

Open-loop streaming Near-VOD client

*Master thesis
(Proyecto Fin de Carrera)*

Ivan Arribas Alonso

July 19, 2001

INSTITUT EURÉCOM
Sophia-Antipolis
France



Universidad Pública de Navarra
Pamplona

Escuela Técnica de Ingeniería Industrial y de Telecomunicación



Tutors in Institut Eurécom:
Prof. Ernst Biersack
Víctor Ramos

Tutor U.P.N.A
Prof. Dr. Jose Ramón González de Mendivil

Contents

1	Introduction	7
1.1	Near Video On Demand	10
1.2	Open-loop and closed-loop schemes	10
1.3	Common terms and concepts	13
1.4	Objectives and system requirements	15
2	System design	18
2.1	Transmission scenario	19
2.2	Module architecture	20
2.3	Main system tasks and characteristics	22
2.3.1	Configuration channel	23
2.3.2	Algorithm implementation	24
2.3.3	Segment management	25
2.3.4	Reception	28
2.3.5	Statistics	32
2.3.6	Display	34
2.3.7	Timeout management	36
2.3.8	Application initialization and session management	39
3	Results	41
3.1	Result interpretation	42
3.2	Results	43
3.2.1	Transmission scenario	44
3.2.2	Tests	46
4	Conclusions	64
5	Open issues	66

List of Figures

1.1	first client arrives in a true-VOD system	8
1.2	N+1-th client arrives in a true-VOD system	9
1.3	Basic transmission scheduling for open-loop NVOD system	11
1.4	A client arrives and starts to store the segment data	14
1.5	Segment multiplexing in multicast channels	15
2.1	Typical transmission scenario	19
2.2	Main system modules	21
2.3	GUI and <code>init</code> modules tasks	22
2.4	<code>bufferscheduler</code> module tasks	23
2.5	<code>VideoObject</code> class	24
2.6	Algorithm implementation by inheritance	26
2.7	Segment management interaction diagram	27
2.8	Interaction diagram for reception	29
2.9	Application Data Unit format	30
2.10	Interaction diagram for the statistic collecting	33
2.11	Interaction diagram for the timeout management	38
3.1	Packet sent scheduling to have more precision in the transmission rate	45
3.2	Aggregate buffer occupation for the test 1	48
3.3	Aggregate reception rate for the test 1	49
3.4	Aggregate buffer occupation for test 2	50
3.5	Aggregate reception rate for test 2	51
3.6	Aggregate buffer occupation for test 3	52
3.7	Aggregate reception rate for test 3	52
3.8	Aggregate buffer occupation for test 4	55
3.9	Aggregate reception rate for test 4	55
3.10	Packet statistics for test 4	56
3.11	Packet statistics for variation in test 4	56

3.12	Packet statistics for second variation in test 4	57
3.13	Aggregate buffer occupation for test 5	59
3.14	Aggregate reception rate for test 5	60
3.15	Aggregate buffer occupation for test 6	61
3.16	Aggregate reception rate for test 6	61
3.17	Packet statistics for test 6	62
3.18	CPU load during test 6	62
A.1	Segmentation in the Birk algorithm	68
A.2	Example of Birk transmission	69
A.3	A client joins at the t_2 time	70

List of Tables

1.1	Positive aspects for NVOD and true-VOD	13
1.2	The most important parameters in a NVOD system	16

Chapter 1

Introduction

The increasing use of the Internet as a massive information and entertainment medium has produced the proliferation and the emergence of new multimedia applications. At the same time, services for broadcasting and packet switched networks are evolving to the next generation of communications. Some examples of these services are already provided by several companies in the world like telephony conferencing, video-telephony, distance learning, video conferencing, etc. However, due to the hard bandwidth requirements, as well in the server/network side as in the client side, the proliferation of these services has evolved very slowly.

The major concern for a service provider is: how to provide such kind of service with a good quality to a numerous group of users keeping the server and network resources in feasible limits?

The straight-forward solution to provide such a service is to open a dedicated connection to each client every time a request is issued to the server. The server has finite I/O bandwidth resources, and as more clients demand its service, the server I/O bandwidth will become the bottleneck of the service.

Current on-demand services in the Internet are audio services that work in their most part in the same fashion as the one described in the last paragraph.

A video-on-demand service consist in providing a given video chosen from a given list of videos to a potential numerous group of clients. If the service is provided in a one-to-one fashion, as we have just described, the scalability of the system will be very soon exhausted since the server I/O bandwidth is proportional to the number of connected clients.

A video-on-demand system generally consists in three major parts:

1. The video server. Which provides access to the list of available videos and

serves the data.

2. The transmission network. Is the communication medium by which clients have access to the service.
3. The clients. These are entities for which the service is provided. A set-top-box (STB) provides controls to access the service, select and request a movie from the list, play the movie, and it will possibly offer interactive functions as Pause, Fast-Backward, Fast-Forward, Slow-Forward, Slow-Backward, etc.

Typically, the STB will be a dedicated user equipment, but for VOD a well suited device that could serve as STB is a personal computer.

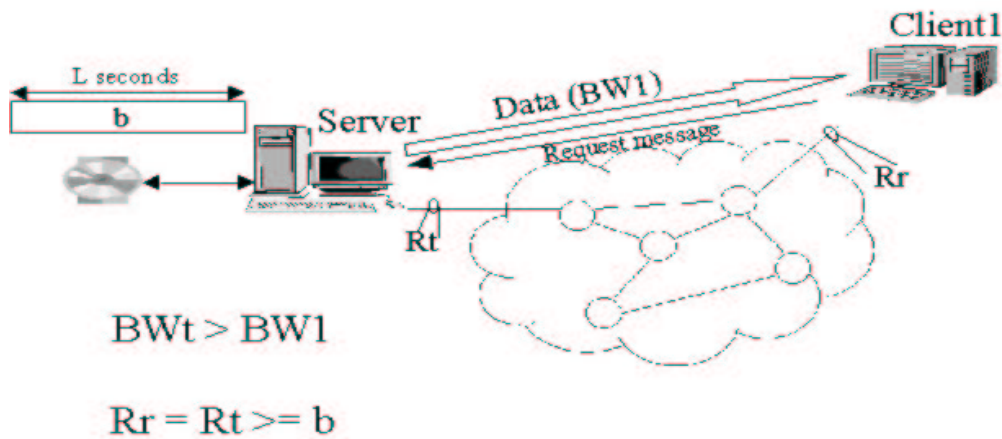


Figure 1.1: first client arrives in a true-VOD system

As we can see in figure 1.1, when a client wants to “see” a movie, it sends a message to the server to indicate the desired request. Afterwards, there can be a dialog protocol between the client and the server to negotiate the parameters of the session, such as minimum transmission rate supported by the film, maximum transmission rate supported by the client/access network, name of film, etc.

After this conversation, the server proceeds to send the movie’s data on a **dedicated channel**. This transmission will go on for at least L seconds, the duration of the movie, and the transmission rate, R_t , will be equal to the movie’s consumption rate, b .

This kind of interaction is why we call this system **on-demand**. The client “demands” to the server the service. The fact that the client has a “dedicated stream of data” is important in order to add interactive functionality in the future, like Fast-Forward, Rewind, or Error recovering.

While the client receives the data, it displays the movie with a little buffering for avoiding starvation of data due to the variable delay of data packets.

This is the most likely situation for the arrival of the first client, but we can also have a transmission rate, R_t , greater than consumption rate, b . In that case, the client must have a considerable buffer for not missing data. We can also do any kind of flow control.

When a client arrives in a situation like the one depicted in figure 1.2, the server has many considerations to do. It must see if it's possible to serve another client by computing the aggregate bandwidth resulting from this operation. If it's possible, it will open **another stream of data dedicated to this client**. If it isn't possible, it will refuse the connection.

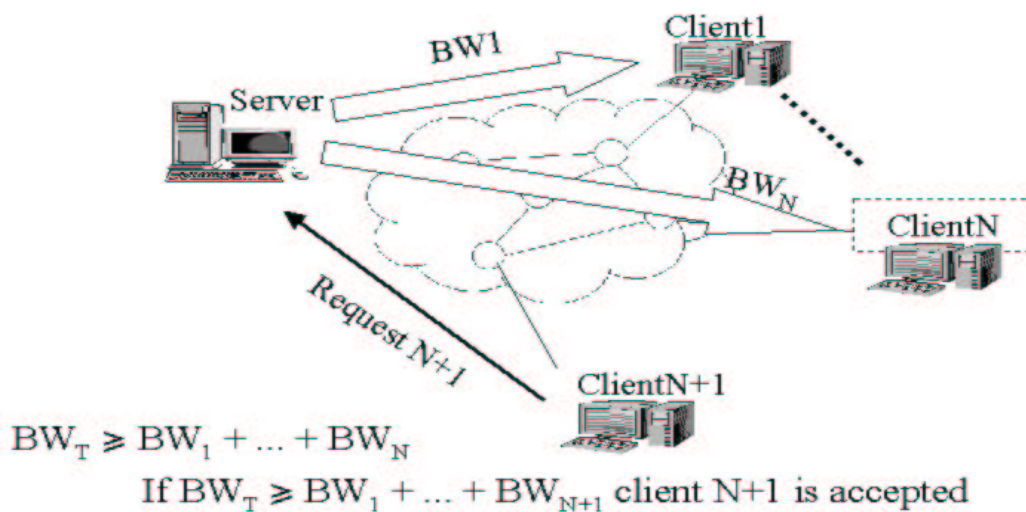


Figure 1.2: N+1-th client arrives in a true-VOD system

It's important to underline that for VOD, the required bandwidth resources both in the server and in the communication network are proportional to the number of concurrent viewers. This leads to an **unscalable system** due to the finite available server bandwidth.

That is the most important drawback of this kind of video server systems. The simplicity is one of its best characteristics.

Scalability is one of the most important properties a distribution system can have. It's trying to achieve this characteristic why we'll introduce the next kind of video server system: Near Video-On-Demand.

1.1 Near Video On Demand

There are important situations in which many viewers wish to view the same content during the same period of time, but not simultaneously. One example is a newly released hot movie that is moreover advertised heavily.

Although VOD could in principle be used to address such situations, it should be noted that the total (over all movies) number of concurrent viewers may be much higher than usual. It would therefore be very costly if not impossible to design the infrastructure (server and communication network) for such peaks. The challenge is to exploit the knowledge that there are many concurrent viewers of the hot movie in order to provide the service more efficiently. The solutions entail **providing to an unlimited number of viewers of the same movie similar service flexibility to that of VOD at a reasonable cost to the server and communication network**. That is, NVOD is an approach to VOD with a little cost for a high number of clients. Ideally, this cost is independent of the number of viewers. The service is named "Near Video On Demand".

One example of how "near" is a NVOD system, is the next constraint typical for these kind of systems. Users will suffer a delay from the instant they demand the movie and the moment they start to visualize it. With true-VOD, this delay is negligible.

Basically, there are two approaches to NVOD. We next describe them briefly.

1.2 Open-loop and closed-loop schemes

NVOD schemes may be classified in two categories: open-loop schemes and closed-loop schemes. With *open-loop schemes*, there is no feedback from the viewing client to the server, so neither server transmissions nor routing on the network are affected by viewer actions.

The basis of these schemes is the **periodic broadcasting** of data. When a client arrives to the system, it joins the transmission of the server and waits an interval of time to be able to start the film display (see figure 1.3). The time it must wait before the starting of playback measures how this system approaches

a true-VOD system. This introduces a new term in our discussion; this time is called **playback delay** or **startup latency**, and we'll denote it by d .

Figure 1.3 gives a very basic fashion of scheduling the movie data from the server to the clients: it broadcasts periodically the entire movie. That leads us to the worst case in which a new client must wait L seconds, the film duration!

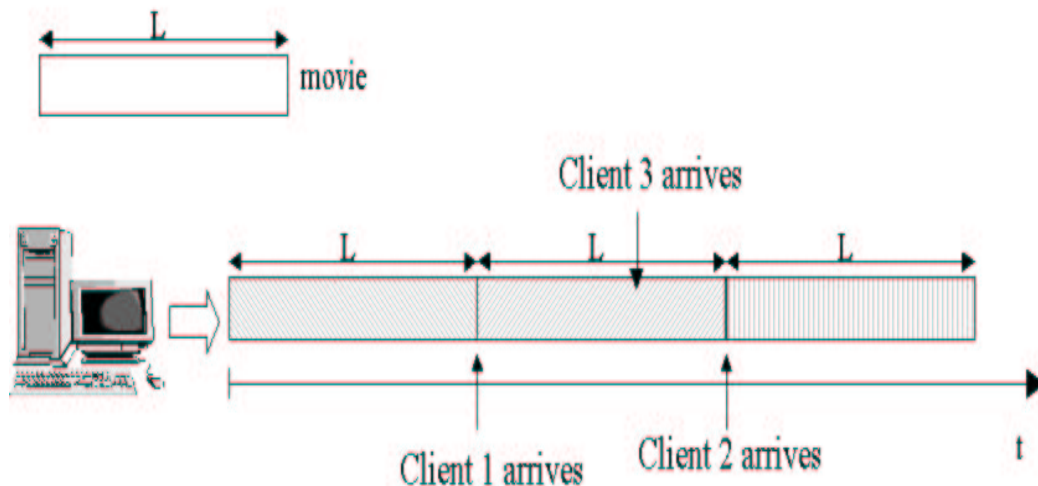


Figure 1.3: Basic transmission scheduling for open-loop NVOD system

To avoid it, various algorithms have been proposed based on the segmentation of the movie to reduce d . They can be divided into two categories, those who split the movie into segments of equal size and those who do it with variable segment size. Examples of the formers are the “Greedy-disk broadcast” (see [7]) and “Skyscraper” (see [4]); both of them are based on the pyramid broadcasting scheme.

For variable segment size, we find examples as “Harmonic broadcasting” (see [6]) and “Pagoda broadcasting” (see [11]). These examples are completed with a general algorithm, that is a central subject in this work, proposed by Birk (see [1]).

For all these algorithms, we can see the reference [12] for the mathematic justification, and [10] for a summary. In all of them, the startup latency is drastically reduced to a reasonable time delay and the **server/network bandwidth is independent of the number of clients**. That is the central issue of NVOD algorithms. But this kind of service has its drawbacks. All of them are placed on the client side.

For example, in true-VOD schemes, the receiving rate needed at the client side is equal to the transmission rate used by the server, and, most of times, equal to the movie consumption rate. But now we will have to record data from various segments simultaneously, that is, the **reception rate, R_r , is increased**.

Another drawback, is that clients must have **a buffer to store the segments that we won't immediately display**. This buffer reaches typically at least the 30% of the film size (see [12]), that is, it's not at all a buffering that can be placed in memory. We must have a device, usually a hard-disk, for storing data. The maximum buffering needed is denoted by S_{max} .

In those systems, the term "on-demand" is not purely correct because the service is provided permanently and not by client requests, but is used to describe the "approximation to a VOD system".

Open-loop schemes lend themselves most naturally to broadcast networks, and are best suited to such networks that have only one-way communication. This is the common case in satellite-based information dissemination networks, cable networks. Multicast IP is also a good candidate.

Closed-loop schemes permit some feedback that allows the server to adapt the transmission to the client requests. For NVOD, the basic closed-loop scheme is based in a batching approach. It's based on **slotted multicast or broadcast**, that is, the server creates "groups" of requests to the same movie (batches) that lie within slots of time and serves a **dedicated data stream for each group**. The slots duration give the maximum startup delay.

For this kind of service, we haven't the drawbacks we had in open-loop systems, but we still have the scalability restrictions of a true-VOD system. Moreover, the savings in bandwidth depends on the distribution in time of client arrivals.

That is derived from the fact that batching systems are pure on-demand systems, that is, there is a request by the client to the server; and the server decides if it can serve the request of service.

We prefer, in this work, to consider open-loop schemes because they give independency to the server about the number of clients that join the transmission.

In contrast, the drawback of these schemes is the lack of communication towards the server in the case of, for instance, losses.

Table 1.1 shows a comparison between NVOD and VOD characteristics.

true-VOD	NVOD
Simplicity	
Easy implementation for FF,Rew	BW is not proportional to client number ⇒ SCALABILITY
Possibility of flow control	
Possibility of error control	

Table 1.1: Positive aspects for NVOD and true-VOD

1.3 Common terms and concepts

The broadcasting protocols concerned by this work have many common concepts. For example, many of the protocols divide videos into **segments**. Segments are consecutive chunks of video. By concatenating the segments together (or by playing them in order), clients will have (see) the entire video.

The **consumption rate**, b , is the rate at which the client processes data in order to provide video output. For MPEG-1, this rate is typically 1.5 Mbps (see [3]). We will represent the consumption rate as b and use it as the unit of measure for NVOD server bandwidth. When all video segments are of the same size, we define the time it takes the client to consume a segment as a **slot**.

Each distinct stream of data is considered to be a logical **channel** on the NVOD server. These channels don't need to be of bandwidth b , and each video may have its segments distributed over several physical channels.

