7

System Support for Distributed Multimedia Applications with Guaranteed Quality of Service

Nguyen Hong Quang, Guy Bernard and Djamel Belaid
Institut National des Télécommunications
9 rue Charles Fourier, 91011 EVRY Cedex, France.
Telephone: +33-1-60 76 45 67. Fax: +33-1-60 76 47 80.
email: {quang|bernard|belaid}@etna.int-evry.fr

Abstract

We consider the problem of executing distributed multimedia applications with Quality of Service (QoS) requirements. Such applications involve several resources (CPU, network, file system, memory). In order to fulfill the QoS requirements of the applications, every resource manager must provide temporal guarantees, but this is not enough. An overall QoS management scheme must be set up for admission control and execution monitoring. Based on an analysis of application memory access time, we identify the key points where temporal guarantees must be provided. We present the tasks that must be achieved for overall QoS management and propose an implementation based on a QoS manager replicated on each site and modified resource managers. Applications interact with one QoS manager only through a well-defined interface. As an illustration, we present the design of a File Server modified for providing guaranteed access time and its interactions with the QoS manager in the context of a distributed microkernel-based operating system.

Keywords

Quality of Service, Multimedia Applications, Distributed Systems, Microkernel.

1 INTRODUCTION

In the recent past years, dramatic increase in the speed of wide-area networks and processors have open the door to the execution of distributed applications involving large data flows between remote sites. Distributed multimedia applications, handling continuous media such as digital audio and video, belong to this category.

However, the amount of data that is handled is not the sole characteristics of multimedia applications. At least as important is the requirement of Quality of Service (QoS), because data items of continuous media must be displayed to the user in a timely fashion. Execution of distributed multimedia applications involves several components from data source to data display, such as file systems, CPU schedulers and networks. A "best effort"

R. Puigjaner (ed.), *High Performance Networking*

behavior of each of these components is not enough to ensure the QoS expected by users – what is needed is a "guaranteed" QoS, requiring real-time semantics for every component, even if this is not always easy to obtain (Danthine *et al.*, 1994). Whereas CPU scheduling and network resource reservation have been the object of numerous research works over the recent past years (see for instance (Govindan *et al.*, 1991), (Coulson *et al.*, 1993(a)), (Mercer *et al.*, 1994), (Nicolaou, 1990), (Campbell *et al.*, 1992)), memory management with QoS requirements did not receive the same attention.

The goal of this paper is to present a generic QoS management model for supporting distributed multimedia applications and to show how this model can be applied for achieving guaranteed QoS for the management of a specific resource, namely a file server, in the context of a microkernel-based distributed operating system. Secondary memory management is just an illustration of our model; we believe that this model could be applied for handling other resources, such as CPU or network, as well.

The paper is structured as follows. In Section 2, we present an analysis of memory access time for general applications and discuss the possibility to achieve time guarantees that are required by distributed multimedia applications. In Section 3 we present a generic management model of QoS and discuss the different approaches for implementing the QoS management functions. In Section 4 we show how one resource manager (the Object Manager) may be modified in order to fit the QoS management model. Section 5 presents related work, and finally Section 6 provides some conclusions and perspectives.

2 MEMORY ACCESS TIME ANALYSIS

In this section we present our system model, we derive general expressions for memory access times, and we discuss the possibility to achieve time guarantees for memory access.

2.1 System Model

Our system model is based on a distributed microkernel. Such a system consists of a collection of machines (*sites*) connected together through some network (see Figure 1). Each site has a copy of a common *microkernel*. The microkernel manages, at the lowest level, the local physical resources of a site. It provides just some minimal basic services as low-level interrupt, trap and exception management, some low-level process management and scheduling, some virtual memory management, and an inter-process communication mechanism. Several *system servers* built on top of the microkernel cooperate together in order to export operating system services to application processes. The cooperation between system servers is achieved exclusively by exchange of *messages*. For instance, a virtual memory manager of a site may send a message to a remote file server (via local and remote network managers) requesting the reading of a file data block.

We have chosen this system model because it is a promising distributed computing system model (Tanenbaum, 1995). One major advantage of this model is its high modularity and flexibility. The system programmer can easily change (*e.g.*, add a new system server or delete/change an existing one) the configuration of each site, even at execution time. By the way, a new service, (*e.g.*, QoS management), can be introduced into a microkernel-based system much more easily than in a distributed monolithic kernel-based one. Existing well-known systems using this model are Amoeba (Mullender *et al.*, 1990),

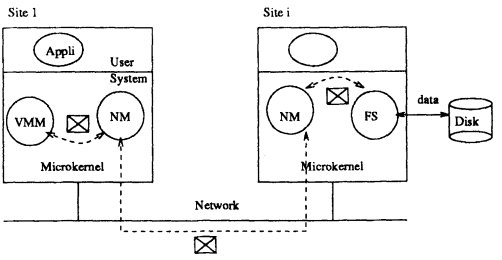

Figure 1 Distributed Microkernel-based Systems.

