QoS-Adaptive Middleware Services

Vittorio Ghini

Technical Report UBLCS-2002-05

March 2002

Department of Computer Science
University of Bologna
Mura Anteo Zamboni, 7
40127 Bologna (Italy)
Recent Titles from the UBLCS Technical Report Series


Dottorato di Ricerca in Informatica
Università di Bologna, Padova, Venezia

QoS-Adaptive Middleware Services

Vittorio Ghini

12 Febbraio 2002

Coordinatore:
Prof. Özalp Babaoglu

Tutore:
Prof. Fabio Panzieri
Abstract

The aim of this thesis is to investigate issues of design of adaptive Quality of Service (QoS) policies, located at an intermediate level between applications and operating systems (i.e. the middleware level), to support effectively geographically distributed real-time applications over best-effort IP-based networks. Specifically, in order to stress the ability to provide effective QoS at the middleware level, two policies are proposed in this thesis. These two policies have been deployed in two application systems that exhibit very different QoS requirements: namely, a replicated Web service, and a voice-based communication service. The two policies, a downloading mechanism for a geographically distributed and replicated Web service, and a synchronization mechanism for voice based communications, have been implemented and evaluated, providing very interesting results. Based on our experience, we identify the principal characteristics of adaptive policies and applications, and propose a middleware architecture that allows one to construct QoS-aware distributed applications out of reusable building blocks (BBs). In summary, the middleware architecture proposed in this thesis adds a QoS-oriented interface to each BB, in order to describe the QoS properties of BB.
Acknowledgments

I would like to thank my tutors, Professor Fabio Panzieri and Professor Marco Roccetti, for their support throughout my research efforts towards this thesis and their helpful discussions on the idea in the work. They have been a very important example for me.

I would like to thank Professor Santosh K. Shrivastava, that made possible my precious experience at the department of Computing Science of the University of Newcastle upon Tyne, and also Dave Ingham, Graham Morgan and Paul Ezhilchelvan.

A particular thank to Professor Jon Crowcroft, for its appreciated comments and suggestions.

Last but not least, a special thank to my friends, Giovanni, Paola, Diego, Stefano, my parents Rosanna and Filippo and my girlfriend Anna.
## Contents

Abstract i  
Acknowledgments iii  
Contents v  
List of Tables ix  
List of Figures xi  

1 Introduction 1  
2 End-to-End Quality of Service 11  
   2.1 The Quality of Service 12  
      2.1.1 The traditional Notion of Quality of Service 12  
      2.1.2 Network QoS Issues 13  
      2.1.3 Approaches to Network QoS Support 14  
   2.2 Applications QoS 17  
      2.2.1 Applications Taxonomies 17  
   2.3 The Notion of End-To-End QoS 22  
      2.3.1 End-To-End QoS Entities 23  
      2.3.2 End-To-End QoS Management 25  

3 The Middleware 27  
   3.1 The Transportable Computation Approach 27  
   3.2 Middleware models 28  
      3.2.1 Object Request Brokers 29  
      3.2.2 Message Oriented Middleware 31  
      3.2.3 Remote Procedure Call 32  
      3.2.4 Transaction Processing Monitors 33  
   3.3 CORBA 34  
      3.3.1 CORBA Structure 35  
      3.3.2 CORBA Interoperability 38  
   3.4 COM/DCOM 40  
      3.4.1 COM/DCOM Architecture 40
6.4.2 Experimental Results and data interpretation  . . . . 124
6.4.3 Comparative Simulation Results  . . . . . . . . 127
6.5 Are the silences sufficient?  . . . . . . . . . . . 130
6.6 Concluding Remarks  . . . . . . . . . . . . . 134

7 An Architecture for Qos-Aware Applications 137
7.1. QoS-Adaptive Middleware  . . . . . . . . . . . . . . 138
7.2. Synchronous and Asynchronous Calls  . . . . . . 139
7.3. Architecture  . . . . . . . . . . . . . . . . . . . . 141
    7.3.1. The Building Block  . . . . . . . . . . . 142
7.4. QoS Parameters Upgrading Mechanism  . . . . 143

8 Concluding Remarks 145

References 147
# List of Tables

5.1 The measured round trip time and lost packet percentage .......................... 88
5.2 Fault Percentage ....................................................................................... 89
5.3 Average URT percentage improvement .................................................... 92

6.1 Playout delay spike management: delay spike detection (left) and playout delay estimation ................................................................. 120
6.2 Experimental results: average playout delay and packet loss .................... 126
6.3 Simulation results: average playout delays (msec) ...................................... 129
6.4 State transition parameters for the modified Brady's model ...................... 131
# List of Figures

2.1 End-to-end QoS delivery ........................................ 13
2.2 Network QoS Taxonomy ............................................ 15
2.3 Applications Taxonomy ............................................. 18
2.4 Guaranties required by the Applications ........................ 20
2.5 End-to-end QoS entities ............................................ 22
2.6 QoS Levels .......................................................... 23
2.7 QoS End System Components ...................................... 24

3.1 Jini architecture ................................................... 28
3.2 Using the Middleware ............................................... 29
3.3 Object Request Broker .............................................. 30
3.4 Message Oriented Middleware ..................................... 31
3.5 RPC model ........................................................... 32
3.6 Transaction processing monitors technology .................... 33
3.7 Object Management Architecture ................................. 35
3.8 Object Management Architecture ................................. 36
3.9 CORBA Entity devoted to Interoperability ....................... 39
3.10 Client Using COM Object Through an Interface Pointer ........ 41
3.11 Three Methods for Accessing COM Objects ....................... 42
3.12 Cross-process communication in COM ........................... 43
3.13 Creating a COM object pointer .................................... 44
3.14 The Call Back Architecture ........................................ 45
3.15 CORBA method call ................................................ 50
3.16 QuO component ...................................................... 50
3.17 QuO's in-band adaptation (upon methods call and returns) .... 52
3.18 QuO's out-of-band adaptation (adaptation on region transitions) ... 53
3.19 The Hierarchical Design of the Agilos Architecture ............. 54

5.1 Implementation of the C²LD Service ............................. 82
5.2 Thread structure of C2LD ........................................... 86
5.3 C2LD as part of the Proxy Service ................................. 86
5.4 Routes between the client and the Web replica servers ............ 87
5.5 Daily Traffic Statistics ............................................. 88
7.1 Asynchronous Call Back Function Mechanism . . . . . . . . . . 140
7.2 The complete Architecture . . . . . . . . . . . . . . . . . . . 141
Chapter 1

Introduction

In this thesis we discuss the design, implementation and performance evaluation of a set of middleware services we have developed in order to provide soft real-time applications, geographically distributed over a best-effort IP network such as the Internet, with QoS guarantees.

The motivation for this thesis can be enunciated as follows.

The current software technology allows us to construct distributed application systems by composing reusable interoperable building blocks (i.e. software components), such as commercial operating systems, communication protocols, and middleware services. The success of a distributed application, constructed out of those building blocks (BB), typically depends on the Quality of Service (QoS) that application can provide its users with (i.e., on the QoS properties that application possesses). Issues of QoS have been addressed principally in the design of communication protocols and mechanisms that permit the control of communication parameters, such as network throughput, delay, delay jitter, and packet loss [1, 2, 3, 4]. These parameters indeed affect the quality of the service perceived by the users, however, as pointed out in [5], further QoS requirements emerge at the application level; these may include performance-oriented requirements (e.g. timeliness of execution and relative processing speed requirements), reliability-oriented requirements (e.g. high availability and relative failure recovery requirements), and security oriented requirements (e.g. authentication, privacy).

As of today, the Internet offers a point-to-point delivery service, which is based on the "best effort" delivery model. In this model, data are delivered to their destination as soon as possible, with no bandwidth or latency guarantees. This service model can be adequate for traditional applications, such as FTP and Telnet, but inadequate for applications requiring timeliness guarantees. For example, distributed multimedia applications need to communicate in real-time and are sensitive to the QoS they receive from the network. For these applications to perform adequately and be widely used,
Quality of Service (QoS) must be quantified and managed. Traditional QoS management is provided by the network layer of the communication system. From the point of view of the QoS, the ability of a network to deliver service needed by a specific network applications from end-to-end, with some level of control over delay, packet loss, jitter, and/or bandwidth, can be categorized into the following three levels of service: Best Effort Service (basic connectivity with no guarantees), Differentiated Service (expedite handling of specific classes of traffic), and Guaranteed Service (a reservation of network resources to ensure that specific traffic gets the specific level of service it requires).

These three levels of service can be provided by the following two different approaches in building the network infrastructure and its software layers: namely, an IP based and an ATM based (Asynchronous Transfer Mode [6]) approaches. The former approach maintains the widespread packet switching IP technology and provides differentiated delivery services for individual flows (the fine grained management provided by Intserv/RSVP [7] by means of resource reservation), or aggregates (the coarse grained management provided by Diffserv [8], MPLS [9], and SBM [10], by means of traffic prioritization). Instead, the latter approach, replaces the IP technology with connection-oriented services. Currently, the trend is to adopt an IP based approach.

One distinguishing feature of QoS is that application level QoS parameters are subject to negotiation between applications and both system and network components, in order to determine whether the application requiring QoS can be provided by the entire system. This process is commonly referred to as QoS Management. Several resource managers (brokers) have been deployed to provide support for the dynamic management of QoS. In [11] an end-point entity called the QoS Broker is described, that operates over networks based on Asynchronous Transfer Mode. The broker orchestrates resources at the end-points, coordinating resource management across layer boundaries. As an intermediary, it hides implementation details from applications and per-layer resource managers. The broker uses services such as translation, admission and negotiation to properly configure the system to meet the application requirements. The broker manages communication between the entities to create the desired system configuration via QoS negotiation. The negotiation involves all the components of the communication system needed for setup. In [12] the concept of QoS Broker is extended in order to provide QoS management for different network
technologies, other than ATM. The proposed Distributed Resource Controller (DRC) technology aims to unify network services (e.g., Diffserv, Intserv, and ATM) and application QoS provisioning by introducing a middleware system and a set of generic interfaces. DRC uses the CORBA-based object-oriented technology. Application QoS requirements are specified in terms of two categories of parameters: Traffic Profile (quantitative) and User Expectation (qualitative).

As stated above, applications can rely on a set of different delivery services. However, a problem arises. What network services do we want the application use? Networks providing some kind of guaranteed service are particularly suitable for supporting applications (such as IP telephony) that present soft real-time requirements. Unfortunately, these networks are not sufficiently widespread and their services can be highly expensive. Hence, in general, applications are designed so as to use best-effort network services, rather than guaranteed services, in order to minimize the cost of network resources. Moreover, there are QoS requirements that cannot be met by using guaranteed network services only. For instance, such QoS requirements as the high availability of a distributed service can be met by deploying appropriate service replication techniques that can face both communication and host failures, regardless of the QoS provided by the network infrastructure.

Thus, in order to investigate a general approach to the provision of end-to-end QoS, in this thesis we assume that the communication subsystem provides a best-effort communication service, such as the IP datagram service currently available over the Internet.

Adoption of best-effort network services means that there is no way of exercising control over network resources. Hence, the performance of these resources may dynamically change during the execution of the applications. This thesis argues that, at the application level, it is possible to develop a large set of architectures/policies capable providing the applications with scalability, interoperability, fault tolerance, availability, timeliness, and to meet soft real-time constraints. Basically, the applications need to be made adaptive (i.e., the applications must adapt to fluctuations in resource availability and performances) and to develop suitable QoS-oriented architectures. In order to dynamically adjust the applications to the behavior of the resources, we need to define a set of services that maintain information about both the network and the computational resources and modify the behavior of the applications. These services consist of two different services, a monitoring service and an
adaptation service. The monitoring service, given some information about the application QoS requirements, evaluates run-time availability of the necessary system resources, and informs the adjustment service. Instead, the adjustment service adapts the applications to changes in the environment according to the information provided by the monitoring service.

Both these services can be implemented either as part of the applications themselves (i.e. the middleware level) or as a separate software layer, so as to allow the applications to concentrate on their application-specific functions.

As pointed out in [13], the term middleware refers to a distributed platform of interfaces and services that reside ‘between’ the application and the operating system and aim to facilitate the development, deployment and management of distributed applications. Essentially, the middleware is a distributed software layer, or ‘platform’ which shields the application from the complexity and heterogeneity of the underlying distributed environment.

The most popular middleware model is the object-based model that provides access to remote objects that implement the services. The Object Request Broker (ORB) model provides a framework for cross-system communication between objects. An ORB allows objects to hide their implementation details from clients. These can include programming languages, operating systems, host hardware, and object location. The most important examples of this type of middleware are the Object Management Group's Common Object Request Broker Architecture (OMG's CORBA) [15, 16] and Microsoft's COM/DCOM (Component Object Model and Distributed Component Object Model) [17, 18]. Both of these platforms offer an interface definition language (IDL), which means that objects can be implemented in any suitable programming language, an object request broker, which is responsible for transparently directing method invocations to the appropriate target object, and a set of services (e.g. naming, time, transactions, replication etc.) which further enhance the distributed programming environment.

Other middleware models include Message-Oriented Middleware (MOM) that provides program-to-program data exchange (for instance, IBM's MQSeries [19] and Talarian's SmartSocket [20]), and Transaction processing (TP) monitors and its object-oriented evolution typically used to manage the writing and reading of transactional data (for instance, IBM's CICS and BEA's Tuxedo [21]).
As previously mentioned, the main contribution of this thesis consists of the design, implementation and evaluation of a number of QoS based policies, that provide support to soft-real-time geographically distributed applications, based on best-effort communication services. Specifically, we show that these policies can be particularly effective when implemented as a set of middleware services that i) monitor the availability and performance of both the communications and computational resources required by the applications, and ii) provide the applications with information that allow them to adapt their behavior to variation of the execution environment. The middleware level, rather than the application level, is particularly suitable for implementing the monitoring of the resource availability and performance as their functions require interactions with both the operating system (CPU workload, memory usage) and the communication subsystem (packets sent/received). In addition, such functions that collect statistical information from different entities (local or remote), and provide process synchronization and notification services can be conveniently implemented at the middleware level as well. Within this scenario, the applications are responsible for providing only the middleware with sufficient information to enable and activate the middleware components, and to map QoS application requirements onto middleware component parameters.

In order to assess the adequacy of our middleware approach to support effective QoS policies, two case studies have been developed, and a number of QoS policies proposed, implemented and evaluated.

Our approach strictly depends on the services considered but, generally, it consists of the following three principal aspects:

1) Identification of distributed services that can take advantage of QoS management and the analysis of QoS constraints.

2) Definition and parameterization of QoS characteristics. Great importance is given to the characterization of QoS properties for a specific service, in order to define a set of parameters that can be used as a QoS service description. From these parameters, we need to define a metric to evaluate the validity of the QoS management solutions adopted.

3) Proposal and evaluation of system architectures and algorithms. This fundamental step consists of: i) an evaluation of existing system architectures that provide support for QoS-dependent distributed applications; ii) the proposal of new architectural models and algorithms devoted to this end; iii) the implementation, development,
experimentation, evaluation and comparison of these architectural models and algorithms, in order to discover their properties, and the interaction with the standard platform, on which they are based. Performance will be evaluated by means of metrics based on the QoS parameterization previously defined.

In order to experiment and evaluate QoS-based policies at the middleware level, for the scope of this thesis, we have examined the following two services that represent, from the point of view of the QoS requirements, two different classes of applications; namely:

- A unicast voice-based audio communication service over the Internet, which exhibit soft real-time constraints.
- A load distribution service among replicated web servers [22], which does not depend on strong real-time constraints, but should posses QoS properties such as availability and responsiveness.

We have proposed the two QoS policies introduced in the following to meet the QoS requirements of the two case studies mentioned above. Both these policies have been implemented; the experimental evaluations we have carried out have confirmed the effectiveness of the design approach we propose.

We have developed a mechanism (named Client-Centered Load Distribution, C\textsuperscript{2}LD) for constructing responsive web services that can work either as an extension of the client-side software, or as part of a proxy server. Specifically, our mechanism provides the clients of a replicated Web service, constructed out of replica servers distributed over the Internet, with timely responses. The main goal of our mechanism is to minimize what we term the User Response Time (URT), i.e. the time that elapses between the generation of a browser request for the retrieval of a Web page, and the rendering of that page at the browser site. That URT represents the main QoS parameter of the Web service. To summarize, our mechanism intercepts each client browser request for a Web page, and, rather than binding a client to its most convenient replica server, as proposed in [23, 24, 25], it fragments that request into a number of sub-requests for separate parts of that document. Each sub-request is issued to a different available replica server, concurrently. The replies received from the replica servers are reassembled at the client end to reconstruct the requested page, and then
delivered to the client browser. Our mechanism is designed so as to adapt dynamically to state changes in both the network (e.g. route congestion, link failures), and the replica servers (e.g. replica overload, unavailability). To this end, our mechanism periodically monitors the available replica servers. Moreover, an event-based monitoring is performed, in particular when a given replica terminates its response or when an exception occurs (e.g. the failure of a given replica or of the network). The QoS parameter used for selecting the best replica from which to require a fragment is the data rate provided by each replica during the response to the last request. The experiments conducted show that the proposed mechanism provides the users with availability and timeliness.

Voice-based communications over the Internet represent one of the most important components of multimedia conferencing. An audio segment consists of short bursts of energy (called talkspurts) separated by silent periods (during which no audio packet is generated). To transport audio over a non-guaranteed packet-switched network, audio samples are encoded, inserted into packets, transported by the network, received in a playout buffer, decoded, and finally played out in sequential order by the audio device. In order to compensate for variable network delays, a smoothing playout buffer is used at the receiver end. Received audio packets are queued into that buffer, and the playout of each packet of a given talkspurt is delayed for some time beyond the reception of the first packet of that talkspurt. In this way, dynamic playout buffers can hide packet delay variance at the receiver end at the cost of additional delay. A crucial tradeoff exists between the length of the imposed additional quantity of delay and the amount of lost packets, due to their late arrival: the longer the additional delay, the more likely it is that a packet will arrive before its scheduled playout deadline. However, excessively long playout delays may in turn seriously compromise the quality of the conversation over the network. The followings two parameters, end-to-end playout delay (the elapsed time between packet audio generation at the sending site and its playout time at the receiver end) and the percentage of audio packet loss, represent the main QoS parameters for Voice-based communications over the Internet. We have proposed a playout delay control mechanism that is suitable for adjusting playout delays of unicast, voice-based communications across the Internet. The mechanism was designed to dynamically adjust the talkspurt playout delays to the network traffic conditions without assuming either the existence of an external mechanism for maintaining an
accurate clock synchronization between the sender and the receiver, or a specific distribution of the end-to-end transmission delays experienced by the audio packets. The mechanism is based on the periodic monitoring of the RTT between sender and receiver. The proposed mechanism guarantees that the talkspurt playout delay may be dynamically set from one talkspurt to the next, without causing gaps or time collisions [26] inside the talkspurts themselves, provided that intervening silent periods of sufficiently long duration are exploited for the adjustment. We have verified the presence of a sufficient number of sufficiently long silences in human conversations by modeling the voice activity characteristics of human conversational speech [27] and by developing several simulation experiments. The experimentation we have conducted highlights that our mechanism provides good performance, in terms of end-to-end playout delay and packet loss percentage.

The effectiveness of the proposed policies raises the problem of integrating these types of polices into QoS-adaptive reusable building blocks (BBs), available for constructing QoS-adaptive applications by composition. Thus, the last contribution of this thesis is the proposal of an abstract middleware architecture, based on an ORB technology, that is suitable for constructing adaptive applications by composing reusable, interoperable, QoS-adaptive building blocks.

The principal motivation for this architecture is that our experience has shown that it is convenient to develop QoS adaptive policies at the middleware level, for different classes of applications. However, the following issues must be considered in the design and implementation of these policies.

- These policies can consist of the setting up of several QoS adaptive sub-services. For instance, a browser can consider a given replica server, accessed by means of HTTP messages and represented by its data rate, as an insulated QoS adaptive sub-service.
- From the point of view of the QoS, for each QoS adaptive sub-service there are some QoS parameters that describe the behavior and represent the QoS status (a sort of QoS interface) of that sub-service. For instance, the parameter that represents the QoS status of the sub-service replica server is the data rate provided by that replica.
• Several sub-services can modify their behavior depending on the input they receive. Thus, the parameters that describe their QoS behavior can change at run-time, depending on the data received. For instance, the QoS parameter *provided data rate* that describes the Web replica server sub-service can change, depending on the amount of data received in the time unit.

• Several sub-services rely on other sub-services, and their behavior may depend on the variation of the QoS parameters of the other sub-services on which they rely. Thus, changing a QoS parameter value of a given sub-service may cause a chain of reactions in the sub-services that rely on that sub-service.

• All the adaptive policies present common characteristics. In particular, they need to monitor (periodically and/or depending on events) the QoS status of the sub-services in order to adapt themselves to the behavior of the sub-systems.

• The applications themselves are characterized by a set of QoS parameters; thus, they can be thought of as high-level sub-services, available for composition.

System design through the composition of QoS-adaptive building blocks (BBs) gives rise to a number of interesting problems. The building block needs to be provided by an interface characterized by means of QoS parameters. Moreover, the BB's interface must to be specified also in terms of frequency of monitoring and updating of QoS parameters. In order to improve reusability, the BB cannot be devoted to a given application. Hence, the BB must be able to configure itself according to the application's policy, without introducing modifications to its implementation. Finally, a mechanism is required in order to specify the QoS behavior of each BB, depending on the changing QoS behavior of other BBs.

We propose an abstract middleware architecture, based on an ORB model, that is suitable for constructing applications by composing reusable interoperable *QoS adaptive building blocks* (QABB). That middleware extends the functionality of the building block by allowing the designer to add a QoS-Oriented Interface (QOI) to the BB, able to specify the QoS behavior that the BB exhibits. Moreover, that interface provides methods for configuring the BB. We named *QoS Adaptive Building Block* (QABB) the object constituted of the BB and the QoS Oriented Interface. The middleware presents a module (namely, Parameters Monitor, PM) that is responsible for monitoring and updating the QoS parameters of the QABBs' interfaces, according to the adaptive policies implemented by the designer. In essence, that Parameters Monitor
acts as a sort of scheduler for the monitoring procedures (implemented by the QABBs) and propagates the effects of QABB behavioral changes among all QABBs.

This thesis is organized as follows. In chapter 2 we introduce the notion of Quality of Service and describe the QoS requirements of the applications. We also briefly introduce the current approaches to QoS management at the network level. In chapter 3, we define the notion of middleware and describe the different models of middleware and two middleware architectures, namely CORBA and COM/DCOM. Moreover, in that chapter, we describe several QoS-providing middleware architectures, such as Real Time CORBA, QuO project [61], TAO [62] and Agilos [63]. In chapter 4 we describe the typical behavior of the applications that adapt themselves to fluctuations in resource availability and performance, and we extend the notion of End-to-End QoS by including characteristics that are strictly dependent on the applications. Moreover, we describe the motivations for providing support for QoS management at the middleware level. Also, we describe the method followed to confirm the ability of the middleware for supporting effective QoS policies, and describe the choice of the case studies. In chapter 5 we describe our $C^2LD$ adaptive mechanism for constructing responsive web services, constructed out of replica servers distributed over the Internet. In chapter 6, we describe our playout delay control mechanism for adjusting playout delays of unicast, voice-based communications across the Internet. In chapter 7, we propose an abstract middleware architecture to allow a designer to build adaptive applications, by composing reusable interoperable QoS-aware building blocks. Finally, in chapter 8, we provide some concluding remarks.
End-to-End Quality of Service

This section introduces the notion of Quality of Service as it is currently provided by the Internet. As of today, the Internet offers a point-to-point delivery service, which is based on the "best effort" delivery model. In this model, data are delivered to their destinations as soon as possible, but with no bandwidth or latency guarantees. Using protocols, such as TCP, the highest guarantee the network provides is reliable data delivery. This is adequate for traditional applications such as FTP and Telnet, but inadequate for applications requiring timeliness guarantees. For example, distributed multimedia applications need to communicate in real-time and are sensitive to the quality of service they receive from the network. For these applications to perform adequately and be widely used, Quality of Service (QoS) must be quantified and managed.

Up to now, QoS has been specified in terms of raw bandwidth and latency for network resources, or CPU and memory utilization for system resources, and the network infrastructures have been deployed to support real-time QoS and controlled end-to-end delays. However, at present the notion of QoS extends from the communication level up to the application level, in order to map QoS application requirements into low-level QoS parameters.

One distinguishing feature of QoS is that the application-level QoS parameters are subject to negotiation between applications and both the system and network components, in order to determine whether the application requiring QoS can be provided by the entire system. Several resource managers (brokers) can be deployed to provide support for the dynamic management of QoS.

In this chapter the traditional concept of Quality of Service at the network level is introduced, a taxonomy of the applications and their QoS requirements is provided, and a more recent notion of end-to-end QoS is presented.
Chapter 2. End-to-End Quality of Service

2.1 The Quality of Service

Standard Internet Protocol (IP)-based networks provide "best effort" data delivery by default. Best-effort IP confines the responsibility of meeting application specific requirements at the application level in the end-hosts; thus, the network typically can remain relatively simple [33]. This approach allows one to construct scalable applications, as evidenced by the ability of the Internet to support its phenomenal growth. As more hosts are connected, network service demands eventually exceed capacity, but service is not denied. Instead it degrades gracefully. Although the resulting variability in delivery delays (jitter) and packet loss do not adversely affect typical Internet applications (e.g., email, file transfer and Web applications) other applications cannot adapt to inconsistent service levels. Delivery delays cause problems for applications with real-time requirements, such as those that deliver multimedia, the most demanding of which are two-way applications such as telephony.

Increasing bandwidth is a necessary first step for accommodating these real-time applications, but it is still not sufficient to avoid jitter during traffic bursts. Even on a relatively unloaded IP network, delivery delays can vary enough to continue to adversely affect real-time applications. To provide adequate service -- some level of quantitative or qualitative determinism -- IP services must be supplemented. This requires extending the network software so as to distinguish traffic with strict timing requirements from traffic that can tolerate delay, jitter and loss of data. That is what Quality of Service (QoS) protocols are designed to do. QoS does not create bandwidth, but manages it so it is used more effectively to meet the wide range of application requirements. The goal of QoS is to provide some level of predictability and control beyond the current IP "best-effort" service.

There is no common or formal definition of QoS. However, there are a number of definitions at the communication level where the notion originated to describe certain technical characteristics of data transmission.

2.1.1 The traditional Notion of Quality of Service

Traditional QoS (ISO standard) was provided by the network layer of the communication system. For example, the ISO standard defines QoS as a concept for specifying how "good" the offered networking services are. So, QoS can be
characterized by a number of specific parameters. The OSI Reference Model has a number of QoS parameters describing the speed and reliability of transmission, such as throughput, transit delay, error rate and connection establishment failure probability. From the point of view of the QoS, the ability of a network to deliver the service needed by a specific network applications from end-to-end with some level of control over delay, loss, jitter, and/or bandwidth can be categorized into the following three levels of service:

- Best Effort Service -- basic connectivity with no guarantees. The Internet is an example of best effort level of service.
- Differentiated Service -- expedited handling for specific classes of traffic.
- Guaranteed Service -- a reservation of network resources to ensure that specific traffic gets a specific level of service it requires.

2.1.2 Network QoS Issues

Since today’s Internet interconnects multiple administrative domains (autonomous systems (AS)) based on IP technology, it is the concatenation of domain-to-domain data forwarding that provides end-to-end QoS delivery (Fig. 2.1).

![End-to-end QoS delivery](image)

**Figure 2.1:** End-to-end QoS delivery.

The management of QoS at the network level involves 5 different issues:

- QoS Specification (RSpec)
  
  In general, each flow should be able to express its delay, jitter, loss and bandwidth constrains.
- Flow Specification (TSpec)
  
  - Each flow should indicate to the network the load it will impose on it (peak rate, average rate).
The network guarantees satisfaction of the QoS specs if the flow does not violate its flow specs.

- **Admission control.**
  Given the QoS and flow specs, the network should perform admission control.
  A flow is admitted only if:
  - its constraints can be satisfied
  - none of the constraints of existing flows are violated

- **Signaling**
  The information regarding the admission of a new flow is propagated in the network to set up appropriate state

- **Policing**
  The network has to ensure that a malicious flow does not violate its flow specifications thereby hurting other flows

- **Router Support**
  - Edge routers would typically be responsible for admission control, policing, and signaling
  - Core routers will need to implement QoS-sensitive packet classification, scheduling and dropping policies

### 2.1.3 Approaches to Network QoS Support

Mainly, two are the approaches to provide QoS at the network level, IP based and ATM based, as depicted in Fig. 2.2. The former approach maintains the wide spread packet switching IP technology, and provides differentiated delivery services for individual flows or aggregates by adding "smarts" to the Net and improving on best effort service. The latter approach, instead, substitutes IP technology by providing connection-oriented services.

