauthe, Francisco Garcia and David Hutchison "QoS Filters: Addressing the Heterogeneity Gap", Proceedings of the European Workshop on Interactive Distributed Multimedia Systems and Services (IDMS 96), Berlin, Germany, March 1996. MPG Internal report number MPG-96-09.

[155] Nicholas Yeadon, Francisco Garcia, David Hutchison and Doug Shepherd "Filters: QoS Support Mechanisms for Multipeer Communications", IEEE Journal on Selected Areas in Computing (JSAC) special issue on Distributed Multimedia Systems and Technology, Volume 14, Number 7, September 1996, pp1245-1262. MPG Internal report number MPG-96-31.

[156] John A. Zinky, David E. Bakken, Richard E. Schantz “Architectural Support for Quality of Service for CORBA Objects, Theory and Practice of Object Systems, Vol. 3 No. 1, January 1997.

# Appendix

This appendix contains the performance measurements that were obtained with the remote copy example, which was presented in section 7.1.1. This data was used to obtain the values for the overhead caused by the Quartz architecture that was analysed in section 7.1.5.

## A.1 ATM

The table below shows the performance measurements obtained with the remote copy application acting as a daemon and using the ATM protocol for transferring a 45Kb file over the network. Values shown are average times per connection, in milliseconds, obtained for a sequence of 100 connection establishments and QoS reservations.

| Total Time (Function+Child) | Effective Function Time | Hit Count | Function Name |
|---|---|---|---|
| 3.33 | 3.33 | 1 | _closesocket@4 |
| 6.214 | 6.214 | 1 | _gethostbyname@4 |
| 33.988 | 33.988 | 1 | _gethostname@8 |
| 0.01 | 0.01 | 1 | _inet_ntoa@4 |
| 11.817 | 11.817 | 47 | _send@16 |
| 26752.07 | 0.002 | 1 | _WinMain@16 |
| 0.117 | 0.117 | 1 | _WSAAsyncSelect@16 |
| 0.008 | 0.008 | 1 | _WSACleanup@0 |
| 0.531 | 0.531 | 1 | _WSAConnect@28 |
| 0.503 | 0.503 | 1 | _WSAConnect@28 w/o QoS |
| 0.341 | 0.341 | 2 | _WSAEnumProtocolsA@12 |
| 0.001 | 0.001 | 1 | _WSAGetLastError@0 |
| 0.132 | 0.132 | 1 | _WSAIoctl@36 |
| 1.127 | 1.127 | 1 | _WSALookupServiceBeginW@12 |
| 0.134 | 0.134 | 1 | _WSALookupServiceEnd@4 |
| 0.717 | 0.717 | 1 | _WSALookupServiceNextW@16 |
| 17.782 | 17.782 | 1 | _WSASocketA@24 |
| 7.622 | 7.622 | 2 | _WSAStartup@8 |
| 0.007 | 0.007 | 1 | AfxGetApp() |
| 0.013 | 0.013 | 1 | AfxGetModuleState() |
| 0.05 | 0.037 | 1 | AfxInitialize() |
| 26752.07 | 19.104 | 1 | AfxWinMain() |
| 0.01 | 0.01 | 3 | CComboBox::~CComboBox() |
| 0.045 | 0.045 | 3 | CComboBox::CComboBox() |
| 0.919 | 0.919 | 3 | CComboBox::SetCurSel() |
| 0.002 | 0.002 | 1 | CDialog::~CDialog() |
| 4.429 | 4.429 | 425 | CDialog::AssertValid() |
| 0.037 | 0.037 | 1 | CDialog::CDialog() |
| 0.045 | 0.045 | 1 | CDialog::CheckAutoCenter() |
| 26730.39 | 78.174 | 1 | CDialog::DoModal() |
| 0.326 | 0.326 | 426 | CDialog::GetRuntimeClass() |
| 1.651 | 1.478 | 1 | CDialog::OnCancel() |
| 41.222 | 1.38 | 125 | CDialog::OnCmdMsg() |
| 4.818 | 1.016 | 1 | CDialog::OnInitDialog() |
| 0.002 | 0.002 | 1 | CDialog::OnSetFont() |
| 0 | 0 | 1 | CDialog::PreInitDialog() |
| 73.325 | 25.569 | 184 | CDialog::PreTranslateMessage() |
| 0.012 | 0.012 | 7 | CEdit::~CEdit() |
| 0.169 | 0.169 | 7 | CEdit::CEdit() |