Mach (Accetta *et al.*, 1986), and Chorus (Rozier *et al.*, 1988). Our experimental system is running Chorus. However, the following analysis may apply to other microkernel-based operating systems as well.

We recall some system object definitions and name system servers presented on the Figure 1. The *virtual memory space* of a process is divided into equal-size *pages*. The virtual memory system works on the page basis. A *region* is a contiguous portion of the virtual memory space of a process. An allocated region can be accessed by the process (*e.g.*, for code execution or for data variable access). The contents (data) of an allocated region may be mapped into some *segment* (*e.g.*, data file or *swap pages*). The File Server (FS) manages these segments. The Virtual Memory Manager (VMM) is responsible for every memory demand from any process executing on the site. The Network Manager (NM) of a site is responsible for every remote communication of any process or server of the site.

2.2 Memory Access Time Breakdown

In order to be able to design mechanisms providing QoS guarantees for time-critical applications such as distributed multimedia applications, we must be able to know how to guarantee memory access times. Therefore, it is necessary to answer to two following questions: *(i)*what is the value of memory access delay experimented by an application (*i.e.*, the amount of time the application must wait when it needs access to some data) ? *(ii)* what system entities are involved in memory access and what are their responsibilities ?

Given our system model, any process memory access falls into one of two following categories:

1. Access to an address in an allocated region of the current process, for instance to fetch the next instruction or to load/store data from/to a program variable.
2. Access to data stored in a external data file, for instance I/O operations *read*, *write*, which transfer the data between an application buffer and a data file.

For the first category, the access time problem appears when the process experiments a *page fault* (*i.e.*, accesses a virtual address of a page that is not in physical memory, for instance the first access to the page or the access to a *swapped out* page). We do not analyze page faults any longer because they can be avoided. In fact, all operating systems

supply primitives allowing applications to reserve and lock physical memory pages. If all program codes are loaded and locked in the system memory, and all program variables are allocated and locked, then there are no chance for page faults to happen. Modern workstations generally have a large physical memory, then this simple solution is generally acceptable.

Let us now analyze the access time for the second category, explicit I/O operations from applications. These operations may be trapped and pre-processed by another system server (*e.g.*, by the *Process Manager* under Chorus/MiX, which traps all application system calls then dispatches them to other system servers if necessary), but they are always transferred to the VMM. The VMM handles I/O operations on a page basis (*i.e.*, the same processing occurs for every page of the application buffer). We distinguish the *read* operation to the *write* one:

For *read* operations, the VMM first checks if the requested page is already loaded in physical memory (*e.g.*, for the needs of some other process). If yes, it copies the data into the application buffer. Otherwise, it allocates a physical memory page for the data page, prepares a message and sends it to the FS, asking this server to read and to send back the requested data page. Then the application process must sleep waiting for the page to be available. If the FS is not in the local site, then the NM is called to send the message over the network. At the FS site, the local NM receives the message, then transfers it to the FS. The FS handles the request and sends back the data page by the reverse way. Upon reception of the page, the application process (that is executing VMM code) is waked up. When scheduled, it copies the data into the application buffer and repeats the operation for the next page if the buffer is not completely read. Otherwise, it returns the buffer to the application process and ends the operation (*e.g.*, by switching the application process from system to user execution mode).

For *write* operations, the VMM allocates a physical memory page and copies data from application buffer in it. Then, it prepares a message and sends it with the data page to the FS, requesting this server to write down the data page into the data file. During this communication, the application process has also to sleep until the arrival of the FS's answer message and the process is scheduled, like in *read* operations.

We identify four system entities that are involved into memory access. The VMM is responsible for local memory handling and message building. The communication system (via the NMs), involved in remote access only, is responsible for message transfer between sites. The FS is responsible for storing or fetching data to/from secondary storage. Finally, the scheduler is responsible for scheduling the application process execution when the data transfer is completed.

Let us now express memory access times for one page, t_{read} for reading and t_{write} for writing, as experimented by the application. Let t_{in}^{VMM} be the sum of all code execution times at the VMM for one data page reading (including table lookup and request preparation), and t_{out}^{VMM} be the sum of all code execution times at the VMM for one data page writing (including data copy and request preparation). Let t^{NM} be the communication time for a message exchange between two sites. It is the same for read and write operations, since it consists in the times for sending and receiving two messages between two NMs, with only one of them containing a data page. Let t_{read}^{FS} and t_{write}^{FS} be respectively the times spent at the FS for a read and write request from its arrival to the FS until the generation of the corresponding answer message by the FS. Finally, let t^{SCH} denote the scheduling time (*i.e.*, the delay between the wakeup of the application process by

the answer message arrival and the resuming of the execution triggered by the system scheduler). We have the following expressions :

$$t_{read} = t_{in}^{VMM} + t^{NM} + t_{read}^{FS} + t^{SCH} \quad (1)$$

$$t_{write} = t_{out}^{VMM} + t^{NM} + t_{write}^{FS} + t^{SCH} \quad (2)$$

These expressions are very general. Some terms may be considered as null in some particular cases. For instance, t^{NM} is null if the FS is local, or only t_{in}^{VMM} is present if the page to read is already loaded in the site physical memory. In this paper, we don't develop detailed expressions for different access cases. A more detailed memory access time analysis is given in (Nguyen *et al.*, 1994).