ATM provides both Data Link and Network OSI levels and is being developed to incorporate the QoS. ATM has the ability to handle a number of traffic types by statistically multiplexing together blocks of data called cells. ATM has five basic traffic classes [6], constant bit rate (CBR), real-time variable bit rate (rt-VBR), non-real-time variable bit rate (nrt-VBR), unspecified bit rate (UBR) and available bit rate (ABR).
These traffic classes are defined by parameters such as sustainable bit rate (SCR), peak cell rate (PCR), cell loss ratio (CLR) and cell delay variation (CDV). Before a connection can be established the network and host have to agree on the parameters to define the traffic class of the flow. This agreement is called a contract. Connection admission control and traffic policing are used to control the amount of traffic entering the network and so prevent excessive congestion. The admission control policy has to measure the available bandwidth within the network and calculate the effect any new connection will have on this available bandwidth. If a new connection is going to have a detrimental effect on existing connections then the new connection should be rejected. Traffic policing is responsible for ensuring that connections abide by their contracted bandwidth requirements. The traffic policing mechanism is only necessary at the switch connected to the source end node.

Although there is variety of choices, there are two major approaches for supporting QoS into IP based network:

- **Resource reservation** (integrated services): network resources are apportioned according to an application's QoS request, and subject to bandwidth management policy.

- **Prioritization** (differentiated services): network traffic is classified and apportioned network resources according to bandwidth management policy criteria. To enable QoS, network elements give preferential treatment to classifications identified as having more demanding requirements.

These two types of QoS managements can be applied to individual application "flows" or to flow aggregates, hence there are two other ways to characterize types of QoS:

**Figure 2.2:** Network QoS Taxonomy.
• *Fine Grained or Per Flow:* it provides "per flow" QoS guarantees and is represented by the integrated services (IntServ) framework. A "flow" is defined as an individual, uni-directional, data stream between two applications (sender and receiver), uniquely identified by a 5-tuple (transport protocol, source address, source port number, destination address, and destination port number). Each router on the path of a flow participates in admission control. The router keeps track of all admitted flows. For each flow, it allocates bandwidth, buffer space, and a priority. A new flow is not admitted unless the router can satisfy its bandwidth and storage needs as well as its delay constraint. For the signalling IntServ adopts the Reservation Protocol (RSVP) [7]. The problem with integrated services is that per-flow state is required in routers, thus this approach has been criticized for lack of scalability.

• *Coarse Grained or Per Aggregate:* An aggregate is simply two or more flows. Typically the flows will have something in common (e.g. any one or more of the 5-tuple parameters, a label or a priority number, or perhaps some authentication information). Flows are classified into a small number of classes and a different behavior for each class is defined. Applications, network topology and policy dictate which type of QoS is most appropriate for individual flows or aggregates. To accommodate the need for these different types of QoS, there are a number of different QoS protocols and algorithms:

• *ReSerVation Protocol (RSVP):* Provides the signaling to enable network resource reservation (otherwise known as Integrated Services). Receiver initiates network resource reservation by sending an RESV message. The amount of reservation guarantees satisfaction of timing, bandwidth, and buffer size constraints. Reservation establishes soft state along the path to the sender and sets aside the required resources. Soft state is refreshed periodically by the receiver or else it times out canceling the reservation. Although typically used on a per-flow basis, RSVP is also used to reserve resources for aggregates (as we describe in our examination of QoS architectures).

• *Differentiated Services (DiffServ)[8]:* Provides a coarse and simple way to categorize and prioritize network traffic (flow) aggregates. Differentiated services classify all flows into a small number of classes and define a different “per-hop-behavior” for each class. 6 bits out of the Type-of-Service byte in the IP header are
used to define per-hop behaviors. Clients pay the network provider for a certain profile of traffic. Edge routers mark client packets. The 6-bit per-hop behavior code in the IP header tells each core router what to do with the packet.

- **Multi Protocol Labeling Switching (MPLS)** [9]: Provides bandwidth management for aggregates via network routing control according to labels in (encapsulating) packet headers.

- **Subnet Bandwidth Management (SBM)** [10]: Enables categorization and prioritization at Layer 2 (the data-link layer in the OSI model) on shared and switched IEEE 802 networks.

### 2.2 Applications QoS

Different applications running on the same distributed system may have different subsets of relevant QoS parameters, with different values required, and some parameters may be not mutually independent, and may be time depending. For instance, there are 5 types of QoS parameters for the distributed multimedia applications (such as video-on-demand services, teleconferencing [34], computer supported cooperative work [35, 36] and tele robotic applications [37]):

- performance-oriented, e.g. end-to-end delay, bit rate;
- format-oriented, e.g. video resolution, frame rate, storage format, compression scheme;
- synchronization-oriented, e.g. the skew between the beginning of audio and video sequences;
- cost-oriented, e.g. connection and data transmission charges, copyright fees;
- user-oriented, e.g. subjective image and sound quality.

We are particularly interested in the QoS application requirements that involve the network.

### 2.2.1 Applications Taxonomies

The taxonomy of the networked applications (see Fig. 2.3) revolves around the timely delivery of packets, but addresses other qualitative issues as well. It is based on the simple classification of applications into two principal categories.
First, there are applications that are sensitive to the delay incurred by their data flows as they traverse the network; these applications are real-time applications. If a packet arrives late to a real-time application, it is no longer useful.

Second, there are applications that always wait for their data to arrive; they are called elastic applications.

**Figure 2.3:** Applications Taxonomy.

**Elastic Applications:** traditional applications such as remote terminal (e.g., Telnet), file transfer (e.g., FTP), name service (e.g., DNS), and electronic mail (e.g., SMTP) are rather elastic in nature, in that they tolerate packet delays and packet losses rather gracefully, and so they are rather well served by the current Internet's best effort service. Also Web browsing may be thought of as an elastic interactive application. A rough set of categorizations for elastic applications might include:

- Interactive burst (Telnet, X, NFS)
- Interactive bulk transfer (FTP)
- Asynchronous bulk transfer (electronic mail, FAX)

The categories are listed here in approximately decreasing order of delay sensitivity. Note that it is the average delay of data that impacts the performance of elastic applications.

**Real Time Applications:** At the other extreme of delay sensitivity are applications with real time requirements. These application are considered playback applications.

Data originating at a source is encoded, packetized, sent across the network, decoded by the receiver, and attempted to be replayed at the destination as a replica of what was encoded initially. However packets are likely to experience a range of delays.
across the network, and earlier packets may take longer to transit the network than later packets. Consequently a real-time application typically buffers the arriving packets at the destination. The buffer allows the application not only to smooth out the delay variation, or jitter, but also to re-order packets if necessary. In any event, data is reconstituted at the destination by establishing a playback point, the point in time after which a packet is delivered. That point is usually a fixed offset from the original departure time. In practical terms a packet is buffered according to this departure time, then a buffering delay is added to smooth the play out of the data stream. Depending on the application, the buffering delay may be quite short on the order of milliseconds, as in a conversational audio application, or seconds and potentially minutes, as in a streaming video-on-demand application. Packets arriving after the playback point are discarded. These are considerations that impact the design of buffering at the end-systems.

In choosing a playback point, an application approximates the amount of delay it is able to tolerate. This approximation may be based on a delay bound promised by a particular service class, on actual measurements, or on predictions about future packet delays. The delay bound need not be fixed for the lifetime of the data flow.

The performance of a playback application is measured by two parameters: latency and fidelity. Some playback applications are particularly sensitive to latency, such as a distributed music performance that relies on interactions between the end systems, whereas other applications are less so, such as an Internet lecture that behaves more like a unidirectional streaming application. Likewise, applications exist that are quite demanding of fidelity of the real-time signal, whereas others are more willing to forego exactness. This latter dimension of performance, leads to two further classifications of the real-time playback applications into those that are tolerant and intolerant of signal fidelity. For those that are tolerant a predictive service class is proposed. For intolerant applications a service model called guaranteed service is proposed.

The calibration of the offset delay is quite important, as it determines the latency of the application. Furthermore, a realistic offset is critical to the fidelity of the application. If the offset delay is incorrect, more packets arrive late and are dropped, making the data stream less complete. Alternatively late packets may cause the application to adjust its playback point, which introduces distortion in the signal. Applications of this sort are considered adaptive playback applications. A final reaction
to late packets might be for the application to alter its traffic generation scheme, e.g., switch to a less demanding audio or video encoding. These applications are known as *rate-adaptive* playback applications. Although this technique may reduce delay, it necessarily compromises data resolution. All of the previous examples show that late packets decrease playback fidelity.

While a tolerant application has the option of selecting one of these methods for handling late packets (the appropriateness of one over the other depends on the application itself), an intolerant application must use a fixed offset delay to avoid loss of fidelity. The intolerant application must choose a “perfectly reliable upper bound on delay”, the maximum delay of any packet, whereas the tolerant application can resort to using a “fairly reliable upper bound”, which is conservatively predictive [38]. The motivation for using predictive service over guaranteed service is that it offers better network utilization and presumably lower cost.

A simpler taxonomy involves real time constraints of the applications. Real-time constraints are classified as *hard*, *firm* and *soft*, depending on the consequences of the constraint being violated. A task with *hard* real time constraint has disastrous consequences if its constraint is violated (for instance, a missile control). A task with *firm* real time constraint has no value to the system if its constraint is violated. A task with *soft* real time constraint has decreasing, but usually nonnegative, value to the system if its constraint is violated.

![Figure 2.4: Guaranties required by the Applications](image)

**Figure 2.4:** Guaranties required by the Applications
Another taxonomy (see Fig. 2.4) classifies the networked applications depending on the guarantees they require from the underlying network; there are three classes of applications:

a) **Guaranteed** – all deadlines are guaranteed to be met all the time. This application gets highest priority and will need resources to be reserved considering the worst case situation.

b) **Statistical** – deadlines are guaranteed to be met with a certain probability (e.g. a service that guarantees that 90% of the deadlines will be met over an interval). The statistical behavior of this class needs to be monitored and maintained.

c) **Best Effort** – no guarantees are given for meeting deadlines. Deadlines are met on a best-effort basis with the resources leftover from those reserved for guaranteed services. These applications can be pre-empted by higher priority classes, and no resources are reserved.

Most of the applications that require guarantees from the network are flexible, in the sense that they can tolerate a QoS range of input quality and resource availability beyond a certain minimum level, and can improve their performance, provided that a larger share of resources is available. Flexible applications monitor the distributed resource and dynamically adapt themselves by modifying their QoS requirements and by negotiating a new QoS level with the environment. If resources above the minimum requirements are shared among all applications, statistical multiplexing gain can be improved. In addition, for the flexible applications that involve interactive activities that cannot be predicted beforehand, it may be hard or impossible to specify a maximum demand for QoS.

Finally, it is worth pointing out that “an application is best-effort based” means it does not require guarantees, but it may adopt several software architectures, mechanisms and policies in order to provide the users with the suitable QoS. For instance, the QoS of a web service, as perceived by the users, may be increased by using caching at the browser side, pre-fetching at the proxy, and replication at the server side (several web server replicas) and at the browser side (mechanism for binding the best replica). Although these three different approaches do not require guarantees from the network, they can provide the user with timely responses, thus increasing the user's satisfaction.

In particular, **adaptive** best-effort applications monitor the performances of the involved resources and dynamically adapt themselves by modifying their behavior in
order to meet the QoS requirements of the user, but unlike flexible applications they do not reserve network resources. The negotiation phase of flexible applications becomes the adaptation phase of best effort applications. For instance, the web browser involved in the previously cited web service may periodically monitor the performances of each web replica server, in order to retrieve a web resource to the best replica.

2.3 The Notion of End-To-End QoS

The original QoS parameters apply mostly to lower network protocol layers, and are not meant to be directly observable or verifiable by the application. Consequently, the resulting QoS coverage of OSI as a whole is incomplete and even inconsistent. This situation, while acceptable when communication networks were used mostly for non-time-dependent data, is no longer satisfactory with respect to the new requirements stemming from distributed multimedia systems. As time-dependent data are becoming prevalent in multimedia applications, the entire distributed system must participate in providing the guaranteed performance levels.

The QoS notion must be extended because many other services contribute to the end-to-end service quality. This in turn means that both the network and the end-system have to contribute towards achieving this end-to-end QoS. Fig. 2.5 above illustrates a basic diagram of the entities involved in providing this end-to-end QoS. Beyond its
intuitive meaning as system characteristics that influence the perceived quality of an application, there is little consensus on the precise meaning, let alone the formal definition of QoS. For instance, the ITU/ISO Reference Model for Open Distributed Processing [39] refers to QoS as "A set of quality requirements on the collective behavior of one or more objects". According to ISO, "QoS characteristics are intended to be used to model the actual behavior of systems. It is defined independently of the means by which it is represented or controlled". This type of definition is too general, since it tends to include all system parameters without distinction. In [40] the following working definition is proposed: "By quality of service we mean the set of those quantitative and qualitative characteristics of a distributed multimedia system, which are necessary in order to achieve the required functionality of an application. This includes the presentation of multimedia data to the user, and in general the user's satisfaction with the application".

2.3.1 End-To-End QoS Entities

To discuss further QoS and resource management, we adopt a layered model of the end-to-end networked applications with respect to QoS (see Fig. 2.6). That model consists of three layers: application, system (including communication services and operating system services), and network. Above the application may or may not reside a human user. This implies the introduction of QoS in the application (application QoS), in the end system (system QoS) and in the network (network QoS). In the case of having a human user, the QoS model may also have a user QoS specification.

![Figure 2.6: QoS Levels](image)

The system layer includes (see Fig. 2.7) several modules [41], such as task scheduler, memory management and I/O subsystem. Therefore, the QoS system parameters mirror the requirements on CPU scheduling (e.g., task start time, priority, duration and deadline), on memory management (buffer allocation) and on I/O management (for
example, the maximal frame rate for a video device). In this view, the QoS requirement originates with an application process which conveys it in terms of QoS parameters to other system components. This is generally followed by a negotiation process whereby the components of the system determine if collectively they are capable of satisfying the requested QoS level. The QoS of a given system is expressed as a set of (parameter-value) pairs, sometimes called a tuple; each parameter can be considered as a typed variable whose values can range over a given set. Note that different applications running on the same distributed system may have different subsets of relevant QoS parameters, with different values required, and some parameters may not be mutually independent.

Resource management is a fundamental task performed by the operating systems. Most current mainstream operating systems implement resource management policies which are oriented towards overall system throughput and fairness. They perform reasonably well for a mix of interactive, batch processing, or server applications. Real-time operating systems are designed to give hard guarantees on resource allocations to allow applications to perform time-critical tasks. This is achieved by allocating fixed shares of resources to processes for their entire lifetime, or at least for a relatively long

**Figure 2.7**: QoS End System Components
Multimedia operating systems (e.g., [42], [43], [44], [45], [46]) give soft real-time guarantees for resource allocations to processes which may be renegotiated dynamically at runtime, encouraging applications to adapt to the changing overall system load. This is usually achieved through explicit resource allocation and revocation. This process is commonly referred to as QoS-Management. In conclusion, one distinguishing feature of QoS is that these QoS parameters are subject to negotiation between applications and both system and network components, in order to determine if these QoS parameters may be provided by the entire system.

2.3.2 End-To-End QoS Management

Many networked multimedia applications are delay-sensitive, and require services with guarantees of resource availability and timeliness. An end-to-end protocol architecture needs QoS guarantees at multiple layers. Delivering end-to-end QoS requires an architecture for resource management at the system end-points (e.g., computer workstation hosts), as well as at the underlying network.

In [11] an end-point entity called the QoS Broker is described, that operate over networks based on Asynchronous Transfer Mode (ATM). The broker orchestrates resources at the end-points, coordinating resource management across layer boundaries. As an intermediary, it hides implementation details from applications and per-layer resource managers. Broker uses services such as translation, admission and negotiation to properly configure the system to meet the application requirements. Configuration is achieved via QoS negotiation resulting in one or more connections through the communications system. The negotiation involves all components of the communication system needed for the setup. An important property of the broker is its role as an active intermediary that insulates cooperating entities from operational details of other entities. The broker manages communication among the entities to create the desired system configuration. It can be viewed as a software engineering technique for distributed multimedia system architectures.

In [12] the concept of QoS Broker is extended in order to provide QoS management for different network technologies, other than ATM. The proposed Distributed Resource Controller (DRC) technology aims to unify network services (e.g., DiffServ, Intserv, and ATM) and application QoS provisioning by introducing a middleware system and a set of generic interfaces. DRC uses the CORBA-based
Finally, in [47] a middleware approach for QoS management, called Adaptware, is proposed, that enables application designers to express “flexible” performance requirements, and adapts application timing behavior to platform capacity and load condition.
Chapter 3

The Middleware

Distributed applications can be constructed out of interoperable software components, running at arbitrary nodes, distributed over a communication network. In order to ease the application development task, it is convenient to shield the application developer from the possible hardware and software heterogeneity of the network nodes and the communication sub-system. To this end, a common software platform can be made available to the application components in the network nodes, typically above the operating system interface; this platform provides those components with a standard Application Programming Interfaces (API).

Currently, two major approaches have been deployed in the design and implementation of such software platforms. For the scope of this thesis we term these two approaches:

a) the transportable computation approach [48], and

b) the middleware approach [13].

In this chapter we examine these two approaches, in isolation. Specifically, the next section describes the concept of transportable computation; section 3.2 illustrates different models for the middleware service, section 3.3 provides a short introduction to the CORBA model, and section 3.4 describes the COM/DCOM architecture and its call back mechanism. Finally, section 3.5 describes several platforms for supporting QoS-sensitive applications, in particular Real Time CORBA (RT-CORBA) [60], QuO project [61], TAO [62] and Agilos [63].

3.1 The Transportable Computation Approach

This approach is based on the concept of layering a virtual machine engine above the operating system of a computer. That virtual machine engine provides an open API
independent from the operating system and can be built as an interpreter. Applications can be implemented by transporting portable code across the network. Mobile code technologies propose the use of interpreted programming languages, as opposed to compiled ones, to deal with the issue of portability of software programs. Many interpreted languages, such as Java [49], have been proposed. The Java source code is compiled to produce an intermediate code, the Java bytecode, that can be executed in the context of an interpreting system, the JVM (Java Virtual Machine), that translates the compiled bytecode into machine instructions for the target local system. The capability of a Java application to invoke methods of another Java application is supported by the Java Remote Method Invocation (RMI) standard [50].

Using RMI as infrastructure to support communications, the Jini™ Technology [51] extends the Java application environment from a single virtual machine to a network of machines, enabling applications to transparently share services and resources over a network, as depicted in Figure 3.1.

![Figure 3.1: Jini architecture](image)

The RMI approach and the Jini approach represent an alternative to CORBA, when Java objects only are used, although CORBA offers a more generic and language-independent approach and provides a wider set of generic services. Even so, Java+RMI represents a widespread environment for the development of applications and services.

### 3.2 Middleware models

As pointed out in [13], the term *middleware* refers to a distributed platform of interfaces and services that reside ‘between’ the application and the operating system and aim to facilitate the development, deployment and management of distributed applications, as depicted in Fig. 3.2.
Middleware services provide a more functional set of Application Programming Interfaces (API) than the operating system and network services [14] so as to allow an application to:

- locate transparently across the network, providing interaction with another application or service
- be independent from network services
- be reliable and available
- scale up in capacity without losing function.

Different middleware platforms support different programming models. The most popular model is the so-called object based middleware model; however other popular paradigms are available. More precisely, we recognize four different middleware models:

- Object Request Brokers,
- Message-Oriented Middleware,
- Remote Procedure Call, and
- Transaction processing (TP) monitors.

### 3.2.1 Object Request Brokers

Object Request Brokers (ORBs) enable the objects that comprise an application to be distributed and shared across heterogeneous networks. In the object based middleware model applications are structured into (possibly distributed) objects that interact via location transparent method invocations.
The relevant functions of an ORB technology are:
- interface definition,
- location and possible activation of remote objects,
- communication between clients and object.

An object request broker provides a directory of services and provides assistance in establishing connections between clients and these services [16]. Figure 3.3 illustrates some of the key ideas of ORB design.

![Figure 3.3: Object Request Broker](image)

The ORB must support many functions in order to operate consistently and effectively; however, many of these functions are hidden from the user of the ORB. It is the responsibility of the ORB to provide the illusion of locality, in other words, to make it appear as if the object is local to the client, while in reality it may reside in a different process or machine. Thus the ORB provides a framework for cross-system communication between objects. This is the first technical step toward interoperability of object systems.

The next technical step toward object system interoperability is the communication of objects across platforms. An ORB supports transparent access to objects by hiding the object implementation details from the clients. There are many ways of implementing the basic ORB concept; for example, ORB functions can be compiled into clients, can be separate processes, or part of an operating system kernel. These basic design decisions may be fixed in a single product; or there may be a range of choices left to the ORB implementer.

Prime examples of this type of middleware are the Object Management Group's Common Object Request Broker Architecture (OMG's CORBA) [15, 16] and Microsoft's COM/DCOM (Component Object Model and Distributed Component
Chapter 3. The Middleware

Object Model) [17, 18, 52] (Microsoft has now done away with DCOM and moved on the current name, COM+). Both of these platforms offer an interface definition language (IDL) which is used to abstract over the fact that objects can be implemented in any suitable programming language, an object request broker which is responsible for transparently directing method invocations to the appropriate target object, and a set of services (e.g. naming, time, transactions, replication etc.) which further enhance the distributed programming environment.

### 3.2.2 Message Oriented Middleware

Message-Oriented Middleware (MOM) provides program-to-program data exchange, enabling the creation of distributed applications. MOM supports asynchronous calls between the applications and requires the recipients of messages to interpret their meaning and to take appropriate actions. This model mainly employs ‘single shot’ communications rather than the request-reply style communication found in object based middleware.

![Figure 3.4: Message Oriented Middleware.](image)

Message-oriented middleware, as shown in Figure 3.4 [53], is software that resides in both portions of a client/server architecture. Exchanged messages can contain formatted data, requests for action, or both. Typically, MOM systems provide a message queue between interoperating processes, so if the destination process is busy, the message is held in a temporary storage location until it can be processed. Most implementations of MOM support also synchronous message passing as well.

MOM is most appropriate for event-driven applications (e.g. process control, Internet news channels) rather than applications that must monitor and react to changes
in their environment. When an event occurs, the client application hands off to the messaging middleware the responsibility of notifying a server that some action needs to be taken. It is claimed that event based middleware has potentially better scaling properties for such applications than object based middleware. Examples of MOM are IBM's MQSeries [19] and Talarian's SmartSocket [20].

### 3.2.3 Remote Procedure Call

Remote Procedure Calls (RPCs) enable the logic of an application to be distributed across the network. This concept defines daemons or server processes that run continuously on a machine and respond to requests sent over the network. Program logic on remote systems can be executed as simply as calling a local routine. In fact, in order to access the remote server portion of an application, special function calls, RPCs, are embedded within the client portion of the client/server application program. Because they are embedded, RPCs do not stand alone as a discreet middleware layer. When the client program is compiled, the compiler creates a local stub for the client portion and another stub for the server portion of the application. These stubs are invoked when the application requires a remote function and typically support synchronous calls between clients and servers. These relationships are shown in Figure 3.5 [53].

![Figure 3.5: RPC model.](image)

By using RPCs, the complexity involved in the development of distributed processing is reduced by keeping the semantics of a remote call the same whether or not the client and server are on the same system.
RPC increases the flexibility of an architecture by allowing a client component of an application to employ a function call to access a server on a remote system. RPC allows the remote component to be accessed without knowledge of the network address or any other lower-level information. Most RPCs use a synchronous, request-reply (sometimes referred to as "call/wait") protocol, which involves blocking of the client until the server fulfills its request. Asynchronous ("call/nowait") implementations are available but are currently the exception because they are difficult to implement [54].

3.2.4 Transaction Processing Monitors

Transaction processing (TP) monitors technology provides the distributed client/server environment with the ability to efficiently and reliably develop, run, and manage transaction applications.

**Figure 3.6:** Transaction processing monitors technology.

TP has been developed originally for mainframe environments. TP monitors sit between front-end applications and back-end databases to manage the writing and reading of transactional data. TP monitor technology is used in data management, network access, security systems, delivery order processing, airline reservations, and customer service. TP monitor technology can provide application services to thousands of clients by multiplexing client transaction requests (by type) onto a controlled number
of processing routines that support particular services, as depicted in Fig. 3.6. In addition, as described in details in [55], TP monitor technology includes numerous management features, such as restarting failed processes, dynamic load balancing, and enforcing consistency of distributed data.

Examples of TP monitors are IBM's CICS and BEA's Tuxedo [21]. Advanced versions of the TP monitors are the Object monitors, also called object TP monitors. Object Monitors provide TP monitor functionality but are built according to object-oriented specifications, like the object request broker models. One example is IBM Component Broker VisiBroker Integrated Transaction Server.

3.3 CORBA

The Common Object Request Broker Architecture (CORBA) is a specification of a standard architecture for object request brokers (ORBs). A standard architecture allows vendors to develop ORB products that support application portability and interoperability across different programming languages, hardware platforms, operating systems, and ORB implementations.

"Using a CORBA-compliant ORB, a client can transparently invoke a method on a server object, which can be on the same machine or across a network. The ORB intercepts the call, and is responsible for finding an object that can implement the request, passing it the parameters, invoking its method, and returning the results of the invocation. The client does not have to be aware of where the object is located, its programming language, its operating system or any other aspects that are not part of an object's interface" [15]. The "vision" behind CORBA is that distributed systems are conceived and implemented as distributed objects. The interfaces to these objects are described in a high-level, architecture-neutral specification language that also supports object-oriented design abstraction.

The CORBA specification was developed by the Object Management Group (OMG), an industry group with over six hundred member companies representing computer manufacturers, independent software vendors, and a variety of government and academic organizations [15]. Thus, CORBA specifies an industry/consortium standard, rather than an international standard such as these emanated by the IEEE, or the ISO or the ANSI.
### 3.3.1 CORBA Structure

The Object Management Architecture (OMA), shown in Figure 3.7, is itself a specification (actually, a collection of related specifications) that defines a broad range of services for building distributed applications.

![Diagram of Object Management Architecture](image)

**Figure 3.7:** Object Management Architecture.

OMA services are partitioned into three categories: CORBAServices, CORBAFacilities, and ApplicationObjects.

The ORB (whose details are specified by CORBA, as detailed in the following) is a communication infrastructure through which applications access these services, and through which objects interact with each other. CORBAServices, CORBAFacilities, and ApplicationObjects define different categories of objects in the OMA; these objects (more accurately object *types*) define a range of functionality needed to support the development of distributed software systems.

CORBAServices are considered fundamental for building non-trivial distributed applications. These services currently include:

1. Naming Service, provides the ability to bind a name to an object.
2. Event Service, supports asynchronous message-based communication among objects.
3. Lifecycle Service, defines conventions for creating, deleting, copying and moving objects.
4. Persistence Service, provides a means for retaining and managing the persistent state of objects.
5. Transaction Service, supports multiple transaction models, including mandatory "flat" and optional "nested" transactions.

6. Concurrency Service, supports concurrent, coordinated access to objects from multiple clients.

7. Relationship Service, supports the specification, creation and maintenance of relationships among objects.

8. Externalization Service, defines protocols and conventions for externalizing and internalizing objects across processes and across ORBs.

CORBA Facilities may be useful for distributed applications in some settings, but are not considered as universally applicable as CORBA Services. These "facilities" include: user interface, information management, system management, task management, and a variety of "vertical market" facilities in domains such as manufacturing, distributed simulation, and accounting.

Application Objects provide services that are particular to an application or class of applications. These are not (currently) a topic for standardization within the OMA, but are usually included in the OMA reference model for completeness, i.e., objects are either application-specific, support common facilities, or are basic services.

CORBA defines also the components of the ORB entity, as depicted in Fig. 3.8 that represents an expansion of the ORB component of the Fig. 3.7. These ORB components are:

**Figure 3.8:** Object Management Architecture.
1. ORB Core. Is the CORBA runtime infrastructure. The interface to the ORB Core is not defined by CORBA, and will be vendor proprietary.

2. ORB Interface. A standard interface (defined in IDL) to functions provided by all CORBA-compliant ORBs.

3. IDL Stubs. Generated by the IDL processor for each interface defined in IDL. Stubs hide the low-level networking details of object communication from the client, while presenting a high-level, object type-specific application programming interface (API).

4. Dynamic Invocation Interface (DII). An alternative to stubs for clients to access objects. While stubs provide an object type-specific API, DII provides a generic mechanism for constructing requests at run time (hence "dynamic invocation"). An interface repository (another CORBA component not illustrated in Figure 2) allows some measure of type checking to ensure that a target object can support the request made by the client.