| | | | |
|---|---|---|---|
| 222.494 | 197.991 | 104 | CEdit::LineScroll() |
| 0.004 | 0.004 | 1 | CFile::~CFile() |
| 0.003 | 0.003 | 1 | CFile::CFile() |
| 0.191 | 0.191 | 47 | CFileException::~CFileException() |
| 0.555 | 0.555 | 47 | CFileException::CFileException() |
| 0.108 | 0.108 | 2 | CMenu::AppendMenuA() |
| 0.08 | 0.001 | 1 | CSampleApp::~CSampleApp() |
| 0.184 | 0 | 1 | CSampleApp::CSampleApp() |
| 26732.87 | 0.005 | 1 | CSampleApp::InitInstance() |
| 0 | 0 | 340 | CSampleDlg::_GetBaseMessageMap() |
| 0.23 | 0.063 | 1 | CSampleDlg::~CSampleDlg() |
| 34.333 | 0.732 | 1 | CSampleDlg::ATMSender() |
| 2.245 | 0.005 | 1 | CSampleDlg::CSampleDlg() |
| 5.697 | 0.021 | 2 | CSampleDlg::DoDataExchange() |
| 40.965 | 0.023 | 1 | CSampleDlg::FillLocalAddress() |
| 0.52 | 0.179 | 1 | CSampleDlg::FindProtocols() |
| 0.536 | 0.536 | 1143 | CSampleDlg::GetMessageMap() |
| 7.612 | 0.004 | 1 | CSampleDlg::InitWS2() |
| 5.149 | 0.028 | 1 | CSampleDlg::OnAsyncSelectATM() |
| 0 | 0 | 1 | CSampleDlg::OnCancel() |
| 37.914 | 1.235 | 1 | CSampleDlg::OnConnect() |
| 58.686 | 0.037 | 1 | CSampleDlg::OnInitDialog() |
| 0.23 | 0.006 | 2 | CSampleDlg::OnPaint() |
| 3302.541 | 3003.988 | 47 | CSampleDlg::OnTimer() |
| 299.147 | 0.225 | 52 | CSampleDlg::PrintStatus() |
| 0.246 | 0.246 | 13 | CString::~CString() |
| 0.006 | 0.006 | 8 | CString::CString() |
| 0.083 | 0.083 | 5 | CString::CString() |
| 0.001 | 0.001 | 1 | CString::IsEmpty() |
| 0.083 | 0.083 | 1 | CString::LoadStringA() |
| 0.061 | 0.061 | 55 | CString::operator char const *() |
| 10.344 | 10.344 | 104 | CString::operator+=() |
| 0.071 | 0.071 | 7 | CString::operator=() |
| 0.08 | 0.08 | 1 | CWinApp::~CWinApp() |
| 4.527 | 4.527 | 371 | CWinApp::AssertValid() |
| 0.184 | 0.184 | 1 | CWinApp::CWinApp() |
| 0.007 | 0.007 | 1 | CWinApp::Enable3dControls() |
| 0.027 | 0.018 | 1 | CWinApp::ExitInstance() |
| 0.465 | 0.465 | 371 | CWinApp::GetRuntimeClass() |
| 0.003 | 0.003 | 1 | CWinApp::InitApplication() |
| 1.905 | 1.905 | 1 | CWinApp::LoadIconA() |
| 0.238 | 0.238 | 289 | CWinThread::GetMainWnd() |
| 0.308 | 0.308 | 183 | CWinThread::IsIdleMessage() |
| 89.245 | 12.291 | 184 | CWinThread::PreTranslateMessage() |
| 1.022 | 0.969 | 102 | CWinThread::ProcessMessageFilter() |
| 26553.78 | 23132.97 | 184 | CWinThread::PumpMessage() |
| 0.374 | 0.374 | 661 | CWnd::ContinueModal() |
| 98.274 | 34.241 | 833 | CWnd::DefWindowProcA() |
| 14.102 | 13.813 | 1 | CWnd::DestroyWindow() |
| 0.027 | 0.027 | 2 | CWnd::DoDataExchange() |
| 0.023 | 0.023 | 1 | CWnd::EndModalLoop() |
| 0.003 | 0.003 | 2 | CWnd::GetSuperWndProcAddr() |
| 0.16 | 0.16 | 1 | CWnd::GetSystemMenu() |