2.3 Time Guarantees for Memory Access

For the execution of applications requiring temporal guarantees, such as distributed multimedia applications handling digital audio or video, it is mandatory to have bounded memory access times. For example, a video playback application requires that the time to read a screen frame must be less than 40 ms. If each frame has n pages, then the time to read a frame, which is approximatively $n * t_{read}$, must be less than 40 ms. Let us consider now the previous expressions from this point of view.

The terms t_{in}^{VMM} and t_{out}^{VMM} are bounded because they are limited to some system table lookups and message generations and/or data copies. The term t^{NM} depends on the used network type. For instance, an Ethernet LAN can't theoretically guarantee the bounded time for one message sent (Metcalfe *et al.*, 1976). In contrast, an ATM-based network can do it (Minzer, 1989). The terms t_{read}^{FS}, t_{write}^{FS} and t^{SCH} depend on the system file organization and algorithms, and on the scheduling policy. For instance, they are not theoretically bounded in a universal time-sharing system such as UNIX (Bach, 1986).

Therefore, in order to support QoS requirements in current systems, the first problem to solve is to modify system components in such a way that they could guarantee bounded access times.

However, bounded access times for each individual system server is not enough. System servers have to cooperate together in order to offer the QoS guarantee to user applications. Hence, there is a second problem to solve, which concerns the overall cooperation management of involved system elements. The overall management is necessary to decide the admission of a new service request and to control system resources execution.

In the two following sections, we will present our approaches respectively for the overall cooperation management problem and for the design of a distributed file server with Quality of Service. The first one addresses all system resource managers. The second focuses on one particular system resource manager that is the File Server.

3 QOS MANAGEMENT MODEL

In this section we present the tasks that are involved in the overall cooperation management of system resources and discuss the various ways to achieve these tasks.

3.1 Overall QoS Management Tasks

The Quality of Service (QoS) covers several system properties that applications can require from the system. In this paper, we tackle only the QoS requirements of distributed multimedia applications handling continuous media, especially for temporal requirements. These requirements are expressed by some QoS parameters, the most important of them being *rate, delay, jitter*, and *throughput.*

In our point of view, the system should supply service with required QoS to applications on the basis of *QoS contracts.* A QoS contract is a commitment between the system and the application, which specifies the duties to fulfill by each side in order to gain the services offered by the other. For instance, a QoS contract of a video playback may specify that the application must consume a video frame during each 40 ms period, and the system must commit to deliver to the application a frame by each such period. The QoS contract must be set up before service execution.

We state three stages of a service with QoS requirements that are the following:

1. *Negotiation* between the system and the application to set up the QoS contract. The application specifies its QoS requirements to the system. The system translates these QoS requirements into a list of system resources, and then tries to make *reservations* on these resources. Resource reservation means a power allocation of a resource to an application. If the reservation of all involved system resource is successful, then the system sets up the corresponding QoS contract and it notifies the application that it can start. Otherwise, the system reports the reasons of failure to the application. The application may then adjust its requirements and initiate a new negotiation, and so on. The QoS negotiation is often called in the literature *resource admission control.*
2. If the contract is set up, then the system *controls and monitors the resources* in order to grant the QoS to the application. The applications suppose that the required resources are available in time according to contract specifications, so they do not have to explicitly make requests to use a resource. For instance, a video playback application does not have to issue a *read* operation in order to obtain a frame from a video file. The system plays a referee role by detecting any contract violation and reporting it to the application. The application can, at any time, stop the execution of its contract.
3. If the contract couldn't be respected by the system, then the system may ask the application to *renegotiate* the QoS contract, or to continue the contract but in *degradation*, or simply to *stop* it. This may happen if some resource managers cannot provide strong guarantees (for instance, a shared Internet link becomes overloaded) or in case of failures.

We propose a two-phase model for QoS management in a distributed system. In this model, we introduce a new system server, the *QoS Manager*, which plays the role of mediator and referee between the application and all involved resource managers, whether local or remote. Resource distribution is transparent to applications. Each site runs a copy of this QoS Manager. The QoS Manager of a site cooperates with the local resource managers and communicates with QoS Managers of other sites in order to negotiate and to execute QoS contracts. The services of the QoS Manager are divided into two phases for each QoS contract as follows:

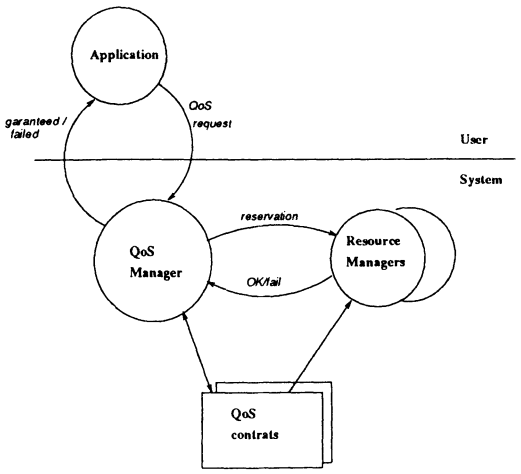

Figure 2 QoS contract negotiation.