5. Object Adaptor. Provides extensibility of CORBA-compliant ORBs to integrate alternative object technologies into the OMA.

6. IDL Skeletons. The server-side (or object implementation-side) analogue of IDL stubs. IDL skeletons receive requests for services from the object adaptor, and call the appropriate operations in the object implementation.

7. Dynamic Skeleton Interface (DSI). The server-side (or object implementation-side) analogue of the DII. While IDL skeletons invoke specific operations in the object implementation, DSI defers this processing to the object implementation. This is useful for developing bridges and other mechanisms to support inter-ORB interoperation.

One element (not depicted in Figure 3) that is crucial to the understanding of CORBA is the interface definition language (IDL) processor. All objects are defined in CORBA (actually, in the OMA) using IDL. IDL is an object-oriented interface definition formalism that has some syntactic similarities with C++. Unlike C++, IDL can only define interfaces; it is not possible to specify behavior in IDL. Language mappings are defined from IDL to C, C++, Ada95, and Smalltalk80.

An important point to note is that CORBA specifies that clients and object implementations can be written in different programming languages and execute on different computer hardware architectures and different operating systems, and that
clients and object implementations can not detect any of these details about each other. Put another way, the IDL interface completely defines the interface between clients and objects; all other details about objects (such as their implementation language and location) can be made "transparent."

To make a request, the Client can use the Dynamic Invocation interface (the same interface independent of the target object’s interface) or an OMG IDL stub (the specific stub depending on the interface of the target object). The Client can also directly interact with the ORB for some functions.

The Object Implementation receives a request as an up-call either through the OMG IDL generated skeleton or through a dynamic skeleton. The Object Implementation may call the Object Adapter and the ORB while processing a request or at other times. Definitions of the interfaces to objects can be defined in two ways. Interfaces can be defined statically in an interface definition language, called the OMG Interface Definition Language (OMG IDL). This language defines the types of objects according to the operations that may be performed on them and the parameters to those operations. Alternatively, or in addition, interfaces can be added to an Interface Repository service; this service represents the components of an interface as objects, permitting run-time access to these components.

### 3.3.2 CORBA Interoperability

ORB interoperability specifies a comprehensive, flexible approach to supporting networks of objects that are distributed across and managed by multiple, heterogeneous CORBA-compliant ORBs.

The inter-ORB bridge support assures that content and semantics of object are mapped from the form appropriate to one ORB to that of another, so that users of any given ORB only see their appropriate content and semantics.

The General Inter-ORB Protocol (GIOP) element specifies a standard transfer syntax (low-level data representation) and a set of message formats for communications between ORBs. The GIOP is specifically built for ORB to ORB interactions and is designed to work directly over any connection-oriented transport protocol that meets a minimal set of assumptions. It does not require or rely on the use of higher level RPC mechanisms. While versions of the GIOP running on different transport services would
not be directly interoperable, their commonality would allow easy and efficient bridging between such networking domains.

Figure 3.9: Entity devoted to Interoperability.

The Internet Inter-ORB Protocol (IIOP) element specifies how GIOP messages are exchanged using TCP/IP connections. The IIOP specifies a standardized interoperability protocol for the Internet, providing “out of the box” interoperation with other compatible ORBs based on the most popular product- and vendor-neutral transport layer. In the Fig. 3.9 [56] the two entities are represented, embedded into the general CORBA architecture. The IIOP’s relationship to the GIOP is similar to that of a specific language mapping to OMG IDL; the GIOP may be mapped onto a number of different transports, and specifies the protocol elements that are common to all such mappings. The GIOP by itself, however, does not provide complete interoperability, just as IDL cannot be used to built complete programs. The IIOP, and other similar mappings to different transports, are concrete realizations of the abstract GIOP definitions.
3.4 COM/DCOM

COM [17] refers to both a specification and implementation developed by Microsoft Corporation which provides a framework for integrating components. This framework supports interoperability and reusability of distributed objects by allowing developers to build systems by assembling reusable components from different vendors which communicate via COM.

COM defines an application programming interface (API) to allow for the creation of components for use in integrating custom applications or to allow diverse components to interact. However, COM defines a binary structure for the interface between the client and the object. This binary structure provides the basis for interoperability between software components written in arbitrary languages. As long as a compiler can reduce language structures down to this binary representation, the implementation language for clients and COM objects does not matter - the point of contact is the run-time binary representation. Thus, COM objects and clients can be coded in any language that supports Microsoft's COM binary structure.

Distributed COM [18] is an extension to COM that allows network-based component interaction. While COM processes can run on the same machine but in different address spaces, the DCOM extension allows processes to be spread across a network. With DCOM, components operating on a variety of platforms can interact, as long as DCOM is available within the environment.

It is best to consider COM and DCOM as a single technology. In fact, COM and its DCOM extensions are merged into a single runtime. This single runtime provides both local and remote access.

While COM and DCOM represent "low-level" technology that allows components to interact, OLE [56], ActiveX, COM+ and MTS [57] represent higher-level application services that are built on top of COM and DCOM.

3.4.1 COM/DCOM Architecture

COM is a binary compatibility specification and associated implementation that allows clients to invoke services provided by COM-compliant components (COM objects). As shown in Fig. 3.10, services implemented by COM objects are exposed through a set of interfaces that represent the only point of contact between clients and the object. A
COM object can support any number of interfaces. An interface provides a grouped collection of related methods.

COM objects and interfaces are specified using Microsoft Interface Definition Language (IDL), an extension of the DCE Interface Definition Language standard [59]. To avoid name collisions, each object and interface must have a unique identifier. Every COM object runs inside of a server. A single server can support multiple COM objects.

![Diagram](image)

**Figure 3.10:** Client Using COM Object Through an Interface Pointer.

As shown in Figure 3.11 there are three ways in which a client can access COM objects provided by a server:

1. In-process server: The client can link directly to a library containing the server. The client and server execute in the same process. Communication is accomplished through function calls.

2. Local Object Proxy: The client can access a server running in a different process but on the same machine through an inter-process communication mechanism. This mechanism is actually a lightweight Remote Procedure Call (RPC).

3. Remote Object Proxy: The client can access a remote server running on another machine. The network communication between client and server is accomplished through DCE RPC. The mechanism supporting access to remote servers is called DCOM.
If the client and server are in the same process, the sharing of data between the two is simple. However, when the server process is separate from the client process, as in a local server or remote server, COM must format and bundle the data in order to share it. This process of preparing the data is called marshalling. Marshalling is accomplished through a "proxy" object and a "stub" object that handle the cross-process communication details for any particular interface (depicted in Fig. 3.12). COM creates the "stub" in the object's server process and has the stub manage the real interface pointer. COM then creates the "proxy" in the client's process, and connects it to the stub. The proxy then supplies the interface pointer to the client.

The same proxy/stub mechanism is used when the client and server are on different machines. However, the internal implementation of marshalling and unmarshalling differs depending on whether the client and server operate on the same machine (COM) or on different machines (DCOM). Given an IDL file, the Microsoft IDL compiler can create default proxy and stub code that perform all necessary marshalling and unmarshalling.

**Figure 3.11:** Three Methods for Accessing COM Objects.
All COM objects are registered with a component database. As shown in Fig. 3.13, when a client wishes to create and use a COM object:

1. It invokes the COM API to instantiate a new COM object.
2. COM locates the object implementation and initiates a server process for the object.
3. The server process creates the object, and returns an interface pointer at the object.
4. The client can then interact with the newly instantiated COM object through the interface pointer.

An important aspect in COM is that objects have no identity, i.e. a client can ask for a COM object of some type, but not for a particular object. However, there are mechanisms (called monikers) to simulate the object identity in COM. Every time that COM is asked for a COM object, a new instance is returned. The main advantage of this policy is that COM implementations can pool COM objects and return these pooled objects to requesting clients. Whenever a client has finished using an object, the instance is returned to the pool.
COM includes interfaces and API functions that expose operating system services, as well as other mechanisms necessary for a distributed environment (naming, events, etc.).

### 3.4.2. The Call Back mechanism of COM/DCOM

COM/DCOM implements some very interesting functionalities, that support asynchronous calls and event notification and management. These functionalities can be provided by the so called Connectable Objects. Some objects require a way to notify clients that an event has occurred. COM allows such objects to define outgoing interfaces to clients as well as incoming interfaces. Thus, an object can define an interface it would like to use (e.g., a notification interface), and clients can implement that interface. This enables a two-way communication between the clients and the objects.

More precisely, we said that from the object’s perspective, the interfaces were “incoming”. “Incoming,” in the context of a client-object relationship, implies that the object “listens” to what the client has to say. In other words, incoming interfaces and their member functions receive input from the outside. COM also defines mechanisms where objects can support “outgoing” interfaces. Outgoing interfaces allow objects to have two-way conversations, so to speak, with clients. When an object supports one or more outgoing interfaces, it is said to be connectable. One of the most obvious uses for outgoing interfaces is for event notification. This section describes Connectable Objects, as depicted in Fig. 3.14.
A connectable object (also called a source) can have as many outgoing interfaces as it likes. Each interface is composed of distinct member functions, with each function representing a single event, notification, or request. Events and notifications are equivalent concepts (and interchangeable terms), as they are both used to tell the client that something interesting happened in the object. Events and notifications differ from a request in that the object expects response from the client. A request, on the other hand, is how an object asks the client a question and expects a response.

In all of these cases, there must be some client that listens to what the object has to say and uses that information wisely. It is the client, therefore, that actually implements these interfaces on objects called sinks. From the sink’s perspective, the interfaces are incoming, meaning that the sink listens through them. A connectable object plays the role of a client as far as the sink is concerned; thus, the sink is what the object’s client uses to listen to that object.

An object doesn’t necessarily have a one-to-one relationship with a sink. In fact, a single instance of an object usually supports any number of connections to sinks in any number of separate clients. This is called multicasting (Note that this usage of the term multicasting may differ from what some readers are accustomed to. In some systems multicasting is used to describe a connection-less broadcast. Connectable objects are obviously connection oriented). In addition, any sink can be connected to any number of objects.
3.5 QoS-Oriented Middleware Architectures

Several platforms for supporting QoS-sensitive applications have been proposed. The majority of those platform have been designed in order to meet real-time requirements. Relevant examples of these platforms includes: Real Time CORBA (RT-CORBA) [60], QuO project [61], TAO [62] and Agilos [63].

3.5.1 Real Time CORBA

RT CORBA is an evolution of the standard CORBA v2.0 to support high performance and real-time application systems. The features of the OMG Messaging specification, that provides several asynchronous method invocation models, are integrated into this new CORBA version. An ORB endsystem [64] consists of network interfaces, operating system I/O subsystems and communication protocols, and CORBA-compliant middleware components and services.

Strict control over the scheduling and execution of processor resources is essential for many fixed-priority real-time applications. Therefore, the RT-CORBA specification enables client and server applications to (1) determine the priority at which CORBA invocations will be processed, (2) allow servers to predefine pools of threads and their priority, in order to control the concurrency level within server ORBs and the applications, and (3) ensure that intra-process thread synchronizers have consistent semantics.

Historically, the CORBA specification and conventional ORBs have supported location transparency, i.e., applications cannot detect whether components are distributed or collocated in the same node. Moreover, the features of the underlying OS, network, and/or bus are encapsulated in what can be thought of as a black box. Although this encapsulation is useful for applications with best-effort QoS requirements, it is inadequate for applications with more stringent QoS requirements. In order to allow applications to control the underlying communication protocols and endsystem resources, therefore, the RT-CORBA specification defines standard interfaces that can be used to specify ORB-specific and transport-specific protocol properties that control various communication protocol features, such as ATM virtual circuits or Internet RSVP [31] traffic specification (i.e. an instance of the Inter-ORB protocol (IOP)). From our point of view, these features are not interesting, because we
consider only IP based network. In addition, client applications can \textit{explicitly bind} to server objects. Implicit binding helps to preserve location transparency by allowing clients to access remote objects. In addition, it helps conserve OS and networking resources, such as socket handlers and ATM virtual circuits, by (1) deferring network connections until they are actually used and (2) allowing multiple client threads in a process to be multiplexed through shared network connections to their corresponding servers. Unfortunately, implicit binding is inadequate for real-time applications with deterministic QoS requirements. In particular, deferring object/server activation and resource allocation until run-time can increase latency and jitter significantly. To avoid these problems, the RT-CORBA specification defines an \textit{explicit binding} mechanism. This mechanism enables clients to (1) pre-establish connections to servers and (2) control how client requests are sent over these connections.

Finally, RT-CORBA provides a means to coordinate access to resources in the CORBA system. The Mutex interface looks much like an operating system Mutex from the POSIX standard. There are methods to coordinate access to system resources, such as create_mutex, lock, unlock, and try_lock. The try_lock method includes a parameter to indicate a maximum wait time.

A thread calling lock on an unlocked Mutex will obtain the lock. All subsequent threads that request the locked Mutex will be blocked by queuing them in CORBA Priority order. When the thread holding the lock calls unlock, the highest priority blocked thread obtains the lock. Note that Mutexes are CORBA-wide entities allowing threads on different nodes to coordinate access to system resources.

This discussion highlights that RT-CORBA provides an interesting set of features to control scheduling and priority, but it does not provide high level primitives for constructing adaptive management of QoS.

3.5.2 TAO

Researchers at Washington University in St. Louis have developed TAO (The ACE ORB) [62]. TAO is a high-performance, RT CORBA compliant ORB that runs on a variety of operating system platforms with real-time features, such as Solaris. TAO's objective is to provide end-to-end Quality-of-Service guarantees at multiple levels in the distributed system. The system consists of four major parts that carry out this
objective: 1) the ORB, 2) the Scheduling Service, 3) the Event Service, and 4) the Real-Time I/O (RIO) subsystem.

TAO's ORB supports real-time by optimizing features such as memory management, network protocols, and code generation. TAO's ORB Core is based on the ACE framework which is a portable object-oriented middleware framework also developed at the Washington University. TAO uses ACE components to provide an efficient ORB Core that can be extended to adapt to new system environments and application requirements. TAO's ORB Core supports a range of transport protocols, including a Real-Time Inter-ORB Protocol (RIOP) [62] that extends GIOP/IIOP with QoS attributes. RIOP is a mapping of GIOP that allows applications to transfer their QoS parameters end-to-end from clients to servants. Such attributes include priority, execution period, and communication class. For optimality, TAO's mapping can selectively omit transport layer functionality and run directly on top of ATM virtual circuits. TAO provides a Real-time Object Adapter (ROA) that can be configured to implement custom mechanisms that dispatch client requests (to the ORB) according to application-specific real-time scheduling policies.

TAO provides a Scheduling Service that guarantees the hard real-time QoS specifications of client requests. The Scheduling Service supports both static scheduling, through off-line schedulability analysis, and dynamic scheduling, through policies such as admission control. There are two main components of the scheduling service: 1) the off-line schedulability analyzer; and 2) the run-time scheduler, which dispatches client requests through the ROA. A TAO client expresses real-time attributes through a structure for each of its schedulable operations and submits them to the scheduling service. The scheduling service examines the attributes of all registered operations, and performs schedulability analysis. If the system is found to be schedulable, the scheduling service assigns the static priorities to the operations. Once the priorities are assigned, the scheduling service determines the number and types of required dispatching queues, based on the number of required static priorities and the chosen scheduling strategy. At run-time, the ORB uses the run-time scheduler to retrieve the thread priority at which each queue dispatches operations and the type of dispatching prioritization used by each queue. Each queue is associated with a static priority. When an operation request arrives from a client, the scheduling service dispatches it to the appropriate queue. If a dynamic scheduling policy is used, the scheduling service also determines the dynamic subpriority at which the operation will
run within the already established static priority. TAO's scheduling service supports a variety of scheduling policies.

TAO's Event Service uses real-time scheduling of CORBA events, instead of typical First-Come-First-Served scheduling to further support best-effort real-time scheduling. Consumers and suppliers specify their execution requirements and characteristics using QoS parameters. These parameters are integrated with the system-wide scheduling policy to determine priorities and preemption strategies. TAO's event service provides filtering and correlation mechanisms that allow consumers to be more selective about which events they receive. Consumers are allowed to subscribe for a particular subset of events. The event service uses these subscriptions to filter supplier events, only forwarding them to interested consumers. Consumers can also specify AND and OR dependencies among the events that it will receive. For instance, a consumer can specify that it be notified only when all of the specified events have occurred. The event service also allows consumers to specify event dependency timeouts. A consumer can request to receive a timeout event if its dependencies are not met within some time period.

TAO's real-time I/O subsystem (RIO) runs in the OS kernel, and is designed to take advantages of the network ATM features.

This discussion highlight that TAO provides a very helpful set of services to control scheduling of the client requests (calls for objects toward the server), but like RT-CORBA, it does not provides high level primitives for supporting composition of QoS adaptive building blocks. However, the Event Service allows to support notification services, that can be helpful to implement event-based QoS policies.

3.5.3 QuO

Quality Objects middleware (QuO) is a framework for including QoS in distributed object applications [61]. QuO is designed to develop distributed applications that can specify (1) their QoS requirements, (2) the system elements that must be monitored and controlled to measure and provide QoS, and (3) the behavior for adapting to QoS variations that occur at run-time. Figure 3.15 illustrates a client-to-object logical method call. In a traditional CORBA application, a client makes a logical method call to a remote object. A local ORB proxy (i.e., a stub) marshals the argument data, which the local ORB then transmits across the network. The ORB on the server side receives
the message call, and a remote proxy (i.e., a skeleton) then unmarshals the data and delivers it to the remote servant. Upon method return, the process is reversed.

![Figure 3.15: CORBA method call.]

In contrast, a method call in the QuO framework is a superset of a traditional CORBA call, and includes the following components, illustrated in Figure 3.16:

- **Contracts** specify the level of service desired by a client, the level of service an object expects to provide, operating *regions* indicating possible measured QoS and representing a possible state of QoS, and actions to take when the level of QoS changes, callback for notifying the client or object during transition between regions.

- **Delegates** act as local proxies for remote objects. Each delegate provides an interface similar to that of the remote object stub, but adds locally adaptive behavior based upon the current state of QoS in the system, as measured by the contract.

- **System condition objects** provide interfaces to resources, mechanisms, objects, and ORBs in the system that need to be measured and controlled by QuO contracts.

In addition, QuO applications may use property managers and specialized ORBs. Property managers are responsible for managing a given QoS property (such as the
availability property via replication management [65] or controlled throughput via RSVP reservation management) for a set of QuO-enabled server objects on behalf of the QuO clients using those server objects. In some cases, the managed property requires mechanisms at lower levels in the protocol stack. To support this, QuO includes a gateway mechanism [66], which enables special purpose transport protocols and adaptation below the ORB.

In addition to traditional application developers (who develop the client and object implementations) and mechanism developers (who develop the ORBs, property managers, and other distributed resource control infrastructures), QuO applications involve another group of developers, namely QoS developers. QoS developers are responsible for defining QuO contracts, system condition objects, callback mechanisms, and object delegate behavior. To support the added role of QoS developer a QuO toolkit is provided that consists of the following components:

- **Quality Description Languages (QDL)** for describing the QoS aspects of QuO applications, such as QoS contracts (specified by the Contract Description Language, CDL) and the adaptive behavior of objects and delegates (specified by the Structure Description Language, SDL). CDL and SDL are described in [67, 68].
- The **QuO runtime kernel**, which coordinates evaluation of contracts and monitoring of system condition objects. The QuO kernel and its runtime architecture are described in detail in [69].
- **Code generators** that weave together QDL descriptions, the QuO kernel code, and client code to produce a single application program. Runtime integration of QDL specifications is discussed in [67].

QuO’s contracts and delegates support adaptation at many levels, from adaptation within an application, to adaptive resource control mechanisms, to adaptation at the transport layer. QuO’s contracts and delegates provide the adaptation that can be used within a single application. QuO’s system condition objects provide a uniform interface to system resources, mechanisms, and managers to translate between application-level concepts, such as operating modes, to resource and mechanism-level concepts, such as scheduling methods and real-time attributes. QuO provides a gateway component, which allows low-level communication mechanisms and special-purpose transport-level adaptation to be plugged into an application [66]. The QuO gateway resides between the client and server ORBs. It is a mediator that intercepts IIOP messages sent
from the client-side ORB and delivers IIOP messages to the server-side ORB (on the message return the roles are reversed). On the way, the gateway translates the IIOP messages into a custom transport protocol, such as group multicast in a replicated, dependable system. The gateway also provides an API that allows adaptive behavior or processing control to be configured below the ORB layer. For example, the gateway can select between alternate transport mechanisms based on low-level message filtering or shaping, as well as the overall system's state and condition objects.

QuO contracts and delegates support two means for triggering manager-level, middleware-level, and application-level adaptation. The delegate triggers in-band adaptation by making choices upon method calls and returns. The contract triggers out-of-band adaptation when changes in observed system condition objects cause region transitions. The QuO delegate supports in-band adaptation (Figure 3.17) whenever a client makes a method call and whenever a called method returns. When a client calls a remote method, the call is passed to the object's local delegate instead. This is transparent to the client, since the remote object and the delegate have the same interface. The delegate can trigger contract evaluation, which grabs the current value of all system conditions measuring aspects of the system's state. The contract consists of a set of nested regions which describe the relevant possible states of QoS in the system. Each of these regions is defined by a predicate on the values of system condition objects. The contract evaluates the predicates to determine which regions are active (i.e., their predicates are true) and passes the list to the delegate. The delegate chooses how to process the method call based upon the current regions. The delegate performs similar processing upon a method return, i.e., it evaluates the contract to obtain the current QoS regions and selects a behavior based upon the current regions.

![Figure 3.17: QuO’s in-band adaptation (upon methods call and returns).](image-url)
QuO contracts and system condition objects support out-of-band adaptation (Figure 3.18) by monitoring conditions in the system, whether they are the states of resources, mechanisms, or managers. Whenever the monitored conditions change (or whenever they change beyond a specified threshold), the system condition object triggers an asynchronous evaluation of the relevant contracts. If this results in a change in contract region (i.e., state), it in turn triggers adaptive behavior that occurs asynchronous to any object interactions.

![Figure 3.18: QuO's out-of-band adaptation (adaptation on region transitions).](image)

QuO appears as a very complete and helpful middleware. From our point of view, the most important features of QuO are the QoS contracts, delegates, and the Quality Description Language (QDL) that constituted a very flexible and reusable interface to manage QoS of building blocks.

### 3.5.4 Other Architectures

Other Distributed Object Computing middleware has emerged, such as Agilos [63] and the real-time specification for Java (RTSJ) [70].

The Agilos (Agile QoS) architecture is a middleware control architecture designed to provide middleware services to assist application-aware adaptations, namely, adaptation mechanisms that are tuned to the performance goals and specific functions of an application. In order to accomplish this objective, Agilos is designed as a three-tier architecture, as depicted in figure 3.19.

In the first and lowest tier, application-neutral adaptors and observers maintain tight relationships with individual types of resources, and react to changes in resource availability. In the second tier, application-specific configurators are responsible for making decisions about when and what adaptive mechanisms should be invoked in a client-server application, based on on-the-fly user preferences and application-specific
rules. Furthermore, though each configurator corresponds to one application, configurators share the same fuzzy inference engine for rule processing. Finally, QualProbes provide QoS probing and profiling services so that application-specific adaptation rules can be either obtained from measurements or specified explicitly by the user. In the third tier, a gateway and negotiators are introduced to control adaptation behavior in an application with multiple clients and servers, so that dynamic reconfigurations of client-server mappings are possible and tuned to the best interests of the application.

---

**Figure 3.19:** The Hierarchical Design of the Agilos Architecture

Agilos appears to be very suitable for supporting adaptation mechanisms, but it considers an application as a single entity, composed of a set of CORBA objects, and does not provide facilities for constructing applications by composing QoS-aware building blocks.

Finally, Java's real-time approach falls outside our interest because it relies on a specific programming language.
Chapter 4

End-to-End QoS at the Middleware

The main aim of this thesis is to investigate issues of design of adaptive QoS policies, located at middleware level, that can support effectively real-time applications and services geographically distributed over best-effort IP-based network. In this context, the notion of QoS needs to be extended to include application-specific requirements such as availability and timeliness. We term this extended notion of QoS “End-to-End QoS”. In this chapter we discuss the notion of End-to-End QoS and motivate our design approach to the provision of QoS adaptive middleware services. Moreover, in order to assess the ability of the middleware approach to support effective QoS policies, two case studies have been evaluated, and two particular QoS policies has been proposed, implemented and evaluated.

This chapter is structured as follows: the first subsection 4.1 describes the characteristics of the adaptation model we have adopted. Section 4.2 introduce the monitoring and adjustment services. Section 4.3 discusses the concept of End to End QoS and the problem of QoS parameterization. Section 4.4 describes the motivations for providing support for QoS management at the middleware level. Finally, Section 4.5 describes the criteria followed to validate the effectiveness of middleware we have developed, to support QoS policies, and describes the choice of case studies.

4.1 Best Effort and Adaptive

In contrast with the traditional but actual best-effort service provided by the widespread IP-based networks, the modern architectures devoted to providing QoS at the application level assume the existence of a network subsystem (RSVP, DiffServ) capable of ensuring some kind of network QoS guarantees. In this context, QoS is thought of and specified in low-level terms, such as raw bandwidth and latency for network resources, or CPU and memory utilization for system resources. Hence, QoS
Brokers can be developed so as to manage both network and host resources, in order to achieve a balance between QoS application requirements, Operating System (OS) resources and network resources and, in particular, to negotiate the use of these resources with remote brokers. The model for providing applications with QoS can be thought of as consisting of the following four steps:

- QoS application requirement assessment;
- mapping assessment results onto QoS parameters for underlying layers;
- negotiation between system components to ensure that resource capacity can satisfy QoS requirements,
- reservation of resources.

This model can be deployed over network infrastructures that provide guaranteed communication services (e.g. ATM). However, application are implemented for using best effort network services (rather than guaranteed services) such as those provided by the Internet. In a best-effort network resources availability may vary arbitrarily; hence, soft real-time applications must to be designed so as to adapt dynamically those variations.

### 4.2 Monitoring and Adjustment

In order to dynamically adjust the applications to the availability of the resources, we need to develop a service that keeps information about these resources and modifies the behavior of the applications accordingly. This service can be split in two different services; namely, a monitoring service and an adjustment service. The former can be implemented by the middleware, the latter can be implemented by cooperation between middleware and the application.

The monitoring service, given some information about the application QoS requirements, evaluates the availability of the necessary system resources, and informs the adjustment service. Typically the monitoring service is executed periodically, while passing the information to the adjustment service may be omitted if there are no modifications to the performance of the resources. Instead, the adjustment service, typically implemented by the applications as well as the middleware, is executed in the event-driven mode and adapts the applications to changes in the environment, according to the information provided by the monitoring service.
Based on the adaptive approach, the model for providing the applications with QoS can be modified as follows:

- Definition of QoS application requirements, by means of high-level characteristics (for instance, user response time, availability).
- Parameterization of QoS characteristics. The QoS application requirements are to be characterized as a set of parameters that can be used as an objective QoS description, in order to i) inform the underlying architecture of QoS requirements, and ii) define a metric to evaluate the status of the QoS of that application (e.g. for a Web service, the response time provided to a given user).
- Definition of policies and high-level software architectures, tailored to the specific application, in order to provide the application with the desired QoS. For instance, for a highly available Web service, it is possible to implement server replication and parallel retrieval (i.e. the high level software architecture) and a policy for selecting the best replica server.
- The negotiation phase can be substituted by a periodical and/or event-driven phase of adaptive adjustment, in which a) the availability and performance of the resources involved in the service are observed by the monitoring service, and b) the behavior of the applications is modified by the adjustment service so as to meet their requirements, according to the policies adopted.

In section 4.4, the motivations for implementing the monitoring and adjustment services at the middleware level are explained in detail.

4.3 Extending the Notion of End-to-End QoS

In order to provide support for QoS over best effort networks, it is important to reconsider and extend the concept of QoS for applications, in particular highlighting characteristics such as end-to-end QoS as perceived by the users.

This thesis argues that the quality requirements of geographically distributed real-time applications/services effectively depend on a set of specific application requirements that include:

- scalability
- interoperability
- fault tolerance
• availability
• timeliness
• real-time constraints (latency, fidelity, etc.)
• other requirements.

**Scalability** is a primary factor that can influence the design and implementation of a distributed system. In particular, mechanisms that work adequately in a small distributed system may fail to do so when deployed within the context of larger systems. Hence, we define as "scalable" a system that can provide its services according to the performance and reliability specifications of those services, regardless of both the number of resources it accommodates and the geographical separation between these resources. Thus, for example, if the users of a Video-Conference application can tolerate at most a 300-ms end-to-end latency of the audio channel, the algorithms that support this application should guarantee that latency is never exceeded, however far apart the application users are geographically and regardless of the number of people using the application concurrently.