When the channel bandwidth is greater than b or when the client must listen to multiple channels, the client must have some form of local storage. Since MPEG-1 requires at least 100 MBytes per minute, the client storage is likely to reside on disk. The storage capacity on the client and the number of channels the client must listen to are the two client requirements that differ between the broadcasting protocols. Therefore these two concepts are essential in the study of algorithms, and, in the conception of our system. So, **the peak client recording rate**, R_r , and its **peak storage requirement**, S_{max} , are two important parameters.

The fewer requirements placed on the client, the less it will cost to be produced, and the more attractive it will be to users.

We define the **playback delay** or **startup latency**, d , for a client to be the time interval between the moment when it arrives, and the moment it starts the playback of the video. The client can start to receive the video as soon as it arrives, but may have to wait for some time to build up a sufficient playback buffer to ensure lossless and starvation-free playback.

We next explain a bit this need of waiting a time before starting the movie

display. A client can arrive at any time to the system. That is, it can start to record data at anytime, not only when the server starts the transmission of the first segment. Then, it's possible, and very probable, that the client starts to record any part of first segment different of the segment start. At least, it can't start to display the movie before the moment it has received data corresponding to the first part of the first segment. Moreover, for avoiding starvation of data, we state a system constraint: the user must wait the arrival of the first segment *complete* before starting the display of data. That is, it won't start the display of the video immediately after the reception of the first data packet from the first segment. So we have to wait, at least, the time the first segment is transmitted for start the reproduction of data. See figure 1.4.

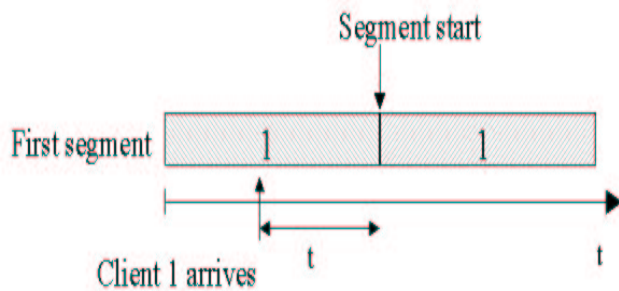


Figure 1.4: A client arrives and starts to store the segment data

We consider the time when a client arrives its **zero time**. Using this convention we can state that a client starts to play the video at time d . A client can receive portions of the segment in any order. In order to guarantee starvation-free playback, segment i has to be completely received by its deadline time, that is, $dead(i)$. Let $dead(i)$ to be the time in which the segment i must be displayed. It will be also the time spent in watching the precedents movie segments (the viewing time of precedent segments plus the initial waiting time d).

$$dead(i) = d + \sum_{j=1}^N b \cdot L_j$$

Every client, then, selects the portions of the transmitted material that are relevant to it at any given time. A client records the selected material and plays it to the viewer at the right time. An NVOD solution thus comprises a transmission scheme, executed by the server, and a corresponding selection algorithm that is executed by each client.

Concerned by the selection algorithm is the next important concept. Known the transmission rate of each segment, and its size, known the deadline time of each of them, the client has to compute the time when it must start to record data. This is a factor dependent of each algorithm, but, in general, we can say that:

$$start(i) = dead(i) - recordTime(i)$$

When we define $start(i)$ as the time when the client must start the data recording, and $recordTime(i)$ the time a client spends on recording the i -th segment.

Table 1.2 shows the most important parameters in a Nvod system.

1.4 Objectives and system requirements

The requirements concerned by this work are the next.

The system must be a Java application dealing with the task of receiving the different segments of data, coming on different multicast channels sent by the server in accordance with an open-loop Nvod algorithm. The application must display the segments (of a movie) in the right order and at the right moment. It must be able to support functions like “Pause” and “Resume”, and possibly, in the future, interactive functions like “Fast-Forward” or “Rewind”.

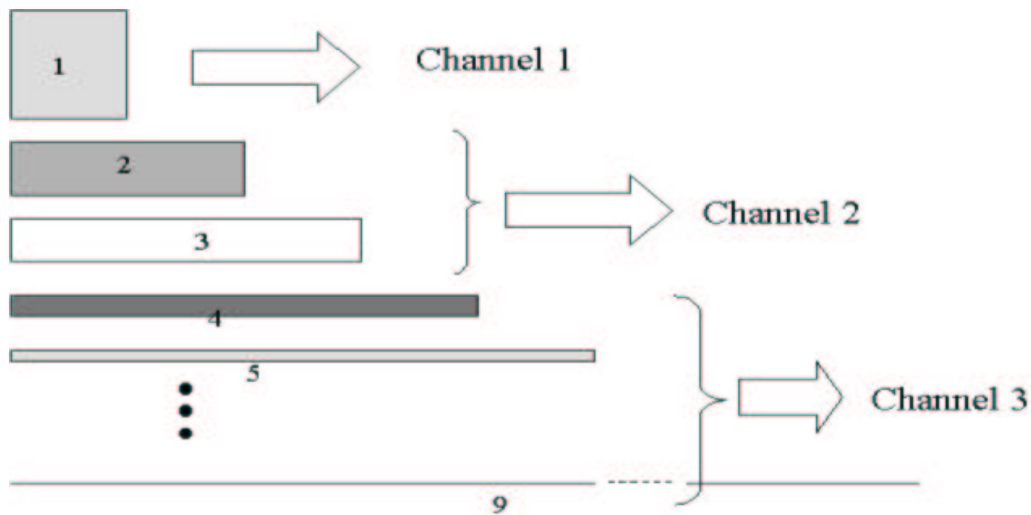


Figure 1.5: Segment multiplexing in multicast channels

As we have just said, we want the system to act in accordance with an open-loop NVOD algorithm. This is due to the desired application extensibility. But our work must be concerned above all by the Birk algorithm who, as we have already emphasized, is a good generalization of many of the NVOD open-loop systems.

The system must be implemented thinking in a future extension in order to recover lost data.

A *configuration channel* will be the medium by which the client will know the available titles, the algorithm used, and the relevant transmission parameters (L , number of segments, transmission rate, d , etc.)

The system requires the possible display at any time of the current system statistics. These include the history (up to the moment these statistics are required) of buffer occupation, recording rate and the number of lost, duplicated and disordered received packets.

Moreover, it must support future extensibility to store the information of reception times, etc.

It's very important for the future of system to do the application **easily extensible, scalable and portable**. The use of **Object-Oriented modelling** and the selected programming language will help us.

Central to any effective periodic delivery scheme is the ability of the client to simultaneously listen to multiple transmission channels and to store frames ahead of their playback times. Periodic broadcast operates well within the buffer space and I/O bandwidth.

The reception of different video frames by the client can be achieved in several ways. In our approach, the server transmits video frames on various multicast channels, with clients joining and leaving the groups to receive the appropriate

Name	Parameter
d	Start-up delay (sec)
R_t	Server-Network transmission bandwidth (bps)
R_r	Maximum instantaneous client reception bandwidth (bps)
N	Movie segment number
L	Length of the video (sec)
b	Film consumption rate (bps)
S_{max}	Maximum instantaneous buffer occupation size (bytes)
$start(i)$	Record starting time for i-th segment
$dead(i)$	Deadline time for i-th segment

Table 1.2: The most important parameters in a NVOD system

frames.

We'll consider a film of duration L sec. transmitted by a video-server on K multicast channels. The film is split in N segments and $n(k)$ segments are transmitted on each channel. Of course, the sum of segments multiplexed in all the channels is the total number of segments (1.1).

$$N = \sum_{k \in K} n(k) \quad (1.1)$$

The distribution of $n(k)$ is in such a way that on each multicast channel we have the same (or similar) $R_t(k)$, and ,therefore, it depends on the specific algorithm used.

The server *continuously* transmits the segment i at rate r_i . Therefore, when we say $n(k)$ segments on each channel, we are talking about simple multiplexing (1.2).

$$R_t = \sum_{i=1}^N r_i \quad (1.2)$$

The server chooses these segment rates and the segment sizes according to a server transmission schedule which is dependent of the algorithm used.

The client receives the video according to a client reception schedule which specifies the time interval over which it must retrieve each segment.

It's time now to talk about the basic advantages of using NVOD in front of the use of true-VOD. The basic improvement is the possibility of having infinite number of clients with bounded (and constant) bandwidth requirements. The trade-off is the excessive need of that bandwidth in presence of little number of clients. It's, thus, thought for hot movies with the certainty of a large number of clients.

The scalability in the number of clients is due to the isolation of server and the clients. Such is achieved due to open-loop scheme.

The client complexity is another drawback of NVOD, but the hardware requirements for these (disk space, I/O bandwidth) are usual for new PC's.

Chapter 2

System design

We have mentioned the main goals of the system design we have to accomplish. We'll use the Object Oriented modelling for our system. It's important to remember the definition that Booch makes about it (see [2]):

Object-oriented design is a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design.

We will try to state other important term we have pointed out in our introduction, modularity (see [2]):

Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.

Obviously, we'll have other important properties of Object Oriented systems assured by choosing correctly the programming language.

But, the most important task of the designer, or one of the most important, precede the implementation phase and is independent of the programming language chosen, for us the Java programming language.

We refer to the terms stated above, the correct design, that is the correct modelling of the different systems behaviors.

First of all, we next present the scenario our application will find. We'll try to search and describe the different main tasks it has to do. And, after that, we'll describe the proposed module scheme for this work, describing the task assigned to each of them, and the way they relate with each other.

2.1 Transmission scenario

Figure 2.1 shows the scenario we can find in our application, at high level.

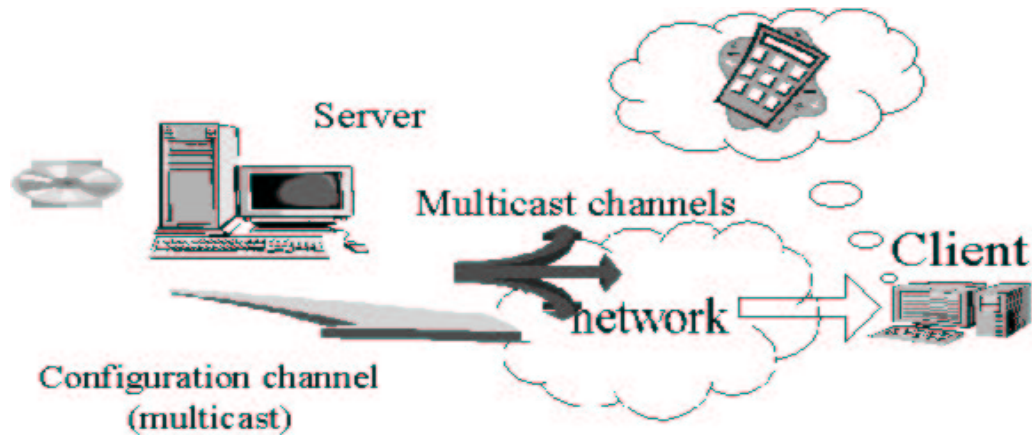


Figure 2.1: Typical transmission scenario

That is formed by three elements that will interact in order to achieve the transmission and display of the movie in the user side. These elements are the server, the network and the client.

The server will split the film data into several segments that will be sent through multicast channels in a datagram-based transmission fashion to the possible end users. Multicast transmission makes possible the independency of the number of clients for bandwidth requirements. Moreover, on these multicast channels we have probably several segments multiplexed. For multicast reference, and more information about multicast related protocols (see [5]).

For channel number saving we will do multiplexing in a fashion that let us to have approximately the same transmission bandwidth on each channel used. For instance, in our Birk case, in which we have an harmonic fashion of assigning the transmission rate at each segment, we will send these segments multiplexed in order to make a good use of the number of channels. That is, we transmit the first segment in one channel, the second and third in another channel, etc. See figure 1.5.

- channel 1 bandwidth: $R_{tx,1} = b$

- channel 2 bandwidth: $R_{tx,2} + R_{tx,3} = \frac{b}{2} + \frac{R}{3} \approx b$
- channel 3 bandwidth: $R_{tx,4} + \dots + R_{tx,9} \approx b$

So, we can avoid channel number explosion when transmitted film is long or the desired start up delay is low. For more information about Birk algorithm see appendix A or [1].

This film data is continuously sent by the server independently of user actions or network needs. So, the server has a very little complexity from a conceptual point of view. To avoid the need of an information exchange between the client and the server to know the server transmission characteristics, the film information, like size, codification type, Near-VOD algorithm used, etc., and the addresses of data multicast channels, the server side will also provide another multicast information channel. This channel is named *Configuration channel* and it will transmit the only information the client needs to be able to join the system. While the server does continuously this work, different clients will join and leave the system. When a client wants to join the system, it will listen from the configuration channel to receive the list of available films provided for the server. When the user choose one of them, the information about this film is also extracted from the communication channel. Once the film information is extracted and received from the configuration channel, the client side is ready for starting its work.

First of all, it computes all the parameters it needs for developing the reception schedule corresponding with the transmission schedule the server uses. This is composed mainly by the group of times it must start to record the different segments in which the movie is split, and the times it must start the display of each of them. When it has computed these times, and therefore, it knows what segments it must listen at which times, the client can start the recording from the multicast channels, whose addresses it have through the configuration channel information, and afterwards the display of the movie.

By the multicast UDP transmission, several clients can join the system without interference among them. They just have to receive data from the different channels the server offers in order to have the segment completely received at its deadline. The client knows how to receive data by the NVOD algorithm implementation we try to do in this work.

2.2 Module architecture

We have proposed a typical scenario the system has to manage. It's time now to translate the language we have just used in tasks the system has to handle for

successfully achieve our objective. We next present also the module architecture the system will have because it answers to the need of managing these tasks.

Figure 2.2 shows the general architecture of our system, and we can see we will have four main modules, with one of them sub-divided in six sub-modules. Each of them will be in charge of one of the tasks in which we will divide the problem. Above all, we'll put the main strength of the system in the `buffer-scheduler` module. It will be decomposed in six sub-modules. Each one will have the aim of managing one of the task the application has to handle. The other modules will have less importance for the developing of NVOD reception, but they'll serve us to manage the application initialization and close, the interface with the user, etc.

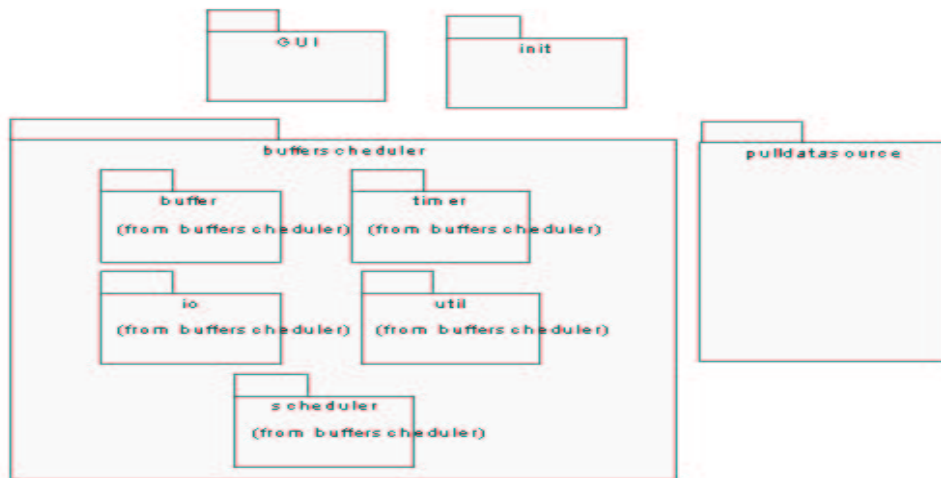


Figure 2.2: Main system modules

We have (figure 2.3) the Graphic User Interface (GUI) to interact with the user and the `init` module where we'll have all the classes having the role of instantiate the NVOD-related objects (those that are in the `buffer_scheduler` module), initialize them, and finalize them when necessary. That is, when the application is started or finished, or when the film chosen is changed by the user. It's important to emphasize that the `init` module is charged also of communication between the requests received from the user (through the GUI) and the `buffer_scheduler` module. That is, it will centralize the information control coming from the user that is addressed to the `buffer_scheduler` module.

For the most important module of the client (figure 2.4) we let the essential system tasks. As other systems based on server-client transmission, we have to

receive and manage the reception of data. In our case, we have also to put the data, possibly arrived in a disordered fashion, in such a way we can display it without discontinuities. Moreover, we have to display it for the user, and we must take care of errors, packet delays, packet duplication, and, as a system requirement, we have to collect several statistics of the transmission.

For implementation reasons, it will appear another module called `pull-datasource` that is in relation with the particular solution for the movie display on the GUI. It is justified by the need of using **JMF API** for Java (see [9]).

2.3 Main system tasks and characteristics

We have just described the main tasks, in high level terms, we need to do for the correct execution of our system. We have also assigned basically the correspondence between the different modules we will have and these tasks. It's time now to explain what classes we'll have, understand more extensively the object functionality in these modules, and sketch the relations among them. We will try also to make clear other important characteristics the applications must have, and must be able to manage, as, for instance, what classes will be charged of receiving the configuration channel data, etc.



