A SEMANTIC-BASED MIDDLEWARE FOR MULTIMEDIA COLLABORATIVE APPLICATIONS

by

Agustín José González
M.Sc. Computer Science, December 1997, Old Dominion University
M.Sc. Electronic Engineering, December 1995, Universidad Federico Santa María, Chile
B.Sc. Electronic Engineering, December 1986, Universidad Federico Santa María, Chile

A Dissertation Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirement for the Degree of

DOCTOR OF PHILOSOPHY

COMPUTER SCIENCE

OLD DOMINION UNIVERSITY
May 2000

Approved by:

Hussein Abdel-Wahab (Director)

James Leathrum (Member)

Kurt Maly (Member)

C. Michael Overstreet (Member)

Christian Wild (Member)

ABSTRACT

A SEMANTIC-BASED MIDDLEWARE FOR MULTIMEDIA COLLABORATIVE APPLICATIONS

Agustín José González
Old Dominion University, 2000
Director: Dr. Hussein Abdel-Wahab

The Internet growth and the performance increase of desktop computers have enabled large-scale distributed multimedia applications. They are expected to grow in demand and services and their traffic volume will dominate. Real-time delivery, scalability, heterogeneity are some requirements of these applications that have motivated a revision of the traditional Internet services, the operating systems structures, and the software systems for supporting application development. This work proposes a Java-based lightweight middleware for the development of large-scale multimedia applications. The middleware offers four services for multimedia applications. First, it provides two scalable lightweight protocols for floor control. One follows a centralized model that easily integrates with centralized resources such as a shared tool, and the other is a distributed protocol targeted to distributed resources such as audio. Scalability is achieved by periodically multicasting a heartbeat that conveys state information used by clients to request the resource via temporary TCP connections. Second, it supports intra- and inter-stream synchronization algorithms and policies. We introduce the concept of virtual observer, which perceives the session as being in the same room with a sender. We avoid the need for globally synchronized clocks by introducing the concept of user's multimedia presence, which defines a new manner for combining streams coming from multiple sites. It includes a novel algorithm for estimation and removal of clock skew. In addition, it supports event-driven asynchronous message reception, quality of service measures, and traffic rate control. Finally, the middleware provides support for data sharing via a resilient and scalable protocol for transmission of images that can dynamically change in content and size. The effectiveness of the midleware components is shown with the implementation of Odust, a prototypical sharing tool application built on top of the middleware.

Copyright © 2000

by

Agustín José González. All rights reserved.

ACKNOWLEDGMENTS

This research is truly the result of the collaboration and my interaction with many people around the world.

The faculty members of the Computer Science Department of the Old Dominion University deserve recognition. In this group, I am especially grateful for Dr. Wahab, my advisor. I thank his permanent advice and weekly hours of constructive guidance and rich discussions. I also appreciate the time he took to promptly review and comment all my partial reports, papers, and all the versions of this dissertation.

Special thanks also goes to Dr. Michael Overstreet. My work on Chapter IV would not have been possible without the tools he gave me in his simulation class and later refined during our numerous discussions. In addition, I consider important the integral relationship that I had with Dr. Wahab and Dr. Overstreet. It went beyond the research arena, gave sense to my studies and balance to my life.

I thank Dr. Maly for his encouragement and for supporting my ideas during the first stage of this research. I also appreciate his detailed review of this dissertation and his valuable comments.

My thanks also goes to Dr. Wild. I benefited a lot from his previous work on image transmission and from our interesting discussions on the issues and alternative solutions to this problem.

My appreciation also goes to Dr. James Leathrum for his collaboration and contribution as the external member in my committee.

Dr. Toida and Dr. Olariu also helped me and contributed to the successful completion of this dissertation. They were always accessible and took the time to discuss some specific matters. Dr. Olariu was especially important in my quest for solutions to the problem of partitioning a rectilinear polygon into a minimum number of non-overlapping rectangles.

I am also grateful for Ajay Gupta and the CS System Group. I thank them very much for keeping the system up.

Thanks to Jaime Glaría of the Federico Santa María Technical University, Chile, for refreshing my memory on a subject that he started to teach me during my first year of undergraduate study. I am glad I remembered some of it and with Jaime’s help I could apply it to this work.

I finally want to thank Cecilia, my lovely wife, and Eduardo and Rodrigo, my wonderful sons. They were extremely patient and understanding during all these years. Cecilia suspended her professional carrier and accepted to follow this journey with me away from Chile to make one of my dreams come true. I thank you all for your never fading love.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Multimedia applications are the result of many years of advances in computer networks and computer technology. The Internet has grown from 4 hosts in 1969 to more than 56 million today worldwide. Likewise, the connection speed has grown from 50 kbps used to connect the first four nodes to 2.5 GBps [80] for the US backbone upgrade started in 1999 and hundreds of kilobit per second for users connected from home. Along with the advances in networking, computer technologies have also experienced important developments that now enable real-time audio and video processing in personal computers. It was not long ago when these technologies crossed a threshold that made distributed multimedia applications feasible at a reasonable quality.

Multimedia applications require high communication bandwidth and substantial data processing. A television quality video, for example, requires around 100 Mbps of information that obviously cannot be transmitted without compression on today's typical user connections. There are three conflicting requirements, bandwidth, processing power, and quality. We would like the bandwidth consumption to be a minimum, the processing to be low, and the media quality to be as close as possible to that perceived in person-to-person encounters. Nonetheless, lower bandwidth can only be achieved at the cost of higher compression processing, which leads to longer delay, or if we want to keep the processing low, the fidelity is reduced. A good tradeoff between the two major technology factors, processing and bandwidth, has enabled multimedia applications with reasonable quality.

The introduction of multicasting in the late '80 is also another cornerstone for the large-scale deployment of multimedia applications. This protocol not only reduces bandwidth consumption in the Internet but also accomplishes lesser latencies when compared with multiple unicast transmissions. With the first MBone [42] audio and

---

The journal model for this dissertation is the IEEE/ACM Transactions on Networking.

video multicasts in 1993 the era of computer-based multimedia large-scale teleconferencing started.

Although multimedia applications are rapidly emerging due to the above technological advances, we believe, like other researches and developers, that there are still problems that require better solutions in the bandwidth, processing, and scalability space. In addition, as some requirements are reasonably met, new ones appear. These include heterogeneity, scalability, reliability, flexibility, and reusability. The Internet is an intrinsically heterogeneous environment. Regardless of a machine’s connectivity, hardware, and operating system, they can exchange messages in the Internet if they support the Internet Protocol (IP). This protocol is the common denominator that supports a wide variety of applications and end-to-end protocols, and on the other hand, it rests upon a wide variety of network and datalink protocols. This feature has enabled the integration of many types of machines and operating systems. At the same time it has forced applications to support a variety of user environments for their deployment in the Internet. Multicasting is a big step towards scalability; however, it mainly tackles the transmission of unreliable datagrams to a potentially large number of users. Other scalability issues in multimedia systems are how to manage control information for large groups and how to obtain feedback from such groups. An example of control exchange is floor management in audio conferencing, tool control in shared tools, and tele-pointers. In these cases, unreliable multicast does not meet the communication semantic requirements, and new reliable multicast protocols do not scale as well as plain multicast. In Chapter I, we propose a framework for floor control that is based on a permanent unreliable multicast channel and temporary point-to-point connections. It achieves the same degree of scalability as that of unreliable multicast protocol. Scalable feedback for large groups is also an ongoing subject of research. Its main applications are in providing feedback in reliable multicast protocols and estimation of the number of receivers for transmission suppression in adaptive protocol [49]. Reliability is also an issue that arises due to the distributed, heterogeneous, and large-scale nature of multimedia applications. As more users join a session, i.e. a collection of communication exchanges that together make up a single overall identifiable task, faults might occur due to configuration or version differences, race conditions, or scenarios impossible to generate in reduced-scale testing. Our approach to reliability is simplicity and a successive refinement methodology based on the identification of the dominant factors of a solution. Finally, flexibility and reusability help tremendously to quickly create new applications and system extensions to fulfill new requirements. During the research and development processes, flexibility and reusability facilitate the refinement steps and the rapid creation of multiple variants to try and measure different tradeoffs and to identify the dominant factors of a solution.

We believe that a main difficulty in designing and developing multimedia applications that meet the above requirements is the big gap between multimedia applications requirements and the system services over which they are developed. Some examples of these services are the traditional Internet protocols, scheduling mechanism of traditional operating systems, system APIs (**A**pplication **P**rogramming **I**nterfaces) for network services, and abstractions for managing multimedia input/output user's devices. The main use of the Internet has traditionally been the transfer of data with no critical real-time constraint. Neither TCP nor UDP were designed to provide real-time delivery as required by multimedia applications. In Section 1.2, we briefly discuss the research efforts to alleviate this shortcoming. Like traditional Internet protocols, operating system scheduling techniques are not adequate for time constrained applications. Traditionally, process and thread priorities have been used to schedule processes and thread in time-sharing systems; however, a priority is not a good abstraction to encapsulate the time requirements for the execution of multimedia components. For instance, software algorithms for audio echo cancellation have not been possible mainly due to the difficulties in scheduling packet processing periodically. Although the operating system network APIs are general enough for a broad variety of applications, often application developers needs to build an additional layer to reach the services required for most of the multimedia components. These include support for asynchronous reception, quality of services measures, transmission rate control, and media synchronization. In Chapter III, IV, and V, we elaborate more on these ideas. Finally, the abstractions for user input/output devices preclude access for multimedia applications. For instance, there is a rigid one-to-one association between mouse and monitor. This complicates the creation of multiple remote mice that control a number of local applications concurrently. We develop this idea further in Chapter VI. The architecture and mechanism for accessing the display also decrease multimedia application performance. Assuming that we have the processing power to display TV quality video on a computer screen, we would need around 20 MBps transfer rate to the display, which is not feasible in X Window protocol without taking most of the resources for that task. Finally, current display's architecture incurs an additional overhead due to at least two copies of the images to be displayed. One copy is managed by the application and is used as backup for redrawing it after uncovering the window, and the other is kept in the display memory buffer to periodically refresh the pixels on the screen. This duplication forces applications to update images twice, in their own data structures and on screen.

The new research and engineering issues brought by multimedia applications have been tackled from distinct angles. These include changing the routing and forwarding semantic of the current Internet architecture in "Active network" research, building support for multimedia applications at the operating system level [37] [34], and providing software systems to enable more complex applications to be built quickly and effectively for distributed computing. We believe that even though these three approaches complement one another, only the third one can be easily deployed and integrated with current technologies. Thus, in this dissertation we propose a semantic-based middleware for multimedia collaborative applications.

1.1 Objective

Our main objective is to investigate and propose heterogeneous, scalable, reliable, flexible, and reusable solutions and enhancements for common needs in developing multimedia collaborative applications.

"The amount of bandwidth available in the Internet is increasing dramatically, both in the backbone networks, as well as in the last mile (broadband access). One consequence is that the delivery of high-quality multimedia data will become feasible, and multimedia data, including audio and video, will become the dominant traffic. As more users gain access to broadband services, new applications and services will become possible. The result will be a growing demand for large-scale multimedia applications and services." [6]. Our contribution is to provide a software infrastructure to facilitate and accelerate the development of such high quality multimedia applications. In general, midleware are software systems, as shown in Fig. 1, that enable more complex applications to be built quickly and effectively for distributed computing.

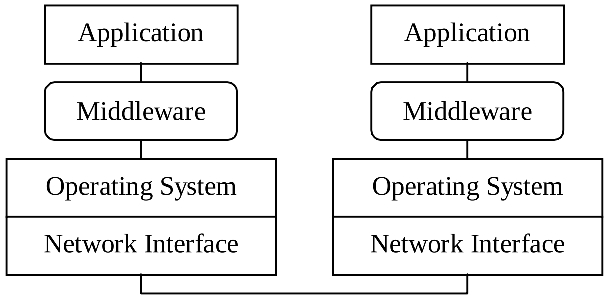


Fig. 1. Middleware location among other software modules.

In other areas middleware has been used to unite and integrate computing resources across disparate environments rather than for building new applications. For example, e-commerce relies on middleware to help systems conduct rapid and secure transaction across diverse computing environments [13]. Our middleware encapsulates common distributed services that we have observed in collaborative multimedia applications. These services include floor control for mutual exclusive resources, multimedia synchronization, enhanced network services, and protocol for dynamic image transmission. Our focus is on collaborative multimedia applications. These could be either synchronous or asynchronous. Synchronous collaboration takes place when the participants involved in common tasks are seated simultaneously at their workplaces. This is the case of virtual-classroom systems for distance learning and multimedia conferencing systems. Nevertheless, a significant portion of collaboration activities in the real world occurs in an asynchronous manner where people-to-people communication takes place in non-real-time fashion. These include web-based distance learning and electronic mail. Some synchronous systems also support asynchronous operation by means of recording and playback. Here, the system saves a transcript of the synchronous session that can be later reviewed. In this work we mainly concentrate on synchronous collaboration although the services we propose can also be applied to asynchronous activities. This can be done by their direct use or by their extension to take advantage of the peculiarities in knowing in advance all what is to be delivered as opposed to obtaining and delivering the information in real time. Finally, our approach is semantic based meaning that, in the middleware design and implementation, we specially took into consideration the behaviors and characteristics of each supported service. An example is the multi-protocol solution we propose for floor control that achieves scalability by reserving reliable communication only to transmit critical pieces of information. Another example is the introduction of a *virtual observer* and *user's multimedia presence* for stream synchronization. These two concepts define a new manner for combining streams coming from multiple sites.

1.2 Related Work

Distributed multimedia applications works on top of a various software and hardware layers, thus their performance depends on the appropriateness and speed of each of them. New network protocols, novel structures for operating systems, and new middleware have been the subject of significant research work recently in order to accommodate and better fulfill the requirements of this new and prominent category of applications.

At the networking layer two major challenges are multicasting and timely delivery. Multicast enables applications that provide service to thousands or even millions of users. Despite the research work on this area, there are still issues in deploying multicast in the Internet and achieving scalable and reliable data delivery. *Active Networks* have addressed the difficulties in developing and deploying new network protocols. It proposes that the Internet service model be replaced with a new architecture in which the network as a whole becomes a fully programmable computational environment. This is an interesting new concept; however, it leads to many technical and economical issues that have prevented its deployment. Timely delivery has been addressed by protocols like Resource Reservation Setup (RSVP) [12] and research on Differentiated Services. At the transport layer new protocols have been proposed to support multimedia applications. These include Real-Time Protocol (RTP) [64] and Scalable Reliable Multicast protocol (SRM) [26].

Nemesis [37] and Exokernel [34] are two recent operating systems that provide support for distributed multimedia applications. One issue addressed in these systems is the interference between applications while sharing a single processor. The load due to other applications influences the performance of each application. Multimedia applications require mechanisms to control this interference. Priority is believed to be inappropriate due to the cost in performing an analysis of the complete system in order to assign them. Their structure aims to uncouple applications from one another and to provide multiplexing of all resources, not just the CPU, at a low level.

Considerable research effort has been targeted to the development of middleware. Sometimes under the name of toolkit or frameworks, they all pursue similar objectives. Work on Multimedia Networking [44] aims to develop technologies and systems to enable multimedia applications over the Internet. It main focus has been audio (*vat* [32]), video (*vic* [43]), and whiteboard (MediaBoard [71]) applications. In order to exchange control information among applications at a given site, they use a *local conference bus* [43]. This resembles a computer bus. It is used to combine messages sent to the multicast group that would be sent otherwise by each component. It is also employed to carry out audio-video synchronization. Our middleware proposes an object-oriented model where the coordination between components is achieved via references to common objects rather than low-level message exchanges through a multicast group. On the other hand, while the conference bus approach allows applications to inter-operate, our middleware requires that they should be implemented in the same programming language. These applications do not support floor control, which makes them convenient for highly dynamic group meeting but might create interference in large-group sessions. An example is aggregated noise due to many open microphones. Our middleware provides the facilities to easily integrate floor control into any distributed application.

Some work has also focused on the support for tool sharing. One approach for tool sharing is to allow the sharing of the user's view of existent unmodified single-user applications. Examples of such systems are X Teleconferencing and Viewing (XTV) [1], Virtual Network Computing (VNC) [61], and Java Collaborative Environment (JCE) [2]. Another technique for tool sharing is to control the execution of multiple synchronized instances of the same application; examples are Habanero [27] and Tango Interactive

[33]. XTV allows one to share any X window application while JCE shares only Java applications. VNC achieves tool sharing by distributing the desktop as an active image. This means an image which users can interact with in a similar way they do with local application windows. This technique allows one to share visual component of any application. Due to the generality of this approach, we developed a protocol for transmitting this type of images, which we refer to as *dynamic images*. XTV, JCE and VNC use TCP as transport protocol. In contrast, our protocol works on top of unreliable multicast transport layer. This makes a crucial difference that lets our protocol be considerably more scalable than the other proposals. Habanero is a Java-based framework for synchronous and asynchronous collaboration. This framework facilitates the construction of software for synchronous and asynchronous communication over the Inernet. It also provides methods that developers can use to convert existing Java applications into collaborative applications. The system employs a centralized architecture and utilizes TCP connections between each client and the central server. Platform independence is gained by using Java. Our middleware shares with Habanero its object-oriented approach and programming language; nevertheless, our middleware supports a much general model for application sharing and higher scalability level. Tango Interactive is also a Java-based collaboration system, but unlike Habanero, it aims to the World Wide Web. Like Habanero, it also utilizes a centralized architecture, but in contrast, Tango collaboration modules are Java applets rather than Java applications. Our approach definitively differs from Tango in that our middleware runs on top of the network layer whereas Tango mainly runs within Web browsers.

Finally, another important multimedia system is Mash [44]. Mash is a comprehensive toolkit for multimedia communication and collaboration over the Internet using IP multicast. It evolved from a number of existing multimedia toolkits including Berkeley's Continuos Media Toolkit, MIT's VuSystem, and LBL/UCB MBone tools. Mash supports live media broadcasting, N-way conferencing, and session transcription and replay. More than a toolkit Mash has evolved as an application for the Internet. The available package does not clearly separate the toolkit from the applications already built and makes its reusability hard outside Mash research group. In order to overcomes this drawback and make an open and portable toolkit available to the research community, the Continuos

Media Middleware Consortium has been recently formed with the support of the National Science Foundation (NSF). While the overall objective of our middleware is inspired on the same needs addressed by the Mash Consortium, our midleware is Java-based rather than C/C++ and was designed and implemented from scratch. In contrast to our middleware that already offers a first version, the Mash Consortium is the recent creation with a first release scheduled for the second half of 2000.

1.3 Outline

The broad scope and numerous research issues faced in designing and implementing a middleware for multimedia applications prevented us from undertaking a comprehensive study of all of them. Rather, we elaborated on those that can contribute effectively to the ongoing research on multimedia applications [41] in the Computer Science Department of the Old Dominion University. As the four components of the middleware to be studied in detail, we selected a lightweight framework for floor control, a lightweight synchronization framework for multimedia, extension of operating systems network services to support interactive multimedia applications, and a resilient and scalable protocol for dynamic image transmission. Each of these components is elaborated and developed in Chapter II through Chapter VI. Then, in Chapter VII, we describe their implementation and use in a prototypical sharing tool application (*Odust*) that showed the effectiveness of our middleware. Finally, we present the conclusions of this research and further extensions of it in Chapter VIII.

CHAPTER II

LIGHTWEIGHT FLOOR CONTROL FRAMEWORK

Shared resource management has been studied widely in various contexts. From the firsts time-sharing systems to current multimedia collaborative applications many algorithms have been proposed for accessing mutual exclusive resources. As computer systems evolve in complexity and spread geographically, early algorithms' assumptions do not apply anymore and novel schemes are required. Traditional algorithms for distributed mutual exclusion, [35] [36] [59] [5], are not suitable for synchronous distributed multimedia applications. These algorithms fail to fulfill new design principles of multimedia applications, such as that all participants must be informed of the identity of the current user of the shared resource, that the algorithm should be scalable, and that it should work well for *loosely-coupled* sessions [26]. In this new context, the problem of accessing shared resources is commonly known as floor control [40], [63], [20]. Floor control is the mechanism by which users of distributed multimedia applications remotely share resources such as tele-pointers, video and audio channels, public annotations, or shared tools without access conflicts.

We distinguish the following characteristics or behaviors of floor control in synchronous multimedia applications:

1. All participants should be informed of the identity of the current floor holder. This contributes to member awareness.
2. While a new floor holder must be timely notified that she has been granted the exclusive resource, the other participants can tolerate some delay in this notification.
3. Any floor control scheme must easily accommodate joining and leaving of participants.
4. As the floor holder uses the exclusive resource, the session view changes for everyone. In other words, the use of the resource produces traffic that is delivered to the group.

5. Participants may have different roles. A role-based policy might apply for obtaining the floor. For instance, the session manager or teacher in distance learning systems [41] could obtain the floor by preempting the current holder. A registered student could obtain it by requesting it from either the current holder in first-come-first-served (FCFS) fashion or the session manager. The student could also snatch the floor from the current holder. Another policy could deny the floor to all not-registered students. Traditional algorithms for mutual exclusion assume FCFS. Malpani and Rowe introduce in [40] a moderator as a special participant who grants the floor. Schubert *et al.* support moderators and FCFS policies in [63].
6. The participants of a multimedia session are human beings, as opposed to processes. They trigger events that eventually become requests for exclusive resources such as audio or video channels, shared tele-pointers, or shared tools. Therefore, the frequency of requests and the expected response time for granting a resource depend on a human being's reactions and expectations, as opposed to a process response time in the case of distributed operating systems or distributed computing.

Floor control algorithms basically extend early techniques for distributed mutual exclusion by relaxing some requirements, such as a reliable multicast transport layer and explicit membership registration in order to support *lightweight sessions* [26] on the Internet. These sessions are based on multicast and lack of explicit control on session membership and centralized control of media sending and receiving. We also notice that, unlike mutual exclusion schemes, the access to floor-controlled resources involves changes in each member's view. To our knowledge, this characteristic has not been explored in the schemes for floor control.

We observe that exclusive resources can be classified according to their distributed nature and scheme for accessing a network channel, as shown in Fig. 2. Distributed modules that operate in exclusive fashion normally have a common network channel that must be used exclusively. For example, resources such as audio are available on each participant and must be activated in exclusive mode if only one participant has the right to deliver audio packets. Another is the case of localized and unique resources such as a shared tool. Here, guaranteeing exclusive access to the localized resource ensures exclusive access to the network channel.

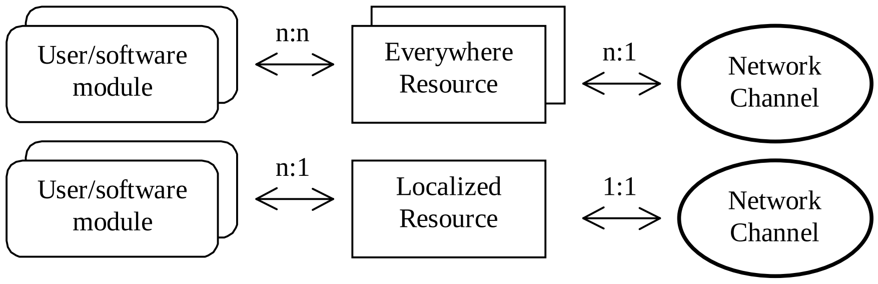


Fig. 2. Relation between users, resources, and network channels.

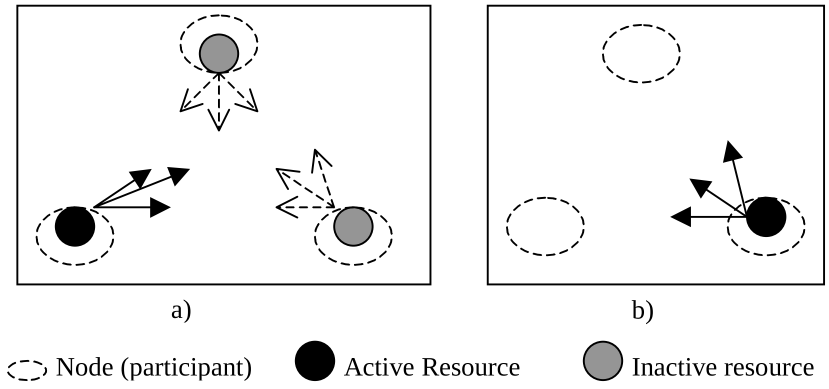


Fig. 3. Models for exclusive resources. a) everywhere resource, b) localized resource.

In Fig. 2 and Fig. 3, we model mutual exclusive resources as either everywhere or localized resources. *Everywhere resources* are those that transmit data to the group from the floor holder host. These include video and audio channels as in *vic* [40] and *vat* [32] and shared tele-pointers as in IRI [41]. A *localized resource*, on the other hand, resides in only one site from which it sends information to the group as the floor holder occupies the resource. Some examples are centralized-shared tools (as in XTV [1] and VNC [61]), group session replay, and site video camera. In short, the exclusive access to network channels is guaranteed by either activating only one everywhere resource or accessing localized resource exclusively. Based on this classification and recognizing different communication patterns, we propose two types of floor controllers to take advantage of the peculiarities of each class of resources.

2.1 Related Work

Fully distributed algorithms involve every participant in the process of requesting and granting access to the exclusive resource [59]. Although this principle could be valid for *tightly-coupled conferencing,* it does not scale well for *loosely-coupled* sessions with hundreds of participants. In sessions with a large number of participants, only some actually request access to the exclusive resource; therefore, we see an advantage in maintaining the floor-related packets to a minimum.

Some distributed algorithms assume reliable packet delivery to all the members of a session [59] [3]. This requirement cannot be efficiently fulfilled for large sessions with today's Internet protocols. In addition, distributed solutions rely on other services, such as total ordering [59] or clock synchronization [63], that are not always available or if so, they increase the cost of the protocol.

Centralized algorithms, on the other hand, are simpler. The requests are addressed to a central coordinator that manages the access to the resource. Their main disadvantage is that the coordinator becomes single point of failure. This criticism weakens though when we analyze the floor control facility in conjunction with the controlled resource. If the resource is localized and its host crashes, distributed algorithms would maintain a floor for which there is no resource and, thus, would cause inconsistency until the resource recovers. In case of everywhere resources, floor control resilience must be ensured, so that the other conferees are not affected by the failure.

Two proposals for floor control tools for lightweight sessions are Questionboard (qb) and Integrated Floor Controller (IFLOOR) [63]. Questinboard uses a moderator as central coordinator (moderator). Users multicast questions on one multicast group that the moderator joins. Their developers decided against unicast for this purpose because several sites on the MBone are behind firewalls, which have been configured to pass only class D addresses (range 224.0.0.0 to 239.255.255.255). The moderator assigns a global identifier (id) and multicasts the question to all. This acts as acknowledgment that the moderator received the question. If the user does not receive this acknowledgement, the question is resent. The moderator propagates questions reliably using a SRM-style NACK suppression scheme [26]. The moderator multicasts a *grant_floor* directive to all participants. This directive is resent in a heartbeat packet. Its transmissions are clustered in the interval immediately following a data transmission. The interval between two heartbeat packets varies from i_{min} to i_{max}, and it is doubled each time. Latecomers wait until a heartbeat packet and multicast a *send_all* request to the group. From the communication point of view, qb utilizes only multicast messages. In order to overcome packet losses, it uses packet retransmission, SRM scheme of reliability, and a periodic heartbeat packet.

On the other hand, IFLOOR is a distributed floor control tool that can be used with moderator or first-come-first-to-speak style. IFLOOR maintains a distributed speaker list that is robust in the face of lost packets, network partitions and disappearance of the moderator. A participant, who wishes to gain the floor, periodically sends a *requestfloor* message to the group. IFLOOR assumes that clocks are sufficiently synchronized to avoid disordering. A speaker removes itself from the speaker list by periodically sending a removal request. Before starting the communication a user needs to explicitly confirm its willingness to use the medium. If no such confirmation is generated during a pre-defined timeout period the moderator or the other members can assume that the member no longer exists and the next member on the list receives the right to use the medium. By looking at a locally maintained time-ordered speaker list, each session member can recognize if it is his turn to speak or who else has the floor. An announcer periodically retransmits heartbeat messages consisting of three components: number of entries of speaker list, last control message, and the name of the moderator if in moderated mode. If the announcer stops, each member schedules its heartbeat for a random time in the future. Upon collision, the announcer with the lower IP address stops sending. From the communication point of view, this tool uses only multicast messages and reliability is achieved by periodic packet retransmission.

2.2 Basis of the Lightweight Floor Control Framework

In this section we describe the basic objects of the floor control framework for exclusive resource management. Later we explain how this framework is integrated with specific resource components to provide floor control in synchronous multimedia collaborative systems.

In 1983, Ricart and Agrawala developed an algorithm for sharing a resource in mutual exclusion manner among several distributed processes [60]. Their algorithm is similar to fully centralized schemes in the sense that there is a unique process that coordinates the access to the exclusive resource. However, unlike traditional centralized approaches, the central coordinator moves with the floor. To access the resource a process multicasts a request message to the group, reliable multicast is assumed. A multicast message is needed since the location of the coordinator is, in general, unknown. After multicasting the request, the process waits for the floor. Once the floor holder releases the resource, it works as coordinator by determining the next process to use the resource and passing to it the floor and the coordinator's state. The state is basically a list of the requests already served. Every process multicasts its request, so that every process can maintain a local list with all the requesters. The coordinator, which is also the current floor holder, analyzes both lists to determine the next floor holder.

The Ricart's and Agrawala's algorithm utilizes a multicast message to request the floor and then a point-to-point message to grant it. We notice the algorithm could be adapted to work the other way around where clients send a point-to-point request message to the floor coordinator and the coordinator multicasts a grant message. In this variation, the coordinator could keep track of the lists of pending requests while processes could maintain the list of served requests. If the coordinator maintains both lists, every process would need to track only the current coordinator. Our framework implements the latter approach, and messages between floor requester and coordinator are sent through temporary point-to-point TCP connections [55].

We also notice the identity of the floor holder does not need to be delivered in a reliable and timely fashion to every participant. While a new floor holder should get a reliable and prompt The other participants could wait for the notification until the actual holder causes a change in their view of the system. This relaxation removes the need of reliable delivery to all the participants while accomplishing high responsiveness with the user requesting the exclusive resource. Our protocol periodically multicasts a *heartbeat* message with the identity of the floor holder. *Heartbeat* messages also convey the communication point with the coordinator (host, port) and the number of pending requests. In addition to overcome lost packets, the periodicity of heartbeats indicates the coordinator is up and running.

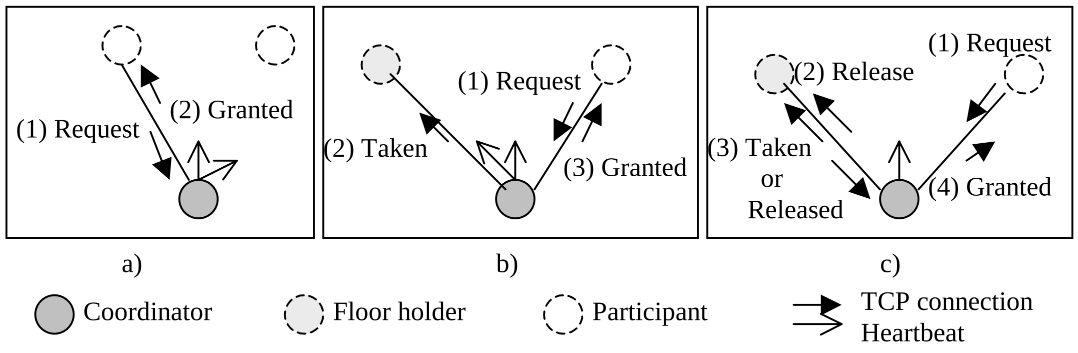


Fig. 4. Basic Floor Control Messages. a) Floor request when the resource is free, b) preemptive floor release, and c) delayed preemptive floor release

The protocol for requesting the floor involves two messages (as shown in Fig. 4a). Participants obtain the coordination’s location through the heartbeat messages. The negotiation to release the floor depends on the user’s role that comes with the request. Numbers with application dependent semantic represent roles. For example, 4 roles in a distance learning systems could be (1) teacher, (2) presenter, (3) student in intranet site, and (4) home student. In section 2.4, we explain how policies are associated to participants’ roles. Thus request messages can trigger the scenarios depicted in Fig. 4b or Fig. 4c. In b, the request is preemptive, so the floor is taken from the holder and granted to the requester. In c, the request is preemptive after a specified delay. Upon receiving this request, the coordinator notifies the floor holder with a *Release* message. If the holder does not release the floor after certain time, the coordinator snatches and grants it to the requester. Pathological cases such as dead are covered in Section 2.3.2.1.

2.3 Floor Control for Localized and Everywhere Resources

One missing entity in our basic architecture is the resource being shared. The floor coordinator could be conveniently placed depending on the resource location. For localized resources like a centralized tool sharing (XTV [1], VNC [61]), a room video camera, or replay and recording servers, the floor can be better managed from the same location where the resource resides. Other resources are distributed by nature. For example, an audio or video channel and a tele-pointer are resources that could be viewed, in most implementations, as replicated and going along with the floor holder. In our proposal, the floor coordinator migrates, as session members access the everywhere resource.

In short, we propose two architectures. One aims to support the localized resource model, and another targets the everywhere resource model. Both cases basically employ the same algorithm in terms of messages, but the communication structure changes.

2.3.1 Protocol Variation for Localized Resources

Floor control for localized resources places the coordinator on the resource's host. In general, the protocol works as described in section 2.2. In addition, some extreme situations demand further refinements. Firstly, the coordinator only accepts pending *Request* messages until a configurable limit and keeps a list with requesters' names. The rationale behind this is our desire to keep some control over the maximum amount of computing resources, such as memory and connections, bounded. Secondly, our framework for localized resources does not recover Coordinator's host crashes. As we mentioned in Section 2.1, electing a new coordinator does not make sense in this resource model, and we assume that the recovery of the resource must include the instantiation of a new coordinator. Upon restarting the coordinator, any pending request must be resent including the former floor holder request. On the other hand, if a participant host fails, recovery is easily achieved by restarting the participant since no state is kept at this site. The last case is a faulty floor holder host. The broken TCP connection with the holder signals the coordinator about this event. The coordinator takes the floor back and reassigns it. Thirdly, network partition only allows the resource's partition group to access it. Again, we assume that an external recovery mechanism handles the partition failure.

The coordinator guarantees the ordering. All requests arrive at the coordinator, which then grants the floor in first-come-first-served discipline for requests within the same policy. This protocol also ensures that only one member at most is given the floor. This comes from the fact that the coordinator does not grant the floor unless it has been either released by the holder or taken back by the coordinator. When no member is accessing the resource, the coordinator holds the floor.

2.3.2 Protocol Variation for Everywhere Resources

The floor control framework provides a floor control mechanism suitable for resources that reside everywhere but cannot be used simultaneously. An audio channel and tele-pointer are typical examples of such resources. Like Ricart's and Agrawala's algorithm, our framework moves the coordinator along with the floor for everywhere resources. As before, to request the floor, a member sends a request to the coordinator, then closes the connection, and waits for the floor. This message contains the requester's communication point to which the coordinator connects to grant the floor. The grant message conveys not only the right to use the floor but also the list of pending requests. This list is extended as new requests come. As soon as the floor moves to another member, this site takes the coordinator role and transmits heartbeat messages.

This protocol has to deal with a race condition not present in its version for localized resources. A request might reach a "coordinator" that has already granted the floor as shown in Fig. 5. The bold line in Fig. 5 indicates the coordinator role. B leaves this role as soon as it hands the floor to C, and C becomes the new coordinator. At *tr*, A's request reaches B, which relays the message to the new coordinator. B determines it by the last heartbeat after B's or whomever B granted the floor to. Since each request message carries the communication point of the initial requester, it does not matter who actually delivers the message.

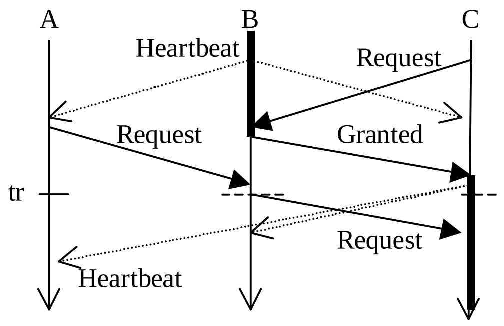


Fig. 5. Request message arriving at a former coordinator.

2.3.2.1 Recovery Protocol

Another issue, not present in localized resources, is the election of the first coordinator. This follows the same protocol as that of coordinator crash recovery. The absence of heartbeat reveals a missing coordinator. Regardless if it is a failure or members just joining a new session, the floor holder becomes "Nobody", and the aware participants initiate an election protocol. They all schedule heartbeats to start in t_{mce}; where:

$t_{mce} = n * t_{ce}/5$ for all former coordinator with $n < 5$,

t_{mce} = random (t_{ce}, $2t_{ce}$) in any other case, where n is the number of coordinators before itself. Members cancel their scheduled heartbeats on the arrival of someone else's before t_{mce}.

This assignation not only suppresses collisions but also gives more chances for former coordinators that we assume are more active in the session. In any case, collisions are resolved in favor of the highest (IP, port) coordinator's connection point. The port number breaks ties in IP addresses. A coordinator crash destroys the record of pending requests, so any pending request must be resent after heartbeat resumes. The protocol for recovering from faulty coordinators also applies to network partition.

A coordinator that cannot connect to a participant to grant the floor removes it from the list of pending requests and hands the floor to the next requester. This takes care of a member's host crash with pending request; otherwise, the participant's failure does not affect the floor management.

2.4 Floor Control Policies

Floor control policies refer to the negotiation for obtaining and releasing the floor. Two disciplines to obtain the floor are moderator-controlled access and first-come-first-served [40] [63]. Abdel-Wahab *et al.* propose in [3] a discipline where a request is granted only after obtaining permissions from any subset of the set {moderator, floor holder, resource}. Each resource defines the type of required permissions to be accessed. One difficulty with moderator-based is the introduction of a special type of participant that requires new patterns of communication. In order to keep our framework simple, the framework supports three basic policies, preemptive, delayed-preemptive, and non-preemptive. The policy might depend on the floor holder role, requester, and state of the coordinator. The mapping is done by a developer-defined method. In preemptive policy, the requester snatches the floor from holder. This policy makes sense in small conferences where social conventions can be used to control the floor especially in present of other media such as video. This policy also allows more dynamic sessions specially when there are multiple instances of the same resource such as audio channels. Delayed-preemptive policy allows the holder to keep the floor for a limited time after which the floor is snatched if the holder has not released it. This policy allows users to wrap up their contributions to the session before releasing the floor and is thus suitable for users that do not want to rudely take the floor from the holder but at the same time want to ensure the floor holder cannot keep the resource indefinitely. Finally, non-preemptive policy sends a *release* to the holder, but it is up to the floor holder to release it. For example, in distance learning sessions the teacher might want to keep her resources as long as she wishes. The three policies can be derived from the delayed-preemptive policy by changing a *timeout* (t_{out}). $t_{out}=0$ implies preemptive floor request. By representing infinity by $t_{out} = t_{outMax}$, we implement non-preemptive policy.

Floor control framework relies on a user-implemented interface (*fcfPolicy*) to determine the policy for getting a floor back from a holder, in other words, the timeout for the current holder. This interface implements three main methods: *requestNotification*(), *withdrawalNotification*() and *holderTimeout*(). The first two are invoked when new requests arrive at the coordinator and a pending request is cancelled respectively. The coordinator invokes the third one to either notify the holder on remaining time or take the floor back. For convenience, the floor control framework provides a class (*fcfBasicPolicy)* that implement the following method for timeout:

holderTimeout(requesterHigherPriority, holderAny, n) $= 0$ preemptive policy,

holderTimeout(requesterNormPrority, holderAny, n) $= t_{outMax}$ for $n = 0$,

$= -(t_a - t_b)*(n-1)/(N-1) + t_a$ for $0 < n \leq N$ and

$= t_b$ for $n \geq N$, or

holderTimeout(requesterLowerPriority, holderAny, n) $= t_{outMax}$ non-preemptive policy.

Where t_a, t_b, and N are constants, and n is the length of the pending request queue.

Developers of applications that use this framework must decide the role sent by each participant process in a floor request message. Then the *fcfPolicy* uses this role to determine the policy for requesting the floor from the holder. For instance, the teacher could snatch the floor from anyone else, but the application could also let her assume a student role and request the floor in a delayed preemptive manner.

So far we have discussed the issue of obtaining the floor back from its holder. Another point is how the floor is assigned to prospective members with pending requests. Again, our framework relies on the developer-implemented interface (*fcfPolicy*) mentioned above to determine who obtains the floor next. In addition to the three methods already described, this interface implements the function *selectNextHolder*(), which returns the name of the member to be granted the floor. The *fcfBasicPolicy* class keeps separate queue for each type of user, i.e. *HigherPriority, NormPriority, and LowerPriority*, and maintains a FCFS order within each queue. The selection is thus done by choosing a member from each queue in round-robin fashion among queues. This discipline ensures bounded waiting.

2.5 Basic Object-Oriented Architecture of the Floor Control Framework

The fundamental architecture of the floor control framework is based on a centralized approach. A central point of control makes easy the coordination among multiple requests. Like traditional centralized algorithms, whenever a participant needs the exclusive resource, a client sends a request message to the coordinator. Then, the coordinator sends back a reply granting permission depending on the floor policy.

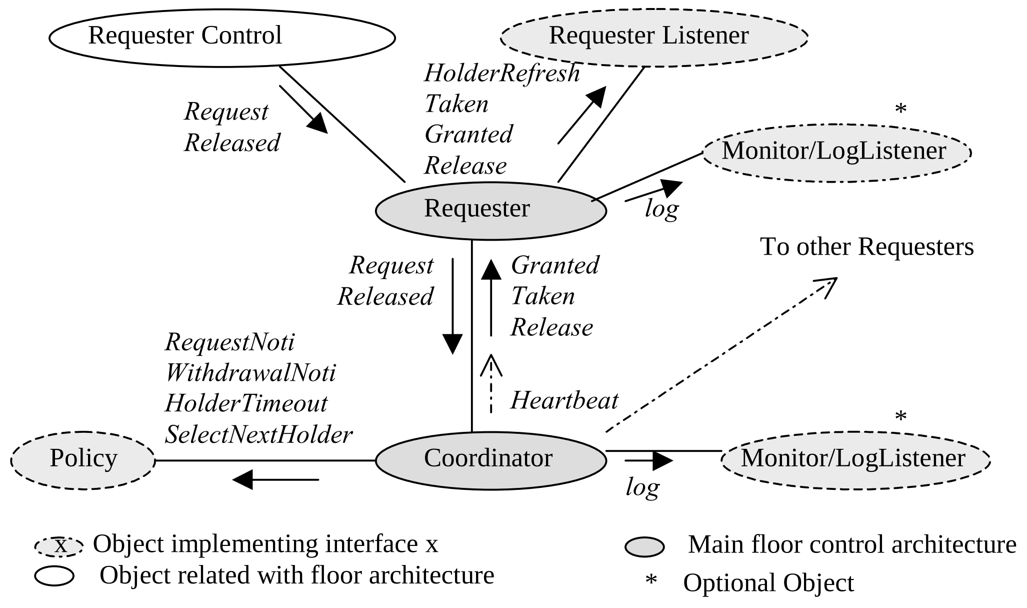


Fig. 6. Basic Architecture for Floor Control in Synchronous Multimedia Collaborative Systems.