**Interoperability** is the ability to hide heterogeneous networks and systems, in order to allow the applications to work over different architectures, independently from specific hardware platforms, operating systems, programming languages and network protocols. As described in chapter 3, the major threads to build computer system that must operate in geographical, open, distributed, highly heterogeneous environments, are: a) the introduction of intermediate software platforms, the middleware [13], and b) the notion of transportable computation [48]. We are particularly interested in the middleware approach as not constrained to a given programming language.

A Fault [71] is an unacceptable degree of variation from the desired behavior of an application; behavior includes the value of the output, timeliness of the output, generation and manifestation of other effects, security of the output. Thus, a service may be described as **fault tolerant** if it is designed to function correctly in the presence of specified faults in other services on which it depends. For example, a Web-based service experiences a fault if the server that provides the service is down or if there is an internet routing problem that prevents the browser from contacting him. A good policy for providing fault-tolerant Web service is based on introducing redundancy by replicating Web servers at distinct sites.
Availability of a service can be defined as the percentage of time that the service is available for the users. This is a very important metric of End-to-End QoS. Users expect services to be continuously available. A service that is frequently unavailable may have the effect of tarnishing the reputation of the service provider or result in a loss of opportunity. The percentage of requests that are satisfied by the service represents an experimental measurement of service availability, as perceived by the users.

Timeliness is the ability to provide the required service to the user quickly. From the user's perspective, a service that exhibits no timeliness is virtually equivalent to an unavailable service. Availability and timeliness constitute the service "responsiveness", that the users perceive as one of the primary End-to-End QoS attributes.

Other quality requirements of geographically distributed real-time applications are strictly application-dependent, in particular, they depend on the application's real-time constraints. There are, for example, two parameters (instances of latency and signal fidelity) that basically describe the quality of packet-based conversation over the Internet: the end-to-end delay between packet audio generation at the sending site and its playout time at the receiver (namely playout delay), and the percentage of audio packet loss. Typically acceptable values of the playout delay are below 200-250 msec. In addition, a percentage of no more than 5 - 10% of packet loss is considered quite tolerable in human conversations [73].

4.3.1 QoS Parameterization

Since it is the quality perceived by the end-user that will determine whether a service or application is a success, it is vital to carry out assessment of the quality delivered through these. In order to classify the distributed applications from the point of view of the perceived QoS, it is absolutely necessary to map the application's QoS requirements into measurable quantities (i.e. the numerical parameters) that describe them. Moreover, a parametric description is indispensable for mapping QoS application requirements onto the QoS parameters for underlying layers.

As described above, most real-time requirements can be defined by means of measurements concerning bandwidth requirements, transmission delay and jitter delay. Instead, for most QoS requirements, parameters that describe them only exist at the
application level: availability is quantified by the percentage of user requests that are being satisfied; timeliness can be specified by mean time to provide the service.

Other QoS requirements, scalability for example, are more complex and harder to describe. Finally, for some multimedia applications, the QoS is strongly subjective, and its measurement is currently being investigated. All these requirements, at the application level, contribute to creating QoS (of a specified service) as perceived by the users.

4.4 Rationale for QoS Management at the Middleware

The discussion about QoS parameterization proposed in the previous subsection 4.2.2, introduces the first rationale for providing QoS at the middleware. The most important reason for introducing a software level between the operating system and the application level is the impossibility of providing application-defined QoS only using low-level services. The notion of QoS has been extended from the communication level up to the application level. In fact, the vast majority of distributed applications/services present peculiar characteristics and their QoS, as perceived by the users, can be represented by means of a wide set of application level parameters, strictly depending on the applications themselves, but not always expressed only using low-level (communication-level) QoS parameters.

For instance, owning to the possibility of network partitions between a web server and its clients, that Web service cannot be made highly available using network resource reservation only. Rather, high availability can be achieved by geographically replicating the Web servers that implement that service, and by providing the user with a mechanism for binding the best replica server. This binding mechanism can be described and controlled above the network and transport levels. Another example of application-level Qos control is provided by SOT [74], an architecture devoted to multimedia conferencing, in which the control of network congestion depends on both low-level parameters, such as packet loss, and very high-level parameters, such as description of the distribution of the applications over the network.

The above observations indicates that the QoS of several applications cannot be provided only at network and operating system levels, but must be managed at higher levels, by strict cooperation between the application level and “ad hoc” policies tailored to the application.
The monitoring of resources is an activity that requires strict interaction between the operating system (OS) and the network level. In fact, many applications need information (CPU workload, accesses to disks, cache hit ratio, memory usage, packets sent/received) concerning the subsystem managed by the OS, but the procedures to obtain this value are strictly dependent on the operating systems. A middleware layer is particularly suitable for providing transparent access to that information.

Finally, a middleware is particularly suitable for collecting statistical information from different entities (local or remote), implementing schedule policies, providing process synchronization and notification services, particularly event-driven services.

All the motivations presented suggest introducing an intermediate software level between the operating system and the application level, in which to concentrate the management of QoS, in particular monitoring resources and application behavior adjustment. The applications should be responsible only for providing this middleware with high level information (an application profile) to enable and activate middleware components, and map QoS application requirements onto middleware component parameters.

In order to obtain a more complete interaction with the application, the middleware may provide some virtual entry points to which the application may link its own function implementation. This mechanism (“call back” function) allows the application to modify the behavior of middleware components.

### 4.5 The Case Studies

In order to assess the effectiveness of our middleware approach to support End-to-End QoS we have examined two case studies, and proposed, implemented and evaluated two specific policies.

Our approach in general, requires that the following three principal issues be dealt with:

1) Identification of distributed services that can take advantage of QoS management and an analysis of QoS constraints. Most existing services (Internet Telephony or Remote Control of Industrial Processes, for example) are strictly QoS dependent; however each distributed application (or service) that interacts with the user can be revised in order to provide support for QoS requirements. Web-based services form a very large application class that can take advantage of QoS management. In
addition, new applications (e.g. music on-demand, accessible by means of a UMTS mobile terminal) present characteristics that can benefit from a QoS-driven design.

2) Definition and parameterization of QoS characteristics. It is crucial that the QoS properties of a specific service be accurately specified, in order to define a set of parameters that can be used as QoS description. From these parameters we need to define a metric to evaluate the validity of adopting specific QoS management solutions. Applications may also use these parameters to inform the underlying architectures of the QoS status.

3) The proposal and evaluation of system architectures and algorithms. This fundamental step consists of: i) evaluating existing system architectures that provide support for QoS-dependent distributed applications; ii) proposing new architectural models and algorithms for this purpose; iii) the implementation, development, experimentation, evaluation and comparison of these architectural models and algorithms, in order to discover their properties, and interaction with the standard platform on which they are based. Performance will be evaluated by means of metrics based on the QoS parameterization previously defined.

In order to evaluate experimentally our QoS-based policies at middleware level, we have so far examined two kinds of service that represent, from the point of view of QoS requirements, very different application classes. In particular, we have examined:

- Unicast voice-based audio communications over the Internet strongly depends of real-time constraints, and
- A service of load distributions among replicated web servers [22] which does not exhibit strong real-time constraints, but should satisfy properties such as reliability and responsiveness. Moreover, this service requires interoperability, meaning that it should be used independently from the choice of browser.

The analysis of these two case studies has been completed, and we have proposed two new QoS policies to address their QoS requirements. The policies we have designed have been implemented and the experimental evaluations have confirmed the effectiveness of the approaches we propose. Our results indicate that, for most classes of applications, it is possible and helpful to implement QoS management at the middleware and application levels.
Chapter 5

The Replicated Web Service Case Study

The first case study concerns a Web service. That service does not depend on strong real-time constraints, but should satisfy properties such as availability and responsiveness.

The success of a Web service is largely dependent on its responsiveness (i.e., its availability and timeliness) in the delivery of the information its users (clients) require. A practical approach to the provision of responsive Web services, recently proposed in the literature [25], is based on introducing redundancy in service implementation, namely by replicating the service across a number of servers geographically distributed over the Internet. Provided that the replica servers be maintained mutually consistent, service responsiveness can be guaranteed by dynamically binding the client to the most convenient replica (e.g., the nearest, lightly loaded, available replica; the available replica with the least congested connection with the client). Based on this approach, we have developed a software mechanism that effectively meets the responsiveness requirement mentioned above. In essence, this mechanism, rather than binding a client to its most convenient replica server, engages all the available replicas in supplying the fragment of the Web document that the client requires. The size of fragment that a replica is requested to supply is dynamically evaluated on the basis of the response time that replica can provide its client with. In addition, the proposed mechanism can dynamically adapt to changes in both the network and the replica server status, thus tolerating possible replica or communication failures that may occur at run-time. Our mechanism can be implemented either as part of the browser software, or as part of a Proxy server. In this chapter, we describe the design, development, and performance evaluation of both these implementations of our mechanism. The performance results obtained from our evaluation exercise illustrate the adequacy of the mechanism we propose, in order to provide responsive Web services.
5.1 Introduction

In this chapter, we address issues of design of responsive (i.e., both highly available and timely) Web services. These are services that can be provided through the use of the HTTP protocol, and its associated client-server architecture [75]. Responsiveness is a crucial issue in the design of those services, as, from the service user perspective, a poorly responsive Web service can be virtually equivalent to an unavailable service.

Because of the complexity of the Web infrastructure, many components could affect the quality of Web services: from network technology and protocols, to hardware and software architectures of Web servers and proxies. As most components of the Web infrastructure are beyond the control of Web system administrators, quality of Web services is very hard to achieve. Web service providers cannot guarantee the services because their actions are limited to a small part of the Web infrastructure. We consider solutions for Web service providers that can act only on their Web systems.

To augment satisfaction percentage of the assessed service levels, they can rely on two classes of actions that are not mutually exclusive: Differentiated Web services and Architecture design. The Differentiated Web services requires the definition of classes of users/services, choice of the number of priority levels, guarantee of different service level agreements through priority dispatching disciplines [76, 77, 78] and monitors for starvation of low priority services. Instead, the goal of the Architecture Design is to find the right architecture that guarantees the service level agreement on all Web users/services. It considers three directions: scale-up by adding memory and CPU power to the single server, local scale-out by replicating servers in a local area, global scale-out by replicating servers in a geographical context.

5.1.1 Differentiated Web Services

Most proposals for guaranteeing quality of Web services look at new Web server architectures that can support differentiated scheduling services to enable preferential treatment of classes of users and services. The main motivation is that first-come-first-served service policies implemented by traditional Web servers can undermine any improvements made by network differentiated service [76]. Since overloaded servers affect all requests in the same manner, a FCFS discipline makes impossible to guarantee a service to preferred clients. To overcome this drawback, priority based
scheduling schemes can be implemented in the Web server to provide differentiated services. The main components of a Web server architecture that provide differentiated service must include a classification mechanism to assign different priority classes to incoming requests, an admission control policy to decide how and when to reject requests according to their priorities, a request dispatching policy that decides the order in which requests should be serviced, and a resource dispatching policy to assign server resources to different classes of priority [76]. Most proposed architectures modify Web servers at application or kernel level to allow differentiated control through dispatching of requests and resources. Quality of service by using priority levels to determine admission priority and performance level. The method used to dynamically classify requests on a per-session basis includes source IP address, TCP port number, and the requested content. Similar Web server prototypes that support differentiated services have been proposed in [79, 78]. To enforce service level agreement constraints, Pandey et al. [77] examine selective allocation of server resources through the assignment of different priorities to page requests. Menasce et al. [80] analyze and compare policies that dynamically assign priorities to customers of a commercial Web site by differentiating between visitors and potential buyers.

5.1.2 Locally Distributed Replica Servers

Most of the previous results consider Web sites consisting of a single server node. The only solutions for scaling server capacity in the past has been to completely replace the old server with a new one. Organizations must discard their investment in the old server and purchase a new one. An expansive short-term solution. A long-term solution requires incremental scalability, which provides the ability to grow gradually with demand. On the other hand, we claim that popular Web sites cannot rely on a single powerful server to support service level agreement for each increasing request load. Scalability, load balancing, and dependability can be only provided by Web server architectures, constructed out of replicated servers, that distribute intelligently client requests across multiple server nodes. A pool of servers tied together to act as a single unit, or server clustering, provides such incremental scalability. Service providers may gradually add additional low-cost computers to augment the performance of existing servers. The main components of a typical multi-node Web system include a dispatching mechanism to route the client request to the target Web server node, a
dispatching algorithm to select the Web server node best suited to respond, and an executor to carry out the dispatching algorithms and support the relative mechanism. The decision on client request assignment can be taken at various network levels. A large set of techniques, e.g. [81, 82, 83, 84, 85, 86, 87, 88, 89, 90], has been proposed in the literature that addresses specifically the issue of providing highly available Web services by relying on replicated servers, locally distributed in a cluster of workstations, and by distributing the client request load among those servers. Thus, service availability is achieved through the redundancy inherent in the service implementation; in addition, the overall service throughput (i.e., the number of client requests per second that can be served) is optimized through careful distribution of the client request load among the clustered servers. In the following paragraph, we propose a classification of existing approaches based on the type of distribution of the server nodes that compose the scalable architecture that is, local distribution and geographical distribution. All Web server clustering technologies are transparent to client browser (i.e., the client browser are unaware of the existence of the server cluster). However, not all clustering technologies are transparent to the Web server software. Several commercial cluster-based Web servers, such as Inktomi [91], are not transparent to the server nodes and require specialized software throughout the system. The adoption of such proprietary systems cannot provide the flexibility and low cost service providers have come to expect with the wide array of Web servers and server extensions available. For this reason, we emphasize the solutions that allow service providers to utilize commodity hardware and software. This implies that the clustering technique must be transparent to both the Web client and the Web servers since the majority of Web servers do not have any built-in clustering capabilities.

In each of the network cluster technologies discussed in this section, one entity, called the dispatcher, sits on the network and acts as a proxy for incoming connections. The dispatcher is configured with a particular network address, called the cluster address. The servers appear as a single host to clients because of the dispatcher (client-side transparency). The dispatcher receives service request from clients and selects a server from the server pool to process the request. Depending on the clustering technology, the dispatcher appears as either a switch (processing incoming data only) or a network gateway (processing incoming and outgoing data) to the servers in the pool. In either case, we assume each server is executing standard web server software designed for a standalone server (server-side transparency). Incoming client requests
are distributed more or less evenly to the pool of servers. This is made possible by protocols such as HTTP which typically have small request and save no state information on the server. In the followings, we present different clustering techniques that operate at different OSI layer, layer two (data link layer), layer three (network layer) or layer seven (application layer).

**DataLink-Layer Clustering**

In DataLink-layer clustering, the cluster network-layer address is shared by the dispatcher and all of the servers in the pool through the use of primary and secondary IP addresses. That is, while the primary address of the dispatcher is the same as the cluster address, each server is configured with the cluster address as a secondary address by use of *interface aliasing*. The nearest gateway is then configured such that all packets arriving for the cluster address are addressed to the dispatcher at layer two (by mean of a static Address Resolution Protocol (ARP) cache entry). If the packet received by the dispatcher corresponds to a TCP/IP connection initiation, the dispatcher selects one of the servers in the server pool to service the request. The dispatcher then makes an entry in a connection map, noting the origin of the connection and the chosen server. The layer two destination address is then rewritten to the hardware address of the chosen server and the frame is placed back on the network. Instead, if the incoming packet is not for connection initiation, the dispatcher examines its connection map to determine if it belongs to a currently established connection. If it does, it rewrites the layer two destination address to be the address of the server previously selected and forwards the packet as before. The selected server accepts the packet and replies directly to the client rather than through the dispatcher. Thus, the dispatcher processes only the incoming data stream. Moreover, the dispatcher does not modify the layer three packet, thus it does not need to recomputes the IP checksum. Among implementations of DataLink-Layer clustering are ONE-IP, eNetwork Dispatcher, LSMAC and ACEdirector.

- **ONE-IP** is one of the first implementation of data-link-layer clustering and has been developed at Bell Laboratories (circa 1996) [92]. It supports two different dispatching methods. With the first methods, when a packet is received by the dispatcher, the client’s address is hashed to obtain a value indicating which server in the server pool will service the request. The second dispatching methods broadcasts packets destined for the cluster on the LAN that connects the dispatcher with the pool.
of servers. Each server in the pool implements a filter on the client address such that a server only responds to a fixed and disjoint portion of the address space. Neither of these algorithms is able to adapt to conditions when clients disproportionately load the server. ONE-IP supports fault tolerance for both the dispatcher and the servers through the use of a watchdog daemon. When a server fails, the dispatcher does one of two things. If it is using the first dispatching methods, it modifies the hash table to take into accounts the reduced server pool. Instead, if the dispatcher is using the second (broadcast-based) methods, it informs the entire server pool of the failed server. Each server then changes its filter accordingly. In the event of dispatcher failure, a backup dispatcher will notice the missing dispatcher heartbeat messages and take over. Since there is no state information, none needs be replicated or rebuilt, and the failover is simple and fast.

- **eNetwork Dispatcher** has been developed by IBM [93] and adopted for the 1998 Olympic Games Web site. The eNetwork Dispatcher runs on a single node and uses a weighted round robin algorithm to distribute connections to the servers in the server pool. Periodically, it recomputes the weights based on load metrics collected from the servers. The eNetwork Dispatcher supports fault detection and masking for both the dispatcher and the servers in the pool. Additionally, the dispatcher may have a hot spare that functions as a backup dispatcher. The primary dispatcher runs a cache consistency protocols with the backup. In the event that the backup no longer receives heartbeat messages from the primary, it takes over. Moreover, through a mechanism called *client affinity*, the eNetwork Dispatcher is able to support services such as FTP and, in particular, SSL. With client affinity, multiple connections from the same client within a given period are directed to the same server. This allows servers and clients to share state information such as SSL session keys during the timeout period.

- **LSMAC** [94] has been developed by the University of Nebraska-Lincoln and implements a DataLink-Layer clustering as a portable user-space application running on commodity systems. Utilizing *libpcap* [95] and *libnet* [96], LSMAC achieves performance comparable to the eNetwork Dispatcher. Like other dispatchers, LSMAC provides fault detection and masking for the server pool. Periodically, the dispatcher sends ARP queries to determine which servers are currently active, thus allowing for automatic detection of dynamically added or removed systems. In addition, it watches
for TCP reset messages corresponding to the service being clustered and removes the nonperforming system from the pool.

- **ACEdirector** has been developed by Alteon and is implemented as an Ethernet switch (both layers two and three) based on a 2.5 Gb/s switch fabric, added by the ability to operate as a DataLink-Layer clustering. Alteon has been the first to offer in-switch clustering at layer two. ACEdirector provides round-robin and least-connections load sharing policies, and allows for some stateful services such as SSL. Moreover, it provides fault detection and masking for the server pool and hot-standby with another ACEdirector switch.

**Network-Layer Clustering**

The Network-layer clustering provides reasonable performance while simultaneously providing the flexibility providers expected by leveraging commodity products. Unlike DataLink-Layer clusters, each constituent server is configured with a unique IP address in Network-Layer clusters. A Network-Layer dispatcher appears as a single host to a client. To the machines in the server pool, however, a Network-Layer dispatcher appears as a gateway. When traffic is sent from the client to the clustered Web server, it is addressed to the cluster address. Utilizing normal network routing rules, this traffic is delivered to the cluster dispatcher. If a packet received corresponds to a TCP/IP connection initiation, the dispatcher selects one of the servers in the server pool to service the request. Similar to that in DataLink-Layer clustering, server selection is based on some load sharing algorithm, which may be as simple as round-robin. The dispatcher also then makes an entry in a connection map, noting the origin of the connection, the chosen server, and other information (e.g., time) that may be relevant. However, unlike in the earlier approach, the destination (IP) address of the packet is then rewritten as the address of the server selected to service this request. Moreover, any integrity codes affected (such as packet checksums, cyclic redundancy checks (CRCs), or error correction checks (ECCs)) are recomputed. The modified packet is then sent to the server corresponding to the new destination address of the packet. If incoming client traffic is not a connection initiation, the dispatcher examines its connection map to determine if it belongs to a currently established connection. If it does, the dispatcher rewrites the destination address as the server previously selected, recomputes the checksums, and forwards as before. In the event that the packet does not correspond to an established connection but is not a connection initiation packet
itself, the packet is dropped. Traffic sent from the servers in the server pool to clients must also travel through the dispatcher since the source address on the response packets is the address of the particular server that serviced the request, not the cluster address. The dispatcher rewrites the source address to the cluster address, recomputes the integrity codes, and forwards the packet to the client. It is obvious that Network-Layer clustering will always outperform Network-Layer clustering due to the overhead imposed by Network-Layer clustering (the necessary integrity code recalculation coupled with the fact that all traffic must flow through the dispatcher). Even if hardware support is provided for integrity code recalculation (as with Gigabit Ethernet), a Network-Layer dispatcher must process much more traffic than a DataLink-Layer dispatcher. Thus, total data throughput of the dispatcher limits the scalability of the system more than the sustainable request rate. The basic Network-Layer clustering approach is detailed in RFC 2391, ”Load Sharing Using Network Address Translation (LSNAT)” [97]. Magicrouter from Berkeley was an early implementation of this concept based on kernel modifications [98]. Cisco’s LocalDirector product is a proprietary commercial implementation, while LSNAT from the University of Nebraska-Lincoln provides an example of a non-kernel space implementation [94].

- **Magicrouter** has been developed at the University of California at Berkeley and provides an early implementation of Network-Layer clustering [98]. Using a kernel modification called ”fast packet interposing”, Magicrouter provides load sharing and fault tolerance. Magicrouter offers three load sharing algorithms: round-robin and random that uses round-robin and random connection dispatching policies respectively, and incremental load that adopts a per-server load estimate plus an additional adjustment based on the number of connections active at the server in question. During connection initiation, Magicrouter selects the least loaded server. To provide fault detection, Magicrouter utilizes ARP as well as TCP reset detection.

- **LocalDirector** [82] developed by Cisco, offers four load sharing policies: i) Least Connections: it selects the server with the fewest currently established connections; ii) Weighted Percentage: this policy is similar to the least connection policy but with the addition that weights may be assigned to each of the servers in the server pool. This allows the user to manually tune the dispatching policy to take into account varying server capacities; iii) Fastest Response: This policy attempts to dispatch the connection
to the server that responds to the connection request first; d) Round-Robin: This is a strictly round-robin policy. LocalDirector also provides failure detection and reconfiguration with regard to the server pool. In the event that a server stops responding to requests, LocalDirector removes it from its list of active servers and marks it as being in a testing phase. It then periodically attempts to contact the server. As soon as it is capable of contacting the server, it is brought back into active duty. Through the use of its sticky flag, LocalDirector can be made to support some stateful services, such as SSL [99] (IBM’s client affinity). When the sticky flag is set, multiple connections from the same client within a given period (five minutes by default) are directed to the same server. This allows servers and clients to share state information, such as SSL session keys, during the timeout period.

• LSNAT [94] from the University of Nebraska-Lincoln is a user-space implementation of the key points of RFC 2391, “Load Sharing Using Network Address Translation” [97]. LSNAT runs on standard hardware under the Linux operating system or any other modern UNIX system supporting libpcap [95] and POSIX threads. Operating entirely in user space, LSNAT achieves a throughput of 30 Mb/s [94]. While this is generally lower than LocalDirector, it may have more to do with poor packet capture performance on the test platform rather than the choice of a user-space or kernel-space implementation. LSNAT also provides failure detection and reconfigures itself accordingly. In the event of dispatcher failure, unlike LocalDirector, LSNAT does not fail over to a dedicated hot spare. Rather, one of the servers in the server pool detects its failure and reconfigures itself as the dispatcher. It then uses a distributed state reconstruction mechanism to rebuild the map of existing connections. If one of the servers fails, LSNAT detects this and removes it from its list of active servers. Upon restarting, the server announces its presence and is placed back in the active server pool. This functionality is achieved with the aid of a small daemon.

**Application-Layer Clustering**

While strictly DataLink-Layer or Network-Layer clustering may be considered solved problems, a great deal of research is currently ongoing in the area of Application-Layer clustering. These approaches use information contained in OSI layer seven (application layer), typically to augment DataLink-Layer or Network-Layer dispatching. This is also known as content-based dispatching since it operates based on the contents of the client request. We examine LARD from Rice University [100], a Web Accelerator from IBM
T. J. Watson Research Center [101], and Web Switches, a commercial hardware product from ArrowPoint Communications.

- **LARD.** Researchers at Rice University have developed a Locality-Aware Request Distribution (LARD) dispatcher for a pool of Web servers. Since servers are selected based on the content of the protocol request, we classify LARD as an Application-Layer dispatcher. LARD partitions a Web document tree into disjoint sub-trees. Each server in the pool is then allocated one of these sub-trees to serve. In this way, LARD provides content-based dispatching as requests are received. For instance, if the pool is constituted by two server, the first server may be capable of handling requests of type A, while the second can handle requests of types B and C. In that context, the dispatcher decomposes the stream of requests into a stream of requests for the first server and one for the second server, based on the content of the requests (i.e., type A, B or C). As requests arrive from clients for the clustered Web server, the LARD dispatcher accepts the connection as well as the request itself. The dispatcher then classifies the requested document and dispatches the request to the appropriate server. The dispatching is done with the aid of a modified kernel that supports a connection handoff protocol: after the connection has been established, the request known, and the server chosen, the LARD dispatcher informs the chosen back-end server of the status of the network connection, and the backend server takes over that connection (communicating directly with the client). In this way, LARD allows each server’s file system cache to cache a separate part of the Web tree rather than having to cache the entire tree, as "ordinary" DataLink-Layer and Network-Layer clustering require. Additionally, it is possible to have specialized server nodes. For example, dynamically generated content could be offloaded to special compute servers while other requests are dispatched to servers with less processing power. While LARD requires a non-commodity operating system on the servers (they must be able to support the TCP handoff protocol), it does allow service providers to choose from commodity Web servers.

- **Web Accelerator,** developed at IBM T. J. Watson Research Center, combines content-based dispatching, based on *layer seven and four switching with layer two packet forwarding*, with Web page caching [101]. However, page caching comes at the cost of reduced parallelism in the cluster. When a client attempts to connect to the clustered Web server, the Accelerator accepts the connection and the client request. If
possible, this request will be served out of an in-memory cache on the dispatcher. In the event that there is a cache miss, the dispatcher contacts a server node and performs the same request as the client. It then caches this response and issues the response back to the client. The performance of Web Accelerator decreases rapidly as response size increases [101]. This is due to the fact that unlike LARD, all outgoing traffic is issued from the Accelerator. Thus, service providers cannot fully exploit the latent parallelism in the cluster since all responses must now travel through the dispatcher. Note that the traffic flow through the system looks similar to Network-Layer clustering.

- **Web Switches**, developed by ArrowPoint, is one of the first hardware devices to incorporate content-based routing. ArrowPoint’s Web switches employ a caching mechanism similar to IBM’s Web Accelerator. ArrowPoint’s Web switches also provide sticky connections in order to support some stateful services. Moreover, one of that switches supports a hot standby unit and fault masking on the server nodes. Cabletron, Intel, and others provide similar products.

For DataLink-Layer dispatchers, system performance is constrained by the ability of the dispatcher to set up, look up, and tear down entries. Thus, the most telling performance metric is the sustainable request rate. Network-Layer dispatchers are more immediately limited by their ability to rewrite and recalculate the checksums for the massive numbers of packets they must process. Thus, in the absence of dedicated check-summing hardware, the most telling performance metric is the throughput of the dispatcher. Finally, Application-Layer solutions are limited by the complexity of their content-based routing algorithm and the size of their cache (for those that support caching). However, as pointed out in [102], by localizing the request space each server must service and caching the results, Application-Layer dispatching should provide higher performance for a given number of back-end servers than DataLink-Layer or Network-Layer dispatching alone. It seems clear that in the future, L7 hardware solutions will continue to dominate software products in terms of performance. Finally, as pointed out in [102], in boosting server performance to the levels supported by Application-Layer hardware solutions (e.g., ArrowPoint switches), the bottleneck is no longer the ability of the server to generate data, but rather the ability of the network to get that data from the server to the client. New research on scalable Web servers must take into account wide area network bandwidth as well as server performance.
5.1.3 Geographically Distributed Replica Servers

As discussed in [25], the techniques based on local clustering and previously described can only partially meet the high availability requirement mentioned above (e.g., they may be vulnerable to failures of the router/gateway that interfaces the service cluster to the rest of the network). In addition, they cannot be deployed in order to meet such timeliness requirements as the client latency time over the network (that may notably affect the timeliness of a Web service as perceived by its users) as these requirements fall beyond the control of these techniques.

In order to overcome these limitations, an alternative approach has been proposed in [25], and explored further in [23, 24]. According to this approach, a responsive Web service can be provided by replicating servers across the Internet, rather than in a cluster of workstations. In other words, in order to reduce network impact on users’ response time and to scale to large traffic volumes, a better solution is to distribute Web servers over the Internet, namely global scale-out. In the Internet context, a successful deployment of this approach will depend on the ability of achieving the following two principal goals: i) dynamically binding the client to the most convenient replica server, and ii) maintaining data consistency among the replica servers.

