

Consistency Algorithms and Protocols for Distributed Interactive Applications

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
der Universität Mannheim

vorgelegt von

Diplom-Wirtschaftsinformatiker Jürgen Vogel
aus Ludwigshafen

Mannheim, 2004

Dekan: Professor Dr. Jürgen Potthoff, Universität Mannheim

Referent: Professor Dr. Wolfgang Effelsberg, Universität Mannheim

Korreferent: Professor Dr. Ernst Biersack, Institut Eurécom, Valbonne, Frankreich

Tag der mündlichen Prüfung: 16. Juli 2004

Abstract

The Internet has a major impact not only on how people retrieve information but also on how they communicate. *Distributed interactive applications* support the communication and collaboration of people through the sharing and manipulation of rich multimedia content via the Internet. Aside from shared text editors, meeting support systems, and distributed virtual environments, shared whiteboards are a prominent example of distributed interactive applications. They allow the presentation and joint editing of documents in video conferencing scenarios. The design of such a shared whiteboard application, the *multimedia lecture board (mlb)*, is a main contribution of this thesis. Like many other distributed interactive applications, the mlb has a replicated architecture where each user runs an instance of the application. This has the distinct advantage that the application can be deployed in a lightweight fashion, without relying on a supporting server infrastructure. But at the same time, this peer-to-peer architecture raises a number of challenging problems: First, application data needs to be distributed among all instances. For this purpose, we present the network protocol RTP/I for the standardized communication of distributed interactive applications, and a novel application-level multicast protocol that realizes efficient group communication while taking application-level knowledge into account. Second, consistency control mechanisms are required to keep the replicated application data synchronized. We present the consistency control algorithms “local lag”, “timewarp”, and “state request”, show how they can be combined, and discuss how to provide visual feedback so that the session members are able to handle conflicting actions. Finally, late-joining participants need to be initialized with the current application state before they are able to participate in a collaborative session. We propose a novel late-join algorithm, which is both flexible and scalable. All algorithms and protocols presented in this dissertation solve the aforementioned problems in a generic way. We demonstrate how they can be employed for the mlb as well as for other distributed interactive applications.

Zusammenfassung

Das Internet hat sich zu einem alltäglichen und universellen Werkzeug der Informationsbeschaffung und der zwischenmenschlichen Kommunikation entwickelt. Durch den Austausch von multimedialen Inhalten über das Internet ermöglichen es so genannte *verteilte interaktive Anwendungen* einer Gruppe von Benutzern, miteinander zu kommunizieren und zusammenzuarbeiten. Beispiele für verteilte interaktive Anwendungen sind Mehrbenutzer-Texteditoren, Systeme zur Unterstützung von Projektteams, virtuelle Realitäten und Shared Whiteboards. Letztere dienen zum Präsentieren und Erstellen von Dokumenten in Videokonferenz-Szenarien. Die Entwicklung eines solchen Shared Whiteboards, des *multimedia lecture boards (mlb)*, ist ein zentraler Beitrag dieser Dissertation. Wie viele andere verteilte interaktive Anwendungen auch hat das mlb eine replizierte Architektur, bei der jeder Benutzer auf seinem lokalen Rechner eine vollwertige und gleichberechtigte Anwendungsinstanz ausführt. Eine solche Architektur hat den Vorteil, dass die Anwendung ohne die Installation von zentralen Serverkomponenten sofort einsetzbar ist. Gleichzeitig birgt eine replizierte Architektur aber auch eine Reihe von technischen Herausforderungen, die von einer verteilten interaktiven Anwendung gelöst werden müssen: Zunächst ist es erforderlich, die Anwendungsdaten über geeignete Netzwerkprotokolle zwischen den beteiligten Anwendungsinstanzen auszutauschen. Zu diesem Zweck wurden im Rahmen dieser Dissertation das RTP/I-Protokoll zur standardisierten Kommunikation und ein neuartiges Protokoll zur effizienten Multicast-Kommunikation auf Anwendungsebene unter Berücksichtigung von Anwendungswissen entwickelt. Des Weiteren werden Verfahren zur Konsistenzerhaltung benötigt, die die replizierten Anwendungsdaten miteinander abgleichen und auf dem neuesten Stand halten. Die Dissertation stellt die Konsistenzerhaltungs-Mechanismen Local Lag, Timewarp und Zustandsanfrage vor und zeigt, wie diese kombiniert werden können. Zusätzlich wird ein neues Visualisierungs-Schema eingeführt, das es den einzelnen Benutzern erlaubt, Konflikte beim gleichzeitigen Zugriff auf die Anwendungsdaten zu erkennen und aufzulösen. Als letztem Schwerpunkt beschäftigt sich die Dissertation mit der Frage, wie neu hinzu gekommene Anwendungsinstanzen mit den aktuellen Anwendungsdaten initialisiert werden können. Der hierzu vorgeschlagene Late-Join-Algorithmus ist sowohl effizient als auch flexibel. Alle besprochenen Algorithmen und Protokolle können als generische Lösungen neben dem mlb für eine Vielzahl von verteilten interaktiven Anwendungen verwendet werden.

Acknowledgements

This dissertation was written during my time as a research staff member at the “Lehrstuhl für Praktische Informatik IV”, University of Mannheim, where I found an excellent working atmosphere. First of all, I would like to thank my advisor Prof. Wolfgang Effelsberg, who gave me the opportunity to pursue my research interests and who supported me in every possible way. Prof. Ernst Biersack was so kind to co-advise this dissertation and gave me very detailed and helpful feedback. I am also grateful to the DFN-Verein which sponsored the first two years of my work on the mlb.

Many ideas presented in this dissertation evolved while collaborating with my colleagues. I am very obliged to Martin Mauve and Volker Hilt, who let me participate in their projects. Together, they formalized the abstract data model of distributed interactive applications that is the foundation of my own work, and they started the RTP/I protocol. Martin was the first to research into consistency control for continuous interactive applications and combined the algorithms of local lag and timewarp. Volker designed the ingenious recording service for distributed interactive applications. I very much enjoyed our endless discussions on major steps and minor details.

My colleagues Jörg Widmer and Dirk Farin contributed to the application-level multicast routing algorithm. They were a never-ending source of ideas. I would also wish to thank Christoph Kuhmüch for his advice, for keeping me physically fit, and for sharing the pain of software development. Our sysadmin Walter Müller helped me to uncover the mysteries of computer technology, and without Betty Weyerer I would not have survived the University’s bureaucracy.

I am very much indebted to my student assistants, guest researchers, and master students. Marcel Busse helped with many aspects of this thesis and never hesitated to implement weird features for the mlb. Andrius Kurtinaitis designed a scheme for presentation animations that dwarfs PowerPoint. Martin Arnold turned the multicast routing algorithm into a working Internet protocol.

Werner Geyer has played a major role in shaping this dissertation. He introduced me to the world of science while I was his student assistant, and Werner’s work on the dlb builds the foundation of this thesis. Even after he left the University of Mannheim, Werner was always

a great inspiration to me. I am deeply grateful to him for making possible my stay at IBM Research in Cambridge, Massachusetts, one that broadened my view in many ways.

This dissertation would not have been possible without the support and encouragement of my family, in particular my parents, Gudrun and Reinhard Vogel.