The basic architecture involves 5 types of objects, as depicted in Fig. 6. There is one centralized floor coordinator (*Coordinator*) and one floor requester per session member (*Requesters*). Each *Requester* is related to a *Requester Control* and a *Listene*r. The *Requester Control* is an application specific object that invokes the *Request* and *Released* methods of the *Requester*. It could be a Graphics User Interface (GUI) object or a higher level object that controls access to compound resources. The *Requester Listener* receives status updates and the replies of the *Requester Control'*s messages. The updates are sent each time that *Requester* object detects a change of the floor holder. An application dependent class should implement the *Requester Listener* interface to update either the GUI associated to the floor or a higher-level floor control for compound resources. In any case, the class for *Requester Control* could also implement the *Requester Listener* interface.

Each *Requester* relays the messages coming from the *Requester Control* to the *Coordinator,* receives messages from it, processes and delivers them to *Requester Listener* when they represent a change of state. The *Coordinator* notifies all the potential floor requesters the identity of the current holder. Each *Requester* keeps track of changes in floor holder and notifies them to its *Requester Listener. Requesters* know about holder changes through the periodic remotely invoked heartbeat method. As we described in section 2.4, the developers must also set a *Policy* object that defines the discipline for taking the floor back and passing to other participants.

Finally, we include an optional monitor/log listener. Its purpose is to process messages that denote important changes in the state of objects and indicate error conditions.

2.6 Inter Object Method Invocation

The floor control architecture relies on three types of message delivery. Direct message invocation is used to call methods on local objects. Examples of such a method are *log* and *HolderRefresh*. The other two are types of remote method invocation. A permanent datagram multipoint connection is kept between the *Coordinator* and each *Requester*. This channel (IP address, port number) is passed as argument to the constructor of the *Coordinator* and *Requester*. Any other connection parameter is configured at runtime and notified on the fly to all participants over this multipoint channel through *heartbeats*. One piece of information received from this channel is the unicast communication point with the *Coordinator*. Thereby, a TCP connection can be established every time a participant triggers a floor request event. Examples of messages delivered over this connection are *Request*, *Granted*, *Release*, and *Taken*. An immediate choice for remote method invocation is Java Remote Method Invocation (Java RMI) [73]; unfortunately, Java RMI prevents us from measuring and controlling the bandwidth associated to these connections. Traffic measure and control become essential for inter-stream adaptation. Also, Java RMI works on TCP connections which makes it inapplicable for multi-point invocations where multicast is used, such as for *heartbeat* invocation. Therefore, in order to make the framework scalable and eligible for data rate control, we decided against Java RMI and use message exchanges via reliable point-to-point connections or unreliable multicast channels.

2.7 Floor Control Architecture for Localized Resources

The placement of the *Coordinator* on the same machine conveys several advantages when the mutual exclusive resource is located in a fixed place, meaning that the resource stays on one machine. First, the communication between the floor and the resource is easily accomplished. In an object oriented model, references allow access to *Coordinator* and to the resource through a well-defined interface. Second, if the resource is dynamic, i.e. instances can be destroyed and new created, it is easy to create instances of *Coordinator* as needed. Finally, the co-location of the *Coordinator* and the controlled resource allows us to use the *Requester-Coordinator* link for passing resource specific information at runtime for the session member to access the resource. For example, in centralized implementations of sharing tools systems, such as XTV [1], the floor holder needs to know the communication point of the tool to operate it remotely. This piece of information can be conveyed along with *Granted* reply. In addition, in synchronous multimedia applications most of the resources we target distribute their traffic to the same audience over multi-point and possibly unreliable channels, so co-location lets developers use an implementation of the floor control architecture that utilizes the resource's multi-point channel for delivering coordinator's heartbeat. Following the sharing tool example, the *Coordinator* can use the resource–participants' link for sending heartbeats. A drawback of this scheme is the multiplexing required at the receiving sites. On the other hand, it reduces overhead and bandwidth consumption by reducing the total number of messages and connections. We elaborate more on this idea in Section 2.9.

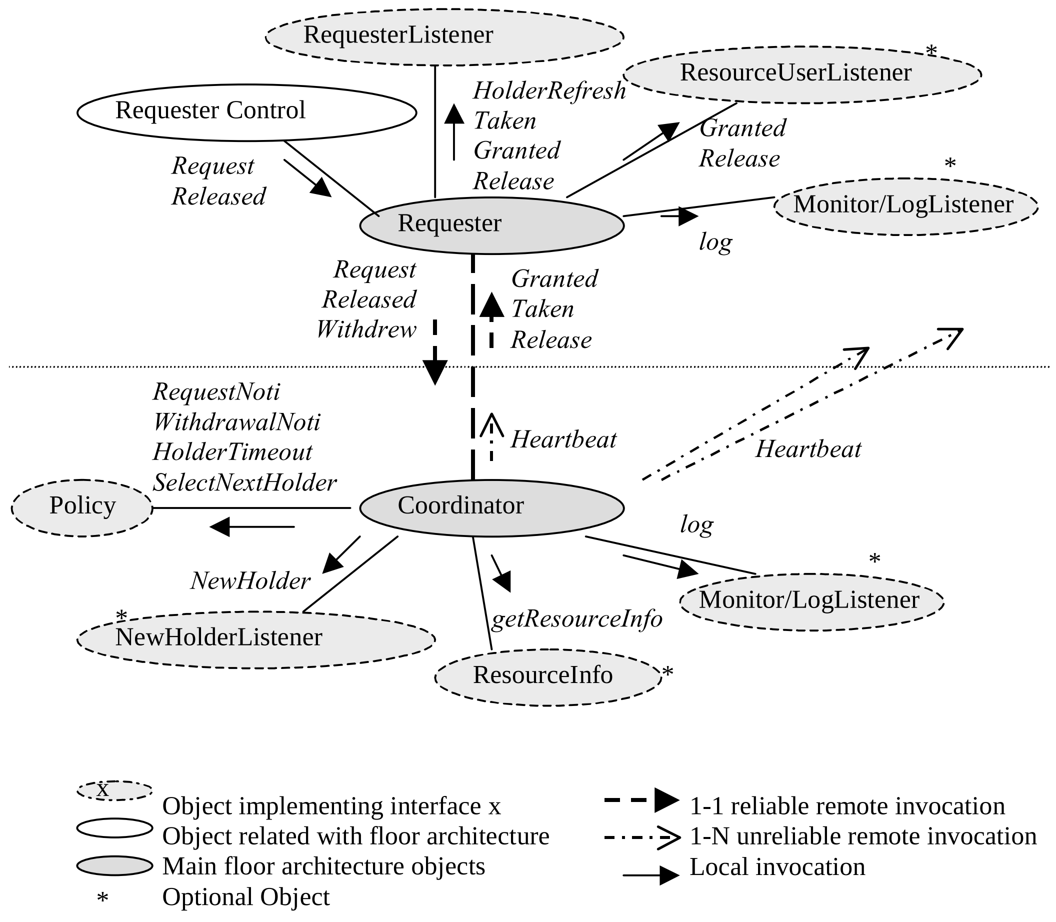


Fig. 7. Floor Control Architecture for Localized Resources.

A *Coordinator* must be created for each mutual exclusive resource, and a *Requester* should be created for each session participant as well. The algorithm works as described in section 2.2. Five optional objects are related with the architecture*: Resource User Listener*, two instances of *Monitor/Log Listener*, *Resource Information*, and *New Holder Listener*. In a floor-based system, once the application obtains a floor, it needs to notify the appropriate module waiting for it. In our architecture, application developers can use the events received by *Requester Listener* or set a *Resource User Listener*. The former one is more suitable for GUI updates though. The latter requires the implementation of only two methods: one for starting and another for stopping the use of the exclusive resource. For instance, after getting the floor to control a shared tool, the local application should notify the module responsible for letting the user operate the shared tool. By providing this hook, we accomplish floor control architecture independence from specific applications. The two instances of *Monitor/Log Listener* and *Policy* object were already described in section 2.2. In some applications a floor holder needs resource specific information in order to access it. For such cases, we have included a hook for an object that implements the *Resource Information* interface. If this object has been set, the *Coordinator* invokes *getResourceInfo* method and appends a resource specific payload to the *Granted* message. Upon receiving this message the *Requester* passes this piggyback message to the *Resource User Listener* as part of the Granted message. Going back to the shared tool example, a client could receive thus the service access point where it can connect to operate the shared tool. Finally, in some applications, it is convenient to have a hook on the *Coordinator* for receiving events indicating changes of the floor holder. This might be necessary to update the resource owner member who might not receive the *Heartbeat* message, for example, when loop back has been disabled. This way the resource owner can also be informed on the current floor holder.

The messages between *Requesters* and *Coordinator* are exchanged through a semi-permanent TCP connection. Here semi-permanent connection means it lasts from the time the *Requester Control* triggers the *Request* message until the *Requester* releases the floor or the *Coordinator* takes it back, whatever comes first. One concern is the scalability of this approach since very many members might want to request the floor at one point of time. Developers can limit the number of Coordinator open connections and therefore bound the number of pending requests. The rationale behind this decision is that in some situations the user whose request was accepted satisfies the need for which the others also want to request the floor. In addition, heartbeats convey the number of pending requests, so users could suppress their requests in face of a fast increase in pending requests after a situation that makes many desire the floor.

2.8 Floor Control Architecture for Everywhere Resources

Some exclusive resources are more efficiently managed if they fully work from the floor holder host. A shared tele-pointer, for example, performs better if its implementation works from to the node where the pointer holder is, as opposed to a centralized implementation where the pointer holder controls it remotely and a centralized object multicasts the updates to all the participants. The latter option increases delay and bandwidth consumption due to an increased number of messages. Other examples of distributed resources are video and audio channels, and a shared pen for public annotations.

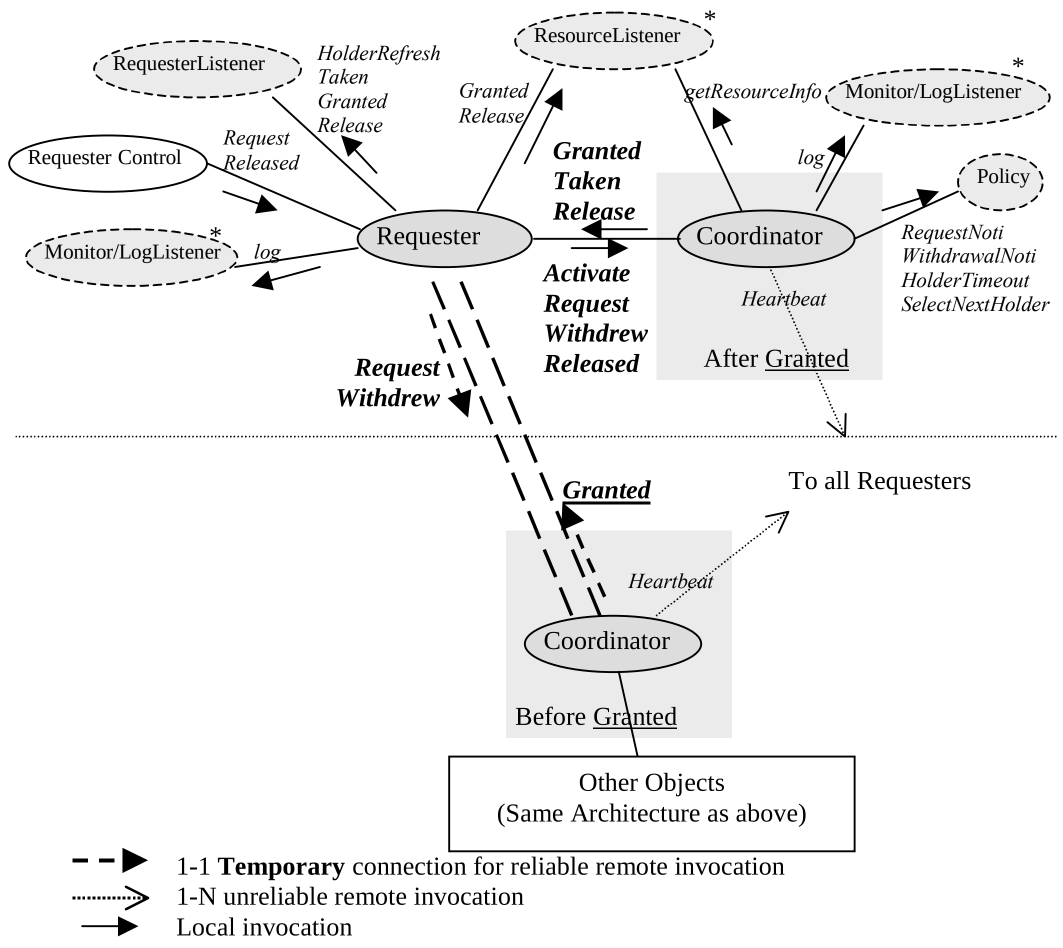


Fig. 8. Floor Control Architecture for Everywhere Resources.

The architecture depicted in Fig. 8 shows the objects that implement the floor control protocol for everywhere resources as described in Section 2.3.2. The architecture does not change much from that for localized resources. The main modification is the migration of the coordinator from one floor holder to another. The TCP connection established between *Requester* and *Coordinator* is now temporary. As we described in section 2.3.2 temporary TCP connections are established to send the *Request* message and then another to send the *Granted* message. Permanent or semi-permanent connection cannot be used since the *Coordinator* moves as the floor moves. It is not shown in Fig. 8 the request message that a former *Coordinator* might receive. It forwards this request to the latest Coordinator based on its local information.

Activate message is the only new message in this protocol. Once a requester receives a Granted, it triggers this method on the local Coordinator. The Granted and Activate messages hand the list of pending requests as argument.

2.9 Integration of Resource and Floor Control Multicast Channels

The integration of the resource and floor control multicast channels reduces the number of messages, bandwidth, and network overhead at the cost of a multicast channel multiplexer. As we have observed before, the floor control framework places the *Coordinator,* that multicasts heartbeats, from the same machine from where the resource traffic is multicast to all the session participants. By noticing the presence of two related multicast channels that reach the same audience, we identify an opportunity for performance gain in some applications. Heartbeat messages are relatively small and can be combined most of the times with resource dependent packets without exceeding the network maximum transmission unit (MTU). The price paid for this integration is the necessary multiplexing in order to discriminate between floor- and resource-related packets. In our framework, *Coordinator* and *Requester* include two data members, one of which is responsible for the delivery of *heartbeats* to the group as shown in Fig. 9. A *Controller* is in charge of sending heartbeats periodically. It normally sends this scheduled event by calling *fcfDeliver* method every t_{beat} seconds unless the heartbeat is sent along with a resource-related packet. The latter case occurs when a loadPlayload message is invoked between $t_{beat}/2$ and t_{beat} after the last *heartbeat. loadPlayload* method is called when *PeriodicCarrier* has a message to send and offers to deliver a heartbeat along with it. The value for t_{beat} determines the upper bounds for the time latecomers learn about the floor holder and the time lost packets are recovered. The Coordinator as described in previous sections does not use *loadPlayload* method since all the multicast messages are sent through *fcfDeliver* calls. However, a resource-related object can implement the interfaces of PeriodicCarrier and fcfReciver and pass them as constructor arguments to accomplish the integration of both floor control and resource multicast channels.

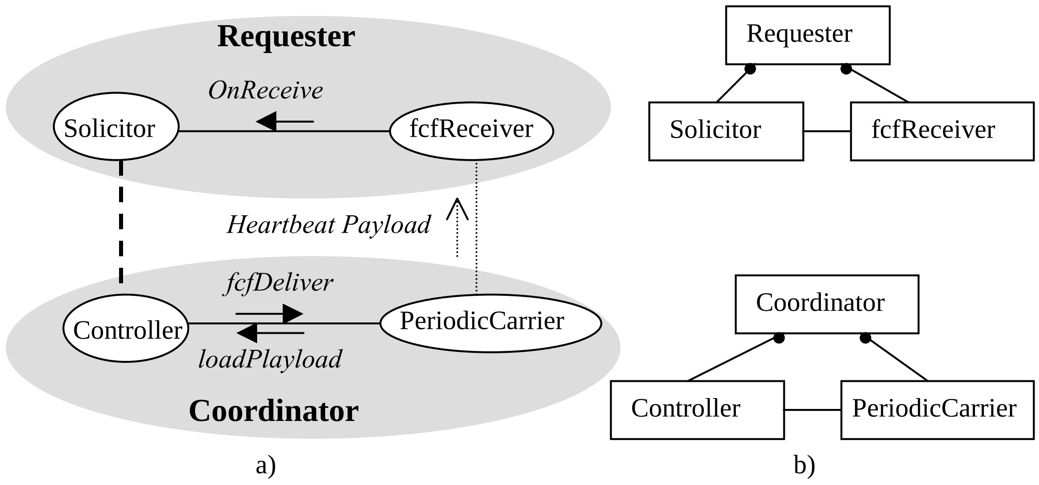


Fig. 9. Requester and Coordinator data members. a) Object link diagram and b) Class relationships.

CHAPTER III

MODEL FOR STREAM SYNCHRONIZATION

The principles of application level framing and integrated layer processing proposed by Clark and Tennenhouse [14] and transport protocols such as Real Time Protocol [64] have driven modular designs where each media or logic component is delivered through an independent stream. Regardless of the transport layer employed, these streams have to be synchronously played out or rendered at receiving sites in order to perceive a consistent and truthful view of the scene and actions at the sender site. The main objective of stream synchronization is to faithfully reconstruct the temporal relationship between events. Stream synchronization can be subdivided into intra-stream synchronization and inter-stream synchronization. While the former refers to preserving temporal relationships of data within a stream, the latter deals with the temporal dependencies across streams.

Multimedia application can be classified as off-line or on-line depending of whether sender sites have access to the entire stream or just up to a point shortly before transmission. While video on demand and multimedia library retrieval are examples of the former case, applications that involve real-time human-to-human interactions are examples of the latter case. Since interactive applications naturally ensure synchronicity at the sender, here stream synchronization focuses on the reconstruction of the temporal relationship at receiving sites without access to future references.

Time model defines the time references used by synchronization algorithms and the terms involved in the synchronization conditions, which is the time condition that data units must satisfy. Normally it includes time references from the sender to the receiving machines. We extend it to include delays outside the reach of application, such as sound waive air propagation. For example, the propagation delay due to a loudspeaker located at 15 m away from a user is 15 milliseconds. Synchronization can be achieved by associating time with data units at senders and then preserving the same time relationship at the receiving sites, but with a delay to absorb unavoidable delay and jitter of the transmission path. At receivers, data units are buffered and played out after a fixed delay. Even though sequence number and timestamp have been used to evaluate the synchronization condition, the flexibility of timestamps for expressing a richer variety in media semantics has made it the preferred approach for media synchronization. Multimedia applications can use a fixed a-priori transfer delay bound when the underlying transport protocols provide it as quality of service, or they can measure the experienced delay on-line and adapt to it in order to reduce delay while maintaining stream synchronization within humans' perception. We propose a generic adaptive timestamp-based algorithm that can be tailored to meet synchronization constraints of each media.

Inter-media synchronization imposes requirements that extend the scope of a single stream. A common approach is to use a globally synchronized clock to relate streams coming from multiple sites. While some studies assume this condition as preexistent [48], [62], others include mechanisms for clock differences estimation within their algorithms [4], [56]. We propose a different session view model that does not require synchronized clock for combining multiple streams.

3.1 Synchronization Model

Our synchronization model derives from our understanding of people's perception of the world. We are familiar with the patterns we have perceived since our childhood and any variation from them catches our attention and sometimes annoys us. For instance, we expect a relationship between what we hear and what we see when someone is speaking. Although this is a subjective matter, certainly a delay of a second between both is annoying. On the other hand, everyone is used to hear a thunder several seconds after seeing the lightning and the delay between the two gives us an idea of how far the event was. Thus, we introduce the idea of a *virtual observer* placed in the scene to define the temporal relationship that should be preserved within and between different streams triggered by the same event.

The case of multi-user multimedia sessions brings up interesting new issues when trying to produce a session view for its participants. Multimedia sessions combine scenes taking place in geographically distributed sites and for which we have to propose a model for the integration of participants' presence. The main difficulties each model tries to overcome are communication delays between sites and their variations. One proposed model attempts to simulate a "face-to-face" encounter by synthesizing at each site a unique view for all session members (e.g. [23] and [62]). Hereafter, we refer to this model as *global synchronization model*. This is accomplished by equalizing all session participants' delays to the maximum delay observed among sites, so the communication path appears to have given the same delay to all flows and they are synchronized at their destination. The main advantages of this approach are that every receiver member has the same experience similar to that of being in the same room with other receivers and that global ordering is guaranteed. The main drawback is the unavoidable worst case delay imposed on all receivers. To illustrate this shortcoming, let us present a couple of case scenarios. Consider a tightly-coupled session with three participants: Edward in Norfolk (VA), Rod in Virginia Beach (20 miles away from Norfolk), and Austin in Berkeley (CA). Clearly the link between Virginia and California defines the worst case delay for all participants, so when Edward speaks, Rod has to face a fictitious long delay and its associated buffering to equal Austin-Edward's delay. The level of interactivity between Edward and Rod has been deteriorated to offer Austin the same experience as Rod and Edward. Consider now a loosely-coupled session where a teacher gives a lecture to students spread over the state of Virginia. Students connected through high bandwidth links to the instructor's site are unfortunately downgraded to experience the delays of remote students attending the class from home, not to mention if someone wants to join the session from oversea. As a result, what the global synchronization does is to place each sender process at a position equidistant from each other, where the delay between them is the worst case delay between any two processes. We could think of the session member as being at the vertexes of an equilateral triangle or a pyramid in the case of 3 and 4 participants respectively, see Fig. 10.

Another drawback of the global synchronization model is the need of synchronized clock among session participants. Some proposals are based on synchronized network clocks (e.g. [23] and [62]), and others include in their protocols mechanisms for achieving this by the use of feedback messages [56] or probe messages[4].

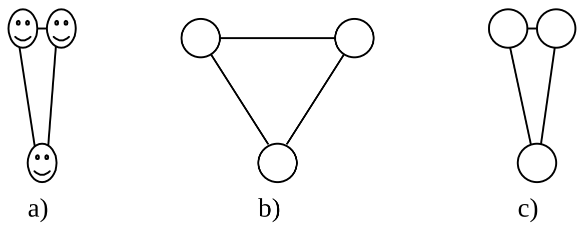


Fig. 10. Synchronization model for integrating three geographically distributed scenes. a) Edges showing physical network delays, b) global synchronization, and c) differentiated synchronization.

Another way to combine each participant's presence in the session is to equalize only the flows produced or controlled by a site participant to a common delay. In other words, the flows produced or controlled by a member are presented in a synchronized fashion to the other session members according to individual sender-receiver link delays. We refer to this model as *differentiated synchronization model*. Differentiated synchronization attempts to meet all users' expectations on the synchronization of the flows originated from or controlled by an end user. The set of flows produced or controlled by a participant constitutes the *multimedia presence* of that member in the session.

While the synchronization requirements of multimedia presence are well defined by the principle of virtual observer mentioned above, it is not clear to us what is the more natural way to synchronously combine multiple multimedia presences based on what a human being would expect. Our differentiated synchronization model imposes a delay between two end users that depends only on the quality of service (QoS) offered by their communication channels. As a result, each user's session view differs from that of others due to the distinct end-to-end latency with which each multimedia presence is rendered at each site. Like in long distance telephone calls or telephone conferences, the end-to-end delays of multimedia presences reflect the processing and transmission times from senders to receiver, and the delay lower bounds are only limited by the technology in used between a receiver and the current senders. The level of interactivity measured in terms of end-to-end delay is highest for a given communication QoS. This is in contrast to the global synchronization model where the protocol adds extra delays to equal the view of the largest delay link.

A review of the two case scenarios presented before shows that differentiated synchronization provides higher level of interactivity and responses according to the location of the interacting members rather than the total membership. A drawback of differentiated synchronization is the lack of global ordering. This could lead to unexpected race conditions. For example in a three-site distance learning session, suppose the teacher is in one site and student A and B in the two other sites. While teacher answers student A's question, it might happen that a long delay between A and B makes B hear the teacher's response before A's question. Our experience with the Internet indicates that this situation, although possible, is unlikely. Global synchronization equals all flows' delays preventing this type of inconsistency. The teacher hears the question at the same time as all the students do; therefore the response will always follow the question for everybody.

An important advantage of differentiated synchronization is that it can be accomplished without relying on globally synchronized clock [78]. Like many other problems in distributed systems, a globally synchronized clock greatly helps and simplifies many solutions; however, with current computer technology, a global clock requires some type of feedback that we have chosen to avoid while synchronizing multiple streams. We hope that in the future computers will come equipped with Global Position System (GPS) that provides absolute computer location and time as well, so there will be no need for computer network based solution for global clock synchronization.

3.1.1 Time and Delay Model

Time and delay model refers to the time components involved from the generation of the original phenomena until these are perceived by human beings at receivers. We distinguish four time components in the end-to-end media delay as depicted in Fig. 11.

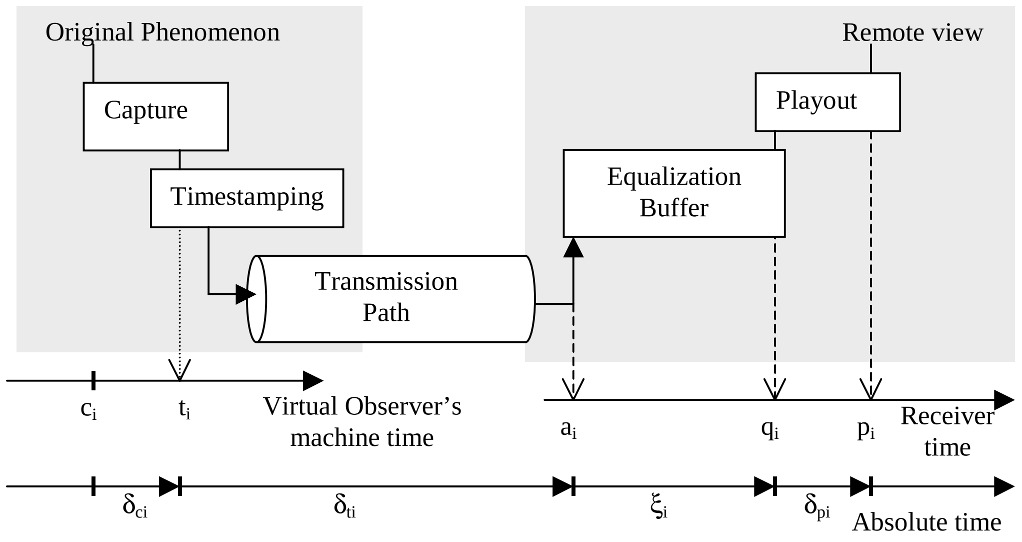


Fig. 11. Components of delay model.

For most of the media, the original phenomenon takes place at the same machine where the virtual observer captures the media presence we wish to synchronize at the receivers' sites. This is the case of audio and video streams. Other streams might be controlled remotely by a user and, therefore, do not originate at the virtual observer's machine; yet they should be synchronized at the receiving sites, as they are related flows and part of a user's multimedia presence. For example, assume a session running a centralized shared tool, such as XTV [1]. The floor holder operates a shared tool remotely and after some delay sees the responses to her commands. The virtual observer at the floor holder machine relates the changes in the shared tool with the audio, video, and possible other streams when the commands responses are displayed on the floor holder machine. Thus, the shared tool stream should be synchronized with any other locally generated flows, such as audio and video streams. For instance, while sharing a web-browser, the virtual observer captures the reactions – face expression and any exclamation - of the user in synchronicity with the appearance of an image or text on the browser.

XTV illustrates a family of systems that implement a centralized architecture to accomplish collaboration. In these systems, the time the media capture takes place is not relevant for synchronization, but the time the controlling user perceives the results of her manipulations, or more precisely the time the virtual observer perceives the scene at the controlling user site. From now on, we will use just the term observer rather than virtual observer when no confusion is possible. Thus, we define:

c_i: perception time; when the observer perceives the scene produced or captured by packet i,

t_i: time a timestamp is assigned to a packet i,

a_i: arrival time of packet i at the synchronization module in the receiver,

q_i: delivery time of packet i to be played out at the receiver, and

p_i: time the end user perceives the result of media packet i.

We do not assume clock synchronization between any pair of machines, but we do assume the difference of simultaneous times reported by any two machines is constant. In Section 3.2, we relax this condition by noticing that differences in time due to drifting are negligible in comparison to inter-packet time. While c_i and t_i are times reported by the clock at the observer's machine, a_i, q_i and p_i are times reported by the clock at the receiver's machine. Please notice that t_i is not the timestamp of the i^{th} packet, but just the time at which it is assigned. The value of the timestamp is really c_i since it is the time the scene was perceived by the observer and, therefore, the inter-stream synchronization relationship the algorithm must preserve at the receiving sites. The perception time c_i could take place before or after the timestamp is assigned to packet i. For example, for audio and video stream, the perception times precede the timestamping time; on the other hand, for applications controlled remotely, the perception times might succeed the timestamping time depending on where packets are transmitted to the session from. In any case, timestamp determination falls outside of the boundary of the synchronization protocol, since it depends not only on the media stream but also on the peculiarities of the application's implementation.

The synchronization algorithm sets the arrival time of a packet when the packet is handed to it. Even though it is not shown in Fig. 11, some processing might take place between the arrival of a packet at a machine and its entry to the synchronization algorithm. In fact, the more indetermination moves before synchronization, the better the resulting playout is. For instance, consider an audio stream transmitted with a Forward Error Correction (FEC) mechanism [10], [53]. Here a number of audio packets are buffered to correct and compensate packet loss; thus this delay can be assimilated when the processing is performed before the equalization module; otherwise, its delay needs to be reflected in the playout time ($\boldsymbol{\delta_{pi}}$). The ith data unit leaves the equalization module at time $\boldsymbol{p_i}$. This instant should be such that the playout of the data unit reproduces the same time relationships that the observer perceived some time ago. In Fig. 11, we have also defined the following time intervals:

$\boldsymbol{\delta_{ci} = t_i - c_i}$ timestamping delay of packet $\boldsymbol{i}$,

$\boldsymbol{\delta_{ti} = a_i - t_i}$ communication delay of packet $\boldsymbol{i}$,

$\boldsymbol{\xi_i = q_i - a_i}$ equalization delay of packet $\boldsymbol{i}$, and

$\boldsymbol{\delta_{pi} = p_i - q_i}$, playout delay of packet $\boldsymbol{i}$.

The timestamping delay cannot be made zero even though it takes negative or positive values depending on the application. Because of the lack of real-time support in general purpose operating systems and communication channels, $\boldsymbol{\delta_{ci}}$ cannot be determined precisely in most of the cases; however, it can be well estimated most of the times by the application. For example, $\boldsymbol{\delta_{ci}}$ for an audio packet $\boldsymbol{i}$ of m samples can be estimated by the time it takes for the audio device to capture m samples at the sample rate specified by the application. Because the application has control on the sample rate and packet size in samples, it can easily estimate the time that the first sample was captured. Although it is not the time the observer heard that sample since the sound wave propagation and other computer related time, such as context switching times, have not been taken into account, it does capture the major portion of $\boldsymbol{\delta_{ci}}$ [1]. Since we assume the

[1] Let's assume sample rate = 8 KHz, m=256, and difference in distance between virtual observer and mike from speaking user equals to 1 [m]. The time from the capture of the first to the last sample is 256/8000 = 0.032 s = 32 ms . The difference in air propagation times between mike and observer is 1 m /

observer's and receiver's machines' clocks progress at the same and accurate rate, the timestamping delay measured by the observer's machine does not differ from that measured on an absolute time line. The communication delay, δ_{ti}, is unknown and variable, and it cannot be estimated with only one-way measurements, so our synchronization algorithm does not rely on it but on preserving differences between scenes while controlling the equalization delay average. The equalization delay, ξ_i, is the only component of the end-to-end delay touched by the synchronization algorithm in order to recover the temporal relationship within and across streams at playout time. Finally, the playout delay, δ_{pi}, must be taken into consideration to accomplish both intra- and inter-stream synchronization since, in general, the playout delay depends on the packet to be rendered. In video, for example, a frame with high spatial or temporal redundancy can be decoded much faster than frames with high entropy, so ensuring synchronization at equalization delivery time is not enough because the temporal relation might be destroyed during rendering. Unfortunately, there is no way the payout delay could be determined by any computer algorithm in the general case. Assume the destination user hears the audio coming out of a loud speaker located somewhere in the room. Although the electronic delay of perhaps amplifiers and audio processing equipment might be negligible, the propagation time of the sound wave in the air might reach tens of milliseconds for normal rooms. As a result, we rely on playout delay estimates furnished by the application, perhaps from the user interface, to advance the equalization delivery time so that synchronization can be accomplished for end user perception.

Playout delay plays an even more important role for inter-stream synchronization. While it might not vary much from data unit to data unit causing little problem for intra-stream synchronization, it varies considerably from media to media. Compare the time it takes to decompress and display a 320x240 pixel video frame with the time it takes to start playing a 256 sample audio packet. The amount of data moved and the decoding

340 m/s = 3 ms. Therefore, unless there is a high priority job blocking the audio thread or process, the sample accumulation time predominates.

complexity make the video stream playout delay larger than audio stream's. Thus, if we concentrated in just achieving synchronization after the equalization module, we would end up with audio being played ahead of video. The previous observation suggests that playout variations and any other indeterminations should be placed before the equalization module whenever it is possible and other tradeoffs, such as buffer space and module encapsulation, allow them.

3.1.2 Synchronization Condition

Based on the above time and delay model, we can now state the condition for preserving temporal relationship of a user multimedia presence to a receiving end user. We say that two concurrent or sequential events, ***i*** and ***j*,** are played out synchronously if and only if for two systems with fixed clock offset:

$$c_i - c_j = p_i - p_j \qquad (1)$$

While the two events belong to the same stream in intra-stream synchronization, they pertain to different streams in inter-stream synchronization. A stream is played out fully synchronized if (1) holds for any pair of events. Synchronicity is not the only concern of end users and synchronization algorithms. Total end-to-end delay and buffer space are other two important considerations. The former impacts directly in the reachable level of interactivity, and the latter one defines a performance index of algorithms. By making the end-to-end delay "big enough", it is possible to achieve synchronization for any pair of events at the price of buffer capacity. This has been the principle behind algorithms that accommodate a fixed delay for all packets [58]. These applications have been classified as rigid applications [15]. By reducing the buffering capacity, we reach a point where some packets arrive so late that they cannot be scheduled for playout and, at the same time, hold (1) with their predecessor event. Protocols and algorithms that use this approach have been refereed to as adaptive applications. The best effort services offered by today's Internet make adaptive protocols or algorithms the only viable choice for interactive applications.

An interesting property of equation (1) is that it does not depend on synchronized clocks. The left side represents the time between the two events as perceived by the virtual observer and measured by the observer's machine clock. The right side is the time between the palyout of the same events as perceived by a receiving end user and measured by her machine clock. As we have assumed that the two clocks might only be off by a constant amount but progress at the same rate, each difference is independent on the clock offset and network delay.

Theorem 1: The synchronization condition ($c_i - c_j = p_i - p_j \;\; \forall$ events i and j), is equivalent to $p_i = c_i + \Delta$ where Δ is a constant.

Proof.

$$\begin{aligned}
&p_i - p_j = c_i - c_j \\
&p_i - c_i = p_j - c_j \quad \forall\, \mathrm{i} \geq 0,\ \mathrm{j} \geq 0;\ \text{in particular} \\
&p_i - c_i = p_0 - c_0 \quad \forall\, \mathrm{i} \geq 0.\ \text{Now, by defining } \Delta \equiv p_0 - c_0 \\
&p_i - c_i = \Delta \\
&\therefore p_i = c_i + \Delta
\end{aligned}$$

We call this constant, Δ, virtual delay. It represents the total playout delay when all the clocks are synchronized, and it differs in an arbitrary amount when the virtual observer's and receiver's have a constant offset.

Theorem 2: The synchronization condition ($c_i - c_j = p_i - p_j \;\; \forall$ events i and j), is equivalent to say that all events are played out with the same absolute delay.

Proof: Let C_i be the time reported by an absolute and "true" clock for the event **i** at the observer's site, and let P_i be the analogous time for events at receiver's site as illustrated in Fig. 12.

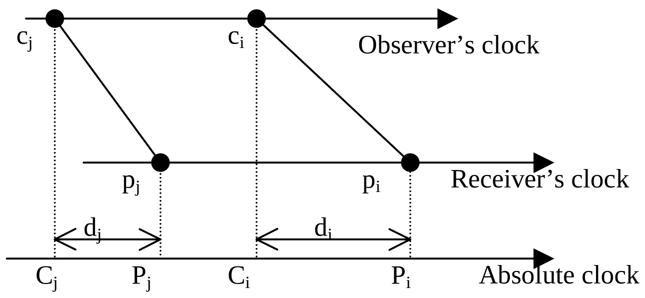


Fig. 12. Constant delay playout.

By measuring all times on an absolute clock, we have:

$$P_j = C_j + d_j \tag{2}$$

$$P_i = C_i + d_i \tag{3}$$

Now, by subtracting (3) – (2), we have:

$$P_i - P_j = C_i - C_j + (d_i - d_j) \tag{4}$$

Since we assume clocks do not drift $P_i - P_j = p_i - p_j$ and $C_i - C_j = c_i - c_j$. Then (4) becomes:

$$p_i - p_j = c_i - c_j + (d_i - d_j)$$

Finally, by using synchronization condition $p_i - p_j = c_i - c_j$, we conclude:

$$d_i = d_j \qquad \forall\, \mathrm{i} \geq 0,\ \mathrm{j} \geq 0$$

For intra-stream synchronization another equivalent synchronization condition, perhaps more intuitive, is $c_i - c_{i-1} = p_i - p_{i-1} \quad \forall\, \mathrm{i} \geq 0$, as illustrated in Fig. 13.

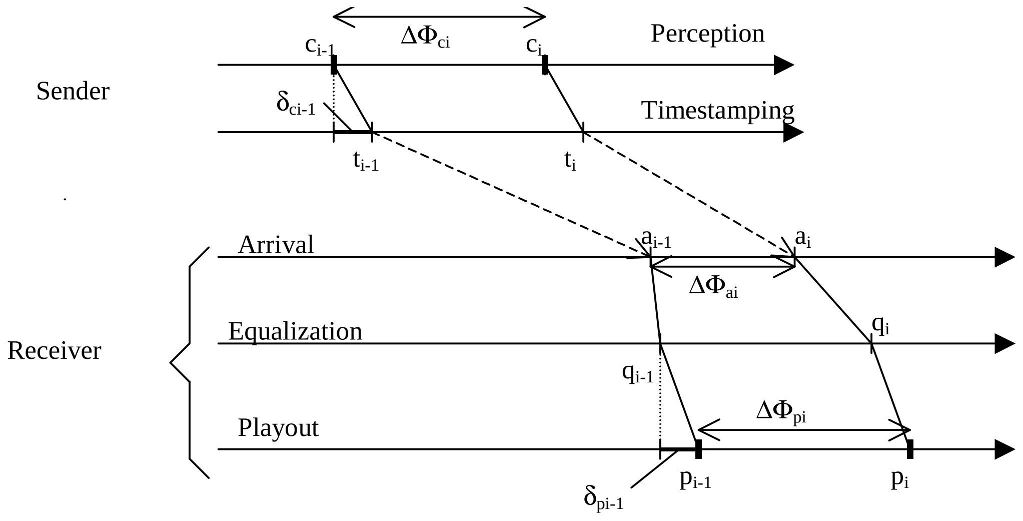


Fig. 13. Time diagram for two consecutive events.

This condition is also equivalent to (1) since we can easily sum the terms from event j+1 to i and have:

$\sum_{k=j+1}^{i}(c_k - c_{k-1}) = \sum_{k=j+1}^{i}(p_k - p_{k-1})$, and developing both series, we reach to

$c_i - c_j = p_i - p_j \quad \forall\, i \geq 0,\, j \geq 0$

Another way to obtain p_i is as a function of arrival time, equalization delay, and playout delay:

$$p_i = f(a_i, \xi_i, \delta_{pi}) = a_i + \xi_i + \delta_{pi} \qquad (5)$$

By using Theorem 1 and (5) we obtain:

$$p_i = c_i + \Delta = a_i + \xi_i + \delta_{pi}$$

$$\therefore \Delta = a_i + \xi_i + \delta_{pi} - c_i \qquad (6)$$

The arrival time, playout delay, and perception time are out of the control of the synchronization algorithm, and since Δ is considered constant, the equalization delay is the only variable that the algorithm controls to make the right side of (6) constant. Another way to write (6) is

$$\xi_i = \Delta - (a_i + \delta_{pi} - c_i)$$

Since the equalization time, ξ_i, cannot be negative,

$$\Delta \geq a_i + \delta_{pi} - c_i \quad \forall \, \mathrm{i} \geq 0 \tag{7}$$

From (7), we determine that the lower bound for the virtual delay, Δ, is $\max\{a_i + \delta_{pi} - c_i, \mathrm{i} \geq 0\}$. The only way to meet the synchronization condition for all pair of events is by selecting the virtual delay according to (7). This can only be guaranteed if the application is based on guaranteed service [24] whose service commitment is based on a worst case analysis. Clearly, in on-line interactive applications running over best-effort services, it is not possible to determine the virtual delay *a priori* since it requires knowledge on arrival times of all packets. Thus, a compromise needs to be made between the number of events that will miss the synchronization condition and the value of the virtual delay.

3.2 Relaxing the Synchronization Condition

As described in Section 3.1.2, allowing a big value for virtual delay solves the problem of intra- and inter-media synchronization. This solution, though, leads to long end-to-end delay and big buffering requirements. While the latter requirement might be fulfilled, large latencies are highly inconvenient and annoying in interactive collaborative applications. Thus, a number of adaptive algorithms that change the end-to-end as network conditions vary have been proposed (e.g.[62], [15], [78], and [70]). Obviously, this cannot be done without relaxing the synchronization condition stated in Section 3.1.2. For Theorem 1, this condition is equivalent to playing at time $p_i = c_i + \Delta$ each data unit perceived at time c_i. In other words, a change in equalization delay to accommodate shorter end-to-end delay and to reduce buffering brakes the synchronization condition between the data units before and after the change. At this point, it is important to take into consideration the characteristics and semantic of the multimedia streams to evaluate the impact of such adaptation.

