Distributed Interactive Media

Inaugural-Dissertation
zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften
der Universität Mannheim

von

Dipl.-Wirtsch.-Inf. Martin Mauve
aus Köln

Mannheim, 2000

This thesis has been published as a book (M. Mauve. Distributed Interactive Media. ISBN 3-89838-471-3. infix. Berlin. 2000) and is copyrighted by infix (www.infix.com). The online version is available for free download with the permission of infix. If you like this work I would like to ask that you order a hardcopy from infix (it can be ordered online).

Thanks,

Martin Mauve

Dekan: Professor Dr. Guido Moerkotte, Universität Mannheim
Referent: Professor Dr. Wolfgang Effelsberg, Universität Mannheim
Korreferent: Professor Dr. Ralf Steinmetz, Technische Universität Darmstadt
Tag der mündlichen Prüfung: 08. September 2000

Foreword

In the early days of computer networking research focused primarily upon developing algorithms and protocols for the transmission of the then all-important discrete media. Text and image data needed to be transmitted between computers. These data are not time-dependent. Only since the early 90s has the focus moved to the transmission of continuous media streams over computer networks, particularly for audio and video. The demands of real time on such continuous media streams pose new challenges for the entire architecture of computer networks that can be summed up as “quality of service”. Networks must be able to provide guaranteed bandwidth and maintain a maximum absolute delay as well as a maximum variance in delay during transmission. Transmission protocols for audio and video streams in the Internet have been developed for just this purpose during recent years, but no uniform standard exists yet for guaranteed quality-of-service.

This Ph.D. thesis deals with a third type of data stream, termed by the author distributed interactive media. Martin Mauve must be credited with the discovery of this important novel class of media streams, he being the first author to describe this new media class. While computer science to date has proceeded on the basis that the data streams to be transported over computer networks can be classified as either discrete (time-independent) or continuous (time-dependent), Dr. Mauve presents tangible arguments for the existence of a third type of data stream, namely distributed interactive media streams that communicate only in discrete data units but are nonetheless time-critical. He describes their features precisely in his new “media model”. The application interface distinguishes between state packets and event packets, without knowledge of the exact semantics of the data for the transport layer. Nonetheless, as Dr. Mauve demonstrates, critical services within a distributed system can be implemented for distributed interactive media solely on the basis of the difference between a state packet and an event packet.

Based on his model of distributed interactive media streams, Dr. Mauve proceeds to re-define the concept of consistency. Though there already exists a great deal of research on consistency in distributed systems, earlier work deals primarily with systems that communicate solely by means of discrete events, it always having been the premise that the state of an instance cannot be autonomously changed. Dr. Mauve demonstrates that the classical concept of consistency no longer suffices in the case of many modern applications, an example thereof being a distributed simulation. During a distributed simulation, in each individual system motion continues locally, while at the same time additional interaction events are being exchanged. He has not only recognized this novel consistency problem but has also developed the local-lag algorithm, with which consistency at all venues can be achieved through use of a distributed interactive media stream. Important applications for his algorithm include distributed interactive simulation (for example, battlefield simulation) and the increasingly popular distributed Internet games.

For a proof of concept, Dr. Mauve developed a distributed system called TeCo3D that allows cooperative interaction with any 3D-model written in VRML. This research was supported by a grant from Siemens with an eye on the growing market for tools that enable cooperative work with 3D-objects over the Internet. The transformation of any existing VRML model, without recourse to manual reprogramming, into a model capable of cooperation represented a particular challenge. The TeCo3D system is completely functional. The instances of the distributed system communicate by means of a distributed interactive media stream.

Based on his model and his experience implementing TeCo3D, the author went on to develop a communication protocol, RTP/I, which serves to transmit distributed interactive media streams. RTP/I is a full-fledged protocol specification for the Internet, complete in all aspects. Such a protocol specification is a great engineering achievement, demanding long experience in the development of communication protocols. RTP/I has since been submitted to the IETF (Internet Engineering Task Force) and is well on its way to becoming an RFC.

But Dr. Mauve takes yet one more step: He demonstrates how to develop generic services for a large class of Internet applications solely on the basis of the concept of a distributed interactive media stream and the RTP/I protocol. He describes the design and implementation of a consistency maintenance service based on local lag; a late-join service that brings a latecoming session participant up to date based on the protocol

rather than engaging the application, and a service to record and play back the RTP/I-based sessions. The fact that one can most efficiently provide three different services on the basis of RTP/I independently of any application is proof of the power of the concept of distributed interactive media streams. Up until now, each individual application developer has had to implement consistency maintenance (at the level of the application protocol), a late-join service, and a recording tool, resulting in substantial overhead given today's number of cooperative applications.

Both the concept of a distributed interactive data stream and the RTP/I protocol have been published at international conferences, among them the IEEE Multimedia 1999 in Florence, Italy, and ACM Multimedia 2000, in Los Angeles, U.S.A., two of the most prestigious conferences world-wide. Two further articles have been accepted for publication in the journals "IEEE Transactions on Multimedia" and "ACM Computer Communication Review". Finally, the Internet Draft on RTP/I has found excellent resonance in the Internet community. Martin Mauves dissertation can reasonably be called a seminal work that unlocks an entire new area within multimedia communication, sure to yield numerous new publication.

Mannheim, im November 2000 Prof. Dr.-Ing. Wolfgang Effelsberg

Abstract

In this thesis distributed interactive media are shown to form a class of media that share many important attributes. The key characteristic of media belonging to this class is that they are distributed media which involve direct user interactions with the medium itself. Typical examples of distributed interactive media are shared whiteboards, networked computer games, and distributed virtual environments. In order to prove that the idea of a common media class is valid, an abstract media model for the distributed interactive media class is given. Furthermore, existing distributed interactive media are investigated and it is shown that these are covered by the media model.

With the definition of a distributed interactive media class it becomes feasible to create a common foundation for the development of shared functionality that is usable by many - or all - media belonging to this media class. The development of an application level protocol as such a foundation for common functionality constitutes the key contribution of this thesis. The application level protocol is called “Real-Time Application Level Protocol for Distributed Interactive Media” (RTP/I). It was inspired by the “Real Time Transport Protocol” (RTP), which is commonly used for the transmission of audio and video over the Internet. The design decisions and the specification of RTP/I are discussed in detail.

In order to show that RTP/I can indeed serve as a common foundation for shared functionality, three generic and reusable services are presented: a consistency service, a service which supports latecomers who wish to join an ongoing session, and a recording service for distributed interactive media. Moreover, it is discussed how the media model, RTP/I, and the generic services have been used to develop a complex real world application for a distributed interactive medium. This application is known as TeCo3D (3D TeleCooperation). It allows a distributed group of users to view and interact with dynamic 3D models that are specified using the Virtual Reality Modeling Language.

Acknowledgments

This thesis has been written while I was working at the “Lehrstuhl für Praktische Informatik IV at the University of Mannheim” and at the “Siemens Telecooperation Center” in Saarbrücken. It was funded by a Ph.D. grant from Siemens, and by the European MECCANO Telematics for Research Project 4007. This thesis would not have been possible without the help and support of my colleagues, my friends, and my family. While the attempt to mention everyone by name would be bound to fail I want to credit at least some of those who helped me.

First of all I would like to thank Professor Dr. Wolfgang Effelsberg for his support and advice. He was always ready to discuss new ideas and to provide very important feedback on all issues and problems that I encountered during my work. I also wish to thank Prof. Dr. Ralf Steinmetz who volunteered to be referee for this thesis. Dr. Jean Schweitzer supported my work in many ways. Most important have been his suggestions regarding collaborative 3D. These provided the initial idea for the TeCo3D application.

The cooperation and the frequent discussions with my colleagues often lead to new ideas and helped me to solve many problems. In particular I wish to thank Volker Hilt and Christoph Kuhmünch for our joint work that led to the development of RTP/I. With his detailed and knowledgeable comments on the initial version of RTP/I Colin Perkins helped us a lot to improve it. Jürgen Vogel implemented the generic late join service and supported RTP/I with great enthusiasm. Tino von Roden implemented the second TeCo3D prototype. I learned a lot about writing a thesis and how to enjoy the social events at conferences from Werner Geyer. Betty Weyerer did a great job at proofreading the initial draft of this thesis. And there is Rüdiger Weis. His remarks about people ignoring security aspects taught me more about security and encryption than reading a couple of books on that subject.

Finally and most importantly I want to thank my family. I am very indepted to Visnja. Numerous discussions with her helped me to see problems from a different perspective and to solve them. She is a very special person. My parents Gisela and Reinhard supported me in every imaginable way. They are the best parents one could ask for.

Distributed Interactive Media

List of Figures

Abbreviations

ADU	Application Data Unit
ALF	Application Level Framing
API	Application Programming Interface
AWT	Abstract Window Toolkit
BBN	Bolt, Beranek and Newman
CBF	Compressed Binary Format
CD	Compact Disk
CID	Container Identity
DIS	Distributed Interactive Simulation
DIVE	Distributed Interactive Virtual Environment
dlb	digital lecture board
DVD	Digital Versatile Disk
DVE	Distributed Virtual Environment
EAI	External Authoring Interface
ESPDE	Entity State Protocol Data Unit
ftp	file transfer protocol
GPS	Global Positioning System
http	hypertext transfer protocol
IEEE	Institute of Electrical and Electronics Engineers
ILP	Integrated Layer Processing
IP	Internet Protocol
JETS	Java-Enabled TeleCollaboration System
JVM	Java Virtual Machine
LAN	Local Area Network
MTU	Maximum Transmission Unit
NAK	Negative Acknowledgment
NTP	Network Time Protocol
PC	Personal Computer

PDU	Protocol Data Unit
RTCP	Real-Time Transport Control Protocol
RTCP/I	Real-Time Application Level Control Protocol for Distributed Interactive Media
RTP	Real-Time Transport Protocol
RTP/I	Real-Time Application Level Protocol for Distributed Interactive Media
RTSP	Real-Time Streaming Protocol
SICS	Swedish Institute of Computer Science
SIMNET	Simulator Networking
SMP	Scalable Multicast Protocol
SNAP	Scalable Naming and Announcement Protocol
SRM	Scalable Reliable Multicast Protocol
TCP	Transmission Control Protocol
TV	Television
UDP	User Datagram Protocol
URL	Uniform Resource Locator
VRML	Virtual Reality Modeling Language
WAN	Wide Area Network

1 Introduction

1.1 Motivation

The Internet has taken the world by storm. Today it is hard to imagine being without access to the world's largest database known as the World Wide Web or to the universal messaging system called e-mail. Currently, successful Internet applications like Web, e-mail, and file transfer focus almost exclusively on a single user's retrieving or transmitting static, i.e., time-invariant, information. This is bound to change in the near future.

In fact, this change is already taking place: applications involving distributed continuous media, such as audio and video transmissions over the Internet, are gaining importance rapidly. Specific applications include Internet TV and radio broadcasts as well as video-conferencing and distance learning solutions. However, audio and video transmission will not be the last step towards more complex media that are distributed over the Internet.

The next step in the evolution of distributed media are applications that allow multiple users to interact with the medium in real-time. Typical examples of this challenging media class of distributed interactive media are shared workspaces, networked computer games, and distributed virtual environments.

There are many indicators that distributed interactive media will play an important role in the future of the Internet. One of these indicators, perhaps even the most important one, is the overwhelming success of networked computer games. With their famous "Ultima Online" Origin has attracted over 100,000 customers who pay $10 per month to participate in a persistent medieval universe, competing with other players for virtual wealth and fame. Today there are few, if any, other Internet services for which consumers are willing to pay a similar amount of money.

However, distributed interactive media are certainly not limited to networked computer games. Another indicator of the eventual importance of this media class is the increasing effort to develop synchronous CSCW (Computer Supported Collaborative Work) applications for a wide range of areas. Examples include shared whiteboards for remote presentations, distributed engineering applications for joint construction, collaborative 3D models for remote product support, and shared web-browsing for innovative call center solutions.

In the remainder of this work we use the term *medium* to describe a combination of an information encoding and a set of applications that allow a user to access the encoded information. Examples for media in this sense are audio, video, and whiteboard presentations. There exist other meanings of the term medium which are not used in this thesis.

1.2 The Problem - Objectives of this Thesis

Unfortunately, applications involving distributed interactive media are not only very attractive and desirable for a large group of users, but they are also very demanding and complex. The primary reason for this complexity is that a distributed interactive medium needs to be kept consistent for all users interacting with it. In a shared whiteboard presentation, for example, all users should be able to see the same current whiteboard page with all modifications and annotations made by each user; in a distributed virtual soccer game, every player needs to know the position of the other players as well as the position of the soccer ball. Moreover, since ‘real’ users are involved, consistency must be realized under real-time constraints.

The complexity of distributed interactive media is not only encountered for the basic functionality provided by an application for this media class. It is also present whenever the application is enhanced by additional functionality. As an example consider the ability to join an ongoing shared whiteboard presentation. Such a ‘late-join’ requires that the application joining the session must be updated to the current state of the presentation, so that the new user perceives the same information as any other user. Generally, the realization of late-join functionality for distributed interactive media is non-trivial, especially if it needs to ensure consistency as well as scalability.

The ability to join an ongoing session is a commonly required functionality that is useful for a large number of distributed interactive media. Another example of commonly required functionality is the ability to record and replay a session. This is not only desir-

able for CSCW applications, but also for military battlefield simulations and networked action games.

In fact, most applications for distributed interactive media can be decomposed into *services* (consistency mechanisms, late-join, recording, object based encryption, floor control, etc.) that could theoretically be used not only by a single medium but also by many other representatives of the distributed interactive media class.

However, currently there exists no common foundation upon which services for distributed interactive media could be built in a generic and reusable fashion. Therefore most applications for this media class require the redesign and reimplementation of already existing functionality. This is particularly wasteful in view of the complexity inherent to the development of applications and services for distributed interactive media.

It is the main aim of this work to solve this problem by providing a common foundation for distributed interactive media upon which applications and services can be developed in a generic and reusable fashion. We divide the development of such a foundation into three fundamental steps. The first step is to define an abstract model for distributed interactive media. An abstract model establishes a common view on the concerned media class and allows the investigation of common aspects and problems on a formal basis. Thus the definition of an abstract model is of great importance for the overall understanding of a media class.

The second step entails the design of an application level protocol for distributed interactive media. The protocol needs to capture the common aspects of this media class, in order to build generic and reusable services on top of the protocol without further knowledge of media-specific details. An application can then be composed from these generic services in combination with some medium-specific functionality. The application level protocol is the ideal place to establish a common foundation for reusable services and functionality since it can be accessed not only by the applications, but also by intermediate systems in the network. These are therefore able to provide middleware services like distributed session recording and playback.

The third, and final, step in developing a common foundation for distributed interactive media is to prove that the approach is valid. This should be done by developing a number of generic services and a complex real-world application.

In the following chapters of this thesis we investigate all three steps in detail, providing for the first time a common view on the main problems, challenges, and solutions for distributed interactive media.

1.3 Chapter Overview

The second chapter of this thesis gives an overview of the class of distributed interactive media. This overview starts with the definition of an *abstract media model*. The model provides the terminology as well as the common view required for the discussion of distributed interactive media. At the same time it clearly outlines the scope of our work - only media that fit into the model will profit from the work presented here. Following the definition of the media model is an introduction into *basic design concepts* of applications for distributed interactive media. These concepts include network support, distribution architectures, and other design principles. The second chapter ends with the description of four *existing distributed interactive media*. For each of these media we show how they fit into the media model and which concepts and design principles they use.

In Chapter Three we discuss the problem of *consistency* in distributed interactive media. Consistency has been investigated in great detail for distributed interactive media which change their state only in response to user actions (discrete media). Shared whiteboards and distributed text editors are typical examples for this sub-group. However, distributed interactive media which change their state because of user actions and because of the passage of time (continuous media) have received only limited attention. Representatives of this sub-group are animated 3D models and networked computer games. We therefore summarize existing work on consistency for the discrete domain, while providing a detailed and formal investigation of consistency for the continuous domain.

The important aspects of distributed interactive media having been discussed in the previous chapters, Chapter Four introduces *a sample application called TeCo3D*. TeCo3D is a shared workspace for animated and interactive 3D models that has been developed in the context of this thesis. Its aim is to allow users to share collaboration-unaware VRML (Virtual Reality Modeling Language) models, i.e., 3D models that were not specifically developed to be used by more than one user at a time. With this functionality it is possible to include arbitrary VRML content, as generated by standard CAD or animation software, into teleconferencing sessions. TeCo3D is a sample application for continuous

distributed interactive media that is used throughout this thesis to demonstrate the viability of our approach.

In Chapter Five we present RTP/I, a *real-time application-level protocol* for distributed interactive media. By identifying and supporting the common aspects of distributed interactive media, RTP/I allows the reuse of common functionality and the development of generic services for this media class. Derived from the experience gained with audio and video transmission using the Real-Time Transport Protocol (RTP), RTP/I is defined as a new protocol that reuses many aspects of RTP while it is thoroughly adapted to meet the demands of distributed interactive media. This chapter discusses major design principles and the requirements for an efficient real-time protocol. It gives an introduction to RTP and the detailed specification of RTP/I. Additionally we show how RTP/I is used for the TeCo3D application.

Chapter Six focuses on the description of the *generic services* we have developed for RTP/I. The first generic service uses the algorithms described in Chapter Three to ensure consistency for continuous distributed interactive media. The second generic service addresses the problem of latecomers by providing a scalable and efficient late-join functionality. The third generic service allows the recording of live RTP/I session as well as random access to and replay of recorded RTP/I sessions.

Chapter Seven describes the *experiences* gained through the development of three distinct TeCo3D prototypes. The first prototype was developed to demonstrate that collaboration-unaware VRML models can be shared. The second prototype proved that such functionality can be integrated into other teleconferencing applications. Both approaches provided only core functionality (no recording, no late join, etc.) and each used a proprietary application level protocol. The third prototype represents the second generation of TeCo3D. Based on the experience with its predecessors, this prototype was built upon RTP/I. The services described in the previous chapter, as well as an RTP/I library were implemented in a reusable fashion as generic services. In addition to explaining the architecture and the implementation of the TeCo3D prototypes, this chapter also contains our experiences with developing a complex application for a specific distributed interactive medium.

In Chapter Eight we summarize the work done in the context of this thesis. Since distributed interactive media are a large and challenging area, many interesting topics remain open for further investigation. The most interesting ones that we identified throughout our work in this area are introduced in a brief outlook on future work.

1.4 Scientific Contributions

The main scientific contributions of this thesis are as follows:

1. The class of distributed interactive media was identified as distributed media that involve user interaction. We developed an abstract model that allows the scientific discussion of this media class.
2. We developed a formal consistency criterion for continuous distributed interactive media. Furthermore we introduced a novel mechanism, called local lag, which ensures that this consistency criterion can be met by a real-world application.
3. We specified and implemented RTP/I, a real-time application level protocol for distributed interactive media. This protocol allows the development of generic, reusable services and functionality for distributed interactive media.
4. Three generic services for distributed interactive media were developed in the context of this work: consistency support, late-join, and session recording and playback.
5. As a real-world example we have developed an innovative mechanism for sharing collaboration-unaware interactive 3D models. This mechanism allows the integration of unmodified standard 3D objects into teleconferencing sessions.

All of the contributions have been fully implemented and tested within a prototype of the TeCo3D application.

2 Distributed Interactive Media - an Overview

What exactly is a distributed interactive medium? What design principles do applications for distributed interactive media follow? What do examples of distributed interactive media look like? These are the three main questions that are answered in this introduction to distributed interactive media.

2.1 Media Model

2.1.1 Classification of Distributed Media

In order to define the scope of our work, we distinguish between distributed media types by means of two criteria. The first criterion ascertains whether the medium is discrete or continuous. The characteristic of a *discrete medium* is that its state is independent of the passage of time. Examples of discrete media are still images or shared whiteboard presentations. While discrete media may change their state, they do so only in response to events, such as a user drawing on a shared whiteboard. The state of a *continuous medium*, however, depends on the passage of time and can change without the occurrence of events. Video and animations belong to the class of continuous media.

The second criterion establishes whether media are interactive or non-interactive. *Non-interactive media* change their state only in response to the passage of time and do not accept events. Typical representations of non-interactive media are video, audio, and images. *Interactive media* are characterized by the fact that their state can be changed by events such as user interactions. Whiteboard presentations and distributed virtual environments (DVEs) represent interactive media. Figure 1 depicts how the criteria characterize different media types.

Distributed media that are neither interactive nor continuous are well understood and therefore discussed no further here. Media types that are non-interactive and continuous have already been investigated to a large extent in the context of audio and video transmission. Especially the Real-Time Transport Protocol (RTP) [40] provides a solid base for the development of applications and services for this media class.

In contrast, applications for distributed interactive media usually do not share a common foundation. Developers of shared whiteboards, for example, will only rarely consider the algorithms and solutions available for networked computer games. While upon first glance these two media types are certainly different, they have something important in common: each needs to manage the shared state of an interactive medium. Given this joint aim, it is likely that one distributed interactive medium might profit from the solutions developed for a different distributed interactive medium. However, this is only possible if there exists a common view on the media class. The best way to establish such a common view is to develop an abstract media model.

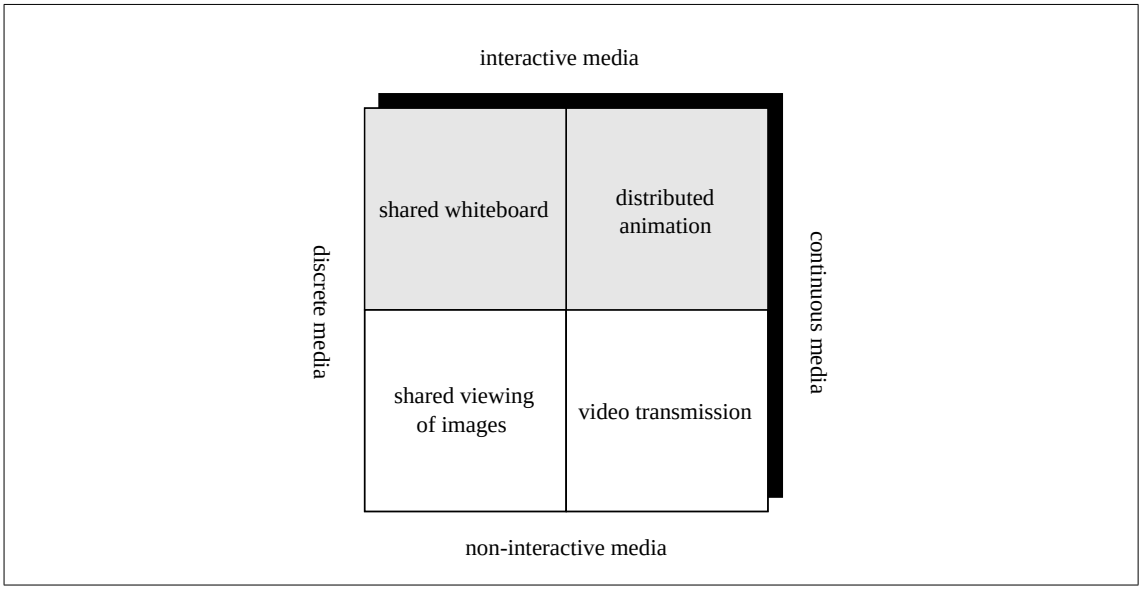

Figure 1: Examples of distributed media

2.1.2 Model for Distributed Interactive Media

2.1.2.1 Shared State

A *distributed interactive medium* has a *state*. For example, the state of a shared whiteboard is defined by the content of all pages present in the shared whiteboard. In order to perceive the state of a distributed interactive medium a user needs an *application*, e.g., requires a shared whiteboard application to see the pages of a shared whiteboard presentation. This application maintains a local copy of (parts of) the medium's state. For all applications participating in a session the local state of the medium should be at least reasonably similar. It is therefore necessary to synchronize the local copies of the distributed interactive medium's state among all participants, so that the overall state of the medium is *consistent*.

From these definitions it becomes clear that applications for distributed interactive media do employ a *replicated distribution architecture*. That is, a full instance of the application

is running on each participants' computer. Media that are displayed by applications with a *centralized distribution architecture* do not have the notion of a shared state and are therefore no distributed interactive media as defined here. An application that is distributed by employing *application sharing* is an example for using a centralized distribution architecture. Application sharing allows a spacially distributed group of users to view and interact with an unmodified single user application. It is typically realized by intercepting the communication between the application and the window system. Doing this, an application-sharing software is able to multiplex the output of a single user application to multiple users and to forward their actions to the application [21,29,67,69]. An application that is shared among multiple users by employing application sharing cannot be the application for a distributed interactive medium since the state of the medium is kept in a single location (the place where the application is running). In the remainder of this thesis we will only consider distributed interactive media.

2.1.2.2 State Changes - Passage of Time and Events

The state of a distributed interactive medium can change for two reasons, either by *passage of time* or by *events*. The state of the medium between two successive events is fully deterministic and depends only on the passage of time. Generally, a state change caused by the passage of time does not require the exchange of information between applications, since each user's application can calculate the required state changes independently. An example of a state change caused by the passage of time is the animation of an object moving across the screen.

Any state change that is not a fully deterministic function of time is caused by an *event*. Events can be separated into *external events* and *internal events*. External events are (user) interactions with the medium, e.g., the user makes an annotation on a shared whiteboard page. Internal events are non-deterministic internal changes in the state of the medium, such as the generation of a random number. Whenever events occur, the state of the medium is in danger of becoming inconsistent because the local copies of the state might cease to be synchronized. Therefore, an event usually requires that the applications exchange information - either about the event itself or about the updated state once the event has taken place.

2.1.2.3 Partitioning the Medium - Sub-Components

In order to provide for a flexible and scalable handling of state information, it is often desirable to partition an interactive medium into several *sub-components*. In addition to breaking down the complete state of an interactive medium into more manageable parts,

such partitioning allows the participants of a session to track only the states of those sub-components in which they are actually interested. Examples of sub-components are 3D objects (a house, a car, a room) in a distributed virtual environment, or the pages of a shared whiteboard. Events, external as well as internal, affect only a single *target* sub-component. Sub-components other than the target are not affected by an event.

It is important to realize that the sub-components must be independent of each other since some applications might track only the state of a subset of all available sub-components. Sub-component A is said to be *independent* of sub-component B if the state of B influences the state of A only by means of events. While the independence of sub-components is usually guaranteed for a medium like a shared whiteboard, other media require the generation of *pseudo* events in order to ensure the independence of sub-components.

We will illustrate this problem with the following example: let the distributed interactive medium be a car race with an arbitrary number of participants. If the medium were not separated into several sub-components, the only events would be user interactions for steering the cars. Each application could always determine the state of the interactive medium just by monitoring these external events.

However, generally it would be a good idea to partition the state of this medium into sub-components: the individual cars in the race. This would allow an application to track only the state of those cars that are close to the car of the user. Also, this would make management the overall state of the medium easier since in the partitioned medium each application is only responsible to make sure that the state of the local user's car remains consistent for all participants. In contrast, a non-partitioned medium would require that all applications cooperatively maintain the state of all cars, which is a much more complex task.

Unfortunately, without further work the states of the cars are not independent of each other. A car might "bump" into another car, thereby changing the state of both cars. For the non-partitioned medium this does not pose a problem since each participating application would see this "accident" happening and could modify the overall state accordingly. In the partitioned medium, however, some applications might track the state of only one of the two cars, because the other car is not deemed "relevant" to the local user. These applications would not notice the accident and would therefore calculate a wrong state for the affected sub-component.

The solution to this problem is the introduction of pseudo events. *Pseudo events* are generated whenever a sub-component would influence the state of another sub-component without using external or internal events. In the car example two pseudo events should be generated: one collision pseudo event for each car. The pseudo events are generated by

the participant(s) responsible for detecting the collision. This could be the controlling application of each car, or a specialized collision detection application. As for internal and external events, pseudo events generally require that the applications exchange either information about the event itself, or about the updated state after the event has taken place. Otherwise the consistency of the overall medium might be damaged.

2.1.2.4 Delta States

For some media the state of sub-components may still be rather large. Therefore it is desirable to be able to identify those parts of a state that have actually changed since the last full state has been determined for a sub-component. We call the information that contains the ‘difference’ between the last determined full state and the current state of a sub-component a *delta state*. For some media delta states may not be accessible, for other media they may be very important, especially when the states of sub-components are large and need to be transmitted frequently.

The concept of delta states is similar to that of P frames in an MPEG-encoded video stream. A delta state can only be interpreted if the preceding full state is also available (see Figure 2). The main advantages of delta states are their smaller size and that they can be calculated faster than full states. Delta states may be calculated for sub-components as well as for the overall state of the medium.

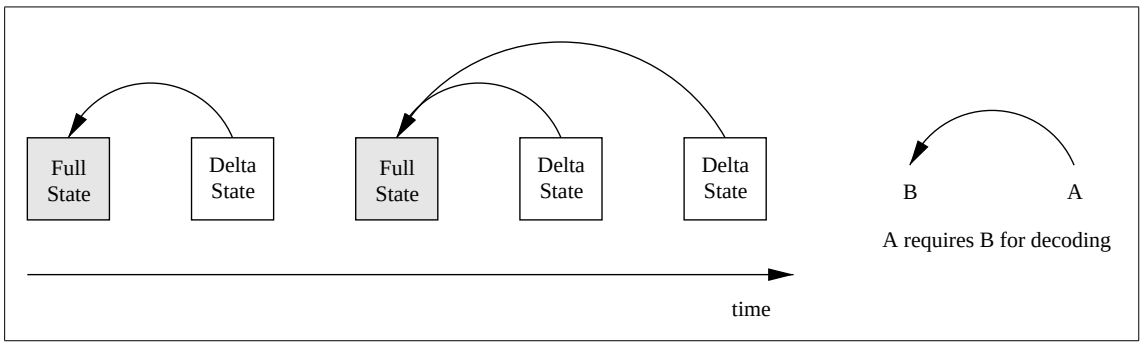

Figure 2: Decoding delta states

2.1.2.5 Environment

While it would be conceivable to declare all the information that is required by the application to display the distributed interactive medium to be part of the medium’s state, this is generally not desirable. Usually a substantial part of this information remains constant over the course of a session. We call this constant information the *environment* of a distributed interactive medium. Examples of environments are the base world descriptions of distributed virtual environments, or the postscript pages of shared whiteboard presentations. Since the environment stays constant, there are no mechanisms required to syn-

chronize it among the participants of a session - the environment information just needs to be received once by each participant.

It is therefore a good idea to make the environment part of the medium data as large as possible and to minimize the amount of state information. The distinction between state and environment information is floating. A participant might choose to introduce a new postscript page on-the-fly into an ongoing shared whiteboard presentation, thereby making this page part of the medium's state. It is therefore the task of the application designer to distinguish between state and environment information. Since the environment is a discrete non-interactive medium we do not further investigate how the environment is shared between participants. For the remainder of this work we merely assume that the environment is present for all participants' applications (e.g., by using multi-destination file transfer or by means of an off-line distribution on CD or DVD).

The introduction of the term environment concludes our definition of a model for distributed interactive media. Now is the right point to consider what the model contributes to the main goal of establishing a common foundation for this media class. The answer is threefold: first, the model gives us the terminology required to discuss distributed interactive media without referring to any single specific medium. Second, it delimits the scope of our work. And third, it shows us the general direction into which we should progress.

In the next section we will investigate important issues of designing an application for this media class.

2.2 General Design Decisions

The developers of applications for distributed interactive media face several significant design decisions. For a given medium it is important to carefully consider the implications of each decision, in order to design the application to fit the medium. Figure 3 shows the logical components of two possible application designs.

Generally, four logical components are common to all applications for distributed interactive media, regardless of the design choice. The *media renderer* is responsible for displaying the medium to the user, it renders the state of the medium's sub-components. One example for a media renderer is a VRML (Virtual Reality Modeling Language) browser that is able to display 3D models. The *input devices* (e.g., a mouse, a joystick, a data glove) allow a user to interact with the medium. The *network interface* is used for communication of the application with peer instances. The last logical component is the *state machine* which is responsible for managing the shared state of sub-components.

In the design shown in Figure 3 (a) the external events generated by the local user are directly incorporated into the state of the affected sub-component. Peer applications are informed about this state change by means of an event transmission. From time to time it might be necessary to transmit the full state of a sub-component, e.g., to accommodate latecomers or as a means of resynchronization.

The application in Figure 3 (b) takes a different approach: here, too, the local events are injected into the state of the sub-component. However, instead of transmitting them, the new state of the sub-component is sent to the peer applications. Both of these approaches have distinct advantages for specific media. The first approach (a) is commonly used by shared whiteboards where the amount of state information is large. The second approach (b) is usually taken by networked action games, where the state is small and the transmission of the entire state information can repair the damage done by lost network packets.

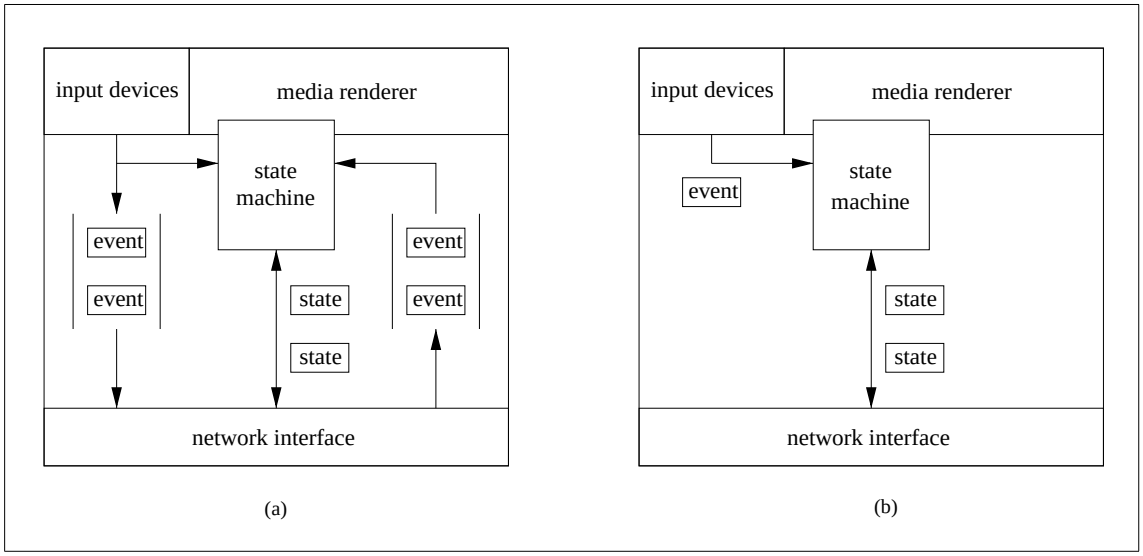

Figure 3: Logical components of applications for distributed interactive media

This shows that there are considerable degrees of freedom in the design of applications for distributed interactive media. In the following sub-sections we will discuss four important areas: which network protocol technique should be employed, how reliability can be achieved, who is responsible for maintaining the shared state of sub-components, and what kind of information should be exchanged between applications. The most important design decision, however, is how the overall consistency of the medium is realized. Since this is a complex topic that required substantial research in the context of this thesis, it is discussed in a separate chapter (Chapter Three).

2.2.1 Network Protocol Techniques

There exist three distinct network layer protocol techniques in the Internet that are used for the communication between two or more end-systems. *Unicast* is the most traditional one. It is employed to convey information from one sender to exactly one recipient. For a

local area network (LAN) the *broadcast* technique can be utilized to deliver information from one sender to all members of the LAN. Finally, *multicast* is used to transmit information from a single sender to a group of end-systems in the Internet. Applications for distributed interactive media have been built with all three network protocol techniques. In the following sub-sections we therefore examine each technique in detail.

2.2.1.1 Unicast

In the Internet unicast is realized by the *Internet Protocol* (*IP*). IP provides an unreliable and connectionless network service in which a sender transmits data in form of *IP packets*. These packets contain a unique *IP address* of the receiver. Intermediate systems between the sender and the receiver use the IP address to forward the IP packet until it arrives at the receiver. These intermediate systems are called *routers*; they represent the access points of individual sub-networks that IP packets might cross in order to reach their destination.

The time required for an IP packet to be transmitted from a sender to the receiver is called *latency*. Latency is an important value for distributed interactive media since it represents the time that an event or state transmission needs to arrive at a remote peer application. If the value for latency is large, then the actions of one participant might take effect with a significant delay for the other participants, resulting in consistency problems. Typical values for unicast latency are less than 1 ms for a LAN, 20 ms within a European country, 40 ms for a continent, and 150 ms for transcontinental transmissions. We will investigate the problems caused by latency in more detail in Chapter Three - for now it is only important to understand that a large latency is undesirable.

Besides forwarding an IP packet towards the receiver, a router can choose to do two important things with an IP packet. First, it might choose to drop the packet. This is done when the router receives more incoming packets than it can handle. As a result IP packets might get lost. While it is conceivable that packet loss might also occur as a result of transmission errors, the overwhelming majority of packet loss in the Internet originates from packets dropped by routers. Packet loss increases the latency value - either the packet remains lost and therefore the latency is infinite, or the loss is detected by the end systems and the packet is retransmitted, thereby increasing the latency significantly.

The second thing that a router can do with an IP packet is to fragment it if it is too large to be handled by a sub-network. Depending on the network technology of a given sub-network, the size of an IP packet is limited to a certain amount of bytes. This limit is called the *Maximum Transmission Unit (MTU*) of that network. If a router forwards an IP packet from a sub-network with a given MTU to a sub-network with a smaller MTU, then the size of the original IP packet might exceed the smaller MTU. In this case the

router needs to fragment the original IP packet. The fragments are transmitted as independent IP packets until they reach the receiver. At the receiver they are reassembled before delivery as a single IP packet.

Generally, IP level fragmentation is considered inefficient [42], the problem being that the loss of a single fragment results in the loss of the entire original IP packet, even if all but one fragment have reached their destination. This increases the likelihood of packet loss and is thus undesirable for distributed interactive media. Instead of relying on IP level fragmentation, applications for distributed interactive media should make sure that the IP packets they are transmitting do not exceed the size of the smallest MTU between the sender and the receiver of the IP packet. The size of the smallest MTU can be discovered by using *Path MTU Discovery* [70]. Essentially Path MTU Discovery works by examining certain Internet Control Message Protocol (ICMP) [79] packets that contain information about fragmentation.

In conclusion it can be noted that IP unicast is both efficient and a well-established network protocol technique for the transmission of information from one sender to exactly one receiver. All the dominating internet services (e.g., the world wide web, e-mail, ftp) are currently built upon IP unicast.

However, unicast may become inefficient when more than one receiver for the same information exists. As depicted in Figure 4, with IP unicast the same information is transmitted separately to each receiver, crossing the same sub-network more than once. Especially those sub-networks close to the sender will thus be burdened with many IP packets all carrying the same information, effectively wasting bandwidth.

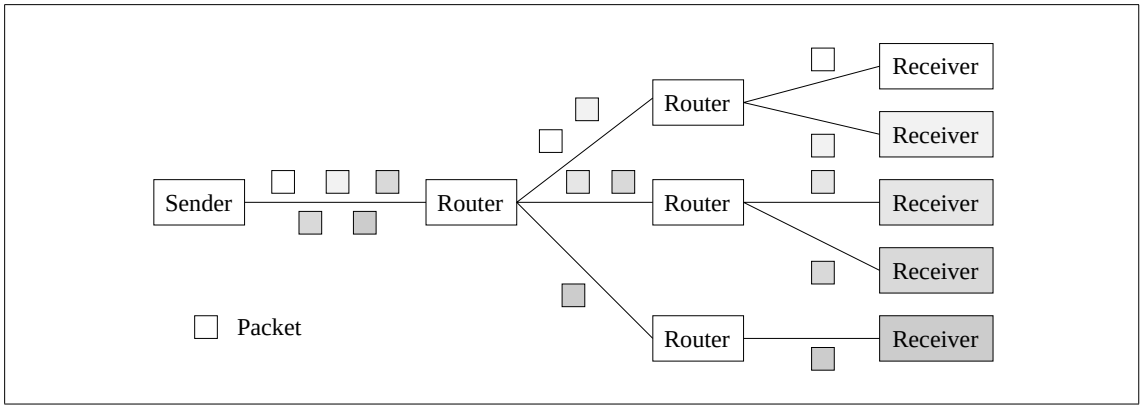

Figure 4: Unicast packet delivery

Since distributed interactive media frequently require the transmission of event and state information to a group of peer applications, they are especially affected by this problem. In order to avoid wasting bandwidth by transmitting identical information over the same sub-network, applications for distributed interactive media frequently rely on broadcast or multicast as a network protocol technique.

2.2.1.2 Broadcast

The term broadcast is used to describe a network protocol technique, in which all packets of a sender are received by all end-systems in a LAN. Rather than the address of a single end system the sender uses a special address for the broadcast packets. Members of the LAN will recognize this address and accept the packet as if it carried their address. This approach makes sure that no part of the LAN will carry more than one copy of the packet.

In the event that the participants of a session share the same LAN, broadcasting packets is a very attractive approach. It is regularly used by computer games that incorporate multi-player functionality for a small number of players. Broadcast is generally unsuitable for usage outside of a single LAN since it would flood the network with packets that are of interest only to a small percentage of the end-systems in the network (exceptions being special purpose networks, such as those used for TV broadcasts, or military battlefield simulations).

Because of their limitation to a single LAN, broadcast solutions typically have only a negligible amount of latency. Also the fragmentation problem is simplified since only a single sub-network is involved in the transmission of the packet. The smallest MTU is therefore equal to the MTU of the LAN.

In summary it can be noted that broadcast is very efficient if all the participants of a session share the same LAN or a specialized network. For applications that need to work over the Internet it is inappropriate. We therefore investigate a third network protocol technique called multicast.

2.2.1.3 Multicast

The general idea of multicast is to establish a tree of routers whose root is a router of the sender's LAN and whose leaves are the routers of the receivers' LANs. This tree is called a *multicast tree*. A packet transmitted by the sender is propagated along the edges of the multicast tree, with the guarantee that only a single copy of each packet passes over each edge. At each inner node of the multicast tree the packet is copied to all outgoing edges. When a multicast packet reaches a LAN containing one or more receivers, it is broadcast in that LAN. With multicast the example depicted in Figure 4 could be optimized to the situation shown in Figure 5.

(a) Multicast Routing

In the Internet there are currently two fundamentally different classes of multicast routing protocols in use. Representatives of the first class establish a separate multicast rout-

ing tree for each pair of a sender and a group of receivers. Examples include the Distance Vector Multicast Routing Protocol (DVMRP) [101], the Multicast Open Shortest Path First Routing Protocol (MOSPF) [71], and the Protocol Independent Multicast Routing Protocol in Dense-Mode (PIM Dense-Mode) [12]. Multicast routing protocols of this class are only viable when a large percentage of the network's LANs includes at least one participating receiver (i.e., when the multicast tree is dense).

The reason for this limitation is that the routers need to maintain one multicast tree for each pair of a sender and a receiver group. While this generally ensures an efficient routing of packets, it also leads to a large amount of routing information in the routers when many senders transmit data to a given group of receivers (e.g., video-conferencing, distributed computer games). While this is acceptable for a dense multicast tree since the number of receivers is high compared to the number of routers, it is inappropriate for sparse multicast trees since the number of routers involved increases significantly. In addition, DVMRP and PIM Dense-Mode occasionally use flooding to inform potential participants about the current senders. This is not acceptable for a sparse multicast tree because many nodes are getting flooding packets from a transmission they are not interested in.

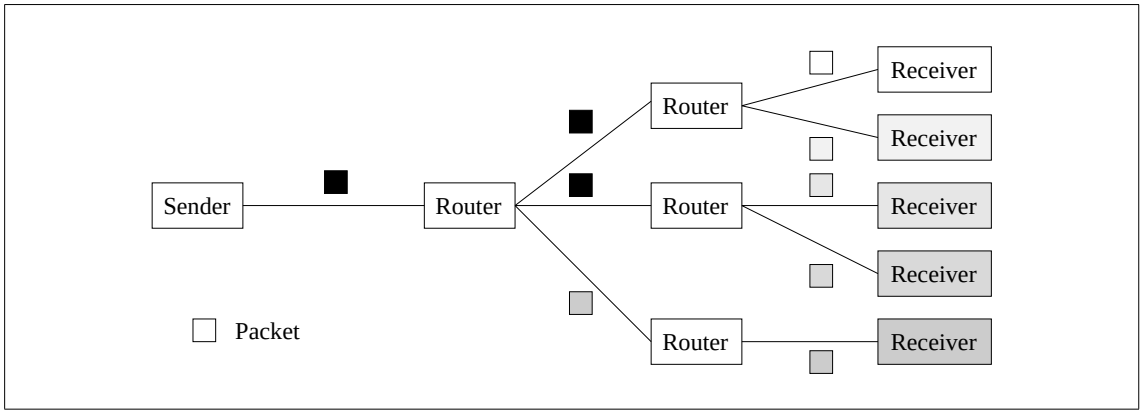

Figure 5: Multicast packet delivery

The second class of multicast routing protocols was developed specifically to be scalable and useful in situations where only a small number of the network's LANs include a receiver that is interested in receiving packets from the same sources. The fundamental idea of these protocols is to use a shared multicast tree for all senders that wish to transmit packets to a given group of receivers.

The two most important representatives of this class are the Protocol Independent Multicast Routing Protocol in Sparse-Mode (PIM Sparse-Mode) [11] and the Core Based Trees (CBT) [4]. Protocols in this class define specific routers as rendezvous routers. Each group of receivers has one such rendezvous router. As shown in Figure 6, a shared multicast tree is built with the rendezvous router as the root node and the routers of the receivers' LANs as leaf nodes. Senders that wish to transmit packets to the group of

receivers forward the packet to the rendezvous router by means of unicast. The packets received by the rendezvous router are then distributed to the group of receivers over the shared multicast tree.

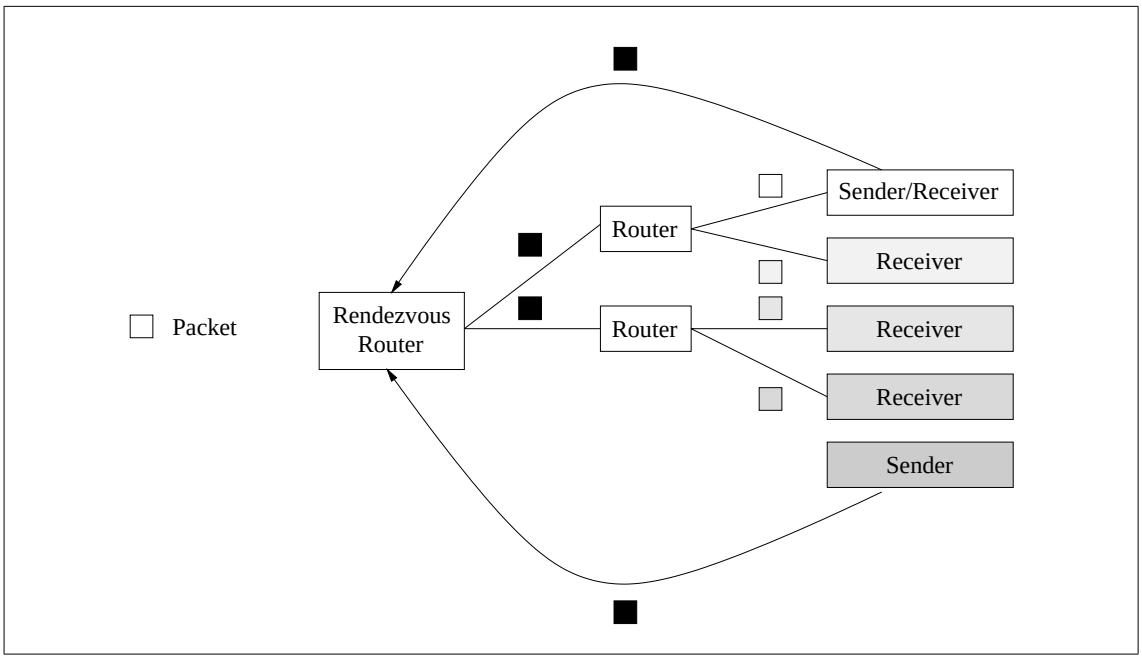

Figure 6: Shared multicast tree

Multicast routing algorithms with a shared multicast tree usually allow for a dynamic migration to a sender-specific multicast tree in order to account for cases where a small group of senders generates a heavy network load. This makes sense when the effort to establish and maintain a separate multicast tree for a given sender is compensated by the bandwidth saved due to the more efficient routing with a sender-specific multicast tree.

(b) MBone

In the Internet only a fraction of all available routers are multicast capable. In order to be able to span multicast trees across routers that are not multicast capable the tunneling mechanism is used. A multicast-capable router uses tunneling by wrapping multicast packets into regular unicast IP packets and by sending them via unicast to the next multicast-capable router in the multicast tree. The destination router of the IP packets unwraps the packets and takes appropriate actions (e.g., re-wrapping and forwarding them along the edges of a multicast tree, or broadcasting them in a LAN). In this manner the collection of all multicast-capable routers in the Internet forms a logical overlay network known as the Multicast Backbone (MBone) of the Internet [52].

(c) Multicast Group Management

The protocol mechanisms for establishing and maintaining the multicast trees are specific to the multicast routing protocol employed. In contrast, the signalling of end-systems that want to participate as receivers is standardized by the Internet Group Management Protocol (IGMP) [16]. IGMP is used between end-systems and the router of the end-system's LAN to signal for which multicast group receivers are present in the LAN. The multicast groups are identified by a class D multicast address. Once the local router knows that receivers are present for a given multicast session, it can take multicast routing protocol-specific actions to be included in the multicast tree(s) for that session.

(d) Evaluation

Let us consider the latency induced by using multicast as the network protocol technique. Because multicast sessions in the Internet generally have a sparse multicast tree, we can assume that in the long run multicast routing protocols with a shared multicast tree will be used. The latency for multicast will therefore be higher than for a unicast connection between the sender and a single receiver. This is caused by two factors: first, the packet has to be transmitted to the rendezvous router before it is distributed along the multicast tree. Second, it is unlikely that the multicast tree will establish an optimal (in regard to latency) route to each receiver.

However, in the unicast scenario, the packet needs to be transmitted once to each participant. This increases the latency at the sender and the likelihood that packets will get lost due to network saturation. Thus there exists a break-even point at a certain number of receivers beyond which multicast will provide a better mean latency than unicast, even when a shared multicast tree is used.

Fragmentation in multicast is inherently more difficult to avoid than in unicast. The reason for this problem is that there might be different minimal MTUs for different receivers. Instead of trying to completely avoid IP fragmentation, multicast applications often use the value 576 bytes as an upper bound on the size of transmitted IP packets. This is reasonable since in most cases the packet will only encounter sub-networks that have MTUs larger than that [6,70].

Summarizing, it can be said that multicast promises to become an efficient way to realize group communications. However, currently the MBone is only an experimental testbed. Today MBone sessions frequently suffer severe problems with high packet loss and network segmentation caused by misbehaving routers and a low allocation of bandwidth for multicast use. In addition, the latency in the MBone is often unacceptably high (between 500 ms and over 1 second for an international session). We have encountered these prob-

lems regularly while employing the MBone for international project meetings. Until these problems are solved, multicast will be commercially viable only in network parts maintained by a single entity (e.g., the private network of a single company), or by specific provisions such as the reservation of dedicated ATM bandwidth for the edges of the multicast tree.

The discussion of multicast concludes our investigation of the different network protocol techniques used for distributed interactive media. This investigation completed, the following question arises: What impact should each individual network protocol technique have on a common application level protocol as the foundation for generic functionality in distributed interactive media? The answer is simple and straightforward: ideally they should have no impact on the protocol at all. On the contrary: a common application level protocol for distributed interactive media should be developed in a way that ensures its independence of the underlying network. Only when this condition is satisfied will it be a common protocol that can be employed by various distributed interactive media using any of the three network protocol techniques.

Independence of the network protocol technique does not mean that diversity is ignored. Instead it requires the careful design of an application level protocol that can be used over any distinct network protocol technique. An example of an application level protocol in which the independence from the network technique has been realized successfully is the Real-Time Transport Protocol (RTP) [40]. RTP can be used in unicast as well as in multicast and broadcast environments to efficiently distribute audio and video streams. Examples of application level protocols that do not satisfy this criterion are ftp or http - they rely on the Transmission Control Protocol (TCP) [78] at the transport layer, which in turn works only over unicast network protocols.

2.2.2 Reliability

Applications for distributed interactive media require that information about those state changes that are caused by events is eventually received by all participants. If this were not the case the local state kept by the individual participants could diverge, and the shared state of the medium could become inconsistent. Applications for distributed interactive media therefore need some form of reliable communication.

Generally there exist two fundamentally different ways to achieve reliability. The first is to use a reliable transport protocol that guarantees the delivery of all information transmitted. The second is to make the application network-aware and let the application establish reliability. These alternatives are described and discussed in the following two sections.

2.2.2.1 Transport Level Reliability

The traditional approach to achieve reliable communication is to use a reliable transport protocol. For unicast in the Internet the Transmission Control Protocol (TCP) [78] provides reliable and ordered packet delivery. In the event that a lower layer (typically the network layer) drops a packet, TCP will make sure that this packet is retransmitted. Any packet sent after a lost packet will be buffered at the receiver until the lost packet has been received and delivered to the application. In addition to guaranteeing a reliable and ordered packet delivery, TCP also provides mechanisms for flow- and congestion control. Currently TCP is used by virtually all unicast-based Internet applications that desire reliability at the transport level (e.g., the world wide web, file transfer, or e-mail).