Contents

List of Figures	xi
List of Tables	xiii
Abbreviations	xv
1 Introduction	1
1.1 Problem Statement and Outline	2
2 Distributed Interactive Applications	5
2.1 Architecture of Distributed Interactive Applications: Centralized vs. Replicated	6
2.2 Formalization of the Data Model	7
2.3 Design Considerations	9
2.3.1 Collaboration Management and User Interface	9
2.3.2 Synchronization Algorithms	12
2.3.3 Communication Model	13
2.3.4 Generic Services	17
2.4 Selected Distributed Interactive Applications	18
2.4.1 TeCo3D	18
2.4.2 Spaceshooter	19
2.4.3 Instant Collaboration	20
2.5 Conclusions	22
3 The multimedia lecture board (mlb)	23
3.1 Related Work	24
3.2 Basic Features	26
3.3 Data Model and Architecture	28
3.3.1 Unique Identifiers	29
3.3.2 Display Order of Objects	30
3.3.3 Data Model	31
3.3.4 Communication Model	32
3.4 Presentation Animations	33
3.5 Collaboration and Awareness	34

3.5.1	Visualization of Participant Information	34
3.5.2	Telepointer	35
3.5.3	Hand-Raising	37
3.5.4	Chat	38
3.5.5	Voting and Feedback	38
3.5.6	Application Launch	39
3.6	Handheld Devices and Pen-Based Interaction	40
3.7	The mlb as a Software Project	42
3.8	Conclusions	42
4	Consistency Control	45
4.1	Consistency Criteria	46
4.1.1	Discrete Interactive Applications	46
4.1.2	Continuous Interactive Applications	50
4.2	Design Considerations: Hard State vs. Soft State	52
4.3	Related Work	53
4.4	Local Lag	57
4.5	Timewarp	60
4.5.1	The Basic Timewarp Algorithm	61
4.5.2	Round-Based Timewarp	62
4.5.3	Filtered Timewarp for Discrete Applications	62
4.5.4	Restricting the Size of the Operation History	64
4.5.5	Timewarp as a Generic Service	66
4.6	Requesting Full States	66
4.7	Experimental Results	68
4.7.1	Results for the mlb	68
4.7.2	Results for Instant Collaboration	70
4.7.3	Results for the Spaceshooter Game	71
4.8	Undo of Operations	72
4.8.1	Formalization of Undo and Redo	73
4.8.2	Design Considerations for Undo Algorithms	73
4.8.3	Related Work	75
4.8.4	Undo of Operations for the mlb	76
4.9	Conclusions	79
5	Visualization of Conflicting Operations	81
5.1	Intention Preservation	82
5.2	Semantic Conflicts in Operations	83
5.3	Design Considerations	84
5.4	Related Work	86
5.5	Visualization of Conflicting Operations	87
5.5.1	Representing the Operation History	89

5.5.2	Exploring Past States	89
5.5.3	Exploring Alternative States and Changing the Current State	91
5.5.4	Summary of Visualization Techniques	92
5.5.5	Lessons Learned and Discussion	92
5.6	Conclusions	93
6	Support for Late-Joining Session Members	95
6.1	Design Space of Late-Join Algorithms	96
6.1.1	Design Criteria	96
6.1.2	The Source of Late-Join Data	97
6.1.3	The Extent of Late-Join Data	98
6.1.4	Distribution of Late-Join Data	100
6.2	Related Work	100
6.3	The Late-Join Algorithm	103
6.3.1	Selection of Application Instances	103
6.3.2	Late-Join Policies	104
6.3.3	Distribution of Late-Join Data	107
6.4	Consistency Control	110
6.4.1	Initialization by Replay of the Operation History	112
6.4.2	Initialization by State Transmission	112
6.4.3	Initialization by State and Operation Sequence	114
6.5	Simulation Results	114
6.5.1	Distribution of Late-Join Data	116
6.5.2	Late-Join Policies	122
6.5.3	Composition of the Late-Join Server Group	124
6.6	Using the Late-Join Algorithm in Sample Applications	126
6.7	Support for Late-Joiners in Instant Collaboration	127
6.7.1	Late-Join Algorithm	128
6.7.2	Simulation Results	130
6.8	Conclusions	132
7	RTP/I - An Application-Level Protocol for Distributed Interactive Applications	135
7.1	Related Work	136
7.2	Design Considerations for a Protocol Framework	138
7.2.1	Architecture and ADU-Related Information	138
7.2.2	Session-Related Information	141
7.3	The RTP/I Protocol	142
7.3.1	The RTP/I Data Transfer Protocol	143
7.3.2	The RTP/I Control Protocol (RTCP/I)	147
7.4	Generic Services	149
7.4.1	Generic Consistency Control Service	150
7.4.2	Generic Late-Join Service	151

7.4.3	Generic Recording Service	152
7.5	Transport of Application Data with RTP/I	154
7.5.1	The RTP/I Protocol Library	154
7.5.2	Payload Type Definition for Shared Whiteboards	155
7.6	Conclusions	157
8	Application-Level Multicast for Distributed Interactive Applications	159
8.1	Group Communication	160
8.1.1	IP Multicast	160
8.1.2	Application-Level Multicast	164
8.2	ALM Routing for Distributed Interactive Applications	165
8.3	Distribution Trees and Metrics	166
8.3.1	Tree Height and Fanout	168
8.3.2	Resource Usage and Minimum Spanning Tree	168
8.3.3	End-to-End Delays and Shortest Path Tree	169
8.4	Related Work	170
8.5	Priority-Based Application-Level Multicast Routing	172
8.5.1	Introducing Application-Level Semantics	173
8.5.2	The Priority-Based MST Algorithm	176
8.5.3	Pseudo-Code for the PST Algorithm	177
8.5.4	Considering Constraints for Distribution Trees	179
8.6	Simulation Results	180
8.6.1	Simulation Setup	180
8.6.2	Simulation Results for Six End-Systems	181
8.6.3	Simulation Results for Eighteen End-Systems	183
8.6.4	Introducing Uncertainty	185
8.6.5	Comparison of Simulation Results	186
8.7	The PST Protocol	187
8.7.1	Application Interface	188
8.7.2	Basic Protocol Functionality	189
8.7.3	Efficient Topology Distribution	191
8.7.4	Maintenance of the Distribution Tree	193
8.7.5	Experimental Results	196
8.7.6	PSTP and RTP/I	197
8.8	Conclusions	199
9	Conclusions and Future Work	201
9.1	Conclusions	201
9.2	Scientific Contributions	204
9.3	Future Work	205
	Bibliography	207

List of Figures

2.1	TeCo3D user interface	19
2.2	Spaceshooter user interface	20
2.3	Instant Collaboration user interface	21
3.1	mlb user interface	26
3.2	Thumbnail overview	28
3.3	Class diagram for mlb objects	29
3.4	Protocol stack	32
3.5	Presentation animation	34
3.6	Participant information window	35
3.7	Indicating the origin of an operation	35
3.8	Awareness information	35
3.9	Telepointer and telebox	36
3.10	Hand-raising tool	37
3.11	Chat window	38
3.12	Feedback tool	39
3.13	Application launch tool	40
3.14	Pocket mlb user interface	41
4.1	Network delay results in different operation orders	46
4.2	Relations among operations	47
4.3	Equalization of the operation delay	58
4.4	The basic timewarp algorithm	61
4.5	Decision algorithm for timewarp with filtering	63
4.6	Conflicting operations	64
4.7	Overwriting operations	64
4.8	mlb experiment	69
4.9	Size of the operation history	71
5.1	Concurrent operations O_1 and O_2 trigger a timewarp at site 1	82
5.2	mlb with visualized operation history	88
5.3	Visualization of the operation history	89
5.4	Exploring alternative states	91