Changes in the delay multimedia streams are played out have different impact depending on the streams. Audio in teleconferences presents periods of audio activity or talkspurts followed by periods of audio inactivity during which no audio packet are generated. In order to faithfully reconstruct speech at a receiving site, all the packets of a talkspurt should be played out preserving the same time relationship at generation time. Even though this is nothing but our synchronization condition that in principle applies to all data units, this condition is not that critical during periods of silence. In other words, there is some flexibility in the time constraint between the last data unit of a talkspurt and the first of the following one. This time can be shortened or lengthened with no major impact on the audio quality perceived by end users. On the other hand silence periods in music -if any- have a different semantic: their time relationship should be treated the same as active periods. Thus, for audio stream we propose adjustments in the virtual delay during silence periods or after some timeout to account for continuous audio streams. Another important point on computer-played audio that differs from other media is that audio applications control the playout only partially, as opposed to fully controlled media playout such as video. Due to its high frequency (up to 44 KHz for CD quality), audio sample playout is actually done by hardware, so applications cannot increase the rate at which packets are submitted to the audio device in order to reduce latency. As opposed to video frames, these audio packets will be played out at the same predefined rate and any difference in latency will have been moved from the synchronization algorithm to the audio device. As a result, in order to reduce audio distortion, upward delay adjustments stream can be made by artificially introducing a gap (i.e. delaying the playout of the next and following data units), and downward delay adjustments can be made by reducing silence periods or discarding audio samples or whole packets. In any cases, corrections during silence periods are preferable, and the developer can decide in other cases. For example, the use of FEC in audio stream might defeat any intention of delay reduction by dropping packets because the playout mechanism will try to regenerate back the discarded packet with no reduction in the total delay.

Compared with audio stream, video streams present different semantic. At receiving site, data unit are processed and displayed frame by frame by the application, and the frame period is normally greater than the playout time in order to leave processing resources for other computations. Thus, the end-to-end delay can be adjusted by inserting artificial gaps or reducing inter-frame playout time. Packet discarding has also been proposed as an option for reducing end-to-end delay; however, we decided against it in the general case, and we leave developers to determine the policy to be used. Due to the fact that video compression techniques create inter-frame dependencies in order to remove temporal redundancy, the elimination of video data triggers a reduction of quality for a number of code-related frames. Thus, we conclude that changing the playout is sufficient and an effective and efficient way to vary the end-to-end delay.

In addition to video and audio streams, multimedia applications transport non-continuous data streams that also require synchronization at receiving site. These include data streams generated by shared applications, tele-pointers, slide shows, and whiteboards. As we discussed in Section 3.1, the main goal of our multimedia stream synchronization algorithm is to faithfully reconstruct the temporal relationships perceived by a virtual observer, and at the same time, to notice the synchronization accuracy need not exceed the limits of human perceptions. This perception is media dependent. While for audio a sample rate of 44 KHz is enough to capture human hearing frequency range, we perceive smooth and continuous motion when frames are captured at a frequency around 30 Hz. If we compare now the temporal relationship between an audio and video stream for humans speaking (also called lip synchronization), we perceive them synchronized when their time relationship is within ±160 ms [69] [68]. These intra- and inter-stream synchronization requirements change notably when we analyze data streams. In this case, it is still desirable to reconstruct the temporal relationship between consecutive data unit, but now the synchronization condition can be relaxed since it is much more difficult for the receiving user to detect lack of synchronicity. For example, there is little expectation for the time between two consecutive pages in a shared browser or the position updates for a tele-pointer. On the other hand, we cannot neglect completely the time relationship for non-continuous media because they convey an important component for users' *awareness*, and what is more critical inter media semantic information needs to be presented in a coherent fashion. In this context, awareness is the ability to know or infer what the others are doing. For example, in lip synchronization, audio and video dependency is clear because one hears and sees two manifestations of the same phenomenon; however, tele-pointer and audio relationship is established only when the presenter relies on the tele-pointer to complete an idea only partially expressed verbally. In addition to differences in synchronization requirements, there is another property that distinguishes most non-continuous media from video and audio. While the state of audio and video is ephemeral, most non-continuous media states form part of a user's view for longer time; thus, data unit discarding must be avoided in order to create a more accurate view. For instance, while using a whiteboard in free hand drawing, we might tolerate some transitory synchronism inconsistency caused by a late delivery, but it would be undesirable to have missing parts of the drawing caused by a discarded data unit. Finally, when the application relies on a reliable transport layer, packet discarding is not an option at all.

3.3 Delay Adaptation and Late Packet Policies

Differences in the temporal semantic and relationship among audio, video, and data streams suggest the use of special policies for adapting end-to-end delay for these media streams. A common specification among these three, however, is a policy for treating late data units; i.e. data units arriving later than their delivery time. We consider three late packet policies to be chosen by developers: late packet discard, resynchronization, and late delivery. The first policy, already discussed in Section 3.2, basically updates internal statistics and dumps the packet. Resynchronization is an increase in the virtual delay to make a packet meet the delivery time with minimum equalization delay. Late delivery just hands the late packet for playing out with minimum equalization delay regardless of any loss of synchronization it might cause.

In addition to late packet disciplines, virtual delay adjustments are performed to reduce excessive delay and to accommodate virtual delays to achieve inter-stream synchronization. We consider upward adjustment policies and downward adjustment policies. While the first ones increase the virtual delay, the second ones reduce it. We propose two policies for virtual delay reduction: Early delivery and oldest packet discard. Early delivery policy reduces the virtual delay in at most the time remaining for the oldest packet to be played. It works by reducing the scheduled time of the oldest packet in the queue. Oldest packet discard is similar to early delivery; however, it might remove the oldest packet from the queue and reschedule the remaining oldest if necessary in order to accommodate a given change in virtual delay. There are two reasons, our proposed synchronization algorithm might enlarge the virtual delay: as a result of the late packet policy or inter-stream synchronization correction. In the latter case, we enlarge the virtual delay by introducing gaps, and in the first case it is governed by the late packet policy.

3.4 Model and Metrics for Buffer Delay Adjustments

The main objective of the intra-stream synchronization algorithm is to play out each packet at a fixed end-to-end delay that translates to a fixed virtual delay in our approach and, at the same time, to maintain the equalization delay “reasonably” low. The difficulty here is to be precise on how much is reasonable. In [78] Xie *et al.* specify a *maximum synchronization buffer delay*, which would be equivalent to a maximum equalization delay in our terminology, and a *maximum object discard ratio* as a synchronization QoS parameters. Two advantages of these parameters are that the algorithm ensures a bounded delay and buffer space, and in face of low delay variations the algorithm ensures a ratio of discarded packets less than a given maximum value. While the second parameter is intuitive, the first one is difficult for a user to estimate. Moon *et al.* dynamically collect the absolute network delay statistics - more precisely the distribution of packet delays - and set the end-to-end delay according to a given percentile [48]. Xie’s maximum object discard ratio behaves like the percentile defined in [48] with the exception that while the discard ratio is computed periodically in [78], the delay distribution is dynamically updated by using a sliding window technique. Ramjee *et al.* use estimates for network delay mean and variation to schedule a packet playout time based on its timestamp, and these two estimates [57]. Stone and Jeffay in [70] took a different approach. Their queue management policy called *queue monitoring* defines a threshold value for each possible queue length. Where the threshold value for queue length n specifies a duration in packet times after which the *display latency* can be reduced without increasing the frequency of late packets. They define the display latency as the total time from acquisition at the sender to display at the receiver. By choosing shorter thresholds values for longer queue lengths, their algorithm approaches minimal latency. Again, the main drawback of this technique is the difficulty for determining the threshold values. In one way or another, these techniques try to absorb network delay variations while keeping the equalization delay low.

The statistics of the total delay of data units from their perception by a virtual observer to their arrival at the synchronization algorithm plays an important role in determining either the equalization delay or the total end-to-end delay from observer's to receiver's perceptions. As discussed in Section 3.1.2 the absolute end-to-end delay value is neither relevant nor required to accomplish synchronization, but it must be ideally constant or have only variations outside human's perception granularity. Thus, only uncontrolled delay variations need to be absorbed by the equalization delay to produce a total constant.

Next we show how to determine the statistics of the absolute delay from data unit observer's perception to arrival at synchronization algorithm from receiver's measurements and how to relate it with the equalization delay. By looking at two data units from their perception to their arrival at synchronization algorithm, as show in Fig. 14, we observe:

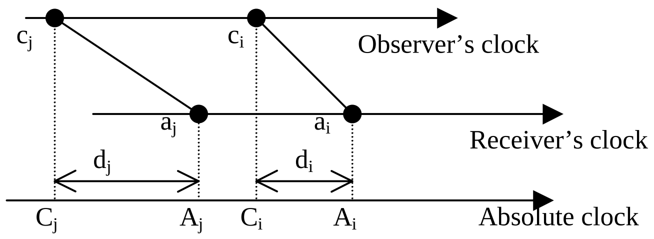


Fig. 14. Arrival times relationship for two data units.

As before, we use uppercase to indicate absolute "true" time for corresponding lowercase events that take place at the virtual observer's or receiver's site.

$$A_i = C_i + d_i \tag{8}$$

$$A_j = C_j + d_j \tag{9}$$

Subtracting (8)-(9), we have:

$$A_i - A_j = C_i - C_j + d_i - d_j$$

If clocks do not drift:

$$a_i - a_j = c_i - c_j + d_i - d_j$$

And grouping terms:

$$a_i - c_i - d_i = a_j - c_j - d_j \quad \forall \text{ i} > 0, \text{ j} > 0.$$

In particular, and by defining $\Gamma \equiv a_0 - c_0 - d_0$,

$$a_i - c_i - d_i = \Gamma$$

$$\therefore \; a_i - c_i = d_i + \Gamma \quad \forall \text{ i} > 0 \qquad (10)$$

Therefore, the difference between arrival time and perception time is equal to the absolute delay between the two events plus a constant regardless of the offset between observer's and receiver's clock. This is an expected result since this difference combines the delay and clock offset (Γ). Let μ_d and σ_d^2 be the mean and variance of the absolute delay between observer's perception and data unit arrival, then:

$$\mu_{a-c} = \mu_d + \Gamma \qquad (11)$$

$$\begin{aligned}\sigma_{a-c}^2 &= E\left\{(d_i + \Gamma)^2\right\} - E^2\{d_i - \Gamma\} \\ &= E\left\{d_i^2 + 2d_i\Gamma + \Gamma^2\right\} - (\mu_d + \Gamma)^2 \\ &= E\left\{d_i^2\right\} - \mu_d^2\end{aligned}$$

$$\sigma_{a-c}^2 = \sigma_d^2 \qquad (12)$$

From (11) we confirm that network and processing delays and clock offset are indistinguishable in one-way measurements at receiving site, and (12) shows that one-way measurements are sufficient to compute the variance of these delays. Now using (6) we have:

$$\sigma_\xi^2 = \sigma_{a-c}^2 + \sigma_{\delta_p}^2 + 2\left(E\left\{\delta_{p_i}(a_i - c_i)\right\} - E\left\{\delta_{p_i}\right\}E\{a_i - c_i\}\right) \qquad (13)$$

In most of the cases, data units tend to be of similar size for a media; thus, we could assume that δ_{p_i} and $a_i - c_i$ are independent; therefore their covariance is zero and $\sigma_\xi^2 = \sigma_{a-c}^2 + \sigma_{\delta_p}^2$. Otherwise, we expect certain degree of correlation existing between

δ_{p_i} and $a_i - c_i$ resulting in a positive covariance. Thus, we can say, in the general case that:

$$\sigma_\xi^2 \geq \sigma_{a-c}^2 + \sigma_{\delta_p}^2$$

Relation (13) can be rewritten as:

$$\sigma_\xi^2 = \sigma_{a-c}\sigma_{\delta_p}\left(\frac{\sigma_{a-c}}{\sigma_{\delta_p}} + \frac{\sigma_{\delta_p}}{\sigma_{a-c}} + 2\rho_{(a-c,\delta_p)}\right),$$

where $\rho_{(a-c,\delta_p)}$ is the correlation coefficient between $a_i - c_i$ and δ_{p_i}, and given that $\left|\rho_{(a-c,\delta_p)}\right| \leq 1$ [28]:

$$\sigma_\xi \approx \sigma_{a-c} \text{ if } \frac{\sigma_{a-c}}{\sigma_{\delta_p}} >> 1 \qquad (14)$$

Practical results in the Internet indicate that variations in playout delay are at least one order of magnitude smaller than network delay variations. For example, in [67] Sisalem *et al*. conclude that delays of video playback vary between 17 and 35 ms for JPEG on Sun Ultra, and in [9] Bolot shows round trip delay variations in the order of up to several hundreds of milliseconds in the Internet. Thus, we use (14) to estimate a "good" virtual delay such that the number of late packets (data units) be controlled. This is graphically illustrated in Fig. 15.

Based on Fig. 15, we can analyze some known techniques for controlling a fixed end-to-end delay using our time model. Ramjee *et al.* would estimate $\mu_{a-c+\delta_p}$ and $\sigma_{a-c+\delta_p}$, then they would set $\Delta = \mu_{a-c+\delta_p} + 4 * \sigma_{a-c+\delta_p}$. Using Moon *et al.* technique, we would collect data in a 10,000-packet sliding window, synthesize the p.d.f. as plotted in Fig. 15a, and set Δ to a given percentile point. Xie *et al.* define three regions based on Δ and ω, which is a QoS parameter: for $a_i - c_i + \delta_{p_i} < \Delta$ the object (packet) waits, for $\Delta \leq a_i - c_i + \delta_{p_i} \leq \Delta + \omega$ the object is playout immediately, and for $a_i - c_i + \delta_{p_i} > \Delta + \omega$ the object is discarded. The condition for changing Δ is based on the number of packets falling within each of these regions during a window of around 800 packets.

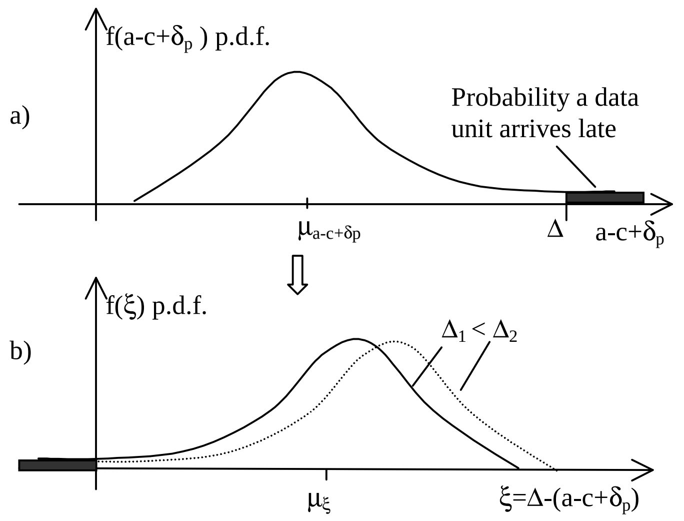


Fig. 15. Tradeoff between the virtual delay (Δ) and the number of late packets and its relationship with $a-c+\delta_p$ variations. a) Total uncontrolled delay p.d.f. and b) equalization delay p.d.f.

While simplicity is an advantage of Ramjee *et al*. proposal, it might be too conservative or tight depending on the factor multiplying the variation estimate. A factor of 4 might not represent well the trailing region of the p.d.f. in all the cases. Xie *et al*. and Moon *et al*. try to control the number of late packets. While this criterion is one of the most important, Xie *et al*. solution does not continuously follow changes in the delay statistics (it reacts only after collecting enough data) and Moon *et al*. solution requires a relative long time to reach a steady state.

We propose to estimate directly and continuously the proportion of packets arriving late rather than counting and comparing them against thresholds. The total number of packets and the number of late packets are two monotonically increasing functions for which we estimate and compare their rate of change using a linear filter. We illustrate this idea in Fig. 16.

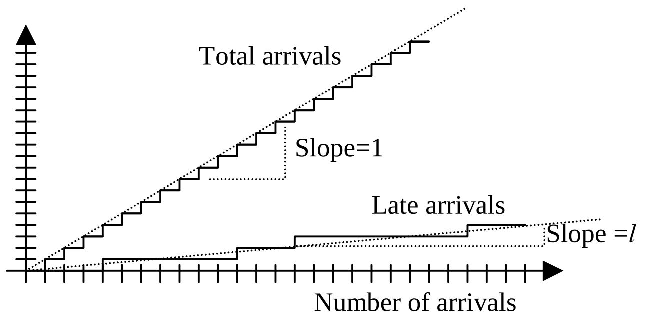


Fig. 16. Accumulation of late packets.

Let L(n) be the accumulated number of late packets after a total of n arrivals. The proportion of late packets is the instantaneous rate of change of L(n):

$l = \frac{\partial L}{\partial n}$, Which can be estimated in Appendix A. There we obtained:

$$l_i = \alpha l_{i-1} + (1-\alpha)(L_i - L_{i-1}) \tag{15}$$

Where $-1 < \alpha < 1$ and $L_i - L_{i-1} = \begin{cases} 1 & \text{if packet i is late} \\ 0 & \text{otherwise} \end{cases}$

By comparing l_i against a threshold, we know whether to increase or decrease virtual delay. As mentioned before, the virtual delay is the result of sender and network related delays plus equalization and playout delays. Since playout delays are difficult to determine on data unit bases, hereafter we assume that the playout delay is constant (δ_p) for all data units pertaining to a stream, and that it only varies from media to media. Let the equalized delay, denoted by *d*, be the total delay between the perception and delivery times. In Chapter IV, we develop algorithms for determining *d* given a desired bound for the percentage of late packets and describe selected traces collected on the Internet that show behaviors of delay times that are relevant for synchronization.

3.5 Stream Synchronization in Translators and Mixers

New multimedia applications and protocols are trying to support the tremendous heterogeneity of the Internet. Differences in bandwidth, network support, and multimedia hardware, create the need for intermediate systems that connect two or more homogeneous systems. Translators and mixers are two types of such systems. The distinction between them is that while translators pass through the streams from different sources separately, mixers combine them to form one stream. The transformation made by either translators or mixers normally involves changes to the synchronization information of each packet. This transformation cannot be made in isolation of other stream related in time. Here we propose a model and mechanism to recreate synchronization information in these streams so that the receiving sites can still synchronize them.

Changes in the synchronization information done by translators are safe for synchronization since these changes can be made indistinguishable from data processing done at sender sites. On the other hand, by combining streams from multiple sources, mixers change the original synchronization information and multi-stream synchronization is required at the mixer site in order to regenerate synchronization information for the combined stream.

For example, assume that a mixer combines two incoming audio streams into one outgoing stream that is sent to a user at home connected through a modem. If one of those audio streams is to be synchronized with a video stream, the new audio timestamp must be related with the video timestamp.

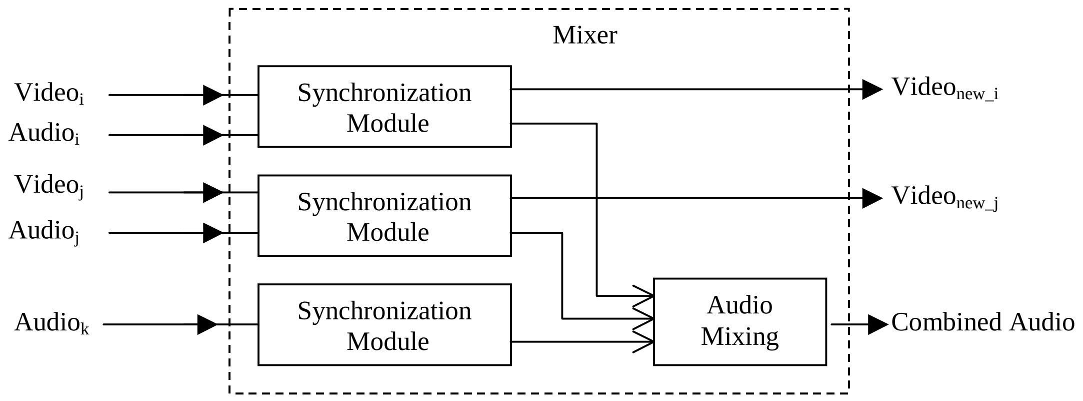


Fig. 17. Regeneration of synchronization information in mixers.

In order to be able to synchronize stream belonging to the same multimedia presence, any change in the synchronization information of one stream forces the regeneration of the timestamps of the others. This can be achieved by synchronizing related streams and then assigning new timestamps based on a new reference. For example, Fig. 17 depicts the basic model and mechanism for mixing three audio streams for which the are two video streams related in time. Here the combined audio stream is assigned new timestamps based on the mixer time reference, and video streams change their timestamps as well to reflex the time relationship with the combined audio stream. Indeed, mixers perform the same synchronization task as receivers; nonetheless, they forward the resulting streams rather than rendering them.

3.6 Clock Skew Estimation and Removal

Clock skew estimation and removal has application in both inter-stream synchronization and qualitative measures of performance of the computer networks. Here, we focus on techniques for converting times reported by multiple clocks to a unique reference clock as required by our synchronization algorithms presented in Chapter IV. We use the receiver's clock as time reference.

This problem has already been discussed in [51] and [52]. These studies estimate and remove the clock skew based on a set of pairs (tr^i, ts^i), where tr^i is the timestamp of the i-th packet arriving at the receiver according to receiver's clock, and ts^i is the timestamp of the i-th packet leaving the sender according to sender's clock. When dealing with inter-stream synchronization on-line, we see the need for a technique that dynamically refines the estimator as new packets arrive.

3.6.1 Model

We first define the terminology used to formally state the problem and explain our approach.

C_s: Sender clock,

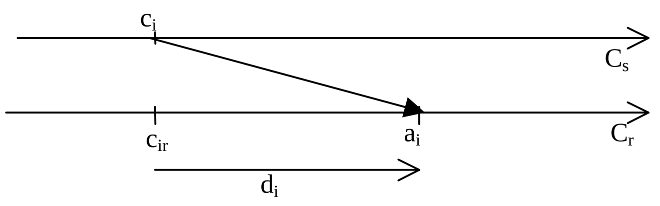
C_r: Receiver clock,

c_i: timestamp of the i^{th} packet leaving the sender according to C_s.

a_i: timestamp of the ith packet arriving at the receiver according to C_r.

c_{ir}: timestamp of the ith packet leaving the sender according to C_r.

m: time period of C_s according to C_r.



The problem can be stated as follows:

Given a sequence of pairs (c_i, a_i), convert each pair to $(\tilde{c}_{ir}, a_i)$; where $\tilde{c}_{ir}$ is the sender timestamp of the packet i according to the receiver clock, c_{ir}, except by a fixed offset.

In one-way measurements it is not possible to determine the exact relationship between the sender and receiver clocks because there is always a fixed offset that cannot be distinguished from a portion of the network delay.

We assume that the sender and receiver's clocks might differ in offset (t_0 = constant) and skew ($m \neq 0$) but do not drift (m = constant). Thus we can establish the following relationship between the two.

$$c_{ir} = m\,c_i + t_0$$

Receivers have only access to the arrival time of each packet and the expected value of the sender clock period (m_0).

If the receiver converts sender timestamps to its clock using the expected clock frequency ($\bar{c}_{ir} = m_0\, c_i$), the resulting delay grows proportional to the clock skew.

$$a_i - \bar{c}_{ir} = m\,c_i + t_0 + d_i - m_0 c_i = (m - m_0)\, c_i + t_0 + d_i$$

Therefore, an estimate for c_{ir} is necessary to remove the clock skew. We refer to it as $\tilde{c}_{ir} = \hat{m}\,c_i$. An estimate for m can be obtained from two packet arrivals as follows.

$$a_i = c_{ir} + d_i = mc_i + t_0 + d_i$$

$$a_j = c_{jr} + d_j = mc_j + t_0 + d_j$$

$$a_j - a_i = m(c_j - c_i) + (d_j - d_i)$$

$$m = \frac{(a_j - a_i) - (d_j - d_i)}{c_j - c_i}$$

Since d_j and d_i are unknown, the estimate from m between i and j is:

$\hat{m}_{ij} = \frac{a_j - a_i}{c_j - c_i}$ and the estimate error is:

$$\varepsilon_{\hat{m}_{ij}} = \frac{\hat{m}_{ij} - m}{m} = \frac{d_j - d_i}{(a_j - a_i) - (d_j - d_i)} = \frac{d_j - d_i}{c_{jr} - c_{ir}}$$

This error must be less than the clock skew that we attempt to remove.

3.6.2 Algorithm

In order to reduce the error of the estimate, we select arrival points with small values of delay, so the delay difference is reduced. We also select distant points to increase the denominator and reduce the error even further.

The basic idea of the skew removal algorithm shown in Fig. 18 is to find points of low delay, compute the slope between them, and use this result to refine the function that relates both clocks. The algorithm receives the sender and receiver timestamps, and for each pair it returns the sender timestamp in receiver time units after skew removal. The initial condition is based on the first point of the sequence, which also defines a new reference to measure the sender timestamps. The absolute sender-receiver clock difference is not relevant for our algorithms in Chapter IV; moreover it cannot be determined with only one-way measures. Another mechanism must be employed if one wants to remove any clock offset between sender clock and a reference clock.

```
Initial conditions:
    intervalLimit = 60/mo; // 60 second, mo in seconds
    c0 = c;
    m =mo;
    amin = a;
    cmin = c;
    ck=c;
    ak=0;
    first = true;

On packet arrival of (ci, ai) pair:      // ai in seconds
    if ( (ci-ck) < intervalLimit ) {
      if ( ai-amin < (ci-cmin)*m ) {
            amin = ai;
            cmin = ci;
      }
    }
    else {
      ak = m*(ci-ck) + ak;
      ck = ci;
      if ( first ) {
            amin_avg = amin;
            cmin_avg = cmin;
            first = false;
      }
      else {
            m = (amin-amin_avg + m*(cmin_avg-c0))/(cmin-c0);
            amin_avg = (amin_avg + amin)/2;
            cmin_avg = (cmin_avg + cmin)/2;
            intervalLimit *= 1.5;
      }
      amin = ai;
      cmin = ci;
    }
    return(m*(ci-ck)+ak);    // $\tilde{c}_{ir}$ in seconds
```

Fig. 18. Skew Removal Algorithm.

As the slope computation requires two points, initially the algorithm only determines the packet with smallest delay within 60 second, and it does not compute any refinement for the expected sender clock time unit. For example for audio, a typical expected sender clock is 8KHz, which is equivalent to 125 microseconds. The interval where the delay is minimized grows in 50% after the first estimate is calculated. This ensures increasing estimate accuracy as the time goes by and allows for a quicker skew removal in the beginning of the stream. To improve accuracy, we also increase the denominator by using an average value of previous smallest delay points rather than utilizing the last interval smallest delay only. Finally, we refine the sender clock period estimate by weighting the new interval estimate in proportion to its accuracy. This is derived as follows:

$$weight \equiv \frac{c_{\min k} - c_{\min\_avgk}}{c_{\min k} - c_0}$$

$$m_k = wight \frac{a_{\min k} - a_{\min\_avgk}}{c_{\min k} - c_{\min\_avgk}} + (1 - weight)\, m_{k-1} = \frac{a_{\min k} - a_{\min\_avgk} + m_{k-1}(c_{\min\_avgk} - c_0)}{c_{\min k} - c_0}$$

3.7 Skew Removal Results

We applied our skew removal algorithm to the four traces captured from live session on the Internet. All the traces presented in this work were collected using *rtpdump* version 1.12 [66]. In one of its modes, this tool records the arrival time, timestamp, and sequence number of each RTP packet received on a particular unicast or multicast channel (IP, Port number). While the arrival time is taken from the system time, in RTP protocol [64] timestamps are media dependent and are described in RFCs of The Internet Engineering Task Force (IETF). Knowing the media type for each trace, we converted media-dependent timestamps to time in milliseconds and redefined the initial time by subtracting the first packet sender time to all the other. Similarly, we removed the minimum difference between sender and arrival times from all arrival times; thus we are able to compare our algorithm not only with algorithms immune to clock offset but also to those relying in global clock synchronization. The details of the traces are shown in TABLE 1.

TABLE 1

RTP TRACES

Trace #	Sender	Media	Pack size	ODU Time	Date	Duration	# Hops
1	NASA HQ	Audio	20 ms	08:30pm	09/30/99	600 sec	7
2	NASA HQ	Video	N/A	08:30pm	09/30/99	600 sec	7
3	UC Berkeley	Audio	40 ms	04:05pm	10/06/99	4664 sec	11
4	UC Berkeley	Video	N/A	04:05pm	10/06/99	4664 sec	11

Fig. 19 shows the clock skew between sender and receiver as a percentage of the expected sender clock frequency:

$$skew\% = 100\left(\frac{1/m - 1/m_0}{1/m_0}\right)$$

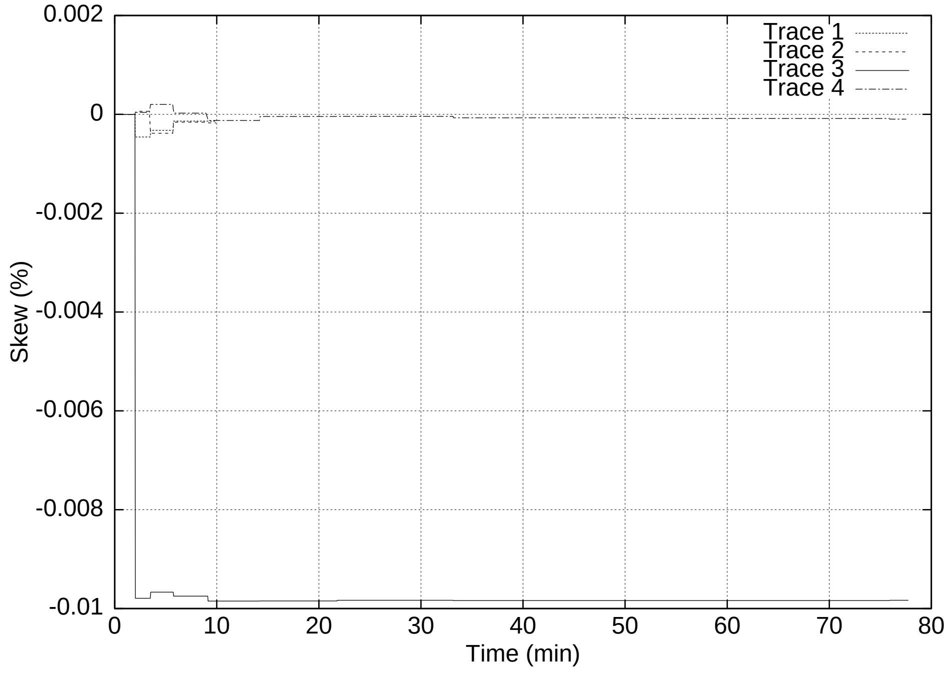


Fig. 19. Clock Skew of Traces 1-4.

The skew is negligible in Traces 1, 2, and 4, but it approximates 0.01% for Trace 3. This explains the accumulated offset of 460 ms in Trace 3, as illustrated in Fig. 20. After applying the skew removal algorithm to the sequence (c_i, a_i) defined by Trace 3, we were able to compute arrival delays plotted in Fig. 21, which shows virtually no skew after 2 minutes.

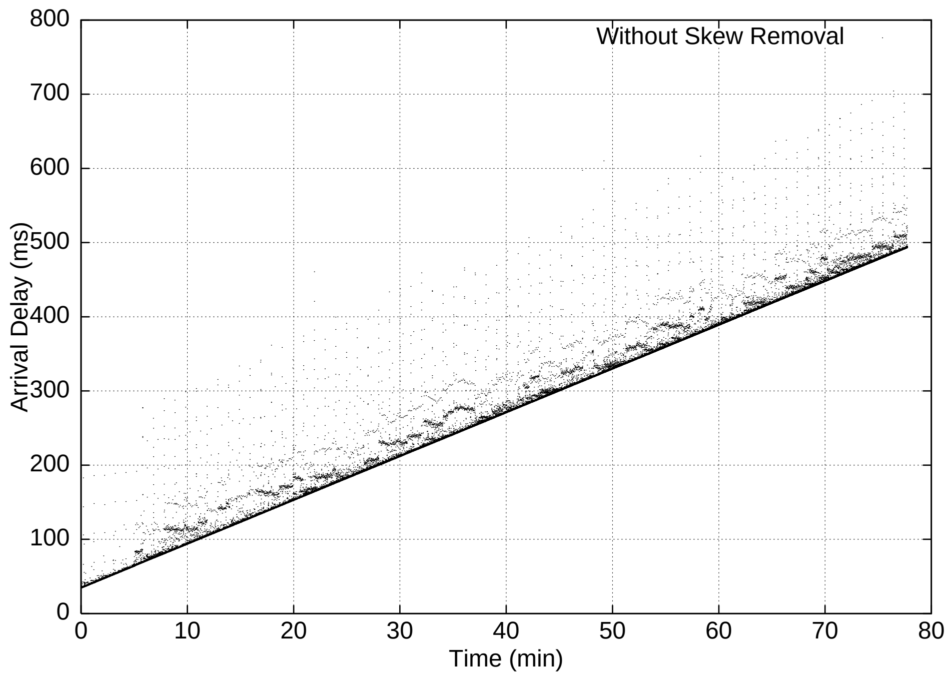


Fig. 20. Effect of Clock Skew in Delay in Trace 3.

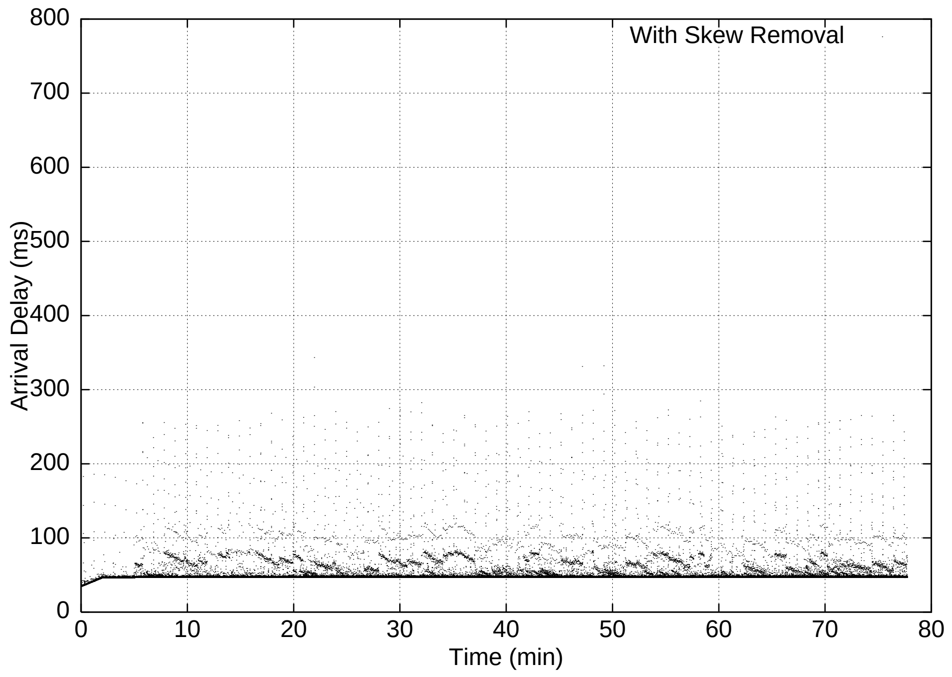


Fig. 21. Arrival delay after removing clock skew in Trace 3.

CHAPTER IV

LIGHWEIGHT STREAM SYNCHRONIZATION FRAMEWORK

In the previous chapter, we introduced the problem of stream synchronization in multimedia applications, presented our conceptual model, and analyzed it in order to achieve synchronization based on the semantic properties of each stream. In this chapter, we use our model to develop specific synchronization algorithms for each media.

Intra-stream synchronization has been addressed in a number of studies in the context of audio applications or video applications. A number of techniques have been proposed to dynamically adjust the total playout delay according to the constantly changing network delay. Stone and Jeffay [70] propose a delay jitter management that we briefly described in Section 3.4 and that defines threshold values for each possible length of the equalization queue. The threshold value for queue length n specified the duration in packet time after which the display latency can be reduced without increasing the frequency of late packets. The main advantage of this approach is its simplicity once the thresholds have been determined; unfortunately in practice they depend on delay statistics that need to be estimated before hand. Other approaches measure the delay statistics on-line and dynamically adapt the delivery delay to reach a good tradeoff between queue delay and late arrival rate. Ramjee *et al*. [57] estimate the delay average, μ, and deviation, σ, values and then set the delivery delay to be $\mu+4\sigma$. This scheme is also simple and automatically adapts to changes in the delay first- and second-order statistics; however, it works only for audio streams since the behavior of video fragments that have the same timestamp is not well captured. Moon *et al.* [48] collect data in a 10,000-packet sliding window, synthesize the delay probability density function, and set the delivery delay to a given percentile. Our scheme for determining the equalized delay basically tries the same goal with fewer resources. As opposed to Moon *et al.*, Xie *et al.* [78] compute probabilities for only three regions in the vicinity, ω, of their estimated delivery delay, Δ. They count the packet arriving at before Δ, between Δ and $\Delta+\omega$, and after $\Delta+\omega$. Packets arriving in the last region are considered late and discarded. Thus, the condition for changing Δ is based on the number of packets falling within each of these regions during a window of around 800 packets. For audio, all these studies propose delivery delay changes only during silence periods.

To the best of our knowledge, inter-stream synchronization has been tackled with synchronized clock only. While Escobar *et al.* [23] and Rothermel and Helbig [62] assume this condition pre-exists in the systems, Agarwal and Son [4] and Ramanathan and Ragan [56] estimate the clock differences by means of probe messages.

4.1 Adaptive Algorithm for Intra-Stream Synchronization

Our synchronization algorithms evolved from a basic and straightforward application of the study presented in Chapter III to more refined versions that take into consideration the time to reach steady state and the peculiarities of each media.

4.1.1 Basic Synchronization Algorithm

The algorithms we present here were obtained after some refinement cycles based on real data collected on the Internet. This data was shown in TABLE 1, and the capture procedure was described in Section 3.7. We present a basic synchronization algorithm that performs well in presence of clock offset and reaches steady state quicker than three already published algorithms. Later we develop variants of this algorithm for each media by taking into consideration specific constraints given by the semantic of each media.

Our first basic synchronization algorithm computes the equalized delay in an amount proportional to the difference between the late packet rate estimated by (15) and an allowed value. Algorithm 1, listed in Fig. 22, defines α and κ as parameters. While α determines how fast the algorithm responds to changes in the rate of late packets, κ controls how fast the equalized delay is adjusted to reach a given fraction of allowed late packets.

```
Initial condition:
    $d_i = a_0 - c_0$;
    $l_i = LatePacketrate$;
On packet arrival to the synchronization module:
    $c_i$ = observer's perception time ;
    $a_i$ = current local time;
    $n_i = a_i - c_i$;
    if ( $n_i > d_i$ )                          /* Late packet */
        $l_i = \alpha\, l_i + 1.0(1-\alpha)$;
    else
        $l_i = \alpha\, l_i$;
    $d_i = d_i + \kappa\,(l_i - LatePacketRate)$;
```

Fig. 22. Algorithm 1: Basic algorithm.

Clock skew and slowness to reach steady state are two important issues that we address in this study. When testing Algorithm 1 with real data, we observed a slight mismatch in clock frequencies in one or both clocks - receiver system clock and media sampling clock - as illustrated in Fig. 23. This drift led to a severe accumulated clock offsets of more than 0.4 seconds after one hour and 15 minutes. If we assume that the receiver's clock has no error, this skew is consistent with a sampling rate of 7,999.2 Hz as supposed to 8KHz.

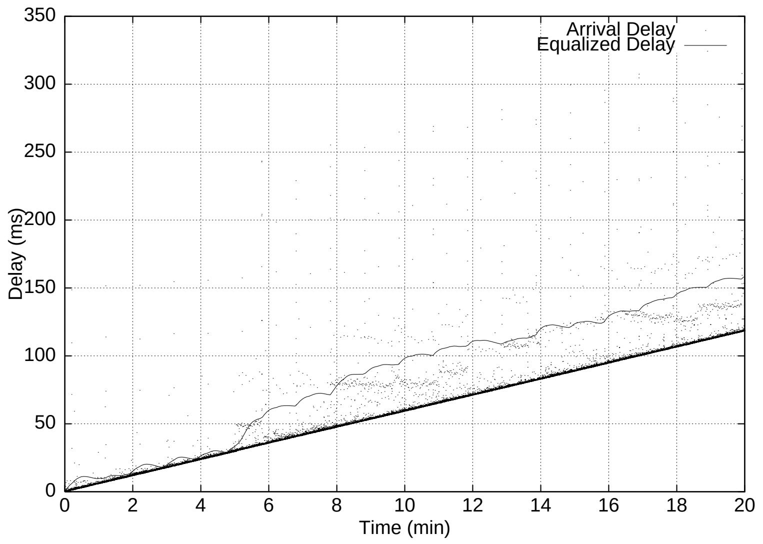


Fig. 23. Equalized delay of Algorithm 1 of Trace 3. The parameters were α=0.996, κ=0.64, and LatePacketRate=0.01.

The clock skew we observe does not have an important impact in intra-stream synchronization because it is insignificant when considered on adjacent packets. Over all packets of Trace 3, for example, 400 ms is equivalent to extra 0.003 ms in the normal 40-ms inter-arrival time of packets. As illustrated in Fig. 24, Algorithm 1 only barely follow the desirable fraction of packet late. Overall Algorithm1 generates an average of late packet of 1.7 % over Trace 3 as opposed to 1%. A clear improvement is gained by considering clock drifting in the model for determining equalized delays. This leads to Algorithm 2, which decomposes the equalized delay in the arrival delay average that follows the drift and an offset that adjusts the equalized delay to achieve a given rate of late packets.

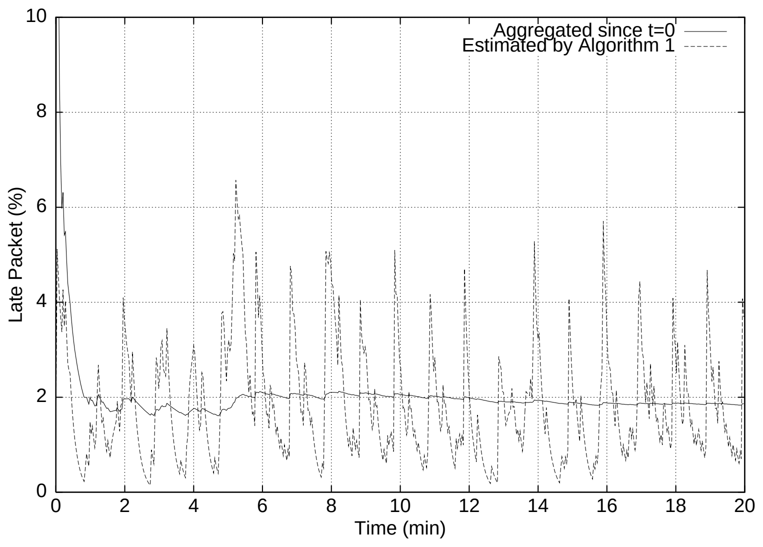


Fig. 24. Resulting late packet rate of Algortihm1 on Trace 3. Parameters as Fig. 23.