Figure 2.3: GUI and `init` modules tasks

2.3.1 Configuration channel

Through the configuration channel, as we have seen when we explained the scenario, the server is sending periodically the list of available films with the information needed for each of them. To abstract this, we include a new class called `VideoObject` whose instance will permit us to know the different main parameters associated with a film. This parameters will be the film name, size, codification rate, NVOD-algorithm used, addresses of channels by which the data is going to be transmitted, etc. See figure 2.5.

Therefore, the server will send a group of objects that form the list of films periodically, but not continuously. That is, it will send the complete list of `VideoObject` instances, for example, each second. But it's not necessary to send, like the segment data in our Birk case (see appendix A), the list each time we have finished of sending it. This send schedule will do the configuration channel will not consume very much bandwidth for transmission purposes. But, in contrast, we have to add the average delay of obtaining the list of films and the data associated to the delay the clients suffer when they want to join our service.

The configuration channel is sent in a multicast fashion and, for that, we can exploit the same advantages that we did with the segment data channels. That is,

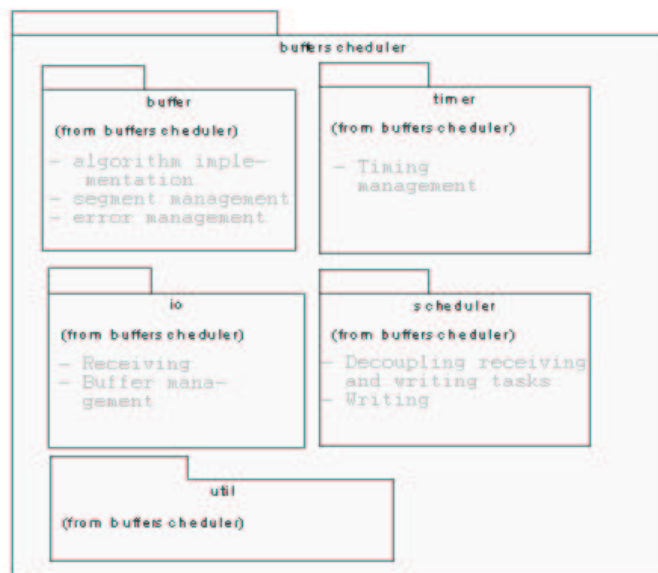


Figure 2.4: bufferscheduler module tasks

we only have to provide the information through one configuration channel and all the users can to receive the information without interfering the transmission and without the need of increasing the bandwidth used for it. Moreover, these configuration channel, that is, its address, is the only information the potential client must know before the connection to the system. When it has contacted, and has received the information about the films and their characteristics, it is able to contact the segment data channel thanks to this information.

In order to manage the task of receiving the configuration channel data, that is, the `VideoObject` class instances, we will introduce a class named `ListLoader`. This class, that is, the class instances, will permit us to download from the configuration channel these `VideoObject` instances and to store them. When the user has chosen one of the titles the server provides, titles that are provided by the `ListLoader` object when it has received all the list objects and has extracted the film name of each of them, the `ListLoader` is requested for returning all the data concerning the chosen film.

2.3.2 Algorithm implementation

An important part of the application is the implementation of NVOD algorithm characteristics. In a Object Oriented design and programming, we'll do so abstracting these characteristics by a class. But we want also the possibility of easy

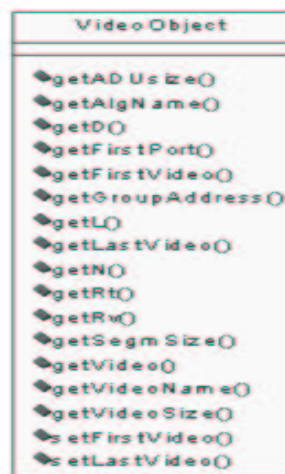


Figure 2.5: `VideoObject` class

extensibility for our application. That is, we will implement the Birk algorithm, but we want it not to be difficult to extend the application to work with, for instance, the Pagoda's algorithm [11]. We next explain how we do this.

The first thing we remark while we try to solve the problem is that, in all the NVOD open-loop schemes we have some common characteristics. Like, for instance, all of them will send the data with a kind of segmentation, or they will need some of the parameters we know by the `VideoObject` instances that we receive through the configuration channel, etc. Moreover, we know they must, independently of their nature, provide us some information, common for all of them, but characteristic of each of them. We talk, for example, about the form the algorithm computes the times it must start to record each segment. This calculation is different for each algorithm, but all of them have to do it.

So we are going to introduce a new class named `VODScheduler` that will give us a kind of abstraction for all algorithms we want to implement. We will declare this class like `abstract`, that is, we won't be able to instantiate it. Within this class, we try to enclose all this common information belonging to all the possible NVOD open-loop schemes. We also try to define a contract for other classes know what they can expect from the object implementing the algorithm that governs the application behavior.

The key of this design is the inheritance we must do to instantiate the `VODScheduler` class. That is, we create another class that inherits from the `VODScheduler` class and that encloses all the information typical of each algorithm (see figure 2.6).

In our case, the class that inherits from our `abstract` class is the `Birk` class, that has all the characteristics, methods and attributes the `VODScheduler` class has. It implements also, that is, gives the sense to, the methods that form the contract, because these methods are dependents of the algorithm used.

We do so in order to achieve the desired extensibility for our system. And to extend it, we just have to inherit another class from `VODScheduler` and give sense to the `abstract` methods we have defined in it, in accordance with the algorithm we want to implement.

2.3.3 Segment management

When the user has chosen the desired film, and the application has got the information concerning it, we can start the reception of the film. In the `VODScheduler` object we have enclosed all the information film-specific. That is, the information about the film itself, and about the NVOD algorithm used for broadcasting the film. With that, we can obtain the group of times that give us the times we

must start the recording and storing of data and the group of times in which each segment has to be displayed. Figure 2.7 shows how the objects of different classes interact for this task.

When the `VODScheduler` object knows all that information, it can send it to an `Agenda`.

Agenda

This class belongs to the `bufferscheduler.timer` submodule. It extends the Java class `java.lang.Thread`. It will give us the complete functionality of its module, and therefore, will be charged of system *timing*. Objects from this class will have a list of events associated with a time. When one time expires, `Agenda` will send a message to the *event listener*. The `Agenda` instance won't worry about the action this *event listener* will do, but this action will be done in the *Agenda's thread*. We'll have three kinds of events. Actually, they'll be a general event and two special cases.

These two cases are special because they are always present in the system. They are *deadline* events and *time to record* events. In both cases, the *event listener* will be a `ChannelManager` instance associated to each `Agenda` instance.

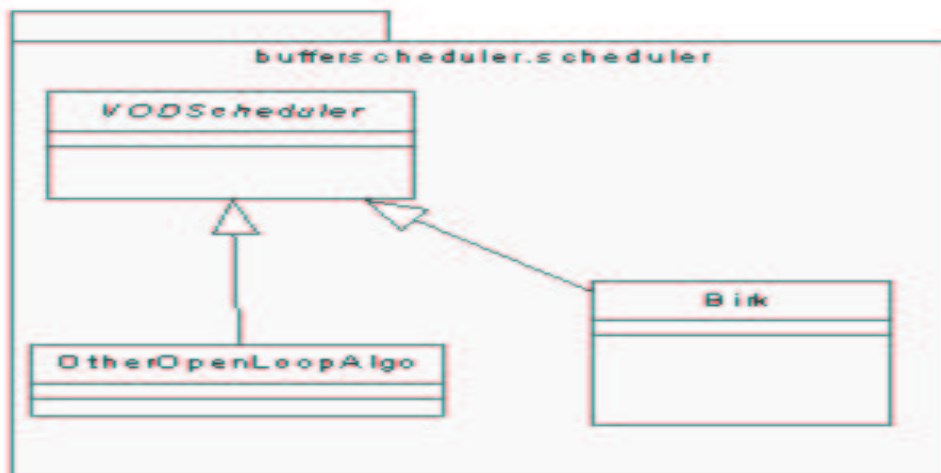


Figure 2.6: Algorithm implementation by inheritance

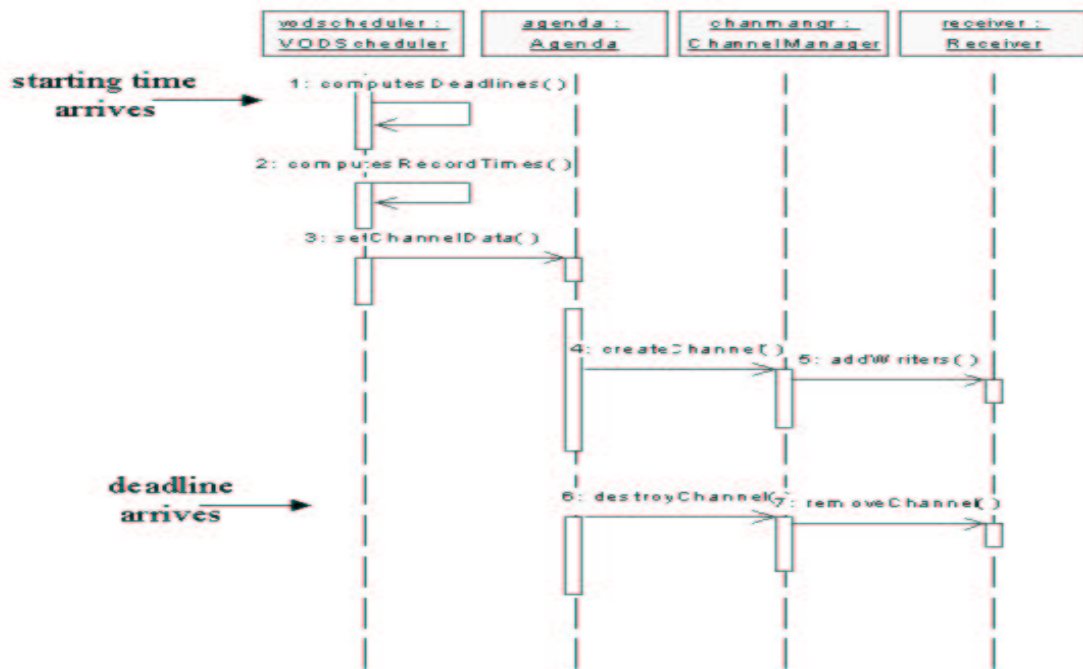


Figure 2.7: Segment management interaction diagram

ChannelManager

This class belongs to the `bufferscheduler.scheduler` submodule. It will have the role of the segment recording management and it will act as an information point for other objects like `Receiver` instances. It will mainly receive the message from its associate `Agenda` object for it to start the recording of a segment (*time to record*) or for it to start the display of a segment (*deadline*). So it must manage the creation and destruction of instances of another class: `Receiver` that is placed in the `bufferscheduler.io` submodule.

Receiver

That is due to the role we have reserved for this class. `Receiver` inherits also from `java.lang.Thread`, and it will be charged of two very important tasks in the application. Maybe the most important ones. It must *receive* the data from the UDP packets that server sends through multicast channels, and it must manage the data ordering within the buffer. But these two important tasks are described in the Section 2.3.4.

Now we emphasize the conceptual idea for `Receiver` objects. Each `Receiver` instance will be charged of the reception from one multicast channel. That is, for each multicast channel, there is one UDP receiver socket, and one `Receiver` instance. By the way, we describe other class closely related with the `Receiver` class and with the UDP sockets each instance of this class have. We talk about the `bufferscheduler.io.SocketFactory` that provides us multicast sockets for the reception.

It's important to make clear that each `Receiver` is charged of one *multicast channel*, that is, possibly of several *segments*. And for updating the number of segments associated with a `Receiver` object we have *active*, this class provides to the `ChannelManager` object two methods. One to add a segment as “active” and other to remove them. With *active* we mean a segment that is being recorded. A segment is not being recorded if its packets are being discarded for the `Receiver` for algorithm-dependent reasons.

2.3.4 Reception

Reception is maybe the most important of the tasks the client side must accomplish. As we have already mentioned, we will have `Receiver` instances charged of this work. Each of these instances will receive UDP data packets from a multicast channel. For that, each of these instances will receive the data from several

segments.

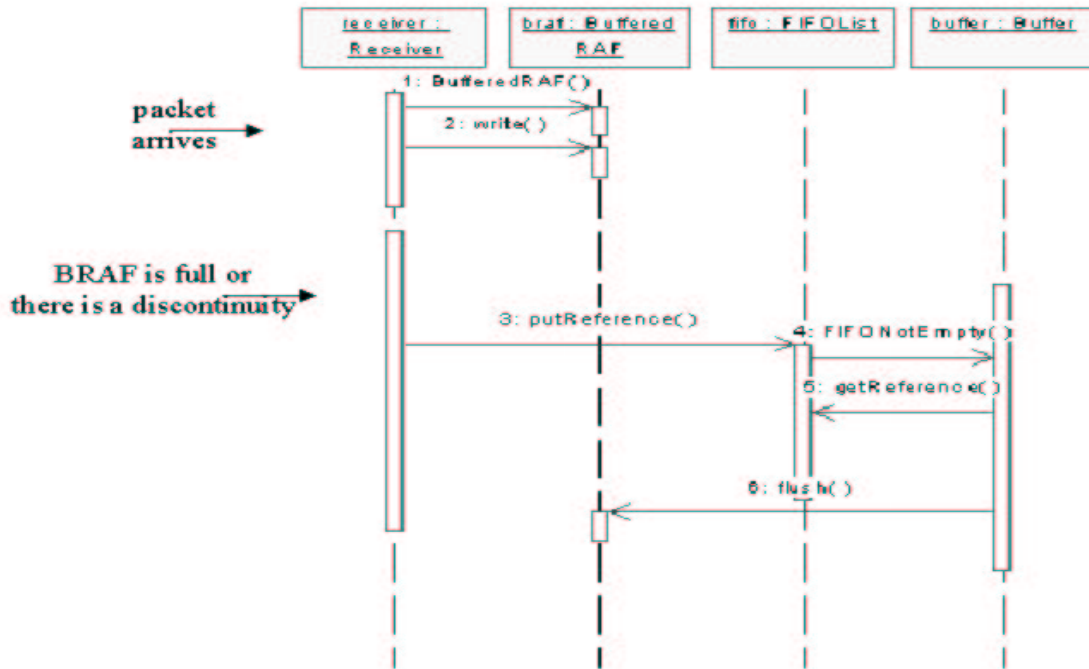


Figure 2.8: Interaction diagram for reception

We next describe how a `Receiver` object can distinguish among the, possibly several, segments assigned to it. In figure 2.8 we can see how objects are related in order to accomplish this task.

Application Data Unit

For this purpose we define the **Application Data Unit (ADU)** as the minimum data unit for our application. Each UDP packet is formed by an UDP header and an ADU. Each ADU is formed by an ADU header and ADU utile data. For us, the ADU header is the way by which we can order the data inside a segment, or we can decide at which segment the data belongs to, etc. Figure 2.9 shows the ADU header. Segment number permit us to classify the data in function of the segment it belongs. Packet number is used for ordering the, possibly disordered, data that arrives through the multicast channel. Remember that we use UDP transmission and it's not sure that the packets arrive in the same order they were sent. Size is

the size in bytes of ADU utile data. And last packet is a boolean (one byte) to know if the packet is the last of the segment. All the other header fields have a 4-byte size.

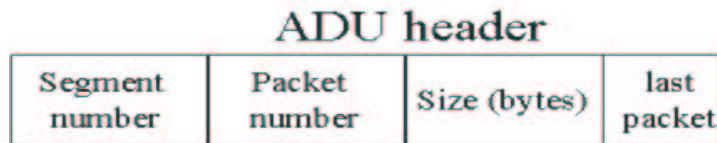


Figure 2.9: Application Data Unit format

BufferedRandomAccessFile

When a `Receiver` instance gets a UDP packet from the multicast channel it looks up the `segment number` field from the ADU header in order to know if the packet belongs to an active segment. Obviously, the segment must be assigned to this `Receiver` object because all the segments multiplexed in a multicast channel are assigned to the `Receiver` that listens the socket bound to that channel. Then, the only consideration to do is if we need to store the data from this segment. That is, if, for this segment, the *time to record* has already arrived, and the *deadline* has not.

If the packet belongs to an active segment, the `Receiver` instance processes it. If the packet is not a duplicate one, the `Receiver` must write packet data in the correct position within the buffer.

We will use a file to store the film data. And due to the film size we must do it in a peripheral device for storing, like a hard disk, etc. This push us to don't allow `Receiver` instances to write directly to these kind of devices because they are very timing consuming. Then we introduce a new class: `BufferedRandomAccessFile` (`BufferedRAF`).

We will instantiate this class for buffering the data before writing in, for instance, a hard disk. Moreover, these instances will permit us to do loosely coupled the task of receiving and writing.

The `BufferedRAF` has, inherited from `java.io.RandomAccessFile`, a pointer to the file in which it writes. It has also an internal buffer where the `Receiver` makes the operation of writing. This buffer is placed in RAM memory and the writing methods over it are not so time consumptive as the ones over the hard disk.