6.1	Different types of late-join data	99
6.2	Event-triggered late-join	106
6.3	Inconsistency in a late-join situation	111
6.4	Simulation results for the shared whiteboard scenario	118
6.5	Simulation results for the online game scenario	121
6.6	Simulation results for different late-join policies	123
6.7	Simulation results for different minimum server group sizes	125
6.8	Simulation results for interleaved synchronous and asynchronous collaboration	131
7.1	Structure of RTP/I ADUs	143
7.2	RTP/I packet header	144
7.3	RTP/I state ADU header	145
7.4	RTP/I state request ADU	147
7.5	RTCP/I source description ADU	148
7.6	RTCP/I object report ADU	149
7.7	Consistency control library API	150
7.8	Late-join library API	152
7.9	Random access for discrete and continuous applications	154
7.10	RTP/I library API	155
7.11	Rectangle state ADU	156
7.12	Move event or cue ADU	157
8.1	Group communication	161
8.2	Distribution trees	167
8.3	Joint path to distant receivers	173
8.4	Example graph	175
8.5	Optimal distribution trees	175
8.6	Approximated path	176
8.7	Modified edge weights	177
8.8	Pseudo-code for the computation of the PST	178
8.9	Priority-based distribution trees	179
8.10	Distribution of application priorities	181
8.11	Simulation results for 6 end-systems	182
8.12	Distribution of application priorities	183
8.13	Simulation results for 18 end-systems	184
8.14	Simulation results for scenario with uncertain delays	186
8.15	PST protocol library API	188
8.16	PSTP packet for types join, leave, and ping	190
8.17	PSTP pong packet	191
8.18	PSTP data packet	192
8.19	PSTP delay packet	193
8.20	Experimental results for RDP	196

List of Tables

4.1	Timewarp performance	70
8.1	Matrix of measured end-to-end delays [ms]	195
8.2	PST fanout distribution	196

Abbreviations

ACK	ACKnowledgement
ADSL	Asymmetric Digital Subscriber Line
ADU	Application Data Unit
ALF	Application-Level Framing
ALM	Application-Level Multicast
ALMI	Application-Level Multicast Infrastructure protocol
AOF	Authoring On the Fly
ARQ	Automatic Repeat reQuest
BGMP	Border Gateway Multicast Protocol
BGP	Border Gateway Protocol
BTP	Banana Tree Protocol
CAN	Content Addressable Network
CBT	Computer-Based Training
CBT	Core-Based Tree
CName	Canonical Name
CSMT	Constrained Steiner Minimum Tree
DHCP	Dynamic Host Configuration Protocol
DiffServ	Differentiated Services
dlb	digital lecture board
DSL	Digital Subscriber Line
DVMRP	Distance Vector Multicast Routing Protocol
FEC	Forward Error Correction
GPS	Global Positioning System
GT-ITM	Georgia Tech Internetwork Topology Models
HMTD	Host Multicast Tree Protocol
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol

ICMP	Internet Control Message Protocol
IGMP	Internet Group Management Protocol
ILP	Integrated Layer Processing
IMoD	Interactive Media on Demand
IntServ	Integrated Services
IP	Internet Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
IRI	Interactive Remote Instruction
ISDN	Integrated Services Digital Network
ISP	Internet Service Provider
LAN	Local Area Network
MAC	Medium Access Control
MACS	Modular Advanced Collaboration System
MALLOC	Multicast Address Allocation Architecture
MBone	Multicast Backbone
mlb	multimedia lecture board
MOSPF	Multicast Open Shortest Path First
MPEG	Moving Picture Experts Group
MSDP	Multicast Source Discovery Protocol
MST	Minimum Spanning Tree
MTU	Maximum Transmission Unit
NACK	Negative ACKnowledgement
NAT	Network Address Translation
NSTP	Notification Service Transport Protocol
NTE	Network Text Editor
NTP	Network Time Protocol
OBJID	OBJect Identifier
OpenPGP	Open Pretty Good Privacy
OSPF	Open Shortest Path First
PC	Personal Computer
PGP	Pretty Good Privacy
PID	Participant Identifier
PIM	Protocol Independent Multicast
PIM-DM	Protocol Independent Multicast Dense Mode

PIM-SM	Protocol Independent Multicast Sparse Mode
PST	Priority-based minimum Spanning Tree
PSTP	Priority-based minimum Spanning Tree Protocol
QoS	Quality of Service
RDP	Relative Delay Penalty
RIP	Routing Information Protocol
RMF	Reliable Multicast Framework
RMFP	Reliable Multicast Framing Protocol
RPC	Remote Procedure Call
RSVP	Resource ReserVation Protocol
RTCP	RTP Control Protocol
RTP	Real-time Transport Protocol
RTCP/I	RTP/I Control Protocol
RTP/I	Real-Time Protocol for distributed Interactive applications
SAP	Session Announcement Protocol
SDR	Session DiRectory
SMP	Scalable Multicast Protocol
SMTP	Simple Mail Transfer Protocol
SOAP	Simple Object Access Protocol
SPT	Shortest Path Tree
SRM	Scalable Reliable Multicast
SSM	Source-Specific Multicast
SV	State Vector
TAG	Topology Aware Grouping
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VRML	Virtual Reality Modeling Language
wb	whiteboard
WILD	Wireless Interactive Learning Devices
WYSIWIS	What You See Is What I See
WWW	World Wide Web
XML	eXtensible Markup Language

Chapter 1

Introduction

Computer technology in general and the Internet in particular have a great impact on how people retrieve information and communicate. They make us independent from space and, if desired, also of time: As long as communication partners have access to a computer with an Internet connection, their locations do not matter, and they may either share information in real-time (synchronously) or delayed (asynchronously). Aside from the ubiquitous tools of email for asynchronous communication and instant messaging for synchronous communication, new technologies have emerged in recent years offering rich functionality and multimedia content. Prominent examples are computer-supported video conferencing systems for private and business meetings, telemedicine and teleteaching, groupware applications for the joint editing of documents or program code, peer-to-peer file sharing, distributed virtual environments, and even multiplayer computer games.

In many video conferencing scenarios, so-called *shared whiteboards* are employed that allow the users to present and edit slides and other documents. For instance, in a teleteaching session where a lecture is transmitted via the Internet to a group of students, the shared whiteboard is a substitute for the traditional blackboard and visualizes presentation slides and annotations to all students. Together with the audio and video of the teacher, a shared whiteboard helps to achieve a good approximation of regular face-to-face lectures. The design of a shared whiteboard, the multimedia lecture board (mlb), is one major topic of this thesis.

The mlb and most of the other applications listed above share important properties. For one, they all connect a set of spatially separated users via a computer network, and they allow the users to access and manipulate the same application data. For instance, in an mlb presentation all users see the same document and may also change it (e.g., by a written annotation). In the ideal case, such a change becomes immediately visible to all users. For this purpose, continuous updates need to be distributed from their point of origin over the network to the

individual end-systems of the other users. Because the application data is distributed and the users may interact with it, we denote such applications as *distributed interactive applications*.

The design of such a distributed interactive application needs to consider aspects from a number of disciplines and poses various challenges: Since components of the application reside at different locations (e.g., the end-systems of the users), the algorithms employed are distributed, and data needs to be synchronized among all locations. The exchange of data is to be handled by appropriate network protocols that meet the application's requirements (e.g., with respect to the propagation delay). In many cases, different media such as video and documents are used together, and their joint handling is in the scope of multimedia technology. Since multiple users collaborate, distributed interactive applications usually have a more complex architecture than a comparable single-user application. Thus, the design of the human-computer interface is an important issue. Finally, the ultimate goal for distributed interactive applications is to facilitate human-human interaction. Thus, findings from humanities such as psychology and sociology have to be incorporated in addition to technical issues. For instance, distributed interactive applications have to explicitly rebuild social protocols that are naturally given in face-to-face situations, such as eye contact and gestures (e.g., hand-raising).

The main focus of this thesis is on synchronization algorithms and network protocols for distributed interactive applications, but we also keep the other factors mentioned above in mind. While the shared whiteboard *mlb* is a starting point for many of our considerations, we seek to find solutions that are applicable to other distributed interactive applications as well.