For multicast the usage of reliable transport protocols is less uniform. There exist diverse reliable multicast protocols [51,90] that seek to provide services similar to TCP for multicast transmissions. This diversity exists because the best way to achieve reliability for multicast depends heavily on the application. The diversity in the following application characteristics may cause distinct reliable multicast protocol mechanisms to provide the best fit for a specific application:

- the number of receivers,
- the number of senders (one sender vs. many senders),
- the acceptable latency,
- the geographical distribution of receivers (in the same LAN vs. a world-wide session), and
- the rate at which senders and receivers join and leave the session.

Because of the application dependency it is considered unlikely that a single reliable multicast protocol will emerge that can satisfy the needs of all applications which desire reliable multicast communication at the transport layer. A more likely situation will be a collection of reliable multicast protocols that share a common interface. An application can then select a protocol that fits its communication pattern.

Realizing reliability by means of a reliable transport protocol has advantages that are independent of the network protocol technique. For both unicast and multicast, the usage of a reliable transport protocol is simple. The resulting architecture is clean and provides all advantages of a layered architecture, namely transparency of and independence from lower layer issues. Basically the application has the same view on the network as it has on the local file system.

However, there are also certain disadvantages to using a reliable transport protocol. Such a protocol knows nothing about the actual communication needs of the application. It just provides its service: the reliable and ordered delivery of packets. For many applications, however, this service might be inefficient.

For example, consider that a single participant transmits an event for sub-component 1 followed by an event for sub-component 2. Imagine that the first event gets lost for any one receiver. Now the second event will be buffered until the packet loss has been repaired. This is inefficient since the application most likely would have been able to process the event for sub-component 2 without waiting for the lost event to be retransmitted.

If the medium is continuous the situation might be even worse. The time at which the event for sub-component 2 should have been executed could have been missed just because it was buffered by the reliable transport protocol. This would most likely result in the triggering of a state repair mechanism that could have been avoided had the event for sub-component 2 not been buffered.

The inefficient buffering of packets is not the only problem that can be caused by a reliable transport protocol. In addition, the simple retransmission of lost packets might be inappropriate if a continuous medium is involved. The reason for this is as follows: if an event gets lost for a continuous medium, the detection of that loss and the mechanisms for the retransmission of the lost packet are likely to consume a large amount of time. Therefore the retransmitted event will almost certainly arrive late at the receiver. As mentioned above, this will likely trigger a state repair mechanism. The problem could be avoided if the (unnecessary) retransmission of the event were replaced by the transmission of some repair information (e.g., the state of the affected sub-component). However, this is beyond the scope of a reliable transport protocol since it does not possess the application level knowledge required to take appropriate actions.

In concluding, it can be noted that the reliability established by a reliable transport protocol is easy to use and leads to a clean architecture, albeit at the cost of efficiency, especially for continuous distributed interactive media. In cases where this cost is too high to be acceptable, reliability should be established by the application.

2.2.2.2 Application Level Reliability

As early as 1990 Clark and Tennenhouse envisioned a problem for high speed networks that is closely related to those described above. In their famous paper on “Architectural Considerations for a New Generation of Protocols” [9] they identified the presentation layer conversion as the main bottleneck for high speed networking. The reason they gave for this assumption was that no other protocol functionality is as costly in terms of CPU cycles than transferring the received data from an array of bytes to some kind of data structure. This consideration led them to the conclusion that the presentation conversion needs to keep going, even when packets get lost or arrive out of order. Otherwise, the application “will fall behind, and since it is the bottleneck, it will never catch up.”[9]

Today, because there is no presentation conversion for high volume data in the Internet, the presentation conversion is no longer the real bottleneck for the large majority of distributed interactive media. In most cases it is the network itself that limits the complexity of applications in this area. However, the approach chosen by Clark and Tennenhouse can solve not only the problem they identified, but also those efficiency problems described for transport level reliability. It is therefore worthwhile to investigate in detail the mechanisms proposed in [9].

The first observation of Clark and Tennenhouse was that lost and misordered packets caused the presentation conversion process to starve when a reliable transport protocol is used. This observation is based on the same reason as the inefficiency problem described above, namely that a reliable transport protocol will enforce an ordered delivery of packets. Even if packets are received by the transport protocol they will not be delivered to the application until all preceding packets have been delivered. Therefore the presentation conversion process might starve even when data is present at the receiver that could theoretically be processed.

In order to allow for out-of-order processing, Clark and Tennenhouse introduced the concept of *application data units (ADUs)*. An ADU contains information that can be processed by the application even if it is received out of order. In particular it contains enough information so that the receiver knows what the ADU refers to. The concept of using ADUs as transmission units and of framing ADUs so that an application knows where they belong is called *Application Level Framing (ALF)*. For distributed interactive media the candidates for ADUs are the transmitted states and events. In order for the receiving application to know what an ADU refers to, it should at least contain an identifier for the sub-component that is affected.

ADUs are delivered immediately to the application upon reception. It is the responsibility of the application to decide what to do in the event that an ADU gets lost or arrives out of order. Generally the application has three options to react to a lost ADU: it can request the retransmission of the missing ADU via transport level functionality, it can request the repair of the problem by the sending application, or it can ignore the issue. This allows for a very flexible and efficient handling of lost packets. For example, if an event gets lost for a continuous distributed interactive medium, the application might request the repair of the problem in terms of a state transmission, rather than a simple retransmission of the lost ADU.

An aspect that has been treated somewhat vaguely by Clark and Tennenhouse is the problem of ADUs that are too large to fit into a single network layer packet. If only the application were responsible for triggering retransmissions, then the loss of a single fragment of an ADU would be treated in the same way as the loss of the complete ADU as

the application is not able to act on incomplete ADUs. This is certainly inefficient; a very large ADU might well cause nearly infinite retransmissions when always at least one fragment gets lost.

A solution to this problem is to provide additional functionality for the repair of lost fragments. The application, however, needs to instruct the additional functionality as to which ADUs should be automatically repaired and as to when the application should be informed about ADUs discarded due to the loss of individual fragments.

Based on the architectural principle of application level framing (ALF), Clark and Tennenhouse developed a new engineering principle called *Integrated Layer Processing (ILP)*. To some extent ILP is an alternative to the traditional, layered engineering principle of network systems. ILP suggests to merge the functionality of layers, in order to reduce the overhead from both additional header information and frequent read/write/copy operations. Clark and Tennenhouse proposed the following overall application structure consisting of two manipulation stages:

> "*First, the transmission data units are received from the network. They are then examined to determine which ADU they belong to (the demultiplexing control operation) and where in the ADU they go (the re-ordering control operation). If part of an ADU is lost, then either some lower layer may try to fix the problem (if it has buffered sufficient information), the receiving application may cope, or the sending application is instructed to resend the whole ADU.*
>
> *Once a complete ADU is received, even if it is out of order with respect to other ADUs in the same application association, it can be passed to the application for the second stage of processing. This processing will include all the required data manipulations, including error and encryption checks, and possibly presentation conversion, if the sender has not performed this task completely. The two state approach is justified by the distinction between two classes of "out of order" events: misordering of transmission units; and misordering of complete ADUs."*[9]

ALF and ILP can be applied to distributed interactive media to eliminate the efficiency problems of transport level reliability. However, it does not come for free. The greater amount of flexibility is gained at the cost of increased complexity. After all, when using ALF and ILP the application needs to provide the functionality required to deal with lost and misordered ADUs.

The positive effect of the layered approach - encapsulation and transparency - is lost when the full flexibility of ALF and ILP is desired. This problem can be somewhat alleviated with a well-structured library that provides building blocks for ALF and ILP oriented reliability. We will investigate such a library in more detail in Section 2.3.2 and propose an alternative in Section 5.4. However, the complexity of an application based on the principles of ALF and ILP will always be higher than that of an application using transport level reliability. It is therefore of great importance for developers to carefully

study the trade-off between simplicity and efficiency before deciding to use any one of the two approaches.

Distributed interactive media have been developed with both approaches. A common foundation for distributed interactive media should therefore be useful for either approach. Neither transport level reliability nor application level reliability should be prevented by a common protocol for this media class.

In concluding the discussion of reliability it should be noted that another concept exists that is used to protect transmitted information from the loss of individual packets in the network. By transmitting redundancy information in addition to the original data the *forward error correction (FEC)* [50] approach *reduces* the likelihood that information does not reach the recipient. While FEC can be particularly useful for protecting time sensitive information like events from packet loss, it cannot guarantee the delivery of the information. The usage of FEC can therefore be regarded as orthogonal to the two reliability alternatives presented here [43].

2.2.3 Managing Shared State - Centralized vs. Distributed Approaches

Applications for distributed interactive media generally use a replicated distribution architecture, i.e., each instance of an application maintains a local copy of the medium's shared state. Depending on the requirements of the application, the local copy of the shared state may be limited to a sub set of all available sub-components. The existence of multiple local copies of the state of a single sub-component leads to the following question: Who is responsible to make sure that these copies are at least reasonably similar?

Depending on the application and the medium there are two alternative solutions to this problem. The centralized solution relies on a dedicated server that maintains a master copy of a medium's state. Whenever a participant generates an event for the medium, this event is transmitted to the server. The server will evaluate the event and transmit an event or an updated state to all participants in the session, including the participant who generated the event. Upon reception of this message all participants will update their local state accordingly.

Having a single point where the events of all participants are evaluated simplifies the management of shared state significantly. It can provide an ordering relation on all events generated by distinct participants of a session. Also, a centralized server makes it easy to prevent multiple participants from interacting with the same sub-component at the same time, e.g., by using locks. However, the centralized management of shared state has the typical disadvantages of a centralized system: it increases the overall latency for the

medium, it is a potential bottleneck for the entire system, and it represents a single point of failure.

It is important not to confuse the responsibility of shared state management with the distribution architecture of an application. In the case that has just been described, the shared state is managed by a centralized server. That is, the centralized server can decide on the order in which events take place, or it can manage access restrictions to the state of certain sub-components. However, the state of the medium is still kept in a fully replicated fashion by all participants, as is usual for a distributed interactive medium.

The second way to manage the shared state of a distributed interactive medium is to use a distributed approach. This requires that all participants provide mechanisms for state management. Typically these mechanisms need to be able to establish the proper ordering and timing of events and states transmitted by different participants of a session. Additionally they either need to avoid concurrent generation of events by distinct participants, or they have to be able to repair the problems arising from concurrent generation of events.

In general, distributed approaches for the management of shared state are more efficient and scalable than centralized ones. This comes at the cost of increased complexity. We will take a closer look at consistency in Chapter Three by investigating formal consistency criteria for continuous and discrete distributed interactive media. In the same chapter we also discuss how these consistency criteria can be met by real world applications. For now, it is only important to realize that the responsibility for consistency can either rest with a dedicated server or be accomplished in a distributed fashion by all participants.

2.2.4 Information Policy

Since applications for distributed interactive media use a replicated distribution architecture, they need to inform peer applications (or a server) about events triggered by the local user. In the previous section we investigated who is responsible to provide this information to the group of participants. In this section we shall discuss what kind of information is transmitted.

The two candidates for information that is exchanged between participants are events and states. Events are characterized by the following properties:

- They can be encoded using a small number of bytes, typically less than 30-50 bytes.
- They need to be delivered in time. An event delivered after it should have taken effect is a potential source of inconsistency and therefore usually triggers some sort of state resynchronization. Discrete media often are more tolerant of delayed events than

continuous media. However, even discrete media suffer from delayed events when the ordering of events is destroyed by an (unexpected) delay.

States have somewhat different properties:

- Their encoding typically requires more bytes. The size of an encoded state can range from around 100 bytes for players in battlefield simulations to several thousands of bytes for complex objects within a shared workspace.
- They are extracted from the model at a certain point in time. The recipient of a state can decode a received state and extrapolate it to accommodate for transmission delay. Usually the maximum time during which the state of a sub-component can be extrapolated is the time between the extraction of the state information and the next event for that sub-component. For example, the state of a shared whiteboard page will remain valid until any one of the users interacts with the page. The state of a car in a distributed car race will remain valid until its controller changes it. In both examples the recipient of the state will be able to extrapolate the state of the sub-component as long as no event has taken effect between the extraction and the extrapolation of the state. Some applications might even be able to automatically incorporate events when the state of a sub-component is extrapolated, thereby extending the time interval during which a transmitted state remains useful.

The ability to request and receive the transmission of state information is an essential functionality for all distributed interactive media. It is required in order for a participant to join an ongoing session and as a means for resynchronization. The state may be extracted and calculated by a centralized server or by a peer application, depending on where the shared state is managed. The transmission of states provides the participants of a session with random access points; this is similar to the transmission of I-frames in an MPEG-encoded video stream.

There exist a number of distributed interactive media that transmit state information only. Instead of conveying information about events, the entity responsible for managing the shared state of a sub-component simply transmits the state of that sub-component after the event has taken place. When this approach is combined with the periodic (re-)transmission of state information it provides effective protection against packet loss (application level reliability). It is commonly used for distributed interactive environments in which the state of sub-components is small. We call this approach *state sharing*.

For media whose size of state information is large, for example, shared workspaces, the frequent transmission of states is not desirable. Applications for these media use a mixture of event and state transmissions, transmitting whenever possible events only. State information is sent only when it is required for the repair of inconsistencies or to accommodate latecomers. We use the term *event sharing* for this information policy.

In summarizing it can be noted that applications for distributed interactive media either transmit states only or states and events. From the different characteristics of states and events it can be derived that an application will likely profit from different quality of service settings for the transportation of each information type. While events are small and need to be delivered in time, states are large and less time sensitive. A common protocol for distributed interactive media needs to take these observations into consideration.

The discussion of the information policy concludes the introduction into the general design issues for distributed interactive media. The only general design decision not considered so far is how to exactly establish consistency for distributed interactive media. Because of the complexity involved in this issue, we will first present some real-world examples of distributed interactive media before investigating consistency in the next chapter.

2.3 Examples

In the following four sections we will outline examples for distributed interactive media that have been chosen to demonstrate the diversity of this media class. For each example we will describe how it fits the abstract media model and discuss the design decisions made by the developers.

2.3.1 The digital lecture board - dlb

The digital lecture board (dlb) [22,25] is a shared whiteboard tool that has been developed in the context of computer-based distance education. Its design was influenced by the developer’s experience with synchronous teleteaching in the TeleTeaching projects of the University of Mannheim [94].

2.3.1.1 Functionality

The digital lecture board provides a page-oriented shared workspace. Each page may contain multiple layers where media objects can be displayed and modified. The media types supported by the dlb range from drawing primitives (text, line, arrow, circle, etc.) to different image formats (GIF, BMP, PCX, Postscript, etc.). Each object can be manipulated independently, e.g., it is possible to resize, move, raise or lower objects. A screenshot of the dlb is depicted in Figure 7.

In order to allow the pages to be prepared in a manner invisible to the rest of the group, the dlb provides a private workspace for individual participants. The prepared pages can be sent to the other participants at any time. Moreover, it is also possible to attach private

annotations to an on-line document. This feature is typically used by students listening to a lecture.

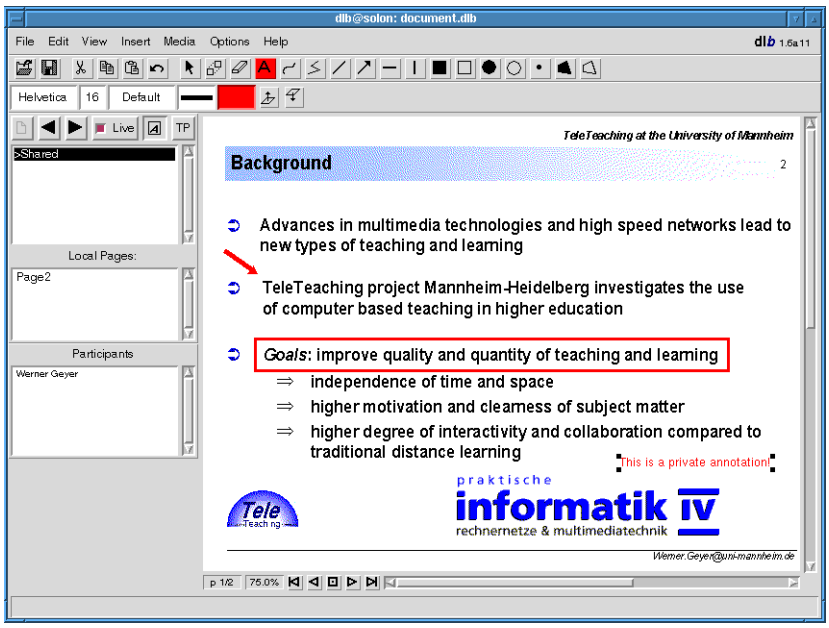

Figure 7: digital lecture board [28]

In addition to the shared workspace, a number of collaboration tools have been realized for the dlb. They are useful to compensate for the lack of social awareness in a distributed lecture. These tools include:

- an online feedback tool, with which students can give anonymous feedback about different aspects of the lecture (e.g., audio quality or difficulty to follow the lecturer),
- a voting tool,
- an attention tool for “virtually raising the hand”, and
- telepointers.

These tools can be considered as separate distributed interactive media that share a common user interface with the main workspace of the dlb. For the sake of simplicity we will just consider the shared workspace aspect of the dlb in the following discussion.

Besides the medium-specific functionality like creating and manipulating objects on the shared workspace, three services have been developed for the dlb that might also be of interest for other distributed interactive media: the ability to record and replay dlb sessions [35], the capability to allow latecomers to join an ongoing session [27], as well as the possibility to use strong encryption [23] to protect the session from unauthorized participants.

2.3.1.2 Media Model

The state of the digital lecture board is defined by the appearance of the objects on the whiteboard pages. This state can only change by user actions, i.e., external events. Internal events do not occur for the digital lecture board since there are no spontaneous or random state changes. Neither does the state of pages change due to passage of time. Thus, the digital lecture board belongs to the class of discrete distributed interactive media (see Figure 1).

The state of the digital lecture board can be broken down into the states of the individual shared whiteboard pages. These can be regarded as the sub-components of this particular distributed interactive medium. Alternatively it is also conceivable to think of the individual objects on the pages as the sub-components of the medium.

The choice of how to partition the state depends on the desired granularity of state transmissions. If it is necessary to transmit the state of a single object on a page then the sub-components should be defined as the objects contained in all pages of the shared whiteboard. If only the state of complete pages needs to transmitted, then it is simpler and more efficient to regard each page as a single sub-component. Since the digital lecture board transmits state information only in units of complete pages (e.g., to support latecomers) the pages should be regarded as the sub-components of this medium.

A dlb session does not have an environment. The developers of the dlb chose to put all the information required to follow a session into the state of the medium. The primary reason for this decision was to make the session self-contained without preliminary download required by the participants. While this allows for a simpler algorithm for joining an ongoing session, it also increases the size of the dlb state. An example of information that could be handled as environment data are the pages presented during a dlb-supported remote lecture.

In summarizing, it can be noted that the abstract media model provides a good fit for the digital lecture board since all important aspects of the dlb can be expressed using the model.

2.3.1.3 Design Decisions

The digital lecture board has been designed for use with either IP-unicast or IP-multicast. Its protocol stack is shown in Figure 8. At the transport level the dlb uses a reliable multicast protocol called Scalable Reliable Multicast Protocol (SMP) [25]. All information regarding the shared workspace is transmitted over this reliable multicast protocol, using transport level reliability. Other tools, such as the telepointers, may also use the unreli-

able transport services offered by the user datagram protocol (UDP) and establish reliability at the application level.

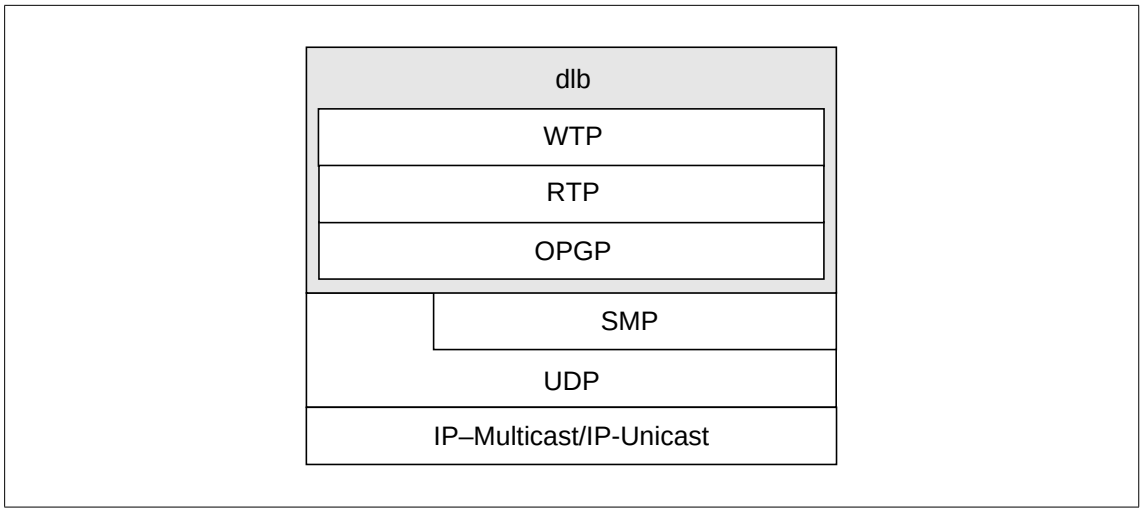

Figure 8: dlb protocol stack [24]

The usage of a reliable multicast protocol at the transport level provides the dlb with a clean and layered architecture. The primitives for transmitting data are simple and straightforward, which is quite important for a complex distributed application. At the same time, the dlb incurs the typical drawback of this approach: If a packet transmitted by one participant gets lost, then all the following packets of this participant will be delayed until the lost packet has been retransmitted and received.

However, for a shared whiteboard this “inefficiency” seems to be quite acceptable since the time needed to generate a new packet by interacting with the shared workspace will usually suffice to deliver any previous operation, including eventual retransmissions. For example, making an annotation on the shared workspace will take a participant a couple of seconds, during which any other action of that same participant can be delivered reliably to all other participants.

The peer instances of the dlb directly exchange events and states with each other. The distributed approach has been chosen to allow for greater scalability and robustness to the failure of individual systems. The dlb relies on the transmission of events whenever possible, since the state of a shared whiteboard page can be rather large. The state of a complete shared whiteboard page is only transmitted to support latecomers or when a page that has been prepared in the private workspace is published to all participants of a session.

States and events are transmitted using the proprietary whiteboard transport protocol (WTP). WTP packets are encapsulated by Real-Time Transport Protocol (RTP) packets that provide the information required to record dlb sessions in synchronization with the audio and video of a distributed lecture. The RTP packets in turn may be encrypted using

Open Pretty Good Privacy (OPGP) before they are handed to the reliable transport protocol.

While the dlb is the most advanced shared whiteboard currently available, the usage of a proprietary application level protocol (WTP) prevents that the services implemented for the dlb can be reused without modifications for other distributed interactive media. This is unfortunate since recording and playback, support for latecomers, and security are frequently needed by many distributed interactive media. Also the use of an unmodified version of RTP seems to be inappropriate since RTP targets continuous media streams produced by a single participant and received by multiple recipients. A common foundation for distributed interactive media in the form of an application level protocol specifically designed for this media class could solve both problems.

2.3.2 The MASH MediaBoard

The MASH MediaBoard is another shared whiteboard application, developed at the University of California, Berkeley. It offers functionality similar to that of the dlb: a number of drawing primitives and the ability to import postscript pages (see Figure 9). In general the MediaBoard offers fewer functions than the dlb. It does not offer typical graphic operations such as grouping and raising/lowering graphic primitives. It also lacks the collaboration tools such as telepointers, attention, online feedback, etc. However, it does realize three interesting new features:

- **Banner windows**. Whenever a user takes action on the shared workspace, the name of the user is shown above the object being modified.
- **Tracking facility**. A user can choose to track another user. If he or she chooses to do so, the local MediaBoard will always try to show the same page as the tracked user.
- **Time-browsing**. A user can time-browse the session by dragging a slider. The MediaBoard then shows the state of the medium at the time indicated by the slider. Time-browsing can be regarded as a (local) recording service for the MediaBoard.

While the MediaBoard and the dlb offer somewhat similar functionality and employ the same media model, they differ radically in the design decisions taken for the application's architecture.

2.3.2.1 Design Decisions

The MediaBoard was developed for use over either IP multicast or IP unicast. In contrast to the dlb it realizes reliability at the application level, avoiding the inefficiency introduced by reliance upon a reliable transport protocol. In this task it is supported by the libsrm library [49], which provides two important services:

- the naming of ADUs, which is required so that the application can address individual ADUs, e.g., to request the retransmission of a specific ADU, and
- basic mechanisms for the detection and repair of lost ADUs in a scalable and application-driven fashion.

Since libsrm is one of the most frequently referenced approaches to realize reliability at the application level, we investigate its two services in more detail.

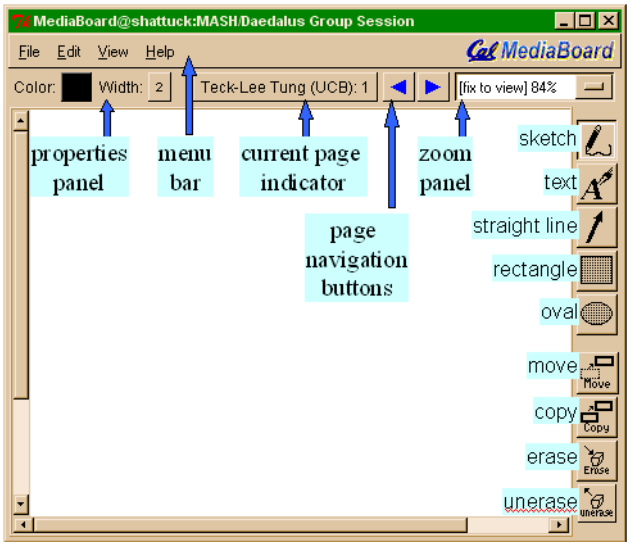

Figure 9: MASH MediaBoard [93]

(a) Scalable Naming and Announcement Protocol

The Scalable Naming and Announcement Protocol (SNAP) [80] enables the application to identify where received or lost ADUs belong. To this end it provides a naming tree for each participant in a session. The nodes and the leaves of that tree represent containers with a unique container ID (CID) and an application level name (page number, object ID, etc.). The containers in the tree may hold ADUs. Each ADU has a field that displays the CID of the container to which it belongs. Since SNAP provides the applications of all participants with a mapping between CIDs and the corresponding application level names, applications can identify where each transmitted ADU belongs (e.g., it may refer to object 2 on page 5). An example of a SNAP tree for a shared whiteboard is shown in Figure 10. The SNAP tree of each participant can be retrieved by other participants on demand. In order to guarantee reliable delivery, SNAP uses the mechanisms of the scalable reliable multicast framework as described below. Meta information about each participants' SNAP tree is regularly announced so that other participants can learn about changes in that tree.

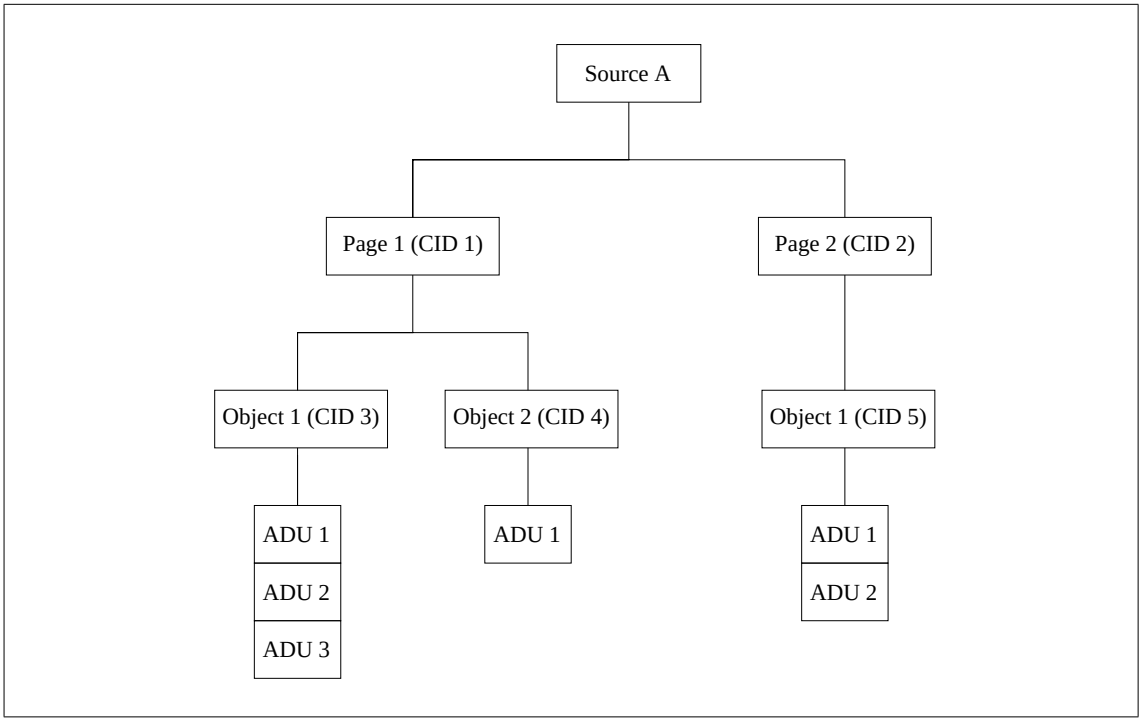

Figure 10: SNAP naming tree

(b) Scalable Reliable Multicast Framework

The Scalable Reliable Multicast framework (SRM) [17] provides basic functionality for scalable application level reliability in multicast environments. A typical time-sequence diagram for SRM-supported loss recovery is shown in Figure 11. The sender (participant 1) of an ADU invokes an *srm_send* method, providing the ADU and the CID of the container to which the ADU belongs. At the receiver side (participant 2) the *srm_recv* method of the application is invoked when the ADU is received. Using this call, the receiving application is provided with the CID and the ADU.

In the event that SRM detects a lost ADU, it invokes the *srm_should_recover* method of the receiving application (participant 3). This call identifies for which CID the loss occurred as well as the sequence numbers of the lost ADUs in that container. It is then up to the application to react to the loss of the ADUs. It can ignore the problem or call the *srm_recover* function of SRM, providing the CID and the sequence numbers of the ADUs which are to be recovered. In the event that a retransmission is requested, SRM uses a scalable NAK (Negative AcKnowledge) algorithm to randomly select one or more participants who should reply to the retransmission request (i.e., any participant can help with the repair of the lost ADUs, not just the sender).

A random timer is used to prevent message implosions that would otherwise occur if all participants who lost a packet immediately requested the retransmission, or if all partici-

pants who are able to retransmit the lost packet immediately did so. In order to preclude message implosions SRM requires that applications wait a random time before transmitting NAKs or repair packets. The value of the random timer is evenly distributed in an interval that depends on the distance (in terms of network delay) to the participant who transmitted the original packet or the NAK. The smaller the distance, the smaller is the upper bound on the timer. Therefore it is likely that applications which are close to the sender of a lost packet or to the sender of a NAK will reply first. Any other participant who could answer with a NAK or repair packet suppresses its own message when it sees that the message has already been transmitted by someone else. An in-depth discussion of SRM's random timer based retransmission process can be found in [17].

An algorithm that prevents message implosion is called a message implosion avoidance or anycast mechanism. It can be used in many situations in which a reply is desired from a group of distributed instances. A performance comparison of different such mechanisms can be found in [19].

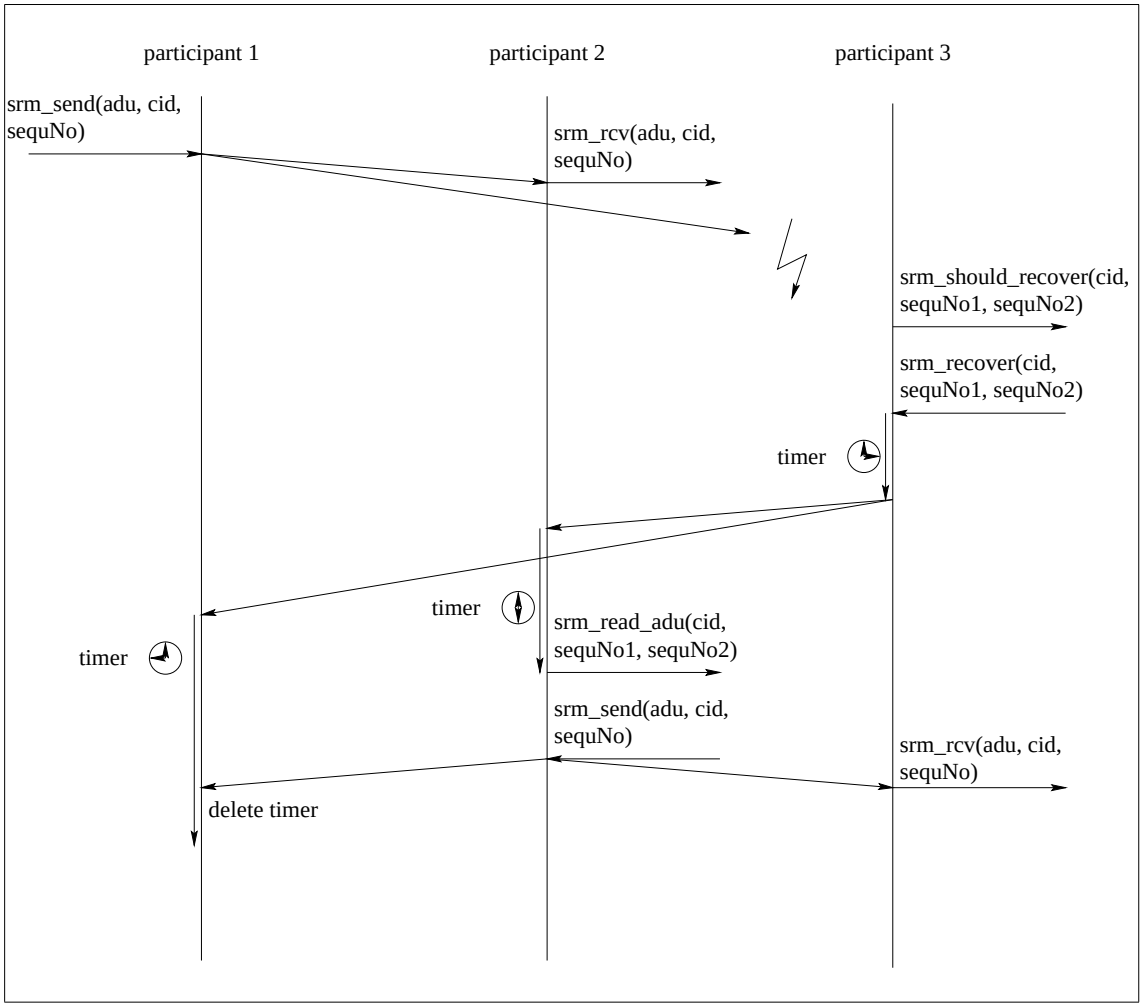

Figure 11: SRM-supported recovery of lost packets

When an ADU needs to be recovered SRM assumes that it got lost for more than one application. For each lost packet it therefore it uses the anycast mechanism to select at

least one of the applications who have requested a recovery of the lost ADU. The selected participant(s) transmits a NAK.

The retransmission of packets is handled similarly, since it is likely that more than one candidate is available for the retransmission of the lost ADU. For each packet that is to be retransmitted, SRM selects at least one candidate by means of the anycast mechanism. A selected application is informed by SRM through an *srm_read_adu* call that contains the CID and the sequence numbers of the ADUs that need to be provided by the application via a regular *srm_send* primitive.

SRM itself does not buffer ADUs but relies on the application to be able to provide the ADU should a retransmission be desired. SRM requires that an application be able to retransmit all transmitted and received ADUs without providing any hint on when these may be discarded. In fact, it is not uncommon for applications using SRM to buffer or to be able to reconstruct ADUs for the whole lifetime of the session.

An important aspect of SRM is that it delegates to the application all decisions that involve repairing packet loss. Also, ADUs are delivered to the application immediately after they have been received, independent of any other transmitted ADUs. While this clearly prevents the inefficiency of reliable transport protocols, it also increases the complexity of the application.

(c) libsrm Usage in the MASH MediaBoard

The MediaBoard is the first application to use the libsrm library. Events such as creating or modifying items on the shared workspace are called *descriptors*. Descriptors are transmitted by the local application to the group of participants using the libsrm functionality. From this we can derive that the MediaBoard manages state in a distributed fashion and that it uses events for its information policy.

The developers of the MediaBoard have chosen to use a *persistent data model*. That is, items on a page are never changed or destroyed. In order to allow objects to be modified or deleted, they introduce shadow descriptors. For example, the shadow descriptor of a delete operation is called a hide descriptor. A hide descriptor targets a regular descriptor (i.e., an object on the shared workspace). The hide operator makes the original descriptor invisible. An empty page in the MediaBoard may therefore contain several objects which have been made invisible.

The developers of the MediaBoard do not discuss in detail the reason for their choice of such an awkward data model [93]. It seems likely that it was inspired by the use of SRM. Since SRM requires that all applications be able to retransmit any received or transmitted ADU at any time, the application has but two choices. Either it buffers every single ADU

transmitted or received, or it is able to recompute each ADU. The latter results in a persistent data model such as the one used by the MediaBoard.

The persistent data model of the MediaBoard also has an impact on the transmission of state information. In the case of a latecomer joining an ongoing session, the SRM mechanism is reused to get the state of individual pages. This is done by sending a NAK for all ADUs belonging to the page whose state is desired. This includes all ADUs that contain information no longer relevant to the state of the page, such as “invisible” objects.

It should be noted that the time-browsing functionality of the MediaBoard necessitates a persistent data model. However, even if this feature were not present, it is unlikely that the data model of the MediaBoard would be different. SRM basically requires a persistent data model in order to work properly. While such a persistent data model may be tolerable for individual applications, it generally leads to inefficiency at the application level (memory usage and complexity). Also it seems inappropriate to use the repair functionality of SRM to convey state information. A dedicated state request and reply can minimize the amount of information required to express the state of a sub-component. In addition, for a continuous distributed interactive medium the simple retransmission of outdated ADUs might not suffice to reconstruct the current state of a sub-component.

Our discussion of the MASH MediaBoard shows that a general application level protocol could improve the overall design of the application, e.g., by providing a dedicated way to request and transmit *state* information. In addition, the SRM interface does not seem to be optimal for the support of distributed interactive media. As we shall show in Section 5.4, only minor modifications would be required to make libsrm more appropriate for this media class.

2.3.3 Distributed Interactive Simulation - DIS

One of the first institutions to explore the use of distributed interactive media was the US Department of Defense. The main objectives of their efforts were the development of military battlefield simulations which can be used for the training of soldiers. The first distributed battlefield simulation - SIMNET (simulator networking) - was developed by Bolt, Beranek and Newman (BBN) between 1983 and 1990. SIMNET connected 11 sites with up to 100 simulators at each site [87].

The protocols used by SIMNET were proprietary and owned by BBN. This limited the deployment of SIMNET to a small number of vendors. In order to change this, the follow-up project - Distributed Interactive Simulations (DIS) - aimed at the standardization of a protocol for military battlefield simulation. The DIS efforts culminated in the IEEE 1278 standard for Distributed Interactive Simulations [39].

2.3.3.1 Functionality

DIS enables a large number of users to participate in a distributed simulation. The users may employ a very broad range of hardware, ranging from high-end workstations to low-cost PCs. Usually, every user controls one entity that participates in a battlefield simulation (e.g., a tank, an aircraft, or a seagoing vessel).

The DIS standard defines 27 protocol data units (PDUs) that can be used by applications to inform each other about actions of the users. Of these 27 PDUs only 4 are of widespread use: entity state, fire, detonation, and collision. Each application is responsible to transmit the PDUs caused by the local user's actions.

The bulk of the DIS data is transmitted with entity state PDUs (ESPDUs). An ESPDU contains the current state of the entity controlled by a user (heading, velocity, etc.). A participating application that receives an ESPDU from another application will model the remote entity according to the data contained in the ESPDU. The remote instance of an entity is called a ghost. In order to reduce the network load, an application will predict state changes of ghosts by means of *dead reckoning* algorithms. A very simple example of dead reckoning would be that a ghost tank with a given heading and velocity will continue moving in that direction with the same velocity. The controller of an entity will issue an ESPDU whenever the state of the real entity differs from that of the ghosts by more than a given threshold. In order to protect against packet loss, ESPDUs are transmitted at least once every couple of seconds, even when the state of the entity is the same as that of the ghosts.

The fire, detonation, and collision PDUs signal that a weapon has been fired, that an explosive ammunition has detonated, or that two entities have collided. The total bandwidth used for those three PDUs amounts only to a very small percentage of the overall bandwidth consumed by DIS.

2.3.3.2 Media Model

Distributed Interactive Simulations belong to the class of continuous distributed interactive media. The state of the medium can change because of events and with the passage of time. DIS exhibits a common characteristic of this media class: state changes caused by the passage of time can be tracked locally by each application without transmitting additional information (dead reckoning).

The sub-components of DIS are the entities controlled by users. Each sub-component can be effected by the full range of events: external events (user actions), internal events (random effects), and pseudo events (two entities colliding). Finally DIS requires that all participants have received the static description of the landscape and buildings before the

simulations begins. This description constitutes the environment of a DIS session. With these characteristics DIS sessions are a perfect example for (continuous) distributed interactive media.

2.3.3.3 Design Decisions

The prime directive for the development of DIS was scalability and robustness to the failure of individual applications. DIS is therefore used exclusively in broadcast or multicast environments. Reliability is established at the application level by the regular transmission of ESPDUs via UPD multicast/broadcast. Lost ESPDUs will eventually be repaired by the data contained in the next ESPDU for the same entity. Entities that stop sending ESPDUs are considered destroyed. The DIS standard uses the term PDU to address transmission units, these DIS PDUs are in fact ADUs as defined by Clark and Tennenhouse.

DIS manages the shared state of all entities in a distributed fashion, using an entity - ghost paradigm with dead reckoning. Most of the time DIS uses only state transmissions (ESPDUs) to synchronize the state of an entity with those of its associated ghosts. However, significant events like explosions, weapon firing and collisions are also transmitted directly.

Since DIS is a standardized protocol for a continuous distributed interactive medium, the question arises whether it might be an appropriate candidate for a common foundation for the whole media class. However, a closer look at the DIS protocol reveals that it is limited by its origin. For example, the ESPDU contains fields for entity location, linear velocity, and orientation. This information might not be relevant to all distributed interactive media (e.g., an animation for distance education would not require these fields). Also DIS implies that state information is small and can be regularly broadcast to all participants.

Because of its specialized nature the DIS protocol cannot be used as a general foundation for distributed interactive media. However, a protocol for military battlefield simulation might become one special instantiation of a more general protocol for this media class. If the same general protocol with a different instantiation were used by the digital lecture board, for example, then the services for one medium could possibly be reused for the other one. A prime example would be a session recorder, which might be able to record military battlefield simulations as well as shared whiteboard presentations.

2.3.4 Distributed Interactive Virtual Environment (DIVE)

The Distributed Interactive Virtual Environment (DIVE) is a flexible toolkit for the development of networked virtual environments [30,18]. It enables the creation of appli-

cations that allow 3D representations of real users (actors) to meet in a virtual world and interact with dynamic 3D objects. A screenshot of a DIVE meeting is shown in Figure 12. The development of DIVE started in 1991 at the Swedish Institute of Computer Science (SICS); work on DIVE is still in progress, with the current version being DIVE 3.3x (July 1999) [86].

2.3.4.1 Functionality

The functionality of DIVE is centered around presenting interactive entities to a group of users. The virtual representation of a user is called an actor. The graphical 3D shape as well as the behavior of entities and actors can be designed in a flexible manner, using 3D primitives and the scripting language Tcl/Tk [73].

Actors and entities are contained in a virtual world which is made visible to the user by a DIVE renderer. A DIVE browser combines the renderer with network support and a user interface to allow multiple users to share a virtual world. The browser can read entities from a local database and publish them to the browsers of other participants in the virtual world. All users can interact with published entities, while DIVE prevents inconsistencies from concurrent access to the entities. For this purpose the published entities are kept in a shared distributed world database.

2.3.4.2 Media Model

DIVE is a continuous distributed interactive medium - entities can change their state because of user actions as well as because of the passage of time. The state of a DIVE world is defined by the state of all entities and actors present in that world. This leads to a partitioning of the medium, with each entity and each actor representing one sub-component. As for DIS, the full range of events is possible (external, internal, and pseudo events).

A DIVE world has properties that remain static over the lifetime of a session. Examples are information on the background and the size of the virtual world. This information can be regarded as the environment of a DIVE world. With the major elements of the abstract media model identified, it is shown that DIVE is another good example of a medium matched by the model.

2.3.4.3 Design Decisions

DIVE is designed to be “inhabited” by a large number of actors at the same time. It therefore employs multicast at the network layer. However, participants who do not have access to the MBone may use a (unicast) proxy to connect to a virtual world.

In order to establish reliability DIVE uses SRM. Unlike the MediaBoard, however, DIVE does not use the SRM loss discovery and repair mechanisms to retransmit lost ADUs. Instead of the missing ADU the current state of the entity for which the ADU got lost is transmitted. The responding application can directly extract the desired information from the entity (e.g., its position and orientation). This prevents the problems associated with the persistent data model used by the MASH MediaBoard, where each original ADU must be available for retransmission at all times. The problem with the DIVE approach is that it is feasible only when the state of the sub-components is small. Otherwise a single lost (small) event might trigger the retransmission of the whole (large) state.

Figure 12: DIVE screenshot [86]

The shared state of entities and actors is managed in a distributed fashion. The concurrent modification of entities is prevented by a floor control mechanism that allows only one user to interact with an entity at any given time. Actors who do not have the floor of an entity must delay their action until they receive the floor. When an actor changes the state of an entity, then the new state of that entity is transmitted directly to all peer DIVE browsers. State changes that occur because of the passage of time are handled by dead-reckoning, as in DIS. Since the state information is usually small, DIVE does not transmit events.

A common application level protocol for distributed interactive media can derive an important idea from DIVE: transmission of the current state of a sub-component is sometimes more appropriate than the retransmission of (outdated) event information. The main benefit that a common application level protocol for distributed interactive media would provide to DIVE is that DIVE could use services available for that protocol (e.g., recording, encryption).

2.4 Chapter Summary

In the first part of this chapter we defined the term distributed interactive medium and presented an abstract model for this media class. Important elements of this model are (shared) state, events, sub-components, the environment, and the application. We divided distributed interactive media into two groups: those which change their state because of events and the passage of time (continuous) and those which are time invariant (discrete).

The second part of this chapter contained a discussion of important design decisions that have to be made by developers of applications for distributed interactive media. These included:

- the choice of the network protocol technique (unicast, broadcast, multicast),
- how to achieve reliability (transport level reliability vs. application level reliability),
- designating responsibility for maintaining the shared state of sub-components (centralized vs. distributed approaches), and
- the kind of information to be exchanged between applications (events and states vs. states only).

In the last part of this chapter we presented four examples of distributed interactive media: two shared whiteboards, a standard for military battlefield simulations, and a general-purpose networked virtual environment. We showed that the abstract media model fits all four examples and explained how each example would benefit from a common application level protocol for distributed interactive media.

In the following chapter we will deal with consistency for distributed interactive media. We will answer the question of how the local copies of a distributed interactive medium’s state can be kept ‘reasonably’ similar for all participants.

3 Consistency in Distributed Interactive Media

In this chapter we investigate how consistency can be ensured for distributed interactive media. Existing consistency criteria for *discrete* distributed interactive media are briefly outlined, and it is shown that algorithms that enforce these criteria do not work for *continuous* distributed interactive media.

In order to allow a thorough discussion of the problem, a formal definition of the term *consistency* in the continuous domain is given. Based on this definition we show that an important trade-off relationship exists between the responsiveness of the medium and the frequency with which short periods of inconsistency occur for the shared state of the medium.

Until now this trade-off was not taken into consideration for consistency in the continuous domain, thereby severely limiting the consistency-related fidelity for a large number of applications. We show that for those applications the fidelity can be significantly raised by deliberately decreasing the responsiveness of the medium. This concept is called *local lag*. It enables the distribution of continuous interactive media that are more vulnerable to short-term inconsistencies than, e.g., battlefield simulations.

3.1 Consistency in Discrete Distributed Interactive Media

As defined in the previous chapter, a discrete distributed interactive medium is a medium that changes its state solely because of events. In the literature about consistency such events are commonly called *operations*. In the following we will use the terms event and operation synonymously.

Generally speaking, consistency for discrete distributed interactive media entails finding a ’correct’ sequential order of all the operations issued by the participants of a session and making sure that, at all participating sites, the state of the medium looks as if all operations had been executed successfully in that particular sequential order. To be more precise, we use the terms *causal ordering relation* and *dependent operation* from Sun and Ellis [92,91,15], which are derived from Lamport’s work on virtual clocks [48]:

*"**Definition 1**: Causal ordering relation "→". Given two operations O_a and O_b generated at sites i and j, then $O_a \rightarrow O_b$, iff: (1) i=j and the generation of O_a happened before the generation of O_b, or (2) i≠j and the execution of O_a at site j happened before the generation of O_b, or (3) there exists an operation O_x, such that $O_a \rightarrow O_x$ and $O_x \rightarrow O_b$.*

***Definition 2**: Dependent and independent operations. Given any two operations O_a and O_b. (1) O_b is dependent on O_a iff $O_a \rightarrow O_b$. (2) O_a and O_b are independent (or concurrent), expressed as $O_a || O_b$, iff neither $O_a \rightarrow O_b$, nor $O_b \rightarrow O_a$."*

In order to illustrate these terms consider the session depicted in Figure 13: in this session the three operations O_1, O_2 and O_3 take place. The causal ordering relation for this example is $O_2 \rightarrow O_3$ since O_2 is executed at site 2 before O_3 is generated. An example for the semantics of O_2 could be that the user at site 2 writes a question onto a shared whiteboard page. In reaction to this, the user at site 3 types an answer to this question as operation O_3. From the ordering relation it can be derived that O_3 depends on O_2, while O_1 and O_2, as well as O_1 and O_3, are independent of each other.

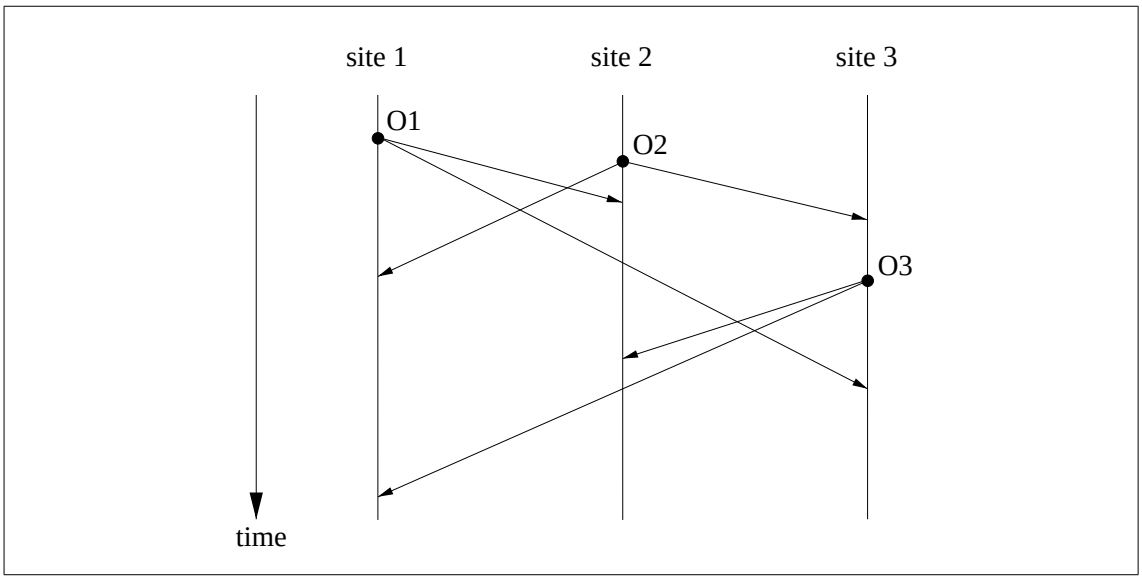

Figure 13: Example of a session with a discrete distributed interactive medium

Based on the definition of the causal ordering relation, there are two consistency correctness criteria that are generally used to define the term consistency for discrete distributed interactive media [92,91,15]:

(1) **Convergence property**. After all operations have been executed the state of the discrete interactive medium is identical at all sites.

(2) **Precedence property**. For any two operations O_a and O_b, if $O_a \rightarrow O_b$, then at each site O_a will be executed before O_b.

The first criterion ensures that all users will eventually see the same state of the interactive medium after all operations are completed, it is the fundamental consistency crite-

rion. A violation of this criterion would result in different states of the distributed medium and therefore render it useless.

The second criterion makes sure that dependent operations are executed in order. If (2) were not required, the user at site 1 might see the result of dependent action O_3 before the result of action O_2 was visible. This would be confusing to the user since the causality of the operations would not be preserved. In the question/answer example the precendence property guarantees that the user at site 1 sees the question before the answer becomes visible.

In order to ensure that the consistency correctness criteria are not violated a number of different approaches exist [66], ranging from a strict single user floor control policy to more sophisticated algorithms such as operational transformation [15,91]. Since at least the convergence property is also desirable for continuous distributed interactive media, the question arises whether the approaches that ensure this criterion for discrete media could be reused in the continuous domain. Unfortunately the answer to this question is “No”.

The reason why the approaches for discrete media fail when they are applied to continuous media is that consistency in continuous distributed interactive media is not only about finding a correct sequence of operations and making sure that at each site the result of all operations looks as if the operations had been executed in that sequence. In addition it requires that the result of all operations looks as if the operations had been executed at the *correct point in time*. The algorithms for establishing consistency in the discrete domain can therefore be regarded as insufficient for the continuous domain.