| | | | |
|---|---|---|---|
| 0.111 | 0.111 | 185 | CWnd::IsFrameWnd() |
| 0.009 | 0.009 | 2 | CWnd::IsIconic() |
| 0.068 | 0.068 | 4 | CWnd::KillTimer() |
| 0.048 | 0.047 | 2 | CWnd::OnChildNotify() |
| 45.078 | 3.977 | 135 | CWnd::OnCommand() |
| 0.215 | 0.011 | 2 | CWnd::OnPaint() |
| 0.67 | 0.263 | 47 | CWnd::OnTimer() |
| 3429.514 | 36.696 | 1151 | CWnd::OnWndMsg() |
| 0 | 0 | 1 | CWnd::PostNcDestroy() |
| 0.003 | 0.003 | 1 | CWnd::PreSubclassWindow() |
| 2.637 | 0.072 | 2 | CWnd::SetIcon() |
| 0.089 | 0.089 | 1 | CWnd::SetTimer() |
| 66.82 | 66.463 | 56 | CWnd::SetWindowTextA() |
| 1.934 | 0.013 | 1 | CWnd::UpdateData() |
| 3455.938 | 0.878 | 1151 | CWnd::WindowProc() |
| 0.003 | 0.003 | 2 | DDV_MinMaxFloat() |
| 0.003 | 0.003 | 2 | DDV_MinMaxInt() |
| 1.065 | 1.065 | 2 | DDV_MinMaxLong() |
| 1.991 | 1.991 | 6 | DDX_CBString() |
| 0.633 | 0.633 | 20 | DDX_Control() |
| 0.806 | 0.779 | 8 | DDX_Text() |
| 0.599 | 0.58 | 2 | DDX_Text() |
| 0.235 | 0.22 | 2 | DDX_Text() |
| 0.315 | 0.298 | 2 | DDX_Text() |
| 0.389 | 0.389 | 31 | operator delete() |
| 0.567 | 0.567 | 62 | operator new() |
| 0.018 | 0.018 | 5 | operator==() |
| 0.001 | 0 | 1 | QzApplication::QzApplication() |
| 0.01 | 0 | 1 | QzATMAgent::`scalar deleting destructor'() |
| 0.01 | 0 | 1 | QzATMAgent::`vector deleting destructor'() |
| 0.001 | 0 | 1 | QzATMAgent::~QzATMAgent() |
| 0.067 | 0.056 | 1 | QzATMAgent::Init() |
| 0.015 | 0.002 | 1 | QzATMAgent::InitWS2() |
| 0.545 | 0.043 | 1 | QzATMAgent::QoSRequest() |
| 0.072 | 0.003 | 1 | QzATMAgent::QzATMAgent() |
| 0 | 0 | 3 | QzATMFilter::QzATMFilter() |
| 0.31 | 0.003 | 1 | QzATMFilter::TranslateQoSDownwards() |
| 0.001 | 0.001 | 1 | QzBypassFilter::QzBypassFilter() |
| 0.002 | 0.002 | 4 | QzSystemAgent::~QzSystemAgent() |
| 0.006 | 0.006 | 4 | QzSystemAgent::QzSystemAgent() |
| 0.001 | 0 | 1 | QzDataPacketFilter::QzDataPacketFilter() |
| 0.118 | 0.029 | 1 | QzDataPacketFilter::TranslateQoSDownwards() |
| 0.002 | 0.002 | 2 | QzNetworkAgent::~QzNetworkAgent() |
| 0.003 | 0 | 2 | QzNetworkAgent::QzNetworkAgent() |
| 0.001 | 0 | 1 | QzOSAgent::~QzOSAgent() |
| 0.001 | 0 | 1 | QzOSAgent::QzOSAgent() |
| 0.145 | 0.002 | 2 | QzQoS::`scalar deleting destructor'() |
| 0.145 | 0.002 | 2 | QzQoS::`vector deleting destructor'() |
| 0.128 | 0.003 | 3 | QzQoS::~QzQoS() |
| 0.261 | 0.261 | 33 | QzQoS::GetParam() |
| 0 | 0 | 12 | QzQoS::GetParam() |
| 0.003 | 0.003 | 6 | QzQoS::QzQoS() |
| 0.613 | 0.16 | 33 | QzQoS::SetParam() |