1.1 Problem Statement and Outline

This thesis covers various aspects that are vital for distributed interactive applications and that range from the design of network protocols over distributed algorithms to human-computer interaction. We believe that it is important to keep the dependencies between these different aspects in mind when designing an algorithm for a certain problem. For instance, some distributed interactive applications require the user to obtain appropriate rights before application data can be accessed or modified. While this simplifies several technical issues, it also prevents spontaneous and effortless collaboration. Another major goal of this thesis is to find solutions not only for the multimedia lecture board but for the whole class of distributed interactive applications. Thus, we repeatedly use two other applications as examples, each with its own challenges: Instant Collaboration, which supports long-term sessions where not all users have to be online at the same time, and a multi-player game where the application state is not only updated by user actions but also by time-controlled changes. In addition to these “real” applications, important results are validated by means of simulation studies.

For the structure of this thesis, a top-down approach was adopted that begins at the application-level, then covers synchronization algorithms, and finally discusses communication protocols.

In Chapter 2, a general introduction to distributed interactive applications is given. After discussing the advantages and disadvantages of different architectures, a formal data model for distributed interactive applications is defined. This data model captures the main characteristics of such applications and allows to address common problems independent from specific applications. Following, the main challenges when designing a distributed interactive application are identified. Finally, three sample applications are presented.

Chapter 3 describes the multimedia lecture board (mlb) that was developed in the course of this thesis and that was the origin of many issues discussed later on. Starting from the user interface, the most important features of the mlb are presented. Aside from the functionality to edit and present documents in video conferencing scenarios with multiple users, the mlb also offers mechanisms to coordinate session members and provides them with information about each other's actions. The mlb can also be used on small handheld devices, which opens novel usage scenarios in face-to-face meetings. The chapter concludes with software engineering aspects.

Distributed interactive applications allow multiple users to manipulate the same application data. For instance, all participants of an mlb session have access to the current presentation slide and may also annotate it. But in fact, each user sees a local copy of that data that is held by the corresponding end-system. Thus, the application needs to ensure the synchronization of all these copies so that all users have the same view. In Chapter 4, consistency control algorithms are discussed that accomplish this task. After defining formal consistency criteria and analyzing existing approaches, a novel and generic consistency control service is presented. Using the examples of the mlb, Instant Collaboration, and the network game, it is demonstrated that this service is well-suited for many different applications. Moreover, a method to undo user actions without endangering the application's consistency is proposed.

But keeping the application data consistent is not only a technical challenge. The users also need to understand how the current state of the application data came to be, or, more specifically, who changed when which parts of the data by which actions and for what purpose. This is especially important in situations where several users modify data within a short period of time or when some actions conflict with each other, and the system is unable to resolve the conflict automatically. In Chapter 5, possibilities to visualize information are investigated that support the user in answering these questions. Following, a powerful visualization technique based on the history of all actions is proposed. A prototype of this visualization technique is integrated into the mlb.

Distributed interactive applications often allow users to join or leave a session at any time. But in case a participant joins an ongoing session, he has missed all user actions that took place since the beginning. Thus, another important synchronization algorithm is necessary to initialize the application instances of late-joining users. For example, a late-comer would need at least the slide currently presented in the mlb so that he is able to participate in the session. Late-join algorithms that provide this vital initialization data are discussed in Chapter 6. Even though the late-join problem is critical, most existing applications integrate only basic late-join mechanisms that might result in high application and network loads. Especially for applications like Instant Collaboration, where users may also modify data while they are disconnected from the network, a carefully designed late-join algorithm is required since late-joins happen frequently, and initialization data cannot always be sent. As is shown by means of extensive simulation studies, the late-join algorithms presented in this thesis meet these challenges and reduce the application and network loads significantly. As in the case of consistency control, the late-join mechanism is realized as a generic service that can be adapted to the specific needs of an application.

The design of generic algorithms that are valid for all distributed interactive applications is facilitated by the common data model of these applications. If the information of that model is exposed in such a way that it can be accessed from outside the application and without any application-specific knowledge, it is also possible to implement these algorithms in the form of generic services that can be reused by all applications. A standardized network protocol is well-suited for this task, and in Chapter 7, the standardized application-level protocol RTP/I is introduced for this purpose. As a consequence, complex functionality needs to be designed, implemented, and verified only once. After describing the protocol functionality of RTP/I, it is demonstrated how RTP/I can be used to realize the generic services of consistency control and late-join, and how the communication model of the mlb is based on RTP/I.

A prerequisite for keeping the local copies of the application data synchronized, all modifications need to be distributed from their originating end-system to the sites of all other users by some form of group communication (multicast). In Chapter 8, a novel multicast routing algorithm is proposed that constructs an efficient, tree-shaped overlay network by connecting all sites at the application level. Unlike other application-level multicast routing algorithms, it allows the application to influence the shape of the distribution tree so that the propagation delay of important messages is minimized. After analyzing the properties of this routing algorithm in simulated scenarios, we present an operational application-level multicast protocol that can be employed together with RTP/I and that is validated in Internet experiments.

In Chapter 9, the main results and contributions of this thesis are summarized and possible areas of future research are outlined.

Chapter 2

Distributed Interactive Applications

Distributed interactive applications allow a group of users connected via a computer network to collaborate in order to accomplish a common task [59, 14]. They facilitate both human-human and human-computer interaction in a wide spectrum of scenarios: The users may be located in the same room or be distributed spatially. And the users may interact at the same time (i.e., synchronously) or collaborate asynchronously at different times [130, 95]. The variety of distributed interactive applications in this spectrum is large and includes such different applications as instant messaging, video conferencing and meeting applications, teleteaching, document sharing and editing, software development, virtual environments, and network computer games. But all these applications have in common that users access and manipulate the same data within a so-called *shared workspace*.

The design of distributed interactive applications is influenced by many research areas [93]: First, such applications belong to the class of distributed systems that generally include all systems where (parts of) the functionality is located on different computers. The message exchange among those parts is managed via a computer network so that the design of network protocols is an important aspect. In many cases, interactive applications integrate different media such as audio, video, and documents, and the handling of such media is in the scope of multimedia technology. Since multiple users are able to access the shared workspace, an appropriate user interface and human-computer interaction paradigm are especially important so that interaction patterns are captured by the application and users are able to interpret the actions of remote participants. Finally, the human-human collaboration, which is the main goal of a distributed interactive application, needs to be supported by findings from sociology, organizational theory, and work flow design. For instance, an application that supports the process of decision finding in group meetings should be aware of the individual roles of the users [59]. In the remainder of this thesis, we will focus on distributed systems and networking aspects but keep the other factors in mind.

First, fundamental choices regarding the architecture of a distributed interactive application are discussed. In Section 2.2, a formalized data model is introduced that captures the common aspects of distributed interactive applications. This data model allows an application-independent discussion of technical challenges and their solutions in Section 2.3. Following, important sample applications are presented in Section 2.4. The general discussion of distributed interactive applications is then concluded in Section 2.5.

2.1 Architecture of Distributed Interactive Applications: Centralized vs. Replicated

The architecture of a distributed interactive application can be varied between the two extremes of a centralized architecture on the one hand and a replicated architecture on the other hand [162]. A *centralized architecture* concentrates the main functionality of an application at a single server process. This means that the server is responsible for maintaining the shared workspace and also for coordinating all accesses to the application data. In case a user wants to change an object of the shared workspace, his client software sends an appropriate request to the server and receives the corresponding update as a reply. Thus, the task of a client process is merely to provide an interface to visualize the shared workspace and to accept actions of the local user. The main advantages of a centralized approach are that it simplifies the coordination among the participating users and that it relieves the client processes from the execution of complicated algorithms. However, it also has several drawbacks: First, the server itself might become a bottleneck in case the tasks consume a high amount of resources such as processing time and memory space. Second, in case the server fails, the application is terminated for all users, and data might be lost. Last, the server itself needs to be set up and administrated so that it is available. This might prevent a spontaneous usage of the application [83].