Algorithm 2 uses a first order linear filter to estimate the arrival delay average. We took the filter parameter, β, equal to 0.998 which has been used in audio applications (e.g. Network Voice Terminal (NeVoT) [65]) to estimate delays.

```
Initial condition:
    $\mu = a_0 - c_0$;
    $l_i = LatePacketrate$;
    $\varepsilon = 0$;
On packet arrival to the synchronization module:
    $c_i$ = observer's perception time ;
    $a_i$ = current local time;
    $n_i = a_i - c_i$;
    if ( $n_i > d_i$ )                          /* Late packet */
        $l_i = \alpha\, l_i + 1.0(1-\alpha)$;
    else
        $l_i = \alpha\, l_i$;
    $\mu = \beta\, \mu + (1-\beta)\, n_i$;
    $\varepsilon = \varepsilon + \kappa\, (l_i - LatePacketRate)$;
    $d_i = \mu + \varepsilon$;
```

Fig. 25. Algorithm 2.

As shown in Fig. 26, the mean value computed by Algorithm 2 closely follows the clock drift and helps for the late packet rate to be closer to the given value when compared with Algorithm 1. Over the complete Trace 3, Algorithm 2 totals 1.1% of late packets. The variation of the instantaneous late packet rate over time is illustrated in Fig. 27.

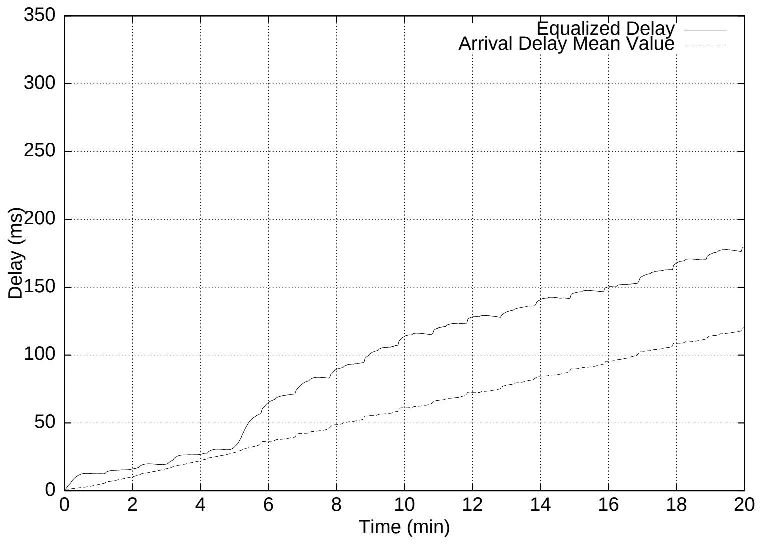


Fig. 26. Equalized delay and arrival delay mean value of Algorithm2 on Trace3. The parameters were α=0.996, κ=0.64, LatePacketRate=0.01, and β=0.998.

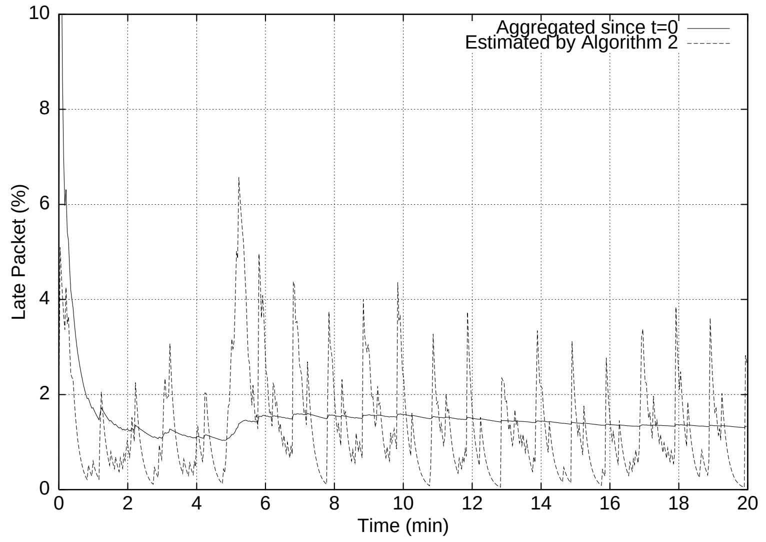


Fig. 27. Resulting late packet rate of Algortihm2 on Trace 3. Parameters as Fig. 26.

Slowness to reach steady state is another important issue for interactive collaborative applications. This problem is clear from Trace 1 where a negligible drift is observed and the delay variations are uniform throughout the trace, as shown in Fig. 28.

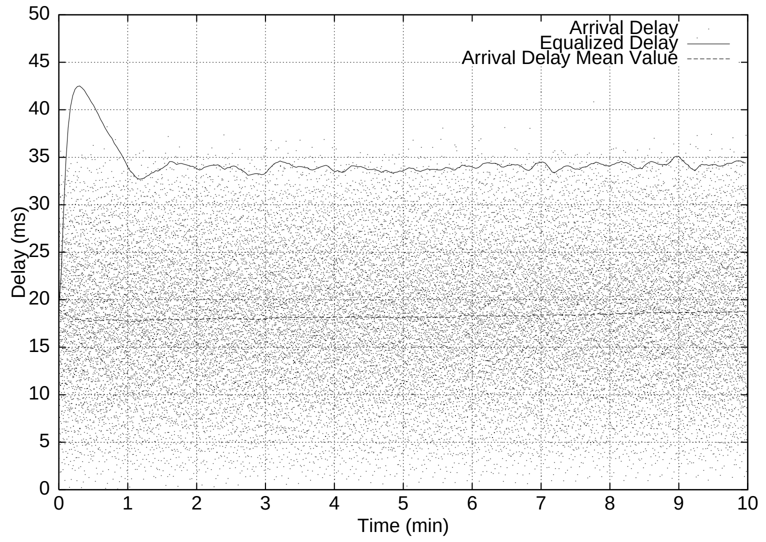


Fig. 28. Initial stage of Algorithm 2 on Trace 1. (α=0.996, κ=0.5, LatePacketRate=0.01, and β=0.998)

The first 30 seconds in each audio stream are of special interest in interactive applications with multiple users because one might intervene for this duration to ask or answer a question and then might leave the audio channel. We think that a stabilization time of more than 10 seconds is not acceptable for interactive sessions, especially for those with potential short time interventions such as distance learning systems. Fig. 29 shows Algorithm 2 with two published algorithms during the first 30 seconds. While Ramjee's algorithm quickly reach its steady behavior, Moon's one takes more than 30 second to reach an stable operation point. This result motivates our refinement of Algorithm 2 to better react during the initial phase. The basic reason for the bad behavior of Algorithm 2 and Moon's during this stage is that they react slowly to changes in arrival delay and tend to maintain a value that equalizes the delay for all packets or at least for packets adjacent or consecutive. This principle defeats a quick response during the first phase. On the other hand, Ramjee's algorithm reacts rapidly initially as desired but does not reach a stable operation point after some time.

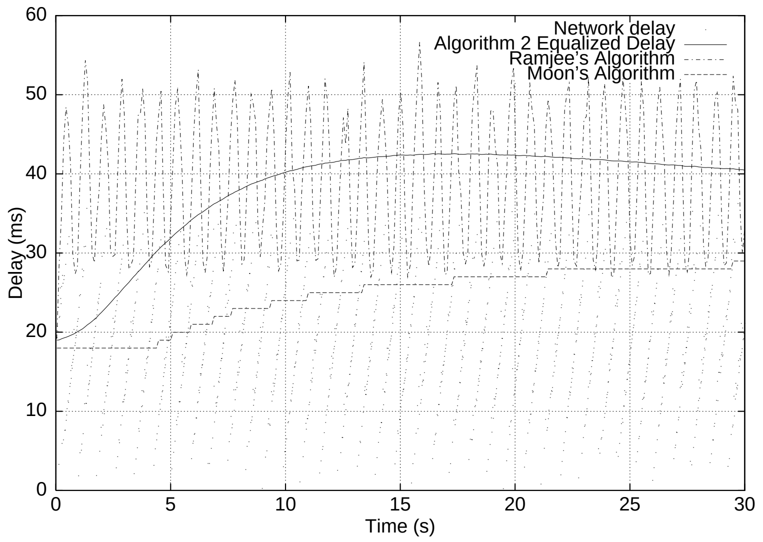


Fig. 29. Stabilization period of three synchronization algorithms on Trace1. The parameters for Algorithm 2 were: α=0.996, κ=0.5, LatePacketRate=0.01, and β=0.998.

Our refinement of Algorithm 2 leads to Algorithm 3 listed in Fig. 30. It is based in the recognition that linear filters that follow recurrence (16) compute an average where the past history has a weight close to one to accomplish a slow response.

$$y_i = \alpha\, y_{i-1} + (1-\alpha)\, x_i \tag{16}$$

Rather than using these weights during the initial stabilization phase, we increase the weight of the history as it effectively conveys more information. Thus, we divide the algorithm in two phases. The first one weights each new data point in proportion to the total number of observed data points. The second phase is reached when the history weight reaches the value we have designed for steady state, either α or β. In other terms, we have broken the recurrence in:

$$\begin{aligned} y_0 &= x_0 \\ y_1 &= \frac{1}{2} y_0 + \frac{1}{2} x_1 \\ y_2 &= \frac{2}{3} y_1 + \frac{1}{3} x_2 \\ &\vdots \\ y_n &= \frac{n}{n+1} y_{n-1} + \frac{1}{n+1} x_n \end{aligned}$$

For a smooth transition from one phase to the other, the algorithm switches to the second phase when *n*/(*n*+1) reaches α. This coefficient can be determined with a recurrence as follows:

$$v_0 = 0$$

$$v_n = \frac{1}{(2 - v_{n-1})} \quad \forall n > 0 \tag{17}$$

Proof: Let v_n be the coefficient *n*/(*n*+1). Then, we prove by induction over n, ∀n>0:

For basic case n=1, we have v_1=1/(2-0) = ½ = n/(n+1)=1/(1+1)= ½.

Inductive hypothesis, we assume that $v_m = \frac{1}{(2 - v_{m-1})} = \frac{m}{m+1}$ holds for n = m.

We now have to prove that (17) also holds for *n*=*m*+1.

$$v_{m+1} = \frac{m+1}{m+2} = \frac{1}{\frac{m+2}{m+1}} = \frac{1}{2 + \frac{m+2}{m+1} - 2} = \frac{1}{2 + \frac{m+2-2m-2}{m+1}} = \frac{1}{2 - \frac{m}{m+1}}$$

And by using the inductive hypothesis, we reach:

$$v_{m+1} = \frac{1}{2 - v_m}$$

```
Initial condition:
    $\mu = a_0 - c_0$;
    $l_i = 0.5$;
    $\sigma = 0$;
    phase = FIRST;
    $v = 0$;
On packet arrival to the synchronization module:
    $c_i$ = observer's perception time ;
    $a_i$ = current local time;
    $n_i = a_i - c_i$;
    if (phase == FIRST)
        $v = 1/(2 - v)$;
        if ( $n_i > d_i$ )                    /* Late packet */
            $l_i = v\ l_i + 1.0(1 - v)$;
        else
            $l_i = v\ l_i$;
        $\mu = v\ \mu + (1 - v)\, n_i$;
        $\sigma = v\ \sigma + (1 - v)|n_i - \mu|$;
        $d_i = \mu + 3\sigma$;
        if ( $v > \alpha \quad \vee \quad v > \beta$ )
            $\varepsilon = d_i - \mu$;
            phase = SECOND;
    else
        if ( $n_i > d_i$ )                    /* Late packet */
            $l_i = \alpha\ l_i + 1.0(1 - \alpha)$;
        else
            $l_i = \alpha\ l_i$;
        $\mu = \beta\ \mu + (1 - \beta)\, n_i$;
        $\varepsilon = \varepsilon + \kappa\ (l_i - LatePacketRate)$;
        $d_i = \mu + \varepsilon$;
```

Fig. 30. Algorithm 3: Fast start refinement.

During first phase we use μ+3σ as equalized delay estimate, and no feedback is employed because in such a short time (around 5 seconds) there is not enough data points to compute the rate of late packets with accuracy. Yet, we estimate l_i during this phase in order to have a good initial condition for the second phase. The values we use here for α and β leads to a first phase of 250-data-point long, equivalent to 10 seconds for Trace 1 (20 ms audio packet) and 20 seconds for Trace 3 (40 ms audio packet). Nonetheless, a reasonable equalized delay value is reached within one second, as shown in Fig. 31, where 50 data points are processed per seconds in Trace 1 (or 25 in Trace 3).

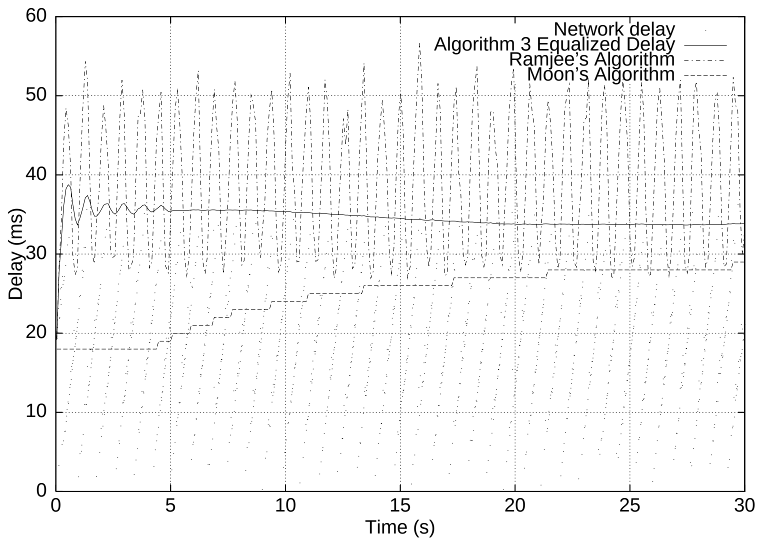


Fig. 31. Stabilization phase of Algorithm3 and two synchronization algorithms on Trace1. Algorithm 3's parameters as in Fig. 29.

So far we have reached step by step an algorithm to collect relevant synchronization statistics and computing an equalized delay. We have left out the computations upon packet delivery. In other words, it has been stated what to do when a new packet arrives and is buffered for later delivery; however nothing has been said on what is to be done when the packet is taken out of the equalization buffer for playout. While the former processing is applicable to any media stream, the latter is media dependent. When looking at media semantics, we identified different valid forms to reduce or increase virtual delay, as we already discussed in Section 3.2, which led us to a number of policies in Section 3.3 to manage adjustments of delay variations. Thus, differences in media semantics suggest that the equalized time computed by our algorithms so far can only be used as a reference, and the actual virtual delay can only be adjusted taking in consideration the semantic of each media.

There are two reasons that make the equalized delay generated by our synchronization algorithm a reference value rather than the base for virtual delay. Firstly, playout constraints might prevent the delivery of data units according to the equalized delay only; and secondly, inter-stream synchronization may result in another media equalized delay being followed rather than its own, as we will see in our discussion on inter-stream synchronization in Section 4.2. On the other hand, by computing the equalized delay, we detach the processing of generic statistics from media playout peculiarities. Thus, the semantic of the media is taken into account in the parameters that feed the algorithm presented in Section 4.1.1 and in the actions performed upon delivery, piece of code we have left out so far.

In the next sections we describe how the basic synchronization algorithm presented here is tailored to fulfill each media semantic requirements.

4.1.2 Audio Intra-Stream Synchronization

There have been many studies on audio playout and audio and video synchronization. Our discussion in Section 4.1.1 was mainly tailored to audio streams due to the fact that it presents more demanding semantic features compared to data or video streams. Yet, we have left out some media specific issues that must be taken into consideration at delivery time. As we mentioned in Section 3.2, virtual delay adjustments cannot take place regardless the audio output device. For instance, increasing data unit delivery rate shifts any synchronization queue delay to the output device queue defeating the original purpose. Thus, the only options to reduce virtual delay are silence period reduction and packet discard. On the other hand, the insertion of artificial gaps is a simple mechanism for increasing virtual delay. Unfortunately, we must also take into account the drift between the system and audio sample playout clocks. As discussed in Section 4.1.1, the same cause that leads to a drift between sampling device clock and synchronization clock, which is based on machine system clock, may also create a drift between synchronization clock and audio output device clock. The effect in the latter case may be either audio device starvation or audio device buffer overflow. For typical sessions no longer than a couple of hours, overflow is unlikely since clocks based on quartz oscillators provide at least 10^{-4} accuracy, which translates to accumulated offset of less than a second in a two-hour session, which is less than 8Kbyte in buffer space at 8KHz sample rate. This problem alleviates in presence of silence periods that naturally flush the device buffer. These considerations might explain why, to the best of our knowledge, this issue is not touched in the literature. Although this is not critical for intra-stream synchronization, neglecting it limits inter-stream synchronization accuracy because of potential unknown audio lag. This is another reason for our framework to define a playout delay, δ_{pi}, in the synchronization module when supporting inter-stream synchronization.

Detection of discontinuities in audio streams due to silence periods is crucial for downward delay adjustments. Our technique is based on inter packet generation time that we assume known by the application. We decided against of computing it on-line as new packets arrive because this is a protocol parameter more than a network uncertainty. Anyway, in case the packetization changes, the algorithm shown in Fig. 32 may be employed to determine inter-packet generation time on-line. Audio packet discontinuity is then detected each time the inter generation time is greater than the given period. Regardless of whether the pause is due to packet loss or a silence period, the gap can be changed slightly without noticeable human perception.

```
Initial condition:
    T = 0;
    $c = c_0$
    timeout = SetValue;
On packet arrival to the synchronization module:
    $\Delta\phi_c = PerceptionTime_i - c$;
    $c = PerceptionTime_i$;
    if ($\Delta\phi_c$ < 3*T/2)
        T = $\Delta\phi_c$;
        timeout = SetValue;
    else if (timeout == 0)
            T = $\Delta\phi_c$;            /* Packet period has changed /
        else
            timeout--;           /* Just a loss or silence period */
```

Fig. 32. Algorithm for inter packet generation time estimation.

Packet discard is the only option in face of no audio pauses. For example, our trace from the NASA involves audio with continuous background music, so the narrator's pauses do not create gaps in the audio stream. In other cases, packet discard is defeated by loss repair schemes that rebuild lost packets [54]. A difference between these two cases is that while lack of silence period is a stream property, packet-loss recovery techniques are under control of the receiver. Thus receivers can disable packet discard when using any repair mechanism or vice-versa.

Fig. 33 is the generic algorithm we propose for packet delivery from equalization queue to the application or player. For convenience, rather than using virtual delay directly, we use the delay from the observer's perception time to the time the packet leaves the synchronization module. We call this delay the delivery delay. As we explained at the end of the previous section, it differs from equalized delay in order to take into consideration the constraints for delay adaptation in each particular media.

```
Initial condition:
   deliveryDelay = equalizedDelay;

On delivering from the synchronization module for playout:
   $c_i$ = equalizationQueue.oldestPacket().observerTimestamp();
   targetDelay = equalizedDelay;
   lag = deliveryDelay – targetDelay;
   if (lag > 0)
        Downward Delay Adjustment Policy(EqualizationQueue, lag, deliveryDelay, $c_i$);
   else
        Upward Delay Adjustment Policy(lag, deliveryDelay);
   rwt = $c_i$ + deliveryDelay – current_local_time();     /*rwt: remaining waiting time */
   if ( rwt < 0 )
        Late Packet Policy (deliveryDelay,EqualizationQueue);              /* late packet /
   else if ( rwt > EPSILON )
              sleep(rwt);                                  /* sleep only if it is worth to do it */
   return(equalizationQueue.dequeueOldestPacket());
```

Fig. 33. Generic algorithm for packet delivery.

The algorithm of Fig. 33 defines the *target delay* as the delay we try to achieve, but due to media constraints we cannot set directly. Depending on how far we are from the *target delay*, defined as *lag*, the algorithm applies a policy for either reducing or increasing the delivery delay. Finally once the delivery delay has been updated, it can be determine whether the packet is late and a late packet policy is applied, or the delivery is delayed. For intra-stream synchronization, the *target delay* is the *equalized delay*;

however, as we discuss in Section 4.2, it could also be given by another stream's delivery delay for inter-stream synchronization.

```
Initial condition:
   deliveryDelay = equalizedDelay;
   gapTimeoutBase=$c_0$;                                   /* for gap detection */
   $c_i$ =$c_0$;

On delivering from the synchronization module for playout:
   $c_{i\_1}$ = $c_i$;                           /* stores previous $c_i$ value /
   $c_i$ = equalizationQueue.oldestPacket().observerTimestamp();
   targetDelay = equalizedDelay;
   lag = deliveryDelay – targetDelay;
   if (lag > 0)                                      /* Downward Delay Adjustment */
      if ( $c_{i\_1}$ + 2*T < $c_i$ )             / * Silence or packet loss */
         deliveryDelay -= min(lag, ($c_i$-$c_{i\_1}$-T)/10);     /* Early Delivery */
         gapTimeoutBase = $c_i$;
      else
         if ($c_i$-gapTimeoutBase > gapTimeout)
            while ( equalizationQueue.length() >1 )   /* Packet Discard */
               if ( deliveryDelay – targetDelay < $c_i$-$c_{i\_1}$ ) break;
               deliveryDelay -= $c_i$-$c_{i\_1}$;
               $c_{i\_1}$ = $c_i$;
               equalizationQueue.dropOldestPacket();
               $c_i$ = equalizationQueue.oldestPacket().observerTimestamp();
            gapTimeoutBase = $c_i$;
   else                                   /* Upward Delay Adjustment */
      if ( $c_{i\_1}$ + 2*T < $c_i$ )             / * Silence or packet loss */
         deliveryDelay += min(-lag, ($c_i$-$c_{i\_1}$-T)/10);    / * gap insertion */
         gapTimeoutBase = $c_i$;
      else
         if ($c_i$-gapTimeoutBase > gapTimeout)          / * Gap insertion after timeout */
            deliveryDelay -= lag;
            gapTimeoutBase = $c_i$;
   rwt = $c_i$ + deliveryDelay – current_local_time();       /*rwt: remaining waiting time */
   if ( rwt < 0 )
      deliveryDelay -= rwt;                      /* Late Packet, resynchronization */
   else if ( rwt > EPSILON )
         sleep(rwt);                               /* sleep only if it is worth to do it */
   return(equalizationQueue.dequeueOldestPacket());
```

Fig. 34. Audio intra stream synchronization algorithm.

In order to account for strictly continuous audio streams, i.e. with no pauses, we propose a hybrid policy that uses Early Delivery in presence of a pause and Oldest Packet Discard after reaching a QoS parameter, *gapTimeout*, with no pauses. It is worth to notice that discarding makes little sense when there is only one remaining packet in the equalization queue because of the risk of having nothing to play when the current playing packet is through. Likewise, when assuming that packets arrive in order, a late packet should not be discard since the previous packet will be already out and it will be the only in the queue. Hereby, we propose resynchronization as late packet policy and rely on downward delay adjustment once the delay peak is over. Thus we reach to our audio synchronization algorithm by completing Algorithm 3 with on delivering segment presented in Fig. 34.

4.1.3 Video Intra-Stream Synchronization

Video packetization characteristics and playout semantic demand special treatment in intra-stream synchronization. Unlike audio, multiple video packets may be required to carry a single frame. As a result, there might be sequences of adjacent video packets with the same timestamp reflecting that all of them belong to the same frame perceived by the virtual observer. In term of the synchronization condition, packets with same timestamp should be played out simultaneously. Nonetheless, they don not normally arrive together, and their arrival times might span hundreds of milliseconds when senders employ some kind of rate control scheme, for example as illustrated in Fig. 35. We observe, though, that these video bursts correlate to changes in scenes such as a camera switch or slide flip, that do not require as strict synchronization as lip synchronization. On the other hand, one or a few video packets per frame are enough to carry audio and video time relationship. Adjusting the synchronization condition with the last packet received of a particular frame will not only lead to a highly variable virtual delay but also an excessive video delay during slight and fine-grained frame changes. To tackle this issue, we define a subsequence of video packets of order k to be the sequence of video packets that contains the first k fragments of any frame. In other words, it is the arriving sequence removing all the packets that carry n^{th} fragment of any frame, $n>k$. The order of the subsequence of video packets is a QoS parameter that controls the synchronization granularity.

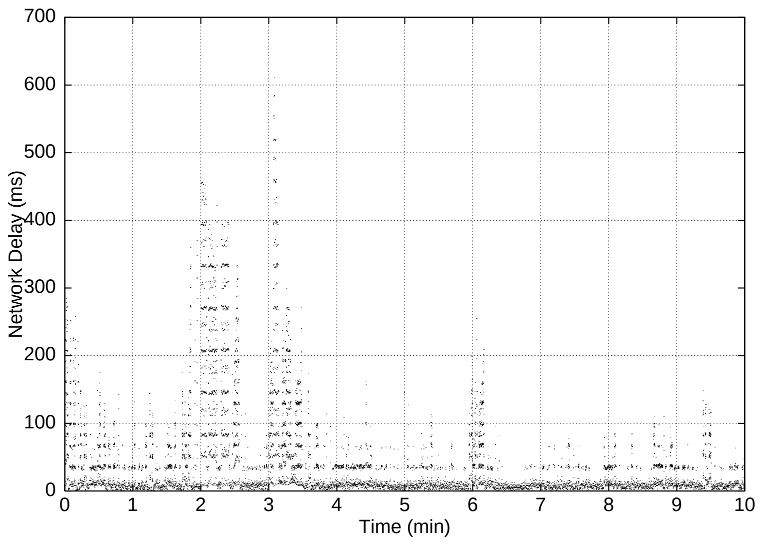


Fig. 35. Network delay in Trace 2. Sender rate control clusters video fragments according to their ordinal position within frames.

Unlike audio samples, video frames inter-delivery time can be varied without permanent increases in playout delay. Unless all the receiving machine resources are utilized at full capacity, frame display rate can be temporarily augmented with no major side effects, and there is no system constraint to reduce this rate. In any case, end users might observe a change in the image pace that could be annoying dependent on how often it happens and human expectations. For example, while watching a moving car, one expects movement continuity; however, our experience indicates that when a slide is flipped, remote users can hardly discriminate half a second in delay. Motion-compensated prediction [16] is a video compression technique that reduces temporal redundancies and leads to smaller compressed frame sizes in face of smooth movements. This technique, in addition to others for spatial redundancy removal, tends to generate less number of packets per frame as our movement expectations rise. It is natural to think that what one could expect can be better compressed. These observations suggest us that good level of synchronization can be achieved by using a subsequence of order k in the video synchronization algorithm and by delivering any higher order packet as late packet. For example, k=2 will ensure that all frames carried in one or two packets will be synchronously delivered, but any frame with 3 or more fragments will deliver the first two fragments in sync and the others late. Fig. 36 shows the additions required in Fig. 30 to compute the video delay parameters based on the first k packet of each frame.

```
Initial condition:
            /* Fisrt part the same as Fig. 30 */
   $c_{i\_1}$ = $c_0$;          /* previous timestamp value */
   k =K_ORDER;            /* frame's packets counter */

On packet arrival:
   $c_i$ = observer's perception time;
   if ( $c_i$ == $c_{i\_1}$ )
        k++;
        if ( k > K_ORDER )
            return;            /* not in subsequence */
   else
        $c_{i\_1}$ = $c_i$;
        k = 1;
   /* it continues as in Fig. 30 */
```

Fig. 36. Video statistics based on subsequence of order-k.

We propose Late Delivery policy for late packet, and Early Delivery and Gap Insertion for downward and upward delay adjustments respectively. These considerations leads to the on delivery section of the video intra-stream synchronization algorithm presented in Fig. 37. We decided against packet discard because its consequences for video decompression algorithms.

```
Initial condition:
              /* no addition to those of Fig. 30 and Fig. 36 */
On delivering from the synchronization module for playout:
   $c_i$ = equalizationQueue.oldestPacket().observerTimestamp();
   targetDelay = equalizedDelay;
   deliveryDelay = targetDelay;
   rwt = $c_i$ + deliveryDelay – current_local_time();        /*rwt: remaining waiting time */
   if ( rwt > EPSILON )
        sleep(rwt);                          /* sleep only if it is worth to do it */
   return(equalizationQueue.dequeueOldestPacket());
```

Fig. 37. On delivering section of video synchronization algorithm.

4.1.4 Non-continuous Media Intra-Stream Synchronization

In this context non-continuous streams are sequence of data units which are time-dependent but occur aperiodically. It includes tele-pointer, shared whiteboard, slide show, and shared tool in general. Their architecture and design make a difference in their temporal dependency. For example, while a sharing tool system such as Virtual Network Computing (VNC) [61] utilizes TCP connections, our sharing tool (Odust) described in Chapter VII uses unreliable IP Multicast as transport layer. Therefore, in VNC all packets must be rendered and the temporal relationship of each one matters; however, in Odust the state of the systems is refreshed every-so-often, so refresh data units do not convey as crucial temporal information as update data units. In addition, the semantics of the stream makes also a difference for synchronization. Removing packets from the equalization queue, for instance, can easily reduce mouse movements delay; nevertheless, all arriving data units should be rendered for free-hand drawing regardless of their tardiness.

Even though there is no clear pattern for synchronization of non-continuous streams, we believe our framework still applies. The statistics can be collected and delay estimated with no or slight modifications of Algorithm 3. Then, our generic algorithm for packet delivery of Fig. 33 can achieve synchronous packet delivery by tailoring it with delay adjustment and late packet policies according to the stream semantic. For example, Fig. 38 shows the delivery algorithm we propose for tele-pointer intra-stream synchronization.

```
Initial condition:
   deliveryDelay = equalizedDelay;
   $c_i = c_0$;

On delivering from the synchronization module for playout:
   $c_{i\_1} = c_i$;                                        /* stores previous $c_i$ value /
   $c_i$ = equalizationQueue.oldestPacket().observerTimestamp();
   targetDelay = equalizedDelay;
   lag = deliveryDelay – targetDelay;
   if ( lag > 0 )
        while ( equalizationQueue.length() >1 )  /* Packet Discard */
              if ( deliveryDelay-targetDelay < $c_i$-$c_{i\_1}$) break;
              deliveryDelay -= $c_i$-$c_{i\_1}$;
              $c_{i\_1} = c_i$;
              equalizationQueue.dropOldestPacket();
              $c_i$ = equalizationQueue.oldestPacket().observerTimestamp();
   else
        deliveryDelay -= lag;                                   /* Upward Delay Adjustment */
   rwt = $c_i$ + deliveryDelay – current_local_time();        /*rwt: remaining waiting time */
   if ( rwt < 0 )
        deliveryDelay -= rwt;                           /* Late Packet, resynchronization */
   else if ( rwt > EPSILON )
              sleep(rwt);                               /* sleep only if it is worth to do it */
   return(equalizationQueue.dequeueOldestPacket());
```

Fig. 38. Tele-pointer packet delivery.

4.2 Inter-Stream Synchronization Algorithm

Inter-stream synchronization restores the temporal relationship among multiple related media. As discussed in Section 3.1, we propose to synchronize only media that form part of a user's multimedia presence. We assume that receiving sites can relate media timestamps and transform them to time values measured on a common clock of that sender. For example, in RTP [64], senders periodically report the relationship between stream timestamps and a wallclock time. Any RTP stream sent by the user used the same wallclock in order to enable inter-media synchronization. Thus, the observer's perception times of each media can be thought as coming from a common clock. When this condition is met, inter-media synchronization is achieved by rendering all streams with a common virtual delay from the wallclock. We define *multimedia virtual delay* to be the common delay used to render all packets regardless of their media origin. Its value is the maximum virtual delay among the streams that compose a multimedia presence. Therefore, unlike intra-media synchronization, inter-media synchronization requires some exchange of information among the intra-stream synchronization modules. We propose a centralized object that computes and maintains the multimedia virtual delay, as shown in Fig. 39.

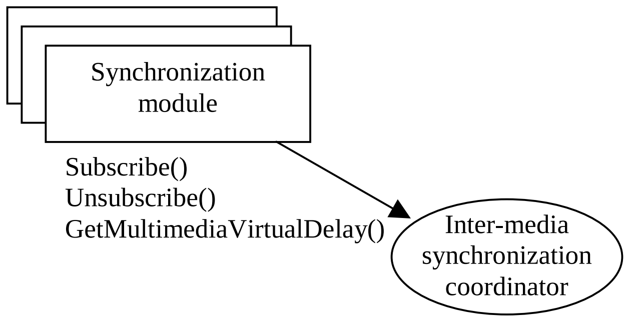

Synchronization module
Subscribe()
Unsubscribe()
GetMultimediaVirtualDelay()
Inter-media synchronization coordinator

Fig. 39. Inter-media synchronization architecture.

Initially, each synchronization module registers itself with the coordinator for it to allocate the resources to manage an additional stream. The maximum among them determines the multimedia virtual delay that is returned by GetMultimediaVirtualDelay(). The stream for which its equalized delay is returned becomes the *master stream*. Every synchronization module posts its equalized delay[2] and follows the returned value as target delay. This approach has the advantage of keeping all the streams following the same target delay. The inter-media synchronization accuracy of this algorithm depends on how well each media can approximate the multimedia virtual delay. The subscription method returns an inter-media synchronization identification (id), which is used later to update the coordinator's state and to remove a stream from the synchronization set upon exiting. Fig. 40 lists in bold the extensions to the Generic Algorithm, shown in Fig. 33, to achieve inter-stream synchronization.

[2] The playout delay, δ_p, needs to be added to reflect the expected playout time.

```
Initial condition:
  deliveryDelay = equalizedDelay;
  inter_syncID =interSyncCoordinator.subscribe();

On delivering from the synchronization module for playout:
  $c_i$ = equalizationQueue.oldestPacket().observerTimestamp();
  bet =  equalizedDelay + $\delta_p$;
  targetDelay = interSyncCoordinator.GetMultimediaVirtualDelay(inter_syncID, bet);
  targetDelay -= $\delta_{p;}$
  lag = deliveryDelay – targetDelay;
  if (lag > 0)
     Downward Delay Adjustment Policy(EqualizationQueue, lag, deliveryDelay, $c_i$);
  else
     Upward Delay Adjustment Policy(lag, deliveryDelay);
  rwt = $c_i$ + deliveryDelay – current_local_time();        /*rwt: remaining waiting time */
  if ( rwt < 0 )
      Late Packet Policy (deliveryDelay,EqualizationQueue);
  else if ( rwt > EPSILON )
         sleep(rwt);                                       /* sleep only if it is worth to do it */
  return(equalizationQueue.dequeueOldestPacket());

On exiting:
  interSyncCoordinator.unsubscribe(inter_syncID);
```

Fig. 40. Inter-media synchronization algorithm.

4.3 Stream Synchronization Results

In this section, we present the results of the intra- and inter-stream synchronization algorithms for audio and video using the traces of TABLE 1. Each trace entry generated by *rtpdump* [66] includes the packet local arrival time as given by gettimeofday() Unix call, the sender's timestamp, and sequence number. We developed a tool to translate the first two to a common time unit as expected by our algorithms. By subtracting a fixed amount to the arrival times, we redefined local zero time to be such that the resulting arrival times are positive values in the order of the inter-arrival variations. Likewise, the unit of the time was change to milliseconds. As the new point for local zero time is arbitrary, absolute delays shown in our graphs do not convey significant information. Sender timestamps were converted by multiplying them by their standard clock frequency and defining sender's zero time to be the first received timestamp. The timestamp clock frequency was chosen as argument of this tool so that we could adjust clock drifting off-line. TABLE 2 and TABLE 3 show the time and timestamp conversion when it is applied over a rtpdump-generate trace.

TABLE 2

AUDIO TRACE CONVERSION FOR SYNCHRONIZATION.

TIMESTAMP CLOCK: 8KHz, TIME OFFSET: 938737649650 (ms).

rtpdump data	Data after conversion
<Arrival time> <Sender's TS> <Sequence #>	<Sender's time> <Arrival time>
938737649.697483 3930816935 10790	0.0 47.48291015625
938737649.717650 3930817095 10791	20.0 67.6500244140625
938737649.737971 3930817255 10792	40.0 87.970947265625
938737649.759406 3930817415 10793	60.0 109.406005859375
938737649.778742 3930817575 10794	80.0 128.741943359375

TABLE 3

VIDEO TRACE CONVERSION FOR SYNCHRONIZATIO.

TIMESTAMP CLOCK: 90KHz, TIME OFFSET: 938737648500 (ms).

rtpdump data	Data after conversion
<Arrival time> <Sender's TS> <Sequence #>	<Sender's time> <Arrival time>
938737648.774484 40097581 47382	0.0 274.4840087890625
938737648.809229 40097581 47383	0.0 309.22900390625
-938737648.833850 40097581 47384 [3]	0.0 333.8499755859375
938737648.840415 40122961 47385	282.0 340.4150390625
938737648.871323 40122961 47386	282.0 371.322998046875

4.3.1 Intra-Stream Synchronization Results

In order to validate the intra-stream synchronization algorithms, we implemented and tried them with data collected from the Internet. For audio stream synchronization we used the Equalized Delay computed with Algorithm 3 (Fig. 30) in conjunction with the audio intra-media synchronization algorithm of Fig. 34. Similarly, for video we used both Algorithm 3 -with the variant shown in Fig. 36 to extract the subsequence of order 2- and the video intra-media synchronization algorithm of Fig. 37. TABLE 4 shows the parameter we used in all the results presented in this section.

[3] A negative arrival time only indicates that the RTP mark bit was set. In this case, this means a frame boundary.

TABLE 4

AUDIO AND VIDEO INTER-MEDIA SYNCHRONIZATION PARAMETERS

Audio		Video	
Parameter	Value	Parameter	Value
α	0.996	α	0.996
β	0.998	β	0.998
κ	0.5 (ms)	κ	0.5 (ms)
LatePacketRate	0.01	LatePacketRate	0.01
gapTimeout	20 (s)	k-order	2

Fig. 41 shows that the Delivery Delay quickly reaches a delay for which most of the packet can be played out synchronously. Then, around 4.5 minutes, longer delayed packets make this value grow and remain high due to the lack of silence periods. The Delivery Delay downward adjustment timeout did not make a difference because the audio delay was less than the inter-packet time (20 ms). In this case the Packet Discard policy would have set the Delivery Delay to a value less than the Equalized Delay.

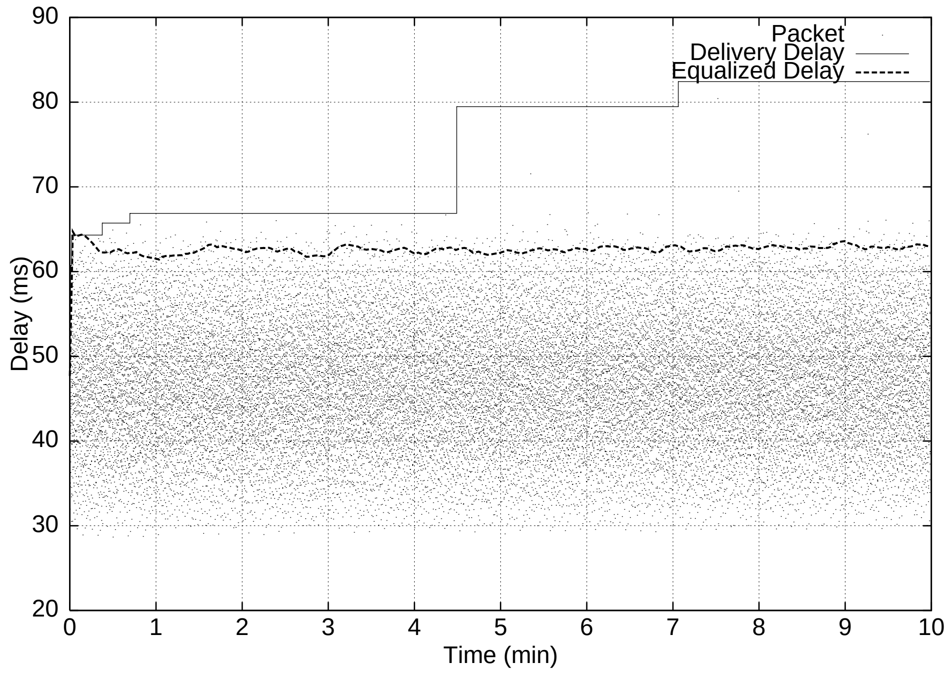


Fig. 41. Audio intra-media synchronization result for Trace 1. Timestamp clock: 8KHz, Arrival time offset: 938737649650 (ms).

Fig. 42 shows another case of intra-audio synchronization. Here the packet delay varies much more; as a result, the Delivery Delay resynchronizes every time a late packet comes and lowers during silence periods. The algorithm naturally adapts to the significant clock drift.

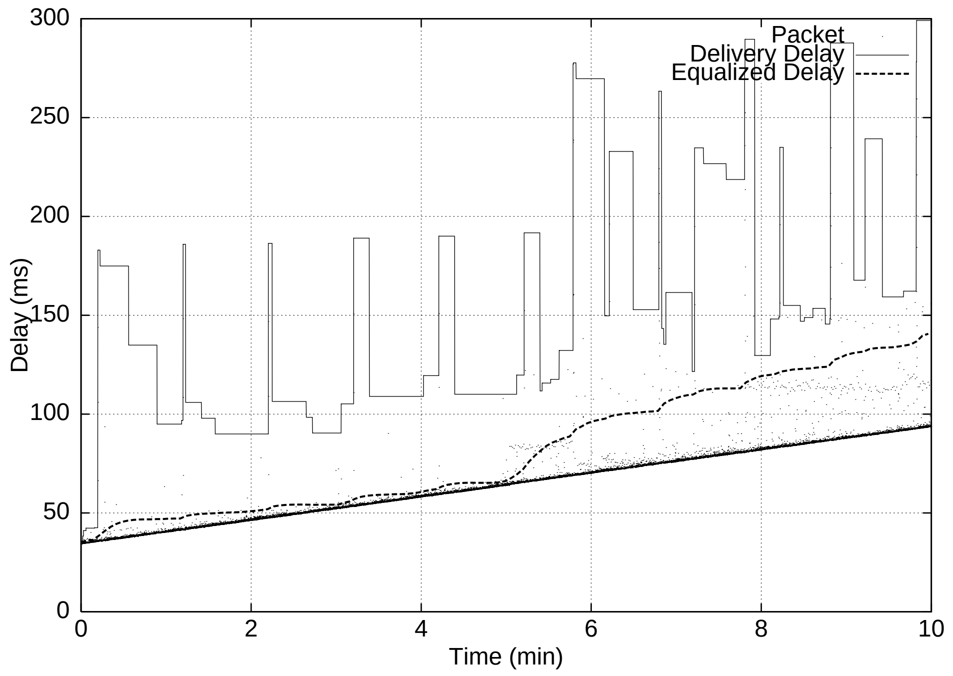


Fig. 42. Audio intra-media synchronization result for Trace 3. Timestamp clock: 8KHz, Arrival time offset: 939240778900 (ms).