In the geographically distributed architectures the requests assignment process can occur in two steps: a first dispatching level where the authoritative Domain Name Server (DNS) of the Web site or another centralized entity selects the target Web server, and a second dispatching level carried out by each Web server through some request redirection mechanism. DNS-based dispatching was originally conceived for locally distributed Web systems. It works by intervening on the address lookup phase of the client request. Load sharing is implemented by translating the site hostname into the IP address of the selected Web server. When the authoritative DNS server provides the address mapping, it can use various dispatching policies to select the best server, ranging from simple static round-robin to more sophisticated algorithms that take into account both client and server state information [103].

Most implemented distributed Web servers evaluate client-to-server network proximity, so that the DNS can return the IP address of the server closest to the user [104]. The goal is to limit the network latency component in the response time. The main problem of DNS dispatching is its limited control on workload reaching the Web site, because of hostname-to-IP caching occurring at various network levels. In
particular, the authoritative DNS of highly popular sites can provide only a very coarse
distribution of the load among the Web servers, as it controls less than 5-7% of requests
reaching the Web site. Furthermore, heterogeneous Web traffic arrivals due to domain
popularity and world time zones are highly amplified by the geographical contest.
Indeed, a geographically distributed Web site that tends to serve closest requests only,
may risk to be highly unbalanced because the amount of request from an Internet
region is strictly dependent on day time. The consequence of time zones and proximity
algorithms alone is to have one or two highly loaded servers in two regions and other
almost idle servers. To address DNS (centralized) dispatching issues we can add a
second level dispatching mechanism. The most common is a distributed dispatching
policy that is carried out by the critically loaded Web servers through some redirection
mechanisms, for example HTTP redirection [103], or IP tunneling [105]. Moreover,
DNS-based solutions are very slow (or unable) to detect server failures and additions of
new servers. Furthermore, while the server host may be running properly, the specific
server software (e.g. httpd) may have failed. In a pathological case, a load balancing
DNS tool may see that a server is under-loaded (because the httpd demon failed) and
give it even higher priority.

As an alternative solution, a distributed Web clusters system can consist of
distributedly distributed nodes, each composed of a cluster of servers. A distributed
Web cluster has one hostname and an IP address for each Web cluster. The requests to
a Web cluster are scheduled through one of the mechanisms devoted to the locally
distributed Web systems. We can distinguish the proposed architectures on the basis of
dispatching levels, typically two or three. The first level dispatching among the Web
clusters is typically carried out by the authoritative DNS of the Web site or another
entity that implements some proximity dispatching strategy. The second level
dispatching is carried out by the Web switches that dispatch client requests reaching the
cluster among the local Web server nodes. Commercial products that provide global
load balancing by implementing this class of architectures include Cisco's
DistributedDirector [104], IBM's Network Dispatcher [106] and Resonate's Global
Dispatch [107].

The main problem is that dispatching algorithms based on network proximity are
not able to react immediately to heavy load fluctuations of Web workload that are
amplified by the geographical context. Therefore, it seems convenient to integrate the
two level dispatching architecture with a third level assignment activated by each Web
server through the HTTP redirection mechanism [108]. This third level dispatching mechanism allows an overloaded Web cluster to easily shift away some portion of load assigned by the first dispatching level. The third dispatching level is necessary to guarantee scalability and load balancing of geographically distributed Web sites, and to enhance quality of Web services by augmenting the percentage of requests with guaranteed response time. On the other hand, request redirection should be used selectively because additional round trip time risks to increase latency time experienced by users.

### 5.1.4 The Parallel Downloading Approach

In the Internet context, a successful deployment of the geographically distributed replica servers approach will depend on the ability of achieving the following two principal goals: i) dynamically binding the client to the most convenient replica server, and ii) maintaining data consistency among the replica servers.

Unfortunately, as previously described, neither of these two goals is easy to achieve. Firstly, the dynamic binding of clients to replica servers can turn out to be difficult to implement, owing to the location based naming scheme used in the Web. This scheme provides a one-to-one mapping (i.e., the Uniform Resource Locator - URL) between a name of a resource and a single physical copy of that resource; hence, dynamic binding of a client to distinct replica servers requires that the client-side software be adequately extended in order to be able to select an available replica, at run-time. Secondly, maintaining replica consistency on a large geographical scale can be hard to achieve, without affecting the overall service performance. In addition, the Internet environment is subject to (real or virtual) partitions that can prevent communications between functioning nodes; hence, within this environment, both clients and replica servers may hold mutually inconsistent views of which replica server is available and which is unavailable.

In view of these observations, we have developed a mechanism, that, in order to provide the clients of a replicated Web service with service responsiveness, exploits the parallelism inherent in the replicated servers architecture that implements that service. Specifically, the principal goal of this mechanism is to minimize what we term the User Response Time (URT), i.e. the time elapsed between the generation of a browser request for the retrieval of a Web page, and the rendering of that page at the browser
site (issues of replica consistency fall outside the scope of this mechanism). To this end, rather than binding a client to the most convenient replica server, as proposed in [25], our mechanism intercepts each client browser request for a Web page, and fragments that request into a number of sub-requests for separate parts of that document. Each sub-request is issued to a different available replica server, concurrently. The replies received from the replica servers are reassembled at the client end to reconstruct the requested page, and then delivered to the client browser.

Our mechanism is designed so as to adapt dynamically to state changes in both the network (e.g. route congestion, link failures), and the replica servers (e.g. replica overload, unavailability). To this end, our mechanism periodically monitors the available replica servers and selects, at run-time, those replicas to which the sub-requests can be sent, i.e. those replicas that can provide the requested page fragments within a time interval that allows our mechanism to minimize the URT. As our mechanism implements effectively load distribution of client requests among the available replicas, we have named it Client-Centered Load Distribution (C\textsuperscript{2}LD). C\textsuperscript{2}LD can be implemented as an extension of the client-side software (i.e., in the browser) or as a module of a Proxy server to which the clients send their requests.

The design of C\textsuperscript{2}LD is based on an analytical model that we have developed. This model is essential in order to determine the size of the page fragment that can be requested to each replica, and the extent of the monitoring period C\textsuperscript{2}LD is to use in order to execute a Web page request.

We have assessed the effectiveness of our C\textsuperscript{2}LD mechanism by validating it through both an experimental evaluation over the Internet, and simulation. In order to carry out experimental evaluation, we have developed a simple Web service implemented by four replica servers located in Italy, the UK, and the USA, and interconnected via the Internet. A browser program, incorporating our C\textsuperscript{2}LD mechanism, was used to access Web pages from that service. The results of this evaluation, discussed in this chapter, show the viability of our approach. However, as our experimental evaluation was based on a single client, these results were not sufficient to show the effectiveness of our C\textsuperscript{2}LD mechanism in the general case in which a replicated Web service is accessed by a large number of clients (say, hundreds), concurrently.

Thus, as it was unpractical to experiment our mechanism in one such real Internet-based scenario, we developed a simulation of that scenario within which we
have evaluated the implementation of our mechanism both as part of the client software, and as part of a HTTP Proxy server. In our simulation, both the workload that can be experienced by four replica servers, under different distribution of client requests, and the network load can be specified by means of simulation parameters.

The following of this chapter is structured as follows. In the next section we discuss the principal issues we have addressed in the design of our C\textsuperscript{2}LD mechanism and introduce the analytical model we have developed in order to address those issues. In section 5.3 we provide details of the implementation of the C\textsuperscript{2}LD mechanism we have carried out. Section 5.4 discusses our validation exercise and illustrates both the experimental and the simulative results we have obtained from that exercise. Finally, Section 5.5 provides some concluding remarks.

### 5.2 Design Issues

The timeliness requirement to be met by our C\textsuperscript{2}LD mechanism can be expressed by means of a User Specified Deadline (USD), i.e. a value that indicates the extent of time a user is willing to wait for a requested Web page to be rendered at his/her workstation. However, as the USD value is user specified, it may differ from the actual response time a browser request may experience over the Internet (i.e. the URT previously introduced), at least in principle; thus, if a user sets an unrealistic USD, the C\textsuperscript{2}LD returns an appropriate exception (in our implementation, the USD is configured by the user prior to the invocation of the browser, and then captured by the C\textsuperscript{2}LD mechanism). Owing to this USD time constraint, each replica server that receives a sub-request for a page fragment must honor that sub-request within a time interval that allow the C\textsuperscript{2}LD mechanism to reconstruct the requested page, out of all the received fragments, before the USD deadline expires. Thus, it is crucial that the C\textsuperscript{2}LD mechanism assess accurately both the size of the fragment each replica is to supply, and the time intervals within which these fragments are to be received at the client site.

To this end, we have developed the analytical model summarized below. In this model, we make use of the HEAD and GET HTTP1.1 types of requests (methods). A HTTP HEAD request provides its invoker with details about a requested Web page (e.g., the total length of that page). A HTTP GET request returns a requested Web page to its invoker. In addition, by setting the option RANGE in a GET request, the invoker
of that request can specify that only a particular range of bytes within a Web page (i.e., a page fragment) be returned.

Assume that exactly $N_{REP}$ replica servers be available, and that the size of the requested Web page be known, and be equal to $DS$ bytes. Moreover, assume that the total amount of time needed to download an entire page be exactly equal to the sum of $N_{INT}$ consecutive time intervals, each of which has a duration of $S$ seconds (i.e., $URT = N_{INT} \cdot S$), and that the time $T$ be set to the value 0 immediately before $C^2$LD begins the downloading of the requested page.

Thus, at time $T=0$, the $C^2$LD issues a HTTP HEAD request to each replica server $i$, in order to obtain the total size of a requested page. Based on the total page size, the $C^2$LD mechanism issues consecutive HTTP GET sub-requests to each replica $i$ in order to fetch page fragments, possibly of different size. The size of each page fragment is determined dynamically by $C^2$LD, based on the measured URT of each replica, as described below.

We denote with the index $r (r \geq 1)$ each single sub-request that can be issued to the replica $i$, $i \in \{1, ..., N_{REP}\}$. Using the replica index $i$ and the sub-request index $r$, we can define the following mapping $k_{i,r} \in \{1, ..., N_{INT}\}$:

$$k_{i,r}(T) = (T \div S) + 1,$$

(1)

where $T$ denotes the exact time instant in which a given sub-request $r$ is issued to the replica $i$, and $(T \div S) + 1$ denotes the time interval containing $T$. Thus, $k_{i,r}(T)$ is the index of the time interval in which the request $r$, directed to the replica $i$, must be sent. For the sake of simplicity, in the following the term $k_{i,r}$ will be used in place of $k_{i,r}(r)$.

Finally, let us denote with $DR_{i,r}$ the data rate that a given replica server $i$ is able to provide during a given time interval $k_{i,r}$, and with $PS_{i,r}$ the size of the document fragment requested to a certain replica server $i$ with the request $r$. Note that the data rate $DR_{i,r}$ may vary unpredictably in different time intervals. In addition, a sub-request $r$, submitted to a certain replica server during a given time interval $k_{i,r}$, may terminate during some later interval $h$ (i.e., $h \geq k_{i,r}$). Hence, a realistic model must be able to represent both the time interval in which each sub-request is transmitted to each replica, and the exact sequence of all the sub-requests transmitted to each replica. Based on these assumptions, the analytical model we have developed is as follows.
In order to assess the data rate that will be provided by a given replica $i$, we adopt the following measurement-based strategy. Assume that the sub-request $r$ is issued to the replica $i$ at the time instant $T$, in the interval $k_{i,r}$, immediately after the termination of the request $r-1$; then, the data rate for the sub-request $r$ can be estimated as:

$$DR_{i,r} = \frac{PS_{i,r-1}}{URT_{i,r-1}},$$

where $PS_{i,r-1}$ represents the size of the document fragment downloaded with the previous GET sub-request, terminated at the time $T$. $URT_{i,r-1}$ is the elapsed time experienced for downloading the previous document fragment of size $PS_{i,r-1}$ (for the sake of simplicity, the term $PS_{i,0}$ represents the number of bytes returned as response to the HEAD request sent to the replica $i$). Note that the $URT_{i,r-1}$ value may be experimentally measured at the time $T$, when the previous sub-request $r-1$ has been completed. Given that value, the size of the next document fragment to be requested to the replica $i$ is:

$$PS_{i,r} = DR_{i,r} \cdot S_{i,r}^*$$

with

$$S_{i,r}^* = k_{i,r} \cdot S - T$$

and

$$(k_{i,r} - 1) \cdot S \leq T < k_{i,r} \cdot S.$$ 

In Eq. (3) above, the term $S_{i,r}^*$ represents the $URT$ expected from the execution of the sub-request $r$ directed to the replica $i$. Depending on the value of $T$, one of the following two events may occur:

1. the sub-request $r-1$ has been completed exactly at the beginning of the interval $k_{i,r}$, when the value of $T$ is equal to $(k_{i,r} - 1) \cdot S$, or
2. the sub-request $r-1$ has been completed during the $k_{i,r}$ interval, i.e. $(k_{i,r} - 1) \cdot S < T < k_{i,r} \cdot S$.

If event 1 occurs, the size of the requested page fragment is proportional to the total duration $S$ of a complete interval. Instead, if event 2 occurs, the size of the requested fragment is proportional to the residual time $k_{i,r} \cdot S - T$ needed to reach the end of the $k_{i,r}$ interval.
Finally, the following requirement must be met by all the replica servers, in order to ensure that a requested page be entirely downloaded within the USD deadline:

\[ DS \sum_{i=1}^{N_{\text{REP}}} \frac{\sum_{r_{-1}}^{N_{\text{REQ}}} DR_{i,r} \cdot URT_{i,r}}{N_{\text{INT}} \cdot S} \leq \text{USD}, \]  

(6)

where \( \{1,...,N_{\text{REQ}}\} \) is the number of GET requests completed by the replica \( i \), within the USD deadline.

The Eq. (6) above states that the sum of the average data rates provided by all the replica servers during the downloading period \( N_{\text{INT}} \cdot S \) must be such that the requested page is completely downloaded before the USD deadline expire. The term \( \sum_{r_{-1}}^{N_{\text{REQ}}} DR_{i,r} \cdot URT_{i,r} \) in the Eq. (6) represents the average data rate \( DR_i \) that a given replica \( i \) has been able to provide during the downloading period.

We wish to point out that our model provides a measurement-based strategy for assessing the actual data rate each replica can provide, as the size of a fragment to be requested in a sub-request \( r \) depends upon the measurement of the \( URT_{i,r-1} \) value experienced at the time the (previous) sub-request \( r-1 \) terminates. Based on the model above, the design of our C\(^2\)LD mechanism can be summarized as follows.

In general, in order to access a Web service, a user invokes a browser by providing it with the URL of that service. However, as motivated earlier, our C\(^2\)LD requires that the user of Web service set his/her own USD timeout value before invoking the browser that he/she will use to access that service. Thus, our mechanism provides its users with a configuration procedure that allows them to set their own USD timeout values before invoking their browser programs (a default USD value is used, in our implementation, if a user does not make use of that configuration procedure). Once the USD has been set, the C\(^2\)LD mechanism operates as described below.

Access to a Web service from a browser entails that a HTTP GET method be invoked by that browser. The C\(^2\)LD mechanism intercepts that HTTP GET invocation, starts the USD timeout and, using the URL in the GET invocation, interrogates the DNS. The DNS maintains the IP addresses of the \( N_{\text{REP}} \) replica servers that implement a Web service. When C\(^2\)LD submits a request to the DNS for resolving a URL, the DNS returns the IP addresses of all the replica servers associated to that URL. As the replica
servers’ addresses are available to the \( C^2LD \) mechanism, this mechanism interacts with each replica as illustrated by the skeleton code in Figure 1, and summarized below.

\( C^2LD \) invokes a HTTP HEAD method on each replica \( i \). The reply from replica \( i \) to the first HTTP HEAD invocation is used by \( C^2LD \) to: i) get the size of the requested page, ii) estimate the current data rate that replica \( i \) can provide, and iii) assess the size of the first fragment that can be fetched from that replica.

\( C^2LD \) maintains a global variable (\textit{download\_done}, in Fig. 5.1) that indicates whether or not all the fragments of a requested page have been delivered. Until a page is not fully downloaded, \( C^2LD \) uses the Eq. (2) and (3) in Section 2 to compute the size of the fragment that is to be requested to the replica \( i \).

```
/* C^2LD */
...
within USD do   /* set USD timeout */
...  
HEAD(...)   /* send HEAD request to replica i */
URT(i) := ...  /* assess URT replica i can provide */
PS(i,1) := ...  /* compute 1st fragment size for replica i */
within S do   /* set timeout of length S */
    if not download\_done then  /* check if page download completed */
        GET (...)  /* get fragment from replica i */
        \[ DR_{i,r} = \frac{PS_{i,r-1}}{URT_{i,r-1}} \]  /* compute expected data rate, based on URT of previous request */
        \[ PS_{i,r} = DR_{i,r} \cdot S_{i,r} \]  /* compute size of next fragment to be requested */
    else
        return;
    od
...   /* handle S timeout exception */
od
```

\textbf{Figure 5.1:} Implementation of the \( C^2LD \) Service

In order to adjust adaptively to possible fluctuations of the communication delays that may occur over the Internet, the fragment size is computed each time a fragment is to be requested, based on the value of the \textit{URT} experienced in fetching the previous fragment. Thus, in essence, as the GET request \( r-1 \) directed to a given replica \( i \) terminates, a new GET request \( r \) can be issued to the replica \( i \) with the fragment size value \( PS_{i,r} \), computed on the basis of the \textit{URT}\(_{i,r-1}\) response time. Once the requested fragment size has been calculated, \( C^2LD \) issues a HTTP GET request to the replica \( i \), in
order to retrieve the required fragment of that size. (Specifically, a fragment of $Z$ bytes
size is requested by invoking a HTTP GET method with the following option set:
"Range: bytes=Y-X", where $X$ and $Y$ denote the bytes corresponding to the beginning
and the end of the requested fragment of size $Z$, respectively.)

Note that some of the replica servers may not respond timely to the HTTP
(HEAD and GET) invocations described above (e.g., they may be unavailable owing to
network congestion). Thus, $C^2LD$ associates a timeout to each HTTP request it issues
to each server $i$. If that timeout expires before $C^2LD$ receives a reply from a replica
server $i$, it assumes that the server $i$ is currently unavailable, and places it in a stand by
list. Replica servers in that list are periodically probed to assess whether they have
become active again. Requests for replicas in the stand by list are redirected to active
replicas.

5.3 Implementation Notes

We have developed the implementation of our mechanism using the Java programming
language and the Java 2 Software Development Kit. In essence, the $C^2LD$ basic
mechanism is implemented as a set of threads, as depicted in Fig. 5.2. These threads
share a lightweight data structure. This data structure is used to maintain shared
variables, such as the USD, the download_done, and the stand by
list, mentioned
above. Semaphores, implemented using the synchronized Java statement, govern the
access to this data structure.

The following threads implement our mechanism. An interface thread intercepts
the browser GET request, and activates the get_document thread. This latter thread
implements the mechanism introduced in the previous section for fetching a Web page,
and returns to the browser the retrieved page, encapsulated into a HTTP message. A
timer thread implements the timeout mechanism and maintains the stand by list; a
monitoring thread manages the monitoring period $S$. Finally, HTTP_connection
threads are used to establish and maintain the connections with the replica servers;
these threads are responsible for both assessing the URT provided by those servers, and
managing the communications with them. Each HTTP_connection thread is connected
to a single replica server. In order to minimize the communication overheads caused by
the connection establishment and release operations, each of these threads maintains
the connection with its replica server 'open' for the duration of a browser session. Thus,
a \textit{HTTP\_connection} thread can transmit consecutive requests to the replica to which it is connected without incurring in the additional costs entailed by opening and closing a connection.

As mentioned earlier, our mechanism can be incorporated either in the browser software or in a HTTP Proxy server. In the former case, the C$^2$LD implementation intercepts the HTTP GET method invocation issued by the browser, and acts as previously described. In contrast, in the latter case, a Proxy server maintains an instance of the C$^2$LD mechanism for each browser accessing that server, as illustrated in Fig. 5.3. Thus, in essence, a Proxy server dispatches requests from its client browsers to their relative C$^2$LD instances, and delivers responses from those instances to their relative browsers.

5.4 Validation

The effectiveness of our C$^2$LD implementation has been assessed through the following two separate evaluation exercises. The first of these exercises has consisted of experimenting our mechanism using the actual Internet. In this evaluation, a single browser program, incorporating our mechanism and accessing a geographically replicated Web service, was involved.

The second evaluation exercise has been based on simulation; specifically, as part of this exercise, we have developed a simulation scenario in which multiple clients accessed the same replicated Web service. In addition, within this exercise, we have evaluated the two C$^2$LD implementations described earlier, i.e., the implementation of the C$^2$LD mechanism within the browser program, and the implementation of our mechanism within a Proxy server. In the following sub-sections we discuss our evaluation exercises, in isolation.

5.4.1 Single Client Experimentation

A large number of experiments (4000, approximately) were carried out during a two-month period. These experiments consisted essentially of a client program (i.e. a browser) downloading Web documents of different size from up to 4 geographically distributed replica servers. These documents were downloaded by the same client
program using both the C2LD mechanism, and the standard HTTP GET downloading mechanism, for comparison purposes.

In this sub-section, we describe in detail the scenario within which these experiments have been carried out, introduce the metrics we have used to assess our implementation, and discuss the performance results we have obtained.

**Experimental Scenario**

The performance of our C2LD implementation has been evaluated using the Internet to connect a client workstation (equipped with our C2LD software) with four replica servers.

The client workstation was a SPARCstation 5 running the SunOS 5.5.1 operating system and the Sun Java Virtual Machine 1.2. This workstation was located at the Computer Science Department of the University of Bologna. The four different replica servers were running the Apache Web server over the Linux platform.

For the purposes of our evaluation, these servers were located in four distinct geographical areas. Specifically, a replica server was located close to the client workstation; namely, at the Computer Science Laboratory of the University of Bologna, in Cesena. This Laboratory is four network hops far from our Department in Bologna. The connection between our Department and this Laboratory has a limited bandwidth of 2 Mbps, and is characterized by a rather high packet loss rate (between 2% and 9%). A second replica server was located in northern Italy; namely, at the International Center of Theoretical Physics in Trieste. This server was reachable through 9 network hops via a connection whose bandwidth ranged between 8 and 155 Mbps.

A third replica server was located at the Department of Computing Science of the University of Newcastle upon Tyne (UK). This server was reachable through 15 network hops from our client workstation.

Finally, a fourth replica server was located at the Computer Science Department of the University of California at S. Diego. This server was reachable through a 19 network hops transatlantic connection. Fig. 5.4, obtained with the *VisualRoute* utility [109], depicts the locations of both the client and the replica servers used in our experiments, and the routes between this client and those servers; this Figure shows that the different routes connecting our client with the four replica servers scarcely
overlap (i.e., route overlapping occurs only up to Milan, when communicating with the replica servers in Trieste, Newcastle, and San Diego).

Figure 5.2: Thread structure of C2LD

Figure 5.3: C2LD as part of the Proxy Service
Within this scenario, a replicated Web service can be configured so as to use one of the following 11 combinations of the four available replica servers. Namely, a service can be replicated across the two servers in Cesena and Newcastle (C+N, in the following), only, or those in Cesena and Trieste (C+T), or in Trieste and Newcastle (T+N), or in Cesena and S. Diego (C+S), or in Trieste and S. Diego (T+S), or Newcastle and S. Diego (N+S). A more redundant service can be implemented across one of the following four combinations of three servers, instead: Cesena, Newcastle and Trieste (C+N+T); Cesena, Newcastle, and S. Diego (C+N+S); Cesena, Trieste, and S. Diego (C+T+S); Trieste, Newcastle, and S. Diego (T+N+S). Finally, a redundant Web service can be implemented across the four replica servers in Cesena, Newcastle, Trieste, and S. Diego (C+N+T+S). Our experiments have been carried out using all these 11 combinations of replica servers. It is worth noting that the four machines running the server replicas were moderately loaded during our experiments. In contrast, the routes to the European Web replica servers were heavily loaded during daytime; network traffic over these routes typically decreased during the night.

![Figure 5.4: Routes between the client and the Web replica servers](image)

The average network traffic conditions on both the European and transatlantic routes, experienced during the evaluation of our C^2LD mechanism, are reported in the following Table 5.1. Specifically, in this Table, the first row indicates the 90% percentile of the Round Trip Time (RTT) obtained with the ping routine from our client workstation in Bologna to the four replica servers; the second row reports the minimum RTT experienced over the routes to those servers; the third row reports the packet loss rate we measured over those routes, as obtained with the ping routine (ICMP).

Finally, note that the network traffic almost saturated the bandwidth of 8 Mbps, available at the University of Bologna routers, during working hours. The Fig. 5.5 and
5.6 depict the traffic statistics at that router, measured in Mbps, on a daily and weekly basis, respectively. In these Figures, the solid area represents the amount of the outgoing traffic generated by the University of Bologna; the black line represents the amount of the incoming traffic, instead. As expected, the ingoing traffic almost saturates the available bandwidth of 8 Mbps during working hours.

<table>
<thead>
<tr>
<th>ping from Bologna to:</th>
<th>Cesena</th>
<th>Trieste</th>
<th>Newcastle</th>
<th>S. Diego</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTT 90% of arrived pkt (ms)</td>
<td>107</td>
<td>95</td>
<td>160</td>
<td>450</td>
</tr>
<tr>
<td>RTT min (ms)</td>
<td>10</td>
<td>38</td>
<td>59</td>
<td>190</td>
</tr>
<tr>
<td>Lost Packet</td>
<td>2%-9%</td>
<td>0%</td>
<td>0%</td>
<td>0%-3%</td>
</tr>
</tbody>
</table>

**Table 5.1:** The measured round trip time and lost packet percentage

![Figure 5.5: Daily Traffic Statistics](Image)

![Figure 5.6: Weekly Traffic Statistics](Image)

**Measures**

The following two metrics have been used to evaluate the effectiveness of our C²LD mechanism: i) the percentage of document retrieval requests successfully satisfied by our mechanism, and ii) the URT (as defined earlier) our mechanism provides. Both these metrics capture the principal user requirements; namely, that a requested document be effectively retrieved, and that it be done timely. Based on these metrics, the evaluation of our mechanism has been carried out using different values of the
following four parameters: i) the number of server replicas that implement a given Web service, ii) the data rate that each different server replica can provide, iii) the download monitoring period adopted by the C²LD mechanism, and, finally, iv) the size of the document to be retrieved.

The performance of our C²LD mechanism has been compared and contrasted with that provided by the standard HTTP document retrieval mechanism, as this is implemented by the HTTP GET function. To this end, each replica server maintained a set of downloadable files of different size, ranging from 3 Kbytes to 1 Mbytes. These servers were requested to retrieve the same Web document, at the same time of the day (i.e. under the same network traffic conditions, approximately) using, alternatively, both the standard HTTP GET request, and our mechanism. The download monitoring period $S$ used by our mechanism ranged from 50 milliseconds to 10 seconds.

A number of experiments were carried out for each given file size and download monitoring period $S$. Each experiment consisted of 15 consecutives download requests. 4 out of 15 of these requests were executed by invoking the HTTP GET function; the other 11 requests were executed by the C²LD mechanism. The experiments used files whose size was 3, 10, 30, 50, 100, 200, 500 and 1000 Kbytes. For each file size, the experiments with the C²LD mechanism were repeated using a different download monitoring period, ranging from 50 milliseconds to 10 seconds.

As mentioned earlier, our experiments were carried out on the 11 possible configurations of a replicated service, introduced in sub-section 4.1.1; however, the values reported in the Tables and Figures in this sub-section are the average of the results obtained in all our experiments.

<table>
<thead>
<tr>
<th>File Size</th>
<th>USD</th>
<th>C²LD</th>
<th>S. Diego (HTTP)</th>
<th>Newcastle (HTTP)</th>
<th>Trieste (HTTP)</th>
<th>Cesena (HTTP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50 Kbytes</td>
<td>120</td>
<td>0 %</td>
<td>0.3 %</td>
<td>0 %</td>
<td>0 %</td>
<td>1.65 %</td>
</tr>
<tr>
<td>100 Kbytes</td>
<td>120</td>
<td>0 %</td>
<td>0.25 %</td>
<td>0 %</td>
<td>0 %</td>
<td>5 %</td>
</tr>
<tr>
<td>200 Kbytes</td>
<td>180</td>
<td>0 %</td>
<td>0.2 %</td>
<td>0 %</td>
<td>0 %</td>
<td>4 %</td>
</tr>
<tr>
<td>500 Kbytes</td>
<td>300</td>
<td>0 %</td>
<td>0.3 %</td>
<td>0 %</td>
<td>0 %</td>
<td>1 %</td>
</tr>
<tr>
<td>1 Mbyte</td>
<td>600</td>
<td>0 %</td>
<td>0.45 %</td>
<td>0 %</td>
<td>0 %</td>
<td>2.5 %</td>
</tr>
</tbody>
</table>

Table 5.2: Fault Percentage

As a first result, Table 2 summarizes the page loss percentage obtained with our mechanism, and that obtained with the standard HTTP GET downloading mechanism.
It can be seen from this Table that the C\textsuperscript{2}LD mechanism provides a highly available service, since it always guarantees the downloading of the requested document. Instead, the standard HTTP mechanism is not always able to provide its client with that document, as shown by the page loss percentage experienced by the S. Diego and Cesena servers, in particular.