The other extreme are *replicated architectures* [39], which are also known as peer-to-peer approach. Here, each user runs an identical instance of the distributed interactive application. Thus, both the complete functionality of the application and the data within the shared workspace are replicated at the each user's site, and the coordination of the users' activities has to be accomplished by the individual application instances. A local user action updating the shared workspace needs to be propagated to all remote instances so that these can synchronize their local state copies. The main advantages of the replicated architecture are that it is very robust against the failure of single instances and that the required resources can be provided by many sites. Moreover, it does not depend on a certain infrastructure, and since data is stored locally, it can be accessed without network connection. Also, the user experiences very short response times since most actions can be executed immediately on local

data. But at the same time, the coordination of all application instances and the synchronization of the individual user actions is rather complex. A replicated architecture also requires more resources at the end-users' sites when compared to the centralized approach.

Between the two extremes of centralized and fully replicated architectures, a distributed interactive application might adopt a hybrid approach combining properties from both possibilities. For example, the application data might be replicated in order to increase the system's robustness but changes of shared objects are coordinated by a specific instance in order to avoid concurrent access. Or some application functionality might be provided by a server in order to save resources (e.g., for storing certain data). Most functionality presented in this thesis is realized in a replicated fashion but in some cases we fall back on a client-server architecture (e.g., see Section 7.4.3).

2.2 Formalization of the Data Model

Even though the variety of distributed interactive applications is rather large, they share a basic data model that is valid for both centralized and replicated architectures. This data model was first described by Mauve [156] and allows us to discuss aspects that need to be addressed when developing a distributed interactive application independent from the application itself.

Distributed interactive applications have a *state*, which includes the values of all attributes needed to encode the data within the shared workspace at a certain point in time. For instance, the state of a shared whiteboard contains the set of all slides together with the graphical objects present on the individual slides. This application state is not constant but might change for two reasons: First, the state might change with the *passage of time*. State changes due to the passage of time are deterministic and can be calculated on the basis of the current state of an object and the physical laws that are valid within the shared workspace. For instance, when an animated object moves across the screen, the state of that object includes its current position, speed, and direction. Together with parameters such as friction and gravitation, the object's future positions can then be determined independently by each application instance. Thus, time-related state changes usually do not need to be propagated to the participating sites.

Second, the state of an interactive application might be modified by *events*. Events are triggered either by user actions (e.g., a user changes an object's direction) or by other non-deterministic influences (e.g., by sensor input or computer-controlled agents). Since an event is non-deterministic, the application state needs to be updated explicitly and has to be propagated to all application instances. We denote applications that allow only state changes due to user actions as *discrete*, whereas applications supporting both types are called *continuous*.

Examples of discrete applications are shared whiteboards, instant messengers, and shared text editors. In the continuous domain, events have a scheduled execution time at which they need to be applied to the application's state [6] since the effect of an event depends on the point in time it is executed. For instance, the new path of an animated, moving object depends on the point in time a "change direction" event is executed. Distributed virtual environments and network games are examples of continuous interactive applications.

For easier handling, the application state may be partitioned into several independent objects. In this case, *identifiers* are required to reference single objects. These identifiers need to be unique during the lifetime of a shared workspace (see Sections 3.3.1 and 7.2.1). Partitioning the state allows an application instance to maintain only the state of those objects the local user is interested in. For instance, consider a large virtual environment where a user is located in a certain room. Then the user's application instance can save a considerable amount of resources when keeping track only of those objects that are actually visible for the user [156, 173]. We denote objects visible to the local user as *active*, and all others as *passive*.

All sites accessing the shared workspace of a distributed interactive application exchange messages via a computer network. These messages may either be status updates or informal notifications, and they can be classified as four different types: (1) A *state* includes all information necessary to (re-)create a certain object (e.g., the color, position, speed, and direction of an animated object). (2) An *event* encodes the change of single object attributes and can be interpreted only on the basis of a valid state. (3) A *delta state* accumulates the state updates of several events and can also be decoded on the basis of a certain state only. (4) Informal notifications that do not alter the shared state of the application are transported as *cues*. Like events, cues may depend on a certain state. Examples of cues are temporary visualizations of user actions such as invoking a menu, transient mouse movements, and indications when a user is idle for some time.

We denote states, events, and delta states as *operations* since they update the state. Which operation type is used to announce a state change is application-dependent: An event is usually more efficient, but the redundant information contained in states and delta states increases the robustness of the message exchange. In the continuous domain, operations and cues are valid at a certain point in time only.

Events cause either relative or absolute state changes. The result of a relative event depends on the former value of the modified attribute (e.g., when an object is moved to the left by ten pixels). In contrast, absolute events replace the former value of the attribute changed (e.g., when an object is moved to position (5, 3)). The advantage of absolute events is that they are more robust and easier to handle whereas relative events introduce dependencies into the data

stream that need to be considered by the application. But in some cases, relative operations might be required semantically (e.g., for shared text editors [59, 238]).

Starting from its initial state, the content of a shared workspace is changed by operations and possibly by the passage of time. All operations issued during a session form a *history*. As we will see in later chapters, this operation history is important for numerous functions such as consistency control, undo, and session recording.

2.3 Design Considerations

The realization of a distributed interactive application is challenging and considerably more complex than an equivalent single-user application [59, 96]. For instance, operations need to be transmitted with appropriate network protocols, the actions of many users have to be coordinated, and replicated data should be in the same state at all application instances. Besides those technical aspects, human behavior has to be considered so that the application allows “natural” human-human interaction. Finding a trade-off between flexible usage of the application, sufficient support for achieving the common task, and a user interface that is easy to use is especially demanding. In the following, we discuss these design considerations in more detail.

2.3.1 Collaboration Management and User Interface

As categorized by Fluckiger, a multi-user application can be either *collaboration-aware* or *collaboration-unaware* [69]. In the latter case, the application itself is not designed for multiple users and lacks the specific functionality of a distributed interactive application. Typical examples are single-user applications that are extended to multi-user scenarios by an application-sharing system [14]. Such systems have a centralized architecture. With application-sharing, the user interface of a regular single-user application is exported from the node it runs on to the sites of all participants. The actions of remote users are captured by the system and delivered to the executing site. Even though remote users are able to change the application’s state, all user actions need to be serialized by means of a turn-taking protocol [14]. The main advantage of application-sharing is that existing and well-known applications can be reused. But at the same time, a single-user application obviously does not support collaboration but relies on the application-sharing system. Moreover, not all scenarios can be realized with application-sharing. For instance, virtual environments or multi-player games usually include representations (“avatars”) of all users that cannot be reproduced when sharing a single-user application.

Thus, we concentrate on collaboration-aware applications specifically designed for multi-user scenarios. Here, the application provides functionality to support and coordinate the interaction among all participants of a session. This functionality is vital since the users are physically separated, and the social protocols that usually regulate the collaboration in face-to-face situations such as eye contact, gestures, and facial expressions have to be reproduced explicitly [69].