Negotiation phase (see Figure 2). In this phase, the QoS Manager plays the role of mediator for the QoS contract negotiation between the application and resource managers. It translates the application QoS request into resource reservations behind resource managers. The result QoS contract is a system data structure updated by the QoS Manager and available for consultation by resource managers. There is a QoS contract for the requesting application at each site that is involved in the services. The QoS contract on each site prescribes the duties that local resource managers have to fulfill during the service. At the end of resource reservations, the QoS Manager notifies the application if its request was accepted or not. If yes, the application can go to the execution phase. Otherwise, it can initiate a new negotiation.

Execution phase (see Figure 3). In this phase, the QoS Manager controls and monitors the activities of local resource managers with respect to current "signed" contracts. The resources are granted to applications without explicit requests (we use dotted lines in the figure to indicate the implicit access to resources). If the application wants to stop its contract, it notifies the local QoS Manager. This manager frees local resources allocated to the contract and notifies its peers at remote sites of the end of the contract. During QoS contract execution, the QoS Manager detects any anomaly and contract violation. It can notify the applications and other QoS Managers about these anomalies, undertake some anticipation actions when some symptoms appear, or initiate a *contract renegotiation* with the applications if necessary.

3.2 Overall QoS Management Implementation

The functions of the QoS Manager could be completely distributed over resource managers, centralized for each site, or a combination of two. In the first case, each resource manager must integrate the QoS management in its functions. The application must know the resource managers (and their locations) that it needs in order to directly negotiate with them in order to set up a QoS contract with each resource. This approach may be simple if all applications have the same behavior and need only one or two specific

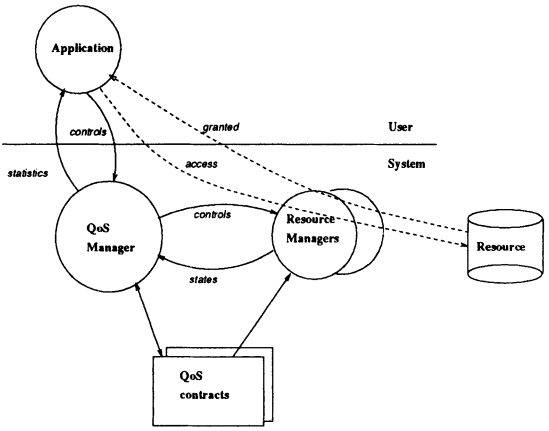

Figure 3 QoS contract execution.

resources. However, QoS management is not transparent at all for the applications, and the interfaces to the system are complicated. We don't believe that a real system may use this approach.

The centralized QoS management implies the existence of the QoS Manager as a central system element at each site. In this case, the QoS Manager can control and realize the access to all local resource managers. These managers become pure executive elements. The QoS management is transparent to applications and the interface to the system may be simple and abstract. The applications must not know about what resource managers they need, neither their locations. But some resource managers are actually quite complicated (*e.g.*, file server or network manager), and it is not straightforward for the QoS Manager to consider them as pure executive elements.

The hybrid approach supposes the existence of a central QoS Manager at every site, but local resource managers are not pure executive elements. They can run in two functional modes: normal and QoS. In normal mode, they behave in their standard way. The QoS Manager can switch them in QoS mode and vice-versa. In QoS mode, QoS contract execution has priority over no-QoS services. The resource managers must then run appropriated (*i.e.*, specialized) algorithms. The central QoS management provides a simple interface between system and application and the transparency. QoS management may be also progressively integrated into an existing system. Our choice is to implement overall QoS management with this hybrid approach.

We have started the implementation with a restricted QoS Manager running on Chorus/MiX, called *Prepager*. Its name means that it only deals with the file server and it reads in advance continuous media streams with respect to some QoS contracts.

The interface between applications and the system (QoS Manager) is realized by a set of primitives that applications can invoke. These primitives may be implemented in the form of a library. Each primitive has a semantics (*e.g.*, *play, stop* a stream) and issues a request from an application to the QoS Manager. The application ignores the dialog between library functions and the QoS Manager. It invokes a primitive, then reacts with

respect to its result. For instance, we have implemented in *Prepager* some primitives allowing the playback of a single stream as follows:

`int PLAY(char* stmname, StreamQoS* qos, VmAddr* buffaddr, int* buffnum);` Playback request for the stream `stmname` with the QoS specified by `qos` (roughly, it consists of a data structure with QoS parameters like *rate, buffsize, jitter*). If the request is accepted, then `PLAY` returns 0 and `buffaddr` gives the address of a buffer pool of `buffnum` buffers of `buffsize` length each. In this case, the application can start the display cycle. Otherwise, `PLAY` returns a error code (*e.g.*, stream not found, can't reserve system resource).