Fig. 43 shows normalized frequency for the size of the audio equalization queue right after a packet is delivered. As Trace 1 varies less than Trace 3, the Trace 1 queue keeps less audio packets in average. In Trace 3 the queue holds up to 5 audio packet which means an extra 200 (ms) delay in order to achieve intra-audio synchronization.

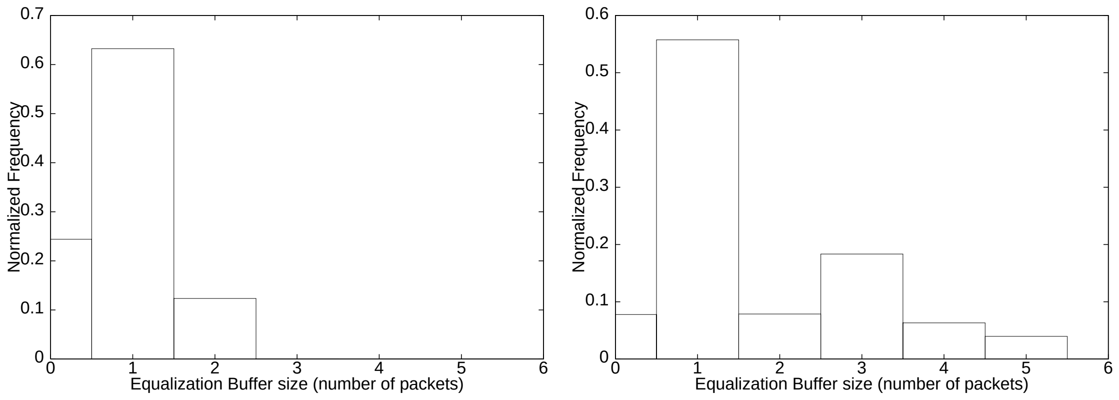


Fig. 43. Equalization queue sizes for Trace 1 (left side) and Trace 3 (right side).

Fig. 44 through Fig. 46 pertain to intra-video synchronization. In contrast to audio intra-media synchronization, video semantic does not constraint delay adaptations, thus the Equalized Delay can be followed very closely as shown by the two overlaying curves.

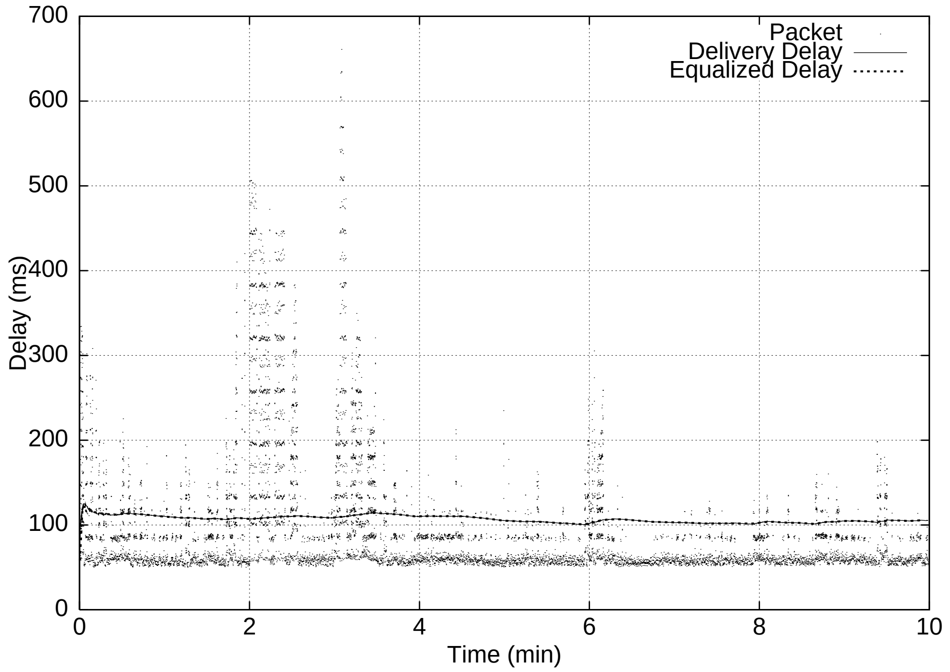


Fig. 44. Video intra-media synchronization for Trace 2. Timestamp clock: 90KHz, Arrival time offset: 938737648500 (ms).

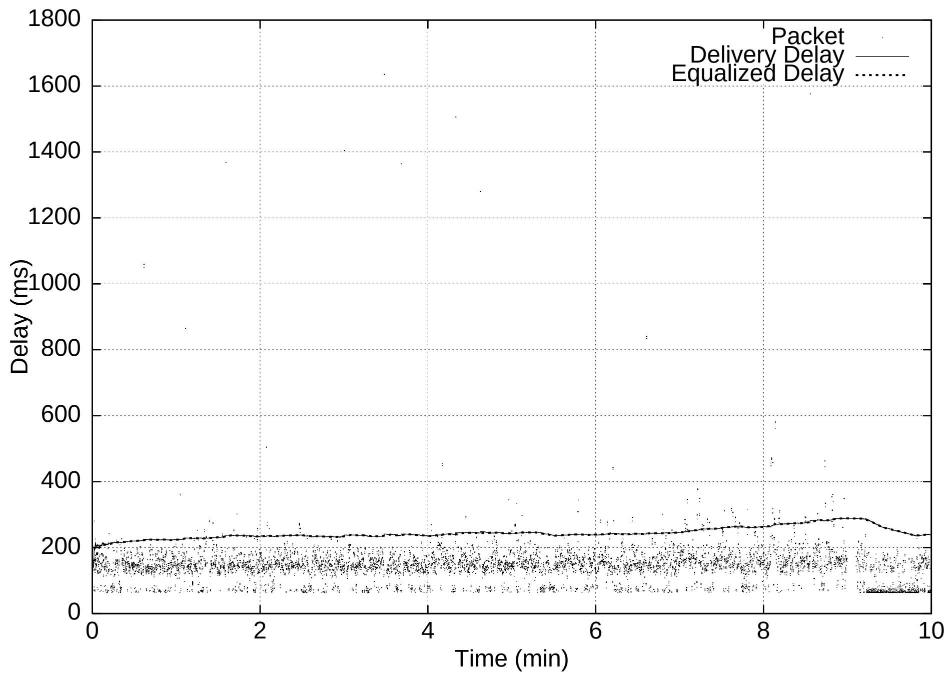


Fig. 45. Video intra-media synchronization for Trace 4. Timestamp clock: 90KHz, Arrival time offset: 939240778900 (ms).

Fig. 46 illustrates the effect of sender rate control on synchronization buffering. In Late Delivery discipline, late packets are delivered immediately so the queue only depend on the packets received in time. Sender rate control in Trace 2 insert an approximate 30-ms pause between fragments; as a result fragments of order higher than 2 are likely to arrive late and, therefore, are not buffered. On the other hand, in Trace 4 more fragments arrive before the Equalized Delay and must wait in the queue; therefore, we observe higher variation in the queue size.

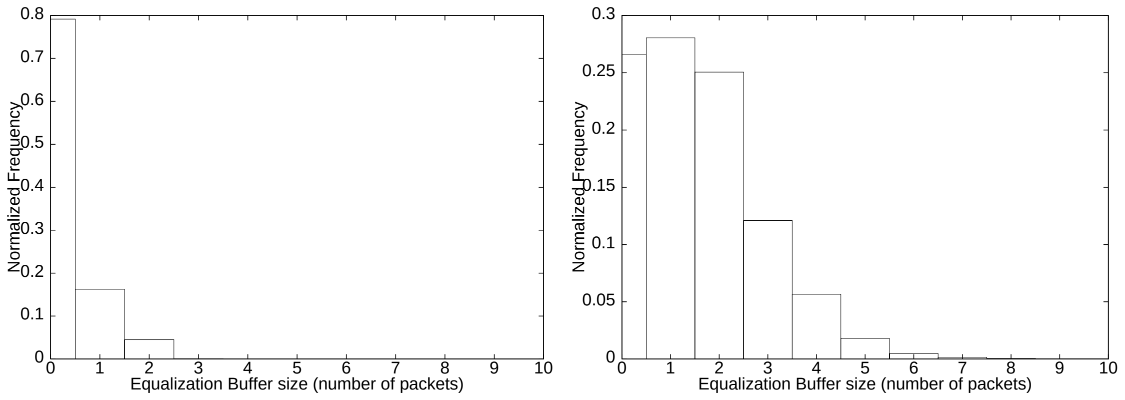


Fig. 46. Equalization queue sizes for Trace 2 (left side) and Trace 4 (right side).

4.3.2 Inter-Stream Synchronization Results

We evaluated inter-media synchronization with two pair of related traces of video and audio. We could not include any non-continuos data stream because even though the Internet Engineering Task Force (IETF) has work in progress in the areas of pointer and text RTP payload formats, we are unaware of any implementations that use RTP protocol as transport layer. We have already mentioned that as opposed to intra-media synchronization, clock drifting must to be corrected for consistent inter-media synchronization. This issue is related to synchronization but we consider it a pre-condition rather than a part of the inter-media synchronization. Thereby, we removed any significant clock drift off-line before we tried our algorithm. We performed drift removal using a slightly different timestamp clock frequency for converting timestamp to milliseconds. TABLE 5 shows the parameters we used to remove clock drifting and time translation and to relate audio and video streams.

TABLE 5

CLOCK DRIFT REMOVAL PARAMETERS

Trace #	Timestamp Clock [KHz]	Arrival Time Offset [ms]
1	8.00000	938737649650
2	90.0000	938737648500
3	7.99921	939240778900
4	90.0000	939240778900

As appreciated in Fig. 47 and Fig. 48, the larger equalized delay plus the playout delay that we assumed the same value for both streams drives the synchronization. In these figures, the video curves overlap, and only one continuos line is shown. Trace1 lacks of silence periods, so its delivery delay adjusts towards the video stream delay only when the *gapTimeout* goes off. After that, audio and video remain within 15-millisecond skew. On the other hand, The numerous periods of silence of Trace 3 allow audio to follow Trace 4 very closely, as illustrated in Fig. 48. In this case the skew does not exceed 10 ms most of the time.

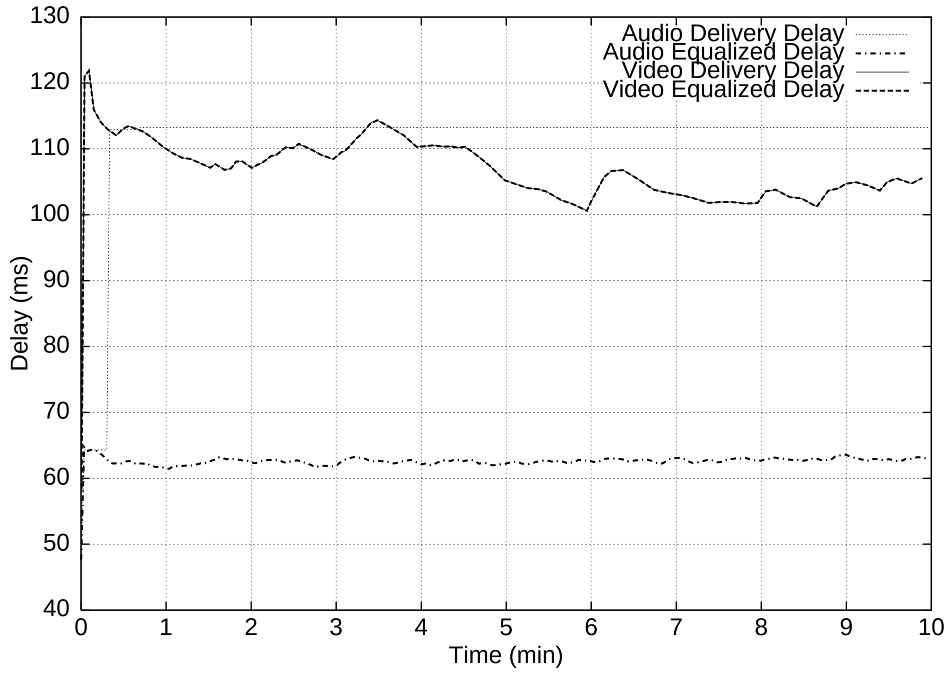


Fig. 47. Audio and video inter-media synchronization result for Trace 1 and 2.

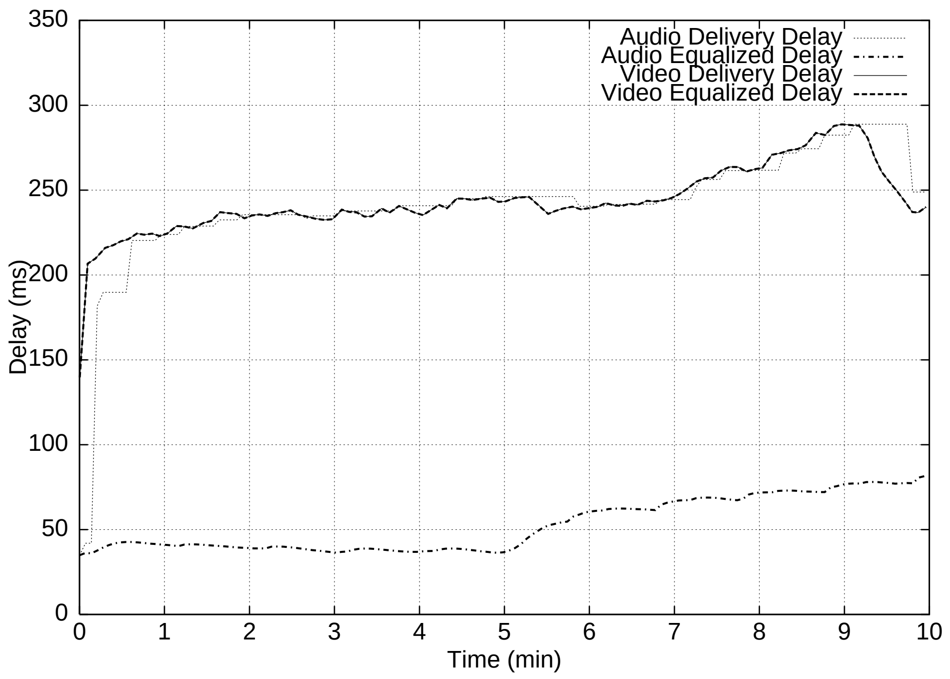


Fig. 48. Audio and video inter-media synchronization result for Trace 3 and 4.

Finally, Fig. 49 and Fig. 50 show the equalization queue size frequency. As expected, streams with smaller equalized delay need to queue more packets to level the multimedia virtual delay. In both cases, video buffer behavior does not change compared to intra-stream synchronization, and audio buffer utilization moves to higher values.

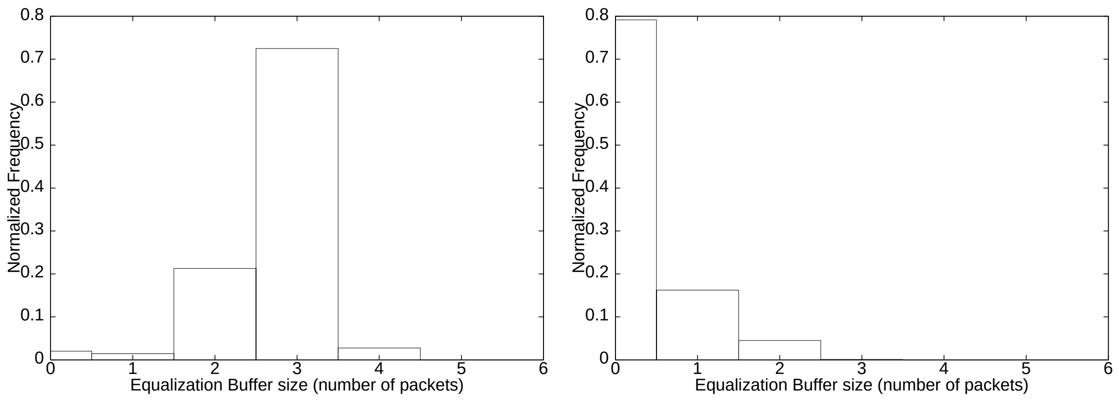


Fig. 49. Equalization queue sizes for Trace 1 (left side) and Trace 2 (right side) during inter-media synchronization.

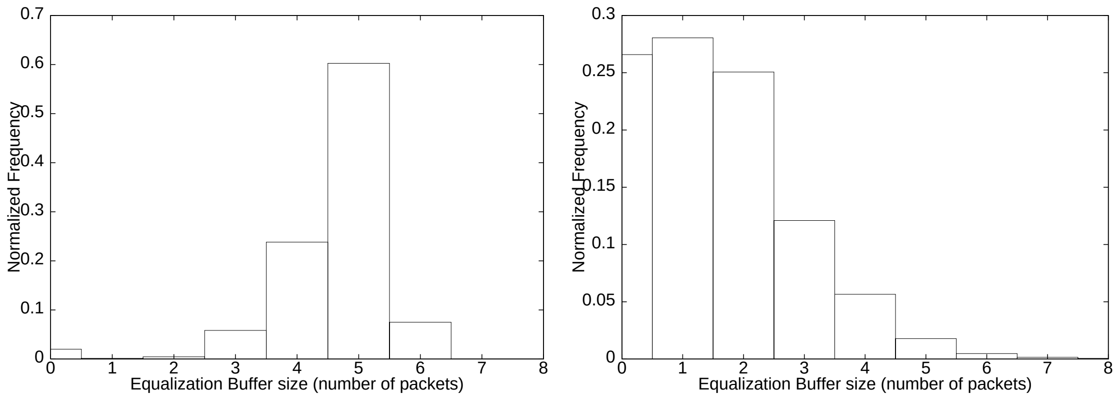


Fig. 50. Equalization queue sizes for Trace 3 (left side) and Trace 4 (right side) during inter-media synchronization.

CHAPTER V

EXTENSION OF OPERATING SYSTEMS NETWORK SERVICES TO SUPPORT INTERACTIVE APPLICATIONS

General-purpose operating systems and high-level programming languages provide abstractions for communicating applications running over a number of geographically distributed sites. Although these services and constructors are general enough for a broad variety of applications, often application developers needs to build an additional layer to reach the services a particular domain requires. Support for asynchronous reception, quality of service (QoS) measures, and transmission rate control are three desirable network services for multimedia applications that are not offered by general purpose networking **A**pplication **P**rogramming **I**nterfaces (APIs). A common pattern in interactive multimedia applications is the arrival of asynchronous messages that are not triggered by any direct local action, but the result of the context of the collaboration among participants. For example, in free audio chat applications users may expect audio traffic from any participant any time, or in a distance learning application a question may be asked at any time. To support this type of pattern, applications use a time-triggered or event-triggered model [74]. While in the former case the application periodically checks the arrival of events, in the latter it blocks and is reactivated by the operating system upon event arrival. Since there are multiple points that can generate events at different rates, an event-triggered model, also called event-driven, is normally employed in interactive applications. It is typically implemented using the UNIX select statement or threads. We propose to encapsulate this behavior in a communication object where applications will receive incoming messages asynchronously, so that developers do not need to implement this common pattern. Another need of multimedia applications is the measurement of a number of quality of service measures such as bandwidth consumption, delay, delay jitter, and packet loss rate. Adaptation layers use these measures to face constantly changing network conditions normally. We believe that bandwidth consumption can be measured more accurately at the lowest layer that applications have control of, which is another service the communication object provides. While local information is enough to compute traffic rate, other measures, such as delay jitter and packet loss, require the participation of the sending site and, therefore, needs to be considered in the application data units. Resource allocation and adaptation is one common module present in many multimedia applications that attempt to offer better quality of service than just best effort. Among other resources, these systems allocate bandwidth to each connection and the application is responsible for enforcing it, so that other streams can get a higher share. We also propose to perform transmission rate control in the communication objects, so multimedia components can gracefully degrade as resources are reallocated.

In addition to the transfer control functions described above, we propose a mechanism by which applications can reduce the number of times the data is copied within the application address space.

5.1 Asynchronous Even-driven Communication

Synchronous and asynchronous communication define two schemes for sending and receiving messages between processes [17]. In the synchronous form of communication *Send* and *Receive* are blocking operations. Whenever a *Send* is issued the sending process – or thread - is blocked until the corresponding *Receive* is issued. This behavior is not suitable for distributed multimedia applications. On the other hand, asynchronous communication allows sending processes to proceed as soon as the message has been copied to a system buffer. Although *Receive* operation can be blocking or non-blocking in this form of communication, blocking *Receive* are usually used because it is easier to use and is supported in most common operating systems. Whenever a *Receive* is issued the process blocks until a message arrives, a timeout can often be specified. Due to the uncertainty in message arrivals, multimedia applications must periodically check for message arrivals or devote a process or thread to attend these events. Another option is polling; however, it might limit the progress of the application when many events are generated and only one can be served at a time. Processes and threads solve this shortcoming; however, threads are more convenient for their ability to access the shared data space of the other components of the application. This allows for more flexible and efficient interactions between the modules of multimedia applications. In order to simplify asynchronous communication, we also propose to encapsulate the blocking

Receive operation in a thread implemented in the communication object. Thus, this approach integrates asynchronous communication into an event-driven model by letting applications register objects that are invoked upon message arrival. This basic idea is already supported in languages such as Visual C++ and Motif; nonetheless, Java and C lack it.

5.1.1 Event-driven Multicast Socket Definition in Java JDK 1.2

In this section we give an example of how Java can be used to offer an event-driven processing of the messages arriving on a JDK 1.2 multicast socket, as illustrated in Fig. 51. An analogous approach can be employed to accept connections on a server socket or incoming messages in a connection oriented socket.

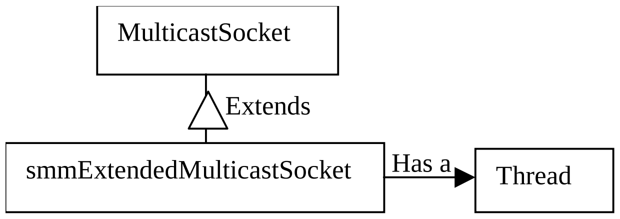


Fig. 51. Java multicast socket extension for supporting even-driven model.

The relevant constructor, data, and function members are shown in Fig. 52. A complete implementation of this class can be found in APPENDIX B.

Basically, the smmExtendedMulticastSocket class supports both synchronous and asynchronous reception of datagrams. The mode is controlled by a boolean data member. In synchronous mode the socket has a Java normal behavior. In contrast, asynchronous mode runs a thread that blocks and waits for datagram arrivals. Indeed, the thread calls the method smmOnReceive() of the smmOnRecvListener object previously registered by the application, and it is responsible for invoking the socket *receive* method. We decided against of calling the *receive* method within the *run* method in order to uncouple the socket from the message buffer. As a result, by implementing the smmOnReceive function, developers can define the behavior of the application under datagram arrivals in a similar manner they associate actions to a button in the GUI.

```
public class smmExtendedMulticastSocket extends MulticastSocket
        implements Runnable {

    private smmOnReceiveListener onRecvListener;
    private boolean asynchronousMode;
    public Thread arrivalThread;

    public smmExtendedMulticastSocket (int port, InetAddress addr, int ttl)
        throws IOException{}
    public void setOnReceiveListener(smmOnReceiveListener l) {}
    public void setSynchronousMode() {}
    public void setAsynchronousMode() {}
    public void run () {}
}

public interface smmOnReceiveListener {
  void smmOnReceive(smmExtendedMulticastSocket sock);
}
```

Fig. 52. Basic Java class description for supporting event-driven model.

5.1.2 Towards a Unified Multicast/Unicast API

Developers are often faced with the problem of designing software to manage two-party point-to-point communications through unicast and multi-party communications through multicasting. The **A**pplication **P**rogramming **I**nterfaces (API) provided by common languages such as C, C++, and Java, are different and reflex the semantic of each type of communication. Despite the differences in the underlying network delivery protocol, there is a common mechanism at the API level for sending messages to a group or a single recipient. The IP address differs between a group and a destination machine in the same way it varies between unicasts to two machines. The main discrepancy is observed between receiving unicast and multicast messages. While the receiving network interface is understood[4] for receivers of unicast messages, recipients of multicast messages need to declare the group they want to subscribe to. The APIs for multicasting provide mechanisms for dissociating a communication end point (socket) from a particular group; however, the APIs for unicast do not allow in general for switching

[4] An exception is a machine with multiple network interfaces (so-called multihomed host). Here the local interface might be provided too.

from an interface to another without closing and reopening a new connection point[5]. In any case, the network overhead of changing a multicast address is similar to that of closing and reopening a new multicast connection. We propose to unify unicast and multicast APIs by binding the socket to an interface or joining a multicast group depending on the IP address. This operation can be hidden from the programmer to achieve a uniform API. A missing interface or multicast address is understood as a binding to the local default interface. In Java, multicast socket is a subclass of datagram socket. This makes datagram methods accessible to multicast sockets too; therefore, after the socket has been created and bound or joined, both unicast and multicast sockets have access to datagram methods. In the need of finer control, multicast socket still can call multicast specific methods such as “setTimeToLeave()” that would cause no harm in the event the socket were actually unicast.

The API presented in Fig. 52 can also be used for point-to-point communications by providing a null address. This makes the socket bound to the default network interface. On the other hand, if the provided address is a multicast address, the implementation of the constructor joins the multicast group. After the socket has been created, all the methods work for unicast or multicast.

5.2 Traffic Measures and Rate Control

We believe that in addition to sending and receiving application data units, the network access point of multimedia applications is the natural place for both collecting statistics regarding the bandwidth consumed by the each network connection and controlling the outgoing traffic rate. Right before a *Send* operation and after a *Receive* operation, applications have access to the total size and departure or arrival time of the data units. This makes the communication object a good candidate not only to compute rate statistic but also to control the transmission rate. Other QoS measures, such as end-to-end delay, jitter, and packet losses, need sender information that is normally encapsulated in the protocol packet. As suggested by the principle of Integrated Layer

[5] While Unix “bind” supports binding to a new address, Java JDK 1.2 does not support such a feature.

Processing [74], these computations should be done along with traffic rate measures; however, they involve protocol packet structures that we do not want to enforce in our framework. It should be up to designers. For example, Real Time Protocol (RTP) [64] a standard of the IETF, includes a sequence number and a timestamp field for this purpose. Therefore, we only measure traffic rate and leave the developers the expansion necessary to measure other parameters once the datagram structure is defined.

Rate control is another service to be offered by the communication object. Goals of traffic control are to prevent congestion and to reduce buffer requirements. In multimedia applications, available bandwidth needs to be carefully allocated to each stream in order to offer the best overall quality to the end users [79]. A number of traffic models have been proposed [7] [18] [25] [79]. They are based on parameters such as maximum message size, maximum message rate, workahead limit, traffic average, peak-to-average traffic ratio, and minimum inter-packet interval. The traffic pattern varies so much from one media to another that it is difficult to find a model that can well represent all of them. A different approach is to design multimedia applications such that their traffic patterns better fit the network services. Buffering data ahead of time at the receiving site is one example and has been highly used in streaming audio in the Internet. To support this approach, we use a simple technique that controls the transmission by limiting the short-time traffic rate ($STTR_k$). It is computed over a time window that spans the last k packets, as depicted in Fig. 53.

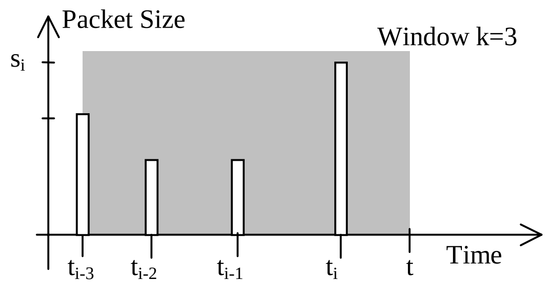


Fig. 53. Short-time traffic rate estimate for k=3.

Let t_i be the arrival or departure time of packet *i* and s_i be its size, then $STTR_k(t)$ is defined to be:

$$STTR_k(t) = \frac{\sum_{j=i-k+1}^{i} s_j}{t - t_{i-k}}$$, where i is the latest packet such that $t_i \le t$ and k>0.

To limit the outgoing traffic, packets are delayed if necessary until the short-time traffic rate is lower than a dynamically configurable threshold. By controlling the outgoing traffic, we are also indirectly controlling the progress and the resources of the applications. Reducing a component traffic gracefully degrades its responsiveness and lets other relevant components keep their performance. For instance, let us consider an audio conferencing application. Assume the presenter's instance of the application detects audio is being transmitted at a lower rate than that dictated by its encoding. The application has at least two alternatives in order to improve audio delivery. It could either change the encoding to fit the bandwidth share it is getting or lessen video transmission rate, which will then gracefully reduce the average frame rate. In the former approach, bandwidth can be taken later from video to change the audio encoding back to normal, and video bandwidth can be gradually increased until the point where audio starts failing in delivering all the traffic in a timely fashion. Although we do not propose an adaptation framework in this work, the services offered by the communication object are the bases over which adaptation objects are to be built.

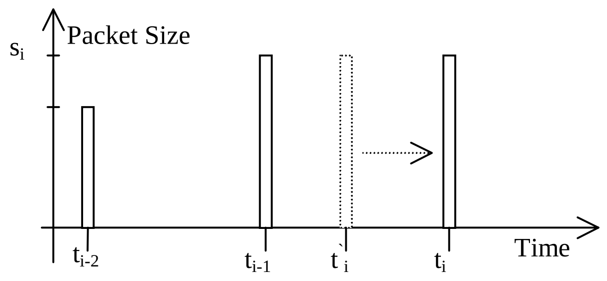


Fig. 54. Traffic rate enforcement.

Fig. 54 illustrates the enforcement of a maximum traffic rate. The delivery of the i^{th} packet is delayed by blocking the thread from $t^{`}_{i}$ to t_i, so that it meets the traffic rate limit.

The value for parameter k has to be decided based on receiver buffering capacity, typical packet size, rate of thread context switches, and time resolution. Large k values generate periodic bursty traffic when the application attempts to produce higher instantaneous rate than the limit. For example, while sending video, senders normally divide each video frame into several packets of compressed data. These packets have all the same timestamp and are transmitted in sequence. A large k value allows for sending many packets before the rate limit is reached because of the large idle time between frames. This packet burst might overflow receivers socket buffer before either an output channel or the end application can consume enough packets. A small k value, on the other hand, reduces the burstness by introducing frequent transmission pauses, and therefore increasing context switches between threads. Finally, poor time resolution might lead to a null denominator in the short-time traffic rate computation. For instance, Java typically measures time with millisecond resolution, which creates problems while sending a burst of small packets. As design criterion, k can be selected in the order of the quotient between the receiver socket buffer size and the typical (or maximum) packet size.

For traffic rate monitoring, the window size used in traffic control might not be convenient. We suggest bigger windows that only depend on the monitor sampling frequency.

Finally, we could have selected a window size based on a fixed span rather a number of packets. We decided against it because of implementation reasons. Fixing the number of packets to be considered in the rate computation sets a limit on the number of entries that a data structure or object needs to maintain. Otherwise, dynamic data types are required in general, which are not as efficient as static data types.

5.2.1 Rate Controlled Multicast Socket Implementation in Java JDK 1.2

Here we present the extension of the multicast socket class of Section 5.1.1 to support monitoring and rate-controlled transmission. We have included methods for traffic rate monitoring and output traffic rate control. For transmission there are two windows: one for computing the short-time rate to be used in rate control and another for monitoring as described in Section 5.1.2. For incoming traffic, on the other hand, only a monitoring window is employed. Fig. 55 lists the data and function members added to the class presented in Fig. 52 to support monitoring and transmission rate control.

```
public class smmExtendedMulticastSocket extends MulticastSocket
     implements Runnable {
    // In addition to the data and function members listed in Fig. 52.
    // Data members for collecting statistics
    protected long startingMeterTime;
    private boolean txRateControlOn;
    private int txRateLimit;            // outgoing traffic rate limit
    protected int totalTxBytes;         // Total bytes sent since meter is on
    protected int txReqTime;            // Last time a send request took place
    protected int totalRxBytes;         // Total bytes received since meter is on
    private boolean meterOn;            // Control whether statistic is collected
    protected int[] txTime;             // Circular buffer for storing Tx times
    protected int[] txSize;             // Circular buffer for storing Tx packet sizes
    protected int txTraffic,  // total tx traffic in rate in controlling window.
                  rxTraffic;   // Total rx traffic in the monitoring window (history)
    protected int[] rxTime;             // Circular buffer for storing Rx times
    protected int[] rxSize;             // Circular buffer to storing  Rx packet sizes.
    protected int txindex, rxindex;     // Indexes to travel Rx and Tx circular buffers
    protected int history;  // Number of packet for short-time computations
    protected int winSize;  // Number of packet for rate control processing

    public smmExtendedMulticastSocket (int port, InetAddress addr,
                                               int ttl, int history)
      throws IOException { }
    public void startMeter () {}
    public void stopMeter() {}
    public boolean  isMeterOn() {}
    public void enableTxRateControl(boolean state) {}
    public boolean isTxRateControlEnable() {}
    public void setTxRateLimit(int rate) {}
    public int getTxRateLimit() {}
    public int setTxRateWindowSize(int windowSize) {}
    public int getTxRateWindowSize() {}
    public void receive (DatagramPacket p )
      throws IOException {}
    public void send(DatagramPacket p, byte ttl)
       throws IOException {}
    public void send(DatagramPacket p)
       throws IOException {}
    // Statistics
    public int avgRxTrafficRate() {} // in byte/s
    public int avgTxTrafficRate() {} // in byte/s
    public int rxSTTR() {}  // Rx short-times Traffic Rate in byte/s
    public int txSTTR() {}  // Tx short-time Traffic Rate in byte/s
}
```

Fig. 55. Multicast socket definition supporting monitoring and rate control.

5.3 Technique for Preventing Multiple Data Unit Moves

The development of multimedia application normally involves multiple functional modules. Object oriented design techniques suggest the identification of well-defined abstractions that can be encapsulated and be used as building blocks. While reusability and extensibility are two important advantages of this approach, low performance could be one of its drawbacks. In other words, although logical decomposition of a problem might suggest multiple functional units, performance considerations might recommend developers to perform a number of manipulation steps in one or two integrated processing loops, instead of performing them serially in separated objects. Therefore, we looked for techniques for reducing part of the overhead introduced by multiple related functional modules. Moving data from one part of the memory to another is one type of overhead we try to reduce by the technique we propose in this section.

When analyzing protocol functions, D. Clark and D. Tennenhouse [14] identified six *Data Manipulation* steps that involve reading and writing the data, and in some cases moving it from one part of the memory to another. Some of them are unavoidable and are outside of the scope of applications such as moving data to and from the network and moving data between application address space and system address space. However, other data manipulations within the application scope, such as RTP packetization, do not need to add additional overhead due to data movements. A difficulty developers have to face is the addition of headers as Application Data Units moves from upper layers to lower layers. This step is normally accomplished by allocating a bigger buffer and copying the higher layer payload after the header[6]. Additions at the end of a packet also require bigger buffer allocations. Even though languages like Java provide classes that automatically increase the size of the buffer as more data is written into it, this process still involves some overhead.

We believe that a natural consequence of Application Level Framing, a design principle proposed in [14] that is very common in datagram-communication based applications, is that the final packet size awareness must exist in every module producing

[6] In Unix the “writev” call gathers the output data from a number of possible scattered buffers. This may be used in the lowest application layer.

payloads. The main reason behind it is the avoidance of fragmentation by ensuring that each ADU is conveyed in a single datagram. Otherwise, the loss of one fragment induces the loss of the entire ADU, and the bandwidth of transferring all the other fragments is wasted. Thereby every payload producer should allocate an ADU buffer big enough to hold any posterior additions at either ends of the payload. In addition, the position of the initial payload must take into consideration any subsequent headers[7]. Analogously, receiving modules must allocate buffers big enough to hold the worst-case packet size.

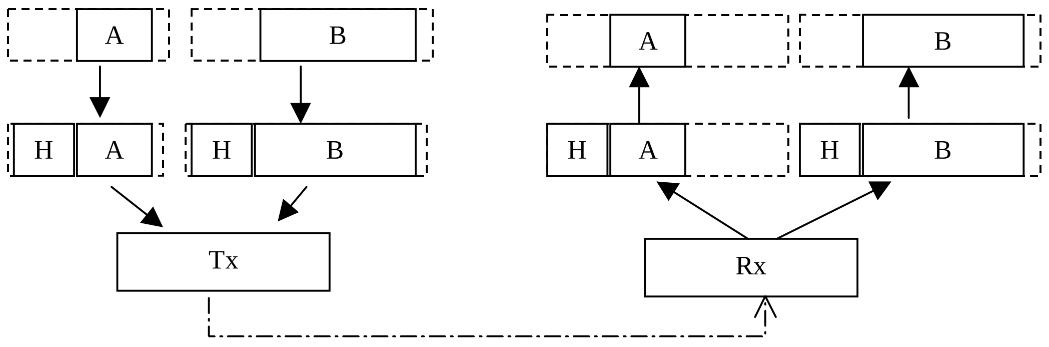


Fig. 56. Buffer allocation for preventing payload moves.

As illustrated in Fig. 56, each payload producer allocates memory to hold the final transmission packet. Rather than moving payload to bigger buffers, lower level modules write their headers into payload's buffers. At receiving sites, in contrast, packet memory allocation is done at the lowest application layer. Each layer reads its corresponding load and passes the entire packet to the upper module. The state and behavior of each buffer object provide isolation between modules by keeping track of the data boundaries.

In summary, multiple data moves can be prevented by buffer allocation at payload producer modules and lowest layer receiver modules and considering the worst-case packet size in each case. The buffer object is aware of data boundaries, so that write operation can be performed at either end of the data. Arriving packets are passed to upper layers, which read the data in reverse order at which is was written, so each level

[7] This condition can be removed by allocating a bigger buffer to cope with extreme cases.

can extract it in isolation of the others. This approach assumes payload producer modules know final packet size and receiver module knows worst-case packet size.

5.3.1 Packet Buffer Implementation in Java JDK 1.2

We implemented Java classes for output datagram packet and input datagram packet (smmInputDatagrampacket). Java provides an elegant Input/Output model that allows developers to easily connect input or output stream to multiple sources and destinations, such as files, sockets, memory, and pipes. Likewise, it provides classes for data input/output and character input/output. We created an smmOutputDatagramPacket class (smmODP in short) to be an extension of the Java OutputStream abstract class and defined an array of bytes as the buffer for our output packets. Being a subclass of OutputStream, smmODP could be used to create a DataOutputStream object that supports all the needs for data input/output. On the other hand, for input packets, we created an smmInputDatagramPacket class (smmIDP in short) to be an extension of the Java ByteArrayInputStream that in turn is a subclass of InputStream class and set an array of bytes to buffer incoming packets. Like in smmODP class, we provided a DataInputStream object to support data input/output.

Our implementation for the input and output packets is not symmetric due to the asymmetry of the Java ByteArrayOutputStream and ByteArrayInputStream classes. While the latter allows setting of the buffer from which data is read, the former provides its own buffer that expands as more data is written into it. This class also encapsulates the writing position, which prevented us from additions in the head of the buffer.

We set arbitrarily the output packet initial position to one fourth of the size of the packet. A more conservative approach is to set it to the middle of the buffer and allocate twice as much memory. Meanwhile, we think these classes need to be used in more scenarios to decide for a more convenient approach; for instance, developers may better decide the initial writing point in the buffer.

```
public class smmOutputDatagramPacket extends OutputStream {
  private DatagramPacket packet;
  private byte [] buf;
  protected int head;
  protected int tail;
  protected int pos;
  public DataOutputStream dataOutStream;
  public smmOutputDatagramPacket (int size) {}
  public smmOutputDatagramPacket (int size, InetAddress iaddr) {}
  public void reset() {}                    // clear packet and set it to initial state
  public void setAddress(InetAddress iaddr) {} // set destination address
  public void write(byte[] b) {}            // override OutputStream class method
  public void write(byte[] b, int off, int len) {} // override OutputStream class method
  public void write(int b) {}               // required by OutputStream abstract class
  public int getPacketPos() {}                        // position where next write will occur
  public void extendHead(int extensionSize) {} // extend head for new header and seek
                                                      // writing position to the head.
  public void seekHead() {}                 // move writing position to packet's head
  public void seekTail() {}                 // move writing positioon to packet's tail
  public int getSize() {}                   // return size of packet so far.
  public DatagramPacket getDatagramPacket () {}  // return datagram holding packet
}
```

Fig. 57. Output Datagram Packet class definition in Java.

Fig. 57 shows the definition for a class that provides the type of abstraction we propose for an output datagram packet, so that each byte is copied only once as the datagram packet is formed. For example, this copy can be done along with data presentation formatting while compressing a multimedia stream.

Fig. 58 shows a class that satisfies the requirements of our input datagram packet. It is much simpler than its output counterpart because in our implementation recipient modules only read forward and in order; however, more scenarios might suggest new behaviors that can easily be added by either defining a subclass or expanding the methods listed here. Complete implementation of the two classes discussed here can be found in APPENDIX C.

```
public class smmInputDatagramPacket extends ByteArrayInputStream {
  private DatagramPacket packet;
  public DataInputStream dataInStream;
  public smmInputDatagramPacket (int size) {}
  public void rewind() {}  // set reading position to first byte of the buffer
  public DatagramPacket getDatagramPacket () {}
}
```

Fig. 58. Input Datagram Packet class definition in Java.

5.4 Related Work

The problem of transferring application information among machines has been addressed in several works from various angles: network protocols, operating systems, programming languages, and frameworks. In 1990, D. Clark and D. Tennenhouse [14] foresaw the need for a new generation of protocols to cope with network of greater capacity, wider heterogeneity, and broader range of services. They suggested the principle of Integrated Layer Processing to group manipulation steps and improve performance and identified six steps where data is moved within a machine in order to transfer application information to other machines. We believe that our technique to handle datagram buffering complements this principle.