To illustrate this problem let us examine a very simple example. Given is a session involving a discrete distributed interactive medium. This session is attended by two users U_a and U_b. Now let U_a perform an operation on the medium. This operation will be executed first on U_a’s copy of the interactive medium and some time later (because of the network transmission delay) on U_b’s copy of the medium. In a discrete interactive medium a single operation cannot result in inconsistencies. Therefore no consistency algorithm from the discrete domain will take any special actions in this example.

Now let us transfer this example to the continuous domain: In a distributed simulation, attended by two spatially separated users, U_a and U_b, a train is approaching a switch. The switch can be operated by any user participating in the session. Imagine that just before the train arrives at the junction, U_a operates the switch. In the copy of U_a’s simulation the operation takes place immediately. However, the information about U_a operating the switch will arrive at the copy of U_b’s simulation at a later point in time. Applying the operation at this point in time to U_b’s copy could lead to an inconsistent state because the train might have already passed the switch in U_b’s copy of the simulation. As explained

above, methods for ensuring consistency in discrete media will take no actions to correct this problem. This reveals the core reason why the mechanisms for consistency used in the discrete domain are not sufficient for continuous media, namely because they neglect the problem of executing operations at the correct point in time.

3.2 Consistency in Continuous Distributed Interactive Media

As defined in the previous chapter, a continuous distributed interactive medium is a distributed medium that changes its state in response to (user initiated) operations as well as because of the passage of time. This definition implies that the medium has access to a physical clock that can be used to measure the progress of time. In the scope of this work we assume that the physical clocks of all the participants are reasonably synchronized - typically with a deviation of less than 50ms (achievable using NTP [68] or GPS clocks). Furthermore we require that the correction of clock drift is done in a way that does not result in decreasing the value of a physical clock, e.g., it can be done by slowing down/ halting the physical clock for a period of time instead of simply decreasing its value. If a given physical clock cannot be controlled in this way, some software may be required that adapts the reading of the physical clock so that it obeys this restriction. In the following we assume that this requirement is met.

In continuous distributed interactive media a user-initiated operation needs to be executed at a specific point in time. Typically the timestamp of a physical clock is used to denote this point in time. Similar to consistency in the discrete domain, consistency for continuous media is about finding a correct order of all operations. However, in addition it must be guaranteed that at all sites, the state of the interactive medium after these operations have been executed is the same as if the operations had been executed in the correct order *at the time identified by their timestamps*.

In order to be able to discuss this in a more formal way we define a partial ordering on the user-initiated operations as follows [64]:

Definition 3: Partial physical time ordering relation “<”. Given two operations O_a and O_b with timestamps T_a and T_b, then O_a is said to happen before O_b, expressed as $O_a<O_b$, iff $T_a<T_b$.

Definition 4: Simultaneous operations “≈”. Any two operations O_a and O_b with timestamps T_a and T_b are said to be simultaneous, expressed as $O_a \approx O_b$, iff $T_a=T_b$.

Note that this definition is based on physical clocks, as opposed to logical-clock based methods for discrete media. The partial physical time ordering relation can be extended to become a complete physical time ordering relation by using an arbitrary tiebreaker for

simultaneous operations. An example of such a tiebreaker could be the IP address, to break ties for simultaneous operations from two different participants, in combination with a counter, to break ties of the same participant.

Definition 5: Complete physical time ordering relation "<<". Given two operations O_a and O_b with timestamps T_a and T_b and tiebreakers B_a and B_b, then $O_a << O_b$, iff (1) $T_a < T_b$, or (2) $T_a = T_b$ and $B_a < B_b$.

Now we are able to define the consistency criterion for continuous distributed interactive media as follows:

(3) **Consistency Criterion for Continuous Distributed Interactive Media**. A continuous distributed interactive medium is *consistent* if after all operations have been executed at all sites, the state of the medium at all sites is identical to the state which would have been reached by executing all operations in the order given by the complete physical time ordering relation at the physical time denoted by the timestamps of the operations.

It is noteworthy that a precedence property is not part of the correctness criterion for continuous distributed interactive media. As we shall see later, the precedence property will be part of the fidelity criteria for the assessment of algorithms that enforce the consistency correctness criterion.

The consistency criterion for the continuous domain is illustrated in Figure 14. The physical clocks of all three sites are slightly out of synchronization, the dashed lines identify the points in time with the same reading of the physical clock at each site. The small filled circles indicate the time when an action is executed. In the example the following ordering of operations applies: O2<O1, O2<O3, and O1≈O3 as well as O2<<O1, O2<<O3 and O1<<O3 (if the site number is used as the tiebreaker). Figure 14 shows a somewhat unrealistic situation in which all operations are always known at all sites at the time denoted by their timestamp. This situation, however, is important, since the meaning of the consistency criterion is that at some time after the last site has received all operations (for example at t=x), the medium will look as if the operations had been executed as depicted in Figure 14.

By guaranteeing that the state of the medium will be identical at all sites after all operations have been resolved, the consistency criterion is the most important prerequisite to making sure that a continuous distributed interactive medium is actually usable. However, even if the consistency criterion is guaranteed by a proper algorithm, it is still possible that consistency-related, transient artifacts occur. In addition to the basic consistency criterion, we therefore define two additional fidelity criteria:

(4) **Avoidance of short-term inconsistencies**. If an operation with timestamp T is not executed at a site i at physical time T, then this operation is said to have caused a short-

term inconsistency of the interactive medium at site i. Ideally, short-term inconsistencies should be non-existent.

Short-term inconsistencies occur when an operation issued at site j arrives at site i after the time denoted in the timestamp of the operation. Because we assume that an algorithm exists to ensure the consistency criterion (3) a late arrival means that the state of the interactive medium at site i needs to be repaired. In the train example this would mean moving the train from the wrong position on one branch of the tracks to the right position on the other branch of the tracks. The fidelity criterion for the avoidance of short-term inconsistencies includes the meaning of the precedence property (2) as it has been specified for discrete media since a reversal of the precedence of operations can only be caused by the late arrival of remote operations.

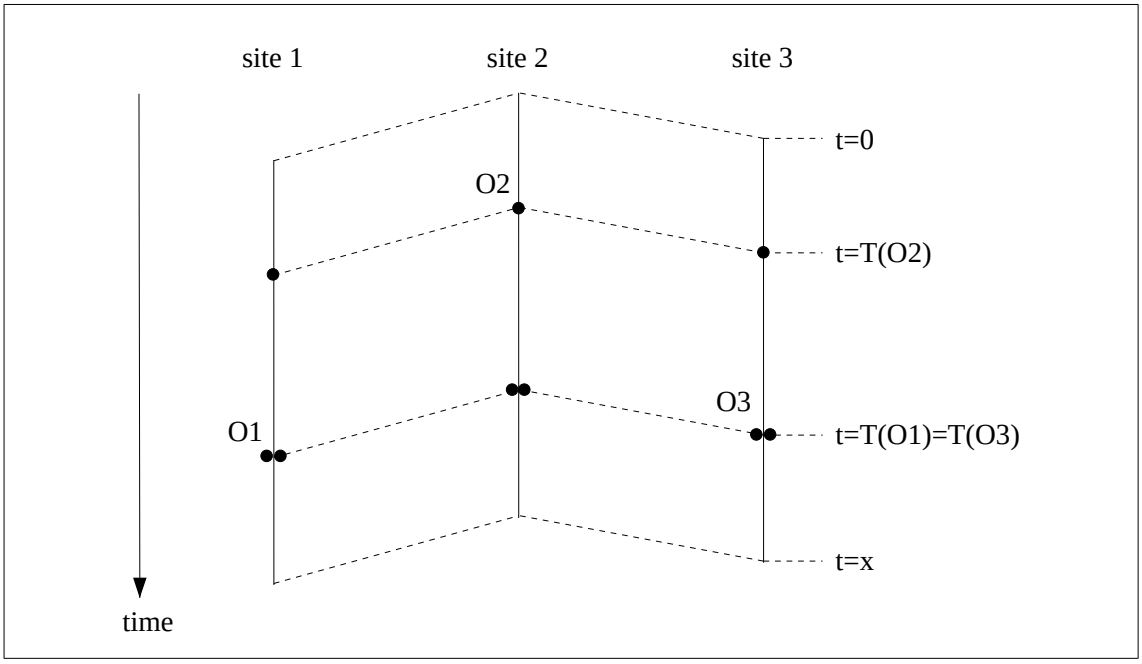

Figure 14: A consistent continuous distributed interactive medium

Short-term inconsistencies basically have three implications: They may cause dependent operations to be executed in the wrong order, they usually result in visual artifacts (the train “jumps” from one position to a different one), and a user might take actions that are based on an inconsistent state of the medium. An example for the last problem is the following situation: Assume that in the train example U_b pulls the brakes of the train after it has passed the switch but before the operation from U_a has arrived. In this case, U_b intends to stop the train because it is going in the wrong direction. After U_b has issued this operation, the operation from U_a arrives (late). In order to satisfy the consistency criterion (3), the train will jump to the correct position. The net result of both operations is that the train will stop while it is headed in the right direction, which is a clear violation of the intention of the users.

(5) **Low response time**. The response time of an interactive medium is the time between the physical time at which a user issues an operation and the timestamp of that operation. Ideally the response time should be zero.

If the response time exceeds a certain threshold the users will notice that a delay exists between the time an operation is issued and the time an operation is executed. This will result in an ‘unnatural’ behavior of the medium. In the context of this work we also use the term *responsiveness* synonymously to response time.

While it would be desirable to be able to guarantee that no consistency-related artifacts occur in a continuous distributed interactive medium, this is not possible. The optimization of the response time and the avoidance of short-term inconsistencies are conflicting goals.

Part (a) of Figure 15 shows an example where the responsiveness is optimal (response time = 0). In this case the operations issued by the users take effect immediately. The points in time when O_2 and O_3 should take effect are indicated by the dotted line. As can be seen, only the user issuing the operation does not experience short-term inconsistencies. Any other user will have a short-term inconsistency for that operation due to the transmission delay of the remote operation.

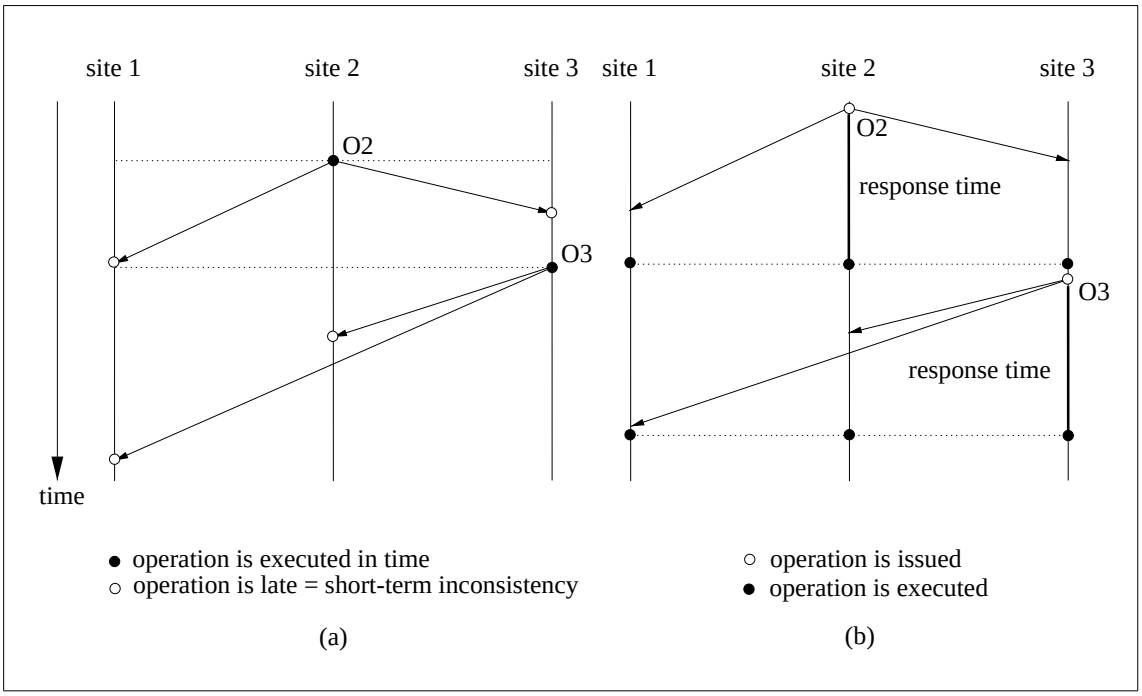

Figure 15: Short-term inconsistency vs. response time: the trade-off

Part (b) of Figure 15 shows the same situation optimized for the short-term inconsistency criterion. Here short-term inconsistencies are avoided at the cost of increasing the response time for each operation to the maximum transmission delay between any two participants. Note, that the execution of O_2 and O_3 is delayed at the originating sites until

all participants have a chance to act simultaneously. An event received at a time before it should be executed does not pose a problem. It should be buffered by the receiver until the time denoted by its timestamp is reached.

An observant reader might have noticed two peculiarities:

1. What if the time deviation between sender and receiver happens to compensate for the transmission delay?
2. In an environment where network packets might get lost, how exactly can we calculate the maximum transmission delay for an operation?

The answer to the first question is that the time deviation might indeed compensate for the network delay between a given sender and receiver. This makes it possible for the sender to have a response time equal to zero while the receiver does not experience any short-term inconsistencies.

However, further reflection reveals that this works only when the sender and the receiver do not switch their roles. As soon as the previous receiver becomes the sender short-term inconsistencies will occur because of the large time deviation which now prevents that any operation arrives at the new receiver in time.

The second observation arises from the problem that in an environment where packet loss occurs, it is impossible to define an upper bound on the delay which an operation needs to arrive at a remote site. After all, the same packet might get lost over and over again as it is being retransmitted by the sender. This leads us to the conclusion that while a reduction of short-term inconsistencies is desirable, a guaranteed prevention is not possible in real networks.

With the definition of the convergence property in place and the trade-off between the two fidelity criteria discussed, we will now investigate a frequently used algorithm for achieving consistency in continuous distributed interactive media.

3.3 Dead Reckoning

An approach that is commonly used to guarantee that the consistency criterion is satisfied in distributed virtual environments (e.g., multi-user virtual reality and battlefield simulations) is a combination of state prediction and state transmission. In this approach the application “knows” how the objects in the virtual environment should behave over time. Examples are a plane that will fly straight at constant speed or a projectile that will take a course dictated by the physical law of gravity and preservation of impulse. As mentioned in the previous chapter, the ability to predict the behavior of objects is called *dead reckoning*.

Each object for which dead reckoning is performed has a single controlling application, e. g., for a plane this will be the application of the pilot. The controlling application is responsible to inform peer applications when the state of the object deviates by more than a certain threshold from the predicted state. In the event of a deviation, the controlling application transmits the complete state of the affected object to all peers. Upon receiving this information, an application discards the outdated state and uses the new state to perform dead reckoning for this object. In order to be able to use this approach over unreliable transport services, the controller of an object additionally transmits the state of the object at regular intervals in the form of so called heart-beat messages.

User initiated operations are immediately applied to the local state of the affected object. The object will thus be put into a state that differs significantly from the predicted state, thereby requiring the controlling application to transmit the new state to its peer applications. Because of this behavior the dead reckoning approach optimizes the response time criterion while each action will result in a short-term inconsistency at each remote site.

While this approach has a number of important advantages such as robustness and scalability, its disadvantages limit its applicability:

- It causes the maximum number of short-term inconsistencies.
- It requires that the state of each object be transmitted for each operation. This could lead to a massive need of bandwidth, especially when the size of the state of the object exceeds a couple of bytes.
- Only one controller for each object is possible. This prevents collaborative actions, such as two users moving a single object.

In Chapter Two we introduced two examples that employ dead reckoning: DIS and DIVE. In a typical battlefield simulation the state of the relevant objects is small (position and velocity) in size and relatively easy to predict, while each object will have only one controller (e.g., the pilot of a plane, the driver of a car, etc.).

The effect of visible artifacts can be reduced by using intelligent algorithms to repair the state instead of immediately adopting the new state. For example, after a plane has changed its direction a remote application can calculate an alternate flight path which will eventually bring the plane back into the correct state (rather than jumping immediately to the correct position). Because of these attributes the state prediction and transmission approach is adequate for battlefield simulations and some distributed virtual environments.

However, this approach becomes less appropriate if an interactive medium has one or more of the following properties:

- the state of objects is complex,
- the future behavior of an object cannot be predicted from its past behavior,

- visual artifacts cannot be concealed,
- true collaboration is important, or
- actions based on an inconsistent state are critical.

For several applications it might therefore be better to have a closer look at the trade-off between responsiveness and short-term inconsistencies before simply maximizing responsiveness. In the following section we will discuss how to exploit the trade-off to improve the consistency-related fidelity of continuous distributed interactive media.

3.4 Local Lag

The concept of *local lag* is simple: Instead of immediately executing an operation issued by a local user, the operation is delayed for a certain amount of time before it is executed. To use the terms from section 3, this means that the value of an operation's timestamp will be greater than the point in time when the operation is issued by the user. The delay that is introduced for the local user in this way is called *local lag*. As depicted in Figure 15 (b), if the value for local lag is sufficiently high, then it can reduce the number of short-term inconsistencies.

The concept of local lag is related to the work of Cristian et al. [10] were a total ordering on broadcast messages from multiple senders is established in order to achieve atomic broadcast. In this approach operations get a timestamp which is greater than the point in time the operation is issued by a sender. Operations are executed at the time denoted by that timestamp. In order to calculate the timestamp Cristian et al. assume that an upper bound on the transmission delay can be given, so that operations never arrive after a certain time.

In contrast we assume that such an upper bound does not exist and that we have to cope with operations that arrive late. The local lag approach therefore differs in two main aspects:

- The amount of local lag is calculated in a different way. It is adapted to the specific needs of consistency for continuous distributed interactive media.
- There exist repair mechanisms should an operation arrive late.

3.4.1 Determining a Value for Local Lag

Clearly it would not be desirable to move from one extreme, where the response time is zero but short inconsistencies are very frequent, to the opposite extreme, where almost no short-term inconsistencies occur but the response time is unacceptably high. Therefore it is important to consider both consistency-related fidelity criteria. For a given continuous distributed interactive medium we propose to do this in three steps: (1) determine a minimum value for the local lag needed to prevent a significant amount of short-term

inconsistencies, then (2) determine the highest acceptable response time, and finally (3) choose a value for the local lag.

3.4.1.1 Determining a Minimal Value for Local Lag

In order for local lag to be useful it needs to significantly reduce the number of short-term inconsistencies for all participants. Moreover, short-term inconsistencies should be reduced for each sender/receiver pair. We therefore propose to use the maximum average network delay between any two participants as the minimum amount of local lag. Typical values for network delays (assuming an uncongested unicast network) are: less than 1ms for a LAN, 20 ms within a European country, 40 ms within a continent, and 150 ms for a worldwide session. Choosing the maximum average network layer delay as the minimal value for local lag implies that short-term inconsistencies will occur only if packets get lost or the delay jitter becomes significant because of network congestion.

If the clocks of the participants are not completely synchronized, the maximum offset between any two clocks needs to be added to the result. This is necessary because the sender of an operation might be behind in time relative to the recipient of the operation. If this time deviation were not accounted for, the time deviation plus the transmission delay might be larger than the local lag which could result in frequent short-term inconsistencies. This problem is shown in Figure 16 where site 1 is ahead in time of site 2. Even though the operation O2 from the user at site 2 has a local lag greater than the maximum network delay, a short-term inconsistency occurs at site 0 due to the deviating clocks. Typical values for time deviation of computers using NTP are 20-50 ms if the NTP server is reached via WAN and less than 10 ms if the NTP server is part of the same LAN.

3.4.1.2 Determining the Highest Acceptable Response Time

The maximum local lag is dependent on how much response time a user can tolerate for a given interactive medium. In order to determine this value, it is necessary to conduct perceptional psychological experiments. Ideally the experiments will deliver two values: the first is the amount of local lag for which a user cannot notice the delay. The second value is the amount of local lag which can be tolerated by the user, i.e., the delay can be noticed but it is not disturbing. It seems to be reasonable to use the second value as the maximum amount of local lag for an interactive medium.

Preliminary experiments suggest that a local lag of up to 80 ms is not noticeable by the users independent of the operation and the medium. The value at which local lag gets to

be disturbing seems to depend heavily on the operation, ranging from 300-400 ms for simple click operations to 100-150 ms for drag operations.

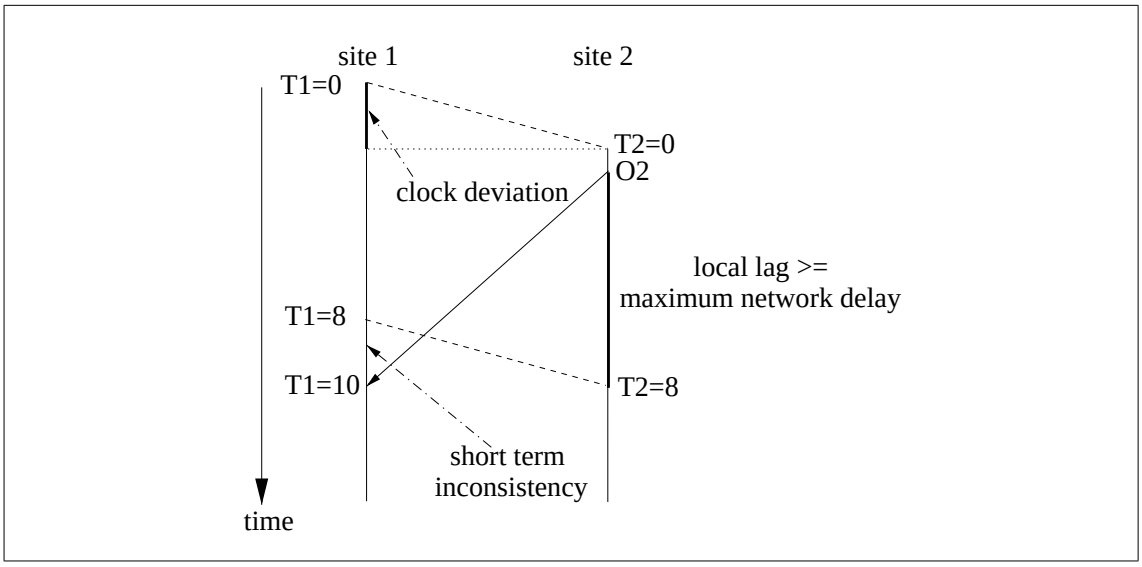

Figure 16: Problem with deviating clocks

3.4.1.3 Choosing a Value for the Local Lag

The previous two steps can result in two main cases: either the minimum value for local lag is smaller than or equal to the highest acceptable response time, or it is not. In the first case a value between the minimum value for local lag and the highest acceptable response time can be chosen. This choice should be based on additional criteria such as whether additional time is required to receive forward error correction packets, etc.

If the highest acceptable response time is lower than the minimal useful value for local lag, then a true trade-off situation occurs. Given the values mentioned above this should be relatively rare, though it might happen for very demanding media and applications. In this case it is necessary to lower the fidelity of both criteria in a way which provides the best overall fidelity to the user. Most likely this will require another set of perceptional psychological experiments.

3.4.2 Repairing Short-Term Inconsistencies

If the amount of local lag is chosen well, it eliminates a significant amount of short-term inconsistencies. However, it cannot completely prevent all short-term inconsistencies since operations might still arrive late or not at all (e.g., if the network is congested or a transport packet gets lost). It is therefore necessary to have a mechanism that will repair the state in these cases. In the following we will summarize three possible approaches to repairing inconsistent shared state.

3.4.2.1 State Prediction and Transmission

An obvious approach is to combine local lag with the mechanism for state prediction and transmission. Instead of immediately executing a local operation on an object, the operation is delayed using the value for local lag: The operation gets a timestamp equal to the current time plus the value for the local lag. However, when the user issues the operation, the new state for the object at the time denoted by the timestamp of the operation is calculated immediately. This state includes the operation and bears the same timestamp as the operation. The calculated state is immediately transmitted to all participating sites.

When the physical time identified by the timestamp is reached, the site where the operation was issued will execute the operation. The remote sites that by then have received the new state for the object will start using the new state for dead reckoning without encountering short-term inconsistencies.

A site that receives the new state late, e.g., because of an unexpected transmission delay, experiences a short-term inconsistency. This inconsistency is repaired just as for regular dead reckoning by replacing the local copy of the object's state with the state that was transmitted by the controller of the object.

The combination of local lag with state prediction and transmission retains two limitations from the original approach without local lag: only one controller per object is possible and the state of an object must be small (since the state is still transmitted for each operation). In the following approach we will allow the state of objects to be complex.

3.4.2.2 Requesting States

In order to allow objects to have a complex state of a size significantly larger than the 200 - 300 bytes used for states in the DIS protocol, it is important that the state transmissions be restricted to cases when a short-term inconsistency needs to be repaired. In all other cases only the operation itself will be transmitted.

In this approach a recipient of a transmitted operation checks the timestamp of the operation. If the operation has arrived in time, it will be buffered until it is due for execution. Only if the operation is late will the state be requested to repair the short-term inconsistency.

An algorithm realizing this approach needs to be able to identify those participants that have the correct state of the object. In addition it must be guaranteed that at least one participant can determine the correct state of the object. A possible solution to these problems is to allow only a single controller at a time for each object. Since only the controller is able to issue operations for the object, the controller of an object will always have the correct state. It is the task of the controller to reply to requests for the state of the

object. The controller of an object can change, as long as it is guaranteed that all operations (and the repair of short-term inconsistencies induced by these operations) of the preceding controller have been executed before a different participant becomes the new controller of the object.

The main limitation to this approach is that no true collaborative operations are possible, i.e., two users are not able to interact with the same object at the same time (unless some more sophisticated algorithm is used to determine who will reply to a state request). Also, the repairing of an inconsistent state requires an additional roundtrip for the state request/reply. Therefore the requesting states approach is useful primarily for applications where simultaneous access to a single object is not required and the likelihood of an inconsistent state is rather low.

3.4.2.3 Time Warp

This method is an adoption of the time warp algorithm used for optimistic parallel discrete event simulation [20] [41]. In time warp the state of each object does not need to be transmitted, even when a short-term inconsistency occurs. Instead local information is used to repair the state. Additionally time warp allows an arbitrary number of users to interact with the same object at the same time.

The fundamental idea of time warp is that each participant saves the state of the interactive medium at certain times. All operations up to a certain point in time in the past are kept in a log. When a short-term inconsistency occurs, the interactive medium is rolled back to the last saved state of the interactive medium before the operation should have taken place. Then the operation that caused the short-term inconsistency is inserted into the log. After that the medium is played in fast forward (“time-warped”) mode, executing the operations from the log at appropriate times until the current time for the medium is reached, and operation is resumed at the normal pace. Only the end result of this operation should be visible to the user.

Choosing the points at which to save the state depends on the application and the medium. If the state is saved too frequently, it might effect the visual quality of the presentation and use a large amount of memory. If the state is not saved frequently enough, many operations will have to be performed when a short-term inconsistency occurs, extending the time during which the inconsistency is visible to the user.

The main drawback to this approach is that it requires a sophisticated application that is able to handle the process of time-warping. In addition, if the state is complex, then the time warp approach might consume a large amount of memory to save the states, as well as place a heavy load on the computer when a time warp has to be executed.

In summary it can be said that the choice of the amount of local lag as well as the choice of a repair mechanism for short-term inconsistencies depend heavily on the application and the medium itself.

3.5 Consistency Support in an Application Level Protocol

Reflecting on the discussion of consistency for distributed interactive media, two important questions remain open: What kind of information should a common application level protocol provide in order to support consistency? And, is it reasonable to use the same application level protocol for both discrete and continuous distributed interactive media, even though they have different consistency requirements?

The answer to these questions is as follows: Even though the algorithms for the establishment of consistency may differ for continuous and discrete distributed interactive media, the information required to realize the algorithms is the same: both sub-classes need the timestamp of a clock. The only difference is that in the case of a discrete medium this may be a virtual clock while for continuous media it must be a physical (synchronized) clock. In addition, if the synchronization of a discrete distributed interactive medium with a continuous medium like audio or video is desired, then the use of a physical clock is also required for the discrete domain. From a consistency perspective it is therefore appropriate to use the same application level protocol for continuous and discrete distributed interactive media.

3.6 Chapter Summary

In this chapter we investigated the problem of consistency in distributed interactive media. It was shown that algorithms for consistency in the continuous domain differ significantly from those used for discrete media. In order to systematically approach the problem, a formal definition of the term consistency in the continuous domain was deduced from the special characteristics of the continuous media class. Based on this definition an important trade-off relationship between responsiveness and the occurrence of short-term inconsistencies was examined. We proposed to make use of the knowledge of this trade-off relationship in order to increase the consistency-related fidelity of the medium. This can be done by deliberately increasing the response time in order to decrease the number of short-term inconsistencies, leading to the concept of local lag. Finally we reasoned that the same application level protocol can be used for continuous and discrete distributed interactive media since the information required to establish consistency is the same, even though the consistency mechanisms may differ significantly.

This chapter concludes the general reflection on common attributes of the distributed interactive media class. In the following chapter we will present an application that realizes a continuous distributed interactive medium. The application was developed in the context of this work and is used as proof of concept in the remainder of this thesis.

4 TeCo3D - Sharing Interactive 3D Models with Dynamic Behavior

In this chapter we introduce the 3D telecooperation tool TeCo3D. TeCo3D was developed in the context of this work. It provides a shared workspace for interactive and dynamic 3D models [58]. These models are specified using the Virtual Reality Modeling Language (VRML). The key feature of TeCo3D is the ability to share 3D models that have not been explicitly developed for use in a distributed environment: Existing 3D models, as well as those produced using standard software, such as CAD programs and animation packages, can be utilized for telecooperation, teleteaching, and telepresentation. Since the 3D models can change their state because of the passage of time and because of user actions, TeCo3D is an application for a continuous distributed interactive medium. It is used as an example medium and as proof of concepts throughout the remainder of this thesis.

In order to illustrate how dynamic and interactive 3D models are described in a platform-independent manner, a short introduction to the Virtual Reality Modeling Language is given at the beginning of this chapter. It is followed by a discussion of how to share VRML content that was not specifically designed for use in a distributed environment. This discussion leads to an architecture for TeCo3D and reveals that two important issues need to be solved: the distribution of user interactions, and the transparent access to the state of arbitrary VRML content. These two issues are discussed in depth. Finally TeCo3D is compared to related approaches such as application sharing.

4.1 Virtual Reality Modeling Language - VRML

VRML is the “file format for describing interactive 3D multimedia on the Internet” [96]. A VRML file contains the platform-independent description of 3D objects and their behavior; this description is called a ‘*world*’. In order to interact with the content of a VRML file, the user needs a *VRML browser* - just as a web browser is required to view HTML files. VRML browsers are available as either stand-alone solutions or in the form of plug-ins to the most common HTML browsers. During 1998, VRML reached a large

momentum, mainly because of the release of the VRML ISO standard [96] and the inclusion of VRML browsers as standard components of all major web browsers. This makes VRML the preferred description language for importing 3D models into a 3D telecooperation tool.

We introduce VRML by examining three examples. The first one deals with the description of a static object, the second shows a simple animation, and the third demonstrates the inclusion of a programming language (Java) into VRML models. This section can only give a limited overview of the capabilities of VRML. Detailed introductions can be found in [2] and [31]. An excellent summary of the Java-VRML interface is presented in [7].

4.1.1 Static World

The basic building blocks of VRML are *nodes*. Each node contains a number of *fields.* Fields can be elementary data types or nodes themselves. Usually not all fields in a node are used in a given situation. In those cases they can be omitted, with the browser supplying default values for the omitted fields. The hierarchy formed by the nodes in a VRML file is called a *scene graph*.

The example shown in Figure 17 describes a red sphere with a radius of 2 meters. This sphere is positioned 1 meter to the right and 1 meter down, relative to the point of origin. The line numbers with the colons have been included for reference only and are not part of the original VRML file.

```
01:#VRML V2.0 utf8
02:
03:Transform {
04:
05:  translation 1 -1 0
06:
07:  children [
08:    Shape {
09:      appearance Appearance {
10:        material Material {
11:          diffuseColor 1 0 0
12:        }
13:      }
14:      geometry Sphere {
15:        radius 2
16:      }
17:    }
18:  ]
19:}
```

Figure 17: VRML description of a red sphere

Line 1 of the sample code declares that this file contains VRML version 2.0 code. The file is encoded with utf8, an international superset of ASCII. The description of the scene

graph starts in line 3 with a top level `Transform` node. This node has two fields that are of interest in this example: the `translation` field, specifying a 3D translation, and the `children` field, containing a list of the nodes affected by the translation. The only child of the `Transform` node is a `Shape` node with two fields: `appearance` and `geometry`. While the `appearance` field describes the structure of the surface of the object (e.g., the color red) the `geometry` field identifies the object as a sphere with radius 2.

Once this file is loaded into a VRML browser, the user can navigate around the sphere, and examine the object from any point of view. However, no ‘real’ interaction with the sphere is possible, and the state of the 3D object is static, i.e., the model is non-interactive and discrete. This will change in the second example.

4.1.2 Simple Animation

Figure 18 shows the VRML code for a sphere that moves when it is clicked on. The sphere starts at the 3D position (0 10 0), then moves to (10 0 0), and concludes the animation with a movement to (0 -10 0). The total duration of the movement is five seconds.

Looking at Figure 18, one recognizes in lines 12 to 26 the description of a red sphere similar to the one in the first example. The `Transform` node has been given the name “TheTransform” by using the DEF (from DEFine) statement in line 12. The name is required in order to address events that can be produced or consumed by the node.

The `Group` node (lines 2 - 28) is used to form a logical union of nodes. Those nodes are contained in the `children` field of the `Group` node. In this example there are 4 children: A `TouchSensor`, a `TimeSensor`, a `PositionInterpolator` and the already mentioned `Transform` node. All of the `children` have been given names so that all of them can be used as producers and consumers of events.

VRML provides nodes like the `TouchSensor` in order to enable user interactions with the VRML model. The main purpose of the `TouchSensor` is to generate events when some part of the 3D geometry has been touched (i.e., is clicked on). These events are called `touchTime` events and they contain the timestamp of the event. An event that is produced by a node is an *event-out* of that node. In order to be of use to this example, a `touchTime` event-out has to be routed to another node. This is done in line 30 by means of the `ROUTE` statement. In this line the `touchTime` event-out of the `TouchSensor` is routed to the `startTime` of the `TimeSensor`. An event that is consumed by a node is called an *event-in* of that node.

Upon receiving the `startTime` event-in through the route, the `TimeSensor` starts ticking, generating `fraction_changed` events-out for the `cycleInterval` of five seconds (line 6). Those events contain the fraction of time (a float value between 0 and 1)

of the five seconds that has passed when the event is generated. The `fraction_changed` events-out of the timer are routed to the `set_fraction` event-in of the `PositionInterpolator`.

```
01:#VRML V2.0 utf8
02:Group {
03:  children [
04:    DEF TouchMe TouchSensor {}
05:    DEF Timer TimeSensor {
06:     cycleInterval 5
07:    }
08:    DEF TheInterpolator PositionInterpolator {
09:      key      [0, 0.5, 1]
10:      keyValue [0 10 0, 10 0 0, 0 -10 0]
11:    }
12:    DEF TheTransform Transform {
13:      translation 0 10 0
14:      children [
15:        Shape {
16:          appearance Appearance {
17:            material Material {
18:              diffuseColor 1 0 0
19:            }
20:          }
21:          geometry Sphere {
22:            radius 2
23:          }
24:        }
25:      ]
26:    }
27:  ]
28:}
29:
30:ROUTE TouchMe.touchTime TO Timer.startTime
31:ROUTE Timer.fraction_changed TO TheInterpolator.set_fraction
32:ROUTE TheInterpolator.value_changed TO TheTransform.translation
```

Figure 18: VRML description of a moving sphere

The `PositionInterpolator` contains two important fields: the `key` and the `keyValue`. Each `keyValue` belongs to one `key`; in this example the `key` 0 identifies the 3D position (`keyValue`) (0 10 0). It is the task of the interpolator to take the `set_fraction` event-in, compare it to the available keys and produce a corresponding output value by using linear interpolation between a pair of `keyValues`. If, for example, a `set_fraction` with the value 0.25 is received by the `PositionInterpolator`, it recognizes this value to be halfway between the `keys` 0 and 0.5 and therefore generates the output value 0.5(0 10 0) + 0.5(10 0 0).

This value is sent (line 32) with the `value_changed` event-out of the `PositionInterpolator` to the `translation` field (line 13) of the `Transform` node, which results in the motion of the red sphere.

It is important to notice that in the VRML context the term *event* is used in quite a different way than for distributed interactive media. In VRML each routing of information between two nodes is called an event. In contrast, for distributed interactive media only the initial interaction, as made possible by a `TouchSensor`, is called an event. All VRML events that can be deterministically derived from another event are not considered to be a distributed interactive media event in our terminology. In VRML the term *event cascade* is used to describe all VRML events that are initiated by a single user interaction.

4.1.3 Inclusion of Java

While VRML provides all the mechanisms required for simple animations, it does not (and was never intended to) support full programming language functionality. However, this functionality is required by complex animations that involve, for example, state information or non-linear interpolation. In order to provide full programming language functionality, VRML provides hook-ups for existing programming languages. Those hook-ups are called `Script` nodes.

The example shown in Figure 19 makes use of a `Script` node by replacing the interpolator from the previous example with a `Script` node. This customized interpolator allows the sphere to move at an increasing speed, according to the physical law of gravity.

Most interesting in this example is the `Script` node in lines 13 to 17; the remaining parts are similar to those in the previous example. The `Script` node contains a reference to the code that will be executed when events are received. In this example it is the Java class `GravInterp.class` in bytecode format. The reference is given as a URL to provide maximum flexibility. In addition to the URL reference, the `Script` node contains the definition of the events-out and events-in that can be received or sent by the `Script` node. In this case the `Script` node receives the time as an event-in and emits the resulting position as an event-out.

The actual computation is done by the referenced Java class, employing the VRML-Java API for `Script` nodes to communicate with the VRML scene. The usage of `Script` nodes is not restricted to any specific programming language; the support of Java and ECMAScript is described in the annexes of the VRML international standard.

In addition to `Script` nodes, VRML browsers can provide another API for programming language interaction: the External Authoring Interface (EAI) [97]. An early version of the EAI is supported by most state-of-the-art VRML browsers (such as WorldView [38]) and provides methods that allow external applications to control a VRML browser. Through the EAI API it is possible to load and modify VRML content, as well as to

observe and send events. The EAI is typically used when an external application (e.g., an applet on a web page) needs to access the functionality of a VRML browser. In contrast, the script node API is used to include programming language functionality into a VRML model.

```
01:#VRML V2.0 utf8
02:
03:Group {
04:
05:  children [
06:
07:    DEF TouchMe TouchSensor {}
08:
09:    DEF TheTimer TimeSensor {
10:      cycleInterval 5
11:    }
12:
13:    DEF TheGravInterpolator Script {
14:      url “GravInterp.class”
15:      eventOut SFVec3f value_changed
16:      eventIn SFTime set_time
17:    }
18:
19:    DEF TheTransform Transform {
20:      translation 0 10 0
21:      children [
22:        Shape {
23:          appearance Appearance {
24:            material Material {
25:              diffuseColor 1 0 0
26:            }
27:          }
28:          geometry Sphere {
29:            radius 2
30:          }
31:        }
32:      ]
33:    }
34:  ]
35:}
36:
37:ROUTE TouchMe.touchTime TO TheTimer.startTime
38:ROUTE TheTimer.time TO TheGravInterpolator.set_time
39:ROUTE TheGravInterpolator.value_changed TO TheTransform.translation
```

Figure 19: Combining Java and VRML

4.2 Sharing VRML Content

The typical approach to share VRML content among a spatially distributed group of users is to specifically develop the content for this purpose. The driving forces behind this approach are multi-user VRML worlds [98,89]. These multi-user worlds allow participants to enter a virtual world, enabling them to interact with the representations of

other participants as well as with virtual objects. Users are represented by 3D shapes known as *avatars*. Typically, the author of 3D objects for multi-user VRML worlds uses specialized script nodes to provide multi-user capabilities. Using these script nodes, all parts of a multi-user world (avatars, objects and the virtual environment) are specifically designed to support distribution. Therefore, we call these 3D models *collaboration-aware*.

However, the vast majority of current 3D models is not designed to be used in a multi-user VRML world. This is quite natural since multi-user worlds are only one small area where VRML can be employed. Most VRML models are generated by authoring tools, and these are unaware of potential collaboration. Further areas of application - amongst many others - include the usage of VRML as the final form for the presentation of CAD output, as a form of product presentation, and in teachware models. We call these VRML models *collaboration-unaware*. A 3D telecooperation software needs to take special actions to make them available to a distributed group of users.

Generally there are two fundamental problems that need to be solved in order to enable the sharing of collaboration-unaware VRML models:

- **Accessing the state of VRML models.** As we have seen in the previous two chapters, the ability to get and set the state is of vital importance for a distributed interactive medium. Without such a functionality the full synchronization of the medium and the support of latecomers is basically impossible. To access the state of collaboration-unaware VRML models poses a problem since there exists neither a common interface that allows the retrieval of that information, nor a standardized format for the representation of VRML state.
- **Sharing user interaction**. In Chapter Two we described two different information policies for sharing user interactions with a distributed interactive medium. In the state sharing approach the state of the medium is frequently transmitted from a single participant (e.g., the floorholder) to all other participants of a session. If the problem of accessing the state of arbitrary VRML models were solved, then it would theoretically be possible to use the state sharing approach to solve the problem of sharing user interaction. However, state sharing for collaboration-unaware models has one prohibitive disadvantage: the amount of data involved in defining the state of a collaboration-unaware 3D model is very large. For complex models the size of the state data can easily exceed 50 kByte. This makes it practically impossible to use state sharing for collaboration-unaware 3D models.

  The second approach is event sharing. The sources of initial VRML events are typically user actions (such as a user clicking on an object). These initial events trigger an event cascade of derived events that can change the state of the VRML model. As mentioned above, the term initial (VRML) event is equivalent to the term event as we

have introduced it for distributed interactive media. In order to share interactions with 3D models by using event sharing, the application needs to transmit just the initial VRML events, relying on the VRML execution model to reproduce the same event cascade and the same state changes for every participant. Due to the small amount of data involved, event sharing requires only very little bandwidth.

However, one prerequisite of TeCo3D is that the 3D model is collaboration-unaware. By definition such a model does not contain any mechanisms that allows TeCo3D to capture local events or inject remote events into the local model. How to circumvent this limitation is the second key problem that has to be solved.

If we assume that these two problems can be solved, then it is fairly straightforward to see how the abstract media model described in Section Two can be mapped to TeCo3D. A TeCo3D *sub-component* is an interactive 3D model that can be loaded into the shared workspace. Generally there could be more than one 3D model present on the shared workspace at the same time. The *state* of a sub-component is the state of a single 3D model. *Events* represent the user interactions with the VRML content, i.e., the initial VRML events. Since the state of dynamic and interactive 3D objects can change because of the passage of time as well as because of user actions, TeCo3D can be regarded as a *continuous distributed interactive medium*.

Now it is also possible to see that the characteristics of collaboration-unaware VRML content have an important impact on the general design decisions presented in Chapter Two:

- as discussed above, TeCo3D should use event sharing rather than state sharing.
- since TeCo3D sessions will usually involve a group of more than two users exchanging large amounts of data, multicast should be used as a network protocol technique.
- the management of shared state should be done in a distributed fashion, primarily because TeCo3D realizes a continuous medium with strict real-time constraints. The additional latency induced by a centralized server that is responsible for managing the shared state could make an acceptable trade-off between local lag and short-term inconsistencies impossible.

The only general design decision not determined by the characteristics of the medium, is how to ensure reliability. Both application level and transport level reliability are possible. The reason for this is that users will generally not interact with two sub-components at the same time (this could cause the buffering inefficiency of transport level reliability, as discussed in Chapter Two). At the same time, application level reliability could treat state and event transmission in different ways, e.g., by providing forward error correction for events and reliable delivery for states. While this is desirable, it would also add to the complexity of the application. Therefore there are valid reasons to use either one of the two approaches. The only definite restriction is that there should be some sort of reliable

transmission for the state of sub-components since the state information may be rather large. The loss of a fraction of the state information cannot be tolerated.

In order to raise the consistency related fidelity of TeCo3D, it is appropriate to use the concept of local lag, as introduced in Chapter Three. Since the state of the TeCo3D sub-components may be large, and because frequent short-term inconsistencies can severely reduce the consistency related fidelity of the medium, it is a good idea to choose the highest possible value for local lag that can be tolerated by the users. Preliminary experiments indicate that this value is around 250 ms for simple click operations.

We now know the two key problems that have to be solved in order to make TeCo3D possible, and we have seen how TeCo3D fits into the context of (continuous) distributed interactive media as explained in the previous Chapters. What remains to investigate are the concepts and the architecture used by TeCo3D to provide the required functionality, as well as a solution to the two key problems, namely state access and sharing user interactions.

4.3 TeCo3D Concepts and Architecture

The basic design idea behind the TeCo3D architecture is the perception of a 3D telecooperation application as an enhanced VRML browser. This idea is valid since a 3D telecooperation application needs the full functionality of a standard VRML browser in addition to special services for telecollaboration. In order to reuse the effort that has been put into current VRML browsers, TeCo3D relies on existing VRML browsers to supply basic VRML support. These browsers are turned into a shared workspace for dynamic and interactive 3D objects by augmenting them with special TeCo3D functionality. The standard way to reuse an existing VRML browser is to employ the External Authoring Interface (EAI) API which is provided by many VRML browsers. Using this API, a 3D telecooperation application can use the VRML browser as a 3D presentation and navigation engine.

In order to realize event sharing, TeCo3D needs a mechanism to receive events from and inject events into a shared VRML model. However, the original VRML model is collaboration-unaware. This means that it does not contain any methods that enable TeCo3D to receive events from or inject events into a shared VRML model. To solve this problem the original VRML model is processed before TeCo3D loads it into a VRML browser. For true support of sharing collaboration-unaware VRML models it is important that this processing can be done automatically at the time a VRML model is selected for presentation within TeCo3D. This guarantees that the processing is transparent to the user. As will be shown in more detail below, the processing that needs to be done for TeCo3D is simple and can easily be executed automatically: Potential sources of initial events (e.g.,

a touch sensor) are replaced with pre-built, specialized script nodes (e.g., a cooperative version of a touch sensor) that have the same VRML interfaces as the original source. These nodes can be used by the TeCo3D application to capture local events or inject remote events into the local model. The remaining parts of the VRML model do not need to be changed in any way. Processing the collaboration-unaware model results in *semi-collaboration-aware* VRML content that can be shared by TeCo3D. We call this content semi-collaboration-aware because it was not explicitly designed to work in a cooperative environment, even though it was automatically adapted for this purpose. How exactly the processing is performed and what the cooperative versions of the sensors look like will be explained in detail in Section 4.4.

While accessing the state of arbitrary VRML objects that are presented in a VRML browser seems to be a natural piece of functionality, it was not offered by VRML browsers at the time we started our work on TeCo3D. The lack of this type of functionality is surprising, since it is required to realize basic functionality, such as saving and loading the state of a 3D presentation. Because this functionality is universally needed we developed a standardized method of transparent access to and encoding of VRML state information. The results of these efforts are an addition to the EAI, as well as a specification of VRML state encoding. We will have a close look at the EAI enhancement in Section 4.5 while the encoding rules for VRML state information are describe in Appendix A. For now it is sufficient to know that we have extended the EAI interface by the functionality to get and set the state of arbitrary VRML objects, and that this extension is not specialized to the needs of TeCo3D but generally useful for VRML browsers.

Based on the three fundamental concepts - using a VRML browser as a 3D presentation engine, preprocessing collaboration-unaware VRML content to get access to events, and transparent access to the state of arbitrary VRML objects - is the TeCo3D architecture as it is shown in Figure 20. At the start-up of the application, the VRML browser is loaded with an empty VRML world. When the local user imports a collaboration-unaware 3D object into the shared workspace, this object is processed so that it becomes semi-collaboration-aware. The processed object is then added to the VRML world by using a regular EAI call. As the object is loaded into the VRML browser, the inserted nodes that are responsible for capturing the local events and inserting remote events into the local model register with the application. In this process they are assigned a unique ID so that remote events can be routed to the correct cooperative sensor.

The browser and the cooperative sensors are controlled by TeCo3D specific functionality. This is done by using EAI calls and direct access to the script nodes that implement the cooperative sensor functionality. The TeCo3D specific functionality is supported by the generic functionality for distributed interactive media developed in this dissertation, such as a generic consistency mechanism and generic support for latecomers.

Finally there may be specific reliability support that cooperates with the application level protocol for an optimal transport of the data between multiple TeCo3D instances. Besides other services the reliability support could provide forward error correction for events and ensure the reliable transmission of states.

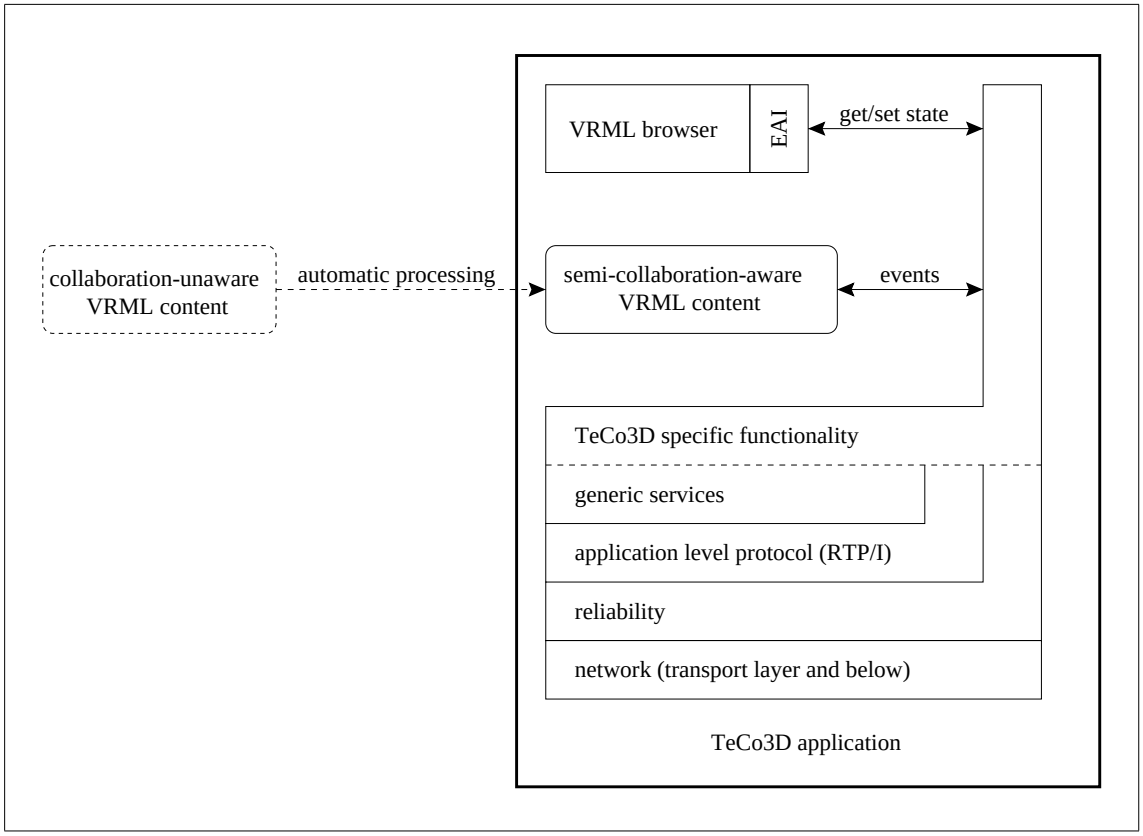

Figure 20: TeCo3D architecture

Once a 3D model has been loaded into the shared workspace of a single participant there are two important things that might happen. First, if there is more than one participant present in the session, the initial state of the 3D model needs to be extracted from the VRML browser and transmitted to the other participants. When a remote application receives this state information, it can load the 3D model into the shared workspace. The delay introduced by extracting, transmitting, and loading the state is automatically compensated by temporal extrapolation when a transmitted state is loaded into the shared workspace. Besides the initial transmission of a new sub-component's state, the ability to extract, transmit and set the state of sub-components is also used to repair short-term inconsistencies and to allow latecomers to join an ongoing session.

The second thing that could happen is that the user starts interacting with the 3D model. The actions of the local user are captured by the cooperative sensors that have been inserted into the model during the pre-processing. When informed about a local event, the TeCo3D specific functionality takes two actions. First, it hands the event to the generic synchronization service, which is responsible to ensure that the event experi-

ences the proper local lag before it is applied to the local 3D model. Second, it is handed to the application level protocol for framing and transmission. After the time specified for the local lag has elapsed, the synchronization service hands the event back to the TeCo3D specific functionality, which in turn injects the local event into the 3D model.

When a remote event is received, it is handed directly to the synchronization service. There it is checked whether the event was received in time. If this is the case, then the event will be buffered for the time that this event arrived early. After this buffering it is treated exactly like a local event (i.e., it is forwarded to the TeCo3D specific functionality and then injected into the 3D model). If the event has not been received in time (e.g., because of packet loss and retransmission), the synchronization service triggers the repair of a short-term inconsistency. The details about the generic synchronization service are described along with other generic services in Chapter Six.

The TeCo3D architecture shows that only a minimal part of the application needs to be developed in a media-specific way. The generic services, the handling of the application level protocol, and the reliability mechanisms can all be realized in a reusable form. Each of these reusable components will be explained in detail in the remaining chapters of this thesis. In addition, Chapter Seven contains a detailed description of the three prototypes that were developed for TeCo3D.