A `Receiver` object will have an instance of `BufferedRAF` for each segment it manages at any given time. But it will have several of them, for the same segment, along the life of the application. When a `BufferedRAF` is full or there is a discontinuity in the segment recording, for instance when a packet arrives in a disordered fashion, the `Receiver` object sends the `BufferedRAF` reference to the `Buffer` object the application has. Afterwards the `Receiver` object creates a new `BufferedRAF` and continues with the reception.

It's important to note that the management of the file, that is, of the buffer where we store the data, is responsibility of the `Receiver` class. `Receiver` objects are charged of managing the pointers of their `BufferedRAF` objects for the writing be correct.

Buffer

In the `bufferscheduler.buffer` submodule we find that `Buffer`.

Very related to `Buffer` class we have also the `FIFOList` class. We will have one instance of each of them in the application. Their task will be to discharge `Receiver` instances of the I/O device writing time-consuming work.

`Buffer` objects will be executed in their own thread, they inherit from the `java.lang.Thread` class, and the `BufferedRAF` reference passing is done through the `FIFOList` instance. Then the `Buffer` object simply calls the `BufferedRAF` method `flush()`. This method writes the `BufferedRAF` internal buffer in the file at the position that its pointer give us.

The `FIFOList` object allows us to avoid long time waits in the relation between the `Receiver` objects and the `Buffer`.

Data store

We have mentioned we will do the store of data in a file. For that, as `java.io.RandomAccessFile` needs, we use a `java.io.File` object to represent the operative system file. It's not easy to manage the size of these kind of files in Java. We can easily erase data in a file but is very possible that the size doesn't change. That is, we can delete data, but it will not save disk space.

A variable in the NVOD system Birk algorithm represents the buffer occupation. This is introduced to try to reduce the user memory need. In our application, we will not worry about this. We think the client has enough space to store all the film. And, for that, we will not delete the data we have already displayed for the user. This will, maybe, help us to implement easily functions as *Rewind*, *FB*, *SB*, *JB*.

Anyway, in order to correctly test the Birk algorithm, we will simulate the deletion of the data how Birk method says. That is, for statistical considerations, the buffer occupation simulates the data is removed. Moreover, if we simulate that data is deleted for any reason before it has been displayed, it must be received and written another time, like it would never been received, as the Birk algorithm proposes. The only difference is that the buffer needs of the application are not at all minimal.

2.3.5 Statistics

As we have said in the introduction, we are primary interested in two parameters, though we can obtain the desired extensibility also in this domain. But, for the moment, the application is interested in collecting statistics concerning the **buffer occupation** and the **reception rate**. Then, our application will allow us to know the evolution of the **aggregate buffer occupation**, that is the occupation due to all segments, during the application execution. Moreover, it will allow us the same about the **aggregate reception rate**, that is, the sum of the reception rates of all segments. We desire also to have the possibility of knowing the number of packets arrived in a disordered fashion, lost packets and duplicated ones. Lastly, we would wish to know the **effective segment reception rate** the client “sees” for each segment received, for comparison purposes with the effective segment transmission rate used by the server. Another desired system characteristic is the possibility of displaying the statistics in a user-demand fashion, that is, at any time, either during the application running, or afterwards.

To collect the statistics in the client-side of the system we will introduce two classes. One of them, `HeaderReceiver`, placed in the `bufferscheduler.util` submodule, will know at a given time of the application execution, the current statistics. Objects of this class will have a group of methods to let instances of other classes to know at any time these collected statistics.

This work is thought for the `BufferHistory` instances. These objects belonging to this class, placed in the `bufferscheduler.buffer` submodule, will have the task of storing execution statistics from the `HeaderReceiver` object. That is, periodically, a `BufferHistory` object will request the current information about the statistics `HeaderReceiver` object has. It will do this through a simple polling each second, by default, and storing the information for future displaying at user demand. Object instantiated from this class will be executed in their own thread.

This two classes try to make loosely coupled the task of collecting the application execution statistics and the one of having updated this statistics.

Figure 2.10 shows the interaction diagram for these classes. When a Receiver object receives a UDP packet, it sends the ADU header along with a time stamp representing the time it has received the packet to the HeaderReceiver object.

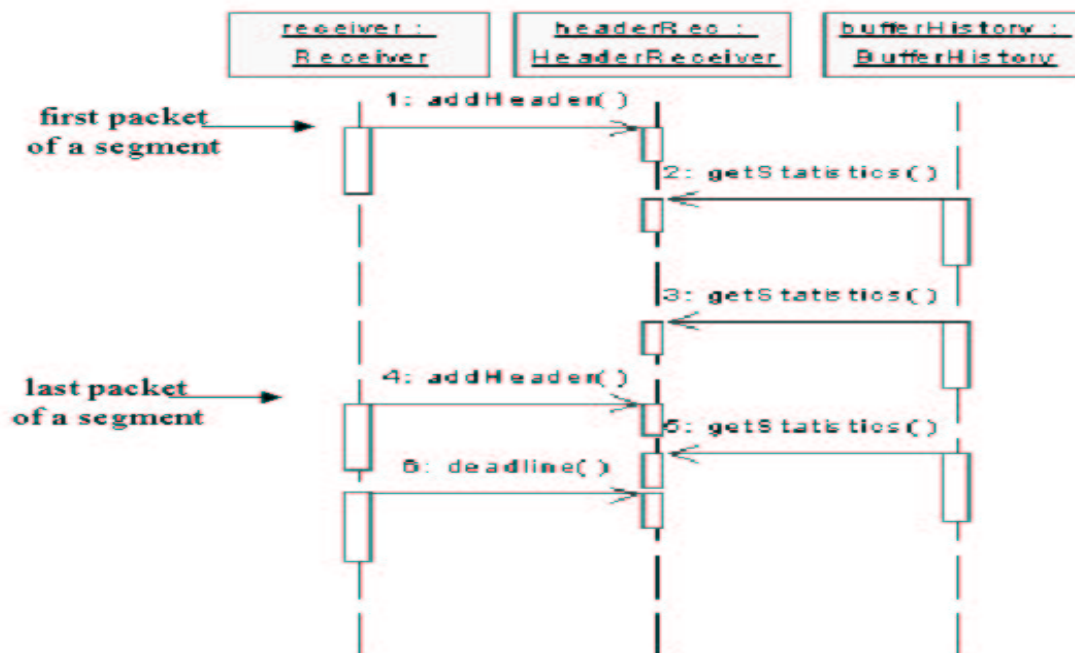


Figure 2.10: Interaction diagram for the statistic collecting

It's possible that a `Receiver` instance won't process the packet received due to that the segment it belongs to, is not active. To `HeaderReceiver` instance could have enough information to manage the statistics, it will be provided also a `boolean` value expressing if the packet has been processed or not. For statistics management purposes, `Receiver` objects will have to "say" the `HeaderReceiver` when a segment is already completely received. These instances will also send the information to `HeaderReceiver` object about lost, disordered and duplicated packets.

To have the possibility of knowing other kind of statistics out of the scope of present application without need of extending the present work we offer an off-line alternative. The `HeaderReceiver` object will create a file where it will store all the header information it receives from the `Receiver` objects. This information

is written in binary mode, and to easily read it in text mode, we provide another class called `HeaderReceiverReader`.

2.3.6 Display

We face now the problem of displaying the movie on the screen we have in the Graphic User Interface. To do this, we have various constraints to analyze.

We will use the capabilities Java Media Framework API for Java (see [9]) gives us for multimedia data processing. This JMF API form a group of Java classes and libraries supporting media data streaming and decoding. JMF is an API that allows us to display and catch the multimedia data from an application. This encloses the playback of compressed video over different formats, like MPEG.

Usually, JMF uses a static file like source from which its classes keep the multimedia data by the use of predefined protocols like `http`, `file`, or `rtp`. In our case we have a group of logical channels through which we receive the raw data while we are displaying the already received segments.

However this Java API give us the possibility to extend its classes and to implement its interfaces to build our custom protocols. That's we will do.

As long as data comes to the client by UDP datagrams through UDP sockets, we have to choose between two possibilities. We can see the data arrival like a data stream we have to display or like a group of data chunks we will have completely received at the moment we will need them.

Nevertheless, if we choose the former option, we will have to force the JMF classes to see the multiple channels with their multiple segments (say "logical subchannels") like a uniform data streaming. It will force us to include a kind of "converter" from sub-segmented data to a data flow. That is like include a relation server-client (of multimedia data) in the application-client side, where the server is who "converts" the segmented data in a data flow.

With the second option, we can take advantage of the kind of storing we have. As we have already explained we use a `java.io.File` for that. The solution would entail the presentation of this `File` we use for storing data like the "static file" the JMF use to have like multimedia data source.

Although the file is not complete when we start the playback, we can, by using our custom JMF classes, simulate the file is completely stored. The only constraint in that sense is that we must have available the data whenever the JMF classes needs it. But that is assured for the correct development of the application because the JMF objects will need the given segment data at its segment *deadline*. And the aim of all the application is, in a great part, to provide the segment data

before the segment *deadline*. Then, if the rest of application is correctly executed, the display must be successful.

We will consider, for possible future “Rewind” implementations, and helped by the new technologies in PC’s fabrication, that the client will have a very large store space. As large as to be able to store the complete movie, about 1 GB.

This will help us to choose the second option. We will explain why. First of all, in contrast with true VOD techniques, data doesn’t come to us in an ordered fashion, (remember we can receive first the data of the end of the segment), and for that, choosing the former option implies a first storage in, for instance, file format, and, afterwards, a reading from this file to “emulate” a streaming coming.

Other reason is that one tendency of NVOD algorithms is to reduce the buffer the client needs for displaying the movie. Also, one of the comparator factors among them is the client’s buffer occupation. Even, in Birk algorithm, is strongly emphasized how to reduce this occupation.

However, even when we will compute the theoretical Birk buffer occupation, we will not remove data from the memory. Neither when data is already displayed (for “Rewind” purposes), nor when the algorithm says that it must remove data for buffer saving purposes.

If we use just one file for all the data we’ll have possibly a 1 GB file in our memory device, but, is tested that the displaying performance is better than with more than one. The first attempt was to record all the segments in separated files. So we could remove each file after the consumption of its media data, freeing the buffer. However, this solution presented problems in data displaying, due to the jumps the JMF custom classes had to do to “emulate” the file storage uniformity. This kind of playback behavior has lead us to the use of an only file for all the movie. For more information about reception see 2.3.4 section.

We will implement our custom `DataSource` class that will implement itself the `javax.media.protocol.PullDataSource` interface. That’s is due to the use of MPEG codification. In the JMF API it’s possible to find information about the need of using this kind of `DataSource` for this kind of media data (see [8]).

We have other possibility, to implement the `javax.media.protocol.PushDataSource`. The difference between `Push-` and `Pull-` `DataSources` is who starts and manages the transaction of data, the client or the server. We must say that, when we refer in this JMF scope to server and client, it isn’t at application level that we talk, but at the JMF media data transaction level. That is, the server is the element makes us available the multimedia data, in our case the client-side of the application charged of receiving the data and storing them. Our JMF-client is the class containing the `javax.media.Player`, who will display the movie

to the user.

In a `PullDataSource`, it's the JMF-client who manages the transaction. That is, is the JMF-clients who "takes" the data from the JMF-server. If a `PushDataSource` is implemented, is the JMF-server who "gives" the data to the JMF-client.

As we have referred, with MPEG coding, we have to use `PullDataSource`. Other examples of `PullDataSource` based protocols are the http and file, where the data is stored and the client "takes" the data when it needs. An example of `PushDataSource` use is a possible application that receives multimedia data from Internet in a live session. The data is delivered as soon as it arrives, and thus is the JMF-server, the element that receives the data, who must manage the data transaction, making a best-effort for playback continuity.

Other restriction for choosing the JMF classes and interfaces to use is caused by the MPEG coding itself. In MPEG coding the frames are correlated, that is, there are frames coded in function of the last or of the next coded-frame. That makes impossible the use of a kind of JMF streams called `Push/Pull-BufferSourceStream`. The `SourceStream` classes are the media JMF uses for communicate the source of multimedia data (represented by the `DataSource`) and the `javax.media.Player`, who will be in charge of rendering and playback the data. The use of buffered `SourceStreams` are based in passing the multimedia data in chunks of one frame. Therefore we will use an extension of `javax.media.protocol.PullSourceStream`.

The last note is about the use of interface `javax.media.Seekable` in the `PullSourceStream` implementation. It's is due to the recommendation of one of the JMF creators (see [13]). It's that who makes not possible the use of, already implemented, file protocol with our system. For MPEG information see [3].

2.3.7 Timeout management

We have created for this application a kind of system of timeout. Why?

In the framework of extensibility we strongly stress from the start of this work, we want the application could be enhanced for *error recovering* supporting. That is, we want it to be extensible to know when the system has suffered a data lost, and to take the correct decision in order to recover this lack of data.

For that is a priority for us to know when we can state that we have suffered a lost of data. As we use UDP based transmission for the communication, the packets can suffer different delays. These delays can make that we see packets arriving in a not expected fashion. For example, a disordered packet sequence. Then, when the application notes the lack of a packet, it must wait, during a

prudential amount of time, the delayed arrival of that packet. But it's possible the packet had been lost in the network. The timeout management is done for the application not to wait forever. When a disorder occurs the application starts a counter (timeout) and if it expires, we can consider the packet is lost. Then is possible to take different actions in order to recover the lost data.

In this work we have not accomplished the task of error recovering, but we have wanted to make available an easy future implementation.

So, we have included several classes to accomplish this timeout management, and one to receive the future error management. If error recovering is desired, it will be necessary the implementation of `ErrorManager` class. Obviously, the error recovering implies, not only the implementation of this class and the use of the provided facilities for timeout management, but the extension of the system with the creation of other classes charged of implement the different possible solutions. But all of them would need the timeout management we provide.

As we have explained in the Section 2.3.3, `Agenda` objects can send a message when the time specified for an *event* is expired. There are two kinds of singular events and a general one. In this general event we will base all our timeout management.

The objects that will realize that there have been a disorder in the reception will be the `Receiver` instances. Then, they will be the objects who send to the `Agenda` instance the request of notification for the time the timeout event will expire. As we have said in Section 2.3.3, the `Agenda` objects register times. When these times arrive, `Agenda` notifies it to the time-expired *event listener*. We define an interface named `EventListener` that will be implemented for the object that wants to use this `Agenda` facility. In our case, `Receiver` will implement this interface. When the timeout expires the `Agenda` instance sends a message to the `EventListener` (`Receiver`) and this will do something with the information it has. In our case, the `Receiver` object that has been notified of the expiration of a timeout will notify itself to the `ErrorManager` instance, sending to it the information necessary. If we override the present methods of `ErrorManager` class we can achieve the error management for implementing the error recovering.

It is possible that a `Receiver` object wants a timeout notification for a packet has not arrived yet and, before the timeout expiration, the packet comes. Then the requested event has no more sense. In that case, the `Agenda` class provides methods for asking it not to notify for such a event.

We can see an example of object interaction in figure 2.11.

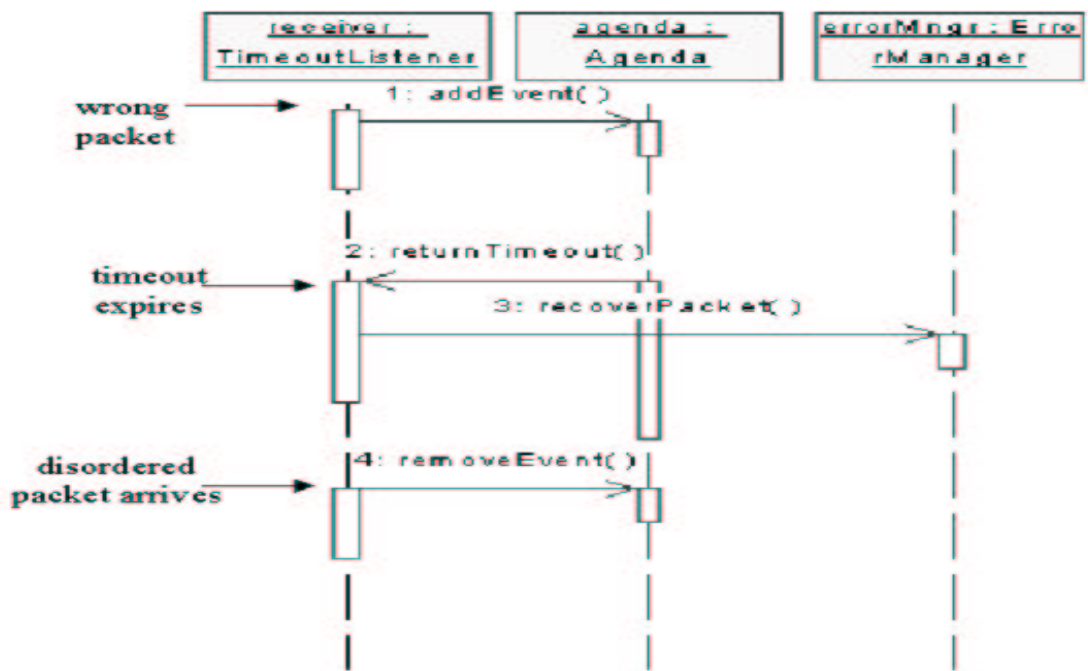


Figure 2.11: Interaction diagram for the timeout management

2.3.8 Application initialization and session management

In Section 2.2 we have talked about the `init` module responsibilities. We next analyze this module.