`int STOP(char* stmname);` Stop the playback of the stream `stmname`. The QoS Manager frees allocated system resources, the QoS contract, and sends back some statistics about the contract execution (*e.g.*, number of data buffers that have been read, number of contract violations, violation mean delay). These statistic data are displayed to inform the user.

`int PAUSE(char* stmname);` Stop temporarily the contract for the stream `stmname`. Allocated resources are still reserved, but the file server stops reading for this stream.

`int CONTINUE(char* stmname);` Resume the execution of the paused contract for the stream `stmname`.

In addition to these primitives invoked by the application, an asynchronous signaling mechanism is used by the system to notify the application when the end of stream is reached or in case of contract violations.

This interface for distributed multimedia applications is in no way complete. Our intention here is only to illustrate the form of such an interface with the overall QoS manager.

4 QOSOM: AN OBJECT MANAGER WITH QOS IN CHORUS/MIX

As an example of realization of the model just described, we present in this section the design of a system resource manager that can run in QoS mode and its interactions with the local QoS Manager. The chosen resource manager is OM (*Object Manager*), which is the file server under Chorus/MiX, the UNIX System V sub-system built on top of the distributed microkernel Chorus.

4.1 Access Time Problems of OM

OM is an independent system server of the Chorus/MiX Operating System (Rozier *et al.*, 1988). It acts both as a *swap device* for virtual memory management and as a file server for file management. Other system managers communicate with it exclusively by message exchanges in order to access data segments.

OM implements the UNIX System V file system semantics. It uses the same data structures and algorithms as in the standard UNIX System V File System. A complete description of the UNIX System V File System can be found in (Bach, 1986).

In analyzing data structures and algorithms used by OM for handling I/O requests, we identify the three following reasons that make the I/O time for a data block indeterminate (and unbounded). For clarity, we only consider the block data reading case:

1. The *layout* of UNIX file data blocks makes the time to read a data block not homogeneous. In fact, UNIX file data blocks are organized into levels using indirect blocks that content pointers to next level indirect blocks or to real data blocks. There are 3 indirection levels. The time to read a data block then varies from 1 to 4 block reading(s) depending on the logical position (offset) of the data block in the file.
2. The *cache management policy* does not allow to estimate the reading delay upper bound, even for one disk block. The server manages a pool of cache buffers. The data transfer between the server and the disk driver is done via these cache buffers. Every disk block corresponds to one buffer (computed from disk block number and system file number). The file server must allocate a cache buffer for the block before formulating a block read request to the disk driver. It has then to wait if either the corresponding buffer is found in the buffer pool but is busy (*e.g.*, it is blocked by another data transfer), or isn't found and the free buffer list is empty. In a competitive system, this cache waiting time is theoretically not bounded.
3. The *strategy algorithm* used by the driver can delay the queuing time for block reading in order to optimize the magnetic head movement. This means that the handling order of block read requests at the disk driver can differ to their arrival order. For instance, a later block read request can be handled before a in-queuing one if its physical position economizes the head movement. This *strategy* does not make the queuing time at the disk driver unbounded, but it makes this time indeterminate.

Therefore, the standard UNIX system file is not suitable to support QoS requirements of multimedia applications. However, we have identified the sources of non-determinism in the file management. The problem now is how to overcome this non-determinism.

4.2 QoSOM design

We propose some modifications of algorithms and data structures used by OM that can make the I/O delay for a data block deterministic and bounded. The resulting OM, called QoSOM, will have a deterministic behavior behind standard UNIX system file characteristics. For our first experimental version, we only consider the *continuous reading* of multimedia streams that are stored as ordinary UNIX files.

The QoSOM can run in one of two modes: normal and QoS. In normal mode, it uses standard algorithms and data structures. When being switched in QoS mode, it must use new algorithms and abstractions as follows:

We define the *trace* of a file reading as being the path composed of 0 to 3 indirect block(s) that lead to the current data block. In standard UNIX, an algorithm called *bmap* is used to do the mapping between the logical and physical block numbers of the current data block. This algorithm consists in finding the *trace* leading to the current data block in order to extract the physical block number of this later. As these indirect block(s) are freed after that, then the next mapping may have to read them again if they have been destroyed between two mappings.

We propose an enhanced algorithm for *bmap*, called *cbmap* (standing for *continuous bmap*), that conserves the *trace* for a *continuous reading*. With trace conservation, indirect block readings are only necessary when the *trace* changes. We can point out the set of logical block numbers where indirect block readings may occur and how many block

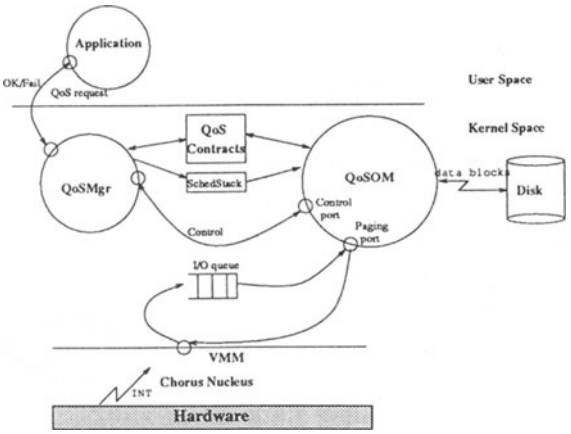

Figure 4 QoSOM System Model.

readings are necessary for each block of the set. Therefore, the number of block readings for a data block can be accurately computed.