In the remainder of this chapter we focus on the explanation of how exactly TeCo3D turns collaboration-unaware VRML content into semi-collaboration-aware content, how the state of arbitrary VRML content can be extracted from a VRML browser, and how TeCo3D relates to other approaches that can be used to share collaboration-unaware 3D models.

4.4 Event Sharing

Event sharing for collaboration-unaware VRML content is simplified by the fact that the VRML specification defines a set of standard nodes that can be utilized to realize user interaction. These nodes are called *sensors*. Sensors can observe user actions ranging from clicking on (touching) some part of the geometry to dragging objects and user navigation. A sensor emits an initial event when it observes a user action of the type it is listening to. The initial event can possibly trigger an event cascade, which usually leads to a state change in the VRML model. A very simple example of an event cascade was shown in the introduction to VRML, where clicking on a sphere started a timer.

In order to demonstrate how input sharing and the processing of VRML data works, Figure 21 (a) shows an excerpt from a standard VRML file. This fragment of VRML code contains a `TouchSensor` (line 13 - 14), waiting for the user to touch (click on)

some geometry. As the geometry is of no further interest in this example, it is just indicated by three dots. The second element in this example is a `TimeSensor` (lines 15 - 17). A `TimeSensor` does not react to user input but to the passage of time. Initially the `TimeSensor` is inactive. As explained in Section 2, the user can activate the `TouchSensor` by clicking on it. In Figure 21 (a) this causes the `TimeSensor` to start ticking, thereby emitting (non-initial) events that indicate the passage of time. These events can then be used to control an animation, like the moving sphere in the previous example.

```
 1 #VRML V2.0 utf8
 2
 3
 4
 5
 6
 7
 8
 9
10
11 Group {
12   children [
13     DEF TouchIt TouchSensor {
14     }
15     DEF Timer TimeSensor {
16       cycleInterval 5
17     }
18     ...
19   ]
20 }
21 ROUTE TouchIt.touchTime TO
22   Timer.startTime
```

(a)

```
#VRML V2.0 utf8

EXTERNPROTO CoopTouchSensor [
  field        SFString name
  eventOut     SFTime   touchTime
  ...
]
[
 "../CoopSensors.wrl#CoopTouchSensor"
]
Group {
  children [
    DEF TouchIt CoopTouchSensor {
    }
    DEF Timer TimeSensor {
      cycleInterval 5
    }
    ...
  ]
}
ROUTE TouchIt.touchTime TO
  Timer.startTime
```

(b)

Figure 21: Example of VRML content processing for TeCo3D

The processing needed to transform the original VRML content into a semi-collaboration-aware VRML model is shown in Figure 21 (b). For each sensor listening to user input, TeCo3D supplies an alternative implementation, called CoopX, where X is the name of the original sensor. VRML provides a standardized means to integrate these customized nodes into a VRML file by using the EXTERNPROTO statement. In this example the EXTERNPROTO statement (lines 3 - 10) is used to declare a `CoopTouchSensor,` which is implemented in a file called `CoopSensors.wrl`. One of the key ideas behind the TeCo3D event sharing is that a customized cooperative sensor accepts the same events-in and provides the same events-out as the original sensor. Figure 21 (b) shows that the event-out `touchTime` of the `TouchSensor` is also included in the declaration of the `CoopTouchSensor`. All other events-in and events-out are indicated by three dots. The remaining VRML code looks very similar to

that in Figure 21 (a) except that the `TouchSensor` has been changed into a `CoopTouchSensor` (line 13).

It is important to notice that the transformation of the VRML model can be done automatically without human intervention. This ensures that the user perceives TeCo3D to provide true support for sharing collaboration-unaware VRML models.

Figure 22 is a code excerpt from the `CoopSensors.wrl` file. It shows how the `CoopTouchSensor` is implemented. The sensor consists of two parts: a regular `TouchSensor` and a `Script` node referencing a Java class. An object of the referenced class is generated whenever a `CoopTouchSensor` is used in an imported VRML file. This object registers with the TeCo3D application as soon as it is created, establishing the link between VRML content and TeCo3D application.

```
 1 #VRML V2.0 utf8
 2
 3 PROTO CoopTouchSensor [
 4   eventOut     SFTime   touchTime
 5   ...
 6 ]
 7 {
 8   DEF TheTouchSensor TouchSensor {
 9     ...
10   }
11   DEF TheCoopTS Script {
12     url      “TeCo3D.netsensors.CooperativeTouchSensor.class”
13     field    SFString theName IS name
14     ...
15     eventIn  SFTime    set_touchTime
16     eventOut SFTime    touchTime_changed IS touchTime
17   }
18   ...
19   ROUTE TheTouchSensor.touchTime TO TheCoopTS.set_touchTime
20 }
```

Figure 22: Implementation of cooperative sensors

The `TouchSensor` (lines 8 - 10) is responsible to provide the functionality of a standard `TouchSensor` for the `CoopTouchSensor`. The output of the `TouchSensor` is routed (line 19) to the Java object of the `Script` node. Whenever an event is generated by the `TouchSensor`, this Java object notifies the application. The application transmits the event to the peer instances of the application, introduces the appropriate amount of local lag, and feeds the event back into the script node. After a (local or remote) event has been buffered to account for local lag, it is sent to the corresponding Java object of the `CoopTouchSensor`. The `CoopTouchSensor` then produces an appropriate VRML event-out.

4.5 Transparent Access to and Encoding of VRML State Information

Currently VRML browsers do not offer a standardized way to save and restore the state of arbitrary VRML objects and worlds. This is a severe constraint for applications which use VRML browsers as 3D presentation and execution engines. These applications are not restricted to 3D telecooperation applications such as TeCo3D. Rather they are a large class of applications ranging from 3D authoring tools to 3D presentation applications and multi-user virtual reality simulations. None of these applications is currently able to access the state of arbitrary VRML content, even though this is necessary to support fundamental functionality such as saving and restoring a certain state or transmitting it to a communication peer.

Because this functionality is universally needed, we decided to develop an extension of the External Authoring Interface. This extension allows external applications to get and set the state of arbitrary VRML content in a standardized way. In order to support diverse applications, the proposed methods allow not only the retrieval of a virtual world's full state, but also the states and state changes of individual 3D objects. Since the results of state access should be independent of browser implementations, we also specify an encoding for state information. Data in this form is either produced or consumed during state access. To encode state information we use an efficient, easy-to-parse binary encoding. The result of these efforts were presented as a proposal for an extension of the EAI at the fourth symposium of the Virtual Reality Modeling Language (VRML'99) [59].

In the following we summarize the requirements for transparent access to and encoding of VRML state information before introducing the additional EAI functions to access the VRML state. We give a simplified overview, a more detailed specification can be found in [59,62]. The grammar for encoded VRML states is specified in Appendix A.

4.5.1 Requirements

Two aspects have to be addressed in order to realize a standard way to save and restore the VRML state. The first is the definition of access methods. A minimal set of functionality includes methods for saving and methods for restoring state information. The second aspect is the specification of a binary encoding that enables browser-independent storage and exchange of the VRML state. A number of requirements pertaining to both aspects can be derived from the areas of application for VRML state access as well as from performance and browser integration issues.

Requirements pertinent to the access methods are:

- **Transparency**. The methods should be independent of the VRML content. The main reason for demanding transparency is that the state of all worlds and objects should be accessible, independent of how they were created.
- **Flexible handling of time**. Time is an integral component of the VRML state. Several fields in VRML nodes may reference points in real time. There are two semantically different methods to handle these references. Both are shown in Figure 23. The first method compensates for the time that has passed between saving and restoring the state. The content will be presented as if the time between saving and restoring the state had never elapsed. This method adjusts all references to real time in the encoded state by adding the difference between the point in time the state is restored and the point in time the state was saved. The time values are adjusted when the state is restored. This method makes sense if users want to save a state and restore it at a later point in time to continue interaction with the same state of the VRML world.
  However, there are scenarios where this behavior is not appropriate. If, for example, two users are interacting with a shared VRML object (e.g., in TeCo3D), it might be desirable for one user to update the other about a state change. If the time needed to save, transmit, and restore the state were compensated when the receiver decoded the state, then the two users would end up with an inconsistent model: The sender would precede the receiver in time. It is therefore necessary to provide a second method for restoring the state. In this method all references to time are kept as they were when the state was saved, no matter at what time the state is restored. This is particularly useful in collaborative or streaming applications, where the passage of time continues synchronously for all communication partners (e.g., the sender and the receivers of state information).
  Both methods can be enhanced by specifying a time offset, to be added to the time values when the state is restored. This provides a limited capability for jumping forward and backward in time, e.g., to compensate for the deviation of physical clocks.
- **Access of sub-components**. As we have seen in Chapter Two, the state of an interactive medium can usually be decomposed into several sub-components. This is also true for VRML worlds, where the sub-components could be avatars or other 3D models and objects. It is therefore desirable to be able to provide state access on the granularity of individual sub-components.
- **Delta States**. Similarly, in cases where the states of worlds and objects are saved and restored frequently, it is important to be able to retrieve only those parts that have changed since the last time the state was saved. Therefore there should be a way to access the delta state of sub-components or of entire VRML worlds.

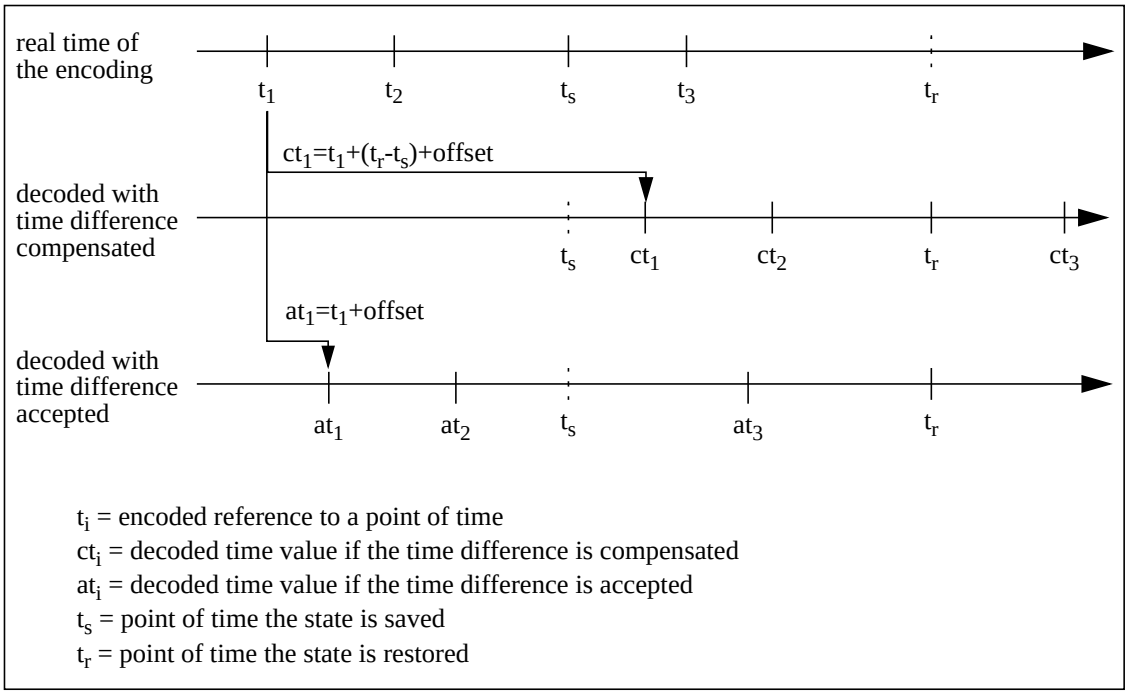

Figure 23: Handling of time references

The encoding has to meet the following requirements:

- **Support of access methods**. The encoding must support the access methods in order for them to fulfil the requirements mentioned above. Specifically it must allow transparency as well as the definition of sub-components and delta states.
- **Efficient encoding**. It is important that the encoding be efficient with regard to the amount of data needed to define VRML states. Since state descriptions are generated automatically and are not supposed to be edited by humans, the usage of a human-readable format is not necessary. Instead a binary format should be used.
- **Simple encoding**. A relatively simple persistence format is needed to enable browsers to save and restore VRML state fast and without too much processing. This is particularly important since the state access might happen at a time when the user wants to interact with the content. Sophisticated compression is therefore not acceptable. For this reason we recommend to use an uncompressed base format that might be compressed off-line using external, lossless compression such as runlength and/or Huffman encoding.

In the following we will present only the access methods we have developed for VRML state information. The rules for the encoding are explained in Appendix A in form of an VRML-state encoding grammar.

4.5.1.1 Accessing State Information of VRML Content

We propose to realize access to the state of VRML worlds by extending the External Authoring Interface (EAI). In this section we introduce an extension to the Java language binding of the EAI that consists of four additional methods for the Browser class. Similar methods can be added to the language-independent EAI or to the Script node API.

The first two methods provide the capability for getting the state of VRML content (see Figure 24). With `getWorldState` it is possible to get the state of a complete world. The caller of the method provides the `OutputStream` to which the encoded state should be saved. If `allowDeltas` is true, the browser will automatically encode a delta state when this is appropriate. This decision is implementation dependent. For example, this decision may be influenced by the amount of changes in the VRML model compared to the last encoded full state. If `allowDeltas` is false, the method will produce only full states. The return value of the method indicates whether a full or a delta state was encoded. The method for getting sub-components (nodes) is called `getNodeState` and has one additional parameter: the node that is encoded. In all other respects it acts exactly like `getWorldState`.

```
int getWorldState(OutputStream os, boolean allowDeltas)
int getNodeState (OutputStream os, boolean allowDeltas,
                  Node subComponent)
return values: FULL_STATE
               DELTA_STATE

void setWorldState(InputStream is, boolean keepTimeDifference,
                   double timeOffset)
void setNodeState (InputStream is, boolean keepTimeDifference,
                   double timeOffset, boolean replace, Node target)
```

Figure 24: Additional EAI methods

The state of VRML content can be restored by using either the `setWorldState` or the `setNodeState` method. The first method accepts an `InputStream` that should contain the state of a VRML world (full state or delta state). Two parameters are used to control how fields that reference time values are restored: If `keepTimeDifference` is false, the difference between the point in time the state is restored and the point in time the state was saved is added to those fields that reference absolute time values. If `keepTimeDifference` is true, the difference is not added. In both cases the appropriate fields that reference time values are modified by adding the `timeOffset` value.

Note that it is only legal to decode a delta state if the preceding full state has already been decoded. If this is not the case, the results of this operation are undefined.

Decoding the state of a single node requires additional information about the placement of this node. In the case of a full encoding, the node can overwrite an existing node

(`replace` set to true, `target` is the node to replace) or be inserted as a child of an existing node (`replace` set to false, `target` is the parent node).

4.6 Related Work

An obvious approach to sharing cooperation-unaware VRML content would be the usage of general application-sharing tools as described in Section 2.1.2.1. While sharing the output of a single VRML browser instance might seem to be similar to the functionality offered by TeCo3D, the application-sharing approach has severe disadvantages. First, and most important, the volume of data produced by sharing the output of a VRML browser is extremely high compared to the sharing of regular applications (e.g., word processing software) whose window content does not change rapidly. At 15 to 25 rendered 3D frames per second, the transmission of window content becomes infeasible. A second disadvantage of the application-sharing approach is the limited functionality for the support of collaborative work, e.g., application sharing would not allow users to view the same VRML content from different points of view.

The first truly distributed VRML browser was developed in the context of the Java-Enabled TeleCollaboration System (JETS) at the University of Ottawa [85]. JETS is a programming interface and toolbox for the development of collaborative Java applets. With JETS, a developer can use the mechanisms provided for consistency, access control and data exchange to develop so-called shared applets. The JETS distributed VRML browser - a sample applet constructed with JETS - allows for the shared viewing of static VRML 1.0 content. TeCo3D, on the other hand, allows the synchronized interaction with dynamic and interactive VRML content, which is not possible in the JETS VRML applet.

The Shared 3D Viewer [44] is a product sold by HP. It allows the 3D visualization of product data for a distributed group of users. The Shared 3D Viewer offers rich collaborative functionality such as annotations and telepointers. However, like JETS, it does not support dynamic or interactive 3D models.

4.7 Chapter Summary

In this chapter we introduced TeCo3D as a shared workspace for collaboration-unaware dynamic and interactive 3D content. The two key problems that had to be solved in order to provide this functionality were how to share user interactions with collaboration-unaware 3D models and how to transparently access arbitrary VRML state information. We developed solutions to both problems. The problem of sharing user interactions was solved by automatically replacing the standard VRML sensors with customized collabo-

ration-aware sensors, while the transparent access to state information required an extension to the application programming interface provided by VRML browsers.

Furthermore, we identified TeCo3D as an application for continuous distributed interactive media. The architecture presented in this chapter reflects the main proposition of this theses, namely that distinct distributed interactive media share a large amount of functionality. Therefore TeCo3D provides only a small amount of specialized functionality, while many parts (e.g., the application level protocol, the synchronization service, support for latecomers, etc.) are realized generically.

In the following two chapters we introduce a generic communication protocol for distributed interactive media and show how generic and reusable services can be developed based on this protocol. In Chapter Seven we return to TeCo3D and investigate the experiences gained by the implementation of three TeCo3D prototypes.

5 RTP/I - an Application Level Protocol for Distributed Interactive Media

In this chapter we present the Real Time Protocol for Distributed Interactive Media (RTP/I). RTP/I provides a common foundation for the distributed interactive media class and thereby allows the development of generic functionality and services. Since RTP/I is customizable to the needs of different distributed interactive media, it is also referred to as a protocol framework.

We start the discussion with an investigation of the fundamental design considerations for this type of protocol. We will then have a look at the Real Time Transport Protocol (RTP), which is commonly used to transport continuous distributed non-interactive media, such as audio and video. While RTP exhibits several characteristics that are also desirable for RTP/I, we shall show that it is not appropriate to directly employ RTP for the transport of distributed interactive media.

In the main part of this chapter we introduce RTP/I as a protocol that reuses many aspects of RTP while it is thoroughly adapted to meet the demands of the new media class. The presentation of RTP/I is followed by a discussion of how application level reliability libraries could offer their service in a manner appropriate for applications using RTP/I. Finally we demonstrate how RTP/I is used to transport TeCo3D data.

5.1 Design Considerations

In this section we examine important considerations that should influence the design of an application level protocol for distributed interactive media. These considerations can be grouped into the following categories:

- core functionality for distributing event and state information,
- support for maintaining the consistency of shared state,
- support for fragmentation of oversized states and events,
- getting the current state of a sub-component in a standardized way,
- conveying meta-information, and

- flexibility to customize the protocol to the specific needs of individual media.

In the following sub-sections we will examine each of these categories in detail.

5.1.1 Core Functionality

The core functionality of an application level protocol for distributed interactive media is to enable the dissemination of event and state information that are framed by a common header. The common header needs to exhibit enough information for generic services to take appropriate actions without knowing anything about the medium-specific encoding of states and events. Two obvious pieces of information that should be visible to a generic service are the type of the data (event vs. state) and an identification of the sub-component to which it refers. With these two pieces of information a generic service can interpret the semantics of a distributed interactive media stream to a high degree.

In the following we use the term *protocol* to denote a standardized framing of transmission units that may or may not include rules on when these transmission units should be sent. This is consistent with the use of this term in the networked multimedia community. The more traditional meaning of the term protocol would require that the rules for the sending of transmission units (i.e., the protocol automaton) be part of a protocol definition. We do not use this traditional meaning of the term protocol.

5.1.2 Consistency

Besides the identification and framing of events and states, the chief aim of an application level protocol for distributed interactive media is to provide the information required to maintain a consistent shared state. As we have seen in Chapter Three, the consistency criterion can vary depending on the medium and the application. The strongest level of consistency would require that at all sites all events and/or state updates be applied in the correct order at the correct point in time. According to the trade-off between short-term inconsistencies and response time, this level should be relaxed by allowing events and state updates to get lost, or be processed out of order, or at a “wrong” point in time. Obviously this requires either some sort of repair mechanism or the willingness of the user to accept a certain amount of inconsistency. Generally it can be said that diverse *consistency policies* exist, that realize different levels of consistency. An application level protocol for distributed interactive media should provide the information required to realize these policies without imposing a certain policy on the medium or the application.

Up to three types of mechanisms can be involved in achieving consistency for a distributed interactive medium:

- reliability may be required so that each participant will eventually learn about all state changes caused by remote actions,

- ordering may be required if events are not commutative (i.e., if the order in which events are applied to a state matters),
- timing may be required when an event is only valid at a single point in time.

In the following we consider the information required for each of these mechanisms and discuss whether and how they should be supported by a general application level protocol for distributed interactive media.

5.1.2.1 Reliability

As we have seen in Chapter Two, there exist two main approaches to establish reliability in distributed interactive media: transport level reliability and application level reliability. Using transport level reliability does not place additional requirements on an application level protocol for distributed interactive media. The information concerning reliability is contained in the transport protocol.

In the context of application level reliability an application level protocol for distributed interactive media can be regarded as a specialization of the ADU framing that is used for application level reliability mechanisms. It provides many information required to realize reliability (e.g., sender identification, sequence numbers, etc.) and adapts this framing to the specific needs of distributed interactive media. The application level protocol should not try to specify reliability mechanisms. This would be inappropriate since the reliability requirements of distributed interactive media vary widely. Instead there should be a well defined way that allows arbitrary application level reliability mechanisms to use and extend the information provided by an application level protocol for distributed interactive media. This would lead to the typical benefits of Integrated Layer Processing [9], namely the avoidance of redundant header information, as well as a reduction of copy operations.

In summarizing, it can be noted that both approaches to achieve reliability should be usable with a framing protocol for distributed interactive media. The protocol should not try to realize reliability (e.g., by providing functionality for retransmission, forward error correction, tail loss detection, etc.) - this is done either by a transport level protocol or by the application, possibly with the help of a library such as libsrm.

However, it *is* the task of an application level protocol to capture the common aspects of a media class. This includes information like an identifier for the sub-component that is concerned by the transmitted data, as well as sequence numbers for the transmitted events and states. This information is frequently required by applications and generic services, even when a reliable transport protocol is used. An application level protocol therefore provides much of the information required to deploy application level reliability, independent of how reliability is actually realized. In fact, applications using a com-

mon application level protocol for distributed interactive media should always communicate in terms of ADUs, even when the application chooses to employ a reliable transport protocol. After all, it is the ADU framing that allows generic services to understand the semantics of arbitrary distributed interactive media streams.

5.1.2.2 Ordering

Sequence numbers are an important tool for the ordering of transmitted events and states. One might argue that sequence numbers are not required if a reliable transport protocol is used since such a protocol will ensure the reliable and ordered delivery of data. However, sequence numbers are not only needed for the detection of lost and misordered packets. They are also useful to address and identify individual ADUs, as required by a number of applications and services (e.g., consistency mechanisms, recording, floor control), independent of the transport protocol. Therefore sequence numbers should be part of the framing specified by an application level protocol for distributed interactive media. Generally it is a good idea to have distinct sequence numbers for each sub-component and ADU type (e.g., events and states). This allows application level reliability to reuse the sequence numbers for fine-grained ADU loss detection and recovery.

While sequence numbers can solve the ordering problem for events and states from a single source, many distributed interactive media additionally require a complete ordering relation for the messages sent from all participants of a session. Such a total ordering of all events and state updates can be established using a timestamp-oriented ordering relation as described in Chapter Three. In general it is possible to use either the timestamp values of a physical clock or of a logical clock for this purpose. If the ADUs of two participants carry the same timestamp, a complete ordering relation can be used to find the proper order of these ADUs by means of a tiebreaker (see section 3.2).

5.1.2.3 Timing

Only timestamps which refer to a physical clock are usable for the timing in continuous distributed interactive media since an event or state for this class of media is only valid at a single point in (real) time. This requires the existence of a common physical time base that can be established using protocols like NTP [68] (typical deviation from the real time: less than 50ms) and/or GPS receivers (deviation of less then 1ms). If a message arrives late in a continuous medium it cannot be directly applied to the local copy of the medium's state. Instead actions need to be taken to repair the problem.

A physical timestamp is also required if the distributed interactive medium is to be synchronized with other media such as audio and video streams. Logical time, as described in Lamport's work on virtual clocks [48], can be used in the absence of a common phys-

ical time. However, logical time is only appropriate for discrete interactive media. Generally it is preferable to use a physical clock since distributed interactive media are very likely to be used in combination with continuous non-interactive media such as audio and video.

5.1.3 Fragmentation

If all distributed interactive media could guarantee each ADU to fit into a minimal network layer packet, then there would be no need to support fragmentation in the application level protocol. Unfortunately, it is very likely that many distributed interactive media will transmit states (and maybe even events) that do not fit into a single network layer packet.

As we have discussed in Chapter Two, fragmentation should not be left to the network layer. An application level protocol for this media class should therefore support fragmentation of ADUs by providing an additional fragment count for the chunks of data belonging to a single state or event. This allows the application to fragment ADUs should this be necessary. The fragment count should be incremented for each transmitted chunk of data. This mechanism can be used by the application to prevent inefficient IP-level fragmentation, while it also supports ALF even when ADUs are large.

5.1.4 Getting the Current State of a Sub-Component

In a number of situations an application or a generic service needs to get the current state of a certain sub-component. Examples of those situations are: resynchronization if an event has been lost or received late, random access in a distributed interactive media recorder, or setting the initial state for a latecomer joining an ongoing session. Since this functionality is universally needed by many distributed interactive media, it should be supported directly by a standardized state query ADU.

As the computation and transmission of state information may be costly, the potential senders of a reply must be able to distinguish between different types of state queries. Recovery after an error urgently requires information on the sub-component's state. These requests will be relatively rare. In contrast to this, a recorder does need to receive the state of the medium in order to enable random access to the recorded medium [33]. It does not need the state transmission immediately but will issue requests frequently. In order to enable potential senders of a state reply to discern between the different levels of urgency, the state request mechanism must support different priorities. Receivers of the state query packet should satisfy requests with high priority (e.g., for resynchronization) very quickly. Requests with low priority can be delayed or even ignored, e.g., if the sender currently has insufficient resources.

5.1.5 Meta-Information

Applications and generic services frequently need additional information about the medium and the participants of a session. This information should be provided by an application level protocol for distributed interactive media.

5.1.5.1 Meta-Information about Sub-Components

An example where meta-information about sub-components is needed are applications that join an ongoing session. A late coming application should be able to decide for each sub-component whether the sub-component is relevant for the local presentation of the medium. Equipped with this information, the application can then take the required actions to get the state of those sub-components that are of interest to the local participant.

In another example a participant may introduce a new sub-component into the session. Depending on the medium, it may not be wise to transmit the state of the sub-component immediately. Instead, just the presence of the sub-component could be announced as meta-information. If any other application is interested in that sub-component, it could use the state query mechanism to get the state of that sub-component. Since the meta-information about sub-components is generally required for distributed interactive media, it should be transported in a standardized way by an application level protocol for this media class.

Generally an application or a service may require up to three distinct types of meta-information about a sub-component: it needs to know that it is present in a session, it might want to know whether it is currently used to display the medium to any one of the participating users, and, finally, the sub-component might have an application level name. While the usefulness of the first information is obvious (amongst others this information is required so that applications know the sub-component space that defines the overall medium), the latter two require more investigation.

Those sub-components that are actively used to display the medium to at least one user define the portion of the medium that is exposed to the attention of the users. We call these sub-components *active sub-components*. Active sub-components are likely to be of immediate relevance for the session. Sub-components that are not active might be handled with a lower priority, e.g., for applications that join an ongoing session, or for a recording service. A prime example of an active sub-component in a shared whiteboard presentation is the shared whiteboard page that is currently visible to all participants of a session. The other pages are not active. Any application that joins the ongoing presentation will try to get the active page with higher priority than the other pages.

Besides the presence of sub-components and their status (active or not active), a third type of meta-information may be desirable for sub-components: application level names. The reason for this is as follows: partitioning the state of a distributed interactive medium into several sub-components leads to the question of how an application can decide which sub-component it is interested in. While for some media it may be possible to encode the required information into the sub-component ID, this is not always feasible. Typically the problem can be solved by providing each sub-component with an application level name. This name should carry information about the sub-component so that an application can decide whether or not it is interested in the sub-component. Examples of application level names are the title of a shared whiteboard page, or the description of a 3D object in a distributed virtual environment.

While it is conceivable to include the application level name in each transmitted event and state, this could easily burden the network with a high load. Also, late coming applications might not be able to learn about sub-components for which currently no events and states are transmitted. An application level protocol for distributed interactive media should therefore use sub-component IDs in event and state ADUs and additionally provide a mechanism for the mapping of IDs to application level names. This mapping is important meta-information for many distributed interactive media.

Generally two approaches can be used to provide the mapping between application level names and sub-component IDs. The first is to employ a special protocol for application level naming such as the Scalable Naming and Announcement Protocol (SNAP) of libsrm (see section 2.3.2). While it transmits information only when it is needed, SNAP also relies on the SRM mechanisms to convey information in a reliable fashion. SNAP can therefore be regarded as a heavyweight approach in respect to its complexity.

The second approach requires applications to regularly announce the mapping between sub-component IDs and application level names by means of a special protocol element. This is commonly known as announce/listen or soft-state approach. The bandwidth needed for all announcements of a medium should be restricted by making the frequency of announcements inverse proportional to the number of sub-components. The key benefits of this approach are its simplicity, its independence of reliability mechanisms, and its robustness to failure of individual participants.

While a specific protocol for the mapping of application level names to sub-component IDs might consume less bandwidth - most information is only transmitted upon request - the heavyweight nature is a decisive disadvantage. Its complexity and dependence on the availability of reliable multicast functionality makes the inclusion of such a protocol into a general application level protocol for distributed interactive media inappropriate. We

therefore recommend to use the announce/listen approach for the mapping of IDs to application level names.

The following brief calculation shows that the data rate consumed by the announce/listen approach is acceptable: let us assume a rather large session with 1,000 sub-components, with long application level names (20 bytes), and large sub-component IDs (8 bytes). Further, we assume that at least every 60 seconds all mapping information should be available. This would result in a data rate of 3.7 kBit/s for the regular announcement of mapping information. This seems reasonable for a large session.

5.1.5.2 Meta-Information about Session Participants

Meta-information about the participants of a session is desirable since many applications that use distributed interactive media require a simple session control functionality. An application level protocol should support this by providing a standardized means to communicate information about participants. Typical participant information includes the name, e-mail address, and phone number. This concept is very successfully used in RTP, where many audio and video tools rely on the participant description that is transported by RTCP, the control protocol of RTP, in order to establish a loose session control.

5.1.5.3 Meta-Information about the Network Quality

Another interesting type of meta-information are reception quality reports. Examples are loss rates, latency and jitter. These values can be transmitted periodically by each participant. However, such a quality feedback is generally inappropriate as a standard component in an application level protocol for distributed interactive media. The reason is that ADUs might be retransmitted either by a reliable transport protocol or by a library for application level reliability. This would render reports on loss rates, latency and jitter invalid. Instead such feedback should be part of whatever mechanism is used to achieve reliability. Alternatively it could also be an optional protocol element used only by those distributed interactive media that realize reliability without the help of retransmissions.

5.1.6 Flexibility

A “good” framing protocol for distributed interactive media will provide a large amount of flexibility, especially in consistency related issues and for the actual encoding of states and events. It is therefore necessary to specify an application level protocol for distributed interactive media as a protocol framework rather than as a monolithic protocol. One type of flexibility has already been discussed: the ability to use different reliability mechanisms. Additional flexibility should be provided so that the protocol can be tailored to

the specific needs of diverse distributed interactive media. Ideally this should be a a two-step process.

The first step should be a specification document that defines the commonalities of a sub-class of distributed interactive media. A possible sub-class may exhibit the following characteristics: continuous distributed interactive medium, frequent transmission of sub-component state information to achieve consistency, no reliability at the transport level, no event transmission. This sub-class would support the majority of battlefield simulations and some networked action games. In analogy to RTP we call such a specification document a *profile*. A profile may specify which reliability mechanism(s) are to be used and how consistency is realized. A profile may also define additional information that is included in the framing of the transmitted information. Furthermore, it can define additional meta-information that is important for this sub-class of media, e.g., quality feedback for media sub-classes that do not use packet retranmsissions. However, a profile may not change the meaning of the information that is specified for the core protocol since this would prevent generic services from working properly for the profile.

The second step in customizing an application level protocol for distributed interactive media should be the specification of how a single medium is carried using the framework set up by the application level protocol and possibly by a profile. This definition contains the encoding of event and state information, as well as the specification of reliability, timing, and consistency constraints that are not already covered by a profile. Such a definition is called a *payload type* specification. Theoretically a payload type specification can be used without a profile or with multiple profiles. However, the usual case is a hierarchical relationship of a payload type definition that operates under exactly one profile specification. This allows generic services to be specified at three levels:

- they can be fully generic services and, as such, be usable by any medium transported over the given application level protocol for distributed interactive media (some of these services, such as application level naming, should be part of the protocol itself),
- they can specify certain profiles for which they are usable,
- and they can be limited to a number of specific payload types.

5.2 Real-Time Transport Protocol

The Real-Time Transport Protocol (RTP) is commonly used to transport audio and video sessions over the Internet. Because some of its functionality is also interesting for distributed interactive media, it has been used for shared whiteboards [22] and distributed virtual environments [30]. We will therefore investigate RTP and show which parts of it are appropriate for an application level protocol for distributed interactive media and which

are not. In particular we will conclude that the definition of a profile and payload types for RTP is not the right way to create a protocol for distributed interactive media.

RTP follows the architectural principle of Application Level Framing (ALF) [9]. It therefore proposes a standardized framing for ADUs transmitted by applications that use RTP. In order to be flexible and useful for a large group of media, RTP has been left incomplete and can be tailored to the specific needs of a medium using profiles and payload type definitions. A profile “defines a set of payload type codes and their mapping to payload formats (e.g., media encodings). A profile may also define extensions or modifications to RTP that are specific to a particular class of applications” [40]. Payload formats are “specification documents, which define how a particular payload, such as an audio or video encoding, is to be carried in RTP” [40].

RTP consists of two closely related protocols: the real-time transport protocol (RTP), which is used to carry data with real-time properties, and the RTP control protocol (RTCP), which conveys meta-information such as reception quality and participant identification. In the following sections we will investigate each of these in detail.

5.2.1 RTP Data Transfer Protocol

The data part of RTP consists of a single ADU type that is used to transport the core real-time data. The general structure of an RTP data ADU is depicted in Figure 25. The first two bits (V) identify the protocol version. This information is followed by a padding bit (P), which may be set to one if the data contained in the ADU includes one or more padding bytes at the end of the packet. These padding bytes may be required by specific encryption algorithms. The presence of an experimental header extension is signalled with the extension (X) bit. If present, the header extensions follows the regular RTP data header information.

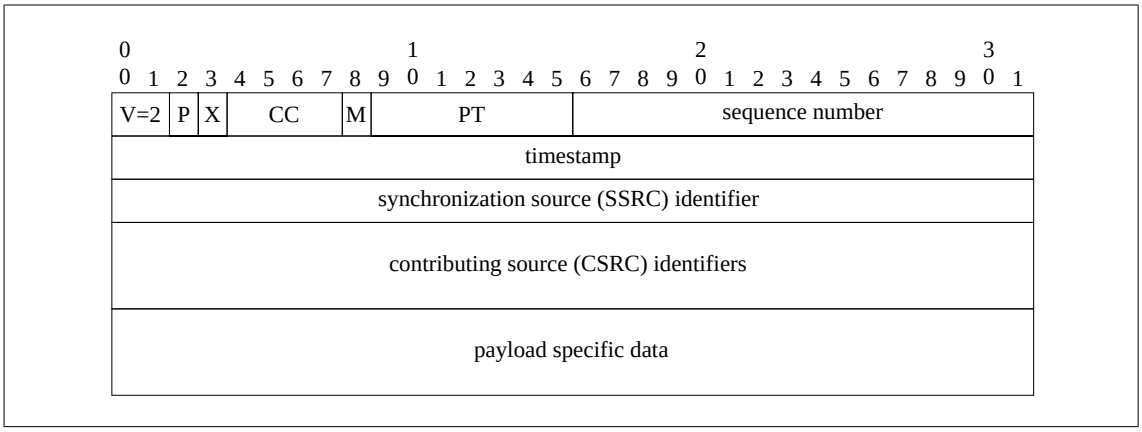

Figure 25: RTP application data unit [40]

In order to be able to identify its sender, an RTP ADU contains a 32-bit synchronization source (SSRC) identifier. The transport address (IP address in combination with a UDP

port) of the packet containing the ADU is not sufficient to identify the original sender since the packet might have been transcoded or forwarded by intermediate systems. The SSRC identifier is chosen randomly by each participant with the intention that it be unique in the session. In the (rare) case that two participants choose the same SSRC identifier, a collision mechanism ensures that at least one of the two conflicting participants chooses a new identifier. Until the collision has been resolved, the ADUs of one or both participants will be discarded by other participants. For audio and video this does not pose a problem since these media have an ephemeral state, which is regularly transmitted by a single sender.

The RTP standard directly supports the presence of mixers. A mixer combines the streams produced by multiple participants into a single stream. For example, an audio mixer may accept the streams of a number of participants and convert them to a single audio stream. This mechanism can be used to save bandwidth. The SSRC identifier of a mixed stream will be the SSRC ID of the mixer. In addition RTP provides an identification mechanism for those senders who originally contributed to a mixed stream. This is done using contribution source (CSRC) identifiers. The CSRC field may hold up to 31 SSRC identifiers of those participants which contributed to the stream produced by the mixer. The CSRC count field (CC) holds the number of the CSRC identifiers present in an RTP data unit.

The interpretation of the marker bit (M) is defined in the payload type specification used for the transmission of the data. Typically it marks important ADUs such as the last ADU in a single video frame. Following the marker bit is the payload type field (PT). This field identifies the payload type of the data transported in this ADU. Examples of payload types are H.261-encoded video or GSM-encoded audio.

The sequence number is incremented for each ADU sent by a given sender. It can be used by receivers to detect packet loss and to restore the original packet order. The timestamp reflects the point in time when the content of the ADU was sampled. Its format is defined in the payload or the profile specification. The initial value of the timestamp is chosen at random. Therefore it is not possible to map the timestamp to a physical time value without further assistance, such as that provided by the RTP control protocol.

5.2.2 RTP Control Protocol

The RTP Control Protocol (RTCP) is based on the periodic transmission of information by all participants of a session. RTCP provides meta-information about the session. This meta-information includes

- reception quality feedback (e.g., packet loss, latency, and jitter),

- the mapping of timestamps to physical time,
- a unique participant identifier that can be used to track participants when their SSRC changes (e.g., due to an SSRC collision), and
- minimal information about the participants of a session (e.g., the names of the users).

Typically there are four packet types used for RTCP. The sender report (SR) is used by active senders and contains information about the data transmitted by the sender. A receiver report (RR) provides feedback on the reception quality that is experienced by a receiver. Each participant reports periodically on the quality of its reception from all senders in the session. The source description (SDES) packet holds information about the participant transmitting the packet. Finally a bye (BYE) packet can be used to signal that a participant is about to quit the session.

The RTCP packets are transmitted as so-called compound packets. Each compound packet consists of at least one report packet (receiver or sender report) and a source description packet. An example of a compound packet is shown in Figure 26. RTCP compound packets are transmitted by each participant at regular intervals. The interval between the transmission of two compound packets is called the report interval. In order to keep the overall data rate consumed by RTCP constant, the duration of the report interval is inverse proportional to both the average size of the compound packets and the number of users in the session.

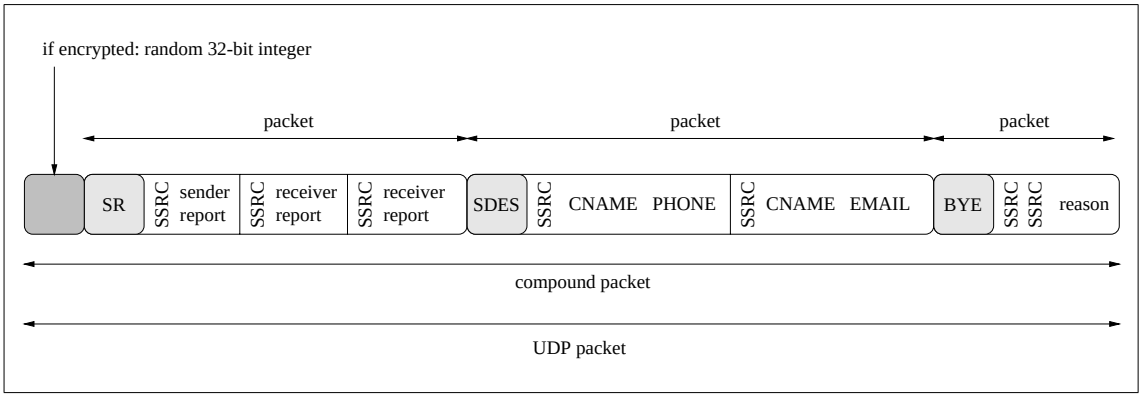

Figure 26: RTCP compound packet [40]

5.2.2.1 RTCP Sender and Receiver Report Packets

The format of an RTCP sender report is depicted in Figure 27. It consists of three main parts: the header, sender information, and report blocks. A receiver report is identical to a sender report without the sender information part.

In the header of a sender report the version (V) and padding (P) bits are followed by a report count (RC) field. The RC field specifies how many report blocks are present in the packet. The payload type field identifies the type of the RTCP packet - in this case a

sender report (SR). RTCP packets require a length field since they are combined into compound packets. The final item in the RTCP header is the SSRC of this report's sender.

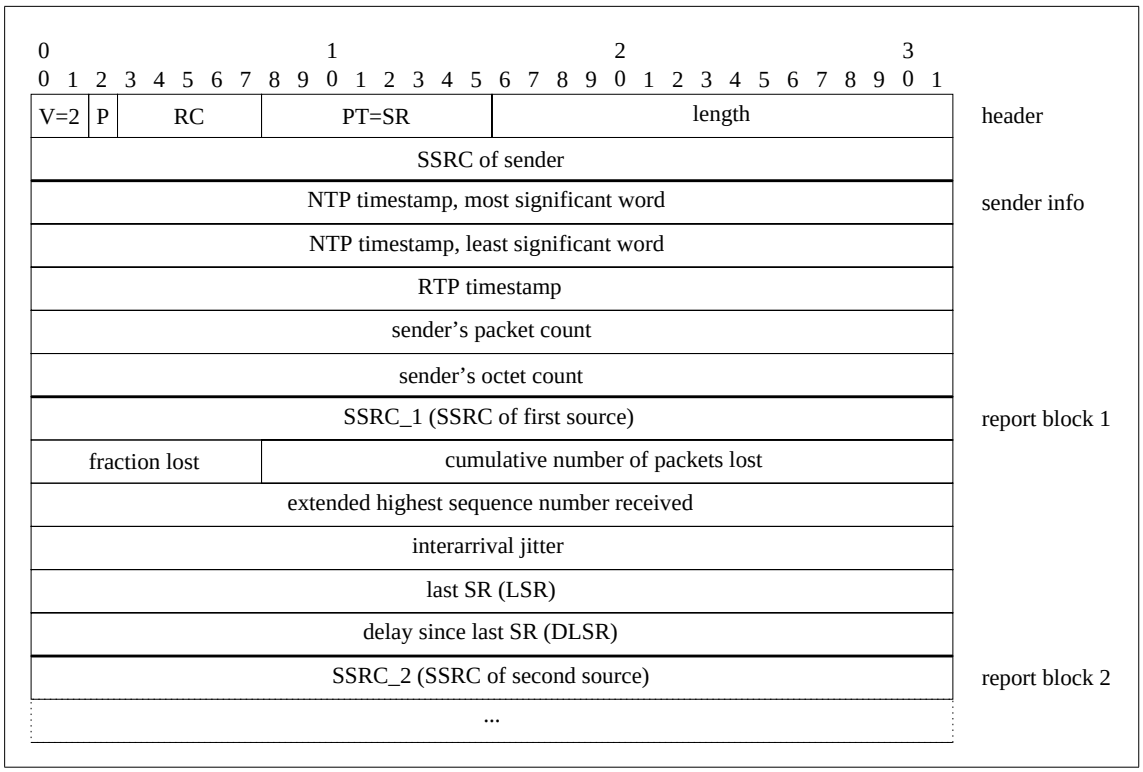

Figure 27: RTCP sender report packet [40]

The main task of the sender information is to enable the mapping of RTP timestamps to physical time values. For this purpose it contains a full 64-bit NTP timestamp and the RTP timestamp that corresponds to the same physical time. In addition to these timestamps there is a packet count field that displays the number of packets transmitted by this sender. The octet count field shows the total number of data octets transmitted by the sender. It can be used to estimate the raw data rate of the media stream, excluding any RTP or transport header bytes.

A report block describes the reception quality of the report's sender in regard to a media stream transmitted by a single source. The source of the media stream is identified by including its SSRC in the report block. The reception quality information is reported as follows: the fraction of lost packets since the last report for that source, the cumulative number of packets lost during the session, the highest sequence number received from the source, and the interarrival jitter of ADUs transmitted by the source. In order to allow for round-trip time estimations, a report block also contains the timestamp of the last sender report received from that source and the time that has passed between receiving this sender report and transmitting the report packet.

5.2.2.2 RTCP Source Description Packet

The Source Description (SDES) packet contains information about a participant. As depicted in Figure 28, the SDES packet has a header similar to that of the report packets. The only difference is the interpretation of the source count (SC) field. In an SDES packet this field is used if a mixer announces source description information for all sources that contribute to the mixed media stream. In this case the SC field contains the number of sources for which source description chunks are included in the packet.

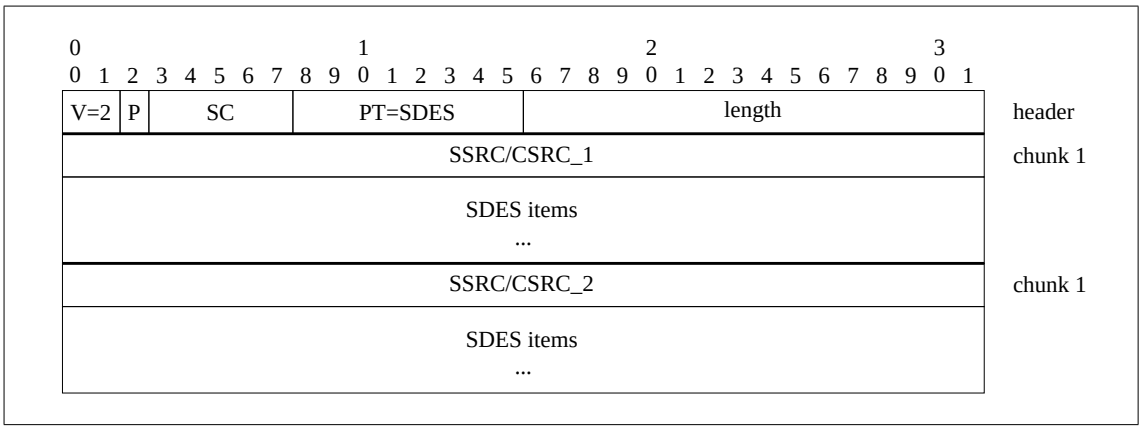

Figure 28: RTCP source description packet [40]

The header of an SDES packet is followed by one or more source description chunks. Each chunk describes one source and holds one or more source description items. The most important item is the canonical name (CNAME). This name should be globally unique, a typical example would be username@host.domain. The CNAME is used to track participants even when they change their SSRC ID. As shown in Figure 29 the CNAME item consists of an identifier, which specifies the type of the item, a length field, and the actual item. Other source description items follow the same format, examples are the email address or the telephone number of the participant.

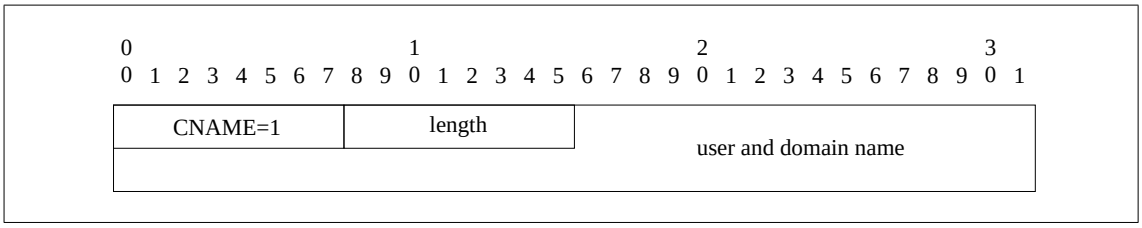

Figure 29: RTCP source description item [40]

5.2.2.3 RTCP Bye Packet

Participants of a session can learn about the leaving of other participants in two ways: Either by not receiving RTCP packets from the leaving participant for a certain number of report intervals, or by the reception of an explicit BYE RTCP packet. The former usu-

ally happens when applications leave the session because they are terminated in an unexpected way (software or hardware failure), the latter is commonly used when the participant leaves a session in a regular way. Sending an explicit bye packet is desirable because it enables the remaining participants to use the new (decreased) group size immediately as the basis for the calculation of report intervals. In addition the BYE packet is used when a participant chooses a new SSRC identifier because of an SSRC identifier collision.

The bye packet is fairly straightforward. Following the usual RTCP header is the SSRC identifier of the participant that will quit the session. In the case of a mixer quitting a session, the SSRC identifiers of all contributing sources are included in the packet. The RTCP BYE packet may contain the reason for transmitting the bye packet in the form of an human-readable string.

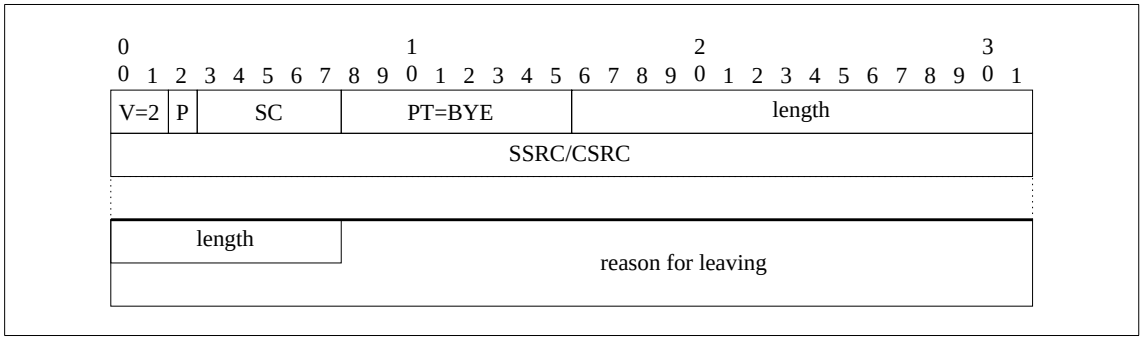

Figure 30: RTCP bye packet [40]

5.2.3 On the Use of RTP for Distributed Interactive Media

RTP has been used not only for the transmission of continuous non-interactive media like audio and video, but also for distributed interactive media (e.g., the dlb [22] and DIVE [18]). Before developing a new protocol for this media class it is therefore necessary to carefully review whether RTP is an appropriate application level protocol for distributed interactive media. Many of the arguments presented here stem from an in-depth discussion with the authors of [76].

We will start the investigation with the RTP ADU (see Figure 25). The sequence number contained in RTP ADUs is linear and non-structured. As mentioned above, one important requirement for an application level protocol that supports distributed interactive media is the presence of a *structured sequence number.* With a single linear sequence number it would not be possible to relate packet loss to an individual sub-component or ADU type (state vs. event).

While a timestamp is required for distributed interactive media, the timestamp provided by RTP cannot directly be mapped to an absolute point in time. This is only possible with the NTP-to-RTP timestamp mapping contained in RTCP sender reports. Audio and video

streams can be played even when this mapping has not yet been performed. Those media just need to know the time difference between two ADUs rather than an absolute point in real time. This is generally not true for distributed interactive media, for which it would be more appropriate to include an *absolute physical timestamp* into the ADU which can be directly interpreted.

The synchronization source (SSRC) identifier is chosen randomly in RTP. While an occasional SSRC collision might be acceptable for media like audio and video, it could have a catastrophic impact on distributed interactive media, especially when access restrictions and object ownership are related to these identifiers. Also the packet loss caused by an SSRC collision is undesirable for this media class. Distributed interactive media need a *non-volatile participant identifier* that is guaranteed not to change over the lifetime of a session.

Finally, it is very unlikely that distributed interactive media can make use of mixers that combine the ADUs transmitted by multiple participants in a single ADU. Therefore the contributing source identifiers (CSRC IDs) and the contributing source count (CC) fields are not required in ADUs of this media class.

As we have seen the RTP ADU already provides strong hints that a direct use of RTP would be inappropriate for distributed interactive media. This is aggravated when we take a closer look at the usage of RTCP. While source description and bye packets may be used directly for distributed interactive media, the *sender and receiver reports do not fit this media class*.

The sender information contained in a sender report (see Figure 27) is only of minor interest when an absolute physical timestamp is contained in the ADUs. The RTP-to-NTP timestamp mapping is redundant, and the remaining information (packet count and octet count) only provides statistical information. Sender reports therefore do not seem to be required for distributed interactive media.

The information carried in RTCP report blocks, as contained in sender or receiver reports, only provides useful information in the absence of packet retransmissions. This information becomes corrupted if RTP ADUs are transmitted reliably, either by the transport layer or by an application level reliability mechanism. Since distributed interactive media may require the use of ADU retransmissions, the report on reception quality should not be part of a general application level protocol for this media class. Instead it should be realized by whatever mechanisms are used to establish reliability. In the event that a sub-class of distributed interactive media does not use retransmissions (e.g., battlefield simulations) this functionality may be part of a profile-specific extension to the general protocol.