As we have said this module is charged of initialize the application and initializing properly `bufferscheduler`. It must also be the point of control for the communication between the user (through the `GUI` module) and the NVOD-related classes we have in `buffer scheduler` module. An instance of `init` module called `Deliver` will be charged of these two tasks.

Deliver

System `Deliver` instance is the highest point of control in our application. It must instantiate, initialize all the classes the application will use and to destroy the created instances. Moreover, it must translate the user requests it receives from the `GUI` module in calls to class methods in the application. It must also, for example, activate and deactivate the use of some buttons depending on the state the system has reached.

We have just said that a `Deliver` object must instantiate all the classes the application will use. But it's not true. Several of this classes will instantiate them by *delegation* in other objects from the `Session` class. Strictly speaking, a `Deliver` instance will only matter of classes whose objects will be the same along the application execution. This classes are, for example, `Agenda`, `ChannelManager`, `Buffer`, `FIFOList`, etc.

But for the objects movie-information dependents, or for those who are specific for a movie and not reusable, as the `File` we are going to write on the movie data, we introduce the `Session` class.

Session

With session we want to talk about the events, actions and objects that are involved in the movie display for the client. Then, we will define as `Session` object tasks the instantiation of classes dependents of the particular movie we are going to display. For example, the `VODScheduler` object is dependent of the movie, because, in general, we can have different movies multicasted with different algorithms. So we can't instantiate a particular `VODScheduler` inherited class without this knowledge.

Like the film dependent information is extracted when the user chooses a film among the movies of the list, we will create a `Session` object each time the user

do it. Of course, when a `Session` is opened, that is, an instance is created, we'll destroy the last if there was one.

When a new session is started, it's possible that some persistent objects along the application life time, like `Agenda`, etc., need to be restarted. That is, their state must be returned to a kind of initial state. That is a work for the `Deliver` object.

Chapter 3

Results

In order to test the application concerning this work we have done it to work in a real laboratory network. For that we have used the internal network implemented in the Institute Eurécom (Sophia-Antipolis, France). This switched network has a 100 Mbps capacity and it's used by about 50 persons.

After we have demonstrate its correct working, we have analyzed the highest capabilities this application can reach. For that, we have used our application running on the Eurécom's laboratory machines.

For the experiments we have used one machine for server tasks, that only had the aim of sending the data through the network up to all the terminals present in it. The sending has been made in multicast fashion. Data is the configuration data sent through the configuration channel, and the film data sent through the multicast data channels. We have made all the tests with the sending of one film each time. That was in order not to waste the network bandwidth. It's important to say that in our work we have focused our attention on the client application only. It's for that we have not analyzed the server application in the present work.

Other machine in the network has been the host for our application. It's important to say that we don't matter about which machine specifically was the host, because the server data was arriving to all the machines in the network. That is due to the multicast sending.

In spite of that, all the tests have been made using the same host machine in order to homogenize the results. Obviously, we can also try to run the application in several machines at the same time. That works fine.

The server is running without interruption, that is, sending the data (film data and configuration data) another time when the last has finished. The client runs the application without any kind of synchronization with the server. It starts to receive data and, after several seconds (the time called *startup delay*), it starts the

film display.

3.1 Result interpretation

We next present a “desirable” real case for using as environment our application has to see. This case will be a kind of model for us in order to interpret the results we have obtained.

In this model case we have the idea of providing a Birk NVOD service with a single film. This film has a duration of 1 hour and half and it’s coded with MPEG-1. We desire the clients won’t have a startup delay greater than five minutes.

A film of one hour and half coded with MPEG-1 has a size of about 1.2 GBytes. If we want a startup delay not greater than five minutes, and knowing that the code rate of MPEG-1 is 1.5 Mbps, we must have segments not greater than 50 MBytes. So, we choose 25 segments of 50 MBytes of size. If we have the segments multiplexed like we have proposed, we’ll need five multicast channels with 1.5 Mbps capacity each of them. We need also the configuration channel, but its capacity is not very important.

To test this “desirable” example we need a great storing capacity in the server. Moreover, we need it also in the client machine due to the kind of data recording we do. In addition, its not trivial to get free films with these sizes. And, if we have found anyone, it had the MPEG header coded or it hadn’t MPEG header. That was a problem because the JMF doesn’t work without the MPEG header.

These reasons lead us to use films with smaller sizes. Sometimes the characteristics weren’t similar to those from the model example, but they were the biggest movies we had found.

Nevertheless we want, if it’s possible, to know the highest limit of our application in a case similar to that we call the model example or model case. For that we next explain how to know the behavior our application would have in that case, knowing the behavior it has in our tests.

For that is essential to understand the next concept: the *effort* the application must do to display the film. With *effort* we mean CPU processing capacity, we mean network I/O bandwidth and storing writing/reading bandwidth. This effort is referred to the resources we need for the correct application work. It’s necessary to remember that MPEG-1 is a codification technic very expensive in terms of CPU, and there will be a great amount of data to be received, recorded, and lastly read.

Let be the model example our present case. We need an amount of resources from the system for the correctly working of the application. To know if we have

that capacity without having a film with these characteristics we must to simulate them.

If we have, for instance, a film with half size, that is 0.6 GBytes, and all the other characteristics equal, we can say the following. We have two obvious possibilities in front of us. One is to reduce the number of segments and to maintain the segment size. The other is to maintain the number of segments and to reduce their size.

With the first option, we have 12 or 13 segments with a segment size of 50 MBytes. In this case we must to maintain also the startup delay the client suffers because the segment length has not changed. We don't need anymore five multicast channels, but only four. Having only 13 segments, in the worst case, we will have to receive, process and record the 13 segments simultaneously. That is a data flux of

$$1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{13}$$

times the first segment transmission rate. This is a best situation that we would have in our model case.

So, in this situation the user visible parameters stay without change with a lower size. As trade-off we find that the need of system resources is not so high and we can save some transmission bandwidth .

In the second situation, in which we reduce the segment size, we can also reduce the client startup delay. In contrast we need always five multicast channels for the data. For that we will receive the same data flux that we would receive in the model case. So, the application must make the same *effort* to manage the data. The only difference is a reduction in the time axe, in the startup delay and in the film length.

With this second option, we can observe how having a film \times times lower than other model film, and if the codification rate is the same for both, and if the number of segments in which we split both films is the same, we demand the same *effort* to the application. The startup delay for the client must be \times times lower.

So, we have tested the application as in the second option to know if it can manage a case like the model example.

3.2 Results

After we have verified the correct working of our application, our next step has been to test it in an environment of maximum efficiency to know how much quickly the application can answer.

For that, we define some minimum events with which we can say that the application works correctly. That is, if the application can receive correctly all the data that arrives through the network and store them while it displays the film in a satisfactory fashion, we can say it works fine.

3.2.1 Transmission scenario

Firstly, we must underline that when we say we have verified the correct application working, we want to say we have tested it with a specific JVM (Java Virtual Machine), for Windows NT, and with JMF (Java Media Framework) API optimized for Windows base operating systems. It's necessary to study more deeply the behavior of the application working over other OS like Solaris, Linux, etc. That is important because one of the aims searched in this work is to construct a *portable* application.

Nevertheless, tests have been done with JVM for Windows because this JVM presents the best skills in relation with the others. That's important because our application need a great synchronization among the different threads there are running inside it.

We must also distinguish between two kinds of storing the data we receive through the network. With one of them we store the data in a virtual file system. That is, we store the data in a remote server and data is sent through the Eurécom's network for that. In the other fashion of storing, we do it in the local hard disk of the machine we are using as host.

In the other hand we have also two cases to analyze our tests when we talk about the transmission quality the client "sees". That is, firstly we analyze the error-free transmission. We can do it because we can assume that the probability of losses in the Eurécom network is zero.

Afterwards we analyze the case with errors. The client will see losses or disorders in the received packet sequence. To achieve that we have forced errors in the transmission deliberately in a deterministic fashion.

We must also underline briefly some other important parameters in the transmission. They have remained constant along the different tests done.

We have sent UDP packets through multicast channels with a UDP data field composed by the ADU header and the ADU data field. The ADU header has a 13 bytes size. The ADU data field has been fixed in 1200 bytes. We must say that, in the next section, we will talk about transmission rates or about data flux in terms of bps, etc. We will be talking about *useful data*, that is, bits or bytes belonging to the data we have in the ADU data field. We won't keep in account neither the ADU header nor the UDP header.

To send data within a desired transmission rate, we have to send a given number of packets per second. It can arrive that we must send a packet each 6.8 seconds to achieve a desired transmission rate. To have more precision, we send the packets in groups of 10 (Figure 3.1).

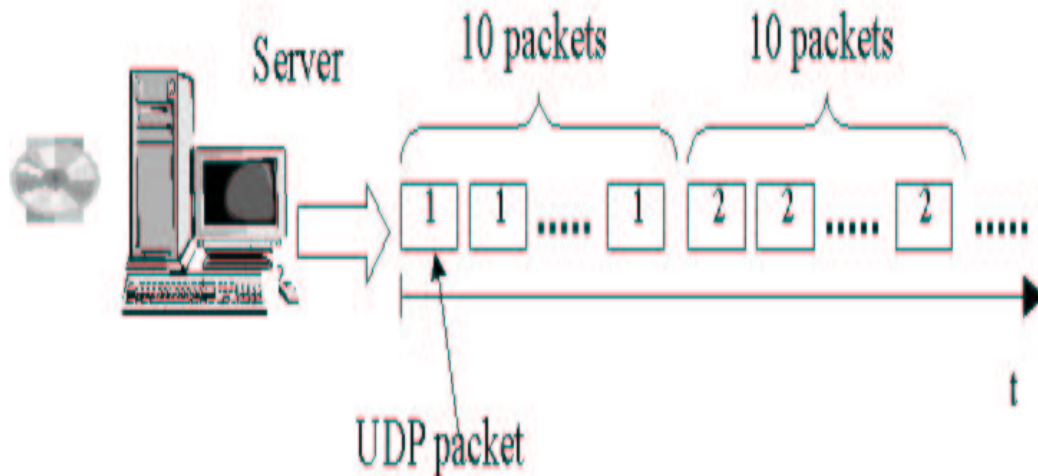


Figure 3.1: Packet sent scheduling to have more precision in the transmission rate

We must note that, even if we will give a transmission rate for each of the tests, this value will be an approximation. The most of the times it will be a minimum edge. That is due to the granularity the Operating System and the Java Virtual Machine give us. That forces us to be a little conservative in order to assure a minimum transmission rate. We need to achieve this minimum to assure the correct application working.

Moreover, the client will not fulfill exactly the time scheduling proposed by the theoretical algorithm. It will add a little margin time to let the application achieve successfully its aim. That is done due to the non-idealities of the system. This margin time will be in the order of one second. This introduction in the system can be tolerated because, usually, the startup delay the client suffers is of an order much higher. Obviously, this margin time will be added to the time the client must wait from the moment it demands the film up to the moment it starts the display.

In the cases we will analyze we talk about a transmission rate or about a transmission bandwidth, or about server/network bandwidth indistinctly. We will refer normally to the transmission rate for the first segment, R_1 , in bps. It's already said

that all the other transmission rates are defined in function of this rate by the way of an harmonic series, see [1].

It's also important to remember that when we will talk about minimum transmission bandwidth, we talk about the case in which the first segment transmission rate R_1 is equal to the film consumption rate b .

Lastly, before to know the results of the tests made, we next give the characteristics for the machine used for running the server application in all the tests.

- machine: loach.eurecom.fr
- Operative System: Windows NT 4.0
- RAM: 128 MBytes
- Processor: Pentium III, 500 MHz

3.2.2 Tests

Test 1. (NFS)

As first result we have started by a case in which we take as model an example similar to the one used before, the 1.2 GBytes film coded with MPEG-1 and with a startup delay lower than five minutes.

We suppose L is such that we have a MPEG-1 file with a size of 1.2 GBytes, with a consumption rate of 1.33 Mbps (a little lower than the typical MPEG-1 consumption rate), and we want the startup delay d of about four minutes. We also suppose that we have multicast channels with 1.5 Mbps capacity. Then, we will send the first segment on the first multicast channel with a rate of 1.5 Mbps. The next two segments on the second channel with rates of 0.75 Mbps and 0.5 Mbps respectively, etc.

To achieve the constraint for the startup delay, we must split the film in segments with size lesser than 50 MBytes. Then we must split it in 25 segments.

As we have already mentioned, if we want to verify the application with similar conditions, without any video with these characteristics, we must have the same number of segments with a proportional size.

In our case we have used a 163 MBytes size video coded with a 1.33 Mbps rate, and split in 25 segments with a segment size of 6.67 MBytes. With these characteristics the startup delay will be 35 seconds.

Remember that if the application works fine with these characteristics we can say it will work also with the example (1.2 GBytes film) characteristics.

- film length, L : 981 sec.
- film size: 163.1 MBytes
- film codification rate, b : 1.33 Mbps
- $R_{tx,1}$: 1.52 Mbps
- segment number, N : 25
- segment size: 6.67 MBytes
- multicast channel number: 5 channels
- Java Virtual Machine: for Windows
- storing system: NFS-type
- host machine:
 - machine: lynch.eurecom.fr
 - Operative System: Windows NT 4.0
 - RAM: 128 MBytes
 - Processor: Pentium II, 450 MHz

When we have made this test, the film display was correct and data was correctly stored in a file. Then we can say that our application works correctly within this environment.

Related with the statistics collected, we next show them after an off-line treatment with a spreadsheet. Figure 3.2 shows the aggregate buffer occupation evolution the long the application is running. In the x axe we have the execution time. In the y axe we have the aggregate buffer occupation normalized by the film size in percentage. We can see how in this case, with a transmission bandwidth 1.126 times the minimum case, we have the maximum occupation in a value of 35%. For our case it represents a maximum of 57 MBytes of data stored. If we would be acting with the example case, we would have the 35% of 1.2 GBytes, that is, 430 MBytes. This value is feasible for the present PC hardware.

An interesting effect we can see in this figure 3.2 is the saw-tooth shape it has. When a segment deadline arrives we can free the memory assigned to store that segment (nevertheless, in our application we haven't done that, but in the statistics we simulate we do it) and that causes this shape.

We can also observe how the curve slope, apart from the discrete jumps in the deadline times, is always positive, but each time less abrupt. That is due to the application must record in the early moments from many segments in a simultaneous fashion. In the case of minimum transmission bandwidth the application must start the recording of all the segments in the zero execution time. When the time goes on and there are several segments already displayed, the data flux incoming is lower, and then, the buffer is filled in a slower fashion. Is for that the slope is lower in the last moments of the execution.

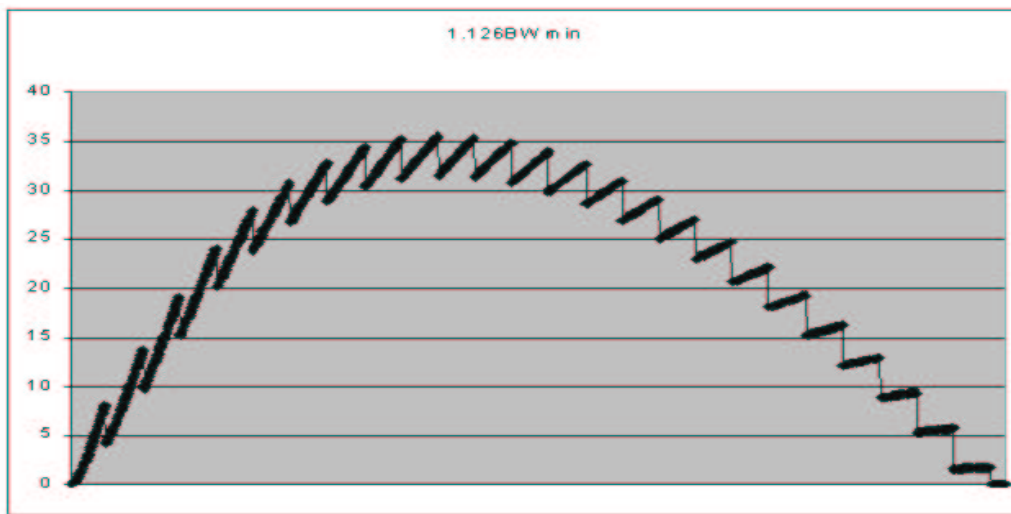


Figure 3.2: Aggregate buffer occupation for the test 1

The slope of the figure 3.2 is strongly related with the next figure (fig. 3.3). In the last, we present the aggregate reception rate need for the client be able to run the application correctly in each time of execution life. It's important to remember that, at each instant, the application must receive from the network this data flux, store it and, if it has already started the film display, it must read data from the storing device and display it.