QoSOM applies a *FIFO strategy* in QoS mode. As QoSOM is an independent server and as all data requests go through a specific *port* (Chorus abstraction that is a communication address and a message queue), it is easy to control the processing order of I/O requests. The *FIFO strategy* means that the server handles one request at a time, without multiplexing with others. For instance, if a read request for one data page needs 4 data blocks, then these data blocks will be completely read before the handling of the next request. Although this strategy may penalize the server overall performance, it allows to determine how long a read request could be, given the disk block fetching delay t_{rblk} by the disk controller. This strategy naturally excludes the cache buffer race problem of the standard strategy.

We propose a new abstraction, namely *partial_read*. This is a read request with a *time quota*. The *time quota* is the number of disk block fetchings that the server can formulate to the disk controller during the request handling. Coming together with the *trace conservation*, the *partial_read* allows to split the reading of an application buffer into several steps without additional delay in comparison with the necessary delay to read the whole buffer once. This abstraction is useful for the QoS Manager to schedule multiple continuous media readings in our system scheme as described below.

4.3 QoSOM System Model

The interaction scheme between the applications, the QoS Manager (QoSMgr) and the QoSOM for multiple continuous media readings is shown in Figure 4. In this scheme, the QoSMgr negotiates with the applications in order to set up QoS contracts. It controls the QoSOM function mode. The QoSOM handles two request sources: the I/O queue consisting of regular (non-QoS) I/O requests, and a QoS scheduled request stack (explained below). In QoS mode, the QoS stack has priority over the I/O queue. Between two consecutive QoS requests, QoSOM tries to serve partially I/O queue's requests.

The QoSMgr achieves the *admission control* for a new QoS playing demand from applications on the basis of *resource capacity*. The capacity for disk controllers is expressed by the number of disk blocks that the controller can fetch in one second. Based on the application QoS specification (rate, buffer size), the QoSMgr computes the *application required capacity*. The QoS application request is accepted if the sum of all already running *application required capacities* and of the new one is lower than the *disk controller capacity*. If yes, the QoSMgr reserves necessary data buffers for application, sets up the corresponding QoS contract and notifies the application that it can go on.

The QoSMgr controls the execution of accepted QoS contracts on the basis of *round*. A *round* is the time interval that is equal to the smallest common multiplier of all contracted stream periods. A new admitted stream can only be started at the beginning of the next round. Before each round, the QoSMgr computes the read scheduling for this round and pushes it onto the *SchedStack* stack. The QoSOM pops scheduled QoS requests from *SchedStack* and executes them. A scheduled QoS request is a triplet (t, f, q) where t is the scheduling time relative to the beginning time of the current round, f is a QoS contract handle, and q is the allocated *time quota* expressed as a number of disk block readings.

The QoSMgr computes the scheduling stack using the *partial_read* abstraction. The algorithm, namely *StmSched*, for this computing is relatively simple. However, two interesting results are deduced from it: *(i)* all application data buffers are available before their deadline, and *(ii)* each stream needs two buffers for continuous reading (*i.e.*, one in use by the application and the other in use by QoSOM – this is the minimum buffer number). The full description and proof of the algorithm is given in (Nguyen, 1995).

We are now achieving the implementation of an experimental version of the QoSOM and the QoSMgr under Chorus/MiX v4 and Chorus Kernel v3 r4.2.

5 RELATED WORK

In this section, we present some related works and compare them with our approaches. We are interested in works about system support for overall QoS management and multimedia file servers. The necessity of resource reservation and resource management with QoS constraints for distributed multimedia applications has been argued in (Herrtwich, 1991). A detailed survey of the state-of-art in QoS management for distributed systems is given in (Hutchison *et al.*, 1994).

The Multimedia Group of Lancaster University has developed a multimedia basic services platform for distributed multimedia application programming (Blair *et al.*, 1992). They propose and implement a prototype of a QoS Architecture (QoS-A). In this architecture, they emphasize the necessity of specialized transport services for continuous media and an orchestration architecture for related continuous media coordination (Campbell *et al.*, 1992). They define an enhanced transport services interface that offers a framework to specify and implement the required performances properties of new multimedia application over multi-service ATM-based networks (Campbell *et al.*, 1993). In their proposed interface, the QoS negotiation result is a contract that specifies an agreed *level of service* for service providers, an agreed *level of traffic* for user application, and a *degree of commitment* for service guarantee. The QoS must be monitored at all system layers, but there is no QoS Manager that coordinates the activities of the different system resource managers.