Summarizing it can be noted that several RTP fields and mechanism are not directly useful for distributed interactive media, while other important information is missing (e.g., sub-component identifiers, support for fragmentation, etc.) However, it is also true that many of these fields and mechanisms require only a reinterpretation (e.g., sequence number, timestamp, SSRC identifier, etc.) in order to become appropriate for this media class. One could therefore argue that the definition of a new RTP profile would suffice to fix those problems. In fact RTP/I, as described in the next section, started out as a new RTP profile. However, a closer look shows that the required reinterpretation would modify RTP to an extent that generic, profile-independent RTP tools would not be usable with the new profile. Most important is the absence of sender and receiver reports and the reinterpretation of the sequence number in RTP ADUs. We therefore consider a new protocol derived from RTP to be the right way to establish a common application level protocol for distributed interactive media. This protocol should reuse aspects of RTP whenever possible, while it needs to be thoroughly adapted to the demands of distributed interactive media.

5.3 Real-Time Application Level Protocol for Distributed Interactive Media

The Real-Time Application Level Protocol for Distributed Interactive Media (RTP/I) has been specifically designed to meet the demands discussed above: framing of event and state data, support for consistency and fragmentation, a standardized way to query the state of a sub-component, the ability to convey meta-information, and a flexible protocol design [60,63]. The full specification of RTP/I has been submitted as an Internet Draft [61] in order to stimulate the discussion of RTP/I in the Internet Community.

RTP/I reuses many aspects of RTP, including the concept of two distinct protocols for the transportation of data and meta-information. The protocol used for the transmission of medium data is called the RTP/I data protocol while the protocol used for meta-information is called RTP/I control protocol (RTCP/I). As with RTP these two protocols are carried over distinct transport addresses.

We start the presentation of RTP/I by discussing the data part of RTP/I. In a second step we move on to explain how meta-information is transported. In both parts we reuse aspects of RTP whenever it is appropriate.

5.3.1 RTP/I Data Transfer Protocol

The core data for a distributed interactive medium - states, events, and requests for state information - are carried in RTP/I data packets. Essentially RTP/I data packets contain

medium-specific information that is framed by common header fields. In RTP/I there exist four distinct data packet types: event, state, delta state, and state query.

5.3.1.1 RTP/I Event Packets

An event packet carries an event or a fraction of an event. It is structured as depicted in Figure 31. The first two bits of an RTP/I event packet contain the version number (V) of the protocol. The E (end) and the fragment count fields are used for fragmentation and reassembly of oversized ADUs. The fragment count starts with 0 for the first packet of an ADU and is increased by one for each packet belonging to the ADU. The end bit is set to one in the last packet of an ADU. Recipients of a fragmented ADU know that they have received all parts of an ADU when they have received a packet with the fragment count 0, a packet with the end bit set, and all packets in-between as identified by the fragment count.

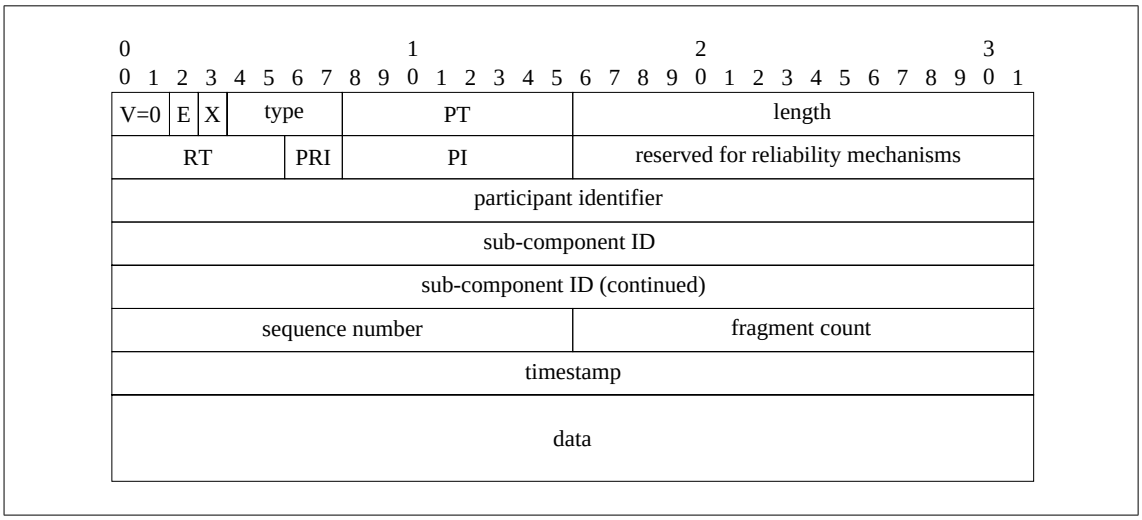

Figure 31: RTP/I data packet

The type (TYPE) field identifies the content of the packet. There are four values used by RTP/I: EVENT, STATE, DELTA_STATE, and STATE_QUERY. In the type field the values of 8 and above are reserved to identify packets used by application level reliability mechanisms, e.g., an additional packet type for the detection of tail loss. Furthermore, application level reliability mechanisms may blend into the RTP/I header. The RT field identifies the reliability mechanism that is used to transmit the packet (e.g., libsrm). The 16 bits that are reserved for reliability may be used by reliability mechanisms to store additional information. Examples could be flags to mark packets for forward error correction or retransmitted packets. If a given reliability mechanism needs more than 16 additional bits, it may append an extension header to the regular RTP/I header. This is signalled through the reliability header eXtension (X) bit. Allowing application level

reliability frameworks to blend into the RTP/I header prevents unnecessary duplication of header information and enables Integrated Layer Processing [9].

An application level reliability mechanism that is used together with RTP/I is specified in an RTP/I reliability specification document (similar to the specification of a payload type). The reliability specification document completely defines the employed mechanism and its cooperation with RTP/I. It may do so by referencing an existing specification of the reliability mechanism and by providing information on how it cooperates with RTP/I. This includes the mapping of the information provided by RTP/I (framing and meta-information) to the information required by the reliability mechanism, the specification of any additional information that is to be stored in the RTP/I framing, and the definition of any additional packet types that may be required. Each RTP/I reliability specification is assigned a unique number in the range of 2-63. There exist two predefined reliability mechanisms that can be used with RTP/I. The first one is no reliability (e.g., pure UDP over IP multicast). The second one is the usage of a reliable transport protocol that is transparent to the application. These mechanisms are assigned the numbers 0 and 1, respectively.

The PT (payload type) and participant identifier fields are similar to the corresponding fields in the RTP framework. The payload type field identifies the payload type transported in this packet (e.g., a specific shared whiteboard encoding).

The 32-bit participant identifier field makes it possible to identify participants independent of their network/transport layer addresses. In contrast to RTP the participant identifier needs to be unique and stays constant for the lifetime of a session. How to choose a unique identifier is outside of the scope of RTP/I. Possible approaches are a centralized ID server or a distributed reservation algorithm.

The 32-bit timestamp indicates the point in time when the event must be applied to the medium. Generally this value should be expressed in milliseconds of a physical clock synchronized by NTP or a GPS receiver.

Since applications for distributed interactive media might want to save transport level overhead, it is possible to transmit multiple RTP/I packets combined into a single transport packet. The length field allows to split those combined packets by specifying the number of bytes contained in an RTP/I packet.

The sub-component ID is a 64-bit value that uniquely identifies a sub-component. As for the participant identifier, the mechanism used to choose unique sub-component identifiers is outside of the scope of RTP/I. By examining this ID an application can decide whether or not it is interested in the event (depending on whether it tracks the state of the sub-component identified by the ID).

Additional information that is defined by a profile may be stored in a field reserved for profile information (PI). This allows profiles to include important information in the framing without having to add another 32-bit word to the header. The last field relevant to the event packet is the sequence number. It is increased by one for each ADU of the appropriate type, i.e., there are four independent sequence numbers for each sub-component: one each for event ADUs, state ADUs, delta state ADUs, and state query ADUs. The medium-dependent event data follows the RTP/I header, its encoding is specified in the payload type definition. The PRI (priority) field is not used by event packets.

5.3.1.2 RTP/I State and Delta State Packets

The RTP/I state packet type is used to transmit a sub-component's complete state (or a fraction thereof if the state is too large to fit into a single network layer packet). A state packet has the same structure as an event packet with the exception that an additional priority (PRI) field is used in state packets. This field is necessary because setting the state of a sub-component can be costly and might not always be reasonable for all participants. The priority field is a two-bit value following the fragment count field. It can be used by the sender of the state to signal its importance. A packet with high priority should be examined and applied by all communication peers who are interested in the specific sub-component. Situations where high priority is recommended are re-synchronization after errors, or packet loss. Basically, a state transmission with high priority forces every participant to discard its information about the sub-component and requires the adoption of the new state. A state transmitted with low priority can be ignored at will by any participant. This is useful if only a subset of communication partners is interested in the state. An example of this case are late joins where only applications joining the session might be interested in certain state transmissions.

The timestamp value of state packets has a different meaning than that for event packets. It denotes the point in time when the state contained in the packet was extracted from the medium. When such a state is applied to a sub-component of a continuous distributed interactive medium at a later time, the time difference needs to be accounted for.

Delta States, as described in Chapter Two, are supported by an RTP/I delta state packet type. The fields of a delta state packet are interpreted in the same way as for the state packet. The only differences are the content of the type field (DELTA_STATE) and that the payload specific part contains a delta state. It is necessary to be able to distinguish state from delta state packets at the RTP/I level so that generic services know what kind of information is contained within a packet.

5.3.1.3 State Query Packet

The state query packet is used by a participant to indicate that it desires the transmission of a certain sub-component's state. The state query is part of the data protocol (and not of the control protocol) for two reasons. First, it requires the same header information as the other RTP/I data packets. Second, it may make use of a reliability service that could exist for the data part of RTP/I. The structure of the state query packet is the same as for all RTP/I data packets (see Figure 31). However, it does not contain any medium-specific data.

Instead of being a regular request - as found in other protocols - a state query packet is only an indication to the receivers that a participant would like to get the state of the concerned sub-component. A regular request/reply mechanism would be inappropriate for a protocol that should work in a multicast environment. The decision whether any given receiver of a state query packet will reply is made locally by the receiver. This decision is influenced by the priority and should be made such that it will prevent a reply implosion if multicast is used. In detail the meaning of the priority contained in a state query packet is as follows:

- Priority 3 is the highest priority and is used when the request needs to be satisfied immediately, e.g., for resynchronization after errors.
- Priority 2 is used when a response is required, but a short delay is acceptable, e.g., for a late join service.
- Priority 1 is used when a response is desirable but not required, e.g., pre-fetching of sub-component state which might be needed later.
- Priority 0 is used when the state request is issued periodically, e.g., for a recording service.

5.3.2 RTP/I Control Protocol (RTCP/I)

The meta-information in RTP/I is transported similar to that in RTP by using a separate protocol over a different transport address from that for the core data of the medium. This protocol is called RTP/I Control Protocol (RTCP/I).

RTCP/I compound packets may consist of up to three packet types: source description packets, sub-component report packets, and bye packets. The source description packets and the bye packets of RTP/I essentially have the same format and meaning as in RTP, they are therefore not further discussed here. The sub-component report packet, however, contains meta-information about the sub-components present in a session. A typical RTCP/I compound packet is shown in Figure 32. The first packet in an RTCP/I compound packet is always a source description packet. The other packet types are only transmitted when required.

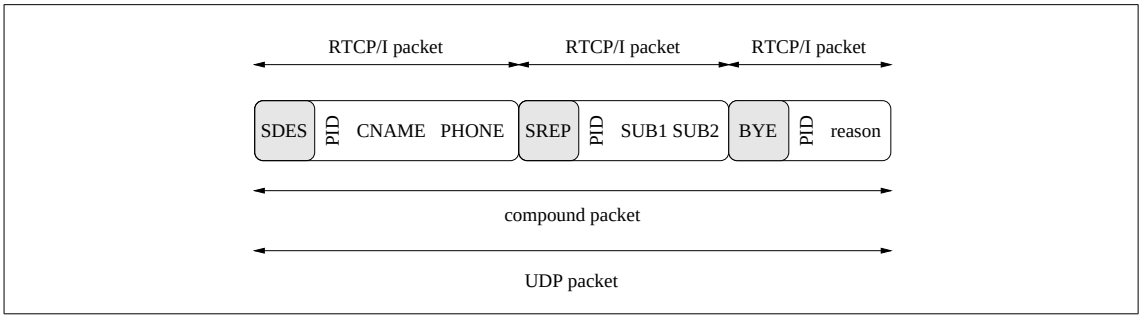

Figure 32: RTCP/I compound packet

5.3.2.1 RTCP/I Sub-Component Report Packet

The sub-component report packet is used to announce the sub-components present in a session, to indicate which sub-components are actively used to display the medium to the users, and to provide information for the mapping from sub-component IDs to application level names. For each report interval every participant checks whether any sub-components that it tracks the state of have not been reported in the last two report intervals by other participants. All unreported sub-components are then reported using a sub-component report packet. This algorithm is simple, robust, and scalable - in most cases each sub-component is reported exactly once per report interval, independent of the number of participants and packet loss.

A report for a sub-component includes its current status: active or passive. As defined above, the *active sub-components* of an application at any point in time are those sub-components that are required by the application to present the interactive medium at that specific time. It is important to note that declaring a sub-component active does not grant permission to modify anything within that sub-component. This permission needs to be granted by the application (e.g., through a floor-control mechanism). It is perfectly reasonable for an application to activate several sub-components just to indicate that these are needed for the local presentation of the medium. However, a sub-component must be activated by a session participant before that participant is allowed to modify (send external events into) the sub-component.

If a sub-component has been reported as passive by a remote application and that same sub-component is active for the local application, the local application will report that sub-component as active. This could result in two reports being sent for a single sub-component in a single report interval. However, this situation will not last since the next time the remote application checks for sub-component reports, it will see the active report and will count that sub-component as reported for the report interval.

In order to allow for the mapping of the sub-component ID to the application level name, the application level names of sub-components may be inserted into the sub-component

report packet. Application level names are optional, a payload type may specify that it does not require application level names for sub-components. A reason for not using application level names could be that the sub-component identifier already carries all the required information to let an application identify the sub-component.

As depicted in Figure 33, the sub-component report packet starts with the protocol version number (V). The name bit (N) signals whether the reported sub-components are accompanied by an application level name. The sub-component count (SC) field holds the number of sub-components reported in this packet; note that one sub-component report packet can carry information about several sub-components. The type field is used to distinguish different RTCP/I packet types (sub-component report packets, source description packets, and bye packets). In order to enable the combination of several RTCP/I packets into one compound packet, a length field is included in each RTCP/I packet. The length field is followed by the participant identifier of the participant sending the packet. The main part of the packet starts with a list of active (A) flags. An active flag is set if the corresponding sub-component is active for the sender. Finally the packet contains a list of sub-component IDs that may be followed by their application level names. The sub-component IDs appear in the same order as the active flags, so that a receiver of the packet can associate the appropriate active flags with a sub-component ID. The reason why the active flags are treated separately from the remaining information about sub-components is to minimize the size of the packet. Alternatively the active flag could be placed in a byte before the length field of the sub-component information. However, this would result in 7 unused bits per reported sub-component.

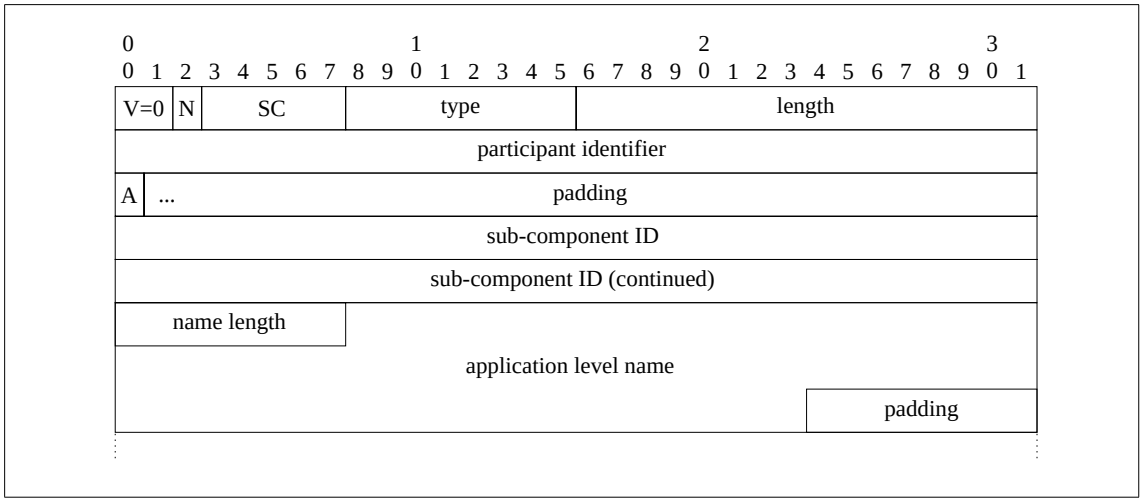

Figure 33: RTCP/I sub-component report packet

With the description of the sub-component report packet the presentation of RTP/I is complete. In the following section we will show how application level reliability could support RTP/I by providing an appropriate interface to the reliability mechanisms.

5.4 RTP/I and Application Level Reliability

In Chapter Two we have discussed the interface of the libsrm library. Our particular concern was that an application using libsrm has to be able to retransmit outdated ADUs. This leads to the problem that either ADUs are buffered indefinitely by the application or that the application has to employ a persistent data model from which any previously transmitted or received ADU can be extracted. In the following we discuss what the interface of an application level reliability library could look like that supports RTP/I-based applications in an appropriate way. The actual mechanisms used to establish reliability are outside of the scope of this work; for details the reader is referred to [47].

It is important to note that there will be RTP/I based applications that use transport level reliability, as well as applications that realize reliability in a way other than relying on retransmissions. These applications do not use a library as described here. Instead they directly use the respective transport interface. Also, the meta-information part of RTP/I has been designed to work in a lossy environment. This is accounted for by using periodic transmission of information, even if this information has not changed. It is therefore not required to use any reliability mechanisms for RTCP/I. For the remainder of this section we consider only the data part of RTP/I for those applications that use retransmission-based application level reliability.

RTP/I has been designed so that application level reliability mechanisms can directly use the information of the RTP/I header without any need to copy the content of a packet. The common usage of header fields and the avoidance of copy operations is the key benefit of using Integrated Layer Processing. To this end RTP/I allows the reliability mechanism to link into the header of RTP/I data packets. A library for application level reliability can therefore reuse the information present in the RTP/I header, while it is still possible to extend the header with reliability-specific fields.

Application level reliability for applications using RTP/I should be adjustable for each participant on a per sub-component and per ADU type basis. For example, an application should be able to specify that it wants to receive states transmitted for a certain sub-component with a certain quality of service. Events for that same sub-component as well as states for other sub-components might be received using a different quality of service. Adjustable reliability is required since states, delta states, events, and state queries have different characteristics. While events are small and need to be received fast, states may be large and can be extrapolated into the future; they are therefore less time-sensitive than events. In addition an application may have different reliability requirements for distinct sub-components.

Generally the quality of service should be set by the receiver of a packet rather than by the sender. This allows each participant to individually select the desired quality while it also fits well with the receiver-oriented reliability approach that is used by most scalable reliable multicast approaches (e.g., SRM). For an optimal support of RTP/I we consider the following quality-of-service settings:

- **No support**. ADUs are delivered as they are received. If a single fragment of an ADU gets lost, then any remaining fragments are discarded.
- **Loss detection**. Same as above, but the application is informed if an ADU gets lost. This requires sequence number monitoring and tail loss detection.
- **Unordered, reliable delivery**. ADUs are delivered immediately after they have been received. If an ADU or a fragment of an ADU gets lost, the loss will be repaired by retransmission but the retransmitted packet may be delivered out-of-order.
- **Source-ordered, reliable delivery**. ADUs transmitted by a single participant are handed to the application in the order in which they were transmitted. If necessary, ADUs are buffered until the packet loss for previous ADUs has been repaired. Only lost ADUs of the same type, referring to the same sub-component, may cause an ADU to be buffered.

With these quality-of-service settings an application can choose to use exactly the level of reliability required for each sub-component and message type. For example, an application realizing a continuous distributed interactive medium might choose to use the ‘no support’ setting for ADUs concerning sub-components that it currently is not interested in. The loss detection setting could be used for events that target relevant sub-components. This could make sense since a retransmission of an event might take too much time to be of use to the medium. Finally the application could choose unordered, reliable delivery for states, delta states, and state queries for those sub-components whose state it tracks.

In addition to the repair of lost packets by using retransmissions, it might be desirable to use forward error correction by transmitting redundant information. This is especially useful for time-sensitive data such as events in a continuous distributed interactive medium. The amount of redundancy as well as the time available to transmit the redundancy need to be specified by the sender of the packet.

Now, consider what an actual library interface could look like. Figure 34 shows the functions that should be implemented by the library and the application. The first library call (`transmitRtpiAdu`) is used to transmit an ADU, possibly consisting of multiple packets. The redundancy specifies how much forward error correction is to be used for the transmission of the packets. The transmission interval defines the time that is available to transmit redundancy packets. Generally it is a good thing to spread the transmission of redundancy packets over a long period of time, so as to avoid burst losses from

affecting both the original data and the redundancy. However, the library needs a hint from the application as to how long this period of time may be. After an ADU has been handed to the library it is buffered there. The application is free to discard the ADU after the function call.

The `setInterest` library call allows the application to choose a quality of service for a pair of one sub-component and one message type. The quality-of-service settings can be changed during the lifetime of a session.

The application needs to implement three functions. The `receiveRtpiAdu` function is called by the library when an ADU has been received completely. In the event that the application has chosen to use the loss detection quality of service, it receives an `rtpiAduLost` message whenever an ADU gets lost for the specified sub-component/ message type pair. This function contains the participant ID of this ADU's sender, the sub-component ID, the message type, the ADU sequence number, and the timestamp of the ADU that got lost.

```
Implemented by the library:

void transmitRtpiADU(RtpiAduPacket[] packets,
                     float redundancy, int transmissionInterval)
void setInterest(RtpiSubID id, RtpiAduType type, int qos)
  with qos element of {NONE, DETECT, RELIABLE, ORDERED}


Implemented by the application:

void receiveRtpiAdu(RtpiAduPacket[] packets)
void rtpiAduLost(RtpiParticipantID pID, RtpiSubID subID,
                 RtpiAduType type, int seqNo, int timeStamp)
void rtpiCouldNotRecover(RtpiParticipantID pID, RtpiSubID subID,
                         RtpiAduType type, int seqNo, int timeStamp)
```

Figure 34: Interface for application level reliability

The final function that needs to be provided by the application is required because ADUs should be buffered for retransmission only during a limited time. This amount of time should be large enough so that it is very unlikely that a repair request will arrive after the ADU has been discarded. However, since a reply to the retransmission request cannot be guaranteed with absolute certainty, the application needs to be informed should this situation occur. The `rtpiCouldNotRecover` function is called by the library whenever an ADU that should have been recovered by a retransmission is unrecoverable. In this

case the application needs to repair the problem, e.g., using a resynchronization with a state query. It should be noted that this is an exceptional situation which should occur only on very rare occasions.

The library interface proposed here is a compromise between application level reliability and transport level reliability. It exhibits most of the beneficial characteristics of application level reliability at a minimal cost in terms of additional complexity. In particular, it allows an application to receive ADUs out of order and to set different quality-of-service levels for each pair of one sub-component and one ADU type. It does not require that the application be involved with the recovery of every single lost packet, as is the case, e.g., in libsrm. Most important is the avoidance of indefinite ADU buffering. Applications for distributed interactive media can accept that on rare occasions it may not be possible to recover an ADU since they generally have to provide a mechanism to resynchronize the state of a sub-component anyway. For example, this resynchronization is also needed to allow latecomers to join an ongoing session, as a means of recovery when an application crashes, or as support for a consistency policy (see Chapter Three).

5.5 TeCo3D RTP/I Payload Type Definition

In Chapter Four we have introduce the 3D telecooperation application TeCo3D which allows the sharing of interactive and dynamic VRML models. In this section we use TeCo3D as an example to show how an RTP/I payload can be defined for a specific distributed interactive medium. The definition of the RTP/I payload for TeCo3D can be separated into four parts: timing, reliability, the encoding of events and states for the data transfer protocol, and the usage of the control protocol.

5.5.1 Timing

TeCo3D is an application for a continuous distributed interactive medium - shared dynamic and interactive 3D objects. As such, events may be applied to the state of a medium only at the point in time denoted by their timestamp. If an event arrives late, it is expected that the application - or a generic service on behalf of the application - will repair the problem.

5.5.2 Reliability

The TeCo3D payload may be carried over diverse reliability services. However, the minimal requirements are as follows:

- states should be received reliably since they are likely to be large and consist of multiple fragments, while

- events do not have to be delivered reliably. However, it is expected that the reliability service will perform loss detection for events. This is required so that the application can repair the inconsistency that results from the loss of an event.

The payload therefore works together with reliability type 1 as well as any other reliability type which fulfills the two requirements described above. It is to be expected that forward error correction will be beneficial for the transmission of events.

5.5.3 Data Transfer Protocol

The RTP/I ADUs are employed without modification, the payload type identifier for TeCo3D data is 1, the PI header field for profile-specific information is not used. Depending on the reliability type, the header may be followed by a reliability header extension.

5.5.3.1 State and Delta State Encoding

The encoding of TeCo3D states and delta states is performed according to the encoding rules described in Appendix A and in [62]. The application level name of the sub-component (3D object) is prepended to the encoded state. This name is the file name of the root VRML file that contains the definition of the 3D object.

5.5.3.2 Encoding of Events

As depicted in Figure 35 events are encoded starting with a four-byte integer that identifies the collaborative sensor that caused the event. This number is assigned to the sensor at the time the VRML model is loaded by a user. It is part of the collaborative sensor's state and as such is transmitted to all participants of a session.

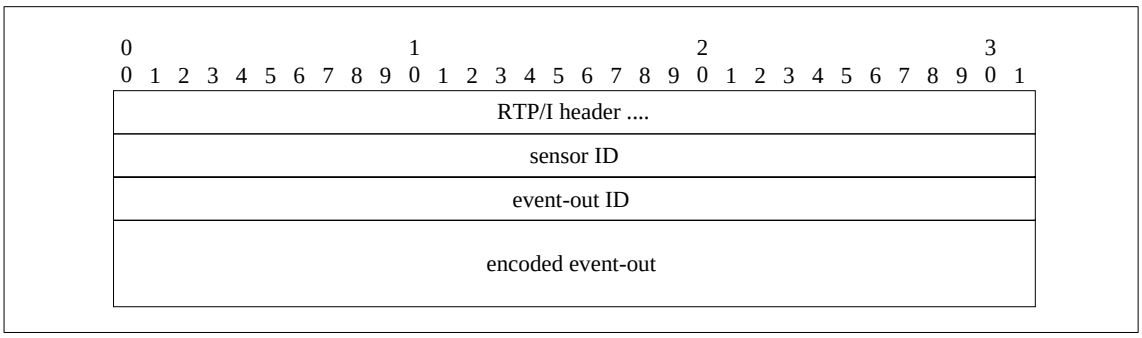

Figure 35: Encoding of TeCo3D events

Following the identifier for the collaborative sensor is the event type that is generated. This type describes which event-out of the collaborative sensor is contained in the message. As for the state encoding, the type is defined by the occurrence of that particular event-out in the VRML97 specification. For a collaborative touch sensor a possible

event-out is the `touchTime`. This event is generated when a user clicks on some part of the 3D geometry. Since the `touchTime` event-out is the seventh field of the touch sensor in the VRML specification, it is assigned the number 7. The actual value of the event-out is encoded in the same way as a field in the VRML state encoding.

5.5.4 Control Protocol

The RTCP/I protocol is used without modifications. The file name of root VRML files is used as the application level name in the sub-component report packets.

5.6 Chapter Summary

At the beginning of this chapter we gave a brief motivation why an application level protocol is the right place to establish a common foundation for distributed interactive media. The main benefits of an application level protocol are flexibility and the ability of intermediate systems to interpret the semantics of the media stream.

A detailed discussion of the design considerations for such a protocol followed the motivation. Key aspects were: distribution of events and states, support for consistency and fragmentation, a standardized way to request the transmission of states, conveyance of meta-information, and a design of a flexible protocol. We then investigated the Real-Time Transport Protocol (RTP), which is used for the transmission of continuous non-interactive media such as audio and video. While RTP is a good starting point, there are several reasons why it is not appropriate to reuse RTP for distributed interactive media. The most important areas where RTP is not appropriate are: the linear sequence number space, relative timestamps, mechanisms for mixers, volatile participant identifiers (SSRCs), and loss rate reporting.

For this reason we presented RTP/I as a separate protocol that is derived from RTP and adapted to the specific needs of distributed interactive media. RTP/I has been completely specified in an Internet Draft. Like RTP, RTP/I consists of a data transfer protocol and a control protocol. The data transfer protocol is used to encapsulate events, states, delta states, and state queries. The control protocol conveys meta-information about sub-components and participants.

Following the presentation of RTP/I we discussed how application level reliability could cooperate with RTP/I. We also proposed an interface to an application level reliability service that accounts for the special characteristics of the distributed interactive media class. Finally, as an example, we specified how TeCo3D data is transported by using RTP/I.

In the next chapter it will be shown how RTP/I can be used to develop generic and reusable services for distributed interactive media.

6 Generic Services for Distributed Interactive Media

In this chapter we show how RTP/I and the media model for distributed interactive media can be used to develop reusable functionality and generic services. Three generic services are presented and discussed in detail. The first is a consistency service that is based on the concept of local lag. It guarantees eventual consistency even in the presence of network partitioning. The second generic service allows participants to join an ongoing session. This situation is called late-join. The late-join service that we have developed is flexible and efficient. The third generic service was developed to reside within the network. It allows the synchronized recording and playback of RTP and RTP/I sessions.

6.1 Generic Services Channel

Instances of a generic service frequently need to communicate with remote instances of the same service or with remote applications. For example, a floor control service needs to communicate with its peer instances in order to request and grant the floor. A recording service might want to announce that it is replaying a recorded session, so that other applications will not disturb the replay by transmitting events. We introduce the concept of a generic services channel to provide a standardized way in which generic services can communicate.

The generic services channel is a single transport session that is used for the communication needs of all generic services. While it would be conceivable to have a separate transport session for each service, this could easily lead to network address clashes and wasted resources. The traffic that is carried over the generic services channel is restricted to the information that the instances of generic services need to transmit in order to provide their functionality. A typical example for this kind of information is the request for a sub-component's floor.

It can be expected that this control information will consume only a a very small amount of bandwidth. However, the control information is usually important. For these two rea-

sons the generic services channel should transport information in a manner that ensures a high percentage of successful delivery to all participants.

Which actual methods are best used for making the delivery of information less vulnerable to packet loss remains to be specified. For the remainder of this work we simply expect that packets sent to the generic services channel are likely to be successfully delivered. This can be achieved, e.g., by using a reliable transport protocol or by an appropriate amount of forward error correction. More sophisticated reliability support for the generic services channel is being investigated in the context of the Educational Multimedia Library (EMuLib) project [34].

One might ask why the information transmitted over the generic services channel is not distributed by using the RTCP/I protocol or an extension thereof. The reason is that the communication pattern for the generic services channel is different from that of RTCP/I. In particular, the generic services channel should offer some protection from packet loss using, e.g., forward error correction or retransmissions. In contrast, RTCP/I was developed to be used without any reliability support.

Each packet sent to the generic services channel begins with a 16-bit identifier field that indicates the service to which it belongs. The remainder of the packet is specific to the generic service. An application that receives a packet from the generic services channel should hand the packet to the appropriate service.

6.2 Local-Lag-Based Consistency Service

One fundamental and universally needed part of applications for distributed interactive media is a mechanism that keeps the state of the medium consistent across all participants. We have therefore developed a generic service for local-lag-based consistency.

The requirements that a medium has to fulfill in order to use this service are as follows:

- There is only one participant that may interact with (send events into) a given sub-component at a time. This participant may change over time, i.e., participants can take turns generating events for a single sub-component.
- Events and states need to be transmitted in a way that allows the detection of lost ADUs (this includes the case that ADUs are transmitted reliably). ADUs that get lost during a network partitioning should be reported as lost (or repaired) after the partitioning has been healed.
- The generic services channel, as described above, is required to provide the transport of messages with a high percentage of successful message delivery. When a network partitioning occurs, the generic services channel may drop messages until connectivity has been restored.

- We concentrate on developing a service for continuous media. While it can be used for discrete media as well, there may be more efficient solutions to achieve consistency in discrete distributed interactive media.

With these requirements the service presented here guarantees eventual consistency. That is, there may be short-term inconsistencies, and there may be longer inconsistencies in the case of a partitioned network. However, these inconsistencies will be repaired automatically, so that all applications will eventually reach the same shared state. We follow the concept of local lag as introduced in Chapter Three. Therefore a distributed interactive medium may decrease the number of short-term inconsistencies by increasing the amount of local lag.

In Chapter Three we illustrated that a consistency mechanism must have a way to repair short-term inconsistencies. We identified three possible solutions: state prediction and transmission, state requests, and time warp. For the service presented here we have chosen to use the state request method, it being the least restrictive. In order to prevent a heavy burden on the network, state prediction and transmission is not possible if the size of the state of sub-components is larger than a couple of bytes. Time warp, on the other hand, requires a sophisticated application that is able to handle the complex timewarp process. We expect that in the future generic consistency services will emerge that also support these two kinds of state repair.

The only requirement by the state request method is that in the case of a short-term inconsistency the state can be requested from another participant. As we have described in Chapter Three, a floor control that allows only a single floorholder per sub-component can be used to identify the participant holding a valid state. A single floorholder cannot experience short-term inconsistencies since he or she is the only one allowed to transmit events and states.

The local-lag-based consistency service therefore consists of two parts:

- a robust floor control, and
- event management and state repair.

Both parts are described in more detail in the following sections.

6.2.1 Robust Floor Control

A significant number of approaches exists to realize floor control for distributed media [32,14,53]. While the main concepts of these approaches are generally usable to support local-lag-based consistency, there are some caveats that require further consideration.

Most important is the robustness of the floor control mechanism. Since the floor control will be used to identify the participant with a valid copy of the shared state, it is of vital

importance to the overall consistency of the medium that there exists exactly one participant who holds the floor for a given sub-component. In the event that the floorholder crashes or when more than one floorholder is present for the same sub-component, there needs to be a well-defined way to repair this problem without destroying the consistency of the medium.

Furthermore, the floor control mechanism needs to be scalable. Otherwise it would limit the scope of the applications that use the consistency service. Scalability can only be achieved if message implosions are avoided during regular operations (e.g., floor handover from one participant to another) and in exceptional situations (e.g., recovery of a lost floor).

In the context of this thesis we have developed a robust and scalable floor control service for RTP/I-based applications that takes these problems into account. It is specifically designed to support the local-lag-based consistency service by identifying the participant with a valid copy of a sub-component's shared state. The main characteristics of this floor control service are as follows:

- A single floor for each sub-component is managed. Only the holder of the floor is allowed to produce events for that sub-component.
- If the floor for a sub-component gets lost, then a new floorholder is elected. The election process ensures that the participant with the "best" information about the sub-component is elected as the new floorholder. This election process is done in a scalable manner that avoids message implosion in large multicast groups.
- Network partitioning may cause multiple floorholders to exist for the same sub-component. However, through an election process a single floorholder will be elected after connectivity is restored.
- The service provides the *mechanisms* to realize floor control. The application can choose to implement different *policies* for getting the floor (e.g., implicit or explicit floor control).

Each application participating in a session has a floor state for each RTP/I sub-component. The two most common floor states are floorholder (FH) and non-floorholder (NFH). Ideally there should be exactly one participant that is the floorholder for each sub-component. The floor for a sub-component can be passed between participants by a floor handover mechanism.

Due to exceptional conditions there may be no floorholder for a sub-component, e.g., if the floorholder's application crashes. In this case, a new floorholder needs to be selected from the group of participants.

On the other hand, there may exist more than one floorholder for a given sub-component, due to network partitioning. This is possible since in each partition that does not contain

the original floorholder, an application could mistake this situation for one in which the original floorholder has crashed. The application would therefore start a selection process for a replacement of the floorholder. This may lead to multiple floorholders for the same sub-component, positioned in different partitions of the network.

The problem of multiple floorholders becomes visible at the time the partitioning of the network is repaired. At that time, a floorholder may receive events or state transmissions for a sub-component of which it is the floorholder. When this happens, a new floorholder needs to be selected from the group of participants, and the state of the affected sub-component needs to be repaired. The floor control service presented here takes these exceptional cases into account and provides a distributed floor claim algorithm that handles lost or duplicated floors.

Generally there are two interesting pieces of functionality involved in the floor control service: the handover of a floor from one application to another application and the election of a new floorholder in the case that no floorholder or more than one floorholder exists for a given sub-component. In both situations the involved messages are transmitted over the generic services channel, using multicast.

6.2.1.1 Floor Handover

Figure 36 depicts a part of the floor control service's finite state machine. It shows the behavior of a floorholder. The initial state is FH (floorholder). In this state two things may happen: RTP/I ADUs or generic services channel packets may arrive that indicate that there is another floorholder for that sub-component. In this case, the floor claim (FC) procedure is started to elect a new (single) floorholder for the sub-component. This should be a rare case.

Alternatively another participant may request the floor by means of a floor handover request (FHOReq). The FHOReq contains the ID of the sub-component and the participant ID of the sender. When a FHOReq is received, the floorholder changes to the state outgoing handover pending (OHP), transmits a response (FHOResp), and sets a timer. The FHOResp contains the participant ID of the application that requested the floor and the sub-component ID of the affected sub-component.

In the OHP state the following things may happen:

- The new floorholder acknowledges that it is now floorholder by means of a FHOAck(ack) packet. In this case the former floorholder's new floor status for this sub-component becomes non-floorholder (NFH).
- The application that requested the floor rejects it because its state for the sub-component has become invalid during the floor transfer. In this case, the floor state of this sub-component is reverted to FH.

- The timer expires. This signals that something has gone wrong with the floor handover. For example, the application that requested the floor may have crashed. This is an exceptional situation. Therefore a floor claim must be performed to elect a new floorholder.
- Finally new floor handover requests from other participants may arrive. These are rejected, since the floor is in the process of being transferred. Those applications that receive a reject may try again to get the floor after they have waited for some time.

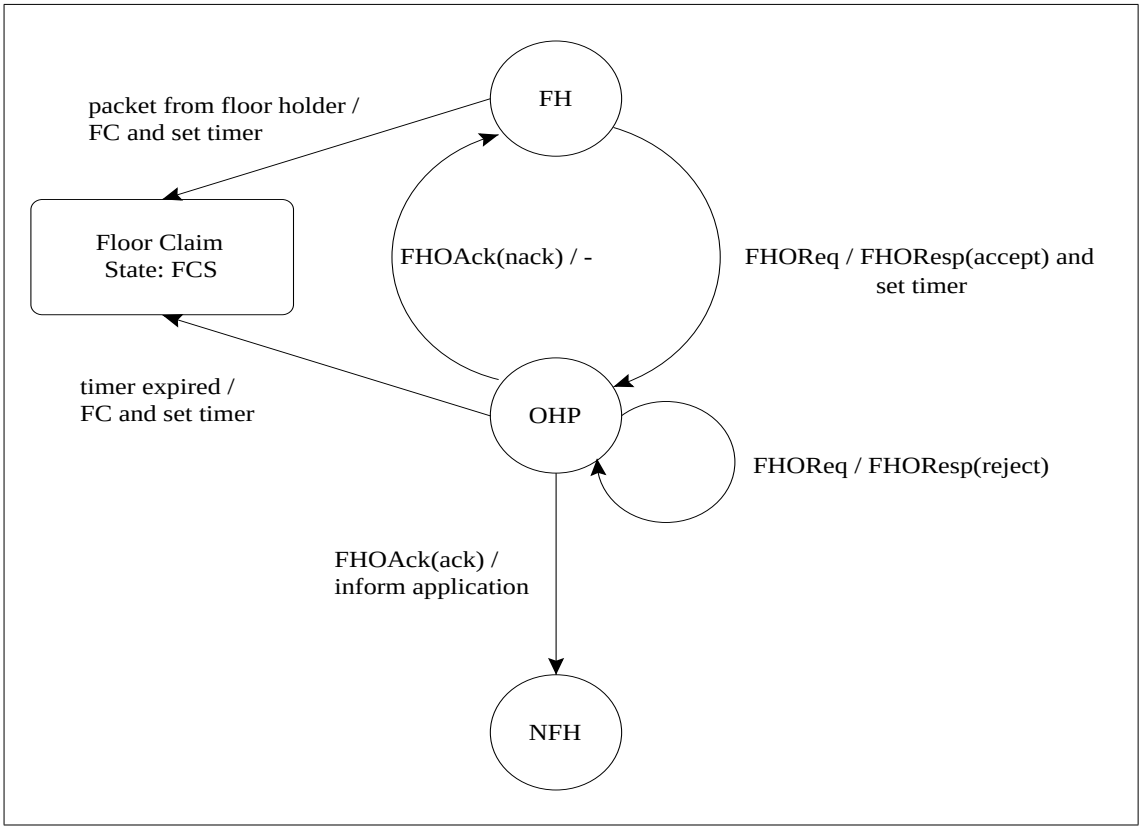

Figure 36: Floor handover (floorholder)

The behavior of the application that wants to get the floor of a sub-component is shown in Figure 37. The initial floor state of the sub-component is NFH. In this state the floor control service may get a floor request from the application. It reacts by sending a FHOReq, setting a timer, and changing to the incoming handover pending (IHP) state.

If the floorholder replies with a reject (e.g., because it is already doing a handover to a different participant) then the floor requester returns to the initial NFH state and informs the application. If the floorholder answers with accept, then the floor control service waits for a brief period of time. This is required so that the loss of events that have been issued by the previous floorholder just before the floor handover, can be detected by the requesting application. After this period of time the current state of the sub-component is checked. If the state is free of short-term inconsistencies and no lost event has been reported, then the floor control service transmits a FHOAck(ack) message and becomes

the new floorholder. If the check fails, then the service rejects the floor with a FHOAck(nack) message and reverts to the NFH idle state. The application must then wait until the state of the sub-component has been repaired before another attempt may be made to get the floor. In the event that the timer expires, a floor claim is initiated to select a new floorholder. This is done when a floor has been lost.

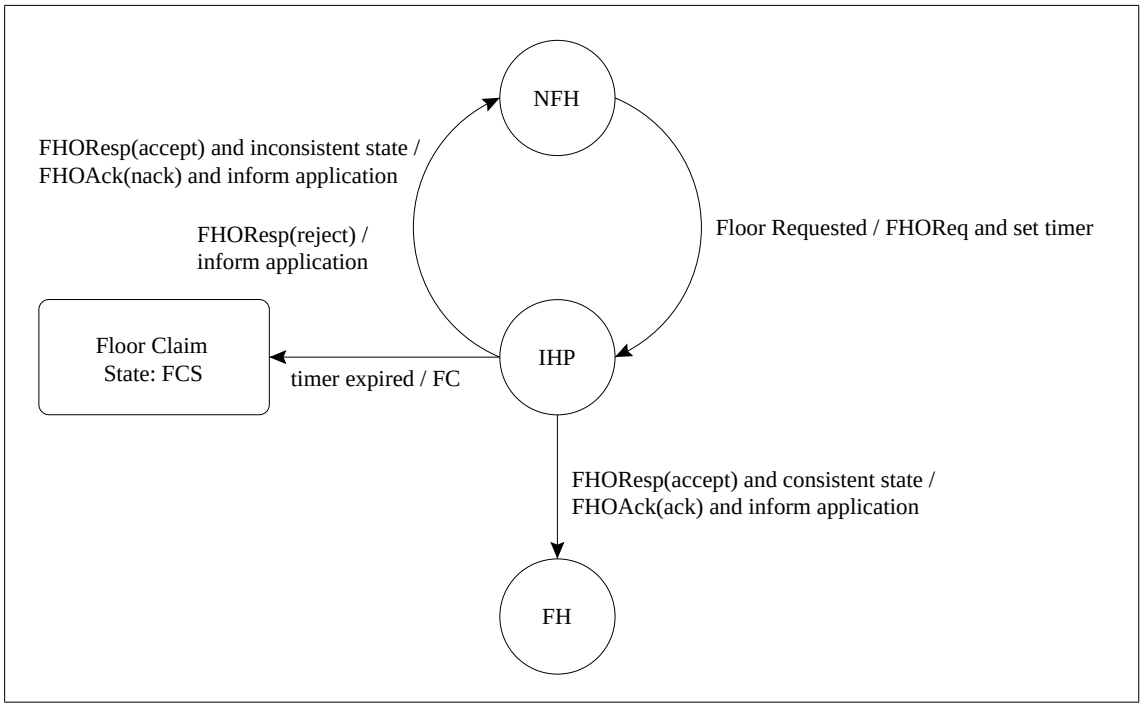

Figure 37: Floor handover (non floorholder)

On rare occasions a lost event from a previous floorholder may be reported as lost only after the floor handover has already been completed. In this case the period of time that the new floorholder waited to ensure that such reports arrive before assuming the role of the new floorholder was not long enough. This does not pose a problem since the new floorholder will treat this situation as if a second floorholder for this sub-component had been detected. It will initiate a floor claim that is regularly followed by a state repair in form of the transmission of this sub-component's state with a high priority. The inconsistency will therefore not last.

6.2.1.2 Floor Claim

A floor claim may be initiated for a variety of reasons. Two have been identified above: when a floorholder detects another floorholder for the same sub-component, and when a floor is requested and no floorholder has answered. In addition the application may initiate a floor claim if state query ADUs have not been answered.

However, there is one caveat about the application initiating a floor claim: While a floor handover is in progress, no participant is able to reply to state queries since for the short

time required to perform a floor handover no floorholder is available. A single state query that received no reply is therefore no indicator that the floor for a sub-component got lost. For this reason a floor claim should be initiated by the application only after multiple state queries have failed.

The floor control service of the application that wants to initiate a floor claim transmits a floor claim packet, sets a timer, and enters the Floor Claim Sent (FCS) state (Figure 38) for that sub-component. The floor claim (FC) packet contains the following information:

- the sub-component ID,
- a bit that indicates whether the sender was floorholder before the floor claim was initiated,
- a bit that indicates whether the sender has a valid state for the sub-component (e.g., no missing events and no short-term inconsistencies), and
- the participant ID of the application sending the floor claim.

Based on this information, any other participant is able to decide whether it has a higher priority to claim the floor than the sender. The primary idea is that the floor should be assigned to the participant most likely to have a current state.

In detail the priority rules are as follows: Anyone who held a floor before the floor claim was sent/received has a higher priority than anyone who did not hold the floor. If there is more than one floorholder participating in the floor claim, the one with the highest participant ID has the highest priority. This additional criterion is required to break ties.

If there are no floorholders, anyone who holds a valid state has a higher priority than anyone who does not. If there are multiple participants with a valid state, the one with the highest participant ID has the highest priority. If only participants are available who have an inconsistent state, the one with the highest participant ID is the one who gets the highest priority.

These rules form a full ordering relation on all the participants of a session. The algorithm used for the generic floor control services assigns the floor to the participant with the highest priority in a scalable manner (e.g., without message implosion).

As depicted in Figure 38, a floor control service that receives an FC checks whether its own priority is higher than that contained in the received FC. If this is not the case, the floor state becomes WFFT (wait for floor taken). This state transition exists for every floor state. In the WFFT state the floor control service waits for the participant with the highest priority to transmit a floor taken (FT) message. When this message is received the new floor state becomes NFH.

Upon entering WFFT a timer is set. The timer expires in the event that the participant who was elected to become new floorholder does not send a FT message (e.g., because it

crashed). In this case the floor claim process is restarted by transmitting an FC message that includes a special restart flag. Participants who receive a message with the restart flag set will act as if the floor state for that sub-component were FH/NHF. In order to avoid a message implosion, the WFFT timer includes a random component.

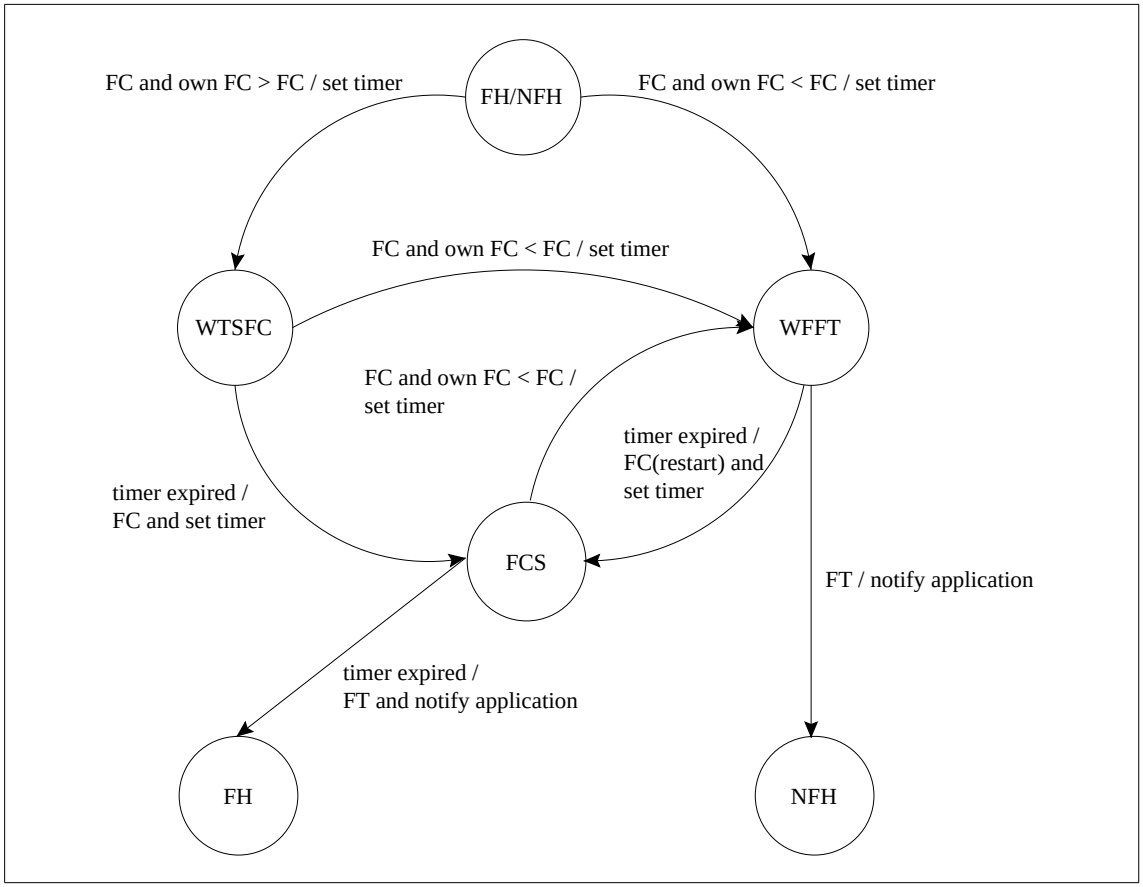

Figure 38: Floor claim

When in the FH/NFH idle state an FC is received whose priority is lower than the own FC priority then a random timer is set, and the floor state becomes “wait to send floor claim” (WTSFC). This random timer prevents a message implosion that otherwise would occur if each participant with a higher priority than the one who initiated the floor claim would send its own FC immediately. The algorithm presented here voluntarily delays the transmission of the FC by a random amount. Therefore FC’s will be distributed over time and applications can suppress their own FC when an FC with higher priority is received while the random timer is running.

If the timer expires, a Floor Claim (FC) is sent, the floor state of the sub-component is set to “floor claim sent” (FCS) and a new timer is set. This timer is deterministic. It is required so that remote applications where the floor state of that sub-component is still WTSFC and that have a higher priority than the local application can still send their own FC before the floor claim process is finished. Therefore the value for the timer must be larger than any remaining random timer plus the maximum transmission delay from any

remote participant to this participant. Since this is not a deterministic value (FC messages may get lost and retransmitted) a sufficiently large constant value is used for the timer.

When the timer expires in the FCS state, the local application becomes the floorholder. It signals this to its peers by transmitting a “floor taken” message (FT). Under exceptional conditions (e.g., a large amount of packet loss in the control channel) it may be possible that more than one participant will take the floor in a floor claim. In particular, this may happen if an FC message is delayed so that the timer in the floor-claimed state expires before delivery of the FC message. This problem should be very rare and is solved by initiating another floor claim round when an FT message is received in the FH state (this is not shown in Figure 38). Similarly a participant may receive an FT message while it is not in the FH or the WFFT state. In this case it should move directly to the NFH state, i.e., it is not required to initiate another floor claim round.

A floor claim is an exceptional situation that is almost always accompanied by an inconsistent shared state. Therefore the floor control service notifies the application upon completion of the floor claim. The application of the new floorholder should then repair the inconsistency by transmitting a state of the affected sub-component with a high priority. In the event that a participant with an inconsistent state was chosen as the new floorholder, the application may decide what actions to take. Generally it could delete the sub-component or it could regard the state as a legal and consistent, transmitting it with high priority to all peers.

6.2.1.3 Floor Control API

The API of the robust floor control is depicted in Figure 39. The application can query whether it currently holds the floor for a sub-component. It should do so for outbound states and events, in order to check whether it holds the floor for that sub-component. If it does not hold the floor, then the event or state must not be transmitted and may not be applied to the local model. Incoming events and states should also be checked to see whether two floorholders exist for the same sub-component.

The floor control service can be instructed to get the floor of a sub-component in two ways: either using a regular `getFloor` primitive or by using a `getAndLockFloor` primitive. The second prevents the floor control service from releasing a floor when requested to do so by peer applications. Instead of sending a FHOResp(accept), the service will reply FHOResp(reject) to requests for floor handover. The lock can be released by calling `freeFloor`. Finally the application may initiate a floor claim for a sub-component by calling `initiateFloorClaim`.