The two higher curves, instead, represent the URT values provided by the two fastest Web replica servers, when interrogated with the standard HTTP downloading mechanism. Specifically, the URT values provided by these two replica servers were obtained as follows. The four different replica servers were exercised with the standard HTTP mechanism; then, out of these four servers, the URT results obtained by the two fastest ones were selected to plot the graph in Fig. 5.7.

As shown in that Figure, the performance of the C\textsuperscript{2}LD mechanism and the HTTP mechanism are equivalent when the document size less than 50 Kbytes. Instead, when the document size is larger than 50 Kbytes, the C\textsuperscript{2}LD mechanism outperforms the standard HTTP downloading mechanism. As already mentioned, the C\textsuperscript{2}LD URT values in Fig. 5.7 represent an average URT value, as it results from all the experiments we have carried out. For the sake of simplicity in this figure we do not report the value of the variance of the URT, as measured in our experiments. However, this value for the
C\textsuperscript{2}LD mechanism varied from 0.099 s to 5 s, approximately, depending on the size of the requested page. In addition, it may be interesting to assess the URT improvement that can be obtained by our mechanism as the number of replica servers, concurrently used, varies. To this end, we have carried out experiments in which the number of active replicas was varying from 1 to 4.

Fig. 5.8 illustrates the results of our experiments, in these four cases. Specifically, in these cases, the URT is measured as a function of the file size. As shown in this Figure, the larger the number of replicas, the better the URT values; in addition, as the file size increases, the advantage of using the C\textsuperscript{2}LD mechanism becomes more notable. Table 3 shows the average percentage improvement of the URT values that has been experienced in the experiments depicted in Fig. 5.8, as the number of replica servers grows.

We have experimented our C\textsuperscript{2}LD mechanism using different values of the download monitoring period parameter S, in order to assess the influence of that parameter on the URT provided by our mechanism. Specifically, we have measured the C\textsuperscript{2}LD URT using values of the monitoring period S ranging from 50 ms to 10 s. Fig. 5.9 illustrates the results of our experiments. This Figure reports the different URT curves that were obtained using files of different size.

![Figure 5.8: URT improvement provided by C2LD as the number of replicas grows](image-url)
Chapter 5. The Replicated Web Service Case Study

<table>
<thead>
<tr>
<th>Number of Replica Servers</th>
<th>URT percentage improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4%</td>
</tr>
<tr>
<td>3</td>
<td>17.2%</td>
</tr>
<tr>
<td>4</td>
<td>21.5%</td>
</tr>
</tbody>
</table>

**Table 5.3:** Average URT percentage improvement

<table>
<thead>
<tr>
<th>monitoring period (milliseconds)</th>
<th>URT (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50K</td>
<td>1000K</td>
</tr>
<tr>
<td>200K</td>
<td>500K</td>
</tr>
<tr>
<td>100K</td>
<td>200K</td>
</tr>
<tr>
<td>50K</td>
<td>100K</td>
</tr>
<tr>
<td>30K</td>
<td>50K</td>
</tr>
<tr>
<td>10K</td>
<td>30K</td>
</tr>
<tr>
<td>3K</td>
<td>10K</td>
</tr>
<tr>
<td>1K</td>
<td>3K</td>
</tr>
</tbody>
</table>

**Figure 5.9:** URT depending on the monitoring period

Note that the monitoring period that provides the lowest URT values is approximately 1 second, regardless of the file size.

Finally, for the sake of completeness, Fig. 5.10 and 5.11 report two additional graphs. These graphs illustrate the performance results obtained by both requesting each replica to fetch a 500 Kbytes file, using a standard HTTP GET request, and exercising our mechanism in order to fetch the same file. The monitoring period used by our mechanism in these experiments ranged from 500 to 2000 milliseconds. Specifically, the histograms in Figure 8 represent the URT values obtained by the C^2LD mechanism (denoted as parallel in this Figure), and the URT values obtained by each single replica. Each histogram illustrating the performance of our mechanism relates to a different combination of the server replicas used during the experiments, as indicated in the Figure 8. Thus, for example, the leftmost parallel histogram reports the URT values obtained with our mechanism when only the replica servers in Cesena and Newcastle were used; the rightmost parallel histogram reports the URT values obtained...
with the $C^2LD$ mechanism when all the four replica servers were used. The remaining histograms (other than the parallel histogram) relate to the individual replica servers, as indicated in the Figures.

![Figure 5.10: C2LD vs. single replicas performances](image1)

**Figure 5.10:** C2LD vs. single replicas performances

Fig. 5.11 compares the percentage of URT improvement obtained by the $C^2LD$ mechanism with that obtained by each single replica (interrogated using HTTP GET). Both Fig. 5.10 and 5.11 show that, in general, the $C^2LD$ mechanism outperforms each single replica in all cases; the only exception occurs when the $C^2LD$ uses only two replica servers, and, out of these two servers, one is very fast (Trieste or Newcastle), and the other is very slow (i.e., San Diego).

![Figure 5.11: Percentage improvement of the C2LD mechanism](image2)

**Figure 5.11:** Percentage improvement of the C2LD mechanism
Finally, in order to assess the workload caused by our C²LD mechanism in each Web server, we show below the number of sub-requests (HTTP GET RANGE) generated by our mechanism to retrieve files of different sizes, when using the best monitoring period (S=1 s). Specifically, Figures 5.12, 5.13, 5.14 and 5.15 report the average number of sub-requests that C²LD sends to the different replica servers of a specified Web service, in order to retrieve a document of a given size, namely 3, 30, 100, and 1000 Kbytes. Recall that, in addition to the GET requests, the C²LD sends a HTTP HEAD request to each replica. As shown in Figure 10, for very small document size (e.g., 3 Kbytes) C²LD sends a GET request only to a single replica. Instead, as the document size increases, C²LD distributes fairly the requests among all the replicas, sending the same number of requests to each replica, and requiring larger fragments from the faster replicas (see Figures 5.13, 5.14, and 5.15).

It is worth observing that the total number of sub-requests, directed toward a Web service (i.e. the sum of the requests directed to all the replica servers implementing that service) increases as the number of replicas increases, as illustrated in the Fig. 5.16. However, the workload of each replica decreases if the number of replicas involved in the Web service increases, as the number of sub-requests directed to each replica server of the Web service reduces (see Fig. 5.17).

![Figure 5.12](image-url)  
*Figure 5.12: Average Number of sub-requests directed to each replica when retrieving a document with size 3 Kbytes*
Figure 5.13: Average Number of sub-requests directed to each replica when retrieving a document with size 30 Kbytes

Figure 5.14: Average Number of sub-requests directed to each replica (100kb)

Figure 5.15: Average Number of sub-requests directed to each replica when retrieving a document with size 1 Mbytes
Figure 5.16: Average Number of sub-requests directed to a Web service

Figure 5.17: Average Number of sub-requests directed to each replica of a Web service

Moreover, the average size of each requested fragment and the response time of each request, increase less than proportionally respect to the increase of the requested document size, as illustrated in Figures 5.18 and 5.19. We suspect that this behavior can be caused by the “TCP slow start” mechanism [110]. Slow start uses a window to the sender’s TCP: the congestion window, called cwnd. When a new connection is established, the congestion window is initialized to one segment (i.e., the segment size announced by the other end). Each time an ACK is received, the congestion window is increased by one segment (cwnd is maintained un bytes, but slow start always increments it by the segment size). The sender can transmit up to the minimum of the congestion window and the advertised window. The congestion window is flow control imposed by the sender, while the advertised window is flow control imposed by the
receiver. The sender starts by transmitting one segment and waiting for its ACK. When
that ACK is received, the congestion window is incremented from one to two, and two
segments can be sent. When each of those two segments is acknowledged, the
congestion window is increased to four. This provides an exponential increase. At some
point the capacity of the Internet can be reached, and an intermediate router will start
discarding packets. This tells the sender that its congestion window has gotten too
large. Due to that slow start mechanism, when small size files are fetched the capacity
of the Internet cannot be reached does not allow one to transmit large fragments when
small size files are fetched owning to the TCP sliding window flow control policy.

**Figure 5.18:** Average size of requested fragments, with monitoring period =1 s

**Figure 5.19:** Average URT for each sub-request, with monitoring period =1 s
5.4.2 Multiple Client Simulation

The results discussed in the previous sub-section indicate that our C^2LD mechanism, when incorporated in a browser's software, can provide a notable speed-up in the communications between that browser and a replicated Web service, compared to the HTTP standard document fetching policy. However, these results are not sufficient to illustrate the effectiveness of our mechanism when multiple clients access concurrently the same replicated Web service (regardless of whether the mechanism be implemented as part of a browser or as part of a Proxy server).

As it was impractical to use the experimental scenario described in sub-section 4.1.1 with a very large number of clients accessing concurrently our four replica servers, we developed a simulation model of that scenario. Based on that model, we have simulated both the browser-based and the Proxy-based architectures. Each of these two architectures has been exercised and evaluated through the implementation of one of the following three Web page downloading policies:

A1: The C^2LD downloading policy described in Section 2;
A2: The same policy without the adaptive mechanism that negotiates dynamically the page fragment size;
A3: The simple downloading policy described in [24], that consists of binding the client to the replica that replies first to the initial HEAD invocation.

The combination of the two architectures and the three policies introduced above has given rise to the 6 different simulation scenarios, discussed in this sub-section. In order to compare and contrast the results obtained from these 6 scenarios with those discussed in the previous sub-section, both the communication infrastructure and the replicated Web services simulated within each of these scenarios are the same as those discussed in sub-section 4.1.1.

Hence, the simulated Web service architecture and communication infrastructure consists of the four servers located in S. Diego, Newcastle, Trieste and Cesena. The client program, or the Proxy server, used to carry out our assessment, are located in Bologna; the workload caused by additional clients has been modeled as described later.

The URT that each server provides consists of the transmission delay between each server and the client, and the processing delay caused by the workload of the server. In case the C^2LD mechanism is implemented by a Proxy server, this URT
includes both the communication delay between the Proxy and the client, and the Proxy
delay caused by its own workload.

The simulation parameters introduced above are detailed in the following.

As far as the transmission delay is concerned, we have chosen for all the Internet
connections to the servers a Gaussian distribution with mean $\mu$ and standard deviation
$\sigma$. The two parameters of such a distribution vary from server to server and have been
estimated as follows through the ping program for a Maximum Transmission Unit
(MTU) of 1.5 Kbytes: $\mu = 9$ ms and $\sigma = 1$ ms for Cesena, $\mu = 27$ ms and $\sigma = 8$ ms for
Trieste, $\mu = 35$ ms and $\sigma = 5$ ms for Newcastle, and $\mu = 100$ ms and $\sigma = 10$ ms for San
Diego. Given these figures, the transmission delay for downloading of a Web page
fragment of X Kbytes from a given server is obtained from the basic transmission delay
of that server by assuming an increment equal to 50% of the basic transmission delay
for each additional slot of 1.5 Kbytes of X, as suggested in [110]. Note that the
transmission delays obtained with this method are sensibly lower than those measured
in our experiments discussed in sub-section 4.1.2. This discrepancy is due to the fact
that after our experimentation, the Internet connections provided by the University of
Bologna were upgraded, and the ping program used to obtain the basic figures above
was executed after this upgrade.

As far as the server load is concerned, we have expressed it through an
exponential distribution in order to test the responsiveness of the three fetching policies
A1, A2, and A3, introduced above, in a pessimistic environment characterized by a
high degree of variability. The average waiting time in each of the four servers is taken
to be 200 ms; hence, the rate of the exponential distribution above is 0.005 requests/ms.
The transmission delay between the client and the Proxy follows a Gaussian
distribution with $\mu = 5$ ms and $\sigma = 1$ ms. This simulation parameters have been chosen
based on the assumption that a Proxy server is expected to provide a client with a
response time shorter than the one that can be provided by the fastest available replica
server. The Proxy load follows an exponential distribution with average waiting time
equal to 50 ms (which is less than the load of the servers).

To conclude the description of our simulation model, the following observations
are in order. Every simulation experiment for each given simulation scenario has
consisted of 20 simulation runs. Each simulation run consisted of 10 Web page requests
of the same size. The page size used in different simulation runs varied from 100 Kbps
to 1000 Kbps, as described later. The simulation results have been obtained with 90% confidence level. The 180 simulation experiments have been conducted using the Solaris operating system on a PC with a 600 MHz CPU and a 128 Mbytes RAM. The 70% of the simulation experiments lasted less than 1 min; the longest simulation experiment lasted 7 min.

Finally, our 6 simulation models have been specified using the \textsc{ÆMPA} technology \cite{111}. \textsc{ÆMPA} is the architectural and visual version of the stochastic process algebra EMPA$_{gr}$ \cite{112}. The 6 \textsc{ÆMPA} based specifications have been processed by the simulative software TwoTowers \cite{113}; this tool has produced the simulation results discussed in the following sub-section.

\textbf{Simulation Results}

We have assessed the responsiveness of the three fetching policies A1, A2 and A3 for the following Web page sizes: 100, 250, 500, 750, and 1000 Kbytes. In the simulation of the policy A1, we have used the following monitoring periods: 300, 600, and 900 ms. The simulation results are reported in Fig. 5.20.

In addition to the results provided by the three policies A1, A2, A3 this Figure shows the URT provided by each single server, accessed via the standard HTTP mechanism; the related curves are labeled with the name of the corresponding server.

As expected, Cesena achieves the best performance, with the standard HTTP mechanism, followed by Trieste, Newcastle and San Diego. If policy A3 is used, we obtain a URT which is worse than that of the Cesena server alone, as the A3 policy suffers from an overhead induced by the initial phase in which the HTTP HEAD request is transmitted. However, A3 performs better than the three individual servers at S. Diego, Newcastle and Trieste that use the standard HTTP mechanism.

Policy A2 performs better than A3 for increasing values of the page size. A3 outperforms A2 for small page sizes (up to 300 Kbytes size) because A3 starts operating as soon as the fastest server replies to the HTTP HEAD requests. In contrast, A2 waits for the response from all the four servers before starting operating, thus resulting in a greater overhead that cannot be compensated for by the load distribution over the four servers in case of small pages. Finally, if algorithm A1 is used, we can observe first that the achieved URT critically depends on the duration of the monitoring period. If the duration is 300 ms, we see a poor performance which is only better than that using the San Diego server alone, with the standard HTTP mechanism.
The reason is that, if we examine the configuration parameters, on average the S. Diego server replies to a HTTP request after $100 + 200 + 100 = 400$ ms; hence, at the end of each monitoring period of 300 ms, most of the page fragments will be downloaded from the Cesena, Trieste, and Newcastle servers.

The URT improves notably when the monitoring period is 600 ms; in this case A1 outperforms A3 for page sizes greater than 900 Kbytes, and it is even better with a 900 ms monitoring period; in this case, A1 outperforms A2 for page sizes greater than 300 Kbytes.

Some of the results illustrated in Fig. 5.20 can be compared with the corresponding results discussed in sub-section 4.1.2. These two sets of results coincide, from a qualitative viewpoint, as the relationship among the URT values are preserved.

In contrast, from a quantitative point of view, Fig. 5.20 shows better results than the experimental ones discussed in sub-section 4.1.2. This is due to the upgrade of the Internet connection at the University of Bologna, mentioned earlier.

**Figure 5.20: URT Measurements - Browser based Scenario**
The Figures 5.21, 5.22 and 5.23 illustrate the results obtained from the Proxy based simulation of the C^2LD implementation. Each of these Figures show the simulation results obtained using a particular hit ratio. Namely, a hit ratio=20% is assumed in Figure 5.21, a hit ratio=30% is assumed in Figure 5.22, and a hit ratio=40% is assumed in Figure 5.22 [114].

These results show that the Proxy based implementation of the mechanism preserves the mutual qualitative relationships among the different Web page fetching policies we have considered. In addition these results show that the URT values decrease as the hit ratio of the Proxy cache increases. Specifically, the URT improvement can be estimated as ranging from 10% to 20% as the Proxy cache hit ratio grows from 20% to 40%.

![Figure 5.21: URT Measurement - Proxy based Scenario (hit ratio = 20%)](image-url)
Figure 5.22: URT Measurement - Proxy based Scenario (hit ratio = 30%)

Figure 5.23: URT Measurement - Proxy based Scenario (hit ratio = 40%)
As a final remark, we wish to point out that the policy A1, with a 900 ms monitoring period, provides one with better performance than any other policies we have considered, on average.

The only exception to this remark is one when a specific replica server, such as our server in Cesena, responds much faster than all the other replica servers. (In our evaluation, the Internet connection between Bologna and Cesena provided a 34 Mb/s bandwidth.) Hence, in one such particular situation, it may not be convenient to make use of any load distribution policy among replicated servers.

5.5 Concluding Remarks

In this chapter, we have discussed a mechanism we have developed to construct responsive Web services. We have shown, through both real experiments and simulation, that this mechanism can be extremely effective in order to minimize the URT, if implemented either as part of a browser software or as part of a Proxy server. We have applied our C^2LD mechanism to the fetching of generic Web resources, such as files and documents; we wish to assess the adequacy of our mechanism when deployed for accessing digital video and audio resources. In addition, we are planning to extend the work described in this chapter by addressing the following topics. Firstly, we wish to examine strategies that optimize the routing of requests to replica servers. Secondly, we wish to extend our mechanism to deal with dynamic Web services. Thirdly, we wish to introduce prefetching [115] to our mechanism. Fourth, we wish to investigate policies for maintaining data consistency among geographically distributed replica servers.

Finally, we wish to point out that the proposed adaptive policy, based on the monitoring of the performance of the different replicas, provide the users with timely responses and high availability.
Chapter 6

The Voice-based Communication Case Study

In this chapter, we describe the design and the experimental evaluation of a playout delay control mechanism we have developed in order to support unicast, voice-based audio communications over the Internet. The proposed mechanism was designed to dynamically adjust the talkspurt playout delays to the traffic conditions of the underlying network without assuming either the existence of an external mechanism for maintaining an accurate clock synchronization between the sender and the receiver during the audio communication, or a specific distribution of the audio packet transmission delays. Performance figures derived from several experiments are reported that illustrate the adequacy of the proposed mechanism in dynamically adjusting the audio packet playout delay to the network traffic conditions while maintaining a small percentage of packet loss.

6.1 Introduction

Sophisticated applications of Internet multimedia conferencing will become increasingly important only if the quality of the communications will be perceived as sufficiently good by their users. The result of extensive experiments has shown that audio is frequently perceived as one of the most important component of multimedia conferencing [116]. A number of problems have been identified which negatively impacts the quality of audio conversations, but probably the more critical one with audio is the loss of audio packets. Basically, two are the main causes for audio packet loss over wide-area packet-switched networks: 1) traffic congestion at the interconnecting routers that cause audio packets to be discarded, and 2) too large transmission delays that cause audio packets to arrive at the destination past the time instant at which they are scheduled to be played out (the playout point). With the term playout delay we refer to the total amount of time that is experienced by the audio
packets of a given talkspurt from the time instant they are generated at the source and the time instant they are played out at the destination. Summarizing, such a playout delay consists of: i) the "collection" time needed for the transmitter to collect audio samples and to prepare them for transmission; this time strongly depends on the operating system scheduler then, as pointed out in [117], modifying the operating system scheduler to provide bounded dispatch latency for application may be an interesting idea; ii) the "transmission" time needed for the transmission of audio packets from the source to the destination over the underlying transport network, and finally iii) the "buffering" time, that is the amount of time that a packet spends queued in the destination buffer before it is played out. A crucial tradeoff exists between audio packet playout delay and audio packet loss: the longer the scheduled playout delay, the more likely it is that an audio packet will arrive at the destination before its scheduled playout deadline has expired. However, if on one side a too large percentage of audio packet loss (over 5-10%) may impair the intelligibility of an audio transmission, on the other side, too large playout delays (e.g. more than 200-250 msec) may disrupt the interactivity of an audio conversation [118]. The main purpose of this paper is to describe a playout delay control mechanism that is suitable for adjusting the talkspurt playout delays of unicast, voice-based audio communications across the Internet. The mechanism was designed to dynamically adjust the talkspurt playout delays to the network traffic conditions without assuming either the existence of an external mechanism for maintaining an accurate clock synchronization between the sender and the receiver, or a specific distribution of the end-to-end transmission delays experienced by the audio packets. Succinctly, the technique for dynamically adjusting the talkspurt playout delay is based on obtaining, in periodic intervals, an estimation of the upper bound for the packet transmission delays experienced during an audio communication. Such an upper bound is periodically computed using round trip time values obtained from packet exchanges of a three-way handshake protocol performed between the sender and the receiver of the audio communication. At the end of such protocol exchange, the receiver is provided with the sender's estimate of an upper bound for the transmission delay that can be used in order to dynamically adjust the talkspurt playout delay. The proposed mechanism guarantees that the talkspurt playout delay may be dynamically set from one talkspurt to the next, without causing gaps or time collisions (formally defined in the Sections 6.3.1) inside the talkspurts themselves, provided that intervening silence periods of sufficiently long duration are exploited for
the adjustment. The need of silent intervals for allowing the mechanism to adjust to the fluctuating network conditions is common to the most part of the existing audio tools (e.g. NeVoT, vat and rat, [119, 120, 116]) but renders the proposed scheme particularly relevant for voice-based applications where conversational audio with intervening silence periods between subsequent talkspurts is transmitted. The design of our mechanism was completed during the Summer of 1997. A prototype version of the mechanism running on workstations equipped with the SunOS 4.3 (BSD Unix) operating system, and based on the datagram based UDP protocol was soon carried out. Based on that prototype implementation, several experiments were conducted over an (IP based) internetworked connection between the University of Bologna (Italy) and the C.E.R.N. Institute in Geneva (Switzerland). The performance figures derived by the experimentations conducted on the field illustrated the adequacy of our mechanism in dynamically adjusting the audio packet playout delay to the traffic conditions of the underlying network while maintaining a small percentage of packet loss. The paper is structured as follows. In the next section, we discuss some background issues that we have regarded as important for the design of our mechanism. The remaining Sections (3 and 4) describe and discuss: i) the proposed playout delay control mechanism, ii) a prototype implementation we have developed, ii) the results of an experimental assessment we have carried out, and finally iii) a performance comparison between our mechanism and another adaptive playout delay adjustment mechanism recently proposed [118]. We conclude this introduction by summarizing the main features of our playout delay control mechanism. It provides: 1) an embedded and accurate algorithm that maintains tight time synchronization between the sender's system clock and the clock that supports the playout process at the receiving host, during an audio conversation; 2) a method for adaptively estimating the audio packet playout time (on a per-talkspurt basis) with an associated minimal computational overhead for both the source and the destination hosts, and 3) an exact and simple method for dimensioning the playout buffer depending on the network traffic conditions.

6.2 Packetized Audio over the Internet

Since the early experiments with packetized voice in the Arpanet network [4], packetized audio applications have become sophisticated tools that many Internet users try to use with regularity. For example, the audio conversations of many international
conferences and workshops are now usually conducted over the Mbone (the multicast backbone), an experimental overlay network of the Internet [121]. The audio tools that are used to transmit packet audio over the Internet (e.g. NeVot [119], vat [120], rat [116], the INRIA audio tool [73]) typically operate by periodically sampling audio streams generated at the sending host, packetizing them, and transmitting the obtained packets to the receiving site by using datagram based connections (e.g. UDP). In addition, at the receiving site, packets are buffered and their playout time is delayed in order to compensate for variable network delays that may be frequently experienced. Such playout mechanisms try to adaptively adjust the playout delay in order to keep this delay as small as possible while minimizing the number of packets that arrive too late (i.e. after their playout point). The next section provides additional information that constitute the background of the algorithm to be presented in this paper. In particular, the main characteristics of the mechanisms that are used to adaptively adjust the playout time for audio packets over the Internet are reviewed.

6.2.1 Background

A typical audio segment may be considered as constituted of talkspurt periods during which the audio activity is carried out, and silence periods during which no audio packet is generated. In order for the receiving site to reconstruct the audio conversation, the audio packets constituting a talkspurt must be played out in the order they were emitted at the sending site. If the delay between the arrival of subsequent packets is constant (i.e. the underlying transport network is jitter-free) a receiving site may simply play out the arriving audio packets as soon as they are received. Unfortunately, this is only rarely the case, since jitter-free, ordered, on-time packet delivery almost never occurs in today's packet-switched networks. Those variations in the arrivals of subsequent packets strongly depend on the traffic conditions of the underlying network. Packet loss percentages (due to the effective loss and damage of packets as well as late arrivals) often vary between 15% and 40% [118]. In addition, extensive experiments with wide-area network testbeds have shown that the delays between consecutive packets may also be as much as 1.5 seconds, thus impairing real-time interactive human conversations. New protocol suites such as the Resource Reservation Protocol (RSVP) [7] might eventually ameliorate the effect of jitter and improve the quality of the audio service over the Internet, but they are not yet widely used. On the other hand, the most
used approach is to adapt the applications to the jitter present on the network. Hence, to transport audio over a non-guaranteed packet-switched network, audio samples are encoded (usually with some form of compression), inserted into packets that have creation timestamps and sequence numbers, transported by the network, received in a playout buffer, decoded in sequential order, and finally played out by the audio device, as seen in Fig. 6.1. A symmetric scheme is used in the other direction for interactive conversation. The smoothing playout buffer is used at the receiver in order to compensate for variable network delays. Received audio packets are queued into the buffer, and the playout of each packet of a given talkspurt is delayed for some quantity of time beyond the reception of the first packet of that talkspurt. In this way, dynamic playout buffers can hide, at the receiver, packet delay variance at the cost of additional delay. A crucial tradeoff exists between the length of the imposed additional quantity of delay and the amount of lost packets due to their late arrival: the longer the additional delay, the more likely it is that a packet will arrive before its scheduled playout deadline. However, too long playout delays may in turn seriously compromise the quality of the conversation over the network.

![Audio data flow over the Internet.](image)

**Figure 6.1:** Audio data flow over the Internet.

Typical acceptable values for the end-to-end delay between packet audio generation at the sending site and its playout time at the receiver are below the threshold of 200-250 msec, furthermore a percentage of no more than 5 - 10% of packet loss is considered quite tolerable in human conversations [73]. Besides adjusting the audio playout delay in order to compensate for the effect of the jitter, modern audio tools typically make
also use of error and rate control mechanisms based on a technique known as forward error correction (FEC) to reconstruct many lost audio packets [73].

More generally, compensating for loss using end-to-end protocols and algorithms can be done using a number of mechanisms [122][123], including local repair (interpolation of missing data using the surrounding packets) and interleaving. There has been much interest in the use of packet-level FEC for sending redundant information ahead of time to compensate for loss, based on parity codes, [124], [125], Reed Solomon codes [126], and redundant speech codecs [127][128] [129] [130]. Oftentimes, the term FEC is only applied to the traditional channel coding approaches, such as parity and Reed Solomon codes. For our purposes, we define FEC as any mechanism which sends additional information along with the media stream, the purpose of which is to aid in packet recovery. For example, the INRIA audio tool adjusts the audio packet send rate to the current network conditions, adds redundant information to packets (under the form of highly compressed versions of a number of previous packets) when the loss rate surpasses a certain threshold, and establishes a feedback channel to control the send rate and the redundant information. Simply put, the complete process is controlled by an open feedback loop that selects among different available compression schemes and the amount of redundancy needed, as described in the following. If the network load and the packet loss are high, the amount of compressed redundant information carried in each packet is increased by adding to each packet compressed version of the previous two to four audio packets. In 5-seconds intervals the receiver reports (using the Real Time Protocol suite RTP-RTCP [131]) quality of service reports to the sender in order to regulate and adapt the quantity of redundant information being sent.