| | | | |
|---|---|---|---|
| 0.036 | 0 | 1 | QzQoSAgent::`scalar deleting destructor'() |
| 0.036 | 0 | 1 | QzQoSAgent::`vector deleting destructor'() |
| 0.028 | 0.004 | 1 | QzQoSAgent::~QzQoSAgent() |
| 0.178 | 0.009 | 3 | QzQoSAgent::AgentFactory() |
| 0.25 | 0.008 | 1 | QzQoSAgent::Init() |
| 1.314 | 0.004 | 1 | QzQoSAgent::QoSRequest() |
| 0.096 | 0.005 | 1 | QzQoSAgent::QzQoSAgent() |
| 0.073 | 0.044 | 1 | QzRSVPAgent::`scalar deleting destructor'() |
| 0.073 | 0.044 | 1 | QzRSVPAgent::`vector deleting destructor'() |
| 0.018 | 0.003 | 1 | QzRSVPAgent::~QzRSVPAgent() |
| 0.002 | 0.002 | 1 | QzRSVPAgent::CleanUp() |
| 0.062 | 0.06 | 1 | QzRSVPAgent::QzRSVPAgent() |
| 0 | 0 | 3 | QzRSVPFilter::QzRSVPFilter() |
| 0.023 | 0.023 | 13 | QzTranslation::QzTranslation() |
| 0.049 | 0.028 | 3 | QzInterpreter::QzInterpreter() |
| 0.233 | 0.039 | 1 | QzInterpreter::TranslateQoSDownwards() |
| 0.003 | 0.003 | 3 | QzUpcall::QzUpcall() |
| 0.013 | 0.002 | 1 | QzWinNTAgent::`scalar deleting destructor'() |
| 0.013 | 0.002 | 1 | QzWinNTAgent::`vector deleting destructor'() |
| 0.002 | 0.001 | 1 | QzWinNTAgent::~QzWinNTAgent() |
| 0.08 | 0.06 | 1 | QzWinNTAgent::Init() |
| 0.337 | 0.236 | 1 | QzWinNTAgent::QoSRequest() |
| 0.003 | 0.002 | 1 | QzWinNTAgent::QzWinNTAgent() |
| 0.002 | 0.001 | 2 | QzWinNTFilter::QzWinNTFilter() |
| 0.098 | 0.003 | 1 | QzWinNTFilter::TranslateQoSDownwards() |

## A.2 UDP + RSVP

These are the results of the performance measurements obtained with the remote copy application using UDP for data transfer and RSVP for resource reservation. The same test scenario that was used for measuring the performance of the application when using the ATM protocol was applied in this case.