The floor control service expects that the application provides a number of methods. First there are notification methods for receiving and losing a floor. Also the application is notified about a completed floor claim. Finally there is a method required that checks whether the application has a valid state of a sub-component. `weGotValidState` returns true if the sub-component is present and there are currently no short-term inconsistencies or lost events for the sub-component.

```
Implemented by the robust floor control service:
boolean hasFloor(long subID)
void getFloor(long subID)
void getAndLockFloor(long subID)
void freeFloor(long subID)
void initiateFloorClaim(long subID)

Implemented by the application:
void floorReceived(long subID)
void floorLost(long subID)
void floorClaimResolved(long subID, boolean weGotFloor)
boolean weGotValidState(long subID)
```

Figure 39: Robust floor control API

6.2.2 Ensuring Consistency

Based on the robust floor control is the generic local-lag-based consistency service. When an event is produced by the floorholder, the application assigns it a timestamp that includes a certain amount of local lag. In respect to the consistency mechanism presented here, the amount of local lag may be different for each event. Usually, however, the local lag will be a constant value for a given medium that has been determined using the process described in Chapter Three.

The event is transmitted by the floorholder and, at the same time, handed to the generic synchronization service (see Figure 40 for the API calls). When a remote event is received by a non-floorholder it is also forwarded to the synchronization service. As depicted in Figure 41 the synchronization service buffers RTP/I event ADUs in a real-time queue. Events are forwarded from the synchronization service to the application when the time denoted by their timestamp is reached.

At the time the event is inserted in the real-time queue its timestamp is checked. If the timestamp is earlier than the current time then the event has arrived late. This can only happen for remote events. In this case a short-term inconsistency has occurred that needs

to be repaired by requesting a correct state from the floorholder. Similarly if the consistency service is notified about a lost event a state query is triggered.

```
Implemented by the robust consistency service:
void putEvent(RTPIEventADU adu)
void stateReceived(RTPIStateADU adu)
void eventLost(RtpiSubID subID, int seqNo, int timeStamp)
boolean weGotValidState(long subID)

Implemented by the application:
void initiateFloorClaim(long subID)
void transmitStateQuery(RTPIStateQueryADU adu)
void injectEvent(RTPIEventADU adu)
```

Figure 40: Local-lag-based consistency service API

If the time denoted by a buffered event ADU is reached, then it is handed to the application. The application may then apply the event to the state of the appropriate sub-component. If more than one event bears the same timestamp, then they are applied in the order of their sender's participant ID. This ensures that the events are executed in the same order by all the participants of a session.

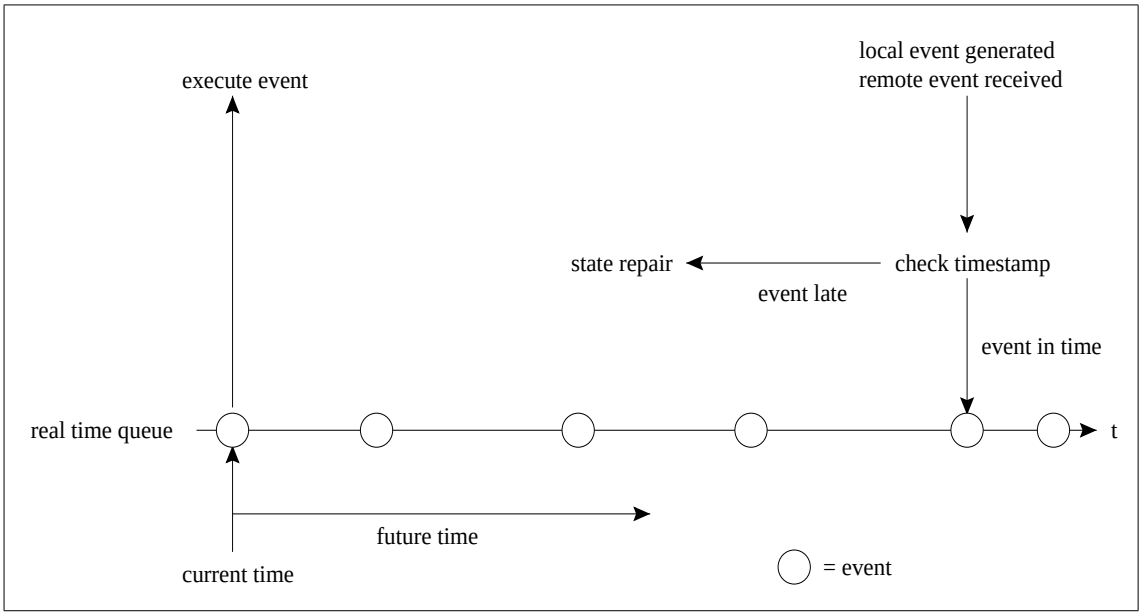

Figure 41: Real-time queue

When the state transmitted by a floorholder is applied to a sub-component of a non-floorholder (e.g., to repair a short-term inconsistency), the synchronization service is informed about the timestamp of that state. If an event has overtaken the state transmission, i.e., the timestamp of the event is greater than that of the state, and the event has

already been due for playout, then the new state of the sub-component needs to be repaired by another state query.

Any event with a timestamp smaller than that of a state that has been applied to a sub-component is already incorporated into the state. Those events will be ignored, they do not trigger a state query if they are reported as lost or arrive late.

The generic local-lag-based consistency service is fairly simple. Yet it guarantees eventual consistency. The reader should note that it is based solely on the common media model and information provided by RTP/I. It does not use any medium-specific information. The service can therefore be used for arbitrary distributed interactive media that fulfill the requirements as stated above. For individual media and media classes there may be more efficient ways to realize eventual consistency. Especially the time warp approach is very attractive. The algorithm presented here should provide a good starting point and a fall-back solution if other improved and specialized services are not available for a given medium.

6.3 Generic Late-Join Service

The term *late-join* is used to denote a process which allows latecomers to join and participate in an ongoing session. This is typically achieved by getting the state of the medium from the current participants and by initializing the application of the latecomer with this information. In the following we will use the term *late-join client* for the application that requests a state initialization; the application that is responsible for the transmission of the current state at a certain point in time is called the *late-join server* [25]. The role of the late-join server may be assigned dynamically and may differ for individual sub-components.

In general a late-join service has to perform two tasks:

1. It must identify those sub-components that are needed by the latecomer to participate in the ongoing session.
2. It needs to provide the late coming application with the state of those sub-components at the appropriate point in time in an appropriate way.

The first task is important since a large part of the medium's sub-components may not be immediately relevant for a latecomer. In a shared whiteboard session, for example, only the state of the current page may be required to enable a latecomer to participate in the session. The state of other pages may be needed only when they become visible later on. A solution to the late-join problem therefore needs a way to explore which sub-components are available in a session and under which conditions the state of a certain sub-component is required.

Once it is decided when the state of a sub-component is needed the second task is to retrieve the state of the sub-components at the appropriate time. This is a complex task, since the state of sub-components may be required under different circumstances, e.g., immediately (the current page of a shared whiteboard), or later on triggered by some user action (an old shared whiteboard page that becomes visible again). Furthermore the state information needs to be retrieved in a way that does not excessively burden the network or the participating applications. This is of particular interest and complexity since distributed interactive media regularly involve a large group of users.

The generic late-join service presented here has been designed to be useful for arbitrary applications that use RTP/I [95]. It is highly customizable to the needs of diverse applications. Furthermore it minimizes the burden that is placed on the network and the participants of a session. The generic late-join service does not rely on the existence of a specific consistency mechanism (such as the one described above). It merely expects that a consistency mechanism be present that is adequate for the needs of the application.

6.3.1 Requirements for a Generic Late-Join Service

Derived from the two main tasks, as well as from general design aspects of distributed applications, the following requirements for a generic late-join service can be identified:

- **Robustness**. The failure of individual components such as a participant's computer or application should not prevent a future late-join. Moreover, a late-join is likely to be executed after an individual component (e.g., the application) has failed, in order to let the affected user (re)join the session.
- **Low initialization delay**. The portion of the late-join that is needed to let the late coming user follow the ongoing session should be performed with an acceptable delay.
- **Low network load**. A late-join should minimize the load on the network. Only the data that is really needed should be transmitted. For example, it might be efficient for a shared whiteboard to request only the state of the currently visible pages rather than requesting the state of all the pages ever shown in the session. Furthermore, a late-join service should seek to minimize the redundant transmission of information. Especially in a multicast environment the same state information may be reused by multiple latecomers.
- **Low application load**. The participants who are not latecomers should not suffer from the data transmissions required for the late-join.
- **Medium independence**. In order to be generic, a late-join service should be customizable to the needs of different media and applications in a simple and efficient manner.

In the following section we discuss existing late-join approaches, focusing especially on how well they meet these requirements for a ‘good’ late-join algorithm.

6.3.2 Related Work

Existing late-join algorithms can be separated into approaches that are handled by reliability mechanisms and those that are completely realized at the application level. Application level late-join algorithms can be subdivided further into centralized algorithms and distributed algorithms.

Representatives of the first category are reliable multicast transport services that offer late-join functionality. Examples of such protocols are the Scalable Reliable Multicast protocol (SRM) as discussed in Section 2.3.2 and the Scalable Multicast Protocol (SMP) [25] as it is used for the dlb. A reliable multicast protocol can offer the late-join service by using its loss recovery mechanisms to supply the late-joining application with all data packets missed since the beginning of the session. The application needs to reconstruct the current state from these packets.

The usage of transport functionality to solve the late-join problem has four major drawbacks:

1. It is inefficient since a large part of the transmitted information may no longer be relevant. For example there could have been an image on a shared whiteboard page that has been deleted in the meantime.
2. It is generally more efficient to transfer the state information than to transmit all event packets that have lead to that state. When editing a text, for example, it makes sense to transmit the current version of the text rather than all the packets that contain the description of a character that has been typed or deleted. This becomes even more important when the header overhead of the packets is considered.
3. The application either has to be able to reconstruct every packet that has ever been transmitted or the transport service needs to buffer all transmitted packets until the end of the session. This is clearly not acceptable for a large number of applications.
4. The state of certain distributed interactive media may not be easily reconstructible from a simple replay of packets. The problem here is that for continuous media (such as a networked action game) an event is only valid at a certain point in time. In order to reconstruct the state of such a medium from outdated packets, the application would have to perform a time warp to the beginning of the session and then a rapid replay of states and events. It is by no means guaranteed that all, or even a significant number of, applications will be able to perform this task.

Because of these problems we generally view the simple replay of packets as inappropriate for late-join support. Instead the current shared state should be explicitly queried by the latecomer. This leads to late-join approaches at the application level.

The distinct advantage of application level approaches is the usage of application knowledge to optimize the late-join process. Centralized late-join approaches require that a single application exists that is able to act as late-join server for the shared state. When a late coming application joins the session then it may contact the state server, which will in turn deliver the relevant state information. An example where a centralized state server is used for late-join purposes is the Notification Service Transport Protocol [75].

A centralized state server results in the typical disadvantages of a centralized solution. Main problems are the existence of a single-point-of-failure (lack of robustness), high network load around the server, and high application load for the server, which might become a bottleneck. Because of these drawbacks we have decided not to use a centralized late-join server for our generic late-join service.

Distributed late-join approaches seek to avoid the problems of a centralized approach by involving multiple applications in the late-join process. In particular, many applications may be able to assume the role of a late-join server for any given sub-component. The failure of a single application can generally be tolerated without preventing a latecomer from joining the session. This is of utmost importance since the late-join is frequently needed when individual applications fail and want to rejoin the session.

The digital lecture board implements a distribute late-join approach. One innovative idea of the method used for the dlb is to employ a separate (multicast) group for the data that is transmitted to latecomers [27]. Requests for the state of a page are transmitted to the regular session by the late-join clients. Replies are sent to the late-join group by the applications that act as late-join servers. An application leaves the late-join group once it has received all required information about the shared state. A reply implosion of the potential late-join servers is prevented by using an anycast mechanism based on random timers, similar to the one in SRM.

The approach used for the dlb has several interesting properties. First, it limits the burden of the late-join activity that is placed on participants who are not latecomers. These applications just need to check whether they have been selected as late-join servers. The transmitted state information is only received by those applications that have not yet finished their late-join activity. Second, for the same reason the approach also minimizes the network load: state information is only transmitted over those parts of the network that lead to an application that still lacks late-join information. Third, the dlb late-join prevents a reply implosion when more than one participant is able to act as a late-join server.

However, there are also some areas in which the dlb solution can be further improved:

1. In addition to preventing a reply implosion, a request implosion should also be avoided. This is particularly important if a user action (such as changing the page of a shared whiteboard) may trigger the need for additional state information. In this case multiple late-join clients might require the state information simultaneously.
2. Some applications that represent potential late-join servers should stay member of the late-join group even if they have completed their late-join process. Requests for state information could then be transmitted to the late-join group, so that uninvolved applications do not have to handle requests for state information. Only as a fall-back solution should the request be sent to the original group. This requires to define criteria that decide which potential late-join servers should enter or leave the late-join session.
3. The dlb approach is application dependent. It is not based upon a generic application level protocol and it is not easily customizable to the diverse needs of different distributed interactive media.

We have chosen to use the dlb approach as the basis for our generic late-join service. Upon this basis we have developed an improved and generic late-join service for distributed interactive media.

6.3.3 General Concept of the Late-Join Service

The architecture of the generic late-join service is depicted in Figure 42. The late-join service intercepts the data (events, states, delta states, and state queries) that arrive from the base session. Since this data is transmitted using RTP/I the generic late-join service can understand the semantics of this data to a degree sufficient to provide the late-join functionality. Knowledge about the medium specific encoding is not required. After examining the data the late-join service forwards it to the application.

The application transmits all regular data directly to the base RTP/I session without informing the late-join service. Late-join information is handed from the application to the late-join service. An example for this type of information is the state of a sub-component that is required by the late-join service to support a remote latecomer. The reason for handing late-join data to the late-join service instead of transmitting it over the base RTP/I session is as follows: the generic late-join service maintains an additional late-join RTP/I session. This session is used to transmit all late-join oriented data. The late-join service joins and leaves this additional RTP/I session at appropriate times. This ensures that only a small subset of all participants need to handle late-join data.

When joining an ongoing session the late-join service will learn about the sub-components that are present in a session through the RTCP/I sub-component reports. Whenever

a new sub-component is detected, the late-join service informs the application and requests information on how the late-join should be performed. The application may choose between a set of policies ranging from no action to immediate retrieval of the sub-component's state. In addition to the sub-component ID, the application may use the information which is delivered via RTCP/I (application level name and whether the sub-component is active or not) to determine which policy is appropriate for a given sub-component.

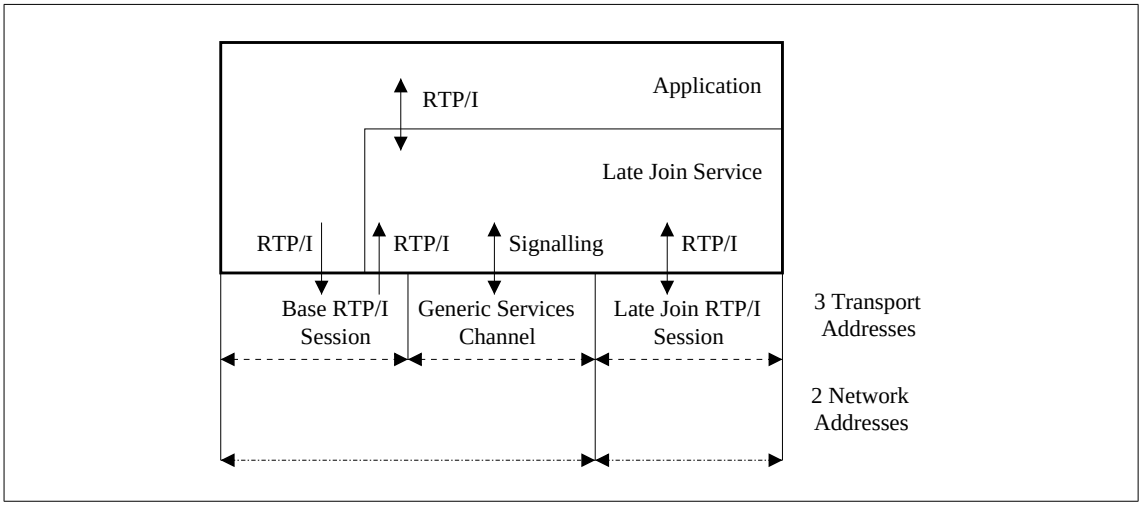

Figure 42: Architecture of the generic late-join service

When the condition occurs that was specified by the policy the state of the sub-component is requested by the late-join service. It does so by transmitting an RTP/I state query to the late-join group using an SRM style, random timer based, anycast mechanism. This mechanism makes sure that no message implosion occurs if multiple latecomers want to request the state of the same sub-component at the same time.

The request is repeated if there is no answer after a certain amount of time. If multiple requests for the sub-component's state remain unanswered it is concluded that there is no late-join server for that sub-component in the late-join group. In this case the late-join service uses the generic services channel to request that a participant who holds the state of the sub-component joins the late-join session and transmits the state. If this too fails the application is informed.

When the late-join client receives the desired state information, it hands the information to the application and marks this sub-component as complete. When there are no new sub-components detected for a period of time and all sub-components are marked as complete, then the late-join is finished. At that time the late-join client may leave the late-join group. If, at any later point in time, a new sub-component is detected, then the application may ask the late-join service to resume its duty and join the late-join group again.

6.3.4 Late-Join Policies

An application that uses the generic late-join service may specify a late-join policy for each sub-component that has been discovered by the late-join service. Setting different policies for sub-components makes it possible to quickly retrieve those sub-components which are needed immediately to present the medium to the user. Other sub-components may be assigned a policy that lets the service request them when network capacity is available or when they become important for the presentation of the medium.

Our generic late-join service offers five late-join policies:

- no late-join,
- immediate late-join,
- event-triggered late-join,
- network-capacity-oriented late-join, and
- application-initiated late-join.

An application may change the late-join policy for a given sub-component at any time. We will examine each of the five policies in the following sections.

6.3.4.1 Late-Join Policy: No Late-Join

This policy is chosen by the application to indicate that it is not interested in the sub-component. In a virtual world this policy could be used for sub-components that the user will never be able to see. By choosing the ‘no late-join’ policy the overall amount of state information that is required for the initialization of the late-join client is reduced. This has a positive effect on the initialization delay as well as on the network and the application load. When the late-join service is notified that the application has chosen this policy for a sub-component, the sub-component is marked as complete.

6.3.4.2 Late-Join Policy: Immediate Late-Join

An application may choose the immediate late-join policy when the sub-component is required immediately to present the medium to the user. An application can derive the information whether a sub-component is required immediately from the sub-component ID, from the application level name, and from the active flag contained in the sub-component report packet.

The finite state machine for the immediate late-join policy is depicted in Figure 43. The initial state is *sub-component discovered (SD)*. At the time the application chooses the late-join policy, a random timer is set and the state changes to *sub-component needed (SN)*. The random timer prevents a request implosion in the event that the sub-component be discovered by multiple latecomers at the same time. Since the discovery of a sub-

component is based upon the reception of a sub-component report, late-joiners who join roughly at the same time are likely to require the same information simultaneously. This is a good thing, since a single state transmission from a late-join server can then be used by multiple late-join clients. For this reason there is a state transition from SN to *complete (COMP)*.

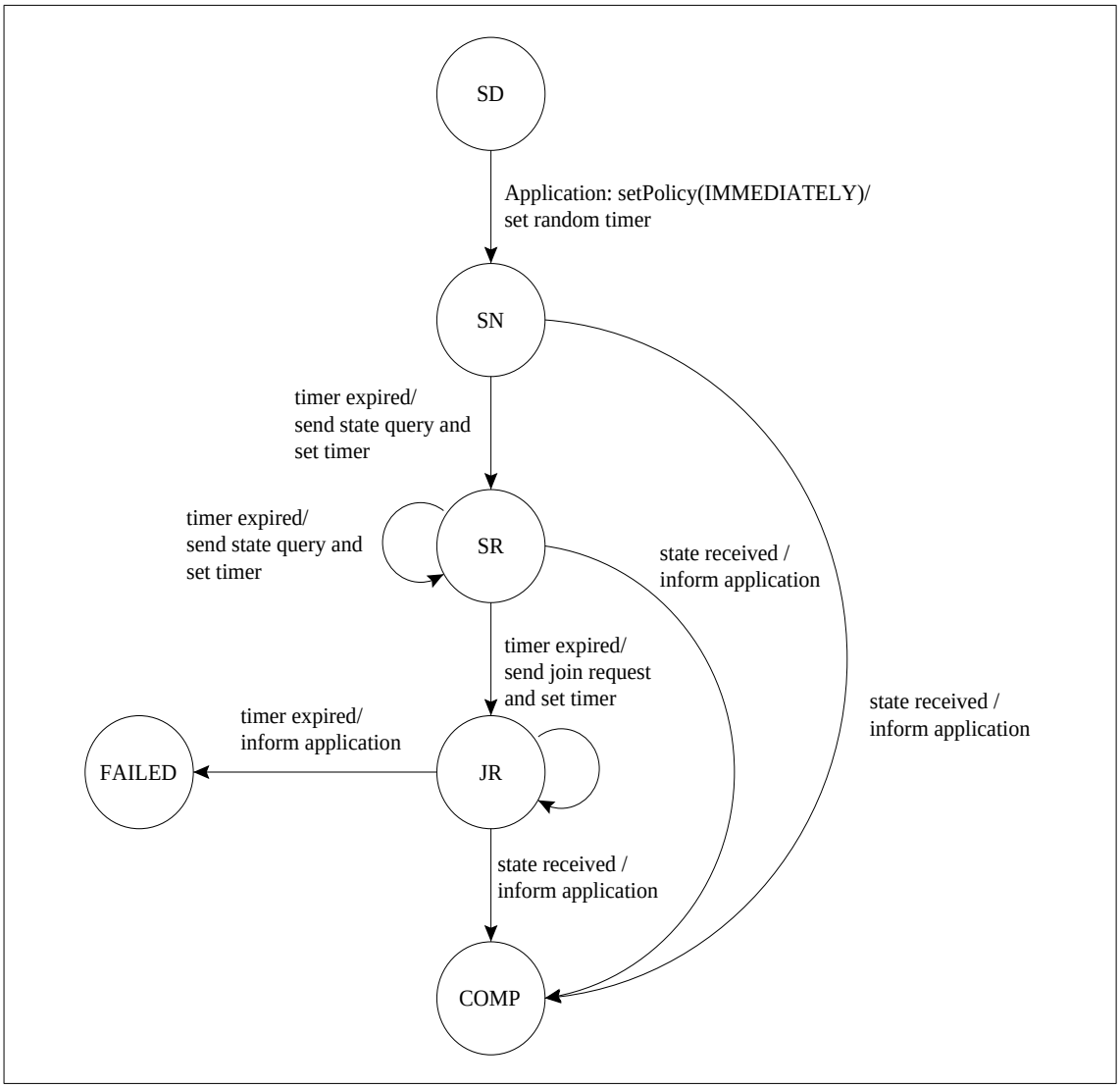

Figure 43: Immediate late-join

When the random timer expires before the state of the sub-component is received, an RTP/I state query packet is transmitted to the late-join group, another random timer is set, and the state changes to *sub-component requested (SR)*. When a reply to this request is received the late-join is complete for that sub-component. If no reply is received before the timer expires, a new request is transmitted. In the event that a remote request for the sub-component is received, then the timer is reset (this is not shown in Figure 43).

If multiple requests fail, there is no appropriate late-join server for this sub-component in the late-join group. In this case a join request is transmitted to the generic services chan-

nel. Upon receiving this request potential servers check whether they should join, using an SRM style anycast mechanism. When an application decides to join as a late join server, it also transmits the required information to the late-join group. If repeated join requests remain without answer the application is notified and the state becomes *late-join failed (FAILED)*. For the late-join service the late-join is regarded as complete if the late-join failed. An application may choose to take further steps to recover the state of the sub-component, e.g., by initiating a floor claim if the consistency service described above is used.

6.3.4.3 Late-Join Policy: Event Triggered Late-Join

An application may decide that a sub-component is required only when it is the target of an event. An example would be a shared whiteboard where a page becomes the active page by means of an activate page event. This is supported through the event-triggered late-join policy. Besides deferring the request of state information until it is actually needed, this policy also increases the likelihood that multiple latecomers may profit from a single state transmission. The reason for this is that the late-join service may refrain for a long time (until the first event for the sub-component occurs) from requesting the state of a sub-component with an event-triggered late-join policy. All latecomers who join during that period will profit from a single transmission of the sub-component's state.

The finite state machine for the event-triggered late-join policy is shown in Figure 44. The main difference to the immediate late-join policy is that there is a *wait for event state (WFE)* that follows the initial subcomponent discovered state. In the WFE state the late-join service listens to incoming RTP/I ADUs and checks whether they contain an event or a state for the sub-component. If a state is received then the state is handed to the application and the late-join state becomes complete (COMP) for the sub-component. This may happen when a state is transmitted for some reasons (e.g., as means of resynchronization) in the base RTP/I session.

When an event for the subcomponent is received, a mechanism similar to the one described for the immediate late-join policy is used. In the event that the request for state information fails, the application is informed and the state is set to WFE. Therefore a new event for the sub-component triggers another request round. This makes sense since an event indicates that the problem has been repaired and there should be a late-join server available.

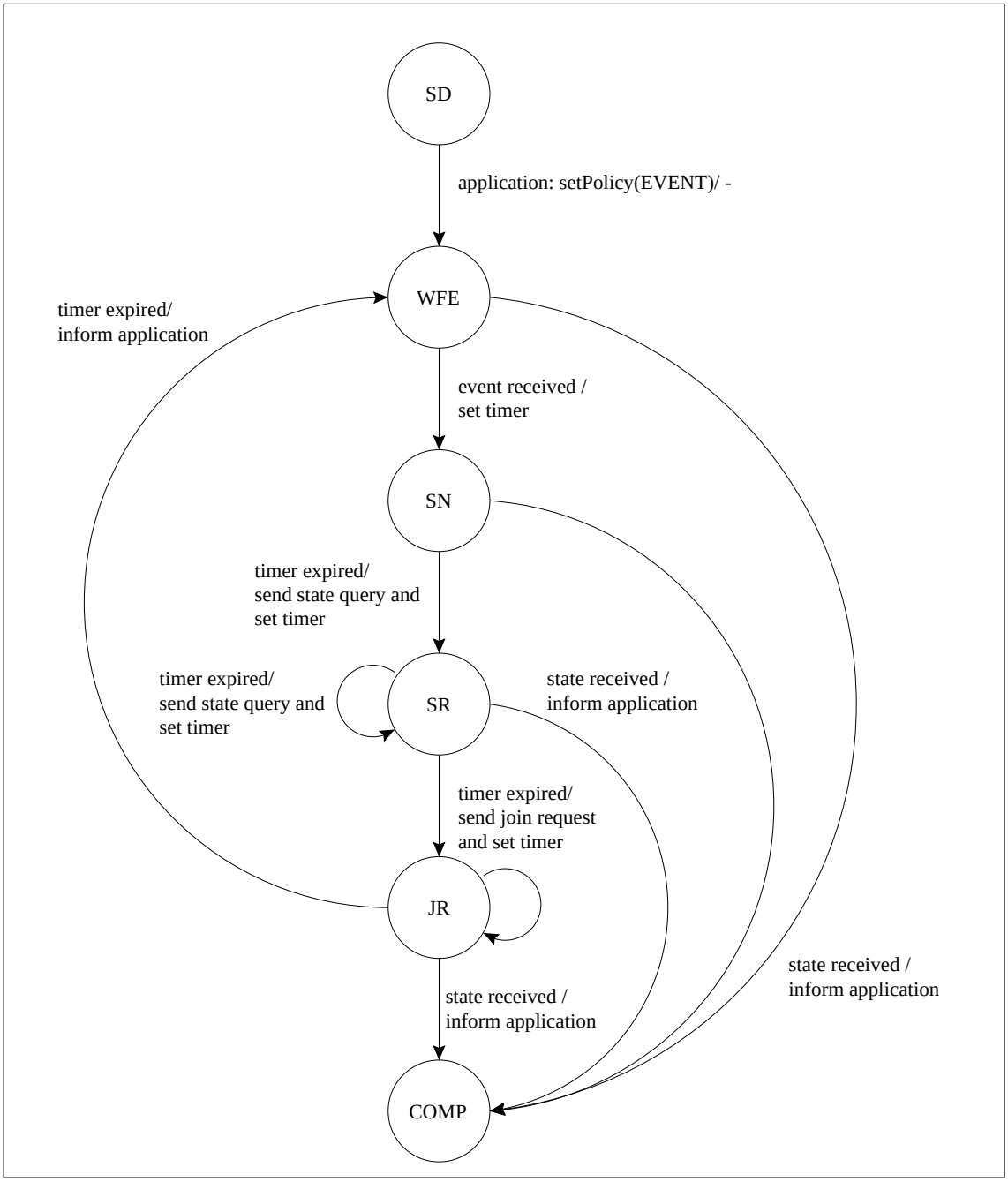

Figure 44: Event-based late-join

6.3.4.4 Late-Join Policy: Network-Capacity-Oriented Late-Join

For sub-components where the state is not immediately required an application may choose the network-capacity-oriented late-join policy. With this policy the late-join service monitors the incoming and outgoing network traffic of the application. This replaces the WFE state from the event-based late-join policy. If the traffic falls below a threshold provided by the application then a transition to the *sub-component needed* state is performed. Whenever a state query is about to be transmitted, the network traffic is checked.

Only when it is still below the threshold is the query actually transmitted. Otherwise the query is delayed further. In all other aspects the network-capacity-oriented late-join policy is identical to the event-based late-join policy.

6.3.4.5 Late-Join Policy: Application Initiated Late-Join

At any point in time the application may choose to change the late-join policy of a sub-component. In this way it is possible to upgrade policies like no late-join or network-capacity-oriented late-join to any other policy if this becomes necessary.

The application can define new late-join policies by setting the late-join policy for a sub-component to no late-join. When the application-defined policy indicates that the state of the sub-component should be retrieved the application can change the policy to immediate late-join. This should be used only for experimental purposes. If another late-join policy becomes important for certain applications, then the late-join service should be expanded to include this new policy.

6.3.5 Joining and Leaving the Late-Join Session

Unlike other existing approaches, our generic late-join service allows a small number of applications which have completed their late-join process to stay in the dedicated late-join group. If these applications are chosen well they can assume the role of late-join servers for future latecomers, while the vast majority of applications (those that have completed their late-join process and that are not members of the late-join group) are completely uninvolved in the late-join process.

This approach raises the question of who should be a member of the late-join group. Obviously all applications that have not yet finished the late-join process for all sub-components stay in the late-join group. A late-join service could theoretically leave the late-join group as soon as the late-join for all sub-components has been completed. However, it should not do so without further consideration since this could leave the late-join group without a late-join server for certain sub-components. This would increase the time that a late-join client has to wait before it gets the state of a sub-component. It is therefore important to define an algorithm that decides which applications should stay members of the late-join group, even if they have completed their own late-join for all sub-components.

There are a number of criteria that need to be considered for an algorithm that decides whether an application should stay in the late-join group or not:

- **Group size**. The late-join group should be as small as possible. The smaller the group, the less network traffic is generated, and the fewer the applications that are

burdened with late-join management. A small group also decreases the likelihood that more than one late-join server will reply to the state query of a late-join client.

- **Sub-component presence**. Optimally each sub-component for which the state is likely to be requested should have a late-join server in the late-join group. This reduces the initialization delay for late-join clients.
- **Group invariance**. The number of join and leave operations should be small for the late-join group since each of these operations is associated with overhead at the network layer (e.g., multicast routing).
- **Simplicity**. The applications should be able to perform the algorithm with a minimal overhead in computation and communication.

Generally there are three different approaches to decide which applications should remain in the late-join group to act as late-join servers: distributed, isolated, and application-controlled.

6.3.5.1 Distributed Membership Management

In distributed membership management the applications exchange information about their capabilities to act as late-join servers. This can be done via the generic services channel. With this information an optimal set of potential late-join servers can be determined. For example, the participants who are able to provide late-join server functionality for many sub-components could be preferred as members of the late-join group.

The main drawback of the distributed membership management is its complexity. Applications need to exchange additional information to allow for this kind of membership management. This information needs to be transmitted and processed, which may lead to a significant overhead, especially for large sessions. For these reasons we have chosen not to use distributed membership management for our generic late-join service.

6.3.5.2 Isolated Membership Management

Isolated membership management seeks to avoid additional messages and processing overhead by using local information. Each application decides on its own whether it should join or leave the late-join group. Isolated membership management increases simplicity at the cost of a slight reduction in the other quality criteria.

The generic late-join service presented here provides isolated membership management. Applications leave the late-join group by means of a ‘smart timeout’ and they enter the late-join group upon the request of a late-join client.

(a) Leaving the Late-Join Group

An application leaves the late-join group if it has not answered any state queries for a certain amount of time. This amount of time is not fixed, instead it is calculated based on three values:

1. An average late-join group membership time provided by the application. In this way the application can provide a hint to the late-join service on how fast applications should leave the late-join session.
2. The number of sub-components that the application can serve as a late-join server compared to the total number of sub-components present in the session. In this way applications that can serve a large percentage of the sub-components will stay longer in the late-join group.
3. The number of late-join state queries that could have been answered by the late-join server compared to the number of late-join state queries that actually have been answered by this application and not by some other late-join server. The lower this percentage is, the less important is the presence of the application in the late-join group.

When the timer expires the application leaves the late-join group. It may therefore happen that the late-join group does not contain a late-join server for a given sub-component. If there is no late-join activity for a prolonged time the late-join group may even become empty. Generally this is a good thing, since it saves resources in the event that late-joins are not frequent. However, there must also be a way to allow applications to rejoin the late-join group if a new late-join client appears.

(b) Joining the Late-Join Group as a Server

As described above, a late-join client transmits a message to the generic services channel if the state queries for a sub-component remain unanswered in the late-join group. All applications that are able to become a late-join server for this sub-component use a message-implosion avoidance algorithm similar to that of SRM to choose who will actually join the late-join group. The generic service of a selected application transmits an acknowledgment to the generic services channel, joins the late-join group and transmits the requested state of the sub-component(s).

6.3.5.3 Application-controlled Membership Management

In some cases the application may want to decide who should join the late-join group rather than leaving this decision to the late-join service. For example, in a medium that uses the generic consistency mechanism described above, only the floorholder can trans-

mit the state of a sub-component. Since there is only one candidate for joining the late-join session, it would be wasteful to use an implosion avoidance mechanism. Therefore our late-join service allows the application to specify that it should immediately enter the late-join group if the state of a certain sub-component is requested.

In order to determine when an application should leave the late-join session, the smart timeout mechanism is also used for application-controlled membership management. This is reasonable since an application will generally not be able to determine with a higher accuracy when the application is no longer needed as a late-join server.

6.3.6 Generic Late-Join Service API

The interface to the late-join service is depicted in Figure 45. The first two functions are called when RTP/I ADUs and RTCP/I packets are received for the original RTP/I session. Based on this information the late-join service discovers new sub-components and triggers requests for the state of sub-components based on the late-join policy.

When a new sub-component is discovered, the late-join service asks the application about the late-join policy that should be associated with the sub-component. If all sub-components should be treated with the same policy it is possible to set a default policy by means of `setDefaultPolicy`. The late-join service will then refrain from asking the application about late-join policies for individual sub-components.

```
Implemented by the generic late-join service:
void rtpiADUReceived(RTPIPacket[] rtpiADU)
void rtcpiPacketReceived(RTCPIPacket rtcpiPacket)
void setPolicy(LJPolicy policy, long subID)
void setDefaultPolicy(LJPolicy policy)
void setJoinGroupPolicy(LJPolicy policy, subID)
void setLeaveGroupBaseTime(long baseTime)

Implemented by the application:
void LJPolicy askForPolicy(long subID)
void RTPIPacket[] getSubComponentState(long subID)
void ljFailed(long subID)
```

Figure 45: Generic late-join service API

An application can call `setPolicy` at any time to assign a new late-join policy to a sub-component. This may also be called on a sub-component for which a late-join has already been completed. The late-join may be used in this way to recover the state of sub-components in a late-join policy driven manner.

The application may specify the policy for joining the late-join group and the base time for leaving it. When the late-join service needs to transmit the state of a sub-component as a late-join server, it requests the sub-component's state from the application by means of the `getSubComponentState` method. Finally the application may be informed of an unsuccessful late-join attempt with a `ljFailed` call.

6.4 Generic Recording and Replay of RTP/I Streams

The ability to record and replay sessions that involve distributed media is one of the most universally needed services. It is required to archive online meetings, to let students view remote lectures that they might have missed, to analyze battlefield simulations, and to provide a replay capability for networked computer games. The generic recording and replay service for RTP/I streams was therefore the very first generic service that was developed for RTP/I [33]. It is a combined effort of the EMuLib [34] and the TeCo3D projects at the University of Mannheim.

Unlike the RTP/I services presented so far, the recording service is not intended to be part of an application. Rather it is a middleware service that resides within the network as a special application. It may be accessed and controlled remotely to record and replay sessions of RTP/I-based applications. In particular, it allows near random access to recorded sessions. Furthermore, it is integrated with a recording service for RTP-based media. Therefore it is possible to use the generic recording service for sessions that involve, e.g., RTP-based audio and video in combination with an RTP/I-based shared whiteboard.

6.4.1 Existing Recording Services for Media Streams

Much work has been done on the recording of media streams. The rtptools [83] are command-line tools for the recording and playback of single RTP audio and video streams. The Interactive Multimedia Jukebox (IMJ) [1] utilizes these tools to set up a video-on-demand server. Clips from the IMJ can be requested via the Web.

The mMOD [74] system is a Java-based media-on-demand server capable of recording and playing back multiple RTP and UDP data streams. Besides RTP audio and video streams, the mMOD system is capable of handling media streams of applications like mWeb, Internet whiteboard wb, mDesk and NetworkTextEditor. mMOD simply records and replays the packets with no possibility to understand the semantics of the recording. However, understanding the semantics of the recorded stream is required to allow random access to the recorded media stream.

The MASH infrastructure [55] comprises an RTP recording service called the MASH Archive System [56]. This system is capable of recording RTP audio and video streams

as well as media streams produced by the MASH MediaBoard [93]. As we have discussed in section 2.3.2, the replay of MediaBoard sessions is possible since the application is forced to keep a complete change history of the whole session. An archive system can therefore simply dump the data to a storage device. The MASH Archive System supports random access to the MediaBoard stream but does not provide a recording service generalized for other interactive media streams. Generalizing the MediaBoard approach would require that all recordable applications adopt the MediaBoard's strategy of keeping a complete change history. This does not seem appropriate.

A different approach is taken by the AOF tools [3]. The AOF recording system does not use RTP packets for storage but converts the recorded data into a special storage format. The AOF recorder grabs audio streams from a hardware device and records the interactive media streams produced by either a specific whiteboard called the AOFwb or the Internet whiteboard wb. Random access as well as fast visual scrolling through the recording are supported but the recordings can only be viewed from a local hard disk or CD. Other interactive media streams can not be recorded.

In the Interactive Remote Instruction (IRI) system [54] a recorder was implemented that captures various media streams from different IRI applications. In all cases a media stream is recorded by means of a specialized version of the IRI application that is used for live transmission. This specific application performs regular protocol action towards the network but stores the received data instead of displaying it to the user. For example, a specialized version of the video transmission tool is used to record the video stream. Such a specialized recording version must be developed for each IRI tool that is to be recorded.

The Cornell Lecture Browser [72] is a systems that allows the non-invasive and structured recording of lectures. The lecturer does not have to aid the recording process and is not required to use a digital whiteboard tool. During the lecture the audio and video of the lecture is captured by the Lecture Browser. After the lecture has finished, the lecturer provides a digital version of the slides that have been presented during the lecture. The Cornell Lecture Board is then able to combine and synchronize the slides and the recorded audio and video streams in order to provide a high quality recording of the lecture. While the Cornell Lecture Browser is a very interesting system for the recording of lectures, it does not address the problem of recording distributed interactive media. Any annotations made to the slides during a lecture are only visible in the captured video. In contrast, our aim is to record the data in its original format. This provides a higher quality than a captured video and it allows for further processing and usage of the data.

6.4.2 RTP Recording Service

We used the existing RTP recording service MBone VCR on Demand (MVoD) [36,37] as the starting point for our development of a generic RTP/I recording service. An RTP recording service such as the MVoD usually handles two network sessions (see Figure 46). In the first the recorder participates as if it were a regular participant. Depending on its mode of operation (recording or playback), it acts as a receiver or a sender towards the other participants of the session.

A second network session can be used to control the recorder from a remote client, e.g., using the Real-Time Streaming Protocol (RTSP) protocol [84]. During the recording of an RTP session, the recorder receives RTP data packets and writes them to a storage device. Packets from different media streams are stored separately. When playing back, the recorder successively loads the RTP packets of each media stream and computes the time at which each packet must be sent using the time stamps of the RTP headers. The recorder sends the packets according to the computed schedule. A detailed description of the synchronization mechanism implemented in the MVoD can be found in [37].

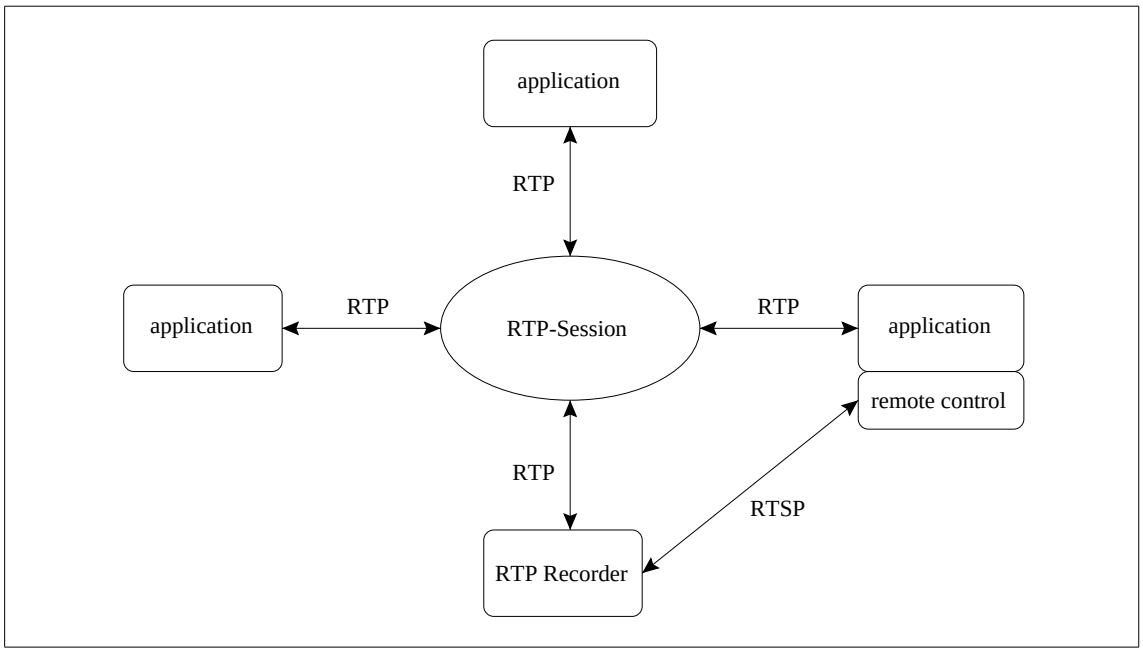

Figure 46: RTP recording scenario

6.4.3 Random Access

In contrast to the traditional media types (in particular audio and video streams) for which random access to any position within a stream is easily possible, interactive media streams do not allow random access without restoring the context of the stream at the desired access position. For example, when the recording of a shared whiteboard is accessed at a time where annotations are being made on the visible page, the state of that page will need to be restored before the actual replay is started. The restoration of the

shared state for random access is the main difference between a recording service for RTP and a recording service for RTP/I. All other aspects, such as the actual recording or timing for replay, need only minor modifications.

The state of an interactive medium at the desired access position needs to be recovered from the recorded media stream. Note that the generic recorder is not able to interpret the media-specific part of the RTP/I packets and thus cannot directly compute the media state and send it to the receivers. What the recorder can do is to compose a sequence of recorded RTP/I packets that put the receivers in the correct state. The task a recorder has to accomplish before starting a playback is therefore to determine the appropriate initialization sequence of recorded RTP/I packets.

The current state for a distributed interactive medium is determined by an initial state and a sequence of events applied to that state. In a discrete interactive medium the event sequence is not bound to specific points in time. Thus, the playout of an initial state and an event sequence will always result in the same media state, independent of the speed at which the sequence is applied. In contrast, the event sequence for a continuous distributed interactive medium is bound to specific points in time. A sequence of events that is applied to the state of a continuous distributed interactive medium will leave the system in the correct state only if each event is applied at a well-defined instant in time.

This difference between discrete and continuous distributed interactive media should be considered when computing the sequence of event and state packets to recover the media state. In the case of a discrete medium, such a sequence can be computed to recover the media state at any point in a recorded stream. In contrast, the media state of a continuous medium can only be recovered at points within a recording where a state is available; events cannot be used for state recovery because they must be played in real-time. Therefore, random access to a continuous distributed interactive media stream will usually result in a position near the desired access point. We call this *near random access*. The more often the state is stored within a stream, the finer is the granularity at which the stream of a continuous distributed interactive medium can be accessed. It should be noted that any algorithm that determines correct initialization sequences for continuous media will also provide a solution for discrete media. However, for discrete media there are more efficient algorithms that allow faster and exact random access by exploiting the fact that events do not have to be replayed in real-time.

Interactive media applications usually send the media state only upon request by another application. Thus, the recorder must request the state at periodic intervals using the regular RTP/I state query packet. These requests use the lowest available priority because a delayed or missing response reduces only the access granularity of the stream, which can be tolerated to some degree.

6.4.4 Mechanisms for Playback after Random Access

The algorithms presented in this section describe how an initialization sequence of recorded RTP/I packets can be determined that recovers the state of the medium at, or near to, the desired access position. The algorithms can be completely implemented in the recording service. An application does not need to be modified to be able to receive such an initialization sequence. From an application's viewpoint the initialization sequence looks like a regular RTP/I media stream.

6.4.4.1 Basic Mechanism for Media with a Single Sub-Component

This simple mechanism is able to recover the initial media state from RTP/I streams that have only a single sub-component. When starting the playback of such a stream, the best case would be if the state were contained in the recorded stream at exactly the access position. Then playback can begin immediately without further consideration. In general, however, the playback will be requested at a position where no full state is available in the recorded RTP/I stream.

Let us consider, for example, a recorded RTP/I stream that consists of the sequence S0 depicted in Figure 47. S0 contains a state, three successive delta (d) states and several events. If a user wants to perform a random access to position tp from the recording, then the state at tp must be reconstructed by the recorder.

A continuous interactive medium does not allow direct access to tp because there is no state available at tp in the recorded stream. However, access to position td3 within the stream is feasible because td3 is the location of a delta state. The complete media state at td3 can be reconstructed from the state located at position ts and the subsequent delta state at position td3 which is the position of the last delta state before tp. The events between ts and td3 can be ignored because all modifications to the state at ts are included in the delta state. The packets that contain states can be sent at the maximum speed at which the recorder is able to send packets. When the recorder finally reaches td3 (and has sent d3), fast playback must be stopped, and playback at regular speed must be started. The start of the regular playback may not be delayed or sped up because events must be sent in real-time relative to the last (delta) state. This is important since in continuous distributed interactive media, events are only valid for a specific state that may change with the passage of time. Altogether, the recorder will play back sequence S1 shown in Figure 47.

For discrete distributed interactive media, fast playback of events is possible. Therefore, random access to position tp can be achieved by also sending the events between td3 and tp at full speed. The resulting sequence S2 is also shown in Figure 47.

6.4.4.2 Mechanism for Media with Multiple Sub-Components

In a more sophisticated mechanism the existence of sub-components can be exploited to reduce the amount of data required for the recovery of the media state. Using sub-components, the state of an interactive medium can be recovered selectively by considering only those sub-components that are actually required to display the medium to the user.

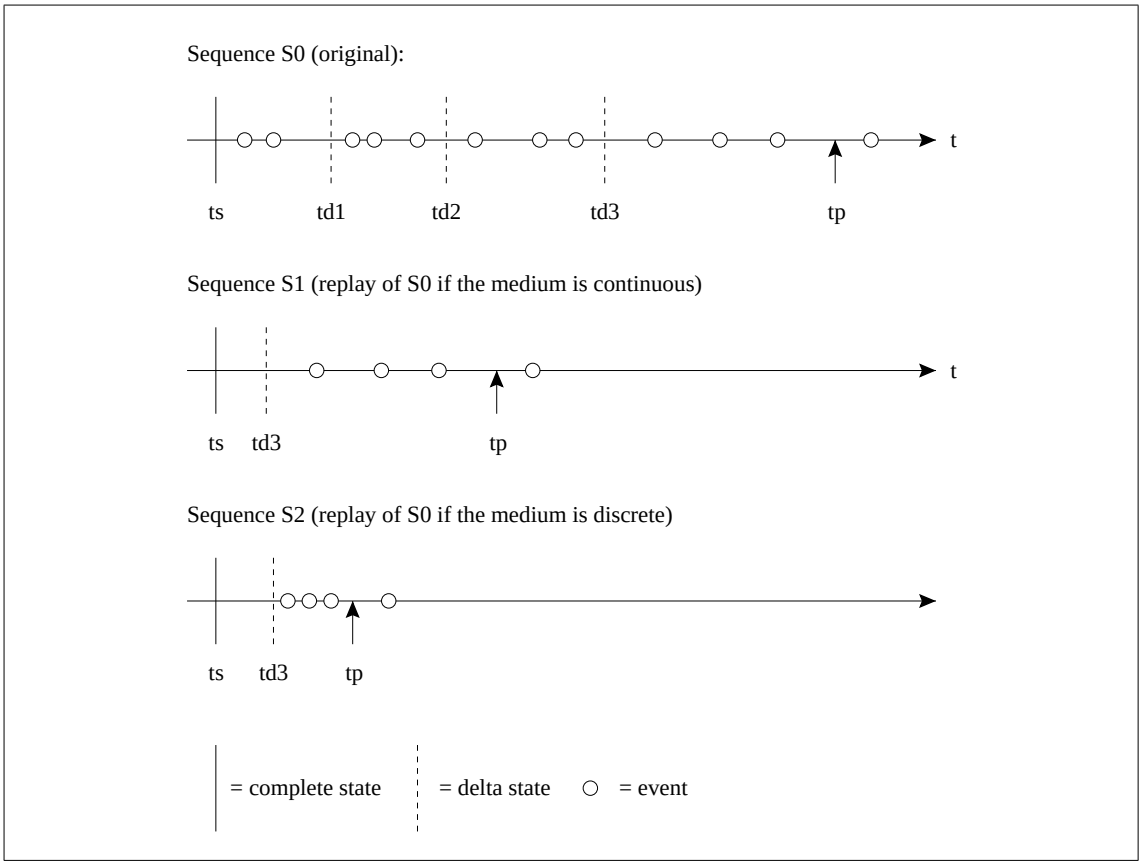

Figure 47: Playback of a recorded RTP/I stream with a single sub-component

For example, a user might want to access the recording of a shared whiteboard at a certain point in time. A recording service could simply restore the state of all sub-components (i.e., the pages) of the shared whiteboard. Generally this is inefficient, especially since previous pages might never be displayed again during the remainder of the recorded session. It is therefore a good idea to recover only the state for those sub-components that are actually visible to the user. The information needed to decide whether a sub-component is required to present the medium to the user can be retrieved from the periodic sub-component reports. Those sub-components that have the active flag set are active while the others are not currently needed to display the medium. With the information of the sub-component reports the set of active sub-components (SAS) at the access position can be determined. The SAS can be kept in an index where each point of the

recording maps to an SAS. The recording service can create the index during the recording of a session.

For a continuous distributed interactive medium the following algorithm can be used to initialize the receivers: with the SAS known, the recorder scans the recording backwards from the access point towards the beginning of the recording. Every time it encounters a state for any element in the SAS it marks the sub-component. At the point of the recording where the last element of the SAS is marked, the replay starts with the transmission of the state for the sub-component that has just been marked. During replay a state is only played if its sub-component is in the SAS and if the state is the last recorded state of the sub-component before the access point. Events and delta states are only played if they target a sub-component for which a state has already been played. All other packets are filtered out and are not transmitted by the recording service.

An example of how this algorithm works is shown in Figure 48. It depicts the recorded RTP/I stream of a session with two senders. Sender 1 operated on sub-components 1 and 3, whereas sender 2 worked with sub-components 2 and 4. If the recorded stream is accessed at position tp, the recorder retrieves the SAS at tp. In this example the SAS contains the elements s1, s2 and s3.

The recorder now scans the recording backwards towards the beginning of the recording and marks the elements in the SAS. At the time ts1 it marks the last element (s1). Thus the recorder has to start playback at position ts1 and recovers the state s1. The recorder must continue with the playback because events referring to s1 may be located between ts1 and tp. Notice that we are considering a continuous interactive medium where all events must be played in real time.

During the playback of the stream between ts1 and tp two problems may occur: First, events may be located in the stream that refer to states that have not yet been sent. The sending of these events must be suppressed because a receiver cannot interpret them correctly. In our example, events concerning s3 and s2 are filtered out. Secondly, there may be sub-component states in the stream that are not in the set of active sub-components at tp (s4 in our example) and thus are not needed for playback at tp. Therefore the state of s4 (and all events referring to s4) must also be filtered out.

Summing up the example, the recorder will start playback at position ts1, sending the state of sub-component s1 and events referring to s1. All other states and events will be filtered out. The next required state is s2, which will be sent as it shows up for the last time in the recording before tp is reached. After that, all events referring to s2 will also pass the filter. The same happens with s3. Finally, once the recorder has reached position

tp, the sub-components s1, s2 and s3 will have been recovered, and regular playback without any filtering may start.