Concerning operating system support for distributed multimedia applications, Lancaster University's researchers have proposed an extension of microkernel abstractions in order to integrate the QoS abstraction in the microkernel (Coulson *et al.*, 1993(a)), (Coulson *et al.*, 1993(b)). They propose and implement an extended API of Chorus microkernel with some new low-level system calls and abstractions as *rtport, devices, rthandlers, QoS controlled connections*, and *QoS handlers*. Two implementation issues are focused: scheduling and communications. Recently, the memory resource has been investigated (Robin *et al.*, 1994). In this work, a *QoS Mapper* is proposed. This *QoS Mapper* is not a Chorus mapper. It manages only the local memory resource and does not manage the cooperation between resource managers as our QoS Manager does.

Tawbi and Horlait (1994) propose a QoS framework which identifies four levels at which QoS should be studied. At the system level, they have developed a QoS management model. In this model, there is an Application QoS Manager (AQoSM) that does some similar tasks to our QoS Manager. They have developed a protocol, namely HLQNP, for inter-AQoSM communications to realize the distributed application management. However, the QoS contract is not mentioned. The interaction model between AQoSM and system resource does not specify clearly how resource managers must be changed in order to be integrated in the model. AQoSM does not seem to do things like resource scheduling as QoSMgr does in our approach. Emphasis is put on the communications system.

Concerning QoS file servers, researchers focus on dedicated multimedia server design. Some significant papers in this field are Anderson *et al.*(1991) for a Continuous Media File System (CMFS) that supports real-time storage and retrieval of continuous media data on disk, Rangan and Vin (1991) for a digital video and video file system for applications like Multi-User On-Demand HDTV playback, Lougher and Shepherd (1993) for a continuous media server implemented on transputers. Common characteristics of these servers are their special disk layout, dedicated techniques and algorithms for storing/retrieving continuous media, and their autonomy with respect to QoS management. Our QoSOM, in contrast, is a standard file server enhanced by QoS mode and running under the control of QoS Manager.

6 CONCLUSIONS AND FUTURE WORK

In this paper we have identified the critical system components involved in the execution of distributed multimedia applications accessing digital audio or video stored on a remote site. The network is only one of these components. Even if high-speed networks can provide today the bandwidth needed by such applications, system support must be provided for granting QoS guarantees to applications. We have presented the design of a system support composed of a QoS manager replicated on each site and of modified resource managers providing temporal guarantees for the access to their own resource. Applications interact with the system through a well-defined interface for expressing their QoS requirements and driving the execution. If the QoS requirements can be fulfilled by the system, then the application is allowed to start execution.

The application-system interface is not complete. We have just presented the minimum interface needed for the execution of distributed applications playing one audio or video stream. This interface will be extended to fit the needs of more general applications

(*e.g.*, handling simultaneous media with synchronization constraints, backward playing or variable speed).

Only a restricted version (*Prepager*) of the QoS Manager has been implemented, in the context of the Chorus microkernel. A more general version is currently being implemented, taking resource managers other than the File Server into account.

The example of the File Server was taken as an illustration of the possibility of designing a resource manager that can provide temporal guarantees, under the control of a QoS manager, for applications with QoS requirements, while behaving in the standard way for other applications. This approach was already followed for CPU scheduling (Govindan *et al.*, 1991). Moreover, we have shown that the file system structure need not be modified in order to support multimedia applications, only the management has to be. This way, non-multimedia applications can cohabit with multimedia applications on the same hardware and software environment, namely standard workstations running UNIX.

REFERENCES

M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. (1986) Mach: A New Kernel Foundation For UNIX Development. In *Proc. Summer 1986 USENIX Technical Conference*, 93–19.

D. Anderson, Y. Osawa and R. Govidan. (1991) Real-Time Disk Storage and Retrieval of Digital Audio/Video Data. Technical Report No. UCB/CSD 91/646, University of California, Berkeley, (USA).

M. J. Bach. (1986) *The Design of the UNIX Operating System.* Prentice-Hall Software Series, Englewood Cliff, New Jersey 07632.

G. Blair, G. Coulson, P. Auzimour, L. Hazard, F. Horn, and J.-B. Stefani. (1992) An Integrated Plaform and Computational Model for Open Distributed Multimedia Applications. In *Proc. 3rd International Workshop on Network and Operating System Support for Digital Audio and Video*, 209–13.

A. Campbell, G. Coulson, and D. Hutchison. (1992) A Continuous Media Transport and Orchestration Service. In *Proc. ACM SIGCOMM '92*, ?–12.

A. Campbell, G. Coulson, F. Garcia, D. Hutchison, and H Leopold. (1993) Integrated Quality of Service for Multimedia Communications. In *Proc. IEEE INFOCOM'93*, ?–17.

G. Coulson, G. S. Blair, and P. Robin. (1993)(a) Micro-Kernel Support for Continuous Media in Distributed Systems. *Computer Networks and ISDN Systems, (Special Issue on Multimedia)*, ?–20.