2.3.1.1 Session Control and Floor Control

Session and floor control establish social protocols in distributed interactive applications. The task of a *session control* is to manage the group of users and their individual roles [49]. The group may either be open to anyone or allow admission by invitation only. In both cases, the composition of a group may be dynamic so that users are allowed to join and leave a session anytime. As found by Ellis et al., a role defines the behavior of a user and often includes certain rights and duties [59]. For instance, consider a teleteaching scenario where a lecturer is giving a presentation. Then the lecturer determines the content of the shared workspace and assigns access rights to his students, i.e., rights to read or change shared objects. Such access rights to certain resources of the application are managed by a *floor control* mechanism. The floor control also defines how access rights are granted, who is allowed to request certain rights, and who grants them.

Depending on the usage scenario, an application might establish strict interaction rules that are managed by session and floor control. This approach is common in workflow applications capturing business processes in a company [14]. Alternatively, the application can provide collaborative tools without enforcing that they are used in a certain way so that the interaction is coordinated by the users rather than the application. This approach is more flexible, but requires coordination through social protocols among the users and might therefore be less efficient. An example is a multi-point video conference.

2.3.1.2 WYSIWIS

The user interface of a distributed interactive application provides access to the objects within the shared workspace. The basic design principle for such a user interface is *What You See Is What I See* (WYSIWIS) described by Stefik et al. [231]: WYSIWIS means that the information of the shared application state is presented to all users in the same way. Strict WYSIWIS requires that all users have an identical view of the shared workspace, i.e., each application instance displays the same information in the same way. While strict WYSIWIS facilitates a joint understanding of the application's state, it also prevents users from exploring and changing data independent from others [14]. In contrast, relaxed WYSIWIS allows users to have an individual view of the application state and also permits private information within the

shared workspace, which is visible only to some participants such as the private annotations of a student in a teleteaching scenario [78].

2.3.1.3 Awareness

Besides visualizing the actual content of the shared workspace, the application should also display information about the individual participants and indicate who is currently present in a session and who is responsible for which actions. For instance, the Modular Advanced Collaboration System (MACS) represents every user with a thumbnail picture and visualizes a floor request with an icon next to the requesting participant [18]. Such information is necessary to create an *awareness* for the status and the actions of remote participants: Awareness allows a user to understand the activities of others and to adapt his own actions in order to accomplish a common task [51, 4]. Establishing a certain degree of awareness is vital for distributed interactive applications [101], especially when they are designed for large groups, allow multiple users to modify the shared state at the same time, and lack strict interaction rules.

Awareness information can either be generated explicitly (e.g., when the users vote to rate a solution) or can be derived from operations (e.g., when a participant is marked as active after issuing events) [51]. Ellis et al. identify *telepointers* as a powerful tool for explicitly creating awareness by visualizing the local mouse movements to remote participants [59] (also see Section 3.5.2). When integrating awareness information into the application, it has to be considered that this might violate the privacy of users [120] and that too many notifications might be distracting [99].

In addition to the shared workspace, many collaborative systems employ separate audio and video communication channels. Even though an audio channel can be used only by one person at a time, and the utility of video significantly declines with an increasing number of participants, they both are important tools to create a social presence.

2.3.1.4 Responsiveness

Another aspect that has a substantial effect on the usability of a distributed interactive application and that needs to be considered is the *responsiveness* of the application [223]: Responsiveness generally describes how fast user actions are absorbed by the shared workspace and displayed to all participants. It has two elements [59]: (1) The *response time* denotes the time span until the user who issued an operation can see its effect, and (2) the *notification time* denotes the time until all remote users can see a state update. They are determined by the architecture (e.g., centralized architectures tend to have higher response times), interaction rules (e.g., requesting access rights increases the response time), and communication model

(e.g., high network delays result in large notification times). Low response and notification times (i.e., high responsiveness) are important for a natural behavior of the application and for a smooth interaction.

2.3.2 Synchronization Algorithms

The task of *synchronization algorithms* is to ensure that all local copies of the application's state that are managed by the individual application instances are identical. This is a major requirement for the collaboration of multiple users [14, 59, 238] and is also expressed by the WYSIWIS principle [231]. In this thesis, we cover synchronization algorithms for consistency control and for the handling of late-join situations.

In case the application has a replicated architecture, a so-called *consistency control* mechanism is required to ensure that all application instances reach the same state after a sequence of operations has been issued. Critical situations may arise when users are allowed to modify the same object at the same time. For instance, two participants might change an object's color simultaneously. In this case, the consistency control algorithm has to select the operation that determines the new color of the object and also has to make sure that all sites eventually reach the same state.

The consistency control mechanism has a strong influence on the usability of the application. While concurrent operations can be prevented by a floor control mechanism that grants exclusive access rights [2], this also restricts the users in their possibilities to collaborate efficiently. In a multi-player game, users might need to manipulate the same object at the same time in order to accomplish a task, making floor control unsuitable. And a centralized algorithm might result in a low responsiveness [59]. In case the response time differs to a large extent from the notification time, participants observe different versions of the same object for a certain time span, which might result in inappropriate operations. These consistency issues and appropriate consistency control algorithms are discussed in Chapter 4.

A second synchronization algorithm is required to handle *late-join* situations where a new participant joins an ongoing session. In case the initial state of a late-joining application instance differs from the current shared state, it needs to be initialized by an appropriate late-join algorithm. Since the current state might be large, a late-join algorithm should minimize the consumption of network and end-system resources. Moreover, the late-joining user should perceive only a small delay until he is able to participate actively in the session. In Chapter 6, an efficient late-join algorithm is presented that meets these demands.

2.3.3 Communication Model

Distributed interactive applications require messages to be exchanged among all participating sites: Actions changing the content of the shared workspace need to be propagated as operations in order to keep replicated data consistent. Moreover, awareness information needs to be transmitted in order to support the collaboration process. Thus, the design of the communication functionality is one of the most important issues for distributed interactive applications [59].

2.3.3.1 The Internet

The message exchange among the instances of a distributed interactive application is handled over a computer network such as the Internet. Basically, the Internet connects independent Local Area Networks (LANs) by a world-wide hierarchy of intermediate nodes [245]. End-systems holding the instances of a distributed interactive application are each located in a certain LAN. A LAN provides a broadcast medium where a message is received by all nodes that are member of the same LAN. But if a message needs to be delivered to an end-system that resides in a different LAN, it is forwarded explicitly by the intermediate nodes of the Internet: Each intermediate node is connected to some other nodes and decides upon reception of the packet about the next link over which the packet should be sent until the destination is reached. This forwarding process is also known as *routing* [122], the intermediate nodes are denoted as *routers*, and the path of a message from its sender to the receiver(s) is determined by a *routing protocol*. In the Internet, the message exchange between end-systems is managed by the Internet Protocol (IP) [197] whereas the routing paths can be established by different routing protocols such as Open Shortest Path First (OSPF) [175] and the Routing Information Protocol (RIP) [152]. Data delivery with IP is best-effort, which means that the network seeks to optimize criteria such as the transmission delay and the network load but does not give any delivery guarantees. Thus, applications need to anticipate exceptional situations such as higher-than-average notification times.

2.3.3.2 Group Communication

The most common form of communication in the Internet is *unicast*, or point-to-point, where a message is transmitted from the sender to a single receiver. IP is responsible for the delivery of data between two end-systems. On the basis of IP, there exist two unicast transport protocols realizing the message exchange between two application instances that are running on these end-systems: The User Datagram Protocol (UDP) [196] provides no additional services aside from the process-level communication link. With UDP, every packet is forwarded separately, without a relationship to other packets. In contrast, the Transmission Control Pro-

protocol (TCP) [198] is connection-oriented and offers additional functionality such as flow and congestion control, and the reliable transport of messages.