| Total Time (Function+Child) | Effective Function Time | Hit Count | Function Name |
|---|---|---|---|
| 0.945 | 0.945 | 2 | _closesocket@4 |
| 19.917 | 19.917 | 4 | _gethostbyname@4 |
| 10.966 | 10.966 | 2 | _gethostname@8 |
| 0.112 | 0.112 | 52 | _htons@4 |
| 0.105 | 0.105 | 5 | _inet_addr@4 |
| 0.621 | 0.621 | 50 | _inet_ntoa@4 |
| 0.001 | 0.001 | 1 | _ntohl@4 |
| 23.274 | 23.274 | 47 | _sendto@24 |
| 12548.99 | 97.302 | 1 | _WinMain@16 |
| 0.014 | 0.014 | 1 | _WSACleanup@0 |
| 1.024 | 1.024 | 4 | _WSAEnumProtocolsA@12 |
| 11.533 | 11.533 | 3 | _WSAIoctl@36 |
| 18.695 | 18.695 | 2 | _WSASocketA@24 |
| 9.73 | 9.73 | 2 | _WSAStartup@8 |
| 0.06 | 0.06 | 1 | AfxInitialize() |

| | | | |
|---|---|---|---|
| 0.097 | 0.097 | 1 | CSampleApp::~CSampleApp() |
| 0.241 | 0.241 | 1 | CSampleApp::CSampleApp() |
| 12451.69 | 11943.65 | 1 | CSampleApp::InitInstance() |
| 0 | 0 | 327 | CSampleDlg::_GetBaseMessageMap() |
| 0.171 | 0.171 | 1 | CSampleDlg::~CSampleDlg() |
| 3.054 | 3.052 | 1 | CSampleDlg::CsampleDlg() |
| 7.279 | 7.279 | 2 | CSampleDlg::DoDataExchange() |
| 20.641 | 0.028 | 1 | CSampleDlg::FillLocalAddress() |
| 0.788 | 0.233 | 1 | CSampleDlg::FindProtocols() |
| 0.902 | 0.902 | 1103 | CSampleDlg::GetMessageMap() |
| 9.72 | 0.004 | 1 | CSampleDlg::InitWS2() |
| 0 | 0 | 1 | CSampleDlg::OnCancel() |
| 53.28 | 1.86 | 1 | CSampleDlg::OnConnect() |
| 61.7 | 25 | 1 | CSampleDlg::OnInitDialog() |
| 0.432 | 0.432 | 2 | CSampleDlg::OnPaint() |
| 385.566 | 12.856 | 47 | CSampleDlg::OnTimer() |
| 353.54 | 353.591 | 50 | CSampleDlg::PrintStatus() |
| 49.473 | 0.469 | 1 | CSampleDlg::UDPSender() |
| 0.002 | 0.001 | 1 | QzApplication::QzApplication() |
| 0 | 0 | 2 | QzATMFilter::QzATMFilter() |
| 0 | 0 | 1 | QzBypassFilter::QzBypassFilter() |
| 0 | 0 | 3 | QzSystemAgent::~QzSystemAgent() |
| 0.003 | 0.003 | 3 | QzSystemAgent::QzSystemAgent() |
| 0 | 0 | 1 | QzDataPacketFilter::QzDataPacketFilter() |
| 0.161 | 0.05 | 1 | QzDataPacketFilter::TranslateQoSDownwards() |
| 0 | 0 | 1 | QzNetworkAgent::~QzNetworkAgent() |
| 0.001 | 0 | 1 | QzNetworkAgent::QzNetworkAgent() |
| 0 | 0 | 1 | QzOSAgent::~QzOSAgent() |
| 0 | 0 | 1 | QzOSAgent::QzOSAgent() |
| 0.136 | 0.019 | 2 | QzQoS::`scalar deleting destructor'() |
| 0.136 | 0.019 | 2 | QzQoS::`vector deleting destructor'() |
| 0.116 | 0.116 | 2 | QzQoS::~QzQoS() |
| 0.186 | 0.186 | 43 | QzQoS::GetParam() |
| 0 | 0 | 12 | QzQoS::GetParam() |
| 0.002 | 0.002 | 5 | QzQoS::QzQoS() |
| 0.725 | 0.725 | 32 | QzQoS::SetParam() |
| 1.709 | 0.011 | 1 | QzQoSAgent::`scalar deleting destructor'() |
| 1.709 | 0.011 | 1 | QzQoSAgent::`vector deleting destructor'() |
| 1.698 | 0.006 | 1 | QzQoSAgent::~QzQoSAgent() |
| 0.105 | 0.031 | 2 | QzQoSAgent::AgentFactory(unsigned int,char *) |
| 9.163 | 0.007 | 1 | QzQoSAgent::Init() |
| 21.176 | 0.007 | 1 | QzQoSAgent::QoSRequest() |
| 0.109 | 0.003 | 1 | QzQoSAgent::QzQoSAgent() |
| 1.683 | 0.014 | 1 | QzRSVPAgent::`scalar deleting destructor'() |
| 1.683 | 0.014 | 1 | QzRSVPAgent::`vector deleting destructor'() |
| 1.669 | 0.03 | 1 | QzRSVPAgent::~QzRSVPAgent() |
| 1.64 | 0.004 | 1 | QzRSVPAgent::CleanUp() |
| 5.214 | 0.008 | 1 | QzRSVPAgent::FillLocalAddress() |
| 0.64 | 0.171 | 1 | QzRSVPAgent::FindProtocols() |
| 9.079 | 0.297 | 1 | QzRSVPAgent::Init() |
| 0.058 | 0.044 | 1 | QzRSVPAgent::InitWS2() |
| 20.208 | 0.006 | 1 | QzRSVPAgent::QoSRequest() |