The structure of operating systems has an impact on the performance and the scope of applications that can be built on physical hardware. Monolithic and micro-kernel operating systems are two familiar structures in traditional operating systems; however, they are incapable of sharing resources in a way new applications such as multimedia application require. While our approach enhances and concentrates only on network services present in current operating systems, new research in operating systems tackles this shortcoming by proposing new operating system structures. Exokernet Operating System [34] at MIT concentrates only on securely multiplexing the raw hardware and provides basic hardware primitives from which application-level libraries and services can directly implement traditional operating system abstractions specialized for appropriateness and speed. In the same lines, Nemesis Operating Systems [37] at University of Cambridge proposes low level abstractions that are close to the machine and high level abstractions that are close to the applications.

Some languages and frameworks also provide constructors to support some of the feature we addressed in this section. Visual C++ and Motif, for example, provide event driven packet reception in a similar fashion GUI events are processed. On the other hand, features such as traffic rate control and traffic monitoring, have only been provided in specific applications (e.g. Mbone tools), and we are unaware of such services being supported in reusable frameworks.

CHAPTER VI

RESILIENT AND SCALABLE PROTOCOL FOR DYNAMIC IMAGE TRANSMISSION

Synchronous multimedia applications are based on three basic components: audio, video, and shared data. While video is optional and its main function is to contribute to gain session awareness, audio is an essential media. Some systems disregard video in the benefit of audio, such as Turbo Conferencing [8], whereas others make it optional, such as IRI [41] and NetMeeting [45]. In any case, multimedia collaboration also includes a data component that normally supports or contains the main idea of discussion. Rather than sending hard copies or faxing the material to remote participants, today's collaboration systems use the network to distribute this information on the fly. Many specialized systems have been developed for that purpose, such as co-browsers [8] [19], and sharing tool engines [1] [61]. In other cases, the collaboration application includes a module for data sharing such as in [45] [41] [46]. Although all these systems provide a number of features, the major contribution of them to a collaborative session is the ability of distributing data information in real-time and to emulate a virtual projection screen or documents on a virtual table. With no doubt, the original electronic form of a document is the more faithful version and concise representation of it. HyperText Markup Language (HTML) [29], for example, is distributed to participants in co-browsers. Unfortunately, this technique is not general enough to distribute information that cannot be put in HTML format, such as the user's view of a running application. In other cases this approach can be inconvenient; for instance, to discuss the abstract and conclusions of this dissertation, the entire document needs to be loaded before displaying it and bandwidth is inevitable wasted. In all the scenarios described above, a common denominator is the desire of sharing a common view. This can be accomplished by sending an image of that visual or a flow of related images when the view changes dynamically. We believe that this paradigm is common in synchronous collaborative tools and general enough to become the building block for sharing data in multimedia collaborative applications.

A case can be made in why not to use already existent video protocols and tools for sending image flows. In fact, there is experience in its use in the MBone [42]. Lawrence Rowe, at University of California at Berkeley, has been using video technology to deliver data information in the Berkeley Multimedia, Interfaces, and Graphics Seminar (MIG). There, they either use a scan converter to translate the computer screen signal into standard video format or employ a stand camera to capture hard-copy slides. While the first video stream is reserved to the presenter’s video, the second one sends the computer screen from the converted and using H.261 format [31]. Another experience in sending data contents through video streams is found in *vic* version 2.8 [75] from University College London (UCL). This video conferencing tool was developed by the Network Research Group at the Lawrence Berkeley National Laboratory in collaboration with the University of California, Berkeley. Later, The Networked Multimedia Research Group at University College London enhanced it. One of the featured added at UCL allows the sender to select a region of the screen for frame capture as opposed to video frames. Thus, a portion of the sender’s view is transmitted to all. By selecting the origin for this rectangular region, a rectangular visual is shared.

The video approach mentioned above fulfills reasonably well the need for data distribution in many cases, especially under the lack of general-purpose alternative; nonetheless, this technique suffers from a number of shortcomings. First of all, video compression has limited the video dimensions to few sizes. This restricts its application when the information to be shared does not fit a predefined video size on the screen. On the other hand, the use of converters for sending the entire display view forces the sender to make her complete view public. In addition, it inevitably reduces the resolution to, for example, 352x288 pixels for CIF (**C**ommon **I**ntermediate **F**orm) size video as opposed to the at least 1024x768 pixels of most of today’s monitors. Another drawback is the video bandwidth requirement. Slide-show-like situations are not well handled by video compression standards. For example, in MIG seminars 64 Kbps are allocated to audio, 100 Kbps to presenter video, and 36 Kbps to presentation video out of the 200 Kbps allowed for public MBone broadcast. However, when there is a slide change, the presentation video needs to be dynamically adjusted to use more bandwidth. This type of behavior is not well managed by video compression techniques. Moreover, the inevitable electronic thermal noise, which is generated by video converters and/or analog circuits of video cards, introduces fictitious changes in the captured digital image and, therefore, leads to more data traffic.

Sending the presenter's view is very general for synchronous collaboration. In computer-based collaboration the presenter uses a view of the information being shared, thus we could think of this image as the highest level representation that any document must be able to generate to be used for discussion. Traditional techniques for sending video are not adequate for distributing this view as we argues above, thus we propose a resilient and scalable protocol for transmitting mutable images in real-time. In this context mutable images are images that can change not only in content but also in size. In addition, the experience gained in this work suggests that a generalization of video compression mechanism to allow for a continuos "video" size would also accomplish information sharing and benefit from hardware compression.

In summary, our protocol for transmitting images aims to the following requirements: a) allow for image dimension and content changes over time, b) preserve image legibility from sender to receivers, c) scalable, and d) resilient. In addition, implementation simplicity was another consideration, so that a prototype could be developed based on standard libraries and formats. In the next sections, we present the protocol and design considerations based on experimental results.

6.1 Dynamic Image Transmission Protocol

The protocol for transmitting dynamic images presented here enables data sharing by disseminating mutable images. From the communication point of view, the two main features of this protocol are resiliency and scalability. It assumes an unreliable transport protocol, so provisions are taken to overcome protocol data unit losses. In addition, no feedback is required from receivers, so it does not preclude scalability.

Dynamic images, like video, contain spatial and temporal redundancy that the protocol removes. Spatial redundancy correlates very well with distance; therefore, most of the still image compression algorithms brake the image in small bocks and then remove local redundancy. Video techniques like H.261 and H.263 [16] also use block-based coding. In principle, we could remove spatial redundancy by using any image compression standard; nonetheless, tiling is also required to remove temporal redundancy, so it has to be visible to the protocol in order to achieve temporal redundancy removal. To remove this type of redundancy motion-compensated prediction [16] has been used in video encoding. It assumes that pixels within the current picture can be modeled as a translation of those within a previous picture. Due to the high computation cost of this operation and the likelihood each image block can change, we decided against motion prediction in the general case and only use motion prediction with null motion vector. That is, we only benefit from blocks that remain unchanged from one sample to another. This also makes sense while analyzing the behavior of dynamic images. In contrast to video, these images tend to be of higher resolution than traditional video images and present lower degrees of motion. The size, for example, can be as big as a full computer screen (1024x768 pixels). Motion appears when the image contains dynamic graphics that behave like video or when it embodies scrollable regions. In any case, the protocol privileges legibility over motion. In other words, while we perceive continuos motion to happen at any frequency rate higher than 15 frames per second, we estimate that a sampling rate of around 2 samples per seconds fulfills the requirements of most types of data sharing. It also takes into consideration the shared computation power utilized in multi-media applications; so that given a bounded CPU allocation for data sharing, the bigger picture processing can only be achieved by reducing the processing cycle rate. Knowing the expected sampling rates, let’s us revisit our decision about motion prediction and better justify our argument. We believe that in low sampling rate, i.e. around 2 Hz, motion prediction loses effectiveness because at this frequency the motion vector is likely to be out of the reach of the search window of motion-compensated prediction techniques. For example, in H.263 the search window for motion prediction is such that motion of at most 16 pixels horizontally and/or vertically can be predicted. Our protocol tiles the image in square blocks, and then it encodes each block using a standard image coding to remove spatial redundancy. Only blocks that change between two image samples are encoded, thus some temporal redundancy is also removed.

Image size changes are also transmitted by the protocol. The size of an image might change from one sample to another. This info is easily distributed as part of the protocol data unit, but that is not all. In computer application the main cause of images change in size is window resizing. We observe that window resizing usually preserves the upper left content of the view regardless the side or corner used for resizing. Therefore, while comparing blocks between an image and its resized version, the protocol assumes that both samples share a common upper left region. Likewise, receivers initialize the new version of the image with the upper left content of the previous instance of the image.

The unreliable transport protocol, which our protocol relies on, forces it to take some considerations to overcome packet losses. We decided against retransmission of lost data because of its difficulties in getting feedback from an undetermined number of receivers [49]. The alternative is to send new data or controlled open-loop retransmission of the same data to eventually repair the consequences of the original lost. As introduced by the principle of Application Level Framing (ALF) [14], we define the protocol data unit (PDU) such a way that each PDU can be processed out of order with respect to other PDUs. As a result, each PDU conveys at least a tile, its coordinates within the image, a tile-based sequence number, and timestamp. We also include the image dimension in each PDU even though this info is not expected to change from tile to tile. This information could be piggybacked every so often with a tile PDU. It can also be figured out from tile's position outside the current image boundary. In principle, each altered tile needs to be sent once; however, we schedule its retransmission one more time after a random time. Thus each tile is sent at least once and at most twice in order to overcome losses. Next, we describe another reason for retransmission that makes the protocol even more tolerant to losses. The random time in sampling periods for retransmission is selected from an interval (0, MAU] (**M**aximum **A**ge of **U**pdate) to span the traffic over time.

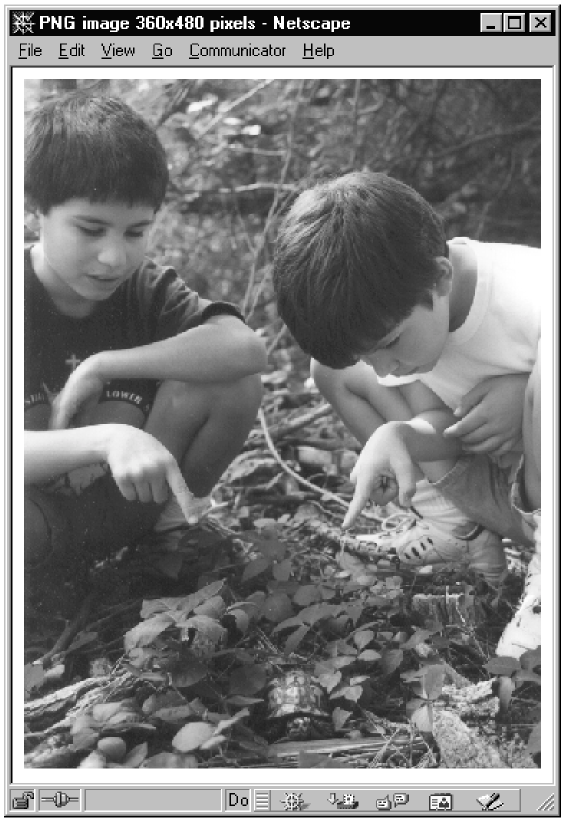
Common events in large group collaboration are participants leaving or new comers joining at any time. The first case has no effect on the protocol since no receiver's information is required; however, late comers must receive the complete image in a bounded time regardless of the image updates. The protocol fulfills this requirement by sending a refresh for each PDU after a random time taken from the interval (MAU, MAR + MAU] (**M**aximum **A**ge of **R**efresh). This ensures a full image retransmission takes place at most every MAR + MAU sampling periods. This type of refresh not only accommodates late comers but also strengthens protocol resiliency and enables the detection of removed or closed images as we discuss below. Any tile update transmission resets and reschedules the corresponding refresh.

Finally, the protocol needs to contemplate image creation and removal. The former is provided by the reception of the first PDU. The latter is a little more involved since there is no guarantee that any explicit close image message will reach all the receivers. Tile refresh messages are used in conjunction with a remove image timeout to determine that the dynamic image was closed and no close message has been heard. The timeout is reset upon arrival of any image tile. Even though the timeout is sufficient to remove closed images, the protocol transmits a close image message when the sender destroys the image, so that receivers of such a data unit can quickly react and reduce the latency of this operation.

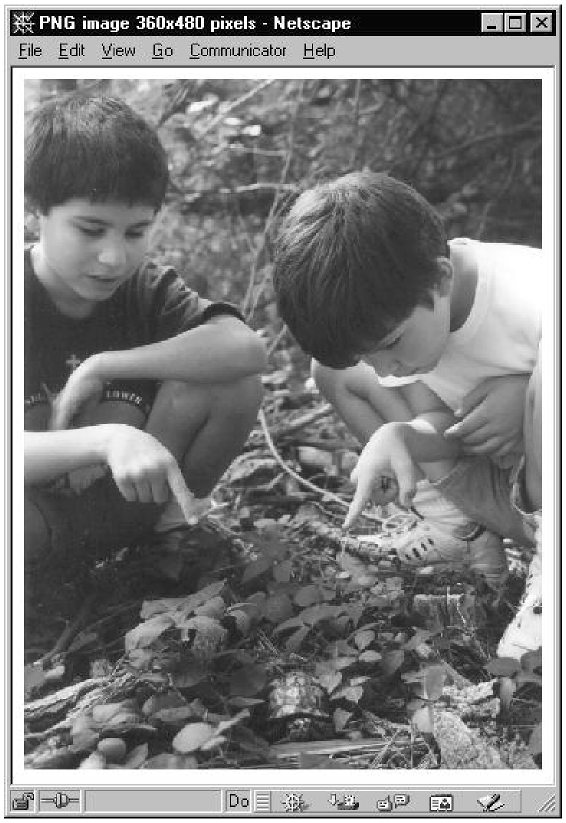
In the following sections we discuss the parameters of the protocols and their impacts on performance. First, we analyze the effect of two common compression standards for still image encoding that we tested for tile compression. Then, we model the processing time for each image sampling and use it to estimate the sampling rate. Finally, we discuss the tradeoffs in selecting the tile size.

6.2 Tile Compression Format Study

The protocol employs a still image compression for tile coding; thus well-tested and refined public domain libraries can be used in the protocol implementations. In our study, we considered and compared Joint Photographic Experts Group (JPEG) [77] and Portable Network Graphics (PNG) [11] encoding implementations. The criteria for selecting the compression technique are compression time, compression ratio, and legibility. These factors were evaluated as a function of the tile size, especially around 32x32 pixels. For relatively small images, the format overhead plays an important role. The compression time depends not only on the machine but also the library. For a particular library, each format can be tested to measure the compression time. Compression ratio is another coordinate in the comparison space. In this case, we measured big variations between these two formats. While for some picture type of images JPEG overruns PNG in a factor of 10, in text type of images PNG is better than JPEG in a similar factor. Another factor in consideration is the lossy and lossless nature of JPEG and PNG respectively. Due to the lossy nature of JPEG, a quality factor needs to be provided for compression. While quality values of around 50% are normally acceptable for pictures, higher values are required for legible text images. PNG, on the other hand, is lossless. It offers good compression rates for text and line type of images, but it does compress well the redundancy of real-world scenes.


PNG image 360x480 pixels - Netscape
File Edit View Go Communicator Help

a)

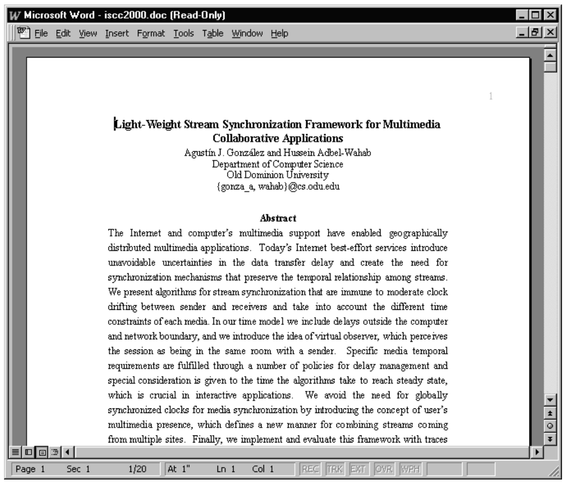

PNG image 360x480 pixels - Netscape
File Edit View Go Communicator Help

b)

Fig. 59. PNG/JPEG comparison for real-world images. a) 113KB PNG image and b) the 46KB JPEG version (75% quality factor).

Fig. 59 and Fig. 60 show two cases where PNG and JPEG formats have totally opposite results in terms of compression ratio. Both figures were obtained with Microsoft Photo Editor, Fig. 59 from a 334KB 388x566-pixel PNG and Fig. 60 from a 21BK 680x580-pixel PNG color pictures respectively by saving them in 8-bit gray scale mode. The real-world picture was reduced to 70% of its original size and the text image to 40%

of its original size. Even though the images' qualities cannot be fully judged due to the loss of resolution of this hard copy, they clarify what we mean by real-world and text images. Although it is not perceivable from these figures, when displayed 100% size on the screen, the real-world JPEG image is not as good as the PNG. On the other hand, the text image seems identical in both formats.


Microsoft Word - iscc2000.doc (Read-Only)
File Edit View Insert Format Tools Table Window Help
Page 1 Sec 1 1/20 At 1" Ln 1 Col 1 REC TRK EXT OVR WPH

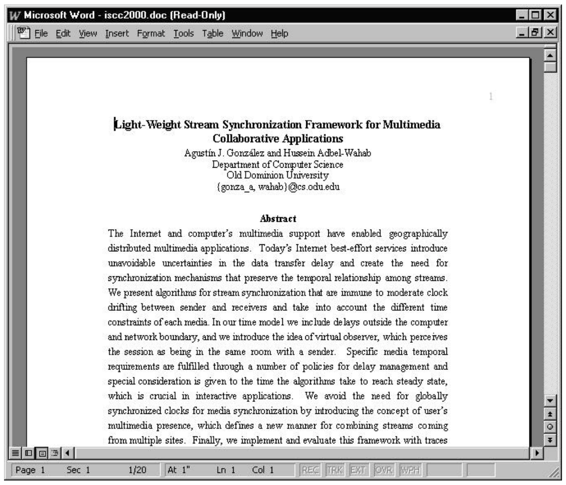
Light-Weight Stream Synchronization Framework for Multimedia Collaborative Applications

Agustín J. González and Hussein Adbel-Wahab
Department of Computer Science
Old Dominion University
{gonza_a, wahab}@cs.odu.edu

Abstract

The Internet and computer's multimedia support have enabled geographically distributed multimedia applications. Today's Internet best-effort services introduce unavoidable uncertainties in the data transfer delay and create the need for synchronization mechanisms that preserve the temporal relationship among streams. We present algorithms for stream synchronization that are immune to moderate clock drifting between sender and receivers and take into account the different time constraints of each media. In our time model we include delays outside the computer and network boundary, and we introduce the idea of virtual observer, which perceives the session as being in the same room with a sender. Specific media temporal requirements are fulfilled through a number of policies for delay management and special consideration is given to the time the algorithms take to reach steady state, which is crucial in interactive applications. We avoid the need for globally synchronized clocks for media synchronization by introducing the concept of user's multimedia presence, which defines a new manner for combining streams coming from multiple sites. Finally, we implement and evaluate this framework with traces

a)


Microsoft Word - iscc2000.doc (Read-Only)
File Edit View Insert Format Tools Table Window Help
Page 1 Sec 1 1/20 At 1" Ln 1 Col 1 REC TRK EXT OVR WPH

Light-Weight Stream Synchronization Framework for Multimedia Collaborative Applications

Agustín J. González and Hussein Adbel-Wahab
Department of Computer Science
Old Dominion University
{gonza_a, wahab}@cs.odu.edu

Abstract

The Internet and computer's multimedia support have enabled geographically distributed multimedia applications. Today's Internet best-effort services introduce unavoidable uncertainties in the data transfer delay and create the need for synchronization mechanisms that preserve the temporal relationship among streams. We present algorithms for stream synchronization that are immune to moderate clock drifting between sender and receivers and take into account the different time constraints of each media. In our time model we include delays outside the computer and network boundary, and we introduce the idea of virtual observer, which perceives the session as being in the same room with a sender. Specific media temporal requirements are fulfilled through a number of policies for delay management and special consideration is given to the time the algorithms take to reach steady state, which is crucial in interactive applications. We avoid the need for globally synchronized clocks for media synchronization by introducing the concept of user's multimedia presence, which defines a new manner for combining streams coming from multiple sites. Finally, we implement and evaluate this framework with traces

b)

Fig. 60. PNG/JPEG comparison for text images. a) 16KB PNG image and b) the 90KB JPEG version (75% quality factor).

In addition to comparing these two formats for full-size images, we compared them on tile-size images. First, we investigate the overhead introduced by each of them by coding a wallpaper type of image and an empty image, as shown in Fig. 61. By duplicating the same content over larger square regions, we generated expanded versions of these images.

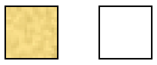


Fig. 61. Wallpaper and empty images.

The results for PNG and JPEG overheads versus the size of uniform images are illustrated in Fig. 62. In both cases, JPEG has an overhead of around 600 bytes while

PNG overhead is near 100 bytes; however, JPEG image size grows with a lower slope compared with PNG's. The higher JPEG overhead is due to the quantization table and huffman table stored in the image marker section of the format. In Section 6.7, we discuss options to factor it out and potentially reduce compressed tile size.

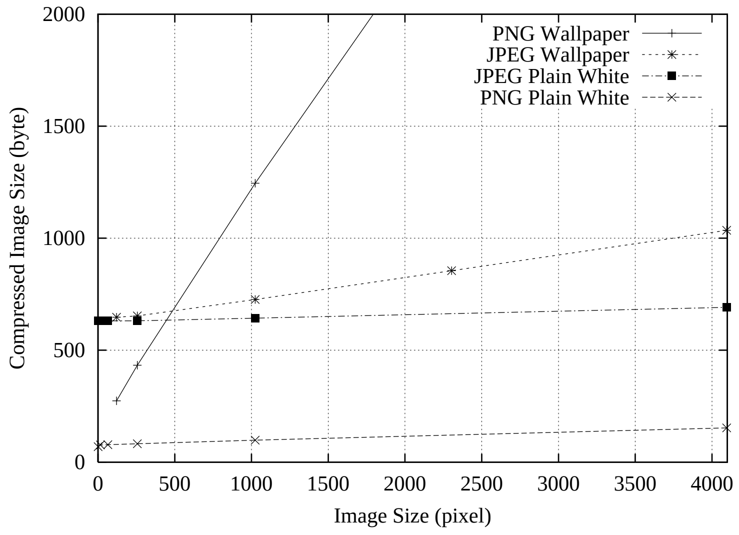


Fig. 62. PNG and JPEG overhead comparison for small images.

In order to measure the effect of the compression format on the protocol traffic for real cases of dynamic images, we implemented the protocol on Java 2 SDKv1.2.2 and also employed Java Advanced Imaging 1.0.2 for compressing tiles [72]. While varying the tile size, we measured the total traffic in bytes due to protocol data units after compressing and packetizing all the tiles. We used the color versions of the images of Fig. 59 and Fig. 60. The results are depicted in Fig. 63.

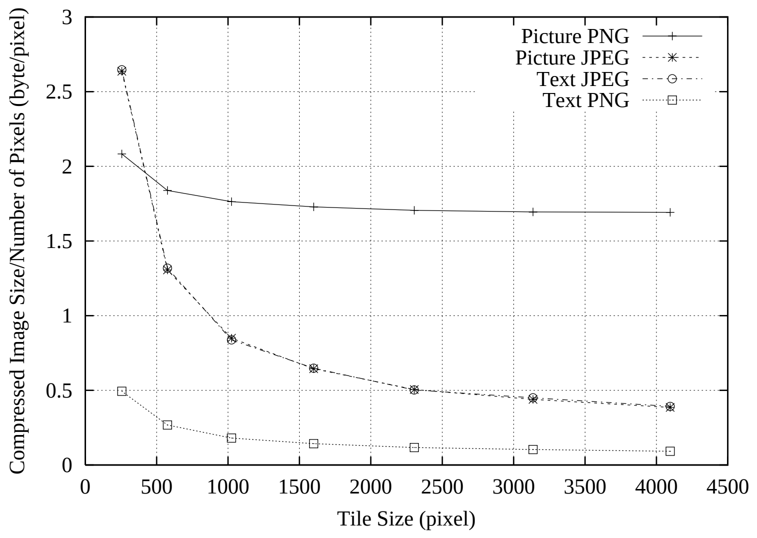


Fig. 63. Protocol compression using PNG and JPEG (75% quality factor).

The protocol traffic is quite stable for JPEG compression, yet it shows a big variation with PNG compression depending on the image. JPEG overhead is manifested here by a decreasing performance as the tile size is reduced. This forces a higher number of tiles to be compressed per image and as a result the fixed overhead per tile defines the compression limit. As for these results, it appears that tiling decreases the performance; nonetheless, the counter argument is that smaller tiles enable more temporal redundancy removal. In addition, protocol data unit fragmentation also plays a role in determining an optimal tile size. We discuss these tradeoffs in the following section.

6.3 Selecting Tile Size

The definition of the tile size has a crucial effect on performance. As stressed by the principle of Application Level Framing (ALF) [14], loss of data unit fragments prevent data unit reconstruction and cause bandwidth misusage due to the reception of data that cannot be processed. We measured packet size after compression using PNG and JPEG coding formats, as shown in Fig. 64.

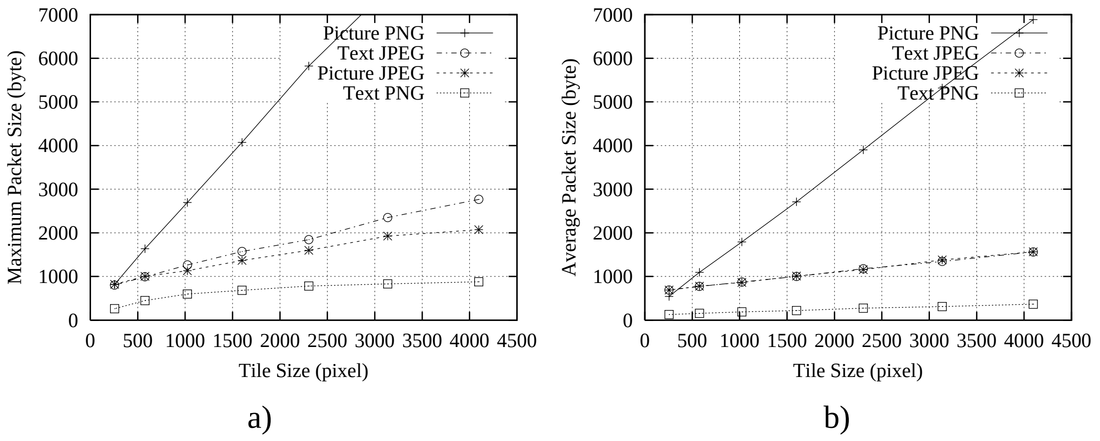


Fig. 64. Application data unit sizes as function of tile size. a) Maximum value and b) Average value.

For PNG encoding, only 16x16-pixel tile size leads to a single network frame per packet for all tiles, and fragmentation is unavoidable for any other size on real-word images. For text-like images, on the other hand, PNG does a very good job in producing a single fragment even for 64x64-pixel tiles. In contrast, JPEG is much more uniform in its results. The average and maximum packet sizes do not vary much with the image content. As a result, we selected the 40x40-pixel tile to be the biggest tile that does not lead to fragmentation on Ethernet whose Maximum Transmission Unit (MTU) is 1,500 bytes. In any case, it is a protocol parameter to be tuned based on each sender network connection. It worth to mention that around 60% of the JPEG packet contains coding tables and only 40% of it the compressed image data. Fragmentation imposes a penalty not only on bandwidth but also in transmission processing time as we elaborate in the next section.

6.4 Model for Protocol Processing Time

The protocol requires the specification of a number of parameters that depends on the processing time of the protocol. The time model presented here describes its major components and quantifies it based on our protocol implementation.

The protocol can be analyzed in the following steps: image capture, temporal redundancy removal, tiles compression, and tiles transmission. At the receiving site, it receives protocol data units, decompresses tiles, draws tile in image, and displays image. Image capture at the sender and image display at the receiver can be thought of as steps outside the protocol scope since they depend on particular applications. We have included them here for completeness.

We use the following notation for the processing times associated to each of these steps.

t_s: time to take a new image sample.

t_{tr}: time to compare all tiles and determine changes in image (temporal redundancy).

t_{sr}: time to compress all changed tiles (spatial redundancy).

t_{tx}: transmission time of all changed tiles.

t_{rx}: reception time for all tiles of an image sample.

t_{de}: single tile decompression time.

t_{dr}: time to draw a tile in recipient image.

t_{di}: time to display tile changes.

These times depend on how much the image changes from one sample to another. For each time, we define extreme values $\widehat{t}_{xx}$ and $\breve{t}_{xx}$ to be the times for a complete image change and no change at all. We assume that $\hat{t}_s = \breve{t}_s$ and that $\breve{t}_{tr} = 0$ since in face of a complete it is detected by the first pixel of each tile. We have neglected the cost of invoking the comparison function. Let t_{ps} be the protocol processing time at the sender for one image sample and be f the fraction of the tiles that have changed, then:

$$t_{ps} = t_s + \widehat{t}_{tr}(1-f) + f\left(\widehat{t}_{sr} + \widehat{t}_{tx}\right)$$

In other words, the processing time is given by the sum of the image sampling time and the tile processing. For each tile, the latter is either the time to detect no change (all the pixels need to be checked) or the compression and transmission time. In an image with partial changes, the two components of the tile processing are weighted by the fraction each situation occurs in the image. For simplicity, we have neglected the retransmission of tiles to overcome losses and support late comers[8].

[8] $\alpha(1-f)t_{tx}$ or $\alpha(1-f)(t_{sr}+t_{tx})$ should be added when compressed tiles are buffered for retransmission or when they are not, respectively. Alpha is a coefficient that depends on the frequency of tile retransmissions.

At the receiving site, the unit of processing changes. While the sender processes complete image samples, receivers compute one data unit or tile at a time. Let t_{pr} be the protocol time at a receiver to process one tile, then:

$$t_{pr} = t_{rx} + t_{de} + t_{dr} + t_{di}$$

Indeed, it is up to the application to display tiles as they arrive or after all the tiles of an image sample have been updated in the receiver's image. Like the expression for t_{ps}, we have neglected tile retransmissions. In this case, there is a cost for tile reception, but the rest of the processing is cancelled when the same tile sequence number has already been received.

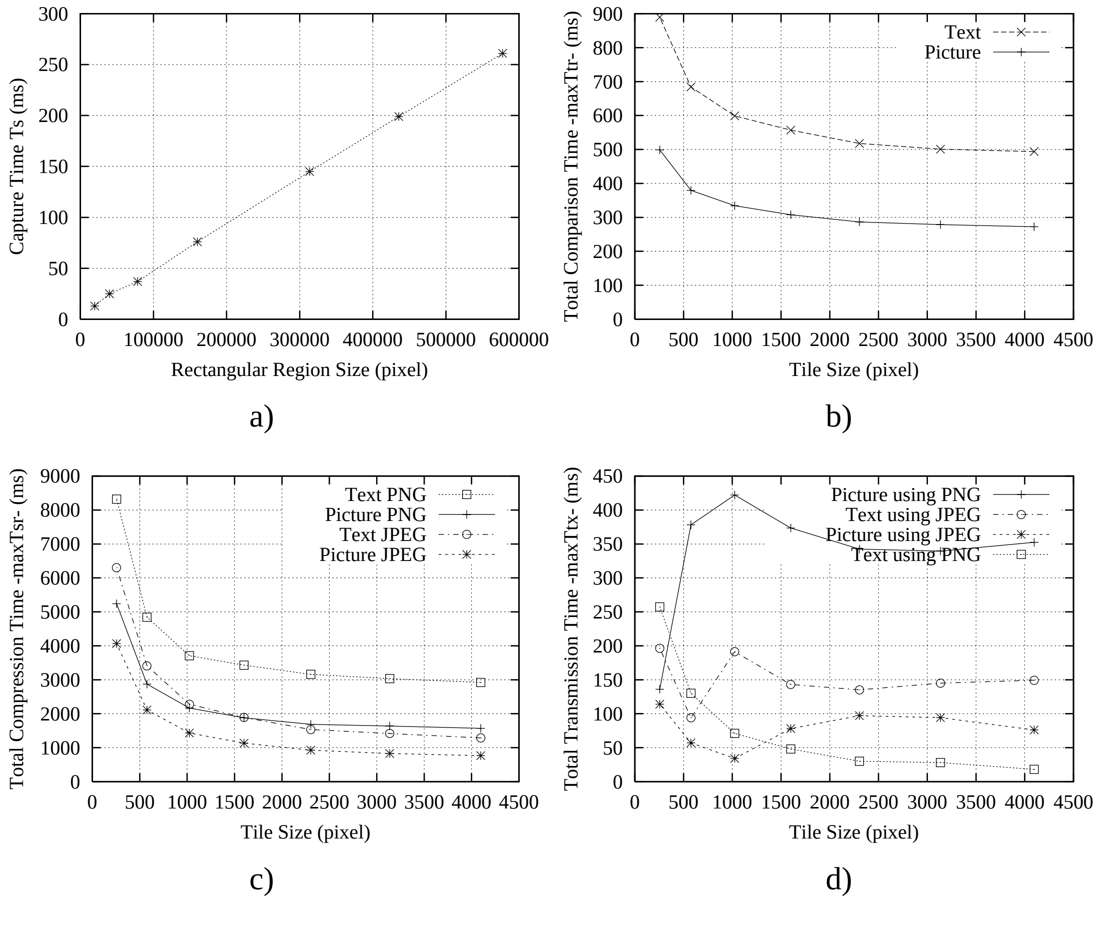


Fig. 65. Protocol processing times for sender in image sharing application.

We measured the processing time using our prototype implementation. Then, we used it in an application that captures rectangular regions from a computer monitor and transmits them using the dynamic image protocol. The receiver's application receives and displays the remote rectangular region. This simple application was written on top of the protocol implementation and accomplishes data sharing by sharing dynamic images captured from sender screen.

Fig. 65 shows the sender processing times due to each of the steps discussed above. We used the two images depicted in Fig. 59 (picture) and Fig. 60 (text). Fig. 65a confirms the linear behavior of the sampling process. Fig. 65b and Fig. 65c plot decreasing functions of the tile size even though the number of processed pixels do not depend on tile size. This trend is explained by the decaying overhead in function calls when the number of tiles per image hyperbolically decreases, as shown in Fig. 66b. We also noted that the compression speed of each coder (PNG or JPEG) is virtually independent of the content of the image, as shown by the quasi-overlapping curves for PNG and JPEG of Fig. 66b. This graph demonstrates the speed up of the JPEG library over PNG's. This behavior was later confirmed with Sun Microsystems; while JAI 1.0.2 uses native methods to accelerate JPEG compression, it does not do that with PNG. Out of the four components of the processing time -sampling, temporal redundancy removal, spatial redundancy removal, and transmission-, spatial redundancy removal is the most expensive (Fig. 65c).

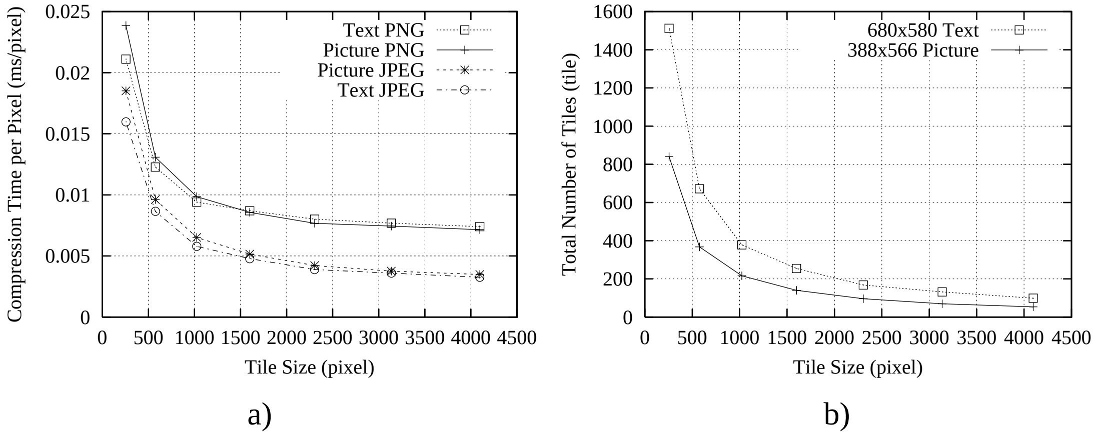


Fig. 66. Compression time. a) Time per pixel and b) time for total number of tiles.

Fig. 65d is quite interesting and shows the impact of protocol data unit fragmentation on transmission time. At first sight, we expected a decaying trend while increasing tile size due to reduced amount of coding overhead, as illustrated in Fig. 62. However, this conduct is only observed in the transmission of PNG text-like tiles. As we saw in Section 6.3, each coded tile of this image fits in one network fragment, so no fragmentation is performed. On the other hand, the real-word image for the same compression format runs into fragmentation for tiles greater than 24x24 pixels. The cost of fragmentation is high since not only more accesses to the media are required but also more work is demanded from the data link layer. As the tile size keeps growing, the protocol data unit size also grows and the number of tiles decreases; as a result, the transmission time tends to stabilize around 350 ms.

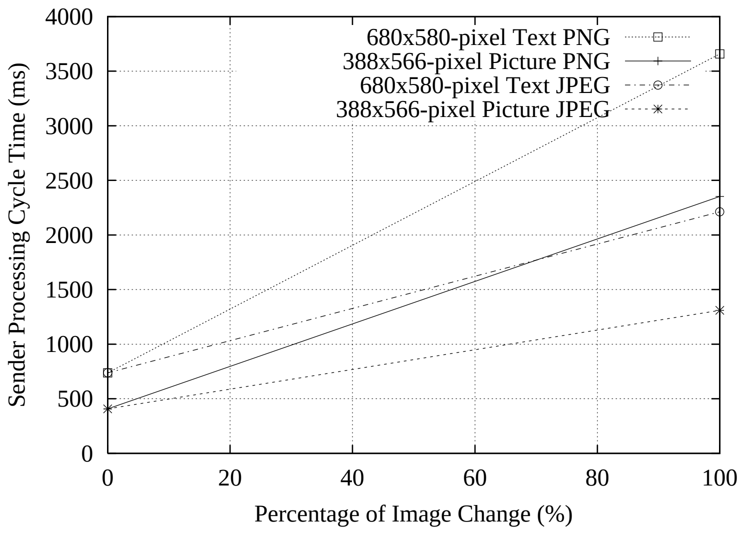


Fig. 67. Processing time model applied to sender (40x40-pixel tile).

Overall, JPEG encoding ended up being faster for computing the complete processing cycle of this application mainly due to its library speedup over PNG, as shown in Fig. 67. This result shows that small updates can be sent at a rate of 2 Hz for this image size, and it takes up to around 2 seconds to send an entire new image. These lower and upper bounds are directly proportional to the image size.

In contrast to the sender part of the test application that depends on native calls for image capture, receivers can be run on WinNT or UNIX machines. The results for both platforms are shown in Fig. 68, Fig. 69, Fig. 70, and Fig. 71. For example, using 40x40-pixle tiles and JPEG compression for spatial redundancy removal, all the tiles for the picture-like image were processed and displayed in 790 ms and 718 ms in WinNT and UNIX respectively. Likewise, the text-like image took 1,008 ms and 1,274 ms in WinNT and UNIX. We noticed three interesting points. First, the resource utilization of the decompression step propagates to the drawing and display steps whose results were believed independent of the image content since these operations are performed on raw pixels. Second, this application revealed the difficulties of X server-client paradigm for updating highly mutable images. Finally, sender rate control had to be introduced to limit to 100 KByte/s the rate of tiles being transmitted. Otherwise, receivers lost tiles, especially on WinNT. We did not measure data unit reception time. It might explain the overall better performance of the UNIX receivers. In addition, the image sender processes all tiles of an image sample in one loop whereas receivers follow a tile driven processing that includes drawing and displaying. These two steps make receiver go slower than sender in processing tiles and tile losses are observed.

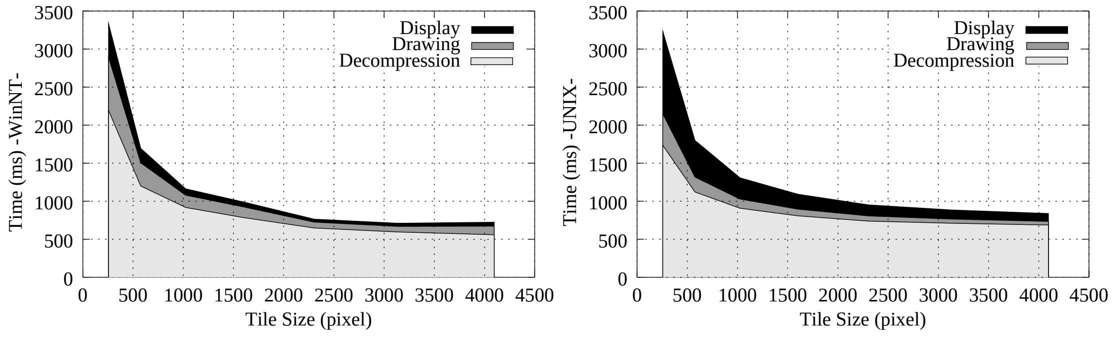


Fig. 68. Processing time for 388x566-pixel picture on WinNT and UNIX using PNG.

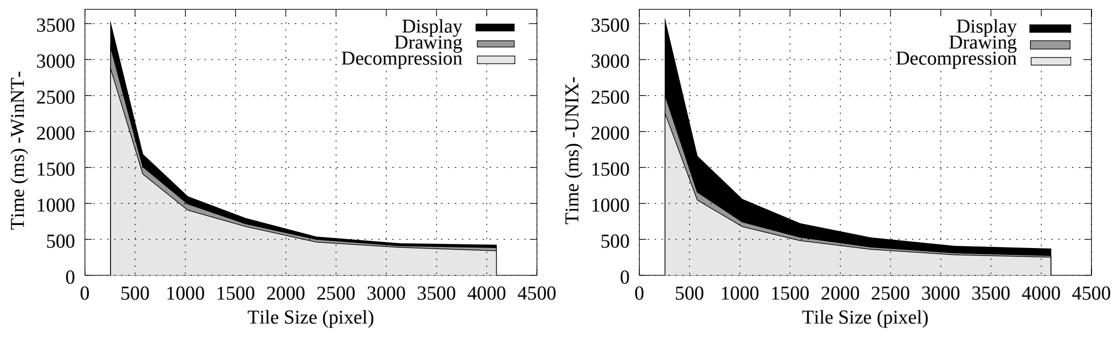


Fig. 69. Processing time for 388x566-pixel picture on WinNT and UNIX using JPEG.