G. Coulson, G. S. Blair, P. Robin, and D. Shepherd. (1993)(b) Extending the Chorus Micro-Kernel to Support Continuous Media Applications. In *Proc. 4th International Workshop on Network and Operating System Support for Digital Audio and Video*, ?–12.

A. Danthine and O. Bonaventure. (1994) From Best Effort to Enhanced QoS. In *Proc. of the Workshop on Communicating Informatics and Distributed Systems (New Technologies and New Requirements)*, 19–26.

R. Govindan and D.P. Anderson. (1991) Scheduling and IPC Mechanisms for Continuous Media. In *Proc. 13th ACM Symposium on Operating Systems Principles*, 68–12.

R.G. Herrtwich. (1991) The Role of Performance, Scheduling, and Resource Reservation in Multimedia Systems. In *Proc. International Workshop on Operating Systems of the*

90s and Beyond, Lecture Notes in Computer Science, Springer-Verlag, **563**, 279–6.

D. Hutchison, G. Coulson, A. Campbell, and G. S. Blair. (1994) *Distributed System Management.* Moris Sloman, Imperial College London.

P. Lougher and D. Shepherd. (1993) The Design of a Storage Server for Continuous Media. *Computer Journal (Special Issue on Multimedia)*, **36**(1), 32–11.

C. Mercer, S. Savage, and H. Tokuda. (1994) Processor Capacity Reserves: Operating Support for Multimedia Applications. In *Proc. of the IEEE International Conference on Multimedia Computing and Systems*, ?–15.

R.M. Metcalfe and D.R. Boggs. (1976) Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, **19**(3).

S.E. Minzer. (1989) Broadband ISDN and Asynchronous Transfer Mode (ATM). *IEEE Communications Magazine*, **29**, 17–8.

S.J. Mullender, G.V. Rossum, A.S. Tanenbaum, R.V. Renesse and H.V. Staveren. (1990) Amoeba: A Distributed Operating System for the 1990s. *IEEE Computer*, **23**, 44–9.

H.Q. Nguyen. (1995) Serveur de segments OM de Chorus/MiX et lecture de flots de média continus avec QoS. Technical report, Institut National des Télécommunications, Evry, (France).

H.Q. Nguyen and G. Bernard. (1994) Gestion de Mémoire Virtuelle avec Garanties de Qualité de Service (QoS) dans un environnement Réparti à base de micronoyaux. Technical report, Institut National des Télécommunications, Evry, (France).

C. Nicolaou. (1990) An Architecture for Real-Time Multimedia Communication Systems. *IEEE Journal on Selected Areas in Communications*, **8**(3), 391–9.

P.V. Rangan and H.M. Vin. (1991) Designing File Systems for Digital Video and Audio. In *Proc. of the 13th ACM Symposium on Operating Systems Principles*, 69–11.

P. Robin, G. Coulson, A. Campbell, G. S. Blair, and M. Papathomas. (1994) Implementing a QoS Controlled ATM Based Communication System in Chorus. Internal Report MPG-94-05, Lancaster University (UK).

M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. (1988) Chorus Distributed Operating Systems. *Computing Systems*, **1**, 305–74.

A. S. Tanenbaum. (1995) *Distributed Operating Systems.* Prentice-Hall Software Series, Englewood Cliff, New Jersey 07632.

W. Tawbi and E. Horlait. (1994) Expression and Management of QoS in Multimedia Communication Systems. *Annales des Télécommunications*, **49**(5-6), 282–14.

Nguyen Hong Quang is a full-time Ph.D student in Computer Science at the Computer Science Department of the Institut National des Telecommunications (France Telecom University). He received the B. Sc. and the Master (DEA) degree in Computer Science from respectively the Free University of Brussels, Belgium in 1984 and the University of Evry, France in 1993. He formerly worked in the Programming Department of the Institute of Informatics, Hanoi, Vietnam. His current interests include distributed systems, microkernel technology and distributed multimedia.

Guy Bernard is graduate from Ecole Centrale of Paris (french "Grande Ecole") and received a Ph.D. in Computer Science from the University of Paris XI-Orsay in 1989. After being assistant professor at the University of Paris XI-Orsay, he was scientific advisor at the Altair INRIA project, in charge of supervising the design of the distribution scheme between clients and servers in the OO-DBMS "O2". He joined the Institut

National des Telecommunications (France Telecom University) in 1989, as a Professor in charge of the Distributed Systems research group. Since 1994 he has been Head of the Computer Department of the Institute. His current interests include distributed systems, object-oriented systems, microkernel technology and distributed multimedia.

Djamel Belaid received the Docteur en Mathématiques Appliquées from the Université Scientifique de Grenoble, France, in 1985. He is presently an Assistant Professor in the Department of Computer Science (Distributed Systems group) at the Institut National des Telecommunications (France Telecom University). His current interests include large scale distributed systems, microkernel technology and distributed multimedia.