| | | | |
|---|---|---|---|
| 0.074 | 0.072 | 1 | QzRSVPAgent::QzRSVPAgent() |
| 9.18 | 0.01 | 1 | QzRSVPAgent::RSVPOpenSocket() |
| 9.825 | 0.011 | 1 | QzRSVPAgent::RSVPRegister |
| 1.034 | 0.004 | 1 | QzRSVPAgent::RSVPRelSender() |
| 0.757 | 0.018 | 1 | QzRSVPAgent::RSVPSender |
| 0 | 0 | 3 | QzRSVPFilter::QzRSVPFilter() |
| 0.413 | 0.075 | 1 | QzRSVPFilter::TranslateQoSDownwards() |
| 0.015 | 0.015 | 12 | QzTranslation::QzTranslation |
| 0.06 | 0.041 | 3 | QzInterpreter::QzInterpreter() |
| 0.27 | 0.055 | 1 | QzInterpreter::TranslateQoSDownwards() |
| 0.001 | 0.001 | 3 | QzUpcall::QzUpcall() |
| 0.009 | 0.011 | 1 | QzWinNTAgent::`scalar deleting destructor'() |
| 0.009 | 0.011 | 1 | QzWinNTAgent::`vector deleting destructor'() |
| 0 | 0 | 1 | QzWinNTAgent::~QzWinNTAgent() |
| 0.077 | 0.077 | 1 | QzWinNTAgent::Init() |
| 0.461 | 0.313 | 1 | QzWinNTAgent::QoSRequest() |
| 0 | 0.001 | 1 | QzWinNTAgent::QzWinNTAgent() |
| 0 | 0 | 2 | QzWinNTFilter::QzWinNTFilter() |
| 0.147 | 0.045 | 1 | QzWinNTFilter::TranslateQoSDownwards() |

## A.3 TCP + RSVP

These are the results of the performance measurements obtained with the "rcp" application using TCP for data transfer and RSVP for resource reservation. The same test procedure that was used in the last two cases was applied here.