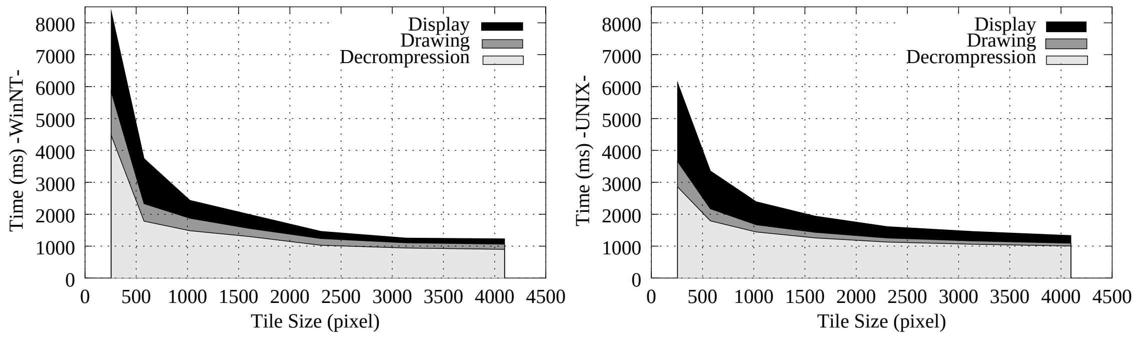


Fig. 70. Processing time for 680x580-pixel text image on WinNT and UNIX using PNG.

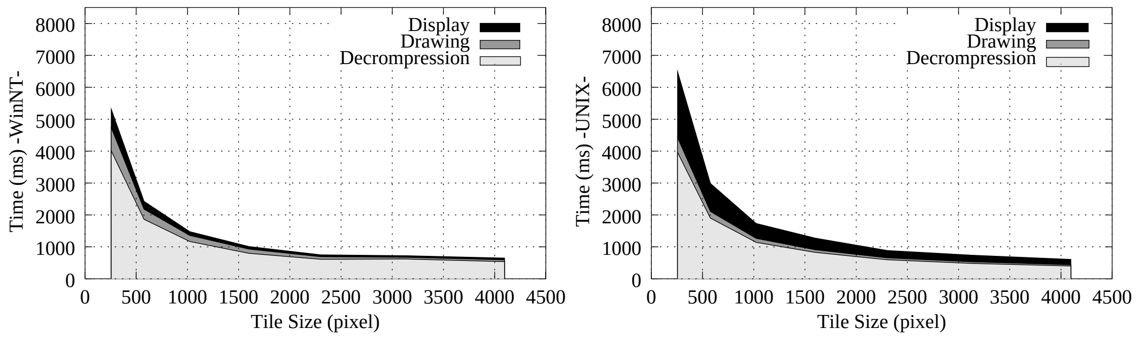


Fig. 71. Processing time for 680x580-pixel text image on WinNT and UNIX using JPEG.

6.5 Protocol Processing Speedup.

Implementations of the protocol described in Section 6.1 may operate with different tradeoffs between information delay versus information accuracy. For example, we have assumed that the dynamic image is sampled and then processed for distribution. Another approach is to sample and then process one tile at a time. The former technique ensures receivers to obtain a sequence of snapshot of the sender's image, whereas the latter scheme ensures each tile is at most one tile processing time old at display moment. When the whole image changes rapidly due to scrolling or quick browsing, image-based processing skips complete images while tile-oriented processing partially displays intermediate images.

We also observed a tradeoff between processing time and accuracy when removing temporal redundancy. No changes in an image can only be determined by comparing every pixel with its corresponding in the previous sample. This operation takes around 300 ms for a 388x566-pixel image. This time and the sampling time determine the maximum sampling rate. On the other hand, it is very likely that tile updates affect many pixels, so we experimented with statistical comparison and sub-sampling comparison to speedup this task. As illustrated in Fig. 72, skipping every other line and column in tiles, we checked 25 % of the pixels in a systematic fashion and obtained a speedup factor of 2. In order to avoid missing tile updates completely, in every sample processing the 25% of compared pixels is distinct, so that the algorithm scans the whole image after 4 samples. We also tried statistic comparison by randomly selecting 5% of the pixels. It led to poor results in processing time and effectiveness. Although fewer pixels are touched, the overhead in computing two random numbers per pixel comparison makes the technique time consuming. Moreover, the lack of control in the pixels being selected for testing leads to longer delays for many altered tiles.

Other enhancements to reduce processing time are discussed in Section 6.7.

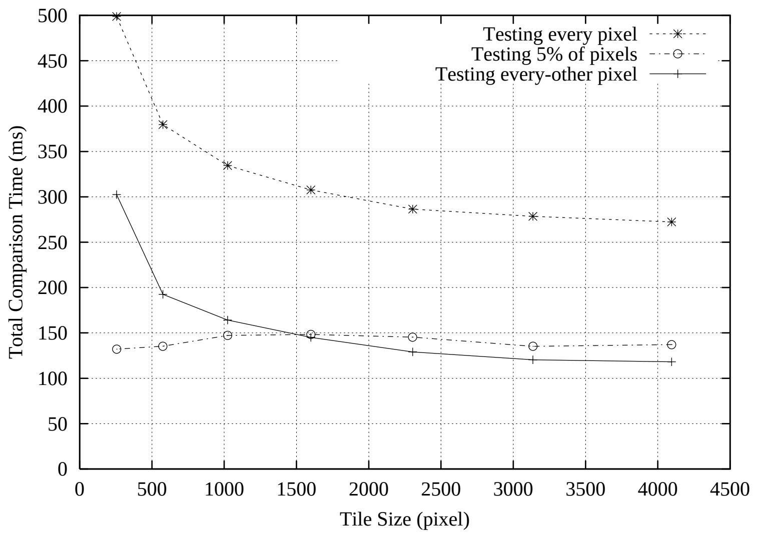


Fig. 72. Comparison algorithm speedup on 388x566-pixel image.

6.6 Related Work

The idea of sharing data by sharing images has been explored in the VNC project [61] at Cambridge University and the IRI project at Old Dominion University. While Virtual Network Computing proposes image distribution over reliable transport protocol, specifically TCP, our protocol also works over unreliable channels. Therefore, data unit size and resiliency considerations are avoided by VNC. On the other hand, we believe our protocol can handles larger groups and provides better responsiveness than VNC. VNC's graphics primitive is, like our protocol, the distribution of rectangle of pixels at a given position. It uses raw-encoding or copy-rectangle encoding. In the first one, the pixel data for a rectangle is simply sent in left-to-right scanline order. In contrast, we use still image compression for tiles. VNC avoids compression time but demands more transmission bandwidth than our protocol. Copy-rectangle encoding allows receivers to copy rectangles of data that are already locally accessible. We decided against this type of primitive because of the high processing cost in determining tile motion or translation.

Mark Palkow, Yanshuang Liu, and Catherine Train[9] also worked on techniques for sharing applications by sharing their images on the screen. For temporal redundancy removal, they transmitted image differences that then were compressed using PNG encoding. While this scheme is suitable for reliable communication channels, it cannot be used over unreliable ones. It also requires special treatments for latecomers. They need a complete image on which they could apply subsequence differences. Their work was of great value and source of inspiration for us in the first stages of our work.

Video Conferencing tool has also been used for data sharing by transmitting dynamic images as video frames. Its main advantage is the access to highly refined and tuned libraries for video streaming that reach higher frame rate than image processing. Its main shortcoming is the limited sharable region of the screen. Similarly, conferences on the MBone have made little use of whiteboard type of tool for data sharing and started to use video for distributing conference content. They capture data information from either a projection screen with a camera or from computer screen with video converters and regular video cards.

6.7 Future Work

This work can be extended in two independent paths. One aims to reduce both processing time and bandwidth consumption of the protocol. The other approach is to adapt current video compression techniques to fulfill the requirements of data sharing. Our experience indicates that a full image update might take up to 2 second with our current implementation. This makes browsing difficult specially when scrolling. We believe that timing out long tile processing can reduce computation time and bandwidth. Let's us use an example to introduce and explain this concept. When one visits a new location on the Internet, a new image sample initiates the 2- or 3-second protocol cycle. Assume that very shortly the sender scrolls the browsing window. Although the image has changed, the current protocol uses bandwidth and processing power in finishing the

[9] Mark Palkow worked as intern during the summer of 1998 at Old Dominion University Computer Science Department. Catherine Train and Yanshuang Liu did their master's projects on different aspects of this sharing tool engine.

processing of the first site view. As an alternative, we propose to timeout tile processing, so that no tile is sent after *x* seconds of sampling. When the timeout is reached, a new sample is taken and the tile processing continues from the last tile sent of the previous image. This modification of the protocol is vital for low bandwidth clients where the transmission time limits the sampling rate.

Another technique for traffic reduction is to use in-time compression format. There is no encoding scheme that performs the best in all the cases; therefore, it is worth to investigate a simple test for identifying the best compression format for a given tile content. For example, a straightforward approach, though expensive, is to compress each tile with each format and then to transmit the smallest result. If the extra time spent in this test is smaller than the saving in transmission, this approach saves not only bandwidth but also processing time. Another heuristic to achieve this gain is to compare the compression ratio against expected values. Poor ratios would suggest the use of another compression format for the same tile in the next image sample. A tile-oriented encoding, as opposed to image oriented, is expected to produce noticeable improvements in face of heterogeneous images such as those of today's web sites.

In our implementation and measures JPEG compression gave the best tradeoff between processing time and traffic. We wonder though if the high apparent overhead of the quantization tables and hamming tables that reach 60% of the total data for small tiles can be factored out. Java Advanced Imaging supports abbreviated JPEG encoding, which lets developers decide to generate either compression tables or data.

New video compression standards have enabled promising techniques for sharing images. H.263+ [16], for example, supports not only five standardized picture formats (sub-QCIF, QCIF, CIF, 4CIF, and 16CIF) but also custom picture size. This feature removes the major drawback we have pointed out of video encoding. In addition, we propose the study of hardware support for video compression to alleviate the normally overloaded CPU in multimedia applications. In the meantime, public-domain implementations can be used instead (e.g. [16]).

CHAPTER VII

IMPLEMENTATION AND EXPERIMENTAL RESULTS

The ideas presented in the preceding chapters could not have been developed and refined without a strong experimental component. We refined and expanded our initial ideas by looking at new scenarios, many of which were the results of our experimentation. With the only exception of intra- and inter-stream synchronization, we implemented prototypes for all the other components we have proposed in this thesis for the semantic-based middleware. The ideas on stream synchronization were tested through simulation as we described in Chapter IV. We implemented the Lightweight Floor Control Framework for localized resources presented in Chapter II. We fully implemented the new Java class for multicast socket presented in Chapter V and the classes for managing the input and output buffers for network Application Data Unit (ADU). The last implemented component of the middleware was the protocol for sharing dynamic images in real time. In addition, we used its basic prototype and extended it for *compound images* transmission. We call compound image to a set of rectangular images that might have overlapped areas and their union forms a rectangular polygon. . The properties of extensibility, reusability, scalability, and flexibility of the middleware could not have been tested without its use in a challenging application that integrated several of the components of this middleware. Thus, we designed and implemented a sharing tool engine. It enables sharing of any application visible on a Win95, Win98 or WinNT[10] workstation. We have successfully tested receivers on WinNT and Solaris 2.6; however, recipients could be on any machine that runs Java and the Java Advanced Imaging package (JAI) [72]. Indeed, only Java is strictly required since JAI can run over pure Java code with some loss in performance. We selected this application for its relevance to multimedia collaboration and distance learning, which are two important areas of research in the Computer Science Department at Old Dominion University.

[10]Hereafter, we will mention only WinNT even though we also mean Win95, Win98, and possibly Win2000.

In the next sections, we describe Odust, the sharing tool developed on top of the components of our middleware, and the extensions of the dynamic image transmission protocol to compound images.

7.1 Odust Description

Odust is a distributed cross-platform application that enables data sharing in synchronous multimedia collaboration. Its current version allows sharing of any application visible on the screen of a WinNT machine. This includes any X-application running over an X-server for WinNT such as ***Exceed***. While the owner of the application to be shared operates the real instance of it on the screen, the other participants see and operate images, which are generated by Odust and are in many ways indistinguishable from the real application. Sharing is done with *process granularity* meaning that all the windows belonging to a process are shared atomically. A floor control service allows any receiver to request the control of the shared application by preempting it from the current holder. Although one receiver can have the floor at a time, the shared tool owner running the real version of it can also operate it at any time. A drawback of this technique is the interference of the floor holder input events, i.e. keyboard and mouse, with the same input devices at the application owner’s machine. Due to the lightweight nature of the middleware protocols, any participant can leave the collaboration session at any moment. Likewise, anybody can join the session at any time. These two situations have virtually no effect on the other participants. For example, if the floor holder crashes or leaves, the floor holder becomes “nobody”. Users joining the session late reach a synchronic view within a bounded time, which is a parameter in Odust. Multiple participants can share their applications at any point of time with a limit of one per site. Each shared tool is displayed in a separate window at receiving users.

Fig. 73 shows one of the multiple scenarios where Odust can be used. Scalability is gained mainly due to the use of IP multicasting, which is a network requirement for Odust to work in more than 2 participant sessions. It also works over unicast network for 2-party sessions. This feature is basically inherited from the unified unicast-multicast API provided by the network services of our middleware, as described in Chapter V. The current version of our middleware does not support application layer multicasting. An extension of the network services could easily include this facility that then automatically would become a feature of this application. The following four figures illustrates the view that each of the four users of the Fig. 73 sees on their screens.

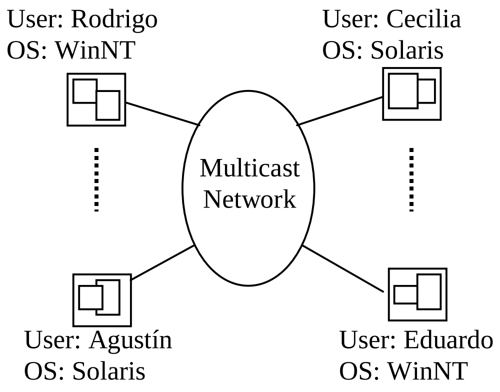
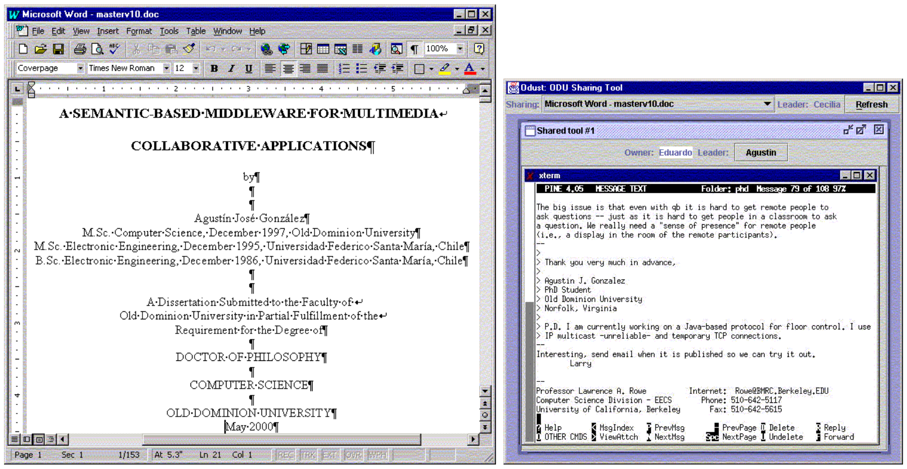


Fig. 73. Tool sharing scenario with Odust.


Microsoft Word - masterv10.doc
File Edit View Insert Format Tools Table Window Help
Coverpage
Times New Roman
12
100%
Page 1 Sec 1 1/153 At 5.3" Ln 21 Col 1
REC TRK EXT OVR WPH
Odust: ODU Sharing Tool
Sharing: Microsoft Word - masterv10.doc
Leader: Cecilia
Refresh
Shared tool #1
Owner: Eduardo Leader: Agustin
xterm

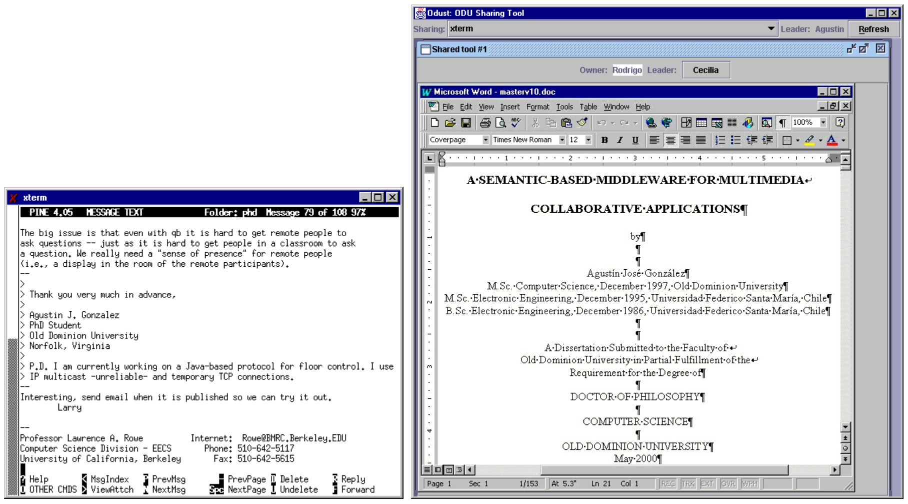
A·SEMANTIC-BASED·MIDDLEWARE·FOR·MULTIMEDIA↵
COLLABORATIVE·APPLICATIONS¶
by¶
¶
¶
Agustín·José·González¶
M.Sc.·Computer·Science,·December·1997,·Old·Dominion·University¶
M.Sc.·Electronic·Engineering,·December·1995,·Universidad·Federico·Santa·María,·Chile¶
B.Sc.·Electronic·Engineering,·December·1986,·Universidad·Federico·Santa·María,·Chile¶
¶
¶
A·Dissertation·Submitted·to·the·Faculty·of·↵
Old·Dominion·University·in·Partial·Fulfillment·of·the↵
Requirement·for·the·Degree·of¶
¶
DOCTOR·OF·PHILOSOPHY¶
¶
COMPUTER·SCIENCE¶
¶
OLD·DOMINION·UNIVERSITY¶
May·2000¶

```
  PINE 4.05   MESSAGE TEXT            Folder: phd  Message 79 of 108 97%

The big issue is that even with qb it is hard to get remote people to
ask questions -- just as it is hard to get people in a classroom to ask
a question. We really need a "sense of presence" for remote people
(i.e., a display in the room of the remote participants).
--
>
> Thank you very much in advance,
>
> Agustin J. Gonzalez
> PhD Student
> Old Dominion University
> Norfolk, Virginia
>
> P.D. I am currently working on a Java-based protocol for floor control. I use
> IP multicast -unreliable- and temporary TCP connections.
--
Interesting, send email when it is published so we can try it out.
        Larry


--
Professor Lawrence A. Rowe          Internet:  Rowe@BMRC.Berkeley.EDU
Computer Science Division - EECS       Phone: 510-642-5117
University of California, Berkeley       Fax: 510-642-5615

? Help        < MsgIndex   P PrevMsg        - PrevPage D Delete      R Reply
O OTHER CMDS  > ViewAttch  N NextMsg      Spc NextPage U Undelete    F Forward
```

Fig. 74. The real MS-word application and Odust interface viewed by Rodrigo.

Rodrigo shares an MS-Word application, as shown in Fig. 74. MS-Word runs outside Odust the same way it does any application on his machine. In addition, he receives the *xterm* being shared by Eduardo (owner label) but controlled by Agustín (leader label). Even though the *xterm* here is a UNIX application, it is run via an XWindow-server on

WinNT. Rodrigo selects what to share from the upper menu of Odust. On this widget, he also learns who has the floor of the tool he shares, Cecilia in at this time.


xterm
Odust: ODU Sharing Tool
Sharing: xterm
Leader: Agustin
Refresh
Shared tool #1
Owner: Rodrigo
Leader: Cecilia
Microsoft Word - masterv10.doc
File Edit View Insert Format Tools Table Window Help
100%
Coverpage
Times New Roman
12
Page 1 Sec 1 1/153 At 5.3" Ln 21 Col 1 REC TRK EXT OVR WPH

```
PINE 4.05   MESSAGE TEXT                Folder: phd  Message 79 of 108 97%

The big issue is that even with qb it is hard to get remote people to
ask questions -- just as it is hard to get people in a classroom to ask
a question. We really need a "sense of presence" for remote people
(i.e., a display in the room of the remote participants).
--
>
> Thank you very much in advance,
>
> Agustin J. Gonzalez
> PhD Student
> Old Dominion University
> Norfolk, Virginia
>
> P.D. I am currently working on a Java-based protocol for floor control. I use
> IP multicast -unreliable- and temporary TCP connections.
--
Interesting, send email when it is published so we can try it out.
       Larry


--
Professor Lawrence A. Rowe          Internet:  Rowe@BMRC.Berkeley.EDU
Computer Science Division - EECS       Phone: 510-642-5117
University of California, Berkeley       Fax: 510-642-5615

? Help        < MsgIndex   P PrevMsg      - PrevPage D Delete     R Reply
O OTHER CMDS > ViewAttch   N NextMsg    Spc NextPage U Undelete   F Forward
```

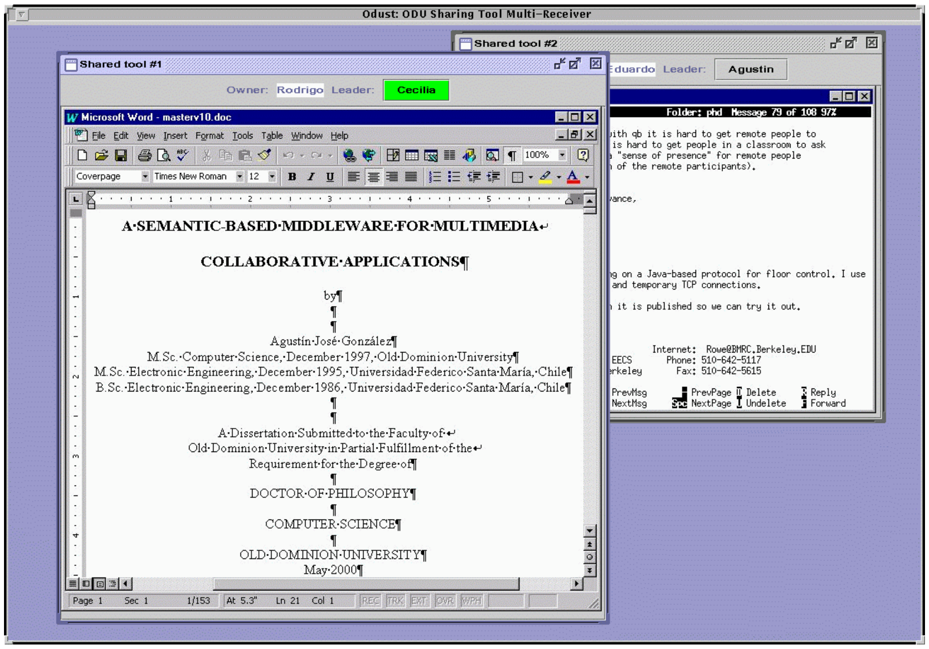
A·SEMANTIC-BASED·MIDDLEWARE·FOR·MULTIMEDIA↵

COLLABORATIVE·APPLICATIONS¶

by¶
¶
¶
Agustín·José·González¶
M.Sc.·Computer·Science,·December·1997,·Old·Dominion·University¶
M.Sc.·Electronic·Engineering,·December·1995,·Universidad·Federico·Santa·María,·Chile¶
B.Sc.·Electronic·Engineering,·December·1986,·Universidad·Federico·Santa·María,·Chile¶
¶
¶
A·Dissertation·Submitted·to·the·Faculty·of·↵
Old·Dominion·University·in·Partial·Fulfillment·of·the↵
Requirement·for·the·Degree·of¶
¶
DOCTOR·OF·PHILOSOPHY¶
¶
COMPUTER·SCIENCE¶
¶
OLD·DOMINION·UNIVERSITY¶
May·2000¶

Fig. 75. Xterm and Odust interface as seen on Eduardo’s machine.

Like Rodrigo, Eduardo also shares an application from his WinNT machine (Fig. 75). Thus, any UNIX application can be shared as well. In contrast to Rodrigo who has the real application, Eduardo sees an image of the MS-Word interface displayed within Odust. With the exception of a hardly observable loss of quality due to lossy JPEG compression, the image in Odust resembles the real application. If other WinNT participants started sharing more applications, Eduardo and Rodrigo would receive them in separate windows within Odust. This is the case of UNIX users in this scenario. They receive Rodrigo’s MS-Word and Eduardo’s xterm in different windows, as illustrated in Fig. 76 and Fig. 77.


Odust: ODU Sharing Tool Multi–Receiver
Shared tool #1
Owner: Rodrigo Leader: Cecilia
Microsoft Word - masterv10.doc
File Edit View Insert Format Tools Table Window Help
Coverpage Times New Roman 12
Page 1 Sec 1 1/153 At 5.3" Ln 21 Col 1 REC TRK EXT OVR WPH
Shared tool #2
Eduardo Leader: Agustin
Folder: phd Message 79 of 108 97%
PrevMsg NextMsg PrevPage NextPage Delete Undelete Reply Forward

A·SEMANTIC-BASED·MIDDLEWARE·FOR·MULTIMEDIA↵

COLLABORATIVE·APPLICATIONS¶

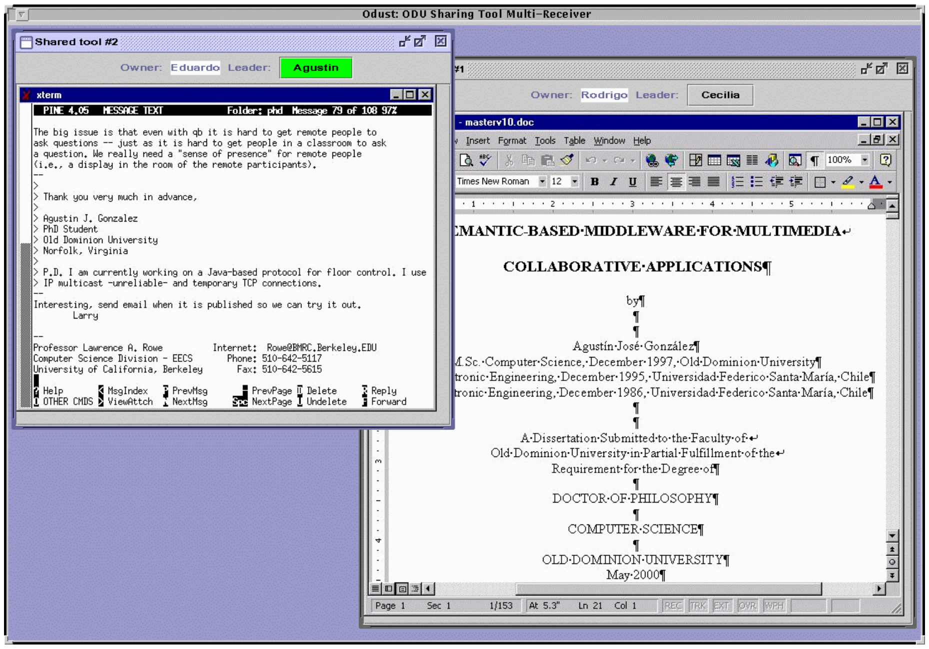
by¶
¶
¶
Agustín·José·González¶
M.Sc.·Computer·Science,·December·1997,·Old·Dominion·University¶
M.Sc.·Electronic·Engineering,·December·1995,·Universidad·Federico·Santa·María,·Chile¶
B.Sc.·Electronic·Engineering,·December·1986,·Universidad·Federico·Santa·María,·Chile¶
¶
¶
A·Dissertation·Submitted·to·the·Faculty·of·↵
Old·Dominion·University·in·Partial·Fulfillment·of·the·↵
Requirement·for·the·Degree·of¶
¶
DOCTOR·OF·PHILOSOPHY¶
¶
COMPUTER·SCIENCE¶
¶
OLD·DOMINION·UNIVERSITY¶
May·2000¶

```
with qb it is hard to get remote people to
 is hard to get people in a classroom to ask
a "sense of presence" for remote people
n of the remote participants).

vance,

ng on a Java-based protocol for floor control. I use
 and temporary TCP connections.

n it is published so we can try it out.

            Internet:  Rowe@BMRC.Berkeley.EDU
 EECS          Phone: 510-642-5117
erkeley          Fax: 510-642-5615
```

Fig. 76. Cecilia’s view of Odust interface on Solaris machine.

Cecilia and any other user in this session receive both shared tools within Odust, as illustrated in Fig. 76. She holds the floor for MS-Word, so she can operate it like its owner, Rodrigo. However, asymmetric operations, such as exiting or minimizing the tool, are irreversible for Cecilia. These operations work in conjunction with the operating system or environment that is not reachable by Odust; for example, one can exit a tool from its interface but normally needs the operating system to start it.

Each tool resides in its own independent widget, so that user can move and minimize them to produce the best view. As new application are shared or exited, new windows dynamically appear or disappear on Odust desktop. The number of shared tools is limited to 256 by design; nonetheless, network and machine resources impose a much lower practical limit with current technology.


Odust: ODU Sharing Tool Multi–Receiver
Shared tool #2
Owner: Eduardo Leader: Agustin
#1
Owner: Rodrigo Leader: Cecilia
xterm
- masterv10.doc
Insert Format Tools Table Window Help
Times New Roman 12
100%
Page 1 Sec 1 1/153 At 5.3" Ln 21 Col 1 REC TRK EXT OVR WPH

```
 PINE 4.05   MESSAGE TEXT              Folder: phd  Message 79 of 108 97%

The big issue is that even with qb it is hard to get remote people to
ask questions -- just as it is hard to get people in a classroom to ask
a question. We really need a "sense of presence" for remote people
(i.e., a display in the room of the remote participants).
--
>
> Thank you very much in advance,
>
> Agustin J. Gonzalez
> PhD Student
> Old Dominion University
> Norfolk, Virginia
>
> P.D. I am currently working on a Java-based protocol for floor control. I use
> IP multicast -unreliable- and temporary TCP connections.
--
Interesting, send email when it is published so we can try it out.
        Larry


--
Professor Lawrence A. Rowe          Internet:  Rowe@BMRC.Berkeley.EDU
Computer Science Division - EECS       Phone: 510-642-5117
University of California, Berkeley       Fax: 510-642-5615

? Help        < MsgIndex   P PrevMsg      - PrevPage D Delete     R Reply
O OTHER CMDS  > ViewAttch  N NextMsg    Spc NextPage U Undelete   F Forward
```

```
EMANTIC-BASED·MIDDLEWARE·FOR·MULTIMEDIA↵
COLLABORATIVE·APPLICATIONS¶
by¶
¶
¶
Agustín·José·González¶
M.Sc.·Computer·Science,·December·1997,·Old·Dominion·University¶
tronic·Engineering,·December·1995,·Universidad·Federico·Santa·María,·Chile¶
tronic·Engineering,·December·1986,·Universidad·Federico·Santa·María,·Chile¶
¶
¶
A·Dissertation·Submitted·to·the·Faculty·of·↵
Old·Dominion·University·in·Partial·Fulfillment·of·the↵
Requirement·for·the·Degree·of¶
¶
DOCTOR·OF·PHILOSOPHY¶
¶
COMPUTER·SCIENCE¶
¶
OLD·DOMINION·UNIVERSITY¶
May·2000¶
```

Fig. 77. Odust interface on Agustín's machine.

Finally, floor control is done on a per shared tool bases. As shown in Fig. 77, Agustín controls the xterm application while Cecilia browses an MS-Word file. This feature enables collaboration at a level it cannot be reached even in face-to-face encounters when two people sit in front of the same computer. We could have this type of views on a single computer screen; nevertheless, we cannot use the computer's keyboard and mouse to simultaneously operate both applications.

7.2 Odust Overall Architecture

Odust's architecture reflects the three main external features of it, application view dissemination, floor control, and remote tool interaction. A distributed object architecture implements the protocol for transmission of dynamic compound images. Then, another set of distributed object implements the lightweight floor control framework for centralized resources. Finally, two application specific objects that work in a client-service architecture support the interaction with the shared application from remote sites. Odust depends on a single multicast group that is provided as command line argument. Indeed, it could also be a unicast address in the case of two-party sessions, but we will assume multi-party sessions in our description. Now, in order to support multiple shared applications at a time, Odust multiplexes the multicast group in up to 256 channels. A distributed multiplexer-demultiplexer object dynamically manages channel allocation as new applications are shared. Each of the basic components of Odust, compound image transmission, floor control, and user's input events is made of two related objects. One centralized object resides on the machine sharing a tool and the others are replicated at every shared tool receiver. The latter object instances are dynamically created and destroyed, so their live time is the same as the shared tool they support. Fig. 78 illustrates a situation where multiple applications are shared. Although a machine that shares a tool can also receive others coming from other sites, we have logically divided Odust in a sender and a receiver component for description purpose.

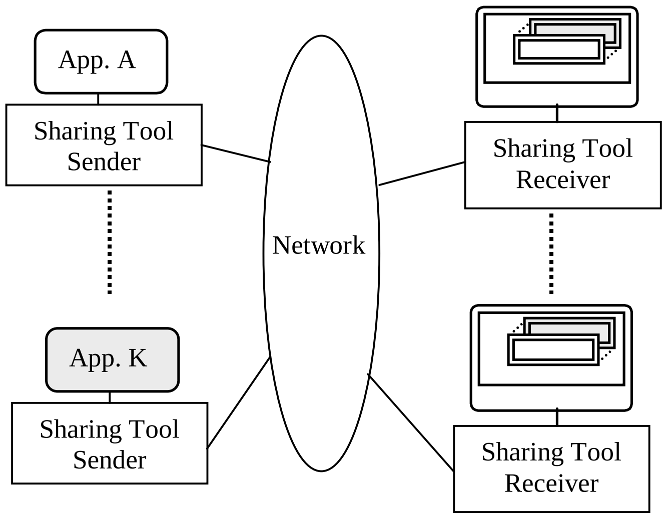


Fig. 78. Odust distributed logic modules.

While Fig. 78 shows the interactions between multiple senders and receivers, Fig. 79 focuses on the internal architecture of one sender and one receiver. All the objects of the sender are instantiated at execution time; however, only the demultiplexer remains up all the time at receiving sites. The demultiplexer listens for messages coming on any channel. Multiplexer (Mx in Fig. 79) and demultiplexer (Dx in Fig. 79) are actually two Java interfaces for the same object. Thus, each multiplexer can keep track of the channel in use and can randomly allocate a new unused channel when the local sender requests one to start transmitting a new shared tool to the session. As soon as its counterparts at each receiver receive an Application data Unit (ADU) from an unallocated channel, each sharing tool receiver creates new *application receiver* object to process subsequent ADUs.

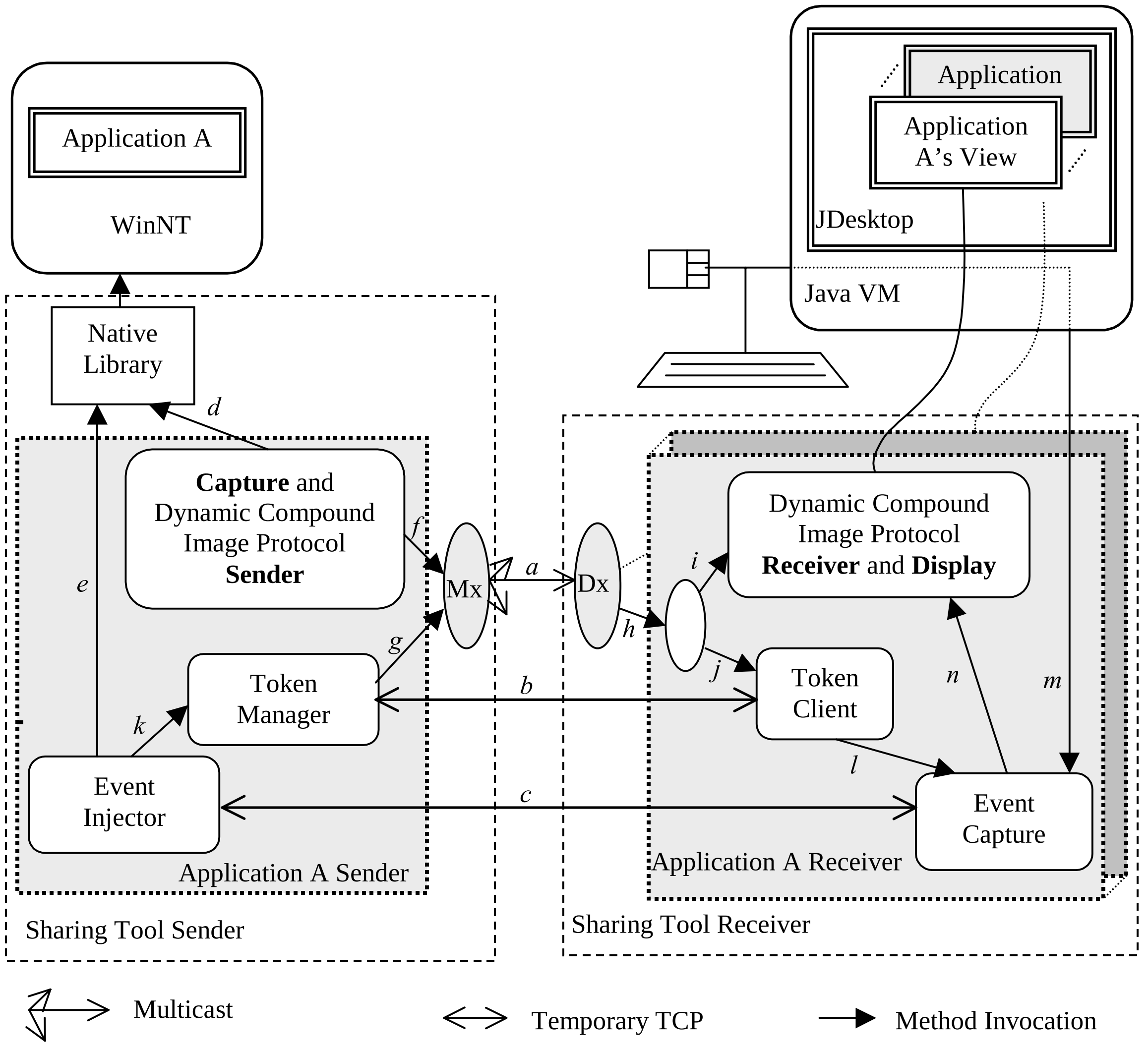


Fig. 79. Odust sender/receiver overall architecture.

Senders blindly transmit ADUs with no feedback from recipients. Both the image transmission and Token Manager objects share the same multicast channel. While the former transmits image protocol related messages, the latter periodically sends a heartbeat with the floor status, mainly floor holder, local host names, and Token Manager service port, so that clients can dynamically connect to the Token Manager (link *b* in Fig.

79). The native library implements three functions required by the Java capture object. These functions are:

1. *int stGetProcessesTitlesAndIds(String[] titleList, long[] pidList)*: It retrieves the process identifiers (PID) of the all applications with windows on the screen. A process title is defined to be the concatenation of all the window titles of that application. The function returns the size of the list. The sender interface displays this information for the user to select the process to be shared.
2. *int stGetAppWinHandles(long PID, long[] winIDList, int[] rasterSizeList)*: It retrieves the list of windows and window's size for the process whose identifier is PID. It returns the number of relevant windows of that process. Relevant windows are not completely contained within another window of the same application. This function lets Odust's capture module prepare and check the number and size of the image pixel buffers. In addition, by comparing the current list of window handler against the previous one, the image transmission object detects removed images.
3. *int stCapAppWin(long winID, byte[] pixels, int[] imageInfo)*: This function finally captures the image and stores the pixels in a Java buffer. In addition, it retrieves the position and dimension of the image, which is conveyed by *imageInfo*. The entries correspond to [0]: x position, [1]: y position, [2]: width, and [3] height. The position (0,0) is the upper left corner of the screen. The return value is the number of lines successfully captured. If the pixel buffer is not big enough fewer image lines are captured.

All the images of the shared application are sampled and transmitted using the protocol for compound image transmission described in Section 7.3. The application image is sampled by capturing screen areas. A shortcoming of this approach is the capture of any other windows that overlay the area defined by the shared application window. This issue is resolved by having the owner of the tool keep the shared application on top of others. At the receiving site, the demultiplexer dispatches the ADUs to the corresponding *application receiver* according to the setting it saves when the *application receiver* is created upon receiving the first ADU (method call *h* in Fig.

79). Then, the *application receiver* dispatches the message to either the compound image receiver (method call *i*) or the Token Client (method call *j*).

The Token Manager and Token Client have graphics interfaces, as shown in Fig. 75 and Fig. 76 respectively. Upon floor request, the Token Client connects to the manager and obtains the service access point of the Event Injector in the Grant message (*b* connection in Fig. 79). The Token Client forwards this information to the Event Capture object (method call *l*) and updates its interface. Finally, connection *c* is established and the mouse and keyboard events of the new floor holder are sent to the application sender. The Event Injector employs the following two calls to insert remote input events into the shared application:

4. *int stKeybdEvent(long winID, byte virtualKeyCode, int functionOption)*: This function was designed to match with the WinNT/Win95 *keybd_event(virtualKeyCode, 0, functionOption, 0)*. Before calling the MS Windows function the key code must be translated from Java key codes to the local platform key codes. The function returns a positive value in case of success.
5. *int stMouseEvent(long winID, int functionOption, int absXpos, int absYpos)*: This fucntion was designed to match with the WinNT/Win95 *mouse_event(functionOption | MOUSEEVENT_ABSOLUTE, absXpos, absYpos, 0, 0)*. Odust uses its own function options that make more sense within Java. They need to be translated to the MS Windows option codes.

Connections *b* and *c* are only kept while the corresponding receiver holds the floor. The Event Capture object listens for input events within the application widget at receiving sites (method call *m*). When an input event is fired by the Java virtual machine, Event Capture forwards the event to its peer Event Injector as long as the event took place within one of the shared application images in the widget. This confirmation is done by a call to the compound image receiver object (method call *n*). This check suppresses events that do not fall into any image even though they are detected within the display widget. The compound image receiver detects when all the windows of the application are destroyed or no tile refresh has taken place after a timeout. It releases all the allocated resources by unbinding the *application receiver* from the channel demultiplexer and locally removing any graphics object for that application.

The Native Library is the only non-Java code. It implements 5 native methods that need to be ported to other platforms in order to share applications running on them. If remote user interaction is not critical, such as in large-scale multicast on the Mbone, only the first three methods are required for a user to transmit her application's view.