Compensating for jitter is accomplished primarily through adaptive playout buffer algorithms (also known as adaptive playout delay algorithms) [132] [133] [134] [118]. These algorithms generally work by taking some measurements on the delays experienced by packets, and updating the playout delay on a talkspurt by talkspurt basis. As discussed above, efficient playout adjustment mechanisms have been developed to minimize the effect of delay jitter. Typically, a receiving site in an audio application buffers packets and delays their playout time. Such a playout delay may be kept constant for the duration of the audio conversation, or dynamically adjusted from one talkspurt to the next. Due to the fluctuating end-to-end (application-to-application) delays experienced over the Internet, constant, non-adaptive playout delays may result
in unsatisfactory quality for audio applications. Hence, two are the approaches widely exploited for adaptively adjusting playout time: the former approach keeps the same playout delay constant throughout a given talkspurt, but permits different playout delays in different talkspurts. In the latter approach, instead, the playout delay is adjusted on a per-packet basis. However, an adaptive adjustment on a per-packet basis may introduce gaps inside talkspurt and thus is considered as of being damaging to the perceived audio quality. On the contrary, the variation of the playout delay from a talkspurt to the next may introduce artificially elongated or reduced silence periods, but this is considered acceptable in the perceived speech if those variations are reasonably limited. Hence, the totality of the above mentioned tools adopt a mechanism for adaptively adjusting the playout delays on a per-talkspurt basis. However, in order to implement such a playout control mechanism, almost all the above cited audio applications make use of the following two strong assumptions.

1. An external mechanism exists that keeps synchronized the two system clocks at both the sending and the receiving site. Usually, the IP-based Network Time Protocol (NTP) is used for this purpose.

2. The delays experienced by audio packets on the network follow a Gaussian distribution.

Extensive experiments have been carried out that shown that the playout delay control mechanisms based on that two assumptions above may be adequate to obtain acceptable values for the tradeoff between the average playout delay and the loss due to late packet arrivals. However, in some circumstances, the cited mechanisms may suffer from a number of problems, especially when they are deployed over wide-area networks. In particular, the following problems may be pointed out [118,135]:

- The ``external'' software-based mechanisms (e.g. the NTP protocol) used to maintain the system clocks synchronized at both the sending and the receiving sites are not typically widespread all over the Internet. In addition, those mechanisms may turn out to be too much inaccurate to cope with the real-time nature of the audio generation/playout process. For example, even if the NTP protocol may achieve computer clock synchronization within a few tens of milliseconds over most paths in the Internet of today, however, there may be frequent exceptions with synchronization values up to a few hundreds of milliseconds, especially if a client host is not directly connected to a primary server of the NTP hierarchy but achieves synchronization through a stratum-2 (or higher) server via a congested link [136].
The problem with clock synchronization is that if the two different clocks (respectively, at the source and at the destination) do not run at the same rate and the synchronization mechanism is not sufficiently accurate, they will tend to drift further and further apart. Extensive experiments have shown that the above mentioned behavior may have a very negative impact on the provided formulas for the calculation of the playout time, thus resulting in an increased number of lost packets [135].

- The widely adopted assumption that the packet transmission delays over the Internet follow a Gaussian distribution seems to be a plausible conjecture only for those limited time intervals in which the overall load of the underlying network is quite light. Indeed, recent experimental studies carried out over the Internet have indicated the presence of frequent and conspicuously large end-to-end delay spikes for periodically generated packets (as is the case with audio packets) [73, 137].

Moreover, the study of FEC for loss recovery, and playout buffer adaptation for jitter compensation, have proceeded independently. However, in [138] Rosenberg, Qiu and Schulzrinne have pointed out that there is a coupling between the two. All of the FEC mechanisms send some redundant information which is based on previously transmitted packets. Waiting for the redundant information results in a delay penalty, and consequently an increase in size of the playout buffers. When network loss rates are high, accepting the delay penalty for increased recovery capabilities is appropriate. However, when network loss rates are low, the FEC may not provide useful information, and increasing the playout buffer sizing to wait for it is not appropriate. The result is that playout buffer adaptation should depend on both FEC and network loss conditions and network jitter. However, existing tools that utilize FEC (such as rat from UCL and freephone from INRIA) use decoupled adaptation algorithms. These algorithms compute some playout delay as if FEC were absent, compute some delay needed to make use of FEC, and then combine these two delays together. These decoupled algorithms may insert insufficient delay when network loss probabilities are substantial, and too much delay when they are not, resulting in poor performance. We further observe that existing playout buffer adaptation algorithms generally aim to minimize the network losses at the expense of delay. Owning from these facts, coupling packet loss and jitter into playout buffer adaptation appears a good choice.
6.3 A Novel Mechanism for Packetized Audio

The adaptive mechanism for the control of the playout delay proposed in this section ameliorates all the negative effects of the audio tools reported above, while maintaining satisfiable values of both the average playout delay and the packet loss due to late arrivals. The proposed policy assumes neither the existence of an external mechanism for maintaining an accurate synchronization at both the sending and the receiving sites, nor a Gaussian distribution for the end-to-end transmission delays of the audio packets. In particular, it provides:

- an internal and accurate mechanism that maintains tight time synchronization between the system clocks of both the sending and the receiving hosts;
- a method for adaptively estimating the audio packet playout time (on a per-talkspurt basis) with an associated minimal computational overhead;
- an exact and simple technique for dimensioning the playout buffer depending on the traffic conditions of the underlying network.

In the following, a description of the main ideas behind the mechanism is provided. For continuous playout of audio packets at the receiving site, it is essential that the audio packets be available at the receiver prior to their respective playout time and that the rate of consumption (i.e. playout) of packets at the receiver meets the rate of transmission at the sender [139]. Hence, when the sender transmits the first packet of an audio talkspurt, it timestamps that packet with the value (say $C$) of the reading of its own clock. As soon as this first packet arrives at the receiver, it sets the clock that supports the playout process (say $CR$) by using the $C$ value, i.e.,

$$CR = C;$$

and immediately schedules the presentation of that first packet. Subsequent audio packets belonging to the same talkspurt are also timestamped at the sender with the value of the reading of the sender's clock at the time instants when the packets are transmitted. When these subsequent packets arrive at the receiving site, their attached timestamp is compared with the value of the reading of clock that supports the playout process at the receiving host (the receiver's clock, for short). If the timestamp attached to the packet is equal to the value of the receiver's clock, that packet is immediately
played out. If the timestamp attached to the packet is larger than the value of the receiver's clock, that packet is buffered and its playout time is scheduled after a time interval equal to the positive difference between the value of the timestamp and the actual value of the receiver's clock. Finally, if the timestamp attached to the packet is smaller than the value of the receiver's clock, the packet is simply discarded since it is too late for presentation. However, (even if an identical clock rate is assumed) due to the fluctuating delays in real transmissions, the values of the clocks of the sender and of the receiver may differ, at a given time instant, by the following quantity

\[ C_S(T) - C_R(T) = \Delta \]

where \( C_S(T) \) and \( C_R(T) \) are, respectively, the reading of the local clocks at the sender and at the receiver (at the same instant \( T \)), and \( \Delta \) is a non negative quantity ranging between 0 (a theoretical lower bound) and \( \Delta_{\text{max}} \) (a theoretical upper bound on the transmission delay introduced by the network between the sender and the receiver). Later, we will show a technique for the estimation of \( \Delta \), and how this value impacts the calculation of the playout time for the audio packets.

A crucial issue of the mechanism is an accurate dimensioning of the playout buffer for audio packets. Both buffer underflow and overflow may occur, thus resulting in discontinuities in the playout process. In order to master all the possible problems deriving from both buffer underflow and overflow, a simple technique is proposed that was first used in [139] in order to guarantee the continuity of the playout process controlled by an MPEG codec for video frames. Such a technique accurately distinguishes among the two possible cases of buffer underflow and overflow. The worst case scenario for buffer underflow (corresponding to the case when packets arrive too late for presentation) is clearly when the first packet arrives after a minimum delay (e.g. 0), while subsequent packets arrive with maximum delay (e.g. \( \Delta_{\text{max}} \)). In this case, due to the minimum delay of the first packet, there is a null difference between the clock at the sender and at the receiver

\[ C_S(T) - C_R(T) = 0 \]

However, consider now a situation in which a subsequent packet arrives at the receiver that suffers from the maximum delay \( \Delta_{\text{max}} \). Suppose that this packet has been transmitted by the sender when its clock shows a time value equal to say \( X \). Due to the imposed synchronization, at that precise instant also the receiver's clock would show a
value equal to $X$. Now, adding the transmission delay of $\Delta_{\text{max}}$, the arrival time of this subsequent packet occurs when the receiver's clock shows the value given by $X + \Delta_{\text{max}}$. Unfortunately, that packet would be too late for playout and consequently discarded. This example suggests that a practical and secure method for preventing buffer underflow (i.e. packets lost due to their late arrival) is that the receiver delays the setting of its local clock of an additional quantity equal to $\Delta_{\text{max}}$, when the first packet of the talkspurt is received. Precisely, when the first packet is received with its timestamp equal to $C$, the receiver sets its local clock to a value equal to $C - \Delta_{\text{max}}$:

$$CR(T) = C - \Delta_{\text{max}}$$

With this simple modification (see Fig. 6.2) the problem of buffer underflow gets solved. Simply put, this policy implicitly guarantees that all the audio packet that will suffer from a transmission delay not greater than $\Delta_{\text{max}}$ will be on-time for the playout.

**Figure 6.2**: Delayed setting of the receiver's clock for preventing buffer underflow.

However, the above mentioned technique introduces another problem: that of playout buffer overflow. The worst case scenario for buffer overflow occurs in the following circumstance: the first packet of a talkspurt suffers from the maximum delay $\Delta_{\text{max}}$, instead a subsequent audio packet experiences the minimum delay 0. At the arrival of the first packet of the talkspurt at the receiving site, the receiver sets its clock equal to the value timestamped in the packet (say $C$) only after $\Delta_{\text{max}}$ time units since the packet is arrived. Due to this setting and to the maximum delay experienced by the first packet of the talkspurt the time difference between the two clocks at the sender and at the receiver at a given time instant $T$ is equal to

...
\[ C_S(T) - C_R(T) = 2 \Delta_{\text{max}} \]

Now, if a subsequent packet arrives at the receiver that has experienced only a minimum delay equal to 0, the receiver’s clock, upon the reception of that packet, shown a time value equal to

\[ C_R(T) = C - 2 \Delta_{\text{max}} \]

where \( C \) is the timestamp attached to the first packet of the talkspurt. From the formula above it is clear that, in order for each packet with an early arrival to have room in the playout buffer, additional buffering space is required at the receiving site equal to the maximum number of audio packets that might arrive in a time interval of \( 2 \Delta_{\text{max}} \) (see Fig. 6.3). In conclusion, the example above dictates that the playout buffer dimension may never be less than the maximum number of packets that may arrive in an interval of \( 2 \Delta_{\text{max}} \).

![Figure 6.3: Additional buffering required for preventing buffer overflow.](image)

Nevertheless, two problems have been left unresolved:

1. the accurate estimation of the value \( \Delta_{\text{max}} \) of the difference between the sender's and the receiver's clocks at a given time instant \( T \), and
2. the dynamical adaptation of the proposed playout control mechanism in order to compensate for the highly fluctuating end-to-end transmission delays that may be experienced over wide-area packet-switched networks such as the Internet.

The following section is devoted to describe a technique that may be used to evaluate an upper bound on the maximum experienced delay, and also to adapt the proposed playout control mechanism to the highly fluctuating transmission delays of wide-area packet-switched networks.

### 6.3.1 Adaptive Adjustment of the Mechanism

A simple technique may be devised to estimate an upper bound for the maximum transmission delay. This technique exploits the so called Round Trip Time (RTT) and is based on a three-way handshake protocol. It works as follows. Prior to the beginning of the first talkspurt in an audio conversation, a probe packet is sent from the sender to the receiver timestamped with the clock value of the sender (say $C$). At the reception of this probe packet, the receiver sets its own clock with the value of the timestamp attached to the probe packet, and sends immediately back to the sender a response packet with the same timestamp $C$. Upon the reception of this response packet, the sender computes the value of the $RTT$ by subtracting from the current value of its local clock the value of the timestamp $C$. At that moment, the difference between the two clocks, respectively at the sender and at the receiver, is equal to an unknown quantity (say $t_0$) which may range from a theoretical lower bound of 0 (that is, all the $RTT$ value has been consumed on the way back from the receiver to the sender), and a theoretical upper bound of $RTT$ (that is all the $RTT$ has been consumed on the way in during the transmission of the probe packet). Unfortunately, a time difference of only $t_0$ between the sender's and the receiver's clocks could not be sufficient to prevent packet loss due to late arrivals, as well as a rough approximation of this value (e.g. $t_0 = RTT/2$) might result in both playout buffer overflow problems and packet loss due to premature arrivals. Based on these considerations, the sender, after having received the response packet from the receiver and having calculated the $RTT$ value, sends to the receiver a final installation packet, with piggybacked on it the previously calculated $RTT$ value. Upon receiving this installation packet, the receiver sets the time of its local clock by subtracting from the value shown at its clock the value of the transmitted $RTT$. Hence,
at that precise moment, the difference between the two clocks at the receiver and at the sender is equal to a value given by

$$\Delta_{\text{max}} = C_S(T) - C_R(T) = t_0 + RTT,$$

where $\Delta_{\text{max}}$ ranges in the interval $[RTT, 2RTT]$.

In essence, with the strategy above, a maximum transmission delay equal to $\Delta_{\text{max}}$ is left to the audio packets to arrive at the receiver in time for playout, and consequently a playout buffering space proportional to $\Delta$ is required for packets with early arrivals. In order for the proposed policy to adaptively adjust to the fluctuating network delays experienced over the Internet, the above mentioned synchronization technique is first carried out prior to the beginning of the first talkspurt of the audio conversation, and then periodically repeated throughout the entire conversation. The adopted period is about 1 second in order to prevent the two clocks (possibly equipped with different clock rates) from drifting apart. Thus, each time a new RTT value is computed by the sender, it may be used by the receiver for dynamically setting both the value of its local clock and the playout buffer dimensions. This method guarantees that both the introduced additional buffering delay and the buffer dimension are always proportioned to the traffic conditions. However, it may be not possible to replace on-the-fly during a talkspurt the current values of the receiver's clock and the dimensions of its playout buffer. In fact, as earlier mentioned, such an instantaneous adaptive adjustment of the synchronization parameters might introduce either gaps or even time collisions inside a talkspurt. (For a formal definition of gaps and time collisions see [26]). Based on this consideration, the installation at the receiver of the values of a new synchronization (namely, the activity of changing the values of the receiver's playout clock and of the buffer dimension) is carried out only during the periods of audio inactivity, when no audio packets are generated by the sender (i.e. during silence periods between different talkspurts). In [26] it is shown in details how the installation of a new synchronization between the sender and the receiver may be conducted during a silence period detected by the sender without introducing either gaps or time collisions inside the talkspurts of the audio conversation.
6.3.2 Detecting and smoothing out playout delay spikes

A possible problem with the playout delay adjustment mechanism that has been proposed in this paper is related to the possible large value for the obtained RTT value, that may be caused by the fact that either the probe or the response packet suffers from a very large transmission delay spike. Due to that, a very large playout delay value (termed playout delay spike) may be introduced that impairs the interactivity of the audio conversation. For example, consider a case when, due to a given synchronization $i$, an $RTT_i$ value of 100 msec has been obtained. Based on this value, the use of the delay playout adjustment mechanism presented in this paper would have the effect to introduce at the receiver a playout delay for the audio packets ranging in the interval $[100; 200]$ msec. Suppose now that a subsequent synchronization $j$ is performed that obtains an $RTT_j$ value equal to 600 msec. This very large value for $RTT_j$ would have the effect of setting the playout delay value to a very large value ranging in the interval $[600; 1200]$ msec. This very large playout delay may be not considered tolerable for an interactive audio conversation. Summarizing, a very large RTT value obtained with a given synchronization may cause an intolerable playout delay value that disrupts the interactivity of the conversation. Nevertheless, that problem may get solved by adopting a policy which was inspired by the delay spike detection and management mechanism proposed in [118]. Our policy works by using two different modes of operation depending on whether a transmission delay spike has been detected or not. In the normal mode (i.e. no transmission delay spike has been detected) the playout delay adjustment mechanism operates by calculating the playout delay to be introduced in the playout process, just as already described. Instead, in the spike-detected mode (i.e. a very large transmission delay spike has been detected) the very large $RTT$ value obtained from the spike-affected synchronization is smoothed out by multiplying it with a smoothing factor $k$ ($k<1$). Let us denote, in the following, with $RTT^*_j$ such a smoothed value obtained with the synchronization $j$. Each audio packet that arrives at the receiver after a spike-affected synchronization $j$ has been installed is played out after a playout delay value equal to $t_j + RTT^*_j$.

In the following we give a high level description of the policy we use to detect a playout delay spike. That policy is based on the comparison between the $RTT$ values obtained from two consecutive synchronization activities $j$ and $i$ ($j>i$). For ease of
understanding, the policy is presented in C-language-like pseudo code in Table 6.1. For each most recently computed $RTT_j$ value, the algorithm checks the current mode and, if necessary, switches its mode to the other one (lines 1-9 in the leftmost side of the table). More precisely, if the most recently computed $RTT_j$ value is larger than some multiple $h$ ($h > 1$) of the $RTT_i$ value then a delay spike is considered to be detected, and the algorithm switches to the spike-detected mode (lines 5-7 in the leftmost side of the table). In such spike-detected mode, the smoothed value $RTT^*_j$ is computed and then used for playout (lines 1-3 in the rightmost side of the table). The end of a spike is detected similarly. If the most recently computed $RTT_j$ value is smaller than some multiple $h$ of the $RTT$ value experienced when the spike-affected period began, the mode is reset to normal (lines 2-4 in the leftmost side of the table). Clearly, when the algorithm operates in the normal mode, the normal $RTT_j$ value is used at the receiver for calculating the playout delay (lines 4-5 in the rightmost side of the table).

(1) For each newly computed RTT \(j\)

(2) IF (mode == SPIKE)

(3) IF (RTT\(j\) <= \(h\) old-RTT\(_i\))

(4) mode = NORMAL

(5) ELSE (mode == NORMAL)

(6) IF (RTT\(j\) > \(h\) RTT\(_i\))

(7) mode = SPIKE

(8) old-RTT\(_i\) = RTT\(_i\)

(9) RTT\(_i\) = RTT\(_j\)

Table 6.1: Playout delay spike management: delay spike detection (left) and playout delay estimation

6.4 Experimental Evaluation of the Mechanism

A working prototype implementation of the playout control mechanism presented in this paper was performed using the C programming language, and the development environment provided by the SunOS 4.3 (BSD Unix) operating system. Using such a prototype implementation, an initial extensive experimentation was carried out aiming
at measuring the performance of the mechanism. In the following three sections, the prototype implementation used for the experimentation, the obtained experimental results, and a set of comparative simulation results are presented.

6.4.1 Prototype Implementation and Measurement Architecture

The Unix socket interface and the datagram based UDP protocol were used to transmit and receive the sampled audio packets. The coding schemes that were used to produce the audio packets use 8-kHz sampled speech with bit rates varying from 5.3-kbit/sec to 8-kbit/sec. In particular, all the audio packets used to perform the measurements were produced using a codec based on the ITU-T G.729 standard [140] that provides coding of speech at 8-kbit/sec while maintaining toll-level audio quality. In addition, in order to reconstruct possibly lost (or corrupted) audio packets, a forward error correction-based mechanism was implemented that was able to add to a given packet redundant information (under the form of a highly compressed version of the previous packet). To this aim, a codec based on the ITU-T G.723.1 standard [141] was exploited that was able to code speech of a low, synthetic quality at 5.3-kbit/sec. Such an alternative coding scheme was adopted based on the consideration that it may provide redundant information by adding only a small amount of byte overhead per packet. This redundant information was piggybacked in the packet following that containing the primary speech codeword. Thus, the loss of an individual packet can be repaired using the redundant information carried by the following packet [116]. Needless to say, the use of this redundancy technique entails an increase in the overall delay due to both the algorithmic delay of the coding process and the reconstruction delay introduced at the receiver. In substance, the measurements were all done using audio packets that were produced by exploiting both the G.729 and the G.723.1 based codecs.
The resulting audio packets all had the structure represented in Figure 6.4. In particular, besides the IP and UDP headers (respectively of 20 and 8 bytes), each packet includes a 30 msec of speech coded with the G.729 primary coding algorithm, and a 30 msec of previous speech coded with the G.723.1 secondary coding algorithm. The 30 msec of the current speech result in a 30 bytes field (denoted "F" in Figure 6.4), while the 30 msec of previous speech are coded using the 20 bytes field (denoted "P" in Figure 6.4).

In addition, to each audio packet two timestamps are associated: the first timestamp (the "Tf" field in Figure 6.4) records using 4 bytes the time of generation of the 30 msec of the current speech. Instead, the second timestamp (the "Tp" field in Figure 6.4) records using 4 bytes the time of generation of the previous 30 msec of speech. The time values recorded by the timestamps use 0:1 msec as time base unit. Hence, a 4 byte timestamp field may be enough to timestamp audio packets for a consecutive 100 hours long period of time. No sequence number was attached to the packet in order to reduce the total amount of bytes needed, but the timestamps were used, at the application level, both to measure end-to-end transmission delays and also to detect packet loss. In addition, the audio packets were also used in the prototype implementation of our protocol for implementing two out of the three phases of the synchronization activity, namely the "probing" and the "installation" phases. Hence, a 4 byte field (denoted "T") was included in each packet to carry the RTT i value needed for the installation of the synchronization i. Moreover, since each audio packet was emitted by the sender only at regular intervals of 30 msec, the 4 byte field "W" was included to record the total quantity of time elapsed from the time instant when the RTT value was available at the sender, and the time instant when that RTT value was effectively transmitted. That time value was used at the receiver in order to calculate the average time of completion of the synchronization activity. Finally, an additional 2 byte field (termed "A" in Fig. 6.4) was used for numbering subsequent synchronizations. Summarizing, the total number of bytes used for producing the IP-based audio packets amounts to exactly 96 bytes, out of which 28 are used to encode both the IP and the UDP headers, while the remaining 68 bytes are used to encode all the audio data and the correspondent timestamps.

One of the most important performance metrics for packet audio is the percentage of packets lost at the destination host. Such loss results from either the late arrival of a
packet (i.e., the arrival of a packet after its scheduled playout point) or a premature arrival of a packet. In the latter case, packet loss derives from the limitation on the finite size of the receiver's playout buffer. In order to manage adequately the receiver's playout buffer, in our prototype implementation a set of communicating processes implements at the application level of the receiving host the buffering/playout policy described in Section 3. In summary, a circular buffering scheme has been adopted according to which only ``on-time'' audio packets are first queued in the empty locations of the buffer and then periodically fetched and sent to the audio device for being played out.

The performance of the prototype implementation of the playout delay control mechanism presented in this paper has been evaluated using an IP-based internetworked infrastructure connecting two SPARCstation 5 workstations running the SunOS 4.3 operating system and situated, respectively, at the Laboratory of Computer Science of Cesena (a remote site of the University of Bologna), and at the C.E.R.N. Institute in Geneva. Each one of the used workstations was locally connected to a 10-Mbit/sec Ethernet LAN. In order to perform measurements while avoiding the issue of sender's and receiver's clock synchronization, the audio packets were sent using the following communication scenario. The source host was the same as the destination host and were located in the same workstation situated at Laboratory of Computer Science of Cesena. Instead, the workstation situated at the C.E.R.N. Institute in Geneva operated as an intermediate host. In essence, each generated audio packet was sent from the source host to the intermediate host. Such an intermediate host, upon the receipt of a packet, simply echoed that packet back to the destination host. This policy has allowed us to take experimental measurements of end-to-end packet delays not affected from the clock synchronization problem, since in the above mentioned scenario the end-to-end delays coincide with round trip delays. In Figure 8 the routes taken by the packets sent over the above mentioned connection are shown that were obtained with the traceroute routine.

Prior to illustrating the measurements that were taken during the experiments, it is also interesting to notice that at the time the experiments were carried out (September-October 1997) the experimental testbed that was used had almost all interconnecting links with bandwidth ranging from a few to several megabits per second. Indeed, the unique bottleneck link of the above mentioned communication scenario was the regional link interconnecting the router situated at Laboratory for
Computer Science of Cesena with the router of the Network Center of the University of Bologna (512 Kbit/sec). It is also interesting to note that the router of the Laboratory for Computer Science of Cesena is typically under a heavy load, since it operates as an interconnecting router for a number of remote university sites of the region.

```bash
poseidon.csr.unibo.it (137.204.72.49):~$ traceroute cms1.cern.ch
1 romr01.csr.unibo.it (137.204.72.254)
 2 137.204.27.254 (137.204.27.254)
 3 137.204.1.20 (137.204.1.20)
 4 192.12.77.29 (192.12.77.29)
 5 nisc1.1bo.infn.it (131.154.151.253)
 6 cnaf-gw1.infn.it (131.154.1.9)
 7 cnaf-int-gw.infn.it (131.154.1.2)
 8 ten-34.garr.net (192.135.34.22)
 9 garr.IT.ten-34.net (193.203.226.25)
10 it.CH-1.ten-34.net (193.203.226.9)
11 ch-1.CH-2.ten-34.net (193.203.226.30)
12 cgate1.cern.ch (192.65.185.1)
13 r513-c-rci47-5-gb0.cern.ch (128.141.211.20)
14 cms1.cern.ch (137.138.129.123)
```

**Figure 6.5:** Route between the Laboratory of C.S. of Cesena and the C.E.R.N. Institute.

## 6.4.2 Experimental Results and data interpretation

Several experiments (about 30) of unicast audio conversations were conducted using the software prototype implementation of our mechanism during daytime (from 7 a.m to 8 p.m) in different days during the period September-October 1997. Each experiment carried out between Cesena and Geneva consisted in transmitting about 15,000 audio packets. Those packets were generated using both the G.729 and the G.723.1 based codecs from prerecorded 10 minutes long audio files. Two are the most important metrics that influence the users' perception of audio data: 1) the percentage of audio packets that arrive too late at the destination to be played out (and so can be effectively considered lost), and 2) the amount of total delay that an audio packet has to experience before it is played out at the destination. Hence, in order to measure the performance of our mechanism, during each experiment we measured: i) the percentage of lost packets, ii) the end-to-end transmission delay experienced by audio packets during transmission, and finally iii) the additional buffering delay imposed by our mechanism. From a statistical analysis of the provided data, it is possible to measure an average value of the
round trip transmission delay almost equal to 119 msec, with a maximum delay spike of 627 msec at the beginning of the transmission, and only another spike exceeding 550 msec (after 1/3 of the period of the audio transmission). It is also worth noticing that the total number of audio packets that were completely lost by the network (i.e. never arrived packets) was rather low (about 40). This is probably due to the large bandwidth that is provided by the communication links that interconnect the end hosts of our experiment.

In Figure 6.6, the evolution of the playout delay, i.e. the total amount of delay that each packet n has to experience before it is played out, is reported (as function of n) that was obtained with our playout delay control mechanism, where the synchronization activity was repeated with the frequency of 1 second. Yet again, on the x-axis of Figure 6.6, the value of the timestamps are reported re-numbered in order to eliminate timestamp gaps. Instead, on the y-axis of Figure 6.6 the values of both the (round trip) transmission delay and the playout delay are reported for each packet expressed in milliseconds. In particular, the values of the transmission delays are shown with grey lines (and denoted as ``RTT'' in the caption inside the figure), while the values of the playout delays are plotted with black lines (and denoted as ``D'' in the caption inside the Figure). More precisely, note that when a grey line exceeds the corresponding black line, this entails that this packet has arrived too late with respect to the playout deadline computed by our playout control mechanism and, consequently, is discarded. On the contrary, if the black line encapsulates the grey line, this means that the corresponding audio packet has arrived in time to be played out at the receiver. From a statistical analysis of the data plotted in Figure 6.6, several interesting considerations may be derived. First, it is important to notice that our playout control mechanism keeps the percentage of lost packets below the threshold of 5%. Furthermore, the average playout delay was calculated as equal to 238 msec. This playout delay value may be considered tolerable for audio conversations and guarantees a good degree of interactivity. In addition, it is worth noticing from Figure 6.6 that a number of playout delay spikes (approx. 15) were produced that exceeded the value of 450 msec. Nevertheless, it is also worth mentioning that our playout control mechanism was used during the experiment with the value of the smoothing factor k equal to 1, that is the mechanism for smoothing out the playout delay spikes was kept deactivated. Finally, in order to fully assess the performance of the proposed playout control mechanism, the percentage of lost packets obtained with our mechanism (i.e. 5%) has
been contrasted with the percentage of audio packets that would be lost if a constant playout delay of 150 msec was used throughout all the performed experiment. The percentage of lost audio packets (due to a playout delay of 150 msec) was measured as equal to 23%.