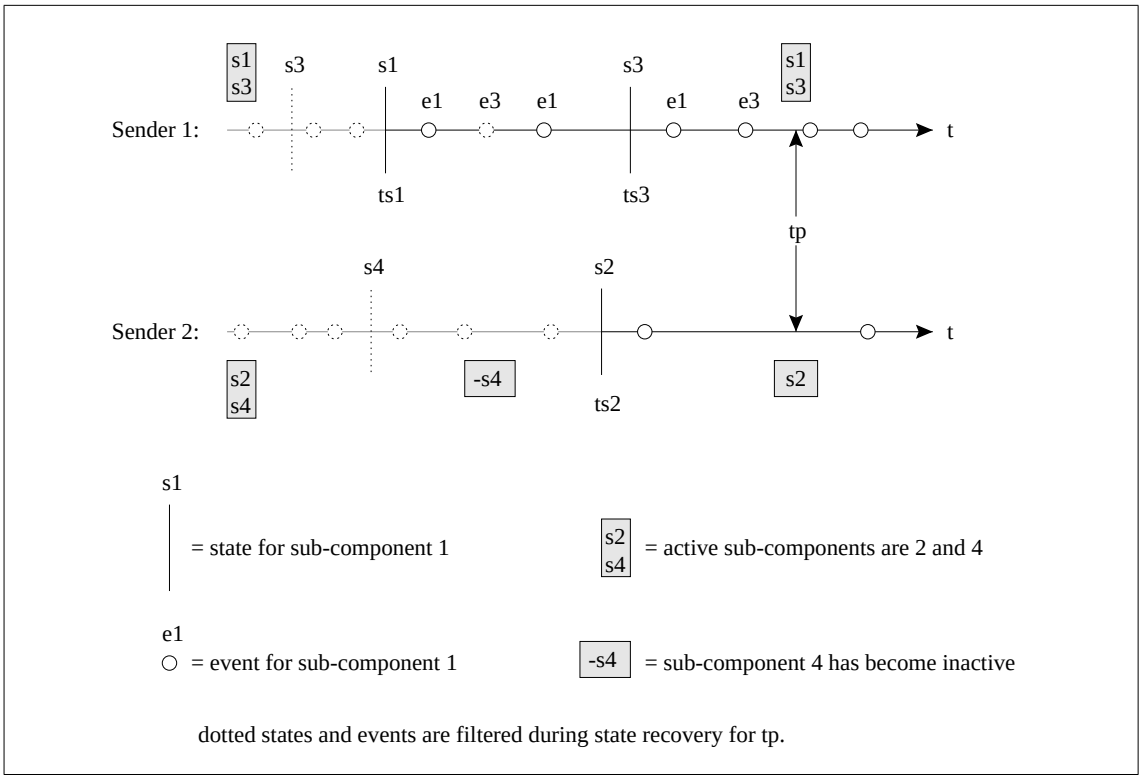

Figure 48: Playback of a recorded RTP/I stream with multiple sub-components

The algorithm for initializing the replay of a continuous distributed interactive medium with several sub-components requires that the state of the individual sub-components be regularly queried and inserted into the recording. The higher the frequency of the inserted states, the faster and more accurate the state initialization can be performed. Generally the regular state requests should be issued with the lowest possible priority.

However, there are two occasions when the state of a sub-component should be immediately requested with a high priority. The first occurs when a sub-component that was passive becomes active. In this case the state of the sub-component is immediately required to be stored in the recording. Applications that did a random access to a time before the sub-component was activated will only be able to present the state of the sub-component to the user when the state of that sub-component is stored in the recording immediately after it becomes active.

The second situation occurs when an event arrives for a sub-component that has not been recognized as active by the recorder. The reason for this could be that a sub-component report got lost, or that an application activated a sub-component and transmitted an event before a sub-component report was sent. As soon as such an event arrives, the sub-component should be marked as active by the recording service and its state should be que-

ried. This needs to be done since the random access mechanism will not restore the state of the sub-component if the access happens right before the time the event is transmitted by the recording service. To repair this problem the recording should contain the state of the affected sub-component within a minimal amount of time from the event that caused the problem.

For a discrete distributed interactive medium with multiple sub-components the same algorithm as for continuous media can theoretically be used. However, a simpler and faster algorithm is available for discrete media. For a discrete medium it is sufficient to fetch the last recorded full state of each sub-component in the SAS, the last recorded delta states, and all events between the last delta states and the access position. The recorder can replay this information at maximum speed to initialize the receivers.

6.4.4.3 Mechanism for Multiple RTP/I and RTP streams

Typically a session involving a distributed interactive medium will also include other distributed media. For example, a videoconference may consist of audio, video, a shared whiteboard, and a 3D workspace. A generic recording service must be able to record and replay those streams in a synchronized fashion.

The solution to this problem is straightforward. The synchronization between the individual streams can be derived from the timestamps included in the RTP and RTP/I packets. For RTP the timestamps need to be adjusted by the offset to real-time which is carried in the RTCP sender report packets. The RTP/I timestamps refer to real-time and therefore do not need an adjustment.

In the case of a random access to the recording the first task is to determine the RTP/I stream for which the recording needs to be rewound farthest for the initialization. This stream will start playing according to the algorithm described above. Other RTP/I streams will join the replay according to the results of their initialization algorithms. Finally the RTP-based media will join at the exact access position. In this way the synchronized random access to, and playback of, sessions involving multiple RTP and/or RTP/I media is possible.

6.4.5 Consistency

A generic recording service should work independently of the consistency mechanisms used by the application. During recording it therefore can not rely on the presence of any specific consistency service. Also the recording service is unable to extract the state of sub-components during replay, e.g., to repair short-term inconsistencies. These two problems can be solved in a manner that works for arbitrary distributed interactive media.

6.4.5.1 Consistency During Recording

During the recording of a session the recording service should choose to receive all events, states, and delta states in a reliable fashion. It does not matter if some events arrive late since the recording service can use the time warp approach to establish a consistent recording. Using time warp is trivial for the recording service because it does not interpret the content of the packets. For a time warp the recording service merely needs to reorder the received packets according to their timestamp.

6.4.5.2 Consistency During Replay

During replay the receivers of a recorded stream should choose to receive all events, states, and delta states in a reliable fashion. In order to allow for the time needed to retransmit lost packets, the recorder uses a large amount of local lag for the replay. This does not pose a problem since no human user will actually experience the local lag - all events stem from the recording. The problems that could arise if the value for local lag is unexpectedly exceeded are repaired by the states that appear periodically in the recording.

6.4.6 Signaling the Replay of a Recorded Session

Note that in replay mode the recording service has exclusive control over the session. When the generic recording service replays a recorded session it should signal this by means of periodic messages to the generic services channel. The applications can use this knowledge to prevent the replay from being disturbed, e.g., by keeping the user from initiating events. Also other generic services, such as the consistency service, can use this information to prevent floor handover, floor claims, state queries, and other actions that are not permissible during the replay of a session.

6.5 Chapter Summary

In this chapter we investigated three generic services that are based on RTP/I:

- A generic, local-lag-based consistency service. This service used a robust floor control to determine the application that should help with repairing short-term inconsistencies by transmitting the state of a sub-component. The robust floor control is able to recover lost floors and handle duplicate floors by means of a floor claim algorithm. Based on this floor control the consistency service guarantees eventual consistency even when network partitioning occurs or floorholders crash.
- A generic late-join service that allows latecomers to join an ongoing session. The state information that is needed by the latecomer is transferred over a specific late-

join session in order to minimize application and network load. The late-join service does not rely on centralized components and is therefore robust to the failure of individual participants. Different applications can choose from a set of late-join policies for the recovery of individual sub-components' state. The late-join service can therefore be adapted to the diverse needs of media in a simple and efficient way.

- A generic recording service. This is a middleware service that resides within the network. It allows the synchronized recording of arbitrary RTP and RTP/I streams. We used the RTP-based MBone VCR on Demand recording service as a starting point and augmented it with support for RTP/I. Especially challenging was the realization of near random access for RTP/I streams, which required the restoration of the state information at the access position.

All three services use the generic services channel to exchange information, such as the request for a floor handover or the request to let a late-join server join the late-join session. Using a single channel prevents transport address clashes that might cause a problem if each service used its own transport session.

In the next chapter we will report on the realization of three TeCo3D prototypes. One of those prototypes uses all three services that have been described in this chapter.

7 Realization and Experiences

In the context of this thesis we have developed three distinct prototypes for the 3D telecollaboration tool TeCo3D. Each has provided us with important insight into the specific problems of and potential solutions for distributed interactive media.

The first two prototypes were developed in parallel. One of them is a ‘weblication’, i.e., an application that resides within a web browser. The other was realized as a service that can be integrated into other shared workspaces. As an example, we have integrated this 3D telecollaboration service into the digital lecture board dlb.

The main aim of the two early prototypes was to show that the interaction with collaboration-unaware VRML models can be shared as described in Chapter Four. In addition they enabled us to discover the key challenges of distributed interactive media, such as consistency and the need for a common application level protocol.

These experiences led us to develop a third prototype, which is based on the ideas and concepts that have been discussed throughout this thesis. This prototype is based on RTP/I, and it uses all the generic services described in the previous chapter. It shows that RTP/I and RTP/I-based generic services indeed provide a valid foundation for the development of distributed interactive media.

In this chapter we describe the realization of the TeCo3D prototypes. Furthermore, the experiences gathered throughout the development of the TeCo3D prototypes are discussed.

7.1 The First TeCo3D Prototype

With the first TeCo3D prototype we aimed to investigate whether it is possible to share collaboration-unaware VRML content [58]. Of particular interest was how to capture and distribute user actions with the VRML content. To this end we developed the idea of replacing regular VRML sensors with customized, collaboration aware sensors, as has been described in Chapter Four.

The general idea of this prototype is to distribute the initial state of a VRML object in the form of regular VRML files. A multi-point file transfer is used for this purpose. Once the initial description of the object has been distributed, the state of the object is kept consistent only by sharing the user interactions. Since a regular VRML browser is used for this TeCo3D prototype, it is not possible to extract the state of the object from the browser. Moreover, we use a simple reliable multicast protocol as means of distribution for the events.

Consistency support is therefore only best-effort: events are injected into the VRML object as soon as they are received. In addition, only one user at a time may interact with any given object, even though no floor control enforces this restriction. This limits the VRML content that can be shared with the prototype to objects that reach a stable state frequently. An example of such an object is a 3D model in which each user interaction triggers a short animation. This can be used, e.g., in a call-center environment to explain the handling of 3D objects, such as the assembly of a piece of furniture.

7.1.1 Architecture

The main part of the first TeCo3D prototype is realized as a Java applet that runs within a web browser. A standard VRML browser, such as CosmoPlayer [8], Contact [5] or WorldView [38] is expected to be present as a plug-in to the web browser. The applet uses the VRML browser plug-in as a 3D presentation engine.

There are three important components of TeCo3D present within the scope of the web browser’s process: The VRML browser, the TeCo3D applet, and the VRML content (see Figure 49). The applet and an initial VRML model are linked to a local web page that is accessed by the user to start TeCo3D. Upon initialization, the applet loads an initial model into the VRML browser by using EAI calls. This initial model contains mechanisms for getting and setting the viewpoint of the local user, and it provides a framework into which other VRML models can be inserted. The applet also displays the user interface that lets the user control the services offered by TeCo3D: connecting to a 3D teleco-operation session, importing a VRML model, and giving a guided tour through the model by coordinating the viewpoints of all participants.

As described in Chapter Four, the collaboration-unaware VRML content is processed by replacing regular sensors with collaborative sensors. The collaborative sensors of an imported VRML object register with the applet. Once registered, they send all events they generate not only to the local VRML model, but also to the TeCo3D applet, which distributes them as applicable. On the other hand, the applet is able to inject events from remote participants into a collaborative sensor, causing an event cascade within the local VRML model just as if the local user had triggered the sensor.

In order to avoid the limitations of Java applets in regard to network access, the communication subsystem was designed as a stand-alone Java application running outside of the web browser's process. The foundation of the communication subsystem is the Java library iBus [88], which provides reliable multicast as a transport service.

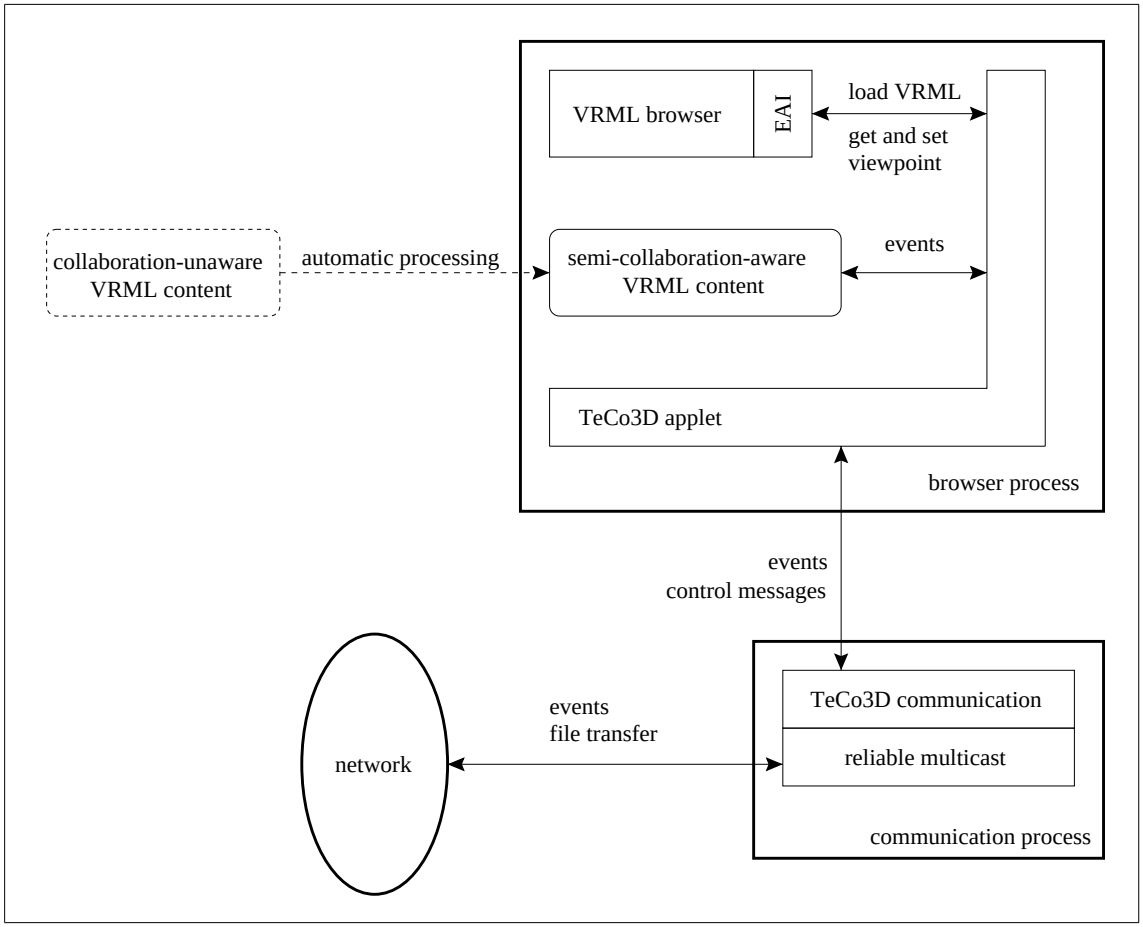

Figure 49: Architecture of the first TeCo3D prototype

The services of the iBus library are used by the TeCo3D communication component, which acts on behalf of the applet, exchanging events and files with peer communication components. The applet and communication component communicate via local socket connections. They exchange two types of messages: *events* that are generated by the user, and *control messages*, which control the behavior of the communication component. While the events are routed through the communication component transparently, the control messages are interpreted by the communication component and trigger actions such as connecting to the network or initializing a file transfer.

7.1.2 Functionality

The first TeCo3D prototype realizes four pieces of functionality that turn a standard VRML browser into a 3D telecooperation application:

- connection management,

- distribution of VRML content,
- distribution of events, and
- viewpoint synchronization.

7.1.2.1 Connection Management

In order to provide its services, TeCo3D needs to connect to a 3D telecooperation session. A session is identified by an IP multicast address paired with a port number. The user provides this information along with the desired time to live (TTL) and the port at which the local TeCo3D communication process is listening. As soon as this information is available, the applet connects to the communication process and requests a connection to the 3D telecooperation session by generating a connect control message. The communication component interprets this message and asks the iBus library to establish the connection to the selected multicast address and port. After establishing this connection, a list of current participants is retrieved, which is handed through the communication component to the applet, where it is displayed on the user interface. Whenever the participants of a session change, this process of setting the list of current participants is repeated. A snapshot of the user interface after the connection has been established is shown in Figure 50.

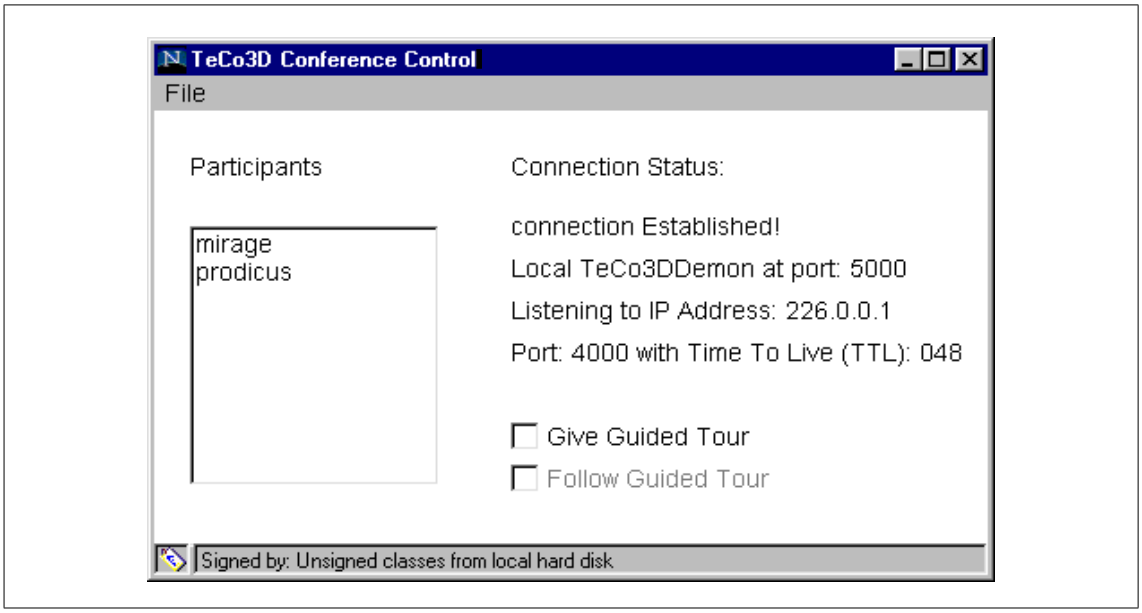

Figure 50: TeCo3D conference control

7.1.2.2 Distribution of VRML Content

Once the connection to a 3D telecooperation session has been established, the user can import VRML models into the common workspace of all participants. This is done by selecting the load item from the file menu. After the user has chosen a VRML file, the

applet asks the communication component to distribute the relevant files via reliable multicast to all participating users. The communication component is responsible for the actual file transfer. It notifies the applet when all files have been transmitted. Upon reception of this notification, the applet of each participant inserts the VRML content into the initial VRML model with an EAI call, making it visible to the user.

The screen-shot in Figure 51 shows what the prototype looks like after a model has been imported. In this example a model is shown that demonstrates the assembly of a piece of furniture.

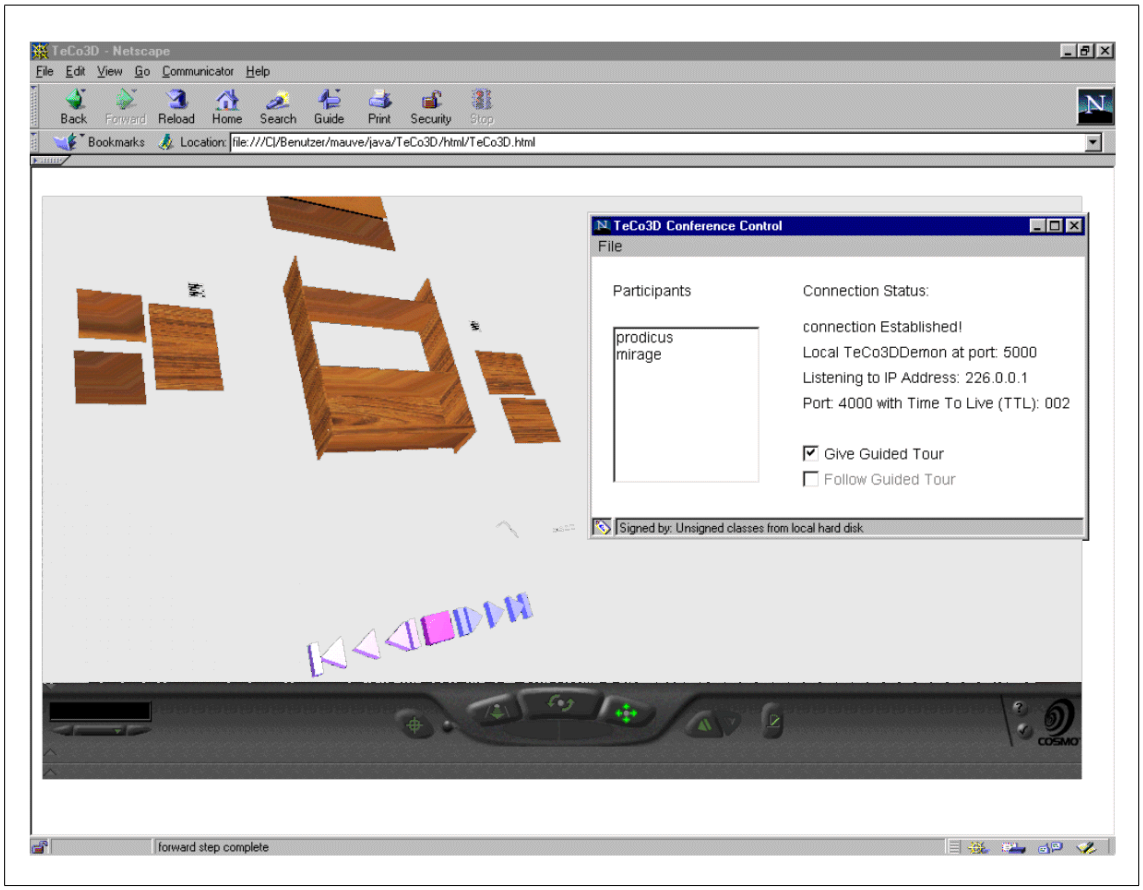

Figure 51: Screen-Shot of the first TeCo3D prototype

7.1.2.3 Input Sharing

User interactions with the imported 3D models are shared in the manner described in Chapter Four. The regular VRML sensors are replaced with specialized collaborative sensors that allow the applet to retrieve events from and inject events into the shared VRML object.

7.1.2.4 Viewpoint Synchronization

Viewpoint synchronization is a mechanism by which one participant (the guide) in a 3D telecooperation session can give a guided tour through shared 3D models. While viewpoint synchronization is in effect, the viewpoints of the guided users follow the guide, allowing all participants to focus on the same part of a model.

TeCo3D realizes the mechanism for viewpoint synchronization by including two VRML nodes into the initial VRML model. One of those nodes is a `ProximitySensor,` which reads the position and orientation of the local user. The other node is a `Viewpoint,` which can be used to set the position and orientation of the local user. The functionality of these two nodes is accessed by the TeCo3D applet with EAI calls. The applet is notified when the viewpoint of the guide changes and, in response, sends a "viewpoint changed" event to the communication component. This event contains the orientation and position of the guide and is distributed by the communication component to all peer TeCo3D instances. As soon as the applet of a guided participant receives the information, it sets the new values of the `Viewpoint` node, changing the user's point of view and orientation.

In order to use the viewpoint synchronization mechanism, the prototype implements a simple policy. Users can decide at any time whether or not they want to participate in guided tours. They can do this by using the checkbox "Follow Guided Tour" on the TeCo3D user interface (Figure 50). If they decide not to follow guided tours, their viewpoint is not changed by viewpoint synchronization. A user can choose to become the guide by checking "Give Guided Tour" during a session. The viewpoint synchronization mechanism is then initialized with all the users who have chosen "Follow Guided Tour" to be true. An already existing guide is reduced to the status of a regular user whenever someone else becomes the guide.

7.1.3 Experiences and Evaluation

The implementation of the first TeCo3D prototype consists of around 8000 lines of code, including the software for both the browser and the communication process. We successfully used the prototype with a number of 3D models that have been taken from the web. Initially these models did not contain any methods or mechanisms for distributed usage. By making it possible to use these models in a distributed session, the first prototype demonstrated the feasibility of our approach to share user actions for collaboration-unaware VRML content.

Even though the main functionality of the prototype was realized as a Java applet, the performance was good. The reason for this is that the time-critical part, namely the rendering of 3D geometry, is performed by the VRML browser. The control and network

functionality is less time-sensitive and can be handled in Java without affecting the overall efficiency of the prototype. The only part were we experienced significant problems with the performance of Java was the communication between the browser- and the communication process. Since each process uses its own Java Virtual Machine (the browser process the JVM of the web browser and the communication process the JVM of the Sun JDK), and since we had no way to control the priority of the processes it was hard to prevent that one of the processes gets all computational resources and thereby starves the other one. This problem becomes apparent when there is a particular high load on the system, e.g., during the distribution of the initial VRML description. We therefore strongly recommend not to use more than one JVM for a given system. Since we could not avoid using two JVMs in our first prototype we solved the problem by implement a rate control for the transmission of the initial VRML content, so that the communication process is not overly burdened with receiving many packets at the same time.

We also encountered a number of other interesting and challenging problems that are not performance related. The most obvious one is the lack of consistency support. While most VRML models can be used in a distributed fashion with the best-effort consistency offered by this prototype, the users must be aware of this shortcoming and avoid interactions that may destroy the consistency. For example, a model that demonstrates the assembly of a piece of furniture can be controlled in different ways. It is possible to view each step of the assembly as a short animation. After each step has been completed a user may initiate the next step by pressing a VCR-like control button (see Figure 51). This works very well with the prototype. But the model also offers the possibility to view the whole assembly process in one long animation that can be stopped for a pause at any time. Pausing the animation generally causes a problem with this TeCo3D prototype since the pause event is executed at different times for the individual participants. Therefore the state of the shared model after the pause has been executed can be inconsistent.

From this problem we derived that a real-time protocol is required, as well as a mechanism to repair or prevent inconsistencies. This led us to the consistency considerations described in Chapter Three. In particular, the ability to extract the state of the model was identified as vitally important to guaranteeing eventual consistency for continuous distributed interactive media.

Moreover, we wanted to offer the ability to join an ongoing TeCo3D session and a way to record and replay sessions. Since similar services were also desirable for other applications that have been developed by our research group, we decided to take a general approach. As a result we developed the media model for distributed interactive media (Chapter Two) and RTP/I - the Real-Time Application Level Protocol for Distributed Interactive Media (Chapter Five). Upon this foundation we were then able to place generic support for consistency, late-join, and recording as described in Chapter Six.

7.2 Distributed Virtual Reality Component

The second prototype is called distributed virtual reality (dvr) component and was developed in parallel to the first prototype [82,26]. Its main objective is to provide the functionality of the first TeCo3D prototype as a C++ based component that can be integrated into other shared workspaces, such as the digital lecture board dlb. For this reason it is not possible to use a standard VRML browser that needs to run within a web browser. Instead we have chosen to use the commercially available C++ VRML browser development library OpenWorlds [13] as the 3D presentation engine for this prototype.

7.2.1 General dvr Architecture

The distributed virtual reality component (dvr) provides its functionality by using the replicated client-server architecture depicted in Figure 52. The *dvr server* is responsible for transferring the files describing a VRML object and for the distribution of events. It also manages the 3D presentation. Convenient access to the services offered by the dvr server is provided by a dvr client-stub. The client-stub communicates with the server via a local socket connection. It is available as a library. The application programming interface (API) provided by the client-stub includes commands for connection management and distribution of VRML content. This content and the information about user interaction are exchanged between distributed instances of the dvr server by means of our reliable multicast protocol SMP [25] on top of IP-Multicast.

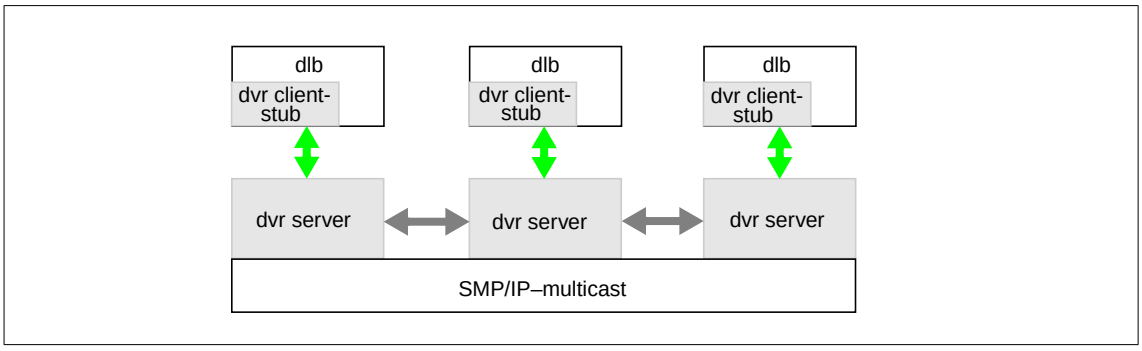

Figure 52: General dvr architecture

7.2.2 dvr Server Architecture

Due to the replicated architecture and the need to integrate a VRML browser library into the dvr server, we have designed the server as indicated in Figure 53. The server consists of the three main parts: *Application Communication Manager, dvr Communication Manager* and *VRML Browser Manager.*

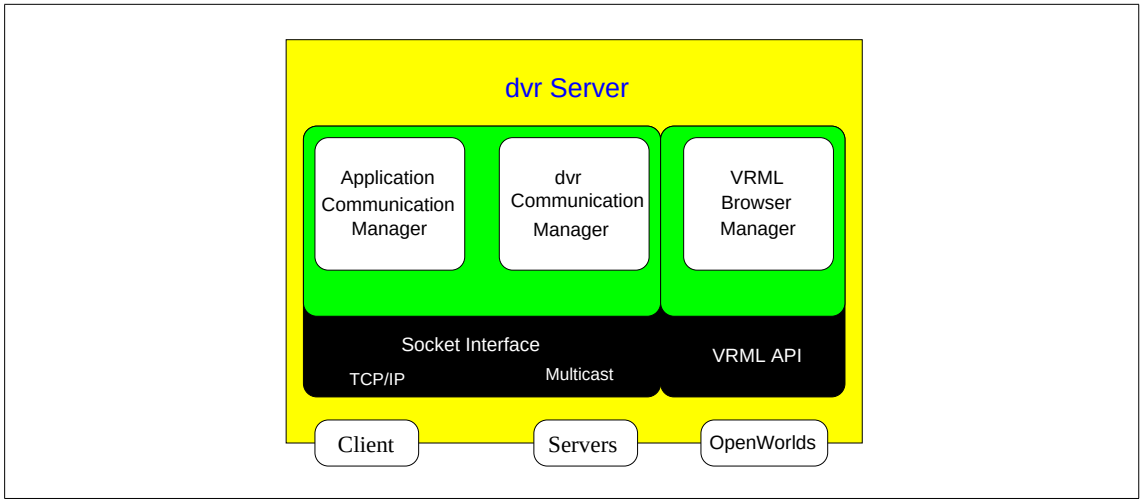

Figure 53: dvr interfaces and components

The Application Communication Manager controls the local communication between the dvr server and the dvr client (e.g., the dlb). This communication includes requests from the local client to distribute and display models. The dvr Communication Manager handles the communication with peer server instances. Main tasks are joining and leaving reliable multicast sessions, transmitting VRML models, and exchanging information about user actions. The VRML Browser Manager provides an interface to a VRML browser. We rely on the functionality of the VRML External Authoring Interface of the OpenWorlds library to instruct the browser to load VRML content and display the content in a separate window.

7.2.3 Dynamic Behavior of the dvr Component

To see how the main dvr parts interact with each other to provide the required functionality, consider the following example, which consists of three steps:

1. A dvr client (e.g., the dlb) requests the distribution and presentation of a VRML model.
2. The local user interacts with the content.
3. A remote user interacts with the content.

The three steps are illustrated in Figure 54. The request of a dvr client to distribute a VRML model (❶) arrives at the local socket of the dvr server. The Application Communication Manager accepts the request and forwards it to the dvr Communication Manager where the files belonging to the VRML model are determined, the original VRML sensors are replaced and the semi-collaboration-aware VRML content is sent via reliable multicast to all peer dvr servers.

As soon as all files have been transmitted, the dvr Communication Manager starts the presentation of the VRML content (❷) by notifying the VRML Browser Manager. The VRML Browser Manager uses the External Authoring Interface of the OpenWorlds VRML browser to request loading of the VRML model. Upon receiving this request, the browser fetches and initializes the VRML content. During initialization all collaborative sensors register with the Sensor Registry, which is part of the VRML Browser Manager.

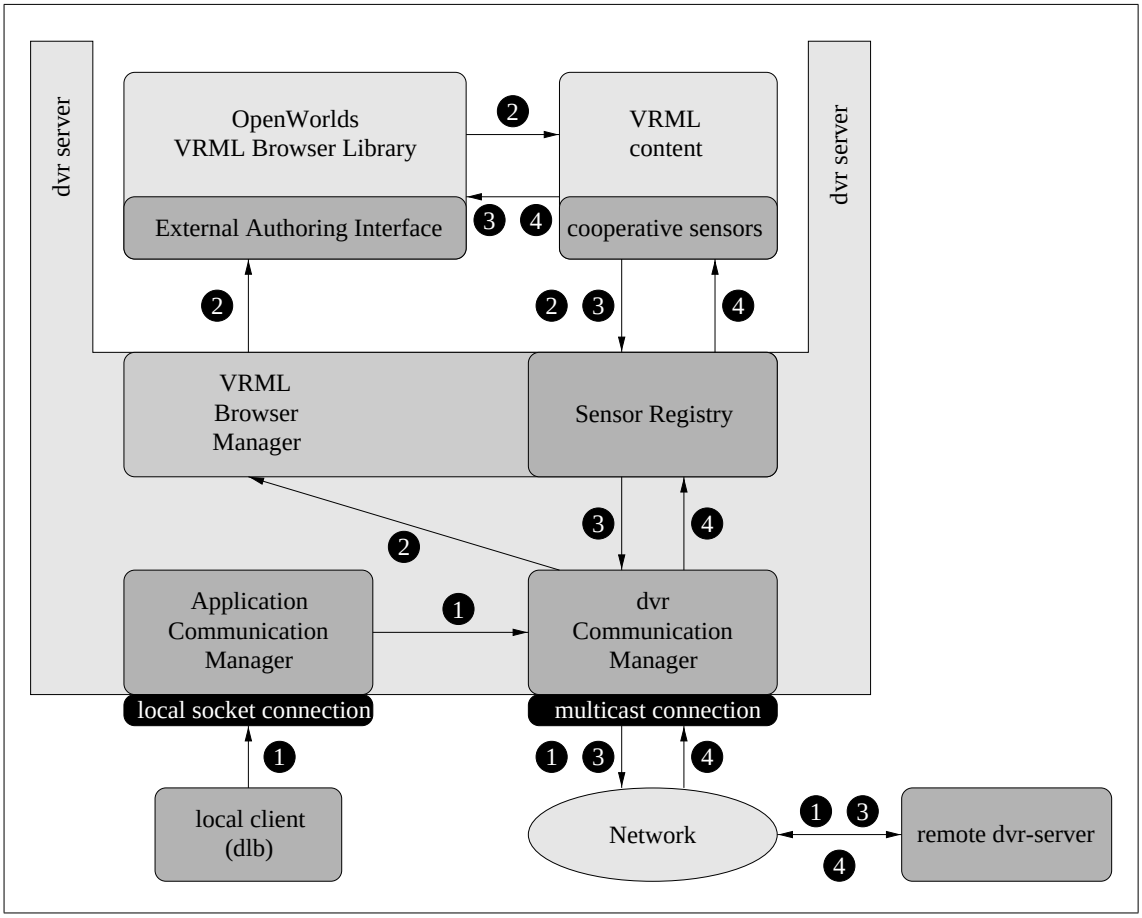

Figure 54: dvr example

As soon as the local user interacts with the VRML model by activating a collaborative sensor (❸), the resulting events are transmitted via the Sensor Registry and the dvr Communication Manager to all dvr server peer instances. Upon their arrival, the events are injected into the VRML model of the recipients. Vice versa, when a remote user interacts with the model (❹), the events received by the dvr Communication Manager are forwarded to the Sensor Registry, where the corresponding collaborative sensor is identified. The event is then injected into the VRML model just as if the local user had activated this sensor.

7.2.4 Controlling the dvr Component

Cooperative applications can control the dvr component by using the dvr client-stub API. The client-stub is a C++ library that can be linked to the client. The application programming interface, which was intended to be kept as simple as possible, offers the following methods to access the dvr service:

- `Connect`. This method allows to establish a connection between client and dvr server. The local port number of the server has to be passed as an argument.
- `Disconnect`. The `Disconnect` method releases the connection to the dvr server.
- `JoinSession`. By using `JoinSession`, the client requests the server to join a multicast session at a specific address supplied by the client.
- `LeaveSession`. The server is instructed to leave a multicast session.
- `LoadVRML`. This method requests the transmission of a VRML model to all participants (dvr servers) and initiates rendering of the content at all sites. The filename of the root VRML file for the model needs to be passed as an argument.

7.2.5 Integration into the dlb

VRML models that use the distributed virtual reality component are autonomous items of the shared workspace of the dlb. Since direct rendering of VRML on the shared workspace of the dlb was not feasible, the VRML item is represented by an icon, as indicated in Figure 55. The creation of a VRML item realizes the interface to the dvr service. It stores the parameters required by the dvr client-stub such as multicast IP address, port number, or SMP parameters. After having selected a VRML file, the parameters are requested through a dialog. Then a VRML item is created and transmitted to the distributed dlb instances, which locally create a corresponding item. The VRML model can be started by any participant via mouse click on the VRML icon. Upon receiving this event the dvr service distributes the VRML file(s) to all involved dvr instances and launches a local VRML window for rendering the VRML content (see Figure 55).

7.2.6 Experiences and Evaluation

The implementation of the second prototype consists of around 20000 lines of code. It is interesting to see that this is more than double the amount that was required to realize the Java-based first prototype, even though both have a very similar functionality. Also the amount of time required for the development was quite different: the first prototype took around 4 months to develop, while the second prototype required more than 6 months of work. The main reason for this is that Java provides a very convenient API and supports the rapid development of prototypes. The C++ implementation tends to be more complex but it has the advantage that the API (e.g., for network communication) is more powerful.

The main aim of the dvr component was to demonstrate that the 3D telecooperation functionality could be realized in a way that allows the integration of the component into shared workspaces. We have achieved this aim through the development of a dvr component prototype and the integration into the digital lecture board. Several VRML models have been shared successfully in the context of dlb test sessions with the dvr component. In all important aspects the dvr component shares the same traits as the first, web browser-based, prototype.

However, we had one expectation that was not fulfilled: By using a VRML browser library rather than a regular VRML browser we expected to gain enough flexibility to expand the VRML browser functionality to meet our needs, e.g., by adding methods for extracting and setting the state of arbitrary VRML content. Unfortunately this did not prove to be the case. OpenWorlds turned out to be a source of numerous problems, particularly since the version that was available to us contained numerous bugs and was incomplete in the implementation of the VRML specification. Furthermore, OpenWorlds is a regular product, therefore we would have had to pay an extra fee in order to access the source code of those parts that we wanted to modify and extend. The combination of these problems led us to discontinue the usage of OpenWorlds. Instead we decided to use a new 3D presentation engine for our third - and final - prototype of TeCo3D.

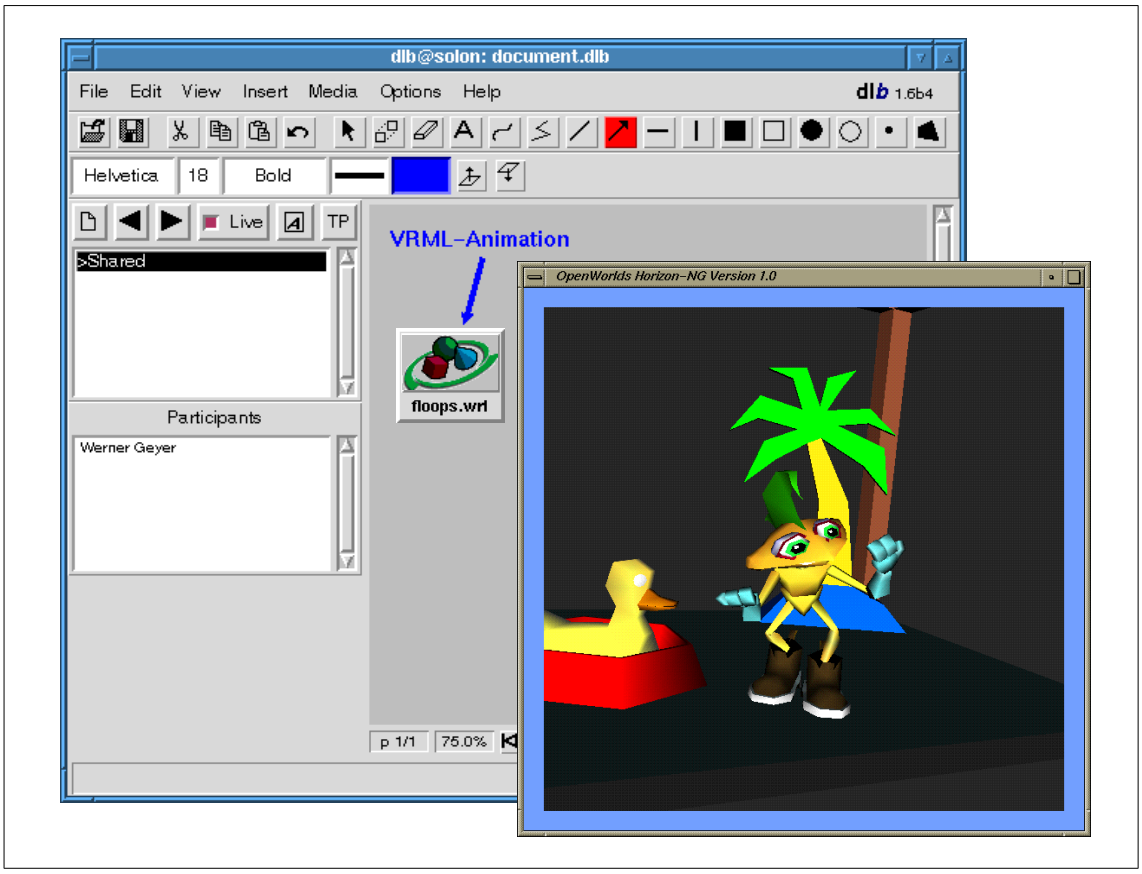

Figure 55: dvr and dlb screen-shot

7.3 RTP/I-Based TeCo3D Prototype

The first two TeCo3D prototypes demonstrated that our idea of sharing collaboration-unaware VRML content is valid. However, they did not address two problems that have to be solved for a fully functional 3D telecollaboration tool: ensuring full consistency at all sites and allowing late comers to join an ongoing session. It was the main aim of the third prototype to demonstrate how these problems can be solved.

While working on the first two prototypes we realized that sharing collaboration-unaware and dynamic VRML models is but one representative of the distributed interactive media class. Much of the functionality that is required for TeCo3D would be reusable for other media that belong to the same media class if there were a common foundation upon which such services could be built. We therefore defined an abstract media model for distributed interactive media and designed the application level protocol RTP/I. Based on this foundation we were able to develop generic and reusable services for *consistency, late join support*, and *recording* of RTP/I sessions.

The design and implementation of the third prototype reflects this concept. It is composed of a TeCo3D-specific part and several generic components. The TeCo3D-specific part handles the rendering of VRML models, the execution of events, and the access to state information. The generic components are the services for consistency and late join, as well as an RTP/I library, and the reliability and network components. All parts have been developed exclusively in Java. To a large extent the previous chapters are based on the experiences and concepts gained through the development of this RTP/I-based, TeCo3D prototype.

The architecture of the third prototype, as depicted in Figure 20, has already been discussed to some extent in Chapter Four. In the following sections we will focus on realization and implementation aspects of the individual components in this architecture. In detail these are network and reliability issues, RTP/I protocol functionality, the application, and the usage of the generic services by the application.

7.3.1 Network and Reliability

We used UPD over IP multicast as well as TCP over IP unicast as the transport protocols. For TCP we developed a server that enables multi-point communication by copying incoming data from one participant to all other participants of a session. The role of this server may be assumed by a session participant or it may be a dedicated process that simply provides a TCP multi-point service. Using either, UDP over IP multicast or multi-point TCP, more than two users can participate in a single TeCo3D session independent of the actual transport service.

Since the transport services vary greatly in their functionality and interface (TCP streams vs. UDP packets), we placed a reliability layer on top of the transport services. This reliability layer is a predecessor of the reliability interface that was described in Chapter Five. It provides a uniform interface for sending and receiving packets. Different reliability instances (e.g., TCP/IP or reliable multicast on top of UDP/IP multicast) may implement this interface. The instantiation of a reliability layer is done by the application, and it is dependent on the transport type. As soon as the instantiation is performed, the difference between the transport types is encapsulated and is invisible to the parts of the application that send and receive packets.

With this architecture it is very simple to use new combinations of transport and reliability services. For example, the integration of a new reliable multicast protocol does not require any modifications in the application other than the code for the initialization of the reliable multicast protocol.

It is noteworthy that the reliability layer is the only real layer in the TeCo3D architecture. It completely hides the details of the transport service. All other components exist side-by-side in parallel, and the functionality they provide may be controlled either directly by the TeCo3D specific functionality or by other parts of the application.

7.3.2 Application Level Protocol

In order to ease the integration of RTP/I into applications, we have developed an RTP/I library. This library provides methods for constructing and parsing RTP/I ADUs as well as for the automatic handling of RTCP/I. In order to enable the review by of this work by the Internet Community we made the source code to the library publicly available under the Lesser GNU Public License [65].

The interface to this library is depicted in Figure 56. The library provides four methods for encoding RTP/I events, states, delta states, and state queries. These methods are called by the application that provides the required information such as priority, affected sub-component, as well as the event, state, or delta state data that is to be transmitted. The RTP/I library performs the actual encoding of this information into the standard RTP/I packets and transmits these packets by using the reliability layer.

The local application usually manages a set of sub-components. New sub-components can be added, sub-components can become active/inactive, and sub-components can be removed from the distributed system. Since the RTP/I library provides RTCP/I functionality and in particular the reporting of sub-components, the application must notify the library whenever the set of local sub-components changes. Therefore the RTP/I library has four methods that allow to signal that a sub-component has been added, removed, activated, or deactivated. Finally, the application can provide the description of the local

RTP/I source, such as the CNAME. This information is used to construct the RTCP/I SDES packets.

The RTP/I library expects the application to implement a number of methods. The first four are used for the delivery of delta states, events, states, and state queries. Also, the library notifies the application about new participants and those participants who have left the session.

```
Implemented by the RTP/I library:

void transmitDeltaState(RTPIDeltaState ds)
void transmitEvent(RTPIEvent event)
void transmitState(RTPIState state)
void transmitStateQuers(RTPIStateQuery sq)
void addSubcomponent(long subID, String appName)
void removeSubcomponent(long subID)
void activateSubcomponent(long subID)
void deactivateSubcomponent(long subID)
void setLocalSourceDescription(SDES localSDES)


Implemented by the application:

void receiveDeltaState(RTPIDeltaState ds)
void receiveEvent(RTPIEvent event)
void receiveState(RTPIState state)
void receiveStateQuery(RTPIStateQuery sq)
void rtpiParticipantAdded(SDES remoteSDES)
void rtpiParticipantRemoved(SDES remoteSDES)
void subcomponentAdded(long subID, String appName)
void subcomponentRemoved(long subID)
void subcomponentActivated(long subID)
void subcomponentDeactivated(long subID)
```

Figure 56: Interface of the RTP/I library

Furthermore, the RTP/I library informs the application about the sub-components that are present in the session. This includes the discovery of a new sub-component, the disappearance of a sub-component, and the activation/deactivation of a sub-component.

Based on the experiences with the implementation of the RTP/I-based TeCo3D prototype, we adjusted and improved our RTP/I specification. Originally RTP/I started out as

an RTP profile. Numerous discussions and the experiences gained with this implementation have led us to specify the RTP/I definition as it is described in this thesis.

7.3.3 VRML Browser

The experience with the first and the second prototypes showed that the ability to get and set the state of a sub-component is of prime importance for a distributed interactive application. Without such a functionality it is hard to prevent consistency from being permanently destroyed by packet loss or unexpected network delay. Furthermore latecomers are unable to join an ongoing session without receiving the current state from another participant.

We therefore developed the method for transparent access to and encoding of VRML state information described in Chapter Four. In order to implement this functionality we needed a VRML browser that is available as source code. At the time we started our work on the third prototype there was only one candidate available that fulfilled this condition: an early prototype of the Java3D VRML browser [100]. While this VRML browser was still in an early stage of development, it allowed us to integrate the additional functionality with acceptable overhead.

We enhanced the Java3D VRML browser by the ability to get and set the state of arbitrary VRML content. Based on the experiences gained with this implementation we improved and streamlined the persistency format several times. An example of an improvement is that we enabled linear writing of persistency information by eliminating forward references. In our first draft of the persistency format it was required that the size of an encoded node be given before the actual encoding of the node. The intention was to allow a parser of the state information to rapidly skip unknown nodes. However, the size of an encoded node can be determined only after it has been encoded. Therefore the size of an encoded node is a forward reference to information that is not available at the time it should be written to the encoded state. While it would be possible to encode the node in a separate buffer, then calculate its size, and finally copy the buffer to the output stream, this would generate a tremendous overhead. We therefore decided to remove the length field from the specification of the persistency format. Several issues like these where found and fixed during the implementation of the state access methods.

With the additional state access functionality the Java3D browser represents the 3D presentation engine of the third TeCo3D prototype.

7.3.4 TeCo3D Specific Functionality and Generic Services

The TeCo3D specific functionality is the ‘glue’ between the generic RTP/I services and the 3D presentation engine. Its main functionality is to route information between the

components (generic services, VRML browser, RTP/I library) of the TeCo3D prototype. It does so by using the interfaces described for the generic services, the RTP/I library and the VRML browser. Besides the routing of information, the TeCo3D-specific functionality also manages the user interface of TeCo3D, which is depicted in Figure 57.

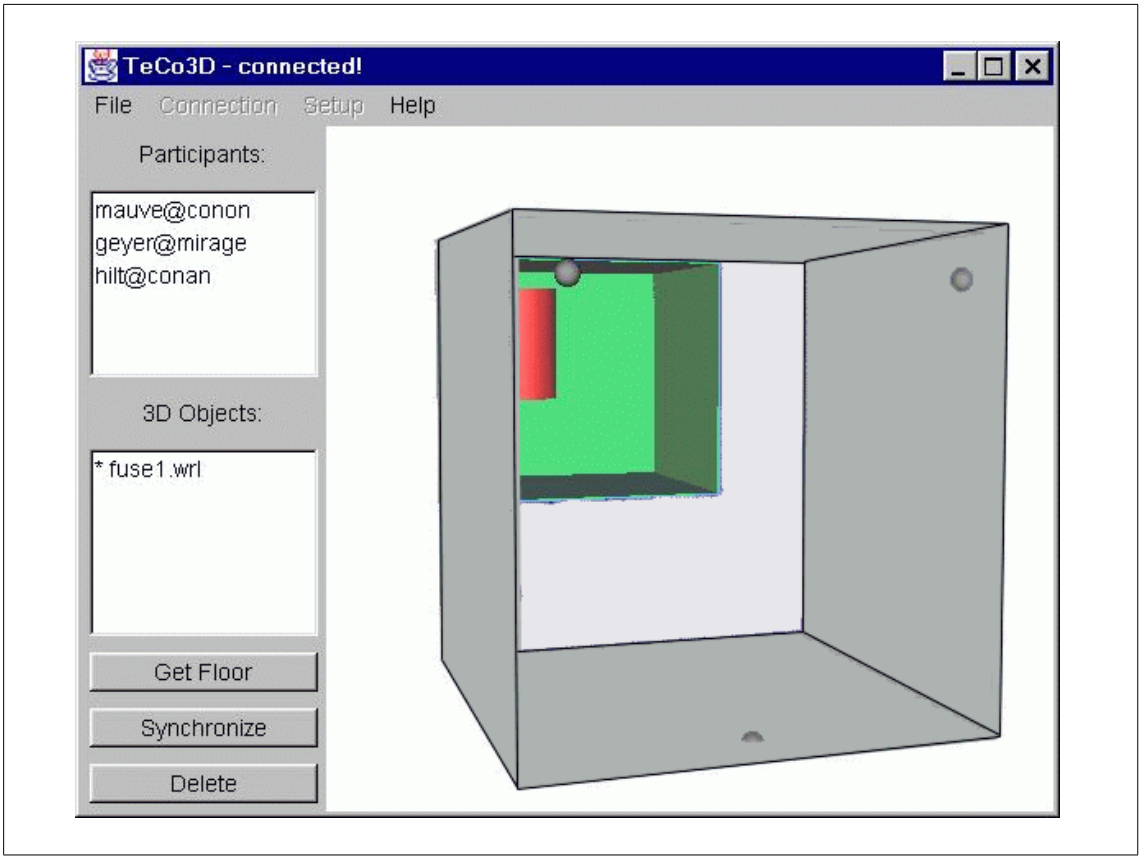

Figure 57: Screen-Shot of the RTP/I based TeCo3D Prototype

The RTP/I-based TeCo3D prototype employs the generic consistency service and the generic late-join service, as described in the previous Chapter. Both of these services have been developed as Java libraries that can be reused for other distributed interactive media.