In this figure 3.3 we have the application execution time in the x axe, and the incoming data flux in the y axe. This incoming data flux is normalized by the value of the film consumption rate. In this figure we can see how the higher data flux is 2.5 times b , that is 3.33 Mbps.

There is a relation between the reception rate and the slope of the buffer occupation. As we can see comparing the two figures, the higher is the reception rate, the higher is the buffer occupation slope. That is logic because the faster data

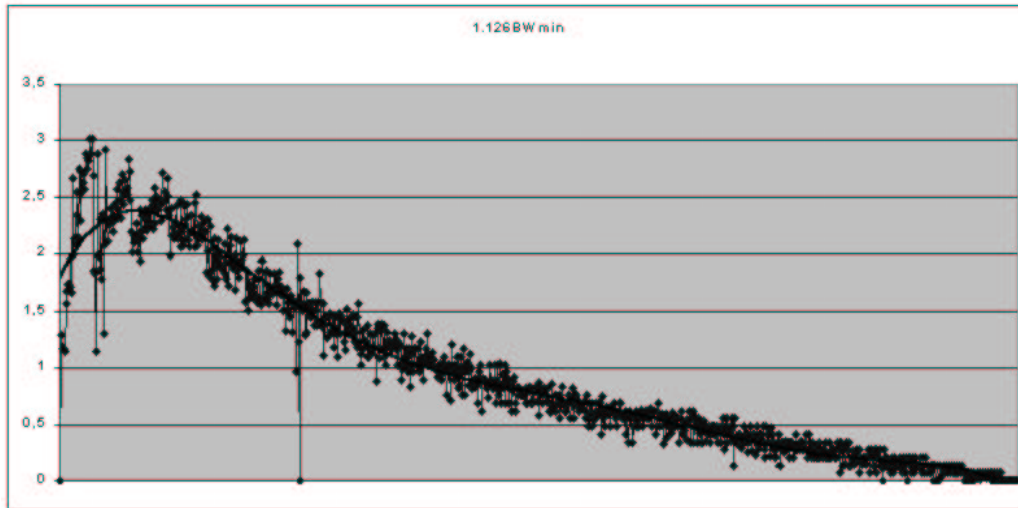


Figure 3.3: Aggregate reception rate for the test 1

arrives, the faster the buffer is filled, and then, the faster the buffer occupation grows.

To sum up we can conclude from this experiment that our application answers fine when it's necessary to display a film like the used with a NVOD system based on Birk. Then we can assure the correct application working with a longer film if we make the startup delay greater.

What have we improve comparing with a true-VOD video server? The answer is simple. To transmit this film we use 5 multicast channels with 1.52 Mbps capacity. That is, we use 7.6 Mbps. A true-VOD video server with the same bandwidth could only provide the service to six clients. While we can provide our service to an unlimited number of clients.

Test 2. (NFS, BW_{min})

Now we present the results of a test done to try to improve the system figures. In the last example we had used a transmission rate greater than the film consumption rate. Due to that, we were not in the minimum transmission bandwidth case. Then, our objective for this test was to know the application behavior in this minimum bandwidth use case.

For that we have maintained the parameters from the first test with the exception of $R_{tx,1}$. Now, this value was 1.35 Mbps, almost the film consumption rate (1.33 Mbps).

In this case we saw that the recording and the display were successfully done. Figures 3.4 and 3.5 show the statistics collected. The graphic axes have the same meaning than those of the first test.

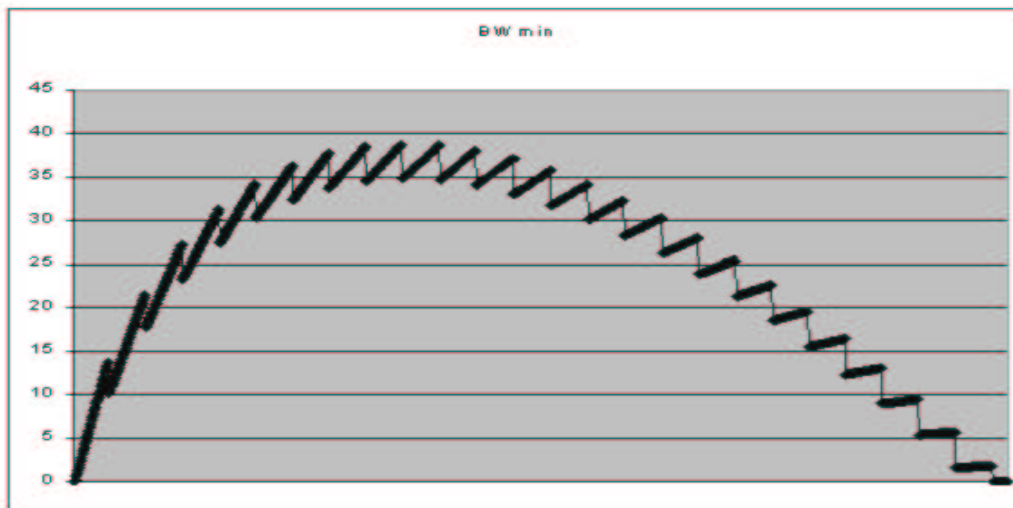


Figure 3.4: Aggregate buffer occupation for test 2

As we can see, the maximum buffer occupation level has reach the 40% of the film size. The buffer occupation curve slope and the client reception rate have the maximum value located in the zero execution time. This last parameter has reach four times the film consumption rate.

As conclusion to this experiment we can say that there is a trade-off between the server/network capabilities and the receptor terminal needs. The higher bandwidth used to send data, the lower the client buffer and the client processing capacity need.

Test 3. (NFS, BW_{min} , d_{min})

The question then was: which is the limit with this kind of storing, virtual file system, and with this terminal used by the client?

To answer this question we have made several tests. We next present the one that gave us the best results.

These are the transmission characteristics. The statistical results are presented in figures 3.6 y 3.7.

- film length, L : 981 sec.

- film size: 163.1 MBytes
- film codification rate, b : 1.33 Mbps
- $R_{tx,1}$: 1.35 Mbps
- segment number, N : 39
- segment size: 4.2 MBytes
- multicast channels: 6 channels
- Java Virtual Machine: for Windows
- storing system: NFS-type
- host machine:
 - machine: lynch.eurecom.fr
 - Operative System: Windows NT 4.0
 - RAM: 128 MBytes
 - Processor: Pentium II, 450 MHz

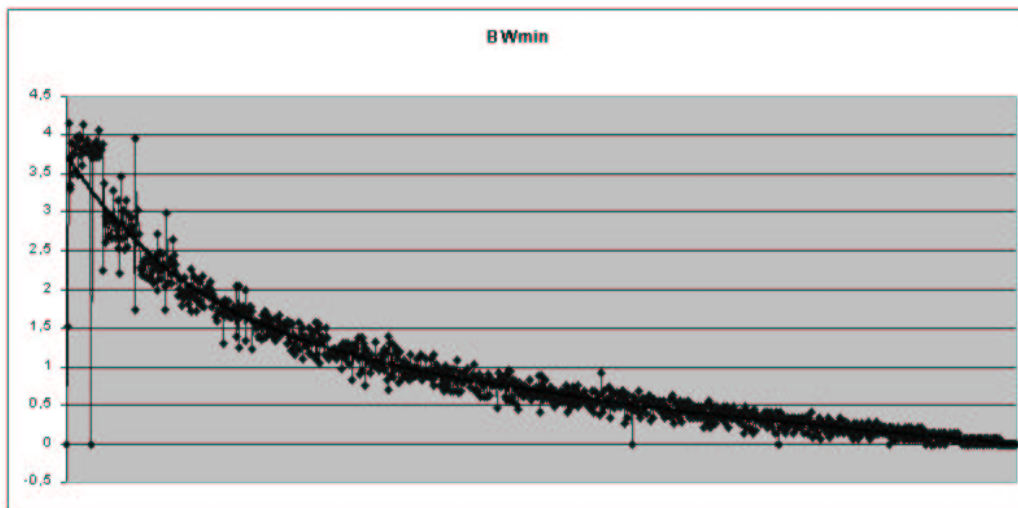


Figure 3.5: Aggregate reception rate for test 2

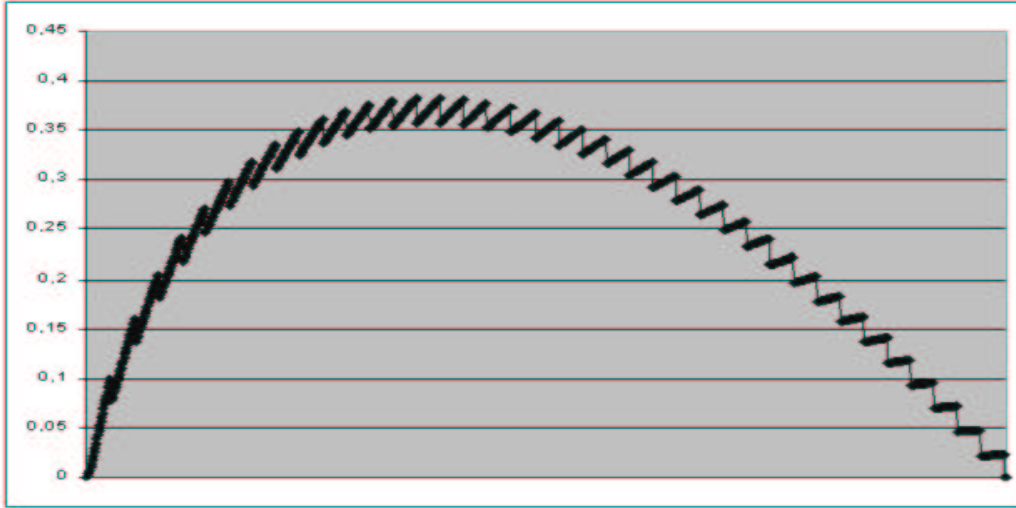


Figure 3.6: Aggregate buffer occupation for test 3

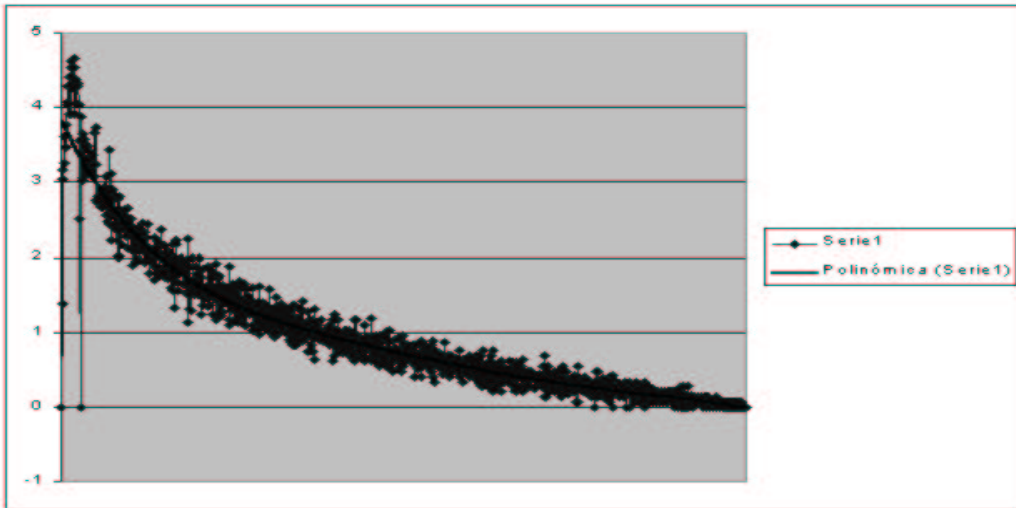


Figure 3.7: Aggregate reception rate for test 3

As we can see, we have used a total bandwidth of six 1.35 Mbps-channels. This sums 8.1 Mbps. With this bandwidth, a true-VOD server would be able to provide service to 6 clients. In the same conditions we can serve the film to an unlimited number of clients. The drawback is that our clients must have more sophisticated hardware and its characteristics must be better. Moreover, they will suffer a non-zero startup delay.

With our system the startup delay is 24.4 seconds. If the film was split in 50 MBytes segments, the model case, it would be about 3 minutes.

After that, we have tried to reduce a half this time the client must wait until the film can be reproduced. For that we had to divide a half the segment size. Then we had to do twice the number of segments. The test was made with 78 segments with a segment size of 2.1 MBytes multiplexed on 10 multicast channels.

It's important to say that we are in the minimum transmission bandwidth case, and then, when the application starts to run, the recording of all the segments starts also. We see while the test is running that a memory problem is presented to us. That is due to the 78 `BufferedRAF` the application must manage in memory. The `BufferedRAF` memory size by default is 700 KBytes.

We must emphasize that varying this by default parameter, or letting the Java Virtual Machine to get more memory, we can solve this problem. It's possible, but it's necessary to study it, that our application manage this situation.

To sum up, we can say that our application, "as is", and with the conditions used, could serve a non-limited number of Eurécom's network clients a one hour and half film coded with 1.33 Mbps. Each of these clients would have to suffer a startup delay about 3 minutes. And the bandwidth cost would be 8.1 Mbps.

Test 4. (local HD)

Now we present the results made with the other type of data storing. Before, we had used a virtual file system through the network. Now the storing device is going to be the machine local hard disk. Intuitively we can think that this is an advantage in order to improve the system characteristics. Surprisingly, as we will see, the effect is the opposite. The application capacity is much lower than the other already explained cases. After, we will give a probable explanation for this effect.

Firstly we had done several iterations in the characteristics to use in the tests in order to see how the application answers with this kind of data storing. The next characteristics are the best case we reach without errors.

- film length, L : 74.4 sec.

- film size: 20.9 MBytes
- film codification rate, b : 1.12 Mbps
- $R_{tx,1}$: 1.14 Mbps
- segment number, N : 11
- segment size: 1.9 MBytes
- multicast channels: 4 channels
- Java Virtual Machine: for Windows
- storing system: local HD
- host machine:
 - machine: lynch.eurecom.fr
 - Operative System: Windows NT 4.0
 - RAM: 128 MBytes
 - Processor: Pentium II, 450 MHz

These characteristics mean that, if we had a 1.2 GBytes size film coded with 1.12 Mbps (lower than the MPEG-1 standard) the startup delay the clients would have to suffer would be about 13 minutes. For that we would have to use a total network bandwidth of 4.5 Mbps. Figures 3.8 and 3.9 show the statistics results for this experiment.

If we try to increase the number of segments in which we split the film to reduce the client startup delay, we find the application doesn't work well. That is, the film display has jumps and is cut. The application also lose packets, even if they aren't lost in the network.

As example we next present disordered and lost packets statistics (figure 3.10) in the last experiment but with 19 segments instead of 11. Now the segment size is 1.1 MBytes. It's important to see that we are using the same multicast channel number, 4, but the startup delay is 40% lower.

As we can see in figures (3.11 and 3.12), if we vary the transmission conditions in order to be lightly more restrictive, we find errors in it. It's maybe due to the lack of CPU capacity. We can say then, the local hard disk storing is slower than the NFS storing.