**Figure 6.6:** Evolution of the playout delay (start time: 02.00 pm 10/9/97).

**Table 6.2:** Experimental results: average playout delay and packet loss.

<table>
<thead>
<tr>
<th>Experiments</th>
<th>Start Time</th>
<th>Playout Delay</th>
<th>Packet Loss</th>
<th>Spike Management</th>
</tr>
</thead>
<tbody>
<tr>
<td># 1</td>
<td>08:20am 10/15/97(Th)</td>
<td>188 msec</td>
<td>7%</td>
<td>yes (k=4/5)</td>
</tr>
<tr>
<td>#2</td>
<td>02:00pm 10/9/97(Fr)</td>
<td>238 msec</td>
<td>5%</td>
<td>no (k=1)</td>
</tr>
<tr>
<td>#3</td>
<td>11:00am 10/4/97(Su)</td>
<td>202 msec</td>
<td>6%</td>
<td>yes (k=4/5)</td>
</tr>
<tr>
<td>#4</td>
<td>7:40pm 9/21/97(Mo)</td>
<td>229 msec</td>
<td>5%</td>
<td>no (k=1)</td>
</tr>
<tr>
<td>#5</td>
<td>04:15pm 9/16/97(We)</td>
<td>207 msec</td>
<td>5%</td>
<td>no (k=1)</td>
</tr>
</tbody>
</table>

We conclude this section by reporting in Table 6.2 the values of the average playout delay and the packet loss percentage of only 5 out of the 30 experiments that were carried out. It is worth mentioning that, besides the results provided in Table 6.2, also in all the other 25 cited experiments both an acceptable value of the average playout delay (ranging in the interval 180-250 msec) and a tolerable loss percentage of up to 6-7% were experienced, and only rarely playout delay spikes exceeding 600/700 msec
were imposed by our mechanism. On the contrary, in all those other experiments that were conducted using a constant playout delay (typically obtained by increasing of a 10% the value of the average transmission delay) an amount of lost audio packets was experienced ranging from about 15% to almost 40%.

6.4.3 Comparative Simulation Results

This section is devoted to the comparison of our mechanism with another playout delay control mechanism recently proposed [118]. A new adaptive (history-based) delay adjustment algorithm was proposed that tracks the network delays of received audio packets and efficiently maintains delay percentile information [118]. That information, together with an appropriate delay spike detection algorithm, is used to dynamically adjust talkspurt playout delays. In essence, the main idea behind that algorithm is to collect statistics on packets already arrived and then to use them to calculate the playout delay. Instead of using some variation of the stochastic gradient algorithm in order to estimate the playout delay, each packet's delay is recorded and the distribution of packet delays is updated with each new arrival. When a new talkspurt starts, the algorithm proposed in [118] calculates a given percentile point (say q) for the last arrived w packets, and uses it as the playout delay for the new talkspurt. In addition, the algorithm accommodates delay spikes in the following manner. Upon the detection of a delay spike, the algorithm stops collecting packet delays, and follows the spike (until the detection of the spike's end) by using as playout delay the delay experienced by the packet that commenced the spike. Upon detecting the end of the delay spike, the algorithm resumes its normal operation mode. The authors of [118] have experimentally shown that their algorithm outperforms other existing delay adjustment algorithms over a number of measured audio delay traces, and performs close to a theoretical optimum over a range of parameters of interest. Thus, in order to assess the performance of the mechanism proposed in this paper, we carried out a simulation experiment that compares our playout delay adjustment algorithm with the history-based mechanism proposed in [118]. As already mentioned, the most important metric that influence the users' perception of audio data is represented by the average playout delay vs. the packet loss. Hence, the two algorithms were compared with respect to these values. To this aim, a simulator was designed and developed that reads in the transmission delay of each packet from a given trace, detects if it has arrived
before the playout time that is computed by each of the two algorithms, and executes the algorithm. The simulator is also able to calculate the average playout delay and the packet loss for a given trace (for each of the two algorithms). Thus, we run the simulator several times in order to simulate the use of the history-based algorithm over the measured audio delay trace reported in Figure 9. The simulator was used by varying (at each run) the percentile point q, in order to reach the following fixed values of loss percentage (approx. 3%, 4%, 5%, 7%, 10%, 12%, 15%, 20%), and then to measure the correspondent average playout delays. The values of the percentile point q that were used to keep the packet loss percentage below the values reported above were the following: .995, .99, .985, .98, .97, .96, .94, .92. Subsequently, we run repeatedly the simulator to simulate our algorithm over the same measured audio delay trace depicted in the Figure 9. The purpose of those simulations was to identify that set of cases where our algorithm reaches approximately the same values of the loss percentage that were obtained with the history-based algorithm, and then to measure the correspondent average playout delays. This allow us to compare the performance of the two algorithms under identical network conditions. To this end, we simulated our algorithm using the buffer size as the control parameter to be varied to achieve different loss percentages, as was done in [118, 134]. Such a variation of the buffer size was achieved by tuning a sensitivity factor in the synchronization formula used to obtain the synchronization values to be installed at the receiver. In essence, instead of using the "regular" synchronization formula (namely $\Delta = t + \text{RTT} \), the following formula was used to calculate the playout delay: $\Delta = t + (p \times \text{RTT})$; where p is a non negative constant. Hence, in order for our algorithm to obtain approximately the same packet loss percentages obtained with the history-based algorithm the following values of p were approx. needed: 1.5, 1.3, 1, 4/5, 3/4, 2/3, 1/2, 2/5. The average playout delays obtained using this simulation technique were then compared with the average playout delays obtained with the simulation of the history-based algorithm, at a parity of packet loss percentage.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>3%</th>
<th>4%</th>
<th>5%</th>
<th>7%</th>
<th>10%</th>
<th>12%</th>
<th>15%</th>
<th>20%</th>
</tr>
</thead>
<tbody>
<tr>
<td>history-based</td>
<td>332</td>
<td>249</td>
<td>215</td>
<td>201</td>
<td>178</td>
<td>172</td>
<td>161</td>
<td>152</td>
</tr>
<tr>
<td>our-algorithm I</td>
<td>298</td>
<td>260</td>
<td>202</td>
<td>194</td>
<td>181</td>
<td>170</td>
<td>166</td>
<td>157</td>
</tr>
<tr>
<td>our-algorithm II</td>
<td>352</td>
<td>301</td>
<td>237</td>
<td>223</td>
<td>210</td>
<td>193</td>
<td>187</td>
<td>172</td>
</tr>
</tbody>
</table>

Table 6.3: Simulation results: average playout delays (msec).
It is worth mentioning that our algorithm was used twice for simulation. First, it was simulated with the playout delay spike smoothing mechanism deactivated (i.e. \( k = 1 \)), then the experiment was repeated with the playout delay spike smoothing mechanism activated (i.e. \( k = 4/5 \)). In conclusion, in Table 6.3 the values of the average playout delays are reported, that were obtained with the simulation of respectively: i) the history-based algorithm (first row in Table 6.3), our algorithm with the playout delay spike smoothing mechanism activated (second row in Table 6.3), iii) our algorithm with the playout delay smoothing mechanism deactivated (third row in Table 6.3). From the analysis of Table 6.3, it is possible to deduce that the history-based algorithm outperforms our algorithm with the playout delay spike smoothing mechanism deactivated. Instead, based on the consideration that audio of acceptable quality may be obtained only if lower delays are achieved while the loss percentage does not exceed the value of 10%, our algorithm (with the delay spike smoothing mechanism activated) shows better performance w.r.t. to the history-based algorithm as long as the loss percentage is kept below the value of 10%. In order to better illustrate the results of our comparison, in Figure 11 the average playout delay is plotted as a function of the loss percentages for each analyzed algorithm. The plot of the playout delay has been obtained by running the simulator over all the 30 experimental traces, and then averaging the results. In order to provide the reader with an understanding of the effect that various delay and loss rates (as well as buffer dynamics) have on the quality of the perceived audio, we have reported in Figure 11 an approximate and intuitive representation of three different ranges for the quality of the perceived audio. The three following audio quality ranges have been adopted from [6]: good, for delays of less than 200/250 msec and low loss rate, potentially useful, for delays of about 300-350 msec and higher loss rates, and poor, for delays larger than 350 msec and very high loss rates. As seen from the Figure, our algorithm (with the playout delay smoothing mechanism activated) shows better average performance w.r.t. the history-based algorithm in both the good and the potentially useful audio quality regions. To conclude this comparison, it is worth noticing that the synchronization policy embedded in our mechanism imposes little computational overhead (both at the source and the destination hosts) w.r.t. the history-based algorithm, where delay statistics have to be collected, and percentile points of the delay distribution have to be calculated on-the-fly.
Chapter 6. The Voice-based Communication Case Study

6.5 Are the silences sufficient?

The need of silent intervals for allowing a playout delay control mechanism to adjust to the fluctuating network conditions renders the scheme proposed in [30] particularly relevant for voice-based applications with intervening silence periods between subsequent talkspurts. In order to assess the efficacy of such one mechanism, an accurate modelling of the talkspurt/silence characteristics of conversational speech is thus mandatory for understanding whether sufficient (and sufficiently long) silent intervals occur in typical human conversations that may permit the periodical activity of dynamically setting the playout delay.

To this aim, the modified eight-state Brady's model of conversational speech introduced in [27] was adopted that is able to describe the main on-off characteristics of human conversations.

The motivations behind the choice of this particular model (represented in Fig. 6.8) are its accuracy and ease of implementation for carrying out simulations and analyses. Fig. 6.8 is divided into quadrants with each quadrant representing a different state for parties A and B engaged in a conversation. In particular, such one model has
been shown able to reproduce all the three following different types of silences occurring in human speech: i) listening pauses, which occur when a party is silent and listening to the other party; ii) long speaking pauses, which occur between phrases or sentences while a party is speaking; iii) short speaking pauses, which occur between words or syllables while a party is speaking.

**Figure 6.8.** Modified 8-state Brady model [27].

In Table 6.4, the values of all the state transition parameters of the Markovian Brady's model depicted in Fig. 3, as calculated in [27], are reported.

<table>
<thead>
<tr>
<th>State Transition Parameters</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>α'1,4</td>
<td>0.83305</td>
</tr>
<tr>
<td>α'6,5</td>
<td>5.4890</td>
</tr>
<tr>
<td>α1,7</td>
<td>2.1572</td>
</tr>
<tr>
<td>α2,1</td>
<td>2.3245</td>
</tr>
<tr>
<td>α7,1</td>
<td>27.62</td>
</tr>
<tr>
<td>α4,1</td>
<td>2.2222</td>
</tr>
<tr>
<td>α5,1</td>
<td>1.0438*</td>
</tr>
<tr>
<td>α1,2</td>
<td>0.27853</td>
</tr>
</tbody>
</table>

* During the first 200 ms after transition to States 4 or 5, α4,1, α5,6, α5,1, and α4,6 are set to 0.

**Table 6.4:** State transition parameters for the modified Brady's model [27].

Based on this model, we have developed a set of simulative experiments in order to assess respectively: the overall quantity, the frequency, and the duration of silence...
intervals within human conversational speech. The first question to be answered in order to understand whether the designed playout delay control mechanism could be profitably applied to human voice was: “Is there a sufficient total amount of silent intervals in human conversational speech to permit an adequate application of the designed playout delay control mechanism?” We obtained a first positive response to the above-mentioned question. In fact, based on the use of the model we observed that the total quantity of silent intervals within a simulated packetized conversation amounts to about 63-66%, depending on the voice packetization interval that is typically chosen in the range of [10-30] milliseconds. This result is summarized in Fig. 6.9 where the total number of silent intervals (with relative duration) is shown, as obtained in a simulated one-hour-long two-party conversation. As seen from the figure, the smaller the packet size (i.e. 10 milliseconds), the larger the number of silent intervals (i.e. 5075) we obtain.

![Figure 6.9: Total amount of silence periods.](image)

Then, since an important parameter that influences the efficacy of our control mechanism is the frequency of the intervening silence periods, we addressed the next following question: “How much frequent are those silent intervals within a human conversation?” Needless to say, the more frequent those silent intervals are, the more likely it is that our mechanism will be successful in dynamically adjusting the playout delay. To this aim, we used the modified Brady's model to understand how many different talkspurts (and consequently silent intervals) are expected in a simulated packetized conversation. From this simulative experiment, we observed the following important result: the smaller the chosen packet voice size, the more likely it is that our mechanism will have the possibility of dynamically setting the playout delay. In fact,
the total number of silence periods increase and the average talkspurt length decreases (to about 244 millisecond) with small-sized packets. Instead, the larger the packet size, the larger the average duration of the talkspurt (i.e. 451 milliseconds). The main result concerning the quantity and the duration of the talkspurts is depicted in Fig. 6.10 where the percentage of talkspurts with length smaller than a fixed amount are reported.

![Figure 6.10: Percentage of talkspurts with duration smaller than X.](image)

The final question we asked prior to the development of our Internet audio mechanism was: “Given that, according to the policy adopted by our control mechanism, the duration of intervening silence periods is artificially reduced when an improvement of the audio packet transmission delays is experienced, how long are in average those silent intervals?” To this aim, an important result obtained from the simulation experiments we carried out concerns the duration of the intervening silence periods in a human conversation. The average silence length of the silence periods obtained in a simulated two-party one-hour-long conversation was measured as ranging in the interval [465-770] milliseconds, yet again depending on the packetization interval. In
particular, the larger the packet size (i.e. 30 milliseconds), the larger the average silence duration (i.e. 770 milliseconds). The main results regarding the silence interval duration are summarized in Fig. 611 where the probability distribution of the silence interval duration is reported.

### 6.6 Concluding Remarks

An adaptive mechanism for the control of the playout delay of audio packets over the Internet has been proposed. This mechanism is suitable for dynamically adjusting the talkspurt playout delays for unicast, voice-based communications where conversational audio with silence periods between subsequent talkspurts is transmitted. We commenced the design and the experimental evaluation of our playout delay control mechanism during the Summer of 1997, and completed them in October 1997. During that period, several experiments were carried out that showed that our design was successful in maintaining satisfiable values of the average playout delay, while minimizing the number of audio packets that were lost during an audio transmission performed over a interconnected (IP based) link. The designed mechanism assumes neither the existence of an external algorithm for maintaining an accurate synchronization at both the sending and the receiving sites of the audio connection, nor a Gaussian distribution for the end-to-end transmission delays experienced by the audio packets. In addition, our mechanism provides: i) an internal and accurate algorithm that maintains tight time synchronization between the sending and the receiving hosts, ii) a method for adaptively estimating the audio packet playout time (on a per-talkspurt basis) with an associated minimal computational overhead, iii) an exact and simple technique for dimensioning the playout buffer depending on the traffic conditions of the underlying network.

Moreover, we have reported on the use of an eight-state Markov model for voice activity in conversational speech which has been exploited in order to assess the performance of an adaptive playout delay control mechanism recently proposed [30,26]. One of the most critical characteristic of this audio mechanism is its reliance on the existence of silent intervals (of sufficiently long duration) for effecting the dynamical setting of the playout delay. Based on the use of the model, we have conducted several simulation experiments that show that a sufficient number of silence
periods occur during a human conversation (about 66%). In addition, those silent intervals result to be both sufficiently long and frequent so as to permit an adequate application of the proposed audio mechanism. Thus, the proposed mechanism appears to work well.
The case studies previously discussed confirm that it is possible to adopt very effective application level policies and architectures for improving the QoS, as perceived by the users, without relying on network level guarantees. That approach can be implemented by using either a general-purpose middleware platform, such as CORBA and DCOM, or a QoS-supporting middleware platform such as RT-CORBA, TAO, QuO and Agilos. However, assuming that distributed applications be constructed by composing reusable and interoperable software components (Building Blocks, BBs), the question arises as to which properties those components must posses in order to allow a designer to build, via composition, flexible, adaptive, and QoS-aware applications.

In order to answer this question, an adequate middleware architecture, capable of supporting arbitrary application level policies for QoS purposes, can be provided.

Each of the above mentioned commercial QoS-supporting middleware platforms provides a useful set of services that allow the programmer to control scheduling, priority and event notification, that are the base of the QoS management. However, RT-CORBA and TAO do not provide specific primitives for supporting QoS adaptations. In contrast, Agilos is designed for supporting adaptation mechanisms, but it does not provide facilities for constructing applications by composing QoS-adaptive building blocks. The QuO project, instead, appears as a very complete middleware platform. The most important features of QuO, such as QoS contracts, delegates, and the Quality Description Language (QDL) provide one with a very flexible and reusable interface to manage QoS of building blocks. However, QuO appears to introduce complexity in composition of QoS-adaptive building blocks, because it adds three working entities to the method call, and define the behavior of the object by means of “nested region”, that are not always simply manageable.

What is required is an agile and simply usable middleware platform capable of supporting deployment of QoS-adaptive policies by allowing composition of QoS-
aware building blocks. One such middleware architecture is proposed and discussed in the following.

7.1. QoS-Adaptive Middleware

Object-based middleware using ORB services, provides a promising platform for the development of QoS adaptive middleware architectures, due to its ability to isolate the problems in different (potentially distributed) objects that interact via location-transparent method invocations. The most relevant functions of ORB technology, such as interface definition, location and possible activation of remote objects, and communication between clients and objects solves a large set of basic problems.

Thus, an analysis identifying an abstract middleware model able to provide support for application-oriented QoS starts by considering object-oriented middleware. Hence, the entities previously called building blocks (BBs) will be treated as objects composed of other objects. We assume that objects can be created, identified, deleted, managed and can communicate with each other.

In order to define a suitable middleware model able to support the proposed policies/architectures, we need to summarize the characteristics the applications need to satisfy to be able to adapt to fluctuation in resource availability. Assuming that the applications are constructed by BBs, these BBs need to perform some typical processing, such as:

- Monitoring low-level parameters strictly dependent on operating system (CPU usage, disk usage, cache hit ratio, network adapter usage, memory usage). These represent a sort of low-level input for the BB;
- Controlling low level operating system parameters (e.g. process priority, sampling period of the audio card) that represent a sort of visible status of the BB and influence the BB behavior;
- Monitoring parameters strictly dependent on the application's operations and on the protocols they use (for instance, data rate provided through given HTTP connections) that represent a sort of high-level input for the BB;
- Controlling parameters, strictly dependent on the application, (e.g. coding scheme of the audio data) that represent a portion of the visible status of the BB;
- Performing I/O operations;
• Reacting to changes in the QoS parameters by modifying their behavior (for instance, in voice communication case studies, when RTT increases rapidly, the receiver switches to the 'spike mode' state) and their visible status, or by modifying several low-level system parameters.

• Reacting to expiration of timeouts, by modifying their behavior and/or their visible status.

• Changing their behavior, depending on the user's actions, by modifying their visible status.

These apparently simple operations that the BBs need to perform have very important consequences on the characteristics of the BBs themselves. The first consequence concerns interoperability. From the monitoring and control of low-level parameters, strictly dependent on operating systems, it is evident that the BBs need a standard and portable interface with the operating system. Moreover, monitoring and reacting may need to be executed periodically, in order to update the parameters that describe the behavior of the BBs. Hence, BB interfaces need to provide methods that the applications and other BBs can use to control the updating frequency of the BB QoS parameters.

7.2. Synchronous and Asynchronous Calls

The most important consequences concern communications between BBs; as we will see, both synchronous and asynchronous calls must to be supported by the BBs. In fact, monitoring and control of parameters is strictly dependent on the applications, and reacting to their changes implies that the BBs dynamically and bi-directionally exchange information about their states. A synchronous request-response mechanism, such as an RPC mechanism, is required for inter-BB communication.

Moreover, and more importantly, reacting is an operation that may need to be executed on an event-based policy. That case occurs, for instance, if a given BB recognizes the changing of some QoS parameters and needs to warn the application or other BBs, in order that they can execute a given operation. The problem is the mechanism a BB may use to call the application asynchronously. The application cannot use the request-response mechanism because it does not know the instant a given event occurs. Moreover, to improve reusability, the BB cannot be devoted to a
given application, so it cannot implement the application's operation by itself (i.e., it cannot call a specific application function). Hence, the BB must be configured accordingly with the application's policy, without introducing modifications to its implementation. Thus, the middleware needs to provide the application (or the other BBs) with a method for configuring the BB (at least at compile time) by indicating the application function (or a given BB object) that the BB must call, when a given event occurs, to warn the application/BBs. That configuration method allows the BB to perform an **Asynchronous Call Back**, like the asynchronous call back function provided by DCOM (see section 4.4.2). This feature represents a method for implementing unidirectional and asynchronous function call. The following Figure 7.1 shows the operations performed to configure the BB (at compile time) and to call the function (at runtime).

![Figure 7.1: Asynchronous Call Back Function Mechanism.](image)

The application implements a given function (object) that must react to the change of a given parameter, and creates a BB that monitors that parameter. At runtime the applications give to the BB a reference to the function to be called when the parameter changes (indicated by the array labeled 1, in Fig. 7.1), and enable the BB. When the parameter changes (event 2 in Fig. 7.1) the component calls the function (object) (indicated by the array labeled 3, in Fig. 7.1). Using that method, the BB does not require modification to its implementations.
7.3. Architecture

Based on the discussion above, the middleware can be structured as follows. As depicted in the Figure 7.2, each host represents a domain in which two types of entities work:

- several QoS Adaptive Building Blocks, and
- one Parameter Monitor.

![Figure 7.2: The complete Architecture.](image)

The QoS Adaptive Building Block (QABB) represents the unit of QoS management. Assuming that a given application is constructed out of several BBs, each of these BBs originates a QABB that consists of the BB itself and of a QoS-Oriented Interface (QOI), that contains the BB and provides the visible QoS status of the BB.

The QABB extends the functionality of its BB 1) by specifying the status and behavior of its BB, 2) by providing the other QABBS with methods for changing the visible status of its BB, 3) by implementing several additional QoS-oriented aspects of the BB’s behavior, and 4) by providing methods and structures to configure its BB, in order to make it reusable. The QABB inherits the interfaces of the BB, in order to
continue provision of the BB's services. Thus, QABB maintains the entry point of its BB (for instance, the transport level entry point (IP address, port number) of a TCP-based service) and adds its own interface. The aim is to provide a separate channel/interface to manage QoS of the different BBs, while maintaining the old BB interfaces that permit standard communications between the BBs. Obviously, a QABB can be designed ex-novo, by creating a QoS interface QOI containing an object that performs the traditional data processing.

The Parameters Monitor (only one instance for each domain) is a sort of local scheduler that i) performs periodical or event-based monitoring of the parameters of each Building Block, and ii) starts the operations that each QABB executes as a reaction to a change in some QoS parameters.

7.3.1. The Building Block

Each QABB, from the point of view of QoS is described by a set of visible parameters that describe the QABB’s QoS status. There is a function for modifying the value of each visible parameter. In particular each of these visible parameters maintains several references to objects each of which implements a function. The first two references indicate two internal objects (objects of the considered QABB), while the other references indicate objects of other QABB.

The first function (PeriodicUpdate) is implemented by the QABB, and is executed when the Parameters Monitor needs to update the parameter value. An auxiliary parameter (FrequencyUpdate) keeps the information about the number of update operations in the time unit the Parameters Monitor must perform. This function may also change some internal or visible parameters of the QABB before returning the new parameter value to the caller.

The second function (SynchronousUpdate) is implemented by the QABB, and is executed when an external QABB (or the QABB itself, in order to implement I/O-driven policies) needs to update the parameter value. Several arguments may be passed as formal parameters of the calls. This function may, for instance, change some internal parameters before returning the new parameter value to the caller, depending on the caller identity.

The third entity (AsynchronousUpdate) is a virtual function, of which the QABB may keep several instances (a list of functions) each of which is implemented by a
different QABB through the CallBack mechanism, in order to obtain an asynchronous updating mechanism, which is practically the basis for event-driven QoS management. This function, if enabled, allows an external QABB to execute the function when the parameter changes its value. Obviously, the QABB itself may provide a default implementation of this AsynchronousUpdate function that other QABBs may use (by specifying suitable arguments). Moreover, the QABB may register itself in order to be notified when one of its parameters changes value. The functions may be disabled if necessary. For instance, the priority of a BB may be considered a constant, if its function references are disabled, so that priority is not modifiable.

7.4. QoS Parameters Upgrading Mechanism

As depicted in Figure 7.2, each system component (e.g., scheduler, memory manager, I/O device, clock), with a role in the QoS management needs to be included in a QABB, in order to provide the QoS-oriented interface. The Parameters Monitor is responsible for the periodical execution of the PeriodicUpdate functions of each QABB.

Each QoS parameter of a given QABB can be modified by a periodic function (PeriodicUpdate) or by a call to an internal function in response to an occurred event (I/O event for instance), or by an explicit call (SynchronousUpdate) performed by the QABB itself, or by an explicit call (SynchronousUpdate) performed by a different QABB, or, finally, it can be modified by calling an AsynchronousUpdate.

In the instant in which a given parameter of a given QABB changes its value, the Parameter Monitor executes each existing occurrence of the AsynchronousUpdate defined for that parameter by a QABB, in order to propagate the effect of the parameters modification to the QABBs.

By means of this upgrading mechanism, the changing of visible parameters falls into a sequence of parameter upgrading that modifies the behavior of the entire system. It is worth point out that:

- The application is constructed by composing several QABBs. Only the QABBs that directly interact with the applications (edge QABBs) are visible, from the point of view of the application, whereas other QABBs (the core QABBs) are hidden. Thus,
from the point of view of the application, only the edge QABBs are responsible for QoS.

- The Parameter Monitor constitutes the core of the so-called monitoring service. It effectively acts as a scheduler of monitoring procedures. Instead, the set of internal functions that manage I/O for all the QABBs, the set of *PeriodicUpdate*, *SynchronousUpdate*, and *AsynchronousUpdate* functions of the core QABBs, and the set of Periodic functions of the edge QABBs, constitute the operator of that monitoring service.

- The adjustment service, as implemented and/or used by the application, consists of the set of *SynchronousUpdate* and *AsynchronousUpdate* functions of the edge QABBs.

The proposed architecture is sufficient to implement QoS-aware building blocks, and capable of supporting applications that implement QoS adaptive policies. These policies are typically best-effort policies because they do not require guarantees from the subsystems. They simply modify application behavior by trying to share resources among all the other applications. Due to this approach, the middleware structure needed to support these policies does not appear to be so complex. It can be implemented on top of both CORBA and DCOM middleware, because they provide a sufficient set of services. Based on that approach, a middleware prototype has been implemented [32] as an extension of the DCOM commercial platform, on a Windows NT 4.0 operating system. That intermediate-level software has been used to implement the synchronization mechanism, devoted to voice-based communication, described in chapter 6. The visible QoS parameter that describes the main QABB of that synchronization mechanism is the latency time of communication between the two end systems involved.
Chapter 8

Concluding Remarks

In this thesis we have discussed issues of design of adaptive Quality of Service (QoS) policies, that can be implemented at the middleware level, in order to support effectively geographically-distributed real-time applications over best-effort IP-based networks. In particular, we have concentrated our efforts on two case studies, two applications that present very different QoS requirements, in order to highlight the ability to provide effective QoS at the middleware level: a replicated Web service, and a voice-based communication. Two policies have been proposed, implemented and evaluated, and the results confirm the validity of our approach, that extends the notion of QoS to include high level application-dependent QoS parameters, such as availability, timeliness, and fault tolerance.

In order to support the construction of adaptive applications that follow that approach, we have proposed an abstract middleware architecture. This architecture is able to provide support in constructing and composing QoS-adaptive reusable building blocks, in order to construct applications by the composition of that blocks.

That architecture is characterized by limited complexity and can be implemented on commercial middleware platform such as CORBA and DCOM. Other QoS-oriented architectures, such as QuO and Agilos appear to be suitable for supporting QoS-aware applications, but does not provide features for supporting at the same time building block composition. Thus, the applications need to implement the mechanisms for controlling (from the point of view of the QoS) the interactions between building blocks. These mechanisms, instead, are provided by our approach.

A number of interesting issues have arises in the course of our research. In particular, as far as adaptive policies are concerned, the C2LD downloading approach appears to be very promising. We wish to examine strategies involving the pre-fetching of Web resources from replica servers. Secondly, we wish to extend our mechanism to deal
with dynamic Web services and previously described strategies for implementing replicated Web servers. Finally, we wish to investigate policies for maintaining data consistency between geographically distributed replica servers.

Instead, from the point of view of the middleware architecture, the main open problem concerns the provision of a programming language allowing the application developer to express and describe the composition rules of the application components, their QoS properties, interferences between blocks due to the composition, and overall QoS application requirements, in order to effectively support the composition of the applications using basic software components.
References


[82] "Cisco Local Director”, CISCO System Inc., White paper, 1996


[117] I. Kouvelas, V. Hardman, "Overcoming Workstation Scheduling Problems in a Real-Time Audio Tool," in Proc. of *Usenix Winter Conference*, (Anaheim,


[141] ITU-T Recommendation G.723.1, "Dual rate Speech Coder for Multimedia Communications Transmitting at 5.3/6.3-kb/s", 1996