For the late-join service we used the following options in order to tune it to the specific requirements of TeCo3D:

- When a late-join server is needed for a certain sub-component in the late join group, the late-join group is joined by the floorholder of the sub-component (`setJoinGroupPolicy`).
- The late-join policy can be determined by the user in a setup menu before he or she joins a session. TeCo3D uses the chosen policy as the default policy for all sub-components (`setDefaultPolicy`).
- The base leave time for the late-join group is 300 seconds (`setLeaveGroupBaseTime`).

7.3.5 Experiences and Evaluation

The implementation of the third prototype was by far the most complex one The extended Java3D VRML browser alone consists of over 30000 lines of code. The modification of that browser to enable the transparent access of VRML state required the addition of over 6000 lines of code. The implementation of the RTP/I library contains around 10000 lines of code while generic services and TeCo3D specific functionality amount to another 8000 lines of code.

The RTP/I-based TeCo3D prototype is a proof of concepts for the main ideas presented throughout this thesis. The telecollaboration with collaboration-unaware VRML content has been demonstrated with different VRML models on several occasions. The simple example displayed in Figure 57 shows a model created by SIEMENS. It is a single user model that explains the required steps to change the fuse of an appliance. With the help of TeCo3D this single user model can also be used in a cooperative fashion. SIEMENS plans to use our RTP/I-based TeCo3D prototype as a starting point for the development of a 3D telecollaboration product for call-center and help-desk solutions.

Breaking down the application into several generic components has proven to be a good idea. The reusability of components is very important for distributed interactive media, especially since they tend to be complex, and tedious to implement. Not only will the developers of new applications profit from reusing generic services but the developers of the generic services will also profit from the experiences and bug reports of those who use the service to develop new applications.

In addition to the reusability argument, the decomposition into components also improved the overall architecture of the TeCo3D prototype. An architecture that is built on components rather than on a monolithic piece of functionality is much easier to understand and individual components can be more easily replaced. In this context it is noteworthy that the late-join service has been designed and implemented completely separate from the remainder of the TeCo3D prototype [95]. Nevertheless the integration of the service into the TeCo3D prototype took only a couple of hours and worked right away. This shows that the concept of generic and reusable services based on RTP/I is valid and provides a significant improvement over a situation in which services are application-dependent.

At the start of this prototype’s development we anticipated problems with the consistency service for two reasons:

1. Java might be too slow to provide for accurate synchronization, and
2. NTP might be too inaccurate, causing frequent short-term inconsistencies even with high values for local lag.

Experimenting with our prototype, we were surprised to learn that neither problem occurs in reality. A whole passage of an event from the generation inside the VRML browser on one machine to the delivery of the event to the VRML browser of another machine took less than 6 ms in a local network with two 400 MHz Intel Pentium II PCs. This includes all protocol overheads (encoding/decoding the RTP/I packet and sending it via UDP multicast) and the processing in the synchronization service. Furthermore, NTP proved to be very accurate. Even when NTP servers from different countries were used, the time offset rarely exceeded 20 to 30 ms. These values are very acceptable for local-lag-based synchronization.

The RTP/I-based TeCo3D prototype is a demonstrator model and it is not intended to be a product or to work in a production environment. The main issue that needs to be addressed before it can be employed as a regular tool in teleconferences is the usage of a different VRML browser as a 3D presentation engine.

The Java3D VRML browser that we used for our prototype is still limited in its functionality and contains numerous bugs. Basically we had to fix several bugs for each new VRML model that we wanted to share with the TeCo3D prototype. Moreover, the state access is rather slow when implemented in Java. Extracting and encoding state information can be done within an acceptable amount of time, in the range of 10 ms to 200 ms. However, the decoding of state information and the creation of the 3D model is very slow for complex VRML models. It can range between 50 ms and several seconds. Depending on the number of short-term inconsistencies, a maximum value of around 200 ms to 300 ms would be acceptable.

Currently all major VRML browsers have become available in source code format. We would therefore recommend to use a C/C++ based professional VRML browser implementation as the 3D presentation engine for a TeCo3D product. A prime candidate would be the blaxxun Contact VRML browser [5]. We expect that using a C/C++ based implementation will provide the required speedup to bring the time for restoring the state of complex VRML models into an acceptable range.

7.4 Chapter Summary

In this chapter we have discussed the three TeCo3D prototypes that were developed in the context of this thesis. All prototypes have been used with various collaboration-unaware VRML models, ranging from instructions on how to assemble a piece of furniture and changing the fuse in an appliance, to 3D animated cartoons.

The early prototypes demonstrated that sharing collaboration-unaware VRML is generally possible. At the same time they provided us with important insights into the key

problems of distributed interactive media. In detail these were: guaranteed eventual consistency, late-join support, and recording capability. Furthermore we realized that a common foundation is required for distributed interactive media upon which generic solutions to these problems could be built in a reusable fashion.

As a result, the third prototype was developed based on the ideas and concepts described in the previous chapters of this thesis. This prototype is a proof of concepts and demonstrates that the idea of a common application level protocol and generic services for distributed interactive media is valid. The prototype, including all the generic services and the usage of RTP/I, is fully functional. Only the VRML browser used as the 3D presentation engine is in a very early stage of development. SIEMENS plans to develop a call-center product based on this prototype.

8 Conclusion and Future Work

8.1 Conclusion

The key idea that we have presented in this thesis is that distributed applications that involve user interactions are not unrelated to each other. We demonstrated that shared whiteboards, distributed virtual environments, and distributed simulations share many common traits and need similar functionality. They therefore form a class of related media. We called this media class *distributed interactive media*. Furthermore we showed that it is desirable to be able to develop the common functionality for this media class in a way that allows the easy reuse of the functionality for many applications. A piece of common functionality that is developed in this way was named *generic service*.

In order to prove that these ideas are valid we developed an abstract media model for the distributed interactive media class. The main components of this media model are the shared state of a distributed interactive application and the events (e.g., user actions) that can change that state. Furthermore we distinguished discrete and continuous distributed interactive media. While the state of discrete distributed interactive media can only be changed because of events, the state of continuous distributed interactive media may also change because of the passage of time. It was demonstrated that this media model can be applied to diverse media such as shared whiteboards, distributed virtual environments and distributed simulations.

With the abstract media model defined we started to explore several aspects of this media class in more detail. These ranged from network protocol usage to the question of where the shared state of a distributed interactive medium is managed. In particular we were concerned with how to realize consistency for the shared state of a distributed interactive medium. While there has been extensive work on the consistency for discrete distributed interactive media, consistency in the continuous domain was not fully understood when we started our work in this area.

We demonstrated that consistency for continuous media cannot be achieved by the same means that are used for discrete media. Therefore a formal consistency criterion for con-

tinuous distributed interactive media was introduced. Furthermore two criteria were defined for the consistency-related fidelity of a medium: the avoidance of short-term inconsistencies and the responsiveness of a medium. We learned that these two criteria conflict. This led us to understand the important trade-off between responsiveness and short-term inconsistencies. We demonstrated how it can be exploited to increase the overall consistency-related fidelity of a continuous distributed interactive medium. To this end we introduced the concept of local lag.

While the abstract media model allowed us to discuss this media class independent of individual media, it was not sufficient to develop generic services that can be easily reused for distinct applications. In order to provide a base upon which such services can be developed, we designed the real-time application level protocol for distributed interactive media (RTP/I). This protocol captures the common aspects of distributed interactive media and allows the development of generic services that are based solely on these common aspects. We reused in the design of RTP/I the experiences gained with the real-time transport protocol RTP, which is used for distributed non-interactive media, such as audio and video transmissions. Basically RTP/I is an adaptation of and extension to RTP to meet the specific needs of distributed interactive media. The full specification of RTP/I has been published as an Internet Draft.

Based on RTP/I we designed and implemented three generic and reusable services. The first generic service ensures eventual consistency for continuous distributed interactive media. It guarantees that the consistency criterion for continuous distributed interactive media be fulfilled by using a robust and light-weight floor control service and the concept of local lag.

The second generic service allows latecomers to join an ongoing session. We called this a generic late-join service. This service is highly customizable to the individual needs of different applications. At the same time, it is efficient and scales well to large numbers of participants.

The third service realized the synchronized recording and playback of sessions involving RTP/I and RTP based applications. This service was developed as middleware that can reside within the network. The main problems that had to be solved were how to provide random access to the recorded streams and how to ensure the consistency during recording and playback.

Throughout this thesis the development of a 3D telecollaboration application called TeCo3D was used both as motivation and as proof of concepts for our work on distributed interactive media. TeCo3D provides a shared workspace for 3D models that are specified using the Virtual Reality Modeling Language (VRML). Since the VRML models are dynamic and interactive, TeCo3D realizes a continuous distributed interactive

medium. The key aspect of TeCo3D is that the VRML models that are presented on the shared workspace do not need to be collaboration-aware. Therefore arbitrary VRML content, such as that generated by 3D presentation and animation programs, can be shared.

In order to provide the functionality of sharing collaboration-unaware VRML content, we had to solve two major problems: how to get and set the state of arbitrary VRML content, and how to share user interactions with collaboration-unaware VRML models. The first problem was solved by developing a standardized extension to existing VRML browsers. This extension allows applications that employ the VRML browser as a 3D presentation engine to get and set the state of arbitrary VRML content in a standardized way. The second problem was solved by transparently replacing the VRML sensors for user input with customized, collaborative versions of the sensors. The customized sensors capture the user interactions with the VRML model and hand it to the application instead of forwarding it directly to the VRML model. Similarly, an application can inject an event into the customized sensors, which will then react as if the local user had interacted with the model. In this way an application is informed about the interactions of the local user with the model and can forward this information to remote TeCo3D applications.

We have implemented three distinct prototypes of TeCo3D. The first two demonstrated the feasibility of the idea to share collaboration-unaware VRML content. During the development of the first two prototypes we realized that they share many aspects with other applications, such as shared whiteboards or networked computer games. Therefore we started our work on defining an abstract media model and an application level protocol for distributed interactive media.

The third prototype is a proof of concepts for the ideas presented in this thesis. It is based on RTP/I and uses the generic services for consistency, late-join, and recording. Only the actual rendering of 3D data and the forwarding of messages between the generic services are implemented as medium-specific functionality. The remaining parts - reliability support, application level protocol, and the generic services - are reusable for other distributed interactive media. The third prototype is a fully functional 3D telecollaboration application. SIEMENS plans to use the RTP/I-based TeCo3D prototype as a starting point for the development of a 3D telecollaboration product for call-center and help-desk solutions.

The overall results of this thesis show that it is a valid idea to regard distributed interactive media as a media class where the members of this class have many traits in common. The design and implementation of three generic services that are based on RTP/I demonstrates that a common application level protocol can be used to establish a solid founda-

tion for generic services. Moreover, the development of the complex real-world application TeCo3D illustrates that *it is possible to use the generic services and RTP/I as the building blocks for a distributed interactive application*.

8.2 Future Work

Many signs indicate that distributed interactive media are going to be a very active area of research in the near future. While this thesis may have provided an initial spark for the research on this media class, there are many items of future work that we did not address.

Most important among them is the advancement of RTP/I on the Internet standards track. The current RTP/I specification has been submitted as an Internet Draft. Only when the application level protocol becomes an accepted standard will the benefits of reusable and generic services become attractive to a large audience of software developers.

One fundamental requirement for the advancement of RTP/I on the Internet standards track is that applications other than TeCo3D use RTP/I as their application level protocol. Currently there are three additional applications that have started the integration of RTP/I: the digital lecture board and a toolkit for remotely controlled Java applets for teleteaching [45,46] both developed at the University of Mannheim, and the Audio Enabled Multicast VNet project [81] at the Communications Research Center in Ottawa/Canada. We expect that the integration of RTP/I into other applications for distributed interactive media will provide valuable feedback on how to improve it.

Besides new applications that rely on RTP/I, we also expect that new generic services will emerge. Potential candidates for such services are a generic encryption and privacy service, a consistency service for discrete distributed interactive media, and a consistency service for continuous distributed interactive media that supports time-warp. Similar to new applications, the additional generic services are likely to yield further information that can be used to improve RTP/I.

Once there exists a larger number of applications and generic services, it will be important to specify RTP/I profiles. These profiles will account for the individual sub-classes of distributed interactive media. By using RTP/I profiles the core protocol can be adapted to the specific needs of an individual sub-class. It will be a challenging task to develop RTP/I profiles in a way that ensures interoperability with the existing generic RTP/I services.

Another area of future research deserving particular attention is reliability support. Throughout this thesis we have shown that existing approaches, both at the transport and at the application level, do not provide optimal support for reliability in distributed interactive media. While we have suggested an interface for libraries that provide reliability

support, it remains for such a library to be designed and implemented. In particular, a reliability service for distributed interactive media needs to provide quality-of-service levels that can be determined on a per sub-component and ADU type basis. Furthermore, it should provide a combination of forward error correction and retransmission-based reliability to account for the different needs of state and event transmissions.

In the context of our work on consistency for continuous distributed interactive media we have conducted preliminary experiments to get an impression of the amount of local lag that is acceptable. In order to put these results on a more solid and scientific footing it is required to conduct a series of perceptional psychological experiments. Ideally these will be done for different operations such as clicking on objects, drag-and-drop operations, and text input. With the results of these experiments it will be possible to determine what amount of local lag should be chosen for a given continuous distributed interactive medium.

In order to turn our 3D telecollaboration application TeCo3D into a product, there is one major step left to be taken. The current version of the Java 3D-based VRML browser needs to be replaced with a professional VRML browser such as Blaxxun's Contact. One challenge for this replacement lies in the integration of the state access and encoding functionality into the new VRML browser. However, this is only a matter of implementation and not of scientific research. A more interesting item of future work for TeCo3D is to move to X3D [102], the successor of VRML, as the description language for the 3D objects that can be imported into the shared workspace. Since X3D retains the idea of sensors as methods of capturing user actions, our mechanisms should also work for X3D objects. However, a more detailed investigation of how to develop collaborative versions of sensors and how to access and encode the state of X3D objects would be required before an X3D based telecollaboration tool could be developed.

A Grammar for the Encoding of VRML State

The specification of our binary state encoding is influenced by the proposal for a VRML Compressed Binary Format (CBF) [99]. Besides compression, the CBF proposal specifies a grammar for the binary description of VRML worlds. This grammar is semantically equivalent to the plaintext encoding contained in the VMRL97 international standard. Stripped of the parts that concern compression, and enhanced to support additional state information, the grammar of the CBF proposal can be used to define an encoding for the VRML state. In this appendix we describe the grammar for the encoding of complete VRML worlds, sub-components, and delta states. The grammar presented in A.1 serves only to demonstrate the basic ideas of the encoding, it has been simplified for this purpose. A full definition of the grammar can be found in A.2.

A.1 Basic Grammar

A.1.1 Elements of the VRML State

Figure 58 shows the top-level structure for the encoding of VRML state information. The description of a VRML state is a sequence of bytes that starts with a header. This header identifies that the sequence of bytes contains a VRML state and includes the version number of the encoding. In order to be able to decode a state in the persistence format, a decoder needs to know the type of information encoded (a complete world vs. a single node and a full state vs. a delta state). This information is included in a TYPE byte that follows the header.

The main part for the state encoding is divided into the VRML browser state and the state of the scene graph. Information that is needed to define the state of the browser includes the user's point of view, the point of time the encoding was done, and the URL of the current world. The state of the scene graph contains the description of node states and routes.

While the encoding and semantics of the VRML browser state are straightforward, the description of the scene graph state needs more detailed consideration. Its fundamental

elements are the states of nodes and routes. In the following we will explain how we encode these elements.

A.1.2 Encoding of Nodes and Routes

As shown in Figure 59 the encoding of a node state starts with a unique node ID. A node's ID may change with each full-state encoding. However, it is illegal to change the ID of a node for the encoding of a delta state. Given these restrictions, delta states can reference nodes that were defined in the previous full state. At the same time, the number space does not fill up with deleted nodes since each full-state encoding can reuse the IDs of deleted nodes. Besides their reference purpose in delta states, the IDs are used to define the target nodes for route statements.

```
VRMLSTATE ::=
    HEADER                    // #VRMLSTATE 1.0\n
    TYPE                      // Information about the encoded content
    BROWSER                   // State of the VRML browser
    SCENEGRAPH;               // State of the scene graph

TYPE ::=
    BIT isCompleteWorld       // encoding complete world (1) vs.
                              // encoding single node (0)
    BIT isFullState           // encoding full state (1) vs.
                              // encoding delta state (0)
    UINT(6);                  // padding: ignored

BROWSER ::=
    DOUBLE currentTime        // the point of time where the state
                              // was recorded
    STRING URL                // the current URL
    if (isCompleteWorld) {
      NODE pointOfView        // the user's point of view is encoded
                              // as a Viewpoint node
    };

SCENEGRAPH ::=
    UINT(32) nNODE            // number of top-level nodes
    UINT(32) nROUTE           // number of routes
    NODE[nNODE]               // state of nodes
    ROUTE[nROUTE];            // state of routes
```

Figure 58: Top-level structure of the VRML state

In order to provide important information to the decoder, the format of a node is encoded as a special byte. This byte contains information for the decoding of delta states and will be explained later.

Nodes have a type to identify what kind of node is being encoded. The values 1 to 54 indicate a default node from the VRML97 specification, where the number is derived from the node's alphabetic order, e.g., Anchor is encoded as 1 and WorldInfo as 54. Negative numbers are used to indicate instances of prototypes (such as the customized cooperative sensors for TeCo3D). Finally a list of node elements is present that is used to

encode the values for the fields of the node. This is the place where the most important state information is kept.

```
NODE ::=
    UINT(32) nodeID                                  // unique ID for this node
    NODEFORMAT                                       // what kind of node is
                                                     // encoded?
    if ((!isUNMODIFIED and
       !isDELETED) or
        isFULLSTATE) {                    // node skipped for delta encoding?
        NODETYPE                          // what type of node is encoded?
        NODEFIELDS                        // regular encoding of node elements
        }
      }
    };

NODEFORMAT ::=
    BIT isUNMODIFIED          // delta encoding:
                              // is this node unmodified (1)?
    BIT isDELETED             // delta encoding:
                              // has this node been deleted (1)?
    UINT(6);                  // padding: ignored;

NODEFIELDS ::=
    UINT(32) nNODEFIELDS
    NODEFIELD[nNODEFIELDS];       // list of fields, exposed fields and
                                  // events out

NODEFIELD ::=
    FIELDNUMBER               // this identifies the node element
                              // which is encoded
    FIELDVALUE;;;             // the value of the node element

ROUTE ::=
    UINT(32) routeID          // unique ID for this route
    if (!isFullState) {       // is this a delta encoding?
      ROUTEFORMAT             // delta encoding:
                              // is this a modified or deleted route?
    }
    if (isFullSate or
     (!isUNMODIFIED and
      !isDELETED) {
      UINT(32) fromNodeID       // source node
      FIELDNUMBER fromField     // source node element
      UNIT(32) toNodeID         // target node
      FIELDNUMBER toField       // target node element
    };

ROUTEFORMAT ::=
    BIT isUNMODIFIED          // delta encoding: is this route unmodified?
    BIT isDELETE// delta encoding: has this route been deleted?
    UINT(6);                  // padding: ignored
```

Figure 59: Encoding nodes and routes

The basic rule for the encoding of node elements is that any node element that does not contain the default value of its data type must be included in a `NODEFIELDS` list. Each list entry consists of two parts: the first part of a node field entry is a number that is defined by the occurrence of the node element in the VRML97 specification and that uniquely identifies the field. The second part is the value of the element. Each elementary

VRML data type (e.g., `SFBool` or `MFString`) has its own encoding which is not shown here.

The encoding of a route starts with a unique ID. Similar to the node ID, the route ID is used to reference the route within delta states. Thus they are subject to the same requirements as node IDs with regard to routes changing their IDs. In the event that a full state is encoded (as opposed to a delta state), the ID is followed by the description of the route's source. The source is defined by the ID of the source node and the number of the source field in that node. The target of a route is specified by the ID of the target node and the number of the target field in the target node.

A.1.3 Encoding of Sub-Components

The encoding of the state of a single VRML node (sub-component) reuses the grammar presented in Figure 58 and Figure 59. The only part of the grammar that is specifically designed to support sub-components is the state of the browser. When encoding a single node the browser state does not contain the user's viewpoint. This is reasonable since the decoding of a sub-component should affect only the sub-component but not the global behavior of a VRML world.

Besides the different encoding of the browser state some additional restrictions have to be taken into account when encoding the state of a sub-component:

- **Nodes**. On the top-level there is exactly one node present - the node that represents the sub-component.
- **Routes**. Routes are only encoded when source and target of a route are part of the sub-component. This ensures that the encoded sub-component is self-contained. After a sub-component has been decoded, new routes may be set up connecting the sub-component with other parts of a VRML world by using the script node API or the EAI.

A.1.4 Delta States

A delta state describes the difference of the current state from the last encoded full state. The grammar from Figure 58 and Figure 59 is also used to encode delta states. It is the scene graph part of the state information that is modified when a delta state is encoded. The browser state part remains unchanged from the encoding of full states.

For the nodes part of the scene graph only those nodes are completely encoded that have been modified or added since the last extracted full state. Unmodified nodes are included only by referencing their IDs. Whenever a node is encoded there are two possibilities:

- **The node and all of its descendants are unmodified**. In this case the node is encoded by supplying its ID and format with the `isUNMODIFIED` flag set to one.
- **A node element or any descendant of the node has been modified**. In this case the node is completely encoded with `isUNMODIFIED` set to zero. The decision whether

or not to do a complete encoding must then be made for each individual descendant of the node.

While it would be possible to encode delta states on the fine-grained basis of node fields, this would require the browser to manage a large overhead just for delta-state encoding. We view the proposed level of granularity as a compromise between size and encoding efficiency.

The delta encoding of routes is performed in the same way as the encoding of nodes. The flags present for delta encoded routes are used to signal which routes remained unchanged, which were deleted, and which were changed or added.

A.2 Full Grammar

The following grammar completely specifies the VRML state encoding. It is derived from the CBF specification [99].

```
VRMLSTATE ::=
    HEADER                    // #VRMLSTATE 1.0\n
    TYPE                      // Information about the encoded content
    BROWSER                   // State of the VRML browser
    SCENEGRAPH;               // State of the scene graph


HEADER

The header consists of the following sequence of ASCII bytes: “#VRMLSTATE 1.0
binary\n”


TYPE ::=
    BIT isCompleteWorld       // encoding complete world (1) vs.
                              // encoding single node (0)
    BIT isFullState           // encoding full state (1) vs.
                              // encoding delta state (0)
    BIT isCompleteList        // delta states: lists of nodes are
                              // encoded with the
                              // Complete List (1) or the Changes Only (0) method
    UINT(5);                  // padding: ignored


BIT ::=
    UINT(1)                   // encoding a boolean value: 1=true 0=false


UINT(n)

A binary encoded n bit unsigned integer.


BROWSER ::=
    DOUBLE currentTime        // the point of time where the state was recorded
    STRING URL                // the current URL
    if (isCompleteWorld) {
      NODE pointOfView        // the user’s point of view is encoded as
                              // a Viewpoint node
      STACKORDER background   // stack order of bindable nodes
```

```
      STACKORDER fog
      STACKORDER navigationInfo
      STACKORDER viewPoint
    }
```

DOUBLE

A double is represented using the IEEE 64bit format

```
STRING ::=
    UINT(32) nUTF8    // Then length of the string
    UTF8[nUTF8];
```

UTF8

A UTF8-encoded character.

```
NODE ::=
    UINT(32) nodeID                                  // unique ID for this node
    NODEFORMAT                                       // what kind of node
                                                     // is encoded?
    if ((!isUNMODIFIED and !isDELETED) or isFULLSTATE){ // node skipped for
                                                     // delta encoding?
      if (isUSE){                                    // is this a USE node?
        UINT(32) referencedNodeID                    // DEFed node for this
                                                     // USE node
      } else {
        if (isDEF) {                                 // is this a DEFed node?
          STRING                                     // name of the DEFed node
        }
        NODETYPE                                     // what type of node
                                                     // is encoded
        if (NODETYPE==39) {                          // is it a Script node?
          SCRIPT                                     // special encoding for
                                                     // script nodes
        } else if (NODETYPE==24) {                   // is it an Inline node?
          INLINE                                     // special encoding for
                                                     // inline nodes
        } else is (NODETYPE<0) {                     // is it an instance
                                                     // of a prototype?
          PROTOINSTANCE                              // special encoding
                                                     // for prototypes
        } else {
          NODEFIELDS                                 // regular encoding of
                                                     // node elements
        }
      }
    };
```

```
NODEFORMAT ::=
    BIT isUSE            // a USE node?
    BIT isDEF            // a DEF node?
    BIT hasIS            // if this node is in a PROTO definition,
                         // does it have IS statements?
    BIT isUNMODIFIED     // delta-encoding: is this node unmodified?
    BIT isDELETED        // delta-encoding: has this node been deleted?
    UINT(3);             // padding: ignored
```

```
NODETYPE ::=
    SIGNEDINT;         // 1=Anchor, ...,negative numbers=prototype instances


SIGNEDINT ::=
    BIT sign           // 1=negative 0=positive
    UINT(31);          // the absolut value


SCRIPT ::=
    SCRIPTINTERFACEDELARATION    // interface declarartion of the Script node
    NODEFIELDS                   // encoding of the node elements
    BIT isCustomizedState        // does the script use the state API
    UINT(7)                      // padding
    UINT(32) length              // length of the script state
    BYTE[length];                // script state


SCRIPTINTERFACEDECLARATION ::=
    UINT(32) nInEvent            // number of eventIn EVENTs
    UINT(32) nOutEvent           // number of eventOut EVENTs
    UINT(32) nField              // number of INTERFACEFIELDs
    FIELD[nInEvent]              // eventIn
    FIELD[nOutEvent]             // eventOut
    FIELD[nField];               // fields


FIELD ::=
    STRING                       // field/event name
    FIELDTYPE;                   // field/event type


FIELDTYPE ::=
    SIGNEDINT;   // Numbered according to the field specification in
                 // the VRML standard
                 // negative numbers designate MF fields


NODEFIELDS ::=
    UINT(32) nNODEFIELDS
    NODEFIELD[nNODEFIELDS]   // list of fields, exposed fields and events out
    if(isPROTO && hasIS {) { // isPROTO is set if NODEFIELDS is contained
                             //within a PROTO
      UINT(32) nIS           // number of IS definitions
      IS[nIS]                // IS definitions
    };



NODEFIELD ::=
    FIELDNUMBER              // this identifies the node element
                             // which is encoded
    FIELDVALUE;;             // the value of the node element


FIELDNUMBER ::=
  SIGNEDINT; // The number of the field in the VRML standard


FIELDVALUE ::=                 // fieldType is known by context
    select fieldType
    case SFBool:
```

```
      BIT[7]                    // padding, ignored
      BIT                       // 0=False, 1=True
    case SFColor:
      FLOAT[3]
    case MFColor:
      UINT(32) nColor           // number of elements in array
      FLOAT[3*nColor];          // color values
    case SFFloat:
      FLOAT
    case MFFloat:
      UINT(32) nFloat           // number of elements in array
      FLOAT[nFloat];            // float values
    case SFImage:
      UINT(32) width            // width
      UINT(32) height           // height
      UINT(32) nCompon          // number of components in the image
      BYTE[width*height*nCompon]// image-data
    case SFInt32:
      SIGNEDINT                 // see notes
    case MFInt32:
      UINT(32) nInt             // number of elements in arry
      SIGNEDINT[nInt];          // integer values
    case SFNode:
      NODE
    case MFNode:
      UINT(32) nNode            // number of elements in array
      NODE[nNode]               // nodes
    case SFRotation:           // see notes
      FLOAT[4]
    case MFRotation:
      UINT(32) nRotation        // number of elements in array
      FLOAT[4*nRotation]           // rotation values
    case SFString:
      STRING
    case MFString:
      UINT(32) nString          // number of elements in array
      STRING[nString]           // string values
    case SFTime:
      DOUBLE
    case MFTime:
      UINT(32) nTime            // number of elements in array
      DOUBLE[nTime]             // time values
    case SFVec2f:
      FLOAT[2]
    case MFVec2f:
      UINT(32) nVec2f           // number of elements in array
      FLOAT[nVec2f]             // vector values
    case SFVec3f:
      FLOAT[3]
    case MFVec3f
      UINT(32) nVec3f           // number of elements in array
      FLOAT[nVec3f]             // vector values
```

FLOAT

A float is represented using the IEEE 32 bit format.

```
BYTE ::=
  BIT[8];
```

```
IS ::=                      // (x IS y)
    FIELDNUMBER              // xFieldNumber (from current NODE)
    FIELDNUMBER;             // yFieldNumber (from current PROTO)


INLINE ::=
    NODEFIELDS              // standard encoding of node elements
    SCENEGRAPH;             // scene graph of the imported world


SCENEGRAPH ::=
    UINT(32) nNODE            // number of top-level nodes
    UINT(32) nROUTE           // number of routes
    NODE[nNODE]               // state of nodes
    ROUTE[nROUTE];            // state of routes


ROUTE ::=
    UINT(32) routeID          // unique ID for this route
    if (!isFullState) {       // is this a delta-encoding?
      ROUTEFORMAT             // delta-encoding: is this a modified or
                              // deleted route?
    }
    UINT(32) fromNodeID       // source node
    FIELDNUMBER fromField     // source node element
    UNIT(32) toNodeID         // target node
    FIELDNUMBER toField;      // target node element


ROUTEFORMAT ::=
    BIT isUNMODIFIED          // delta-encoding: is this route unmodified?
    BIT isDELETE              // delta-encoding: has this route been deleted?
    UINT(6);                  // padding: ignored


PROTOINSTANCE ::=
    NODEFIELDS              // the encoded node elements of this instance
    SCENEGRAPH;             // the scene graph of this instance


STACKORDER ::=
    nStackSize              // number of elements on the stack
    UINT(32)[nStackSize]      // the stack od node IDs
```

References

[1] K. Almeroth and M. Ammar. The Interactive Multimedia Jukebox (IMJ): A New Paradigm for the On-Demand Delivery of Audio/Video. In: *Proc. of the Seventh International World Wide Web Conference*, Brisbane, Australia, 1998. Available on CD-ROM.

[2] A. L. Ames, D. R. Nadeau and J. L. Moreland: *VRML 2.0 Sourcebook*, Second edition, John Wiley & Sons, New York, 1997.

[3] C. Bacher, R. Müller, T. Ottmann and M. Will. Authoring on the Fly. A new way of integrating telepresentation and courseware production. In: *Proc. of ICCE '97*, Kuching, Sarawak, Malaysia, 1997, pp. 89 - 96.

[4] A. Ballardie, B. Cain and Z. Zhang. *Core Based Trees (CBT version 3) Multicast Routing - Protocol Specification*. Internet Draft, 1998. Work in progress.

[5] Blaxxun. blaxxun Contact. http://www.blaxxun.com/products/contact/index.html

[6] R. Braden (Editor). *Requirements for Internet Hosts -- Communication Layers*. Internet Request for Comments 1122, Internet Engineering Task Force, Network Working Group, 1989.

[7] D. P. Brutzman. The Virtual Reality Modeling Language and Java, *Communications of the ACM*, Vol. 41, No. 6, June 1998, pp. 57 - 64.

[8] Cosmo Software. Cosmo Player 2.1. On-line: http://www.cosmosoftware.com or http://www.karmanaut.com/cosmo/player/

[9] D. Clark and D. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In: *Proc. of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM) '90*, 1990, pp. 201 - 208.

[10] F. Cristian, H. Ahali, R. Stron, and D. Dolev, Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement. *In Proc. of the 15th Int. Symp. on Fault-Tolerant Computing (FTCS-15),* Ann Arbor, MI, USA, pp. 200 - 206, IEEE Computer Society Press, 1985.

[11] S. Deering, D. Estrin, D. Farinacci, A. Helmy, D. Thaler, M. Handley, V. Jacobson, C. Liu, P. Sharma and L. Wei. *Protocol Independent Multicast-Sparse Mode (PIM-SM): Protocol Specification*. Internet Request for Comments, RFC 2362, 1998.

[12] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, A. Helmy, D. Meyer and L. Wei. *Protocol Independent Multicast Version 2 Dense Mode Specification*. Internet Draft, draft-ietf-pim-v2-dm-01.txt, 1998. Work in progress.

[13] P. Diefenbach, P. Mahesh, D. Hunt. Building OpenWorlds (TM). In: *Proc. of the Third Symposium on the Virtual Reality Modeling Language (VRML) '98*, Monterey, California, USA, published by ACM SIGGRAPH, New York, New York, USA, 1998, pp. 33 - 38.

[14] H. P. Dommel, J. J. Garcia-Luna-Aceves. Floor control for multimedia conferencing and collaboration. In: *Springer/ACM Journal on Multimedia Systems*, Vol. 5, No. 1, 1997, pp. 23 - 38.

[15] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In: *Proceedings of the 1989 ACM SIGMOD Conference on Managements of Data*, Portland, OR, USA, 1989, pp. 399 - 407.

[16] B. Fenner. *Internet Group Management Protocol, Version 2*. Internet Request for Comments, RFC 2236, 1997.

[17] S. Floyd, V. Jacobson, C. Liu, S. McCanne and L. Zhang. A reliable multicast framework for leight-weight sessions and application level framing. In: *IEEE/ACM Transactions on Networking*, Vol. 5, No. 6, 1997, pp. 784 - 803.

[18] E. Frécon and M. Stenius. Dive: A scaleable network achitecture for distributed virtual environments. *Distributed Systems Engineering Journal* (special issue on Distributed Virtual Environments), Vol. 5, No. 3, 1998, pp. 91 - 100.

[19] T. Fuhrmann. *On the Scaling of Feedback Algorithms for Very Large Multicast Groups*, Technical Report TR-01-2000, Reihe Informatik, Department for Mathematics and Computer Science, University of Mannheim, February 2000.

[20] R. M. Fujimoto. Parallel Discrete Event Simulation. *Communications of the ACM*, Volume 33, Number 10, 1990, pp. 30 - 53.

[21] D. Garfinkel, B. Welti, and T. Yip. HP SharedX: A Tool for Real-Time Collaboration. In: *Hewlett-Packard Journal*, April 1994, pp.23-36.

[22] W. Geyer and W. Effelsberg. The Digital Lecture Board - A Teaching and Learning Tool for Remote Instruction in Higher Education. In: *Proc. of 10th World Conference on Educational Multimedia (ED-MEDIA) '98*, Freiburg, Germany, 1998. Available on CD-ROM.

[23] W. Geyer and R. Weis. A Secure, Accontable, and Collaborative Whiteboard. In: *Proc. of the International Workshop on Distributed Multimedia Systems and Services (IDMS) '98*, Oslo, Norway, Springer, Heidelberg, LNCS 1483, September 1998, pp. 3 - 14.

[24] W. Geyer and R. Weis. The Design and the Security Concept of a Collaborative Whiteboard. In: *Computer Communications* 23(2000), Elsevier, 2000, pp. 233-241.

[25] W. Geyer. *Das digital lecture board*. Ph.D. Thesis, University of Mannheim, Department for Mathematics and Computer Science, in German, ISBN 3-89601-458-7, infix, Sankt Augustin, Germany, 1999.

[26] W. Geyer and M. Mauve. Integrating Support for Collaboration-unaware VRML Models into Cooperative Applications. In: *Proc. of IEEE Multimedia Systems (ICMS) '99*, Florence, Italy, published by IEEE Computer Society, Los Alamitos, California, USA, Vol. 1, pp. 655 - 660.

[27] W. Geyer, J. Vogel, and M. Mauve. An Efficient and Flexible Late Join Algorithm for Shared Whiteboards. To appear in: *Proc. of the Fifth IEEE International Symposium on Computers and Communications (ISCC'2000)*, Antibes, France, July 4-6, 2000.

[28] W. Geyer. digital lecture board web pages. http://www.informatik.uni-mannheim.de/~geyer/dlb/dlb.eng.html

[29] T. Gutekunst, D. Bauer, G. Caronni, and B. Plattner. A distributed and policy-free general-purpose shared window system. IEEE/ACM Transactions on Networking, Vol. 3, No. 1, February 1995, pp. 51 - 62.

[30] O. Hagesand. Interactive multiuser VEs in the DIVE system. In: *IEEE Multimedia*, Vol. 3, No. 1, 1996, pp. 30 - 39.

[31] J. Hartman, J. Wernecke. *The VRML 2.0 Handbook*, Addison-Wesley, Reading Massachusetts, 1996.

[32] V. Hilt and W. Geyer. A Model for Collaborative Services in Distributed Learning Environments. In: *Proc. of the International Workshop on Interactive Distributed Multimedia Systems and Telecommunications Services (IDMS) '97*, Darmstadt, Germany, R. Steinmetz and L. Wolf (Eds.), LNCS 1309, Springer Verlag, Berlin, Germany, 1997, pp. 364 - 375.

[33] V. Hilt, M. Mauve, C. Kuhmünch and W. Effelsberg. A Generic Scheme for the Recording of Interactive Media Streams. In: *Proc. of the International Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services (IDMS) '99*, Toulouse, France, M. Diaz et. al. (Eds.), LNCS 1718, Springer Verlag, Berlin, Germany, 1999, pp. 291 - 304.

[34] V. Hilt. *Educational Multimedia Library Project Homepage*. http:// www.informatik.uni-mannheim.de/informatik/pi4/projects/emulib/index.en.html

[35] V. Hilt, W. Geyer and W. Effelsberg. A New Paradigm for the Recording of Shared Whiteboard Streams. In: *Proc. of SPIE Multimedia Computing and Networking (MMCN) 2000*, San Jose, California, January 2000, pp. 154-165.

[36] W. Holfelder. Interactive Remote Recording and Playback of Multicast Videoconferences. In: *Proc. of the International Workshop on Interactive Distributed Multimedia Systems and Telecommunications Services (IDMS) '97*, Darmstadt, Germany, R. Steinmetz and L. Wolf (Eds.), LNCS 1309, Springer Verlag, Berlin, Germany, 1997, pp. 450 - 463.

[37] W. Holfelder. *Aufzeichnung und Wiedergabe von Internet-Videokonferenzen*. In German, Ph.D. Thesis, University of Mannheim, Department for Mathematics and Computer Science, Shaker-Verlag, Aachen, Germany, 1998.

[38] Intervista. WorldView. 1999. http://www.intervista.com.

[39] IEEE Computer Society: *IEEE standard for information technology - protocols for distributed simulation applications: Entity information and interaction*. IEEE Standard 1278-1993. New York: IEEE Computer Society, 1993.

[40] V. Jacobson, S. Casner, R. Frederick and H. Schulzrinne. *RTP: A Transport Protocol for Real-Time Applications*, Internet Draft, Audio/Video Transport Working Group, IETF, draft-ietf-avt-rtp-new-04.txt, 1999. Work in progress.

[41] D. R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, Volume 7, Number 3, 1989, pp. 404 - 425.

[42] C. A. Kent and J. C. Mogul. Fragmentation considered harmful. In: *Proc. of the ACM workshop on frontiers in computer communications technology (ACM SIGCOMM) '87*, Stowe, Vermont, USA, 1987, pp. 390 - 401.

[43] R. G. Kermode. Scoped hybrid automatic repeat reQuest with forward error correction (SHARQFEC). In: *Proc. of the ACM SIGCOMM'98 conference on applications, technologies, architectures, and protocols for computer communication*, Vancouver, Canada, 1998. pp. 278 - 289.

[44] H. Kress and B. Anderson. Shared 3D Viewer - Ein verteiltes 3D Visualisierungssystem für Produktdaten. In German, in: *unix/mail*, Volume 14, Number 5, Carl Hanser Verlag, München, Germany, 1996, pp. 329 - 334.

[45] C. Kuhmünch, T. Fuhrmann and G. Schöppe. Java Teachware - The Java Remote Control Tool and its Applications. In: *Proc. of ED-MEDIA/ED-TELECOM'98*, Freiburg, Germany, 1998, available on CD-ROM.

[46] C. Kuhmünch. *Collaborative Applications in Java*. Technical Report 15-98, University of Mannheim, Department for Mathematics and Computer Science, Mannheim, 1998.

[47] F. Kuo, W. Effelsberg and J.J. Garcia-Luna-Aceves. *Multimedia Communications - Protocols and Applications*. Prentice Hall, Englewood Cliffs, ISBN 0-13-856923-1, Chaper 6, 1997.

[48] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, Volume 21, Number 7, 1978, pp. 558 - 565.

[49] lbsrm. libsrm web pages. http://www.mash.CS.Berkeley.ECU/mash/software/srm2.0

[50] S. Lin, and D. J. Costello. *Error control coding*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1983.

[51] J. C. Lin, and S. Paul. RMTP: A reliable multicast transport protocol. In *Proc. of IEEE INFOCOMM '96*, 1996, Volume 3, pp. 1414 - 1424.

[52] M. R. Macedonia and D. P. Brutzmann. MBone Provides Audio and Video Across the Internet. In: *IEEE Computer*, Vol. 27, No. 4, 1994, pp. 30 - 36.

[53] R. Malpani and L.A. Rowe. Floor Control for Large-Scale MBone Seminars. In: *Proceedings of The Fifth Annual ACM International Multimedia Conference '97*, Seattle, Washington, November 1997, pp 155 - 163.

[54] K. Maly, C. M. Overstreet, A. González, M. Denbar, R. Cutaran, N. Karunaratne. Automated Content Synthesis for Interactive Remote Instruction, In: *Proc. of ED-MEDIA '98*, Freiburg, Germany, AACE, June 1998. Available on CD-ROM.

[55] S. McCanne, et. al. Toward a Common Infrastructure for Multimedia-Networking Middleware. In: *Proc. of the 7th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV) '97)*, St. Louis, Missouri, USA, 1997, pp.39 - 49.

[56] S. McCanne, R. Katz, E. Brewer et. al. *MASH Archive System*. http://mash.CS.Berkeley.edu/mash/overview.html

[57] M. Mauve. *Protocol Enhancement and Compression for X-Based Application Sharing*, Master's Thesis, University of Mannheim, Department for Mathematics and Computer Science, University of Mannheim, 1997.

[58] M. Mauve. TeCo3D: a 3D telecooperation application based on VRML and Java. In: *Proc. of SPIE Multimedia Computing and Networking (MMCN) '99*, San Jose, CA, USA, published by SPIE, Bellingham, Washington, USA, January 1999, pp. 240 - 251.

[59] M. Mauve. Transparent Access to and Encoding of VRML State Information. In: *Proc. of the Fourth Symposium on the Virtual Reality Modeling Language (VRML) '99*, Paderborn, Germany, published by ACM SIGGRAPH, New York, USA, February 1999, pp. 29 - 38.

[60] M. Mauve, V. Hilt, C. Kuhmünch and W. Effelsberg. A General Framework and Communication Protocol for the Transmission of Interactive Media with Real-Time Characteristics. In: *Proc. of IEEE Multimedia Systems (ICMS) '99*, Florence, Italy, published by IEEE Computer Society, Los Alamitos, California, USA, June 1999, Vol. 2, pp. 641 - 646.

[61] M. Mauve, V. Hilt, C. Kuhmünch, J. Vogel, W. Geyer and W. Effelsberg. *RTP/I: An Application Level Real-Time Protocol for Distributed Interactive Media.* Internet Draft: draft-mauve-rtpi-00.txt, 2000. Work in progress.

[62] M. Mauve. *Access to and Encoding of VRML State Information*. Technical Report TR-13-98, Reihe Informatik, Department for Mathematics and Computer Science, University of Mannheim, December 1998.

[63] M. Mauve, V. Hilt, C. Kuhmünch and W. Effelsberg. *A General Framework and Communication Protocol for the Real-Time Transmission of Interactive Media*. Technical Report TR-16-98, Reihe Informatik, Department for Mathematics and Computer Science, University of Mannheim, December 1998.

[64] M. Mauve. *Consistency in Continuous Distributed Interactive Media*. Technical Report TR-9-99, Reihe Informatik, Department for Mathematics and Computer Science, University of Mannheim, November 1999.

[65] M. Mauve. *The RTP/I Homepage*. http://www.informatik.uni-mannheim.de/informatik/pi4/projects/RTPI/

[66] E. Mayer. Synchronisation in kooperativen Systemen. In German, Ph.D. Thesis, University of Mannheim, Department for Mathematics and Computer Science, Vieweg Verlag, Braunschweig, Germany, 1994.

[67] Microsoft. Netmeeting Product Page, March 2000. http://www.microsoft.com/windows/netmeeting

[68] D. L. Mills. *Network Time Protocol (Version 3) specification, implementation and analysis*. DARPA Network Working Group Report RFC-1305, University of Delaware, 1992.

[69] W. Minenko. *Advanced Design of Efficient Application Sharing Systems under X Window*. Ph.D. Thesis, Department of Computer Science, University of Ulm, 1996.

[70] J. Mogul, and S. Deering. *Path MTU Discovery*. Internet Request for Comments, RFC 1191, 1990.

[71] J. Moy. Multicast Extension for OSPF. In: *Communications of the ACM*, Vol. 37, No. 8, 1994, pp. 61 - 66.

[72] S. Mukhopadhyay and B. Smith. Passive Capture and Structuring of Lectures. In: Proc. of the ACM Conference on Multimedia ‘99, Orlando, Florida, 1999, pp. 477 - 487.

[73] J. K. Ousterhout. *Tcl and Tk Toolkit*. Addison-Wesley, Reading, Massachusetts, USA, 1994.

[74] P. Parnes, K. Synnes and D. Schefström. *mMOD: the multicast Media-on-Demand system*. http://mates.cdt.luth.se/software/mMOD/paper/mMOD.ps

[75] J. F. Patterson, M. Day and J. Kucan, Notification servers for synchronous groupware. In: *Proc. of the ACM conference on Computer supported cooperative work (CSCW) ‘ 96*, Boston, USA, 1996, p. 122.

[76] C. Perkins and J. Crowcroft. *Notes on the use of RTP for shared workspace applications.* To appear in ACM Computer Communication Review, 2000.

[77] J. Postel. *User Datagram Protocol*, Internet Standard, 1980.

[78] J. Postel. *Transmission Control Protocol*. Internet Standard, 1981.

[79] J. Postel. *Internet Control Message Protocol*. Internet Standard, 1981.

[80] S. Raman and S. McCanne. Scalable data naming for application level framing in reliable multicast. In: *Proc. of 6th ACM International Conference on Multimedia*, Bristol, UK, 1998, pp. 391 - 400.

[81] J. Robinson, J. Stewart, and I. Labbe. WVIP - Audio Enabled Multicast VNet. In: *Proc. of the VRML2000 Symposium on the Virtual Reality Modeling Language (VRML)*, Monterey, California, USA, published by ACM SIGGRAPH, New York, USA, February 2000.

[82] T. von Roden. *Steuerung und Synchronisation verteilter VRML-Animationen in einer kooperativen Lernumgebung*. In German. Master’s Thesis, University of Mannheim, Department for Mathematics and Computer Science, 1998.

[83] H. Schulzrinne. *RTP Tools*. Software available on-line: http://www.cs.columbia.edu/~hgs/rtp/rtptools/

[84] H. Schulzrinne, A. Rao, R. Lanphier. *Real Time Streaming Protocol (RTSP)*. Internet Request for Comments 2326, Multiparty Multimedia Session Control Working Group, IETF, April 1998.

[85] S. Shirmohammadi and N. D. Georganas. JETS: a Java–Enabled TeleCollaboration System, in: *Proc. of IEEE ICMCS*‘97, Ottawa, Canada, 1997, pp. 541 - 547.

[86] SICS. *The DIVE Homepage*. http://www.sics.se/dive/

[87] S. Singhal and M. Zyda. *Networked Virtual Environments Design and Implementation*, ACM press, New York, 1999.

[88] SoftWired. *iBus web pages*. http://www.softwired.ch/ibus

[89] J. Sonstein. *VNet Web Page*. 1999. http://ariadne.iz.net/~jeffs/vnet/

[90] W. T. Strayer, B. J. Dempsey and A. C. Weaver. *XTP: The Xpress Transfer Protocol*. Addison-Wesley, Reading, Massachusetts, 1992.

[91] C. Sun, X. Jia, Y. Zhang, Y. Yang and D. Chen. Achieving Convergence, Causality-preservation, and Intention-preservation in Real-time Cooperative Editing Systems. *ACM Transactions on Computer-Human Interactions*, Vol. 5, No. 1, March 1998, pp 63 - 108.

[92] C. Sun and C. Ellis. Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements. In: *Proceedings of the ACM 1998 conference on Computer Supported Cooperative Work (CSCW'98)*, Seattle, Washinton, USA, 1998, pp. 59 - 68.

[93] T. L. Tung. MediaBoard: *A Shared Whiteboard Application for the MBone*. Master's Thesis, University of California, Berkeley, California, USA, 1998.

[94] University of Mannheim. *TeleTeaching web pages.* http://www.informatik.uni-mannheim.de/informatik/pi4/projects/teleTeaching/

[95] J. Vogel. *Entwurf und Implementierung eines generischen Late Join Mechanismus für interaktive Medien*. In German. Master's Thesis, Department for Mathematics and Computer Science, University of Mannheim, 1999.

[96] VRML Consortium. *Information technology -- Computer graphics and image processing -- The Virtual Reality Modeling Language (VRML) -- Part 1: Functional specification and UTF-8 encoding*, ISO/IEC 14772-1:1997 International Standard, 1997. http://www.vrml.org/Specifications/

[97] VRML Consortium. *Information technology -- Computer graphics and image processing -- The Virtual Reality Modeling Language (VRML) -- Part 2: External authoring interface*. ISO/IEC 14772-2:1999 Committee Draft, 1999. http://www.vrml.org/WorkingGroups/vrml-eai/Specification/

[98] VRML Consortium. *Living Worlds - Making VRML 2.0 Applications Interpersonal and Interoperable - Draft 2*, April 1997. On-line: http://www.vrml.org/WorkingGroups/living-worlds/draft_2/index.htm

[99] VRML Consortium. *The Virtual Reality Modeling Language Compressed Binary Format Specification*, ISO/IEC 14772-3 Editor's Draft, 1997. http:// www.research.ibm.com/vrml/binary/specification/draft5/VRMLB-ED5/spec.DIS/

[100] VRML Consortium. The VRML - Java3D Working Group, 1999. On-line: http://www.vrml.org/WorkingGroups/vrml-java3d/

[101] D. Waitzman, C. Partridge and S. Deering. *Distance Vector Multicast Routing Protocol*. Internet Request for Comments, Network Working Group, RFC 1075, 1988.

[102] Web3D Consortium. *X3D Web page*. http://www.web3d.org/x3d.html

Index

A

B

C

D

E

F

G

I

J

L

M

N

P

R

S

T

U

V

Zusammenfassung

In dieser Arbeit wird gezeigt, daß verteilte interaktive Medien eine Medien-Klasse bilden. Mitglieder dieser Klasse haben viele gemeinsame Eigenschaften. Insbesondere sind sie verteilte Medien, die Benutzerinteraktionen mit dem Medium selbst erlauben. Typische Beispiele hierfür sind shared whiteboard Systeme, Netzwerk-basierte Computerspiele und verteilte Virtual Reality Umgebungen. Um zu zeigen, daß die Idee einer gemeinsamen Medienklasse gültig ist wird ein abstraktes Medienmodell präsentiert. Weiterhin werden existierende verteilte interaktive Medien untersucht und es wird gezeigt, daß diese von dem Modell abgedeckt sind.

Durch die Definition einer Klasse für verteilte interaktive Media wird es möglich eine gemeinsame Basis für die Entwicklung wiederverwendbarer Funktionalität zu erstellen. Die Entwicklung eines Anwendungsprotokolls als eine solche Basis ist der Hauptbestandteil dieser Arbeit. Das hier vorgestellte “Real Time Application Level Protocol for Distributed Interactive Media” (RTP/I) wurde in Anlehnung an das “Real-Time Transport Protocol “ (RTP) entwickelt. RTP hat weite Verbreitung für die Übertragung von Audio und Video über das Internet gefunden. RTP/I verwendet viele Mechanismen von RTP, wurde jedoch von Grund auf an die speziellen Bedürfnisse von verteilten interaktiven Medien angepaßt. Die Design Entscheidungen, die bei dem Entwurf von RTP/I eine wesentliche Rolle gespielt haben, sowie die Spezifikation von RTP/I, werden in dieser Arbeit detailliert besprochen.

Um zu zeigen, daß RTP/I tatsächlich als gemeinsame Basis für die Entwicklung wiederverwendbarer Funktionalität dienen kann, werden drei generische Dienste vorgestellt: ein Dienst zur Konsistenzsicherung, ein Dienst, der es Nachzüglern ermöglicht in eine bestehende Sitzung einzutreten, und ein Dienst für das Aufzeichnen und Abspielen von Sitzungen. Außerdem wird dargestellt, wie das Medien-Modell, RTP/I und die generischen Dienste verwendet wurden um eine komplexe Anwendung mit dem Namen TeCo3D zu realisieren. Diese Anwendung erlaubt einer verteilten Benutzergruppe dynamische 3D Modelle zu betrachten und mit ihnen zu interagieren. Diese Modelle können mit Hilfe der Virtual Reality Modeling Language spezifiziert werden und sie müssen keine spezielle Funktionalität für eine kooperative Benutzung beinhalten.