| Total Time (Function+Child) | Effective Function Time | Hit Count | Function Name |
|---|---|---|---|
| 9.884 | 9.884 | 2 | _closesocket@4 |
| 0.687 | 0.687 | 1 | _connect@12 |
| 0.002 | 0.002 | 1 | _GetCurrentProcess@0 |
| 19.704 | 19.704 | 4 | _gethostbyname@4 |
| 10.708 | 10.708 | 2 | _gethostname@8 |
| 0.003 | 0.003 | 1 | _GetModuleHandleA@4 |
| 0.013 | 0.013 | 1 | _GetStartupInfoA@4 |
| 0.004 | 0.004 | 6 | _htons@4 |
| 0.118 | 0.118 | 5 | _inet_addr@4 |
| 0.046 | 0.046 | 3 | _inet_ntoa@4 |
| 76.114 | 76.114 | 47 | _send@16 |
| 0.285 | 0.285 | 1 | _SetPriorityClass@8 |
| 0.028 | 0.028 | 1 | _setsockopt@20 |
| 14111.97 | 26.879 | 1 | _WinMain@16 |
| 1.742 | 1.742 | 1 | _WSAAsyncSelect@16 |
| 0.016 | 0.016 | 1 | _WSACleanup@0 |
| 0.994 | 0.994 | 4 | _WSAEnumProtocolsA@12 |
| 0.003 | 0.003 | 1 | _WSAGetLastError@0 |
| 23.841 | 23.841 | 3 | _WSAIoctl@36 |
| 111.24 | 111.24 | 2 | _WSASocketA@24 |
| 9.591 | 9.591 | 2 | _WSAStartup@8 |
| 0.057 | 0.057 | 1 | AfxInitialize() |
| 0.093 | 0.093 | 1 | CSampleApp::~CSampleApp() |

| | | | |
|---|---|---|---|
| 0.251 | 0.251 | 1 | CSampleApp::CSampleApp |
| 14085.09 | 13281.14 | 1 | CSampleApp::InitInstance() |
| 0.072 | 0.072 | 323 | CSampleDlg::_GetBaseMessageMap() |
| 0.17 | 0.17 | 1 | CSampleDlg::~CSampleDlg() |
| 2.516 | 2.514 | 1 | CSampleDlg::CSampleDlg() |
| 8.13 | 8.13 | 2 | CSampleDlg::DoDataExchange() |
| 20.833 | 0.034 | 1 | CSampleDlg::FillLocalAddress() |
| 0.756 | 0.224 | 1 | CSampleDlg::FindProtocols() |
| 1.27 | 1.27 | 1040 | CSampleDlg::GetMessageMap() |
| 9.581 | 0.006 | 1 | CSampleDlg::InitWS2() |
| 160.594 | 0.232 | 1 | CSampleDlg::OnAsyncSelectTCP() |
| 0 | 0 | 1 | CSampleDlg::OnCancel() |
| 121.013 | 1.875 | 1 | CSampleDlg::OnConnect() |
| 67.974 | 30.347 | 1 | CSampleDlg::OnInitDialog() |
| 0.159 | 0.159 | 1 | CSampleDlg::OnPaint() |
| 448.215 | 11.75 | 47 | CSampleDlg::OnTimer() |
| 477.656 | 476.973 | 51 | CSampleDlg::PrintStatus() |
| 117.208 | 0.298 | 1 | CSampleDlg::TCPSender() |
| 0.002 | 0.001 | 1 | QzApplication::QzApplication() |
| 0.003 | 0.001 | 2 | QzATMFilter::QzATMFilter() |
| 0.002 | 0.002 | 1 | QzBypassFilter::QzBypassFilter() |
| 0 | 0 | 3 | QzSystemAgent::~QzSystemAgent() |
| 0.004 | 0.004 | 3 | QzSystemAgent::QzSystemAgent() |
| 0.004 | 0.003 | 1 | QzDataPacketFilter::QzDataPacketFilter() |
| 0.195 | 0.09 | 1 | QzDataPacketFilter::TranslateQoSDownwards() |
| 0.002 | 0.002 | 1 | QzNetworkAgent::~QzNetworkAgent() |
| 0.002 | 0.001 | 1 | QzNetworkAgent::QzNetworkAgent() |
| 0.001 | 0.001 | 1 | QzOSAgent::~QzOSAgent() |
| 0.003 | 0.002 | 1 | QzOSAgent::QzOSAgent() |
| 0.129 | 0.023 | 2 | QzQoS::`scalar deleting destructor'() |
| 0.129 | 0.023 | 2 | QzQoS::`vector deleting destructor'() |
| 0.108 | 0.108 | 3 | QzQoS::~QzQoS() |
| 0.208 | 0.208 | 43 | QzQoS::GetParam() |
| 0.009 | 0.009 | 12 | QzQoS::GetParam() |
| 0.007 | 0.007 | 6 | QzQoS::QzQoS() |
| 0.708 | 0.708 | 32 | QzQoS::SetParam() |
| 9.525 | 0.013 | 1 | QzQoSAgent::`scalar deleting destructor'() |
| 9.525 | 0.013 | 1 | QzQoSAgent::`vector deleting destructor'() |
| 9.512 | 0.01 | 1 | QzQoSAgent::~QzQoSAgent() |
| 0.117 | 0.028 | 2 | QzQoSAgent::AgentFactory() |
| 9.149 | 0.009 | 1 | QzQoSAgent::Init() |
| 119.025 | 0.016 | 1 | QzQoSAgent::QoSRequest() |
| 0.13 | 0.01 | 1 | QzQoSAgent::QzQoSAgent() |
| 9.487 | 0.017 | 1 | QzRSVPAgent::`scalar deleting destructor'() |
| 9.487 | 0.017 | 1 | QzRSVPAgent::`vector deleting destructor'() |
| 9.47 | 0.04 | 1 | QzRSVPAgent::~QzRSVPAgent() |
| 9.429 | 0.006 | 1 | QzRSVPAgent::CleanUp() |
| 5.257 | 0.008 | 1 | QzRSVPAgent::FillLocalAddress() |
| 0.629 | 0.167 | 1 | QzRSVPAgent::FindProtocols() |
| 9.061 | 0.287 | 1 | QzRSVPAgent::Init() |
| 0.056 | 0.04 | 1 | QzRSVPAgent::InitWS2() |
| 117.998 | 0.018 | 1 | QzRSVPAgent::QoSRequest() |
| 0.082 | 0.08 | 1 | QzRSVPAgent::QzRSVPAgent() |