Even though the traffic due to the floor holder only affects two machines per floor in the session, we use mouse event filtering to reduce the number of events fired by mouse moves. Mouse movements are only sent to the application if they are far apart in position or time. Two parameters govern the granularity of the filter.

Odust only supports sharing of a single application per machine at a time. Yet, remote user's events interfere with the application owner's input events. Current abstractions for computers' display establish a one-to-one relationship between display, mouse, and keyboard. This limits this approach for collaboration since we cannot smoothly associate two (or even more) mice and keyboards to one display.

An alternative approach to steady image sampling is to capture the view on the screen only after an input event has been sent to the application. In VNC [61], for example, the update protocol is demand-driven by receivers; i.e. an updates is only sent by the sender in response to an explicit request from a receiver. We decided against it because in today's applications the state of the display changes for many reasons other than the user input interactions. Some examples are clock displays, dynamic webpages, and graphic simulations. Furthermore, applications' response time varies, and it is unpredictable in general case. While a local editor takes a fraction of a second to echo our keystrokes, a telnet session or an Internet webpage request might take several seconds. Another approach is to monitor application's events that produce a change on the display. We also decided against it because of the difficulties in implementing a native method to detect such conditions in every platform.

7.3 Extension of the Dynamic Image Transmission Protocol

Sharing the window images of an application cannot be simply implemented by transmitting multiple images using the protocol presented in Chapter VI, although it can be easily extended to accommodate new requirements. The relative positions of the windows must be preserved, and they might be overlapping each other. As a result, besides image dimension, the position of each image must be sent as well in each tile data unit. In addition, the protocol for sending images needs some modifications to reduce processing and traffic in overlapped regions. Even though the latter is not as crucial as the transmission of the application windows' layout on the screen, it is an import performance enhancement when the shared application spawns multiple windows. For instance, it is the case of *gnuplot*.

The problem of partitioning a rectilinear polygon into a minimum number of non-overlapping rectangles appears in many applications besides our imaging application. These include two-dimensional data organization [39], optimal automated VLSI mask fabrication [50], and image compression [47]. The problem is illustrated in Fig. 80. In our application, a simple and straightforward approach would capture and transmit each window. The result is that the overlapped regions (in dark) would be processed twice. In general, some areas could be computed as many as the total number of windows when some pixels intersect every window.

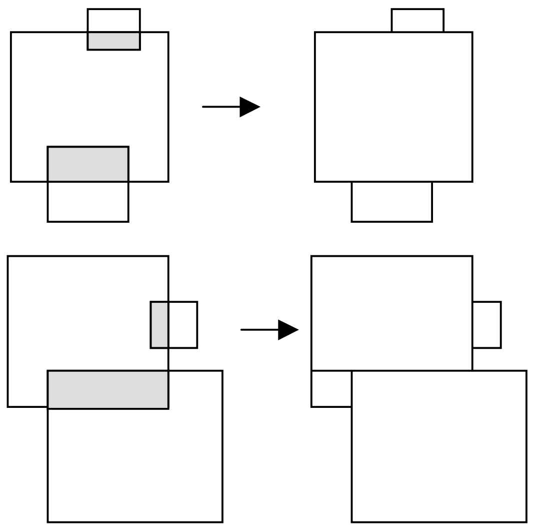


Fig. 80. Overlapping regions in Compound Images.

The minimum partitioning problem was optimally solved in [39] and [50]. Ohtsuki's algorithm runs in $O(n^{5/2})$ time in the worst case. Later, in [30] Imai and Asano proposed an algorithm that requires $O(n^{3/2}\log n)$ time. Liou *et al.* proposed in [38] an optimal

$O(n \log \log n)$-time algorithm for partitioning rectilinear polygon without holes. Despite the optimality of the previous algorithms, their complexity has precluded their usage in applications that require fast encoding operations [47]. In practice, a simple and fast sub-optimal algorithm might be more valuable than a complex optimal solution.

We opted for a sub-optimal solution that could be easily integrated with the tiling technique for image transmission. Our algorithm progressively receives the rectangles being transmitted and returns for each tile the already sent rectangle that fully contains it, as shown in Fig. 81. One advantage of this scheme is its easy integration with the straightforward approach for compound image transmission described above.

```
Initial Condition:
        $\boldsymbol{R} = \phi$;                // set of already sent rectangles.
Before transmission of tile $\boldsymbol{x}$:
        for each rectangle $\boldsymbol{r}$ in $\boldsymbol{R}$:
                if ( $\boldsymbol{x}$ is fully contained in $\boldsymbol{r}$ )
                        return $\boldsymbol{r}$;
        return null;
After transmission of image within rectangle $\boldsymbol{r}$:
        $\boldsymbol{R} = \boldsymbol{R} \cup \{\boldsymbol{r}\}$;
```

Fig. 81. Algorithm to suppress overlapped region retransmission.

The protocol for sending dynamic images is slightly changed by integrating the algorithm for overlapping suppression. If a tile is already at the receiving site, a copy message is transmitted for the receiver to take the tile from the already received image. Obviously the algorithm is not optimal for tiles bigger than 1x1 pixel since a tile that partially falls into an already sent rectangle is transmitted anyway. In addition, tiles that span across a number of sent rectangles but none of then fully contains them are also sent. Due to the fact that most commonly used applications spawn only one window, we have deferred the implementation of this refinement for later versions of Odust.

CHAPTER VIII

CONCLUSIONS AND FUTURE WORK

It is a fact of life that once one has completed a study, inevitably new questions and ideas come up. New goals are set. Rather than a summit, we have just reached a plateau from where new peaks are discovered and others are much clearer. This research under no mean is an exception to it. Below, we summarize our more relevant conclusions and briefly describe some future extensions for this research work.

8.1 Conclusions

The Internet has been expanding in size and increasing in bandwidth since its creation more than 30 years ago. Likewise, the performance of personal workstations has tremendously increased in the last fifteen years. As a result of these changes, the applications that depend on these technologies have also evolved from text-only applications to the current high-bandwidth large-scale real-time multimedia applications. The demand for the latter ones is expected to grow and their traffic to become the dominant Internet traffic. Nevertheless, besides bandwidth and processing power, this emerging type of applications also demands timely information delivery and scalability. The first one is an intrinsic requirement of continuos media that becomes even more stringent in synchronous or interactive collaboration. Scalability is an issue in large-scale distributed applications that involve thousands or even millions of users. Real-time and scalability are two new requirements that had not been faced by massive non-specialized-user applications before and, therefore, have been poorly supported by the Internet and traditional operating systems. Internet bandwidth and hardware resources are easy to deploy; however, the deployment of new Internet protocols has lagged. The two traditional Internet transport protocols, TCP and UDP, do not support real-time delivery nor do they scale. Then, the introduction of multicasting enabled large-scale application in the Internet. On the other hand, traditional operating systems lack real-time services. In order to provide scalability and real time services for multimedia applications, a considerable amount of research work has been dedicated to new computer network protocols, new structures and abstractions in operating systems, and multimedia middleware.

In this thesis we proposed a semantic-based multimedia middleware. It aims to the encapsulation of refined solutions to common needs in developing large-scale multimedia applications. It follows an object-oriented design and was implemented in Java. It is reusable, extensible, flexible, and scalable. It supports four frequently used services, floor control, stream synchronization, extended network services, and dynamic image transmission.

We proposed two scalable and lightweight protocols for floor control. One is based on a centralized architecture that easily integrates with centralized resources such as a shared tool. Its simplicity provides high reliability and efficiency. The other is a distributed protocol targeted to distributed resources. It basically implements an extension of the first protocol by moving the central coordinator along with the floor. It also includes a recovery mechanism to overcome coordinator crashes. Scalability is achieved by having the coordinator periodically multicast a heartbeat that conveys enough state information for the clients to know the identity of the floor holder and the coordination service point. Clients establish temporary TCP connections with the coordinator to request the floor.

Today's Internet best-effort service introduces unavoidable uncertainties in the data transfer delay that creates the need of stream synchronization mechanisms. In order to preserve the temporal relationship among streams, we presented algorithms that are immune to clock offset between sender and receivers and take into account the different time constraints of each media. Their time model includes delays outside the computer and network boundary. We introduced the concept of *virtual observer*, which perceives the session as being in the same room with a sender. Intra-stream synchronization is achieved by adjusting a sender-to-receiver latency delay for each data unit. The latency is dynamically adapted to control a given percentage of late packets. Specific media temporal requirements are fulfilled through a number of *playout policies*. The proposed policies for late arrivals are *packet discard*, *resynchronization*, and *late delivery*. In order to adjust latency delay, we proposed *early delivery* and *oldest packet discard* for reducing delay latency and *gap insertion* for increasing it. The algorithm works in two modes.

The initial mode, which is crucial in interactive applications, rapidly reaches steady state. In the second mode, the algorithm smoothly adapts to delay changes. We avoided the need for globally synchronized clocks for media synchronization by using a per user model for inter-stream synchronization. We referred to it as the *user's multimedia presence*. We also proposed a novel algorithm for on-line estimation and removal of clock skew. It is based on the same timing information already available for media synchronization.

We also enhanced traditional network API by supporting event-driven asynchronous message reception, quality of service measures, and traffic rate control. Asynchronous reception was achieved by embedding a thread in an extension of the Java socket class and having a higher level object register itself as listener. In addition, in each socket we measure input and output accumulated traffic and traffic rate. The output rate can also be controlled. Delaying data unit transmission keeps the traffic in a moving window below a threshold. We also addressed the loss of performance due to multiple copies or data moves while application data units pass across software layers. We proposed objects to encapsulate the needs of buffering in transmission and reception.

Along with audio and video, data sharing is a crucial component in multimedia collaboration. In the middleware, we included support for data sharing via a protocol for image transmission. These images can change in size and content. This resilient and scalable protocol compresses a sequence of image samples by removing temporal and spatial redundancy. Tiling and changes detection achieve the former, and a standard image compression technique accomplishes spatial redundancy removal. Protocol data unit losses are overcome by randomly re-transmitting tiles. This technique also provides support for latecomers. We did an extensive study on the sensitivity of the dominant parameters of the protocol. These included tile compression format, tile size, sampling rate, and tile change detection technique.

Finally, we verified the effectiveness of the midleware with the implementation of Odust. This sharing tool application disseminates images of the shared application and accepts remote user input events as if they were coming from the local tool owner. In the design and implementation of this application, we tested the extensibility of the middleware, its modularity, and scalability. This application made intensive use of the floor control framework, network services, and an extension of the protocol for image transmission to achieve *compound image* transmission. In addition, the reusability of the middleware was demonstrated with the easy integration of Odust with a new develop of IRI based on Java. The middleware was tested on Win85, Win98, WinNT, and Solaris operating systems. The middleware met the expectations in terms of flexibility, extensibility, scalability, and heterogeneity. Finally, having the middleware components available greatly simplified the design and implementation of this sharing tool engine. Future extension of the middleware to include a framework for shared tele-pointer and annotation will enhance these time-constrained large-scale multimedia applications even further.

8.2 Future Work

The current version of the middleware can be extended in two ways, by improvements and enhancements of the already existing components, and by adding new reusable components. Below we summarize some extensions for each of the middleware modules and suggest new components to be integrated.

Floor Control: In the current version, the floor can be held by at most one client. A more general model is to allow up to N clients to access the shared resource. Audio can benefit from this service especially in small-scale highly interactive sessions. Rather than switching the floor back and forth, a number of users can be allowed to have the audio floor simultaneously. This type of control also has applications in video in order to limit the bandwidth allocated to the aggregation of video streams.

Stream Synchronization: An enhancement for the middleware is the integration of the synchronization and clocks skew removal algorithms. Another specific problem is audio playout. Our algorithm ensures audio samples are delivered to the output device in synchrony with their capture; however, the capture clock might differ from the playback clock. This causes either accumulation or starvation of audio samples in the output device. Audio sample starvation might not be noticeable if not frequent; however, samples accumulation need to be detected and corrected.

Network Services: The current services unify multicast and point-to-point communications. An extension is to provide group communication by using *application* *layer multicasting.* Here, an application module is responsible of transmitting copies of the data unit to each member of the group. It is quite useful when the network does not support multicast. Extensions could use a centralized or distributed architecture. In the former, the current framework could connect to the central server, which forwards copies to all the clients already connected. The only extension to the middleware would be the central relaying server. Another approach is to have each sender transmit a copy of the message to the other participants. For a three-party session the latter technique has a cost of 2 messages per transmission while the former scheme uses 3. For more than three participants, the first scheme is better. Perhaps an adaptive group communication service may dynamically detect the number of users and use the less expensive communication approach.

Dynamic Image Transmission Protocol: In addition to the extension already suggested in Section 6.7, we foresee the need of a gateway to accommodate bandwidth heterogeneity. In contrast to the other components of the middleware, this protocol consumes an amount of bandwidth that is not available on all the segments connected to the Internet. Basically, the gateway has to change the current tradeoff between processing and bandwidth to meet the traffic requirement of the outgoing network.

New Middleware Services: Obviously our middleware can be further extended for many more services. Next, we describe some of them. *Pointing and annotation facilities* are also common needs in interactive applications. We suggest a unified component for both services. The protocol for data distribution needs to take into consideration some differences in semantic though. For example, while pointing, intermediate positions are not that critical; however, in drawing resiliency is important. Moreover, latecomers do not need to receive old pointer positions, but they do expect to see any drawing or annotation up-to-date. *Encryption* can also be provided by the middleware. Audio filters may be included to support *audio mixing* and *audio silence detection*. For video, the middleware could encapsulate protocols for *multi-layer transmission*. *Large-group feedback* is another important and often needed service in multimedia application. Some algorithms have already been published [49] and could be encapsulated in the middleware. Support for recording and playback could also be integrated into the middleware.

Finally, we have established the first version of this middleware and shown its usefulness as developing infrastructure. We now expect it will evolve as the natural result of its use in current challenging applications and new ones to come.

REFERENCES

[1] H. Abdel-Wahab and M. Feit, “XTV: A Framework for Sharing X Window Clients in Remote Synchronous Collaboration,” in *IEEE Tricomm '91: Communication for Distributed Applications & Systems*, Chapel Hill, NC, USA, 1991. IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 157-167, 1991.

[2] H. Abdel-Wahab, O. Kim, P. Kabore, and J.P. Favreau, “Java-based Multimedia Collaboration and Application Sharing Environment,” in Proceedings of the Colloque Francophone sur I’Ingenierie des Protocoles (CFIP ’99), Nancy, France, April 26-29, 1999.

[3] H. Abdel-Wahab, A. Youssef, and K. Maly, “Distributed management of exclusive resources in collaborative multimedia systems,” *Proceedings Third IEEE Symposium on Computers and Communications. ISCC'98*. IEEE Comput. Soc, Los Alamitos, CA, USA, pp.115-19, 1998.

[4] N. Agarwal and S. Son, “Synchronization of distributed multimedia data in an application-specific manner,” in *2nd ACM International Conference on Multimedia*, San Francisco, California, pp. 141-148, 1994.

[5] D. Agrawal and A. El Abbadi, “An Efficient and Fault-Tolerance Solution for Distributed Mutual Exclusion,” *ACM Transactions on Computer Systems*, vol. 9 no. 1, pp. 1-20, February 1991.

[6] K. Almeroth and J. Nonnenmacher, CALL FOR PAPERS for Special Issue of Computer Communications on Integrating Multicast into the Internet to be published in Fall 2000. Message posted in rem-conf@es.net mailing list on Dec. 9, 1999.

[7] D. Anderson, “Metascheduling for Continuous Media,” *ACM Transactions on Computer Systems*, vol. 11, no. 3, pp. 226-252, 1993.

[8] C.Bisdikian, S. Brady, Y.N. Doganata, D.A. Foulger, F. Marconcini, M. Mourad, H.L. Operowsky, G. Pacifici, and A.N. Tantawi, “Multimedia Digital Conferencing: A Web-enabled multimedia Teleconferencing system,” *IBM Journal of Research and Development*, vol. 42, no.2, pp. 281-298, March 1998.

[9] J. Bolot, “End-to-End Packet Delay and Loss Behavior in the Internet,” in *SIGCOMM 1993*, Ithaca, New York, USA, pp. 289-298, September 1993.

[10] J. Bolot and A. Garcia, "Control mechanisms for packet audio in the internet," in *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, San Francisco, California, Mar. 1996.

[11] T. Boutell, “PNG (Portable Network Graphics) Specification: Version 1.0,” Request for Comments RFC 2083, January 1997.

[12] R. Branden, L. Zhang, S. Berson, S. Herzog, and S. Jamin, “Resource ReSerVation Protocol (RSVP)- Version 1 Functional Specification,” Request for Comments (RFC), 2205, September 1997.

[13] J. Charles, “Middleware Moves to the Forefront,” *IEEE Computer*, pp. 17-19, May 1999

[14] D.D., Clark and D. Tennenhouse, “Architectural considerations for a new generation of protocols,” in *SIGCOMM Symposium on Communications Architectures and Protocols*, Philadelphia, Pennsylvania, IEEE, pp. 200-208, Sept. 1990.

[15] D.D. Clark, S. Shenker, and L. Zhang, “Supporting real-time applications in an integrated services packet network: Architecture and mechanism,” *SIGCOMM ’92 Communications Architectures and Protocols*, pp.14-26, 1992.

[16] G. Côté, B. Erol, M. Gallant, and F. Kossentini, “H.263+: Video Coding al Low Bit Rate,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 8, no. 7, pp. 849-866, November 1998.

[17] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems: Concepts and Design*, 2nd. Edition, Eddison-Wesley, 1994.

[18] R. Cruz, “A Calculus for Network Delay, Part I: Network Elements in Isolation,” *IEEE Trans. Information Theory*, vol. 37, no. 1, pp. 114-121, 1991.

[19] J.Z. Davis, K. Maly, and M. Zubair, "A Coordinated Browsing System”, *Technical Report TR-97-29*, Old Domimion University, Norfolk, VA, May 1997.

[20] H.P. Dommel and J.J. Garcia-Luna-Aceves, “Group coordination support for synchronous Internet collaboration*,” IEEE-Internet-Computing*, vol.3, no.2, pp. 74-80, March-April 1999.

[21] H.P. Dommel and J.J. Garcia-Luna-Aceves, “Floor control for multimedia conferencing and collaboration,” *Multimedia System*, vol.5, no.1, pp. 23-38, Jan. 1997.

[22] H.P. Dommel and J.J. Garcia-Luna-Aceves, “Network Support for Turn-Taking in Multimedia Collaboration,” *Proceedings of the IS&T/SPIE Symposium on Electronic Imaging: Multimedia Computing and Networking 1997,* San Jose, CA, pp. 304-315, February 1997.

[23] J. Escobar, C. Partridge, and D. Deutsch “Flow Synchronization Protocol,” *IEEE/ACM Transactions on Networking*, vol. 2, no. 2, pp. 111-121, April 1994.

[24] D. Ferrari, “Client requirements for real-time communication services,” RFC (Request for Comments) 1193, 1990.

[25] D. Ferrari, “A New Admission Control Method for Real-Time Communication in an Internetwork,” in *Advances in Real-Time Systems*, Editor Sang H. Song, Prentice Hall, pp.105-116, 1995.

[26] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. Zhang, “A reliable multicast framework for lightweight sessions and application-level framing”, in *Proceedings of SIGCOMM 1995*, Cambridge, MA, pp. 342-356, August 1995. Also in *IEEE/ACM Transactions on Networking*, vol. 5, no.6, pp.784-803, December 1997.

[27] HABANERO, On-line from: http://www.ncsa.uiuc.edu/SDG/Software/Habanero.

[28] R. Hogg and A. Craig, *Introduction to Mathematical Statictics*, 3rd Edition, Macmillan Publishing, 1970.

[29] HyperText Markup Language Reference Specification, W3C Recommendation, http://www.w3c.org/MarkUp

[30] H. Imai and T. Asano, "Efficient algorithm for geometric graph search problems," *SIAM Journal on Computing*, vol. 15, pp. 478-494, 1986.

[31] ITU Telecommunication Standardization sector of ITU, "Video codec for audiovisual services at p x 64 kbit/s," *ITU-R Recommendation H.261*, March 1993.

[32] V. Jacobson and S. McCanne, "*vat* (Visual Audio Tool) Unix Manual Pages," Lawrence Berkeley Laboratory. Berkeley, California, USA; Software on-line at ftp://ftp.ee.lbl.gov/conferencing/vat.

[33] T. Jurga, Master Thesis Paper of the Computer Science Department of the Graduate School of Syracuse University, 1997. On-line from: http://www.npac.syr.edu/tango/

[34] M.F. Kaashoek, D.E. Engler, G.G. Ganger, H.M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. "Application Performance and Flexibility on Exokernel Systems," in *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP)*, pp. 52-65, 1997.

[35] G. Le Lann, "Distributed Systems – Toward a Formal Approach," *Proceedings of the IFIP Congress 77*, pp. 155-160, 1977.

[36] L. Lamport, "Time, Clocks, and and the Ordering of Events in a Distributed System," *Communication of the ACM*, Vol. 21, No. 7, pp. 558-565, July 1978.

[37] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden, "The Design and Implementation of an Operating System to Support Distributed Multimedia Applications," *IEEE Journal on Selected Areas in Communications*, vol. 14, no. 7, pp. 1280-1297, September 1996.

[38] W.T. Liou, J.J. Tan, and R.C.T. Lee, "Minimum Rectangular Partition Problem for Simple Rectilinear Polygons," *IEEE Transaction on Computer-Aided Design*, vol. 9 no. 7, pp. 720-733, 1990.

[39] W. Lipski, E. Lodi, F. Luccio, C. Mugnai, and L. Pagni, "On two-dimensional data organization II," *Fundamenta Informaticae*, vol.2, no. 3, pp. 245-260, 1979.

[40] R. Malpani and L.A. Rowe, "Floor Control for Large-Scale MBone Seminars," *Proceedings of The Fifth Annual ACM International Multimedia Conference*, Seattle, WA, pp 155-163, November 1997.

[41] K. Maly, H. Abdel-Wahab, C.M. Overstreet, C. Wild, A. Gupta, A. Youssef, E. Stoica, and E. Al-Shaer, "Distant Learning and Training over Intranets," *IEEE Internet Computing*, vol. 1, no.1, pp. 60-71, 1997.

[42] MBone http://www.mbone.com/.

[43] S. McCanne and V. Jacobson, "*vic*: A Flexible Framework for Packet Video," *ACM Multimedia* 1995.

[44] S. McCanne, E. Brewer, R. Katz, L. Rowe, E. Amir, Y. Chawathe, A. Copersmith *et al.*, "Toward a common Infrastructure for Multimedia-Networking Middleware," In Proceedings of the Fifth International Workshop on Network and OS Support for Digital Audio and Video (NOSSDAV), St. Luis Missouri, May 1997.

[45] Microsoft's NetMeeting,http://www.microsoft.com/netmeeting.

[46] Microsoft's PowerPoint,http://www.microsoft.com/powerpoint.

[47] S.A. Mohamed and M.M. Fahmy, "Binary Image Compression Using Efficient Partitioning into Rectangular Regions," *IEEE Transactions on Communications*, vol. 43, no. 5, pp. 1888-1893, May 1995.

[48] S. Moon, P. Skelly, and D. Towsley, "Estimation and Removal of Clock Skew from Network Delay Measurements," in *Proceedings of 1999 IEEE INFOCOM*, New York, NY, March 1999

[49] J. Nonnenmacher and E.W. Biersack, "Scalable Feedback for Large Groups," *IEEE/ACM Transactions on Networking*, vol. 7 no. 3, June 1999.

[50] T. Ohtsuki, "Minimum dissection of rectilinear regions," In Proceedings *IEEE International Symposium on Circuits and Systems*, New York, USA, vol. 3, pp. 1210-1213, 1982.

[51] A. Oppenheim and R.Schafer, *Discrete-Time Signal Processing*, Prentice Hall, New Jersey, 1989.

[52] V. Paxson, "On Calibrating Measurements of Packet Transit Times," in *Proceddings of SIGMETRICS '98*, Madison, Wisconsin, pp. 11-21, June 1998.

[53] C. Perkins and O. Hodson, "Options for repair of streaming media," *Request for Comments* (Informational) RFC 2354, Internet Engineering Task Force, June 1998.

[54] C. Perkins, O. Hodson, and V. Hardman, "A Survey of Packet-Loss Recovery Techiques for Streaming Audio," IEEE Network Magazine, September/October 1998.

[55] J. Postel, "Transmission Control Protocol", STD 7, RFC 793, September 1981.

[56] S. Ramanathan and P.V. Rangan, "Continuous media synchronization in distributed multimedia systems," in *3rd International Workshop on Network and Operating System Support for Digital Audio and Video*, pp. 289-296, 1992.

[57] R. Ramjee, J. Kurose, D. Towsley, and H. Schulzrinne, "Adaptive Playout Mechanism for Packetized Audio Applications in Wide-Area Networks," in *IEEE INFOCOM '94*, Montreal, Canada, 1994.

[58] P. Rangan, H. Vin, and S. Ramanathan, "Communication Architecture and Algortihms for Media Mixing in Multimedia Conferences," *IEEE/ACM Trans. Networking,* vol. 1, no. 1, pp. 20-30, February 1993.

[59] G. Ricard and A. Agrawala, “An optimal Algortihm for Mutual Exclusion in Computer Networks,” *Communications of the ACM*, vol. 24, no. 1, pp. 9-17, January 1981.

[60] G. Ricart and A. Agrawala, Authors’ response to letter On Mutual Exclusion in Computer Nextworks. *Technical Correspondence, Communication of ACM*, vol. 26, no. 2, pp. 146-148, February 1983.

[61] T. Richardson, Q. Stafford-Fraser, K. Wood, and A. Hopper, “Virtual Network Computing,” IEEE Internet Computing, vol. 2, no.1, pp. 33-38, Jan/Feb 1998.

[62] K. Rothermel and T. Helbig, “An Adaptive Protocol for Synchronizing Media Streams,” *Multimedia Systems*, ACM/Springer, vol. 5, pp. 324-336, 1997.

[63] I. Schubert, D. Sisalem, and H. Schulzrinne, “A Session Floor Control Scheme,” *Proceedings of International Conference on Telecommunications*, Chalkidiki, Greece, pp. 130-134, June 1998.

[64] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, “RTP: A Transport Portocol for Real-Time Applications,” RFC (Request for Comments) 1889, January 1996.

[65] H. Schulzrinne, “Voice Communication Across the Internet: a Network Voice Terminal,” Technical Report, Depto. Of Computer Scsience, U. Massachusetts, Amherst, MA, July 1992.

[66] H. Schulzrinne, “RTP Tools,” URL:

ftp://ftp.cs.columbia.edu/pub/schulzrinne/rtptools/rtptools.html

[67] D. Sisalem, H. Schulzrinne, and C. Sieckmeyer, “The Network Video Terminal,” in *HPDC Focus Workshop on Multimedia and Collaborative Environments, Fifth IEEE International Symposium on High Performance Distributed Computing*, Syracuse, New York, IEEE Computer Society, Aug. 1996.

[68] R. Steinmetz and C. Engler, “Human Perception of Media Synchronization,” Technical Report 43.9310, IBM European Networking Center Heidelberg, Heidelberg, Germany, 1993.

[69] R. Steinmetz and K. Nahrstedt, *Multimedia: Computing, Communication and Applications*, Prentice Hall, 1995.

[70] D. Stone and K. Jeffay, “An Empirical Study of Delay Jitter Management Policies,” *Multimedia Systems*, ACM/Springer, vol. 2, no. 6, pp. 267-279, January 1995.

[71] T. Tung, “MediaBoard: A Shared Whiteboard Application for the MBone,” Master’s Report at the Computer Science Department of the University of California, Berkeley, January 1998.

[72] Sun Microsystems, Java language, http://java.sun.com/products.

[73] Sun Microsystems Java Remote method Invocation: Java RMI. Documentation on line from: http://java.sun.com/products//jdk/1.1/docs/guide/rmi/index.html

[74] A. Tanenbaum, *Distributed Operating Systems*, Prentice Hall, New Jersey, 1995.

[75] Videoconferencing Tool, The Networked Multimedia Research Group at University College London,

http://www-mice.cs.ucl.ac.uk/multimedia/software/vic/.

[76] User Guide for VIC v2.8 Version 1 (DRAFT), University College London, Computer Science Department, September 29, 1998. http://www-mice.cs.ucl.ac.uk/multimedia/software/vic/documentation/vic-userguide.zip

[77] G.K. Wallace, "The JPEG Still Picture Compression Standard," *Communications of the ACM*, vol. 34, no.4, pp. 30-44, April 1991.

[78] Y. Xie, C. Liu, M. Lee, and T. Saadawi, "Adaptive multimedia synchronization in a teleconference system," *Multimedia Systems*, ACM/Springer, vol. 7, pp. 326-337, 1999.

[79] A. Youssef, "A Framework for Controlling Quality of Session in Multimedia Systems," Old Dominion University, Norfolk, VA, Ph.D. dissertation, December 1998.

[80] R. H Zakon, "Hobbes' Internet Timeline," http://info.isoc.org/guest/zakon/Internet/History/hit.html.

APPENDIX A

SLOPE ESTIMATE

Let $f(x)$ be a continuous function whose slope we want to estimate, and let $y(x)$ its slope estimate. In order to follow the slope changes, the estimate $y(x)$ should go up or down depending on whether it is below or above $\partial f(x)/\partial x$. Thus, we establish the following condition:

$$\frac{\partial y}{\partial x} = k\left(\frac{\partial f}{\partial x} - y\right)$$

$$\frac{1}{k}\frac{\partial y}{\partial x} + y = \frac{\partial f}{\partial x}$$

As approximation for discrete case, we use:

$$\frac{1}{k}\left(\frac{y(x) - y(x-\varepsilon)}{\varepsilon}\right) + y(x) = \frac{f(x) - f(x-\varepsilon)}{\varepsilon}$$

Or:

$$y(x) = \frac{1}{1-k\varepsilon}\, y(x) + \frac{k}{1-k\varepsilon}\left(f(x+\varepsilon) - f(x)\right)$$

When x is a natural number; i.e. $x \in N_0$, the minimum value for ε is 1; Hence:

$$y_x = \frac{1}{1-k}\, y_{x-1} + \frac{k}{1-k}(f_x - f_{x-1})$$

By defining $\alpha \equiv \dfrac{1}{1-k}$,

$$y_x = \alpha\, y_{x-1} + (1-\alpha)(f_x - f_{x-1})$$

The stability analysis of this estimate using z-Transform [51] is as follows:

$$Y(z) = \alpha\, z^{-1} Y(z) + (1-\alpha)\left(F(z) - z^{-1}F(z)\right), \qquad \text{then}$$

$$\frac{Y(z)}{F(z)} = \frac{(1-\alpha)\left(1 - z^{-1}\right)}{1 - \alpha\, z^{-1}} = \frac{(1-\alpha)(z-1)}{z-\alpha}$$

The stability, or region of convergence, is established by the values that make the denominator zero (also called poles), this is $z = \alpha$, and the condition it must hold is $|z| = |\alpha| < 1$.

APPENDIX B

JAVA MULTICAST SOCKET CLASS EXTENSION

```
/*
  This class extends the Java MulticastSocket class services
  in order to include traffic statistics and control
  and support for event driven model.
 */

import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.SocketException;
import java.net.InetAddress;
import java.io.IOException;
import java.net.MulticastSocket;

public class smmExtendedMulticastSocket extends MulticastSocket
  implements Runnable {

  // Default number of packet taken into consideration
  // to compute statistics.
  private final int DefaultTrafficHistory = 4;

  private smmOnReceiveListener onRecvListener;
  private boolean asynchronousMode;
  Thread arrivalThread;

  // Data members for collecting statistics
  protected long startingMeterTime;
  private boolean txRateControlOn;
  private int txRateLimit;   // outgoing traffic rate limit
  protected int totalTxBytes;// total bytes sent since meter is on
  protected int txReqTime;   // last time a send request took place
  protected int totalRxBytes; // total bates received since meter is on
  private boolean meterOn;   // control whether statistic is collected or not
  protected int[] txTime;    // circular buffer for storing tx times
  protected int[] txSize;    // circular buffer for storing tx packet sizes
  protected int txTraffic,  // total tx traffic in rate in controlling window.
                rxTraffic;  // total rx traffic in the monitoring window (history).
  protected int[] rxTime;   // circular buffer for storing rx times.
  protected int[] rxSize;   // circular buffer for storing rx packet sizes.
  protected int txindex, rxindex;  //indexes to travel rx and tx circular buffers.
  protected int history;  // number of packet for short-time monitoring.
  protected int winSize;  // number of packet for rate control processing.
```

```
public smmExtendedMulticastSocket (int port, InetAddress addr, int ttl)
  throws IOException {
  super(port);
  setTimeToLive(ttl);
  if (addr != null)
    if (addr.isMulticastAddress())
        joinGroup(addr);
  onRecvListener = null;
  meterOn = false;
  txRateControlOn = false;
  asynchronousMode = false;
  arrivalThread = null;
  history = DefaultTrafficHistory; // number of packet taken into
                                   // account for computing traffic rate.
}

public smmExtendedMulticastSocket (int port, InetAddress addr,
                                   int ttl, int history)
    throws IOException {
    this(port,addr,ttl);
    this.history = history;
}

public void setOnReceiveListener(smmOnReceiveListener l) {
  onRecvListener = l;
}

public void startMeter () {
  meterOn = true;
  if (txTime==null) {
        txTime = new int[history];
        txSize = new int[history];
        rxTime = new int[history];
        rxSize = new int[history];
  }
  for (int i=0; i<history; i++)
    txTime[i] = txSize[i] = rxTime[i] = rxSize[i] = 0;
  rxindex=txindex=0;
  txTraffic=rxTraffic =0;
  startingMeterTime = System.currentTimeMillis();
  totalTxBytes = 0;
  totalRxBytes = 0;
}

public void stopMeter() {
  meterOn = false;
}

public boolean  isMeterOn() {
  return meterOn;
}

public void enableTxRateControl(boolean state) {
  txRateControlOn = state;
}

public boolean isTxRateControlEnable() {
  return txRateControlOn;
}

public void setTxRateLimit(int rate) {
  txRateLimit = rate;
}

public int getTxRateLimit() {
  return txRateLimit;
}

public int setTxRateWindowSize(int windowSize) {
  if ( windowSize < history)
    winSize = windowSize;
  else
    winSize = history-1;
  return winSize;
}

public int getTxRateWindowSize() {
  return winSize;
}

public void setSynchronousMode() {
  asynchronousMode = false;
}

public void setAsynchronousMode() {
  asynchronousMode = true;
  if (arrivalThread == null) {
    arrivalThread = new Thread(this);
    arrivalThread.start();
  }
}
```

```
public void receive (DatagramPacket p )
  throws IOException {

  super.receive(p);
  if (meterOn) {
    rxindex = (rxindex+1)%history;
    rxTraffic -= rxSize[rxindex];
    // To get time and size in same scale (milli xx)
    rxTraffic += (rxSize[rxindex]=p.getLength()*1000);
    totalRxBytes += rxSize[rxindex];
    rxTime[rxindex] = (int)(System.currentTimeMillis()-
                                    startingMeterTime);
  }
}

public void send(DatagramPacket p, byte ttl)
  throws IOException {

  if (meterOn || txRateControlOn) {
    int index, size, serviceTime;
    index = (txindex+history-winSize)%history;
    txTraffic -= txSize[index];
    // To get time and size in same scale (milli xx)
    txTraffic += (size=p.getLength()*1000);
    txReqTime = (int)(System.currentTimeMillis()-
                              startingMeterTime);
    if (txRateControlOn)
          try {
            serviceTime = txTraffic/txRateLimit + txTime[index];
            // wait to meet traffic rate limit
            if ( serviceTime > txReqTime )
              Thread.currentThread().sleep(serviceTime-txReqTime);
          } catch (InterruptedException e ) {}
    super.send(p, ttl);
    txindex = (txindex+1)%history;
    // To get time and size in same scale (milli xx)
    txSize[txindex] = size;
    txTime[txindex] = (int)(System.currentTimeMillis()-
                                    startingMeterTime);
    totalTxBytes += size;
  }
  else
    super.send(p, ttl);
}

public void send(DatagramPacket p)
```

```
  throws IOException {

  if (meterOn || txRateControlOn) {
    int index, size, serviceTime;
    index = (txindex+history-winSize)%history;
    txTraffic -= txSize[index];
    // To get time and size in same scale (milli xx)
    txTraffic += (size=p.getLength()*1000);
    txReqTime = (int)(System.currentTimeMillis()-
                              startingMeterTime);
    if (txRateControlOn) {
          serviceTime = txTraffic/txRateLimit + txTime[index];
          // wait to meet traffic rate limit
          if ( serviceTime > txReqTime )
            try {
              Thread.currentThread().sleep(serviceTime-txReqTime);
            } catch (InterruptedException e ) {}
    }
    super.send(p);
    txindex = (txindex+1)%history;
    // To get time and size in same scale (milli xx)
    txSize[txindex] = size;
    txTime[txindex] = (int)(System.currentTimeMillis()-
                                    startingMeterTime);
    totalTxBytes += size;
  }
  else
    super.send(p);
}

public void run () {
  while(asynchronousMode) {
    if (onRecvListener != null)
          onRecvListener.smmOnReceive(this);
    else
          try {
            arrivalThread.sleep(100);
          } catch ( InterruptedException e) {}
  }
  arrivalThread = null;
}

// Statistics
public int avgRxTrafficRate() { // in byte/s
  if (meterOn)
    return (int)(totalRxBytes/
                        (System.currentTimeMillis()-startingMeterTime));
    else
      return 0;
  }

  public int avgTxTrafficRate() { // in byte/s
    if (meterOn)
      return (int)(totalTxBytes/
                        (System.currentTimeMillis()-startingMeterTime));
    else
      return 0;
  }

  public int rxSTTR() {  // Tx short-times Traffic Rate in byte/s
    int divisor = (int)(System.currentTimeMillis()-startingMeterTime);
    int index = (rxindex+1)%history;

    if (meterOn)
      if ((divisor -=rxTime[index]) > 0)
            return ((rxTraffic-rxSize[index])/divisor);
      else
            return (rxTraffic-rxSize[index]);
    else
      return 0;
  }

  public int txSTTR() {  // Tx Instantaneous Traffic Rate in byte/s
    int divisor =(int) (System.currentTimeMillis()-startingMeterTime);
    int index = (txindex+1)%history;

    if (meterOn) {
      int total=0;
      for (int i=0; i< history; i++)
            total += txSize[i];
      if ((divisor -= txTime[index]) > 0)
            return ((total-txSize[index])/divisor);
      else
            return (total-txSize[index]);
    }
    else
      return 0;
  }
}
```

APPENDIX C

INPUT AND OUTPUT DATAGRAM PACKET CLASSES

Output Datagram Packet Class

```
/*
  This class extends OnputStream to allow
  programmers to reset the output stream based on a
  output packet array so that the same object can be reused
  for multiple transmissions.
  In addition, it implements the buffer where data can be
  incremetally written either in the beginning or end of the
  buffer without copying everything to allocate a new header.
*/

import java.io.OutputStream;
import java.net.DatagramPacket;
import java.net.InetAddress;
import java.io.DataOutputStream;

public class smmOutputDatagramPacket extends OutputStream {
  private DatagramPacket packet;
  private byte [] buf;
  protected int head;
  protected int tail;
  protected int pos;
  public DataOutputStream dataOutStream;

  public smmOutputDatagramPacket (int size) {
    buf = new byte[size];
    packet = new DatagramPacket(buf, buf.length);
    head = size/4;
    tail = head;
    pos = head;
    dataOutStream = new DataOutputStream(this);
  }

  public smmOutputDatagramPacket (int size, InetAddress iaddr) {
    this(size);
    setAddress(iaddr);
  }

  public void reset() { // clear packet and set it to initial state
    head = buf.length/4;
    tail = head;
  pos = head;
}

public void setAddress(InetAddress iaddr) { // set destination address
  packet.setAddress(iaddr);
}

public void write(byte[] b) { // override OutputStream class method
  System.arraycopy(b, 0, buf, pos, b.length);
  pos+=b.length;
  if (pos > tail) tail = pos;
}

public void write(byte[] b, int off, int len) { // override OutputStream class method
  System.arraycopy(b, off, buf, pos, len);
  pos+=len;
  if (pos > tail) tail = pos;
}

public void write(int b) { // required by OutputStream abstract class
  buf[pos++] = (byte) b;
  if (pos > tail) tail = pos;
}

public int getPacketPos() {  // position where next write will occur
  return pos-head;
}

public void extendHead(int extensionSize) { // extend head for new header
  // and seek writing position to new head.
  head-=extensionSize;
  pos = head;
}

public void seekHead() {  // move writing position to packet's head
  pos = head;
}

public void seekTail() { // move writing positioon to packet's tail
  pos = tail;
}

public int getSize() { // return size of packet so far.
  return tail-head;
}
```

```
  public DatagramPacket getDatagramPacket () {  // return datagram holding packet
    packet.setData(buf, head, tail-head);
    return packet;
  }

  public void printState() {
    System.out.println("head= " + head + " tail= "+tail+ " pos= "+pos);
  }
}
```

Input Datagram Packet Class

```
/*
  This class extends ByteArrayInputStream to allow
  programmers to rewind the input stream based on the
  input packet so that the same object can be reused
  for multiple receptions.
*/

import java.io.ByteArrayInputStream;
import java.net.DatagramPacket;
import java.io.DataInputStream;

public class smmInputDatagramPacket extends ByteArrayInputStream {
  private DatagramPacket packet;
  public DataInputStream dataInStream;

  public smmInputDatagramPacket (int size) {
    super(new byte[size]);
    packet = new DatagramPacket(buf, buf.length);
    dataInStream = new DataInputStream(this);
  }

  public void rewind() { // main contribution of this class.
    pos = mark = 0;
  }

  public DatagramPacket getDatagramPacket () {
    packet.setLength(buf.length);
    return packet;
  }

  public void printState() {
    System.out.println("pos=" + pos + " count="+count+ " Mark="+mark);
  }
}
```

VITA

Agustín José González was born in Los Andes, Chile. He received his Bachelor of Science in Electronic Engineering from Federico Santa María Technical University (UTFSM), Valparaíso, Chile, in December 1986. He worked as Lecturer for the Department of Electronic of UTFSM from 1987 to 1994 and earned a Master of Science in Electronic Engineering in 1995 from the same university. The same year, Agustín was awarded a Fulbright/LASPAU scholarship for a Master’s Degree Program in Computer Science along with a scholarship from UTFSM to follow graduate studies towards a doctoral degree. He joined Old Dominion University in August 1995 and received a Master of Science in Computer Science in December 1997. In December 1996 and after having finished his master’s courses, Agustín was offered a research assistantship and was accepted in the Ph.D. program of the same department. He will resume his faculty position at UTFSM immediately after his dissertation defense.