Distributed interactive applications usually require that data is delivered to multiple destinations, i.e., in a replicated architecture a state update has to be forwarded to all participating sites even if they are distributed over different LANs. This communication form is also known as *multicast* or *group communication* [273]. In the Internet, there exist two possibilities to realize group communication: IP multicast and application-level multicast. The multicast version of the IP protocol provides efficient group communication where the source sends each packet only once, and the distribution of a packet to multiple destinations is handled by the routers alone, i.e., packets are duplicated within the routers where necessary [41]. UDP is the only transport protocol supporting IP multicast.

Alternatively, group communication can be realized with application-level multicast [30]. Here, the end-systems are responsible for the distribution of data and explicitly send a packet to multiple destinations via separate unicast connections, i.e., the end-systems build a tree-shaped overlay network and duplicate the packets where necessary. Application-level multicast can be implemented using either UDP or TCP as transport protocols. In Chapter 8, IP and application-level multicast are discussed in more detail, and a novel application-level multicast protocol for distributed interactive applications is introduced.

2.3.3.3 Reliable Delivery of Data

Many distributed interactive applications require that data be transported reliably in order to keep replicated data consistent and to allow seamless collaboration. In the Internet, data may be lost for several reasons: First, a packet might be corrupted so that the received sequence of bits differs from the original sequence, and the receiver has to discard it. Moreover, physical links might fail completely or routes may change while a packet is in transfer. But in most cases, data is lost when routers receive more packets than they are able to handle and drop these overflowing packets.

While TCP implements a mechanism to repair packet loss for unicast connections, there is no equivalent reliable transport protocol for multicast. Instead, the application has to integrate appropriate reliability mechanisms, irrespective whether it uses IP or application-level multicast¹. In the past decade, many different mechanisms have emerged, and it is widely believed that there are too many application scenarios for a single algorithm to fit all cases [67, 147].

¹In the case of application-level multicast, individual links between two end-systems might be protected with TCP. But since inner nodes of the tree might fail or get overloaded, a reliability mechanism is nevertheless necessary.

The first class of reliability mechanisms retransmit lost packets and are also known as Automatic Repeat reQuest (ARQ) approaches. They can be either sender-based or receiver-based. In *sender-based* approaches, the sender ensures that data is successfully delivered to all receivers: Each data packet has to be acknowledged by all receivers. In case an acknowledgment (ACK) is still missing after a certain period of time, the packet is retransmitted. Thus, the sender might have to process a high number of ACKs, resulting in a so-called ACK-implosion. Additionally, receivers can trigger a retransmission explicitly by sending a negative acknowledgment (NACK). But for both ACKs and NACKs, the sender has to keep track of each receiver so that the sender-based approach does not scale well with the number of receivers. For this reason, *receiver-based* approaches reverse the responsibility for repairing packet loss to the receivers that request retransmissions by NACKs. When many receivers detect a missing packet, this might lead to a NACK-implosion. The Scalable Reliable Multicast (SRM) protocol developed by Floyd et al. seeks to prevent NACK-implosions with a timer-based suppression algorithm [68]: Before sending a NACK to the multicast group, it is delayed for a certain time span T . In case another NACK for the same packet is received during T , the own NACK is not sent. While NACKs and NACK suppression reduce the load for the original sender of a packet, the source is still burdened when a packet is lost.

Hierarchical approaches seek to spread the responsibility for processing ACKs and NACKs and for retransmitting packets from the original source to several members of the multicast group [145]. For instance, in SRM all sites that have received the requested data can answer a NACK, and SRM preferably selects a site with a low network delay to the NACK's sender, reducing the repair time [68]. Besides this locally-scoped multicast, hierarchies can also be constructed by layered multicast with multiple groups [134] or by organizing participants in trees [192]. But hierarchical approaches also introduce an overhead for managing the hierarchy, they are problematic if the group of participants changes dynamically, and they may require several request rounds when a packet is lost close to its source (which increases the total distribution delay).

The main drawback of the ARQ approaches is that they increase the notification time for messages by the time needed to detect, request, and resend a lost packet. An alternative idea is to distribute redundant information together with the original data. In case a packet is lost, it can be recovered as long as enough redundancy information is received [121, 19]. This approach is also known as Forward Error Correction (FEC). While it decreases the average propagation delay, it also increases the network load and the overhead for processing packets. For cases where too many packets are lost for compensation to work, FEC might be combined with ARQ [136].

2.3.3.4 Flow and Congestion Control

The network traffic generated by a distributed interactive application might be too high for some receivers or for the network. In the first case, when sites are connected via a low-bandwidth link, the sending rates have to be adapted such that all sites can handle the amount of network traffic. This is the task of a *flow control* algorithm. Second, the network routers might receive more packets than they are able to process. In this case, packets are dropped and the network is said to be congested. A *congestion control* algorithm adapts the sending rates such that network congestion is avoided. In [271], Widmer and Handley propose a multicast congestion control protocol that adjusts a sending rate to the maximum TCP-friendly rate that can be achieved without causing packet loss.

Both flow and congestion control algorithms require that the network traffic is adapted to the current conditions. But for distributed interactive applications, lowering the sending rate is possible only for non-vital messages such as cues. For operations, this is impractical: Unlike audio and video, operations usually cannot be encoded in a lower quality (which would decrease the packet size), and messages cannot be omitted or delayed without endangering the consistency of the shared state and obstructing the collaboration of the users. In [272], Widmer et al. therefore propose to lower the load for a congested network by temporarily disconnecting some of the receivers in a controlled way.

2.3.3.5 Security

The term *network security* covers a wide range of aspects [245]. It includes that certain resources cannot be accessed without having the appropriate authorization. For instance, messages sent over a network can be protected by encryption. For sensitive information, it is also important that the source can be authenticated and that the information cannot be manipulated by others. Security issues can be critical for distributed interactive applications that are used in companies such as the teamwork software Groove [94].

In [268], we propose a novel encryption and authentication algorithm on the basis of the Open Pretty Good Privacy (OpenPGP) [21] protocol where the secret keys are stored on a smart card and never leave it. Instead, the smart card generates a session key that is used on the more powerful end-system to encrypt and decrypt network traffic. This is called *remotely keyed encryption*.

2.3.3.6 Protocol Architecture

The communication functionality of the Internet is organized in independent *layers* [245]. For instance, the network layer is responsible for routing and the transport layer for reliability and congestion control. Each layer performs its tasks independent from the other layers

but provides services to the next layer. Information is exchanged between two layers via a standardized interface, the so-called service access point. The layering approach has several advantages. First, the complex design of a communication model can be reduced to basic functions that are implemented and tested independently. Moreover, layers are generic and can be reused for different applications. Finally, parts within a layer can be exchanged easily as long as the interface and the encoding of packets are not changed.

But at the same time, the layering approach tends to be inefficient since a single packet is processed several times and might contain redundant information for independent layers (e.g., sequence numbers). In addition, the application has only limited possibilities to influence individual functions.

An alternative approach is *Integrated Layer Processing* (ILP), which was proposed by Clark and Tennenhouse [33]. The main idea of ILP is to combine all communication functionality and to process a packet in a single step so that the number of copy operations is reduced, and packet header fields can be used by different functions. ILP requires that all messages are composed such that they can be processed and interpreted independent from other messages [33]. This principle is called Application-Level Framing (ALF), and the messages are also denoted as Application Data Units (ADUs). ALF complies with the operations and cues in our data model for distributed interactive applications. ALF allows to process ADUs immediately even when they arrive out of order. Moreover, protocol functionality can be customized for different ADU types to fit the needs of the application. For instance, while events are transmitted reliably with ARQ, lost cues could be ignored. But when compared to the layering approach, the realization of such an integrated communication module is considerably more complex. Thus, it might be reasonable for applications to combine elements of both approaches.