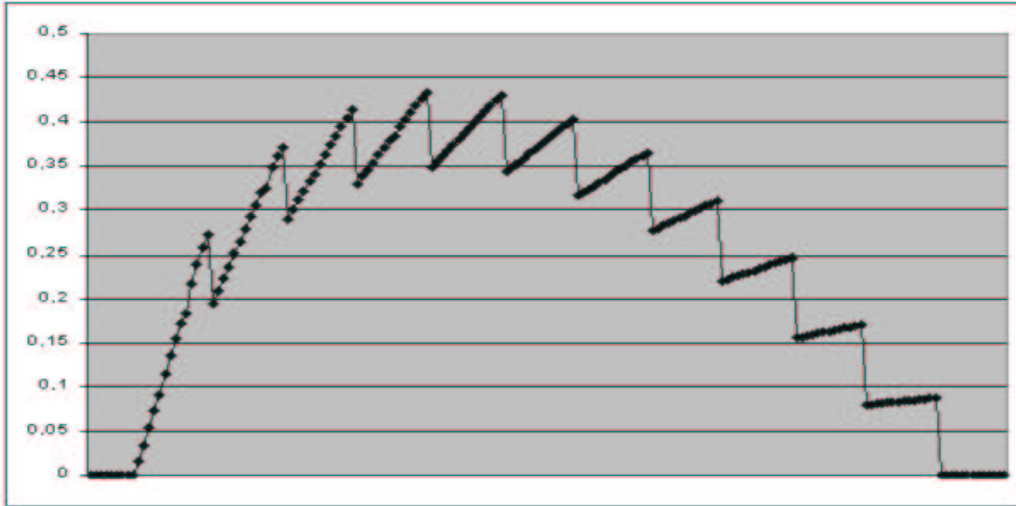


Figure 3.8: Aggregate buffer occupation for test 4

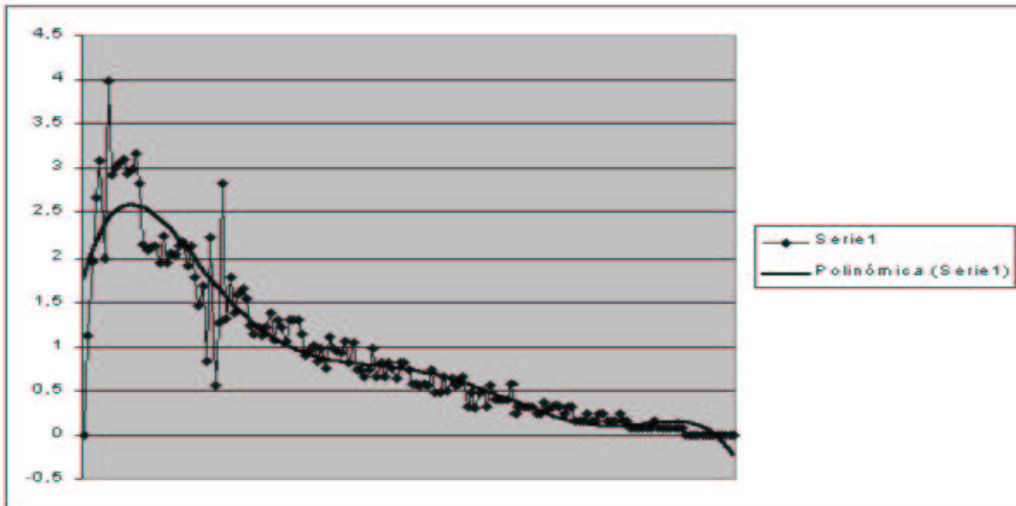


Figure 3.9: Aggregate reception rate for test 4

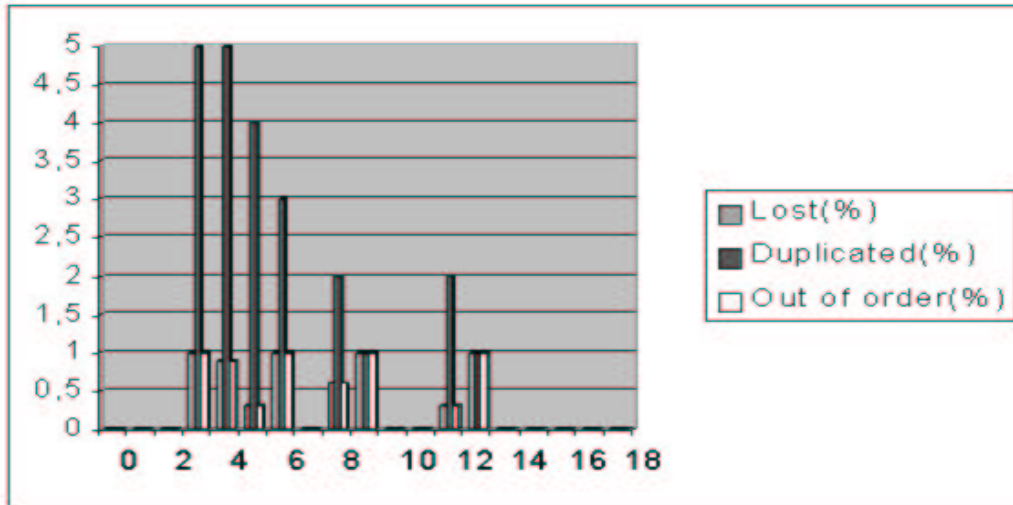


Figure 3.10: Packet statistics for test 4

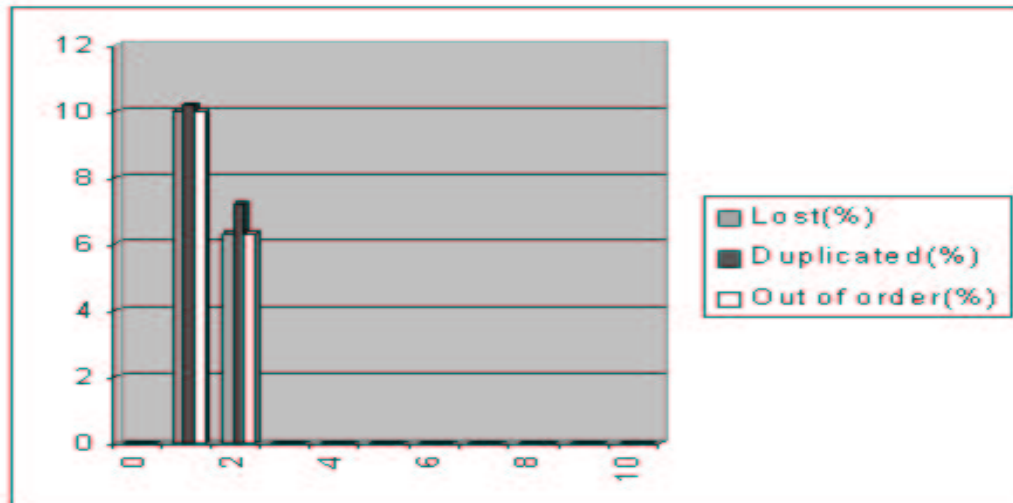


Figure 3.11: Packet statistics for variation in test 4

One important characteristic about these figures is that we found almost always that packets are lost because the application has not time enough to read from the network all the segments multiplexed in the same multicast channel.

We must also underline that we have made experiments in which we compare the local hard disk storing with the NFS storing under the same film and transmission conditions. In these experiments we find the application with NFS type data storing works fine while if it works with local hard disk storing it goes wrong.

The explanation for this behavior when we compare the NFS with the local hard disk storing can be one among the next possibilities, or a mixing of both. Nevertheless, they are suppositions and then there exists an open field for further investigations.

The first of these possibilities is the occurrence of hard disk fragmentation. This will do the read/write actions be very time expensive. Then, when we want, for example, read a segment from the storing device, we have an overload due to the internal indexation, etc.

The second possibility is the next. It's possible that using a NFS server could take us away an amount of work. It's possible that, using NFS storing, we divide the work between the CPU and the network card-NFS server couple. That can explain in part the behavior observed.

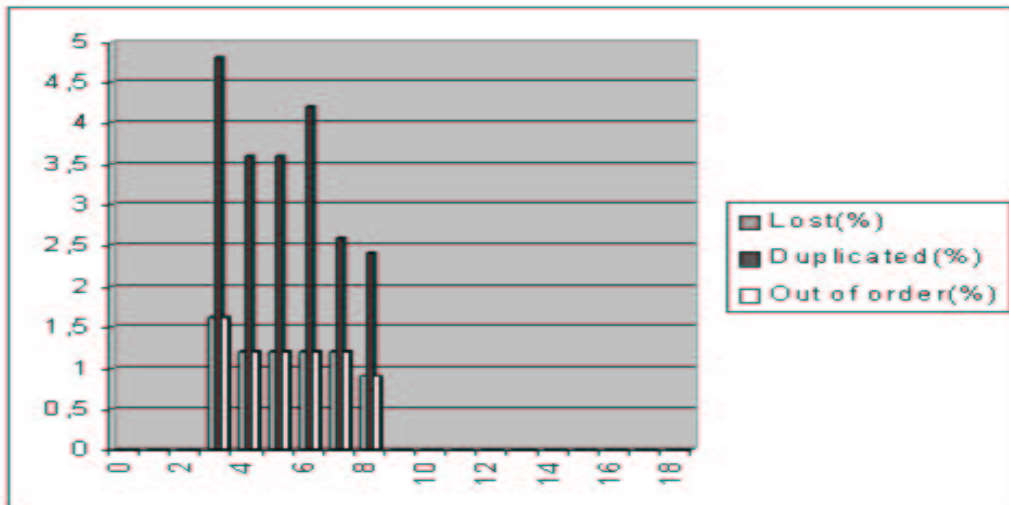


Figure 3.12: Packet statistics for second variation in test 4

Test 5. (JVM for Solaris and Linux)

Now we present the results for the tests made when our application is running over operating systems different from Windows family. The experiments have been made using JVM for Solaris and for Linux. The JMF versions are also for these two operating systems (the one for Linux is a beta version).

The first thing we find when we use these JVM is that the film display is interrupted many times or simply it is not displayed. That occurs with both operating systems. It can be due to incompatibilities between the different versions of JMF. It can also be due to the use in our application of some instruction OS-depending. This last idea is not impossible but we haven't found this error.

We have experimented with the application "as is". Then we have taken the results even if the display was not correct. Even if there wasn't display at all. We have searched the correct reception and we have observed the statistics. Nevertheless, it's clear that if the reproduction is not going on, the results are not reliable. If the MPEG decoding, that is very CPU demandant, is not occurring, the results are not very confident.

Nevertheless, we next present the transmission characteristics from the Solaris experiment.

- film length, L : 55.8 sec.
- film size: 20.9 MBytes
- film codification rate, b : 1.5 Mbps
- $R_{tx,1}$: 1.52 Mbps
- segment number, N : 11
- segment size: 1.9 MBytes
- multicast channels: 4 channels
- Java Virtual Machine: for Solaris
- storing system: local HD
- host machine:
 - machine: tulipe.eurecom.fr
 - Operative System: OS Solaris 2.5.1

- RAM: 640 MBytes
- Processor: Ultra SPARC Iii, 640 MHz

We remember that, as we have already comment, the film is correctly received but is not displayed. Figures 3.13 and 3.14 show the collected statistics.

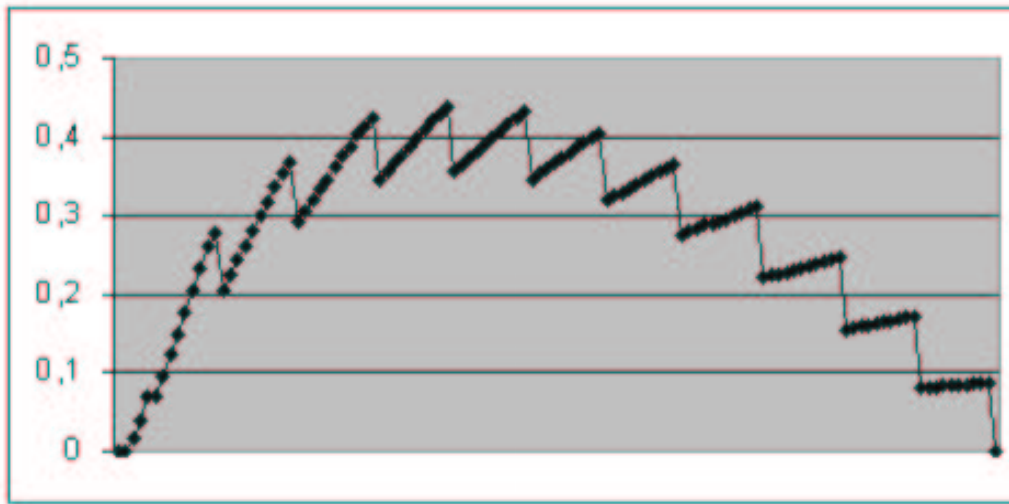


Figure 3.13: Aggregate buffer occupation for test 5

Test 6. (Non error-free network)

Lastly in this section we next analyze the case with induced errors. We have already said that we have always used an error-free private network. Then, for this experiment, we have disordered the packet sequence in a deterministic fashion.

If we do it, we see that when the local hard disk is used for the data storing, the CPU load reaches often 100% (figure 3.18). That makes us to think the application hasn't enough processing resources to do correctly its task.

For example, we present the characteristics of an experiment and the statistic figures 3.15, 3.16 and 3.17.

- film length, L : 148.8 sec.
- film size: 20.9 MBytes
- film codification rate, b : 0.56 Mbps

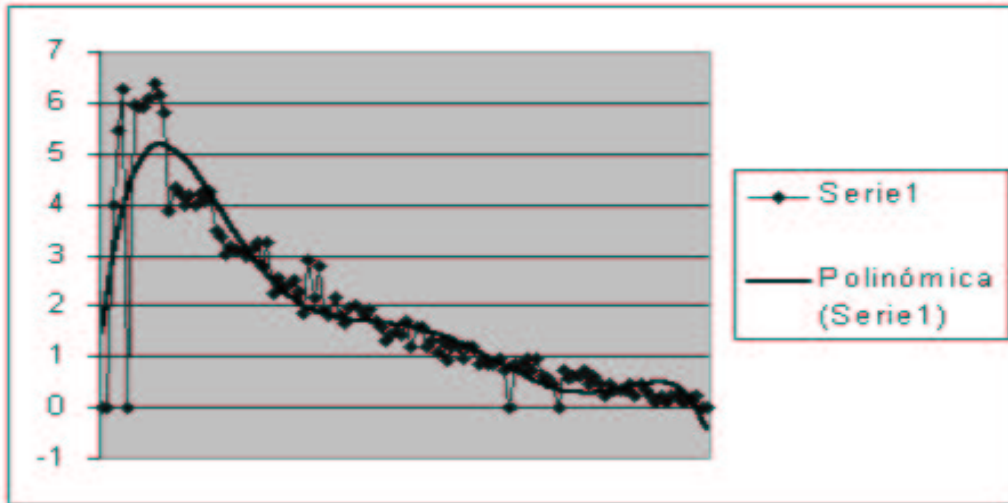


Figure 3.14: Aggregate reception rate for test 5

- $R_{tx,1}$: 0.76 Mbps
- segment number, N : 11
- segment size: 1.9 MBytes
- multicast channels: 4 channels
- Java Virtual Machine: for Windows
- storing system: local HD
- host machine:
 - machine: lynch.eurecom.fr
 - Operative System: Windows NT 4.0
 - RAM: 128 MBytes
 - Processor: Pentium II, 450 MHz
- disorder packets rate: 2%
- loses rate: 0%