| | | | |
|---|---|---|---|
| 101.933 | 0.014 | 1 | QzRSVPAgent::RSVPOpenSocket() |
| 14.519 | 0.016 | 1 | QzRSVPAgent::RSVPRegister() |
| 8.375 | 0.009 | 1 | QzRSVPAgent::RSVPRelSender() |
| 1.042 | 0.018 | 1 | QzRSVPAgent::RSVPSender |
| 0 | 0 | 3 | QzRSVPFilter::QzRSVPFilter() |
| 0.452 | 0.104 | 1 | QzRSVPFilter::TranslateQoSDownwards() |
| 0.022 | 0.022 | 12 | QzTranslation::QzTranslation() |
| 0.059 | 0.04 | 3 | QzInterpreter::QzInterpreter() |
| 0.306 | 0.079 | 1 | QzInterpreter::TranslateQoSDownwards() |
| 0.002 | 0.002 | 3 | QzUpcall::QzUpcall() |
| 0.015 | 0.013 | 1 | QzWinNTAgent::`scalar deleting destructor'() |
| 0.015 | 0.013 | 1 | QzWinNTAgent::`vector deleting destructor'() |
| 0.003 | 0.002 | 1 | QzWinNTAgent::~QzWinNTAgent() |
| 0.078 | 0.078 | 1 | QzWinNTAgent::Init() |
| 0.449 | 0.013 | 1 | QzWinNTAgent::QoSRequest() |
| 0.007 | 0.004 | 1 | QzWinNTAgent::QzWinNTAgent() |
| 0.001 | 0.001 | 2 | QzWinNTFilter::QzWinNTFilter() |
| 0.146 | 0.044 | 1 | QzWinNTFilter::TranslateQoSDownwards() |