2.3.4 Generic Services

As indicated above, a distributed interactive application has to address many different issues such as support for collaboration and awareness, consistency control, and network communication. Instead of application-specific solutions, it is therefore desirable to develop generic algorithms that can be employed by different distributed interactive applications so that complex mechanisms have to be designed, implemented, and tested only once [50]. At the same time, such generic solutions should be flexible enough to be adapted to the specific needs of an application.

Generic services can be realized on the basis of our data model for distributed interactive applications [156, 158]: In Chapter 4, a generic consistency control service is proposed, and in Chapter 6 a generic late-join mechanism is presented that allows participants to join

an ongoing session at any time. Other services conceivable are floor and session control, authentication and encryption, and recording and archiving of sessions. All these services can be implemented on the basis of a network protocol that captures and reveals the common characteristics of distributed interactive applications [158]. Such a protocol is introduced in Chapter 7.

2.4 Selected Distributed Interactive Applications

Shared whiteboards are a prominent example for distributed interactive applications. They offer a shared workspace for the presentation and joint editing of documents, and they are used in electronic meetings, group discussions, teleteaching, etc. In Chapter 3, we discuss shared whiteboards in general and then present the multimedia lecture board developed in the course of this thesis. In the following, three other sample applications are introduced: TeCo3D, a shared workspace for interactive 3D models, a multi-player game, and Instant Collaboration, a system for synchronous and asynchronous collaboration. Other examples of distributed interactive applications not discussed here are shared text editors [106], distributed virtual environments [104], distributed interactive simulations [123], instant messengers [275], meeting environments [82], shared animations [142], and workflow systems [1].

2.4.1 TeCo3D

TeCo3D was developed by Mauve at the University of Mannheim [154] and allows to share interactive and dynamic 3D models that are defined in the Virtual Reality Modeling Language (VRML) [264]. Unlike distributed virtual environments, the models themselves are collaboration-unaware, and the collaborative functionality is provided by TeCo3D. For this purpose, a 3D model is loaded into the shared workspace of TeCo3D as depicted in Figure 2.1. The users themselves are not represented by the model but act from an outside perspective. TeCo3D follows the relaxed WYSIWIS principle so that each user may choose its individual view point. The only awareness information given is a list of participants.

Since 3D models are dynamic and may change with the passage of time, TeCo3D falls into the category of continuous applications. Its architecture is replicated, the state of the 3D model is managed as a whole, and the message exchange among the individual application instances uses state and event operations [154]: States distribute the 3D model when it is loaded by a certain user. Interactions with the model are propagated as events (e.g., when a user assembles the parts of a cupboard as shown in Figure 2.1). The message exchange is based on IP multicast, enhanced by a separate reliability layer. All user actions are serialized by a floor control mechanism: Before modifying a model, a participant has to acquire the

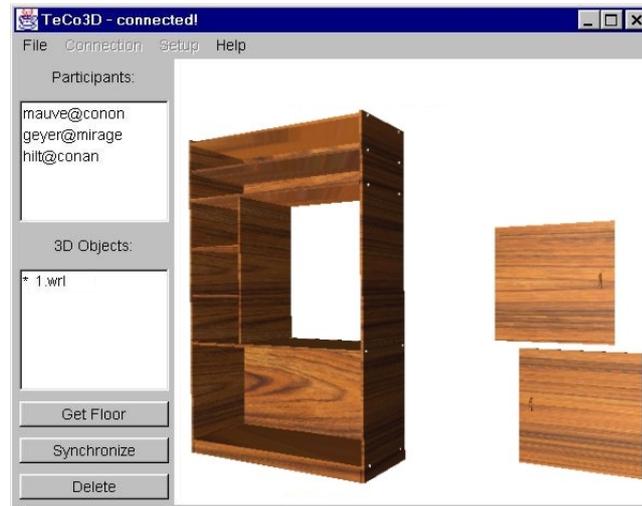


Figure 2.1: TeCo3D user interface

floor. Consistency of the shared state is ensured by the algorithms *local lag* and *timewarp* that are discussed in Chapter 4. As stated by Mauve, an important issue when designing TeCo3D was to realize functionality in the form of generic services [156].

2.4.2 Spaceshooter

Spaceshooter is a simple network game for multiple players that was developed by Friedrich at the University of Mannheim [73] to demonstrate the feasibility of consistency control by local lag and timewarp (see Section 4.7.3). As depicted in Figure 2.2, each player controls a spaceship, which can accelerate, decelerate, turn, and shoot at other ships with a laser beam of a certain range. The spaceship of the local user is marked by a dot in its center. Aside from this marking, the strict WYSIWIS approach is pursued. The game field makes players who leave over one edge reappear at the opposite side. Each spaceship has a predefined vulnerability counted in hit points that are decremented every time it is hit. If no hit points remain, the spaceship is removed from the game. The last remaining player wins.

The game has a replicated architecture, and each player is represented as a spaceship. Aside from the ships, no other objects are present. Since the game field is rather small and unsegmented, all objects are active for all sites. Shooting or changing the direction or speed of a ship triggers an event, which is transmitted via IP multicast. As long as a user does not interfere, a ship follows its current trajectory. Thus, the game is a *continuous* interactive application, which makes the design of an appropriate consistency control mechanism challenging. This will be addressed in Chapter 4.

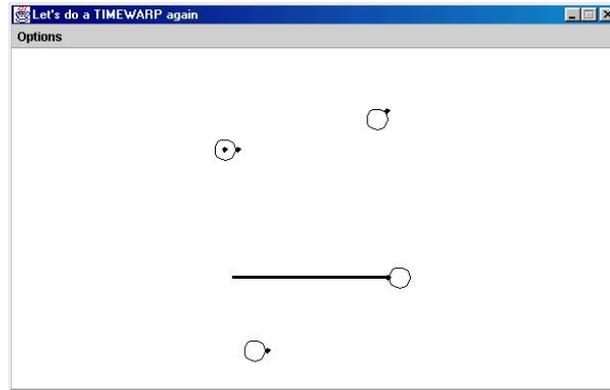


Figure 2.2: Spaceshooter user interface

2.4.3 Instant Collaboration

As indicated in Section 2.3.1, distributed interactive applications realize different levels of formalization with respect to the collaboration among users [79]: Tools such as email and instant messaging are very flexible and ad-hoc forms of collaboration but offer only little support for the users. With email, small pieces of information can be shared easily, but more extensive activities are difficult to manage with messages scattered in the inbox, multiple versions of attachments, varying subjects and addressees, and the lack of real-time collaboration [269]. At the other extreme, groupware systems structure and coordinate interaction but are inflexible, heavy-weight, and require to set up a workspace before content can be shared.

The motivation of Geyer and Cheng when designing Instant Collaboration at IBM Research was to bridge the gap between unstructured and highly structured collaboration by overriding the separation between workspace and content [79]: The content itself becomes collaboration-aware, and each shared piece of information (i.e., object) has an individual set of members, provides object-level awareness, and enables group communication. Shared objects allow both asynchronous and synchronous types of collaboration: If other participants are present at the time of accessing an object, they can work synchronously, if not, work is asynchronous. From a more technical perspective, objects can be considered as “infinite” persistent sessions. So far, there exist five types of shared objects in Instant Collaboration that are all discrete: Message, chat, screen shot with annotations, file, and to-do item. These objects can be structured into larger activities by the users as the collaboration evolves [177].

The prototype of Instant Collaboration is integrated into an email-client as depicted in Figure 2.3: Inbox (1) and message view window (2) are located on the left of the main window, and all shared objects are visualized in a tree hierarchy on the right (3). This design was inspired by Ducheneaut’s and Bellotti’s observation that email is very often the seed of a collaborative activity [53]. It allows the seamless transition between email and object-centric sharing: Regular emails can be turned into shared objects via drag and drop. Shared objects