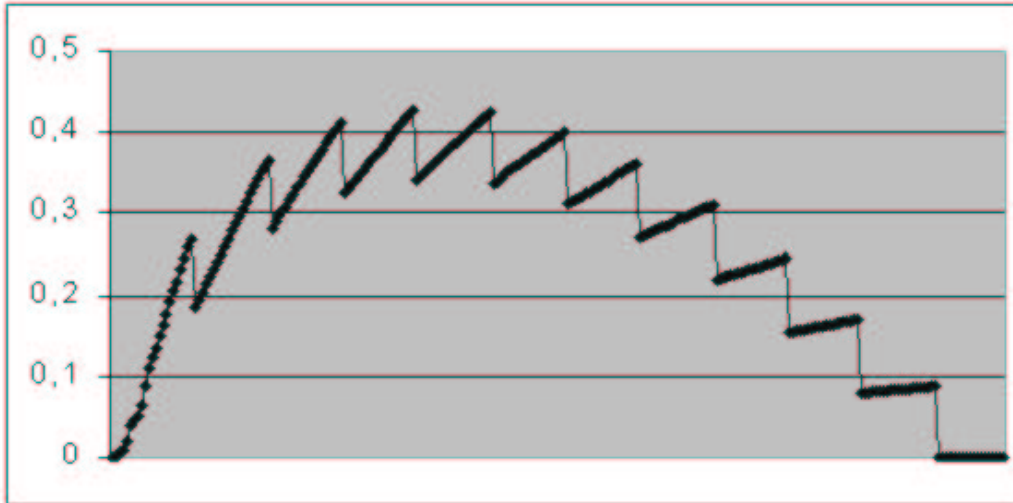


Figure 3.15: Aggregate buffer occupation for test 6

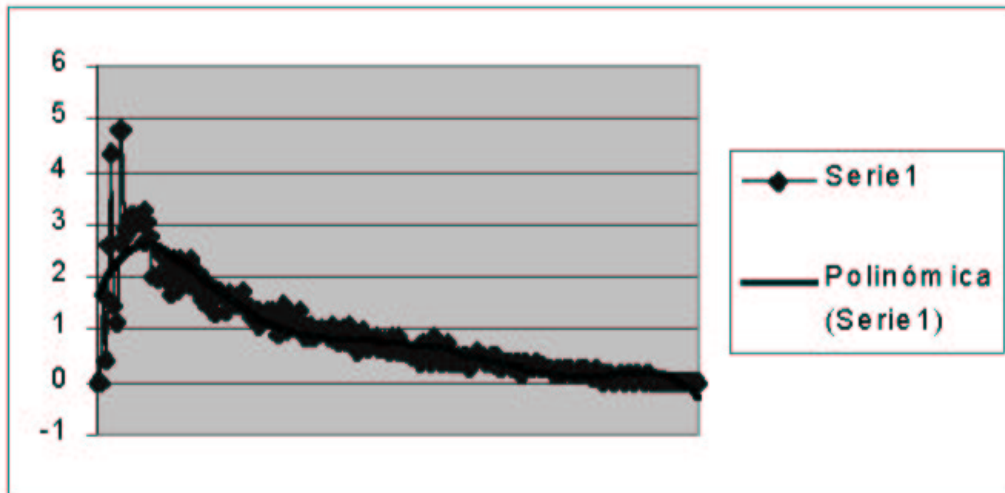


Figure 3.16: Aggregate reception rate for test 6

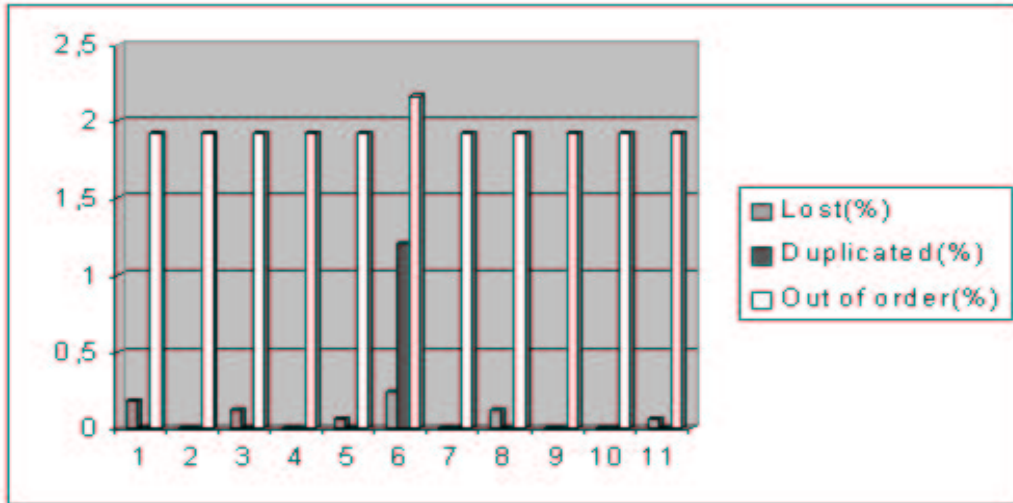


Figure 3.17: Packet statistics for test 6

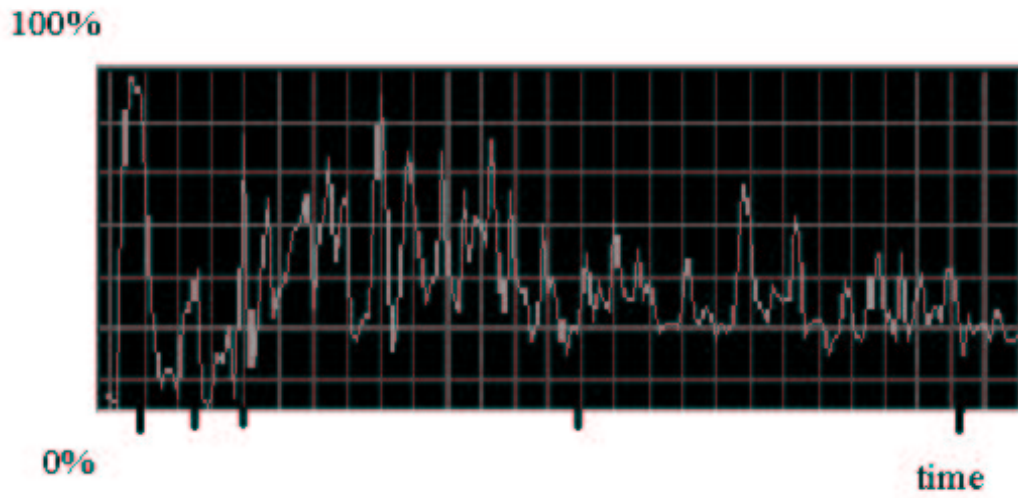


Figure 3.18: CPU load during test 6

We can observe several details. First we can see that errors occur, even if there are not errors in the network transmission. We can also remember that, even in more restrictive cases, in the error-free experiments the transmission is done successfully.

That is due to the application error management. More properly, to the application time-out management because it's a very CPU expensive fact. It's, then, a not recommended application for networks with a high error probability or very different delays in the packet transmission. Logically, with more powerful, in CPU terms, machines this effect will be tempered.

Moreover, it's important to point out an observation made with the last and other experiments in which there have been errors in the display and reception of the movie. When there are errors due to the lack of ability in the application to manage all the data it receives, we can observe an avalanche effect. That is, the error appearance makes the application need more CPU resources just in the moment it has a CPU problem, remember the error appearance is due to the excessive need of CPU. This effect will be worst if there is an application extension to provide a error recovering method because that would add more CPU need.

Chapter 4

Conclusions

When we read this document that talks about our work, one can ask himself if the primitive aims have been reached. Is near-VOD an acceptable substitute of true-VOD? Is near-VOD an improvement for the video-on-demand service providers?

One of the conclusions we have extracted from the present work is the answer to these questions. Even when, as all the engineering related things, has its benefits and its drawbacks. Then the question is if the benefits are larger than the drawbacks.

We have seen some comparisons in the results section that show the near-VOD behavior has great possibilities to collect economic benefits for the Video-On-Demand service providers. That is due to the unlimited number of possible users that could enjoy the service and the limited needed resources. Our application can be used by no clients or by a great number of final users. That fulfills the scalability promise made at the begin of this work.

But one of the conditions to achieve this scalability is the open-loop transmission we make. Then the server works independently of the group of clients or their needs. That has its cost because the server must provide the same amount of data when there are one thousand of clients or when there are none. Then, these near-VOD systems are only profitable when there is a guarantee of the service is going to be **highly demanded**. Even so, that is not a great drawback because in true-VOD systems the server must have all its resource capacity reserved even if there are no client.

Other problems related with the near-VOD system are the next. The client must have a higher complexity. That is due to the fact it must compute a non-trivial algorithm and must carry out a group of connections and disconnections to different channels, distinguishing between data relevant or not at each time. That has the consequence that the PC-host client requirements are high. With the section 3

we have tried to prove that the present commercial PC's are enough for this task.

If we had used a dedicated hardware device to be used as client (set-top-box), its cost would be higher than the equivalent device for a true-VOD system due to the complexity inherent to our system.

In our system, also, the clients must suffer a given delay (startup delay) between the moment they demand the service and the moment the film display starts. In this work we have wanted to show that is not ridiculous to think in acceptable delays of 4 minutes with the used technology for a one hour and half duration film.

All these conclusions are referred to the near-VOD general characteristics. With this work we have tried to make these characteristics visible by the way of an implementation.

But we can also analyze some characteristics related with the application itself and not with the NVID system generalities.

In this work we have chosen the use of MPEG-1 coded contents due to their wide acceptance and their high quality. For that, as drawback, we find a high CPU demand in order to decode this movie format and to display it. As we have already note, that choosing can suppose a problem in certain cases. If so, we can opt for other kind of film coding, less expensive in CPU terms. JMF gives us the possibility of using other formats.

Another decisive factor for the application developing is the machine over which we run it. Nowadays, a domestic PC has characteristics much higher than the one used for the experiments. We can find PC's with speed of 1 GHz and hard disk with capacities higher than 10 GBytes and I/O bandwidth very large. Then we can wait the application made in this work be easily used with better results in present machines.

In an application that receives data from several channels at the same time and that must show it in ordered fashion, the synchronization is essential. Then it's very important the time granularity. This granularity is provided by the operating system and by the Java Virtual Machine. In the moment we have made this work the only couple that let us the granularity level we require was the JVM for Windows and this operating system. It's natural that, as the hardware improves, the operating systems gives us better results each time.

Chapter 5

Open issues

The most important open points in this work are those for which we have tried the application be easily extensible. We make a description next:

- Interactive functions implementation
 - Fast Forward
 - Rewind

These functions implementations would be a qualitative advance very large in terms of user application and conceptually.

We have tried to make the application with a modularity level as to let the extension in order to have these new characteristics without interfering the rest of the system.

- Error recovery

In the present work there is a packet management that lets us to distinguish between lost and disordered packets. That is indispensable in order to do, in the future, any kind of error management.

The fashion in which the application will recover these errors is an open work. But it's important to underline that, independently of what is the kind of error recovering implemented, this will penalize more the need of processing resources.

For this work execution we set out different ways in the error recovering field. One of them is the use of FEC technics or the use of Tornado Codes. Nevertheless it's a open field for future study.

- Other open-loop near-VOD algorithms

Of course, it rests as future work the implementation of other algorithms. We have made the implementation of Birk algorithm, but the most important idea that promoted this work was that we wanted to compare different algorithms. After, it would be interesting to compare the statistics of each of them.

Moreover there are other topics open to a study deeply. These topics don't lead to an application extension.

- Other operative systems

It's a possible future work the application study running over other kind of operating systems as Solaris or Linux.

- Application and transmission internal parameters effect.

It's possible and desirable a more deep study in the consequences in the application performance of varying some application internal parameters. Some examples are the by-default memory size used by the `BufferedRAF` objects, the time used to consider a packet has been lost (by default 0.5 seconds), the relation between UDP packet size and the network MTU or the reception buffer size of reception sockets.

Appendix A

Birk's algorithm

In this appendix we want to explain lightly how the Birk algorithm works for its use in near-VOD systems. To see it, we show an example. For more information see [1].

As is the case in the rest of open-loop NVOD system based algorithms, we do a film segmentation on the movie we want to transmit. We divide in N segments of equal size. After, the server starts to broadcast each of them *continuously*. Each of these segments is broadcasted with a different transmission rate. The faster is the first segment and the slower is the last of them, as we can see in figure A.1.

The Birk algorithm assigns the transmission rate in this fashion. If we consider the next values normalized by the transmission rate of the first segment we have that this, the first segment, is broadcasted with a 1 rate. The next segments are sent with rates in a harmonic series. That is, the second is broadcasted with a $1/2$ rate, the third with $1/3$ rate, etc.

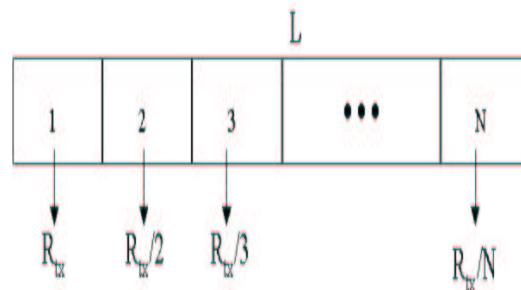


Figure A.1: Segmentation in the Birk algorithm

Then the server is sending each segment continuously and with its respective

transmission rate. We can see an example in the next figure (A.2) with $N = 4$.

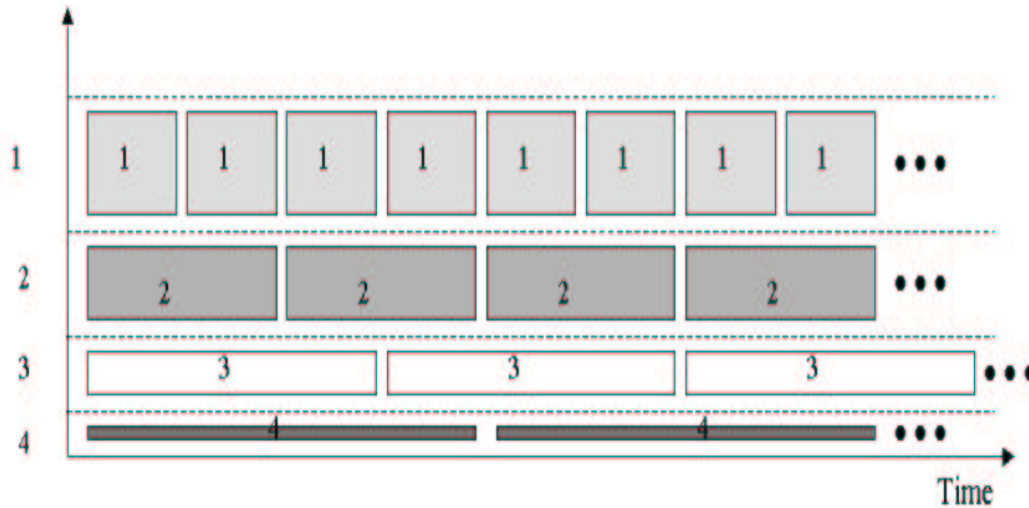


Figure A.2: Example of Birk transmission

We must remember that all the segments has the same size (excepting, maybe, the last). This is shown in the figure A.2 in the rectangles area. The fact that someone of them are longer than others in the time axe is due to the different transmission rates assigned.

When a client wants to connect to the service provided, it must distinguish what data have to start to record in that moment and when start to record the other data. In the Birk solution, we consider that a segment has to start to be recorded if there will not be received completely before its deadline. For example, if we use the minimum possible transmission bandwidth, the client must start to record *all* the segments in the same moment it starts the connection.

Figure A.3 shows the situation produced when a client arrives, it's connected to the system, at the moment t_2 . The shadow parts of the segments are the data recorded by this client in its buffer. In this example we are in the minimum transmission bandwidth case.

In this example segment deadlines are, for each segment, t_3 for the first segment, t_4 for the second, t_5 for the third, t_6 for the fourth, and the film would end its display at t_7 .

In figures A.3 and A.2 we can observe several important algorithm details. First, we observe how the transmission is made in a continuous fashion for each segment.

Second, we can see that there are moments in which we have to record data coming through several segments at the same time.

Third, we see that *it's not necessary to start the data recording in a ordered fashion*. In this example the segment number 3 is recorded starting with the last part of it, and after is recorded the rest. The important thing is the segments *must* be recorded completely before the arriving of its deadline.

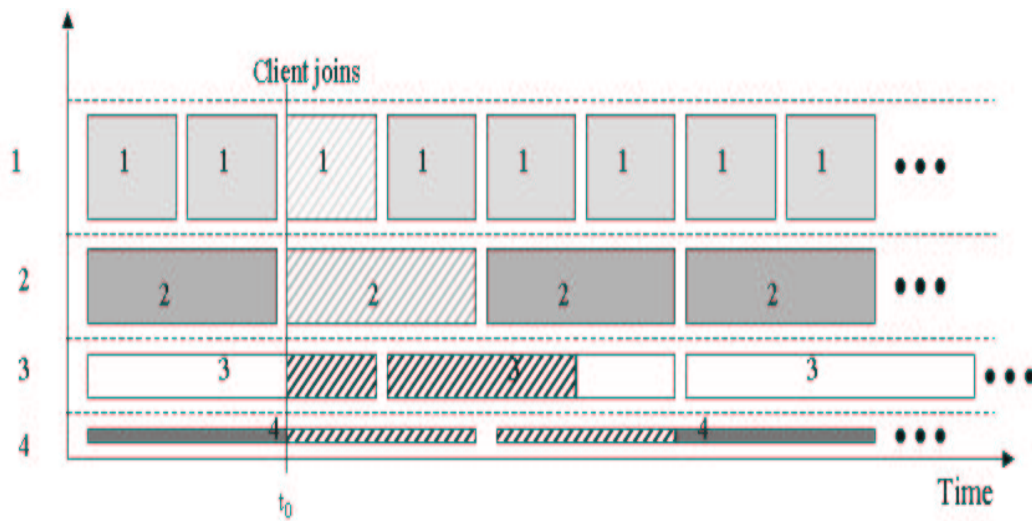


Figure A.3: A client joins at the t_2 time

Bibliography

- [1] Yitzhak Birk and Ron Mondri. Tailored transmissions for efficient near-video-on-demand service. *Proc. IEEE International Conference on Multimedia Computing Systems*, pages 226–231, 1999.
- [2] Grady Booch. *Object-oriented analysis and design, 2nd edition*. Benjamin/Cummings, 1994.
- [3] Leonardo Chiariglione. Short mpeg-1 description. <http://www.cselt.it/mpeg/standards/mpeg-1/mpeg-1.htm>.
- [4] K Hua and S Sheu. Skyscraper broadcasting: A new broadcasting scheme for metropolitan video-on-demand systems. *Proc. ACM SIGCOMM*, September 1997.
- [5] Christian Huitema. *Le routage dans l'internet*. Eyrolles, 2ème tirage, 1995.
- [6] Li-Shen Juhn and Li-Ming Tseng. Harmonic broadcasting for video-on-demand systems. *IEEE Trans. Broadcasting*, pages 268–271, September 1997.
- [7] D Towsley L Gao and J Kurose. Efficient schemes for broadcasting popular videos. *Proc. Inter. Workshop on Network and Operating System Support for Digital Audio and Video*, July 1998.
- [8] Sun Microsystems. archives of jmf-interest@java.sun.com. <http://archives.java.sun.com/archives/jmf-interest.html>.
- [9] Sun Microsystems. Java Media Framework API. <http://java.sun.com/products/java-media/jmf/2.1.1>.
- [10] Darrell D. E. Long Steven Carter and Jehan-Francois Pâris. Video-on-demand broadcasting protocols.

- [11] Darrell D. E. Long Steven Carter and Jehan-Francois Pâris. A simple low-bandwidth broadcasting protocol for video-on-demand. *Proceedings of the 8th International Conference on Computer Communications and Networks*, October 1999.
- [12] Lixin Gao Subhabrata Sen and Don Towsley. Frame-based periodic broadcast and fundamental resource tradeoffs. 1999.
- [13] Ivan Wong. Re: programming a processor w/ custom data-source. <http://archives.java.sun.com/cgi-bin/wa?A2=ind0012L=jmf-interestF=S=P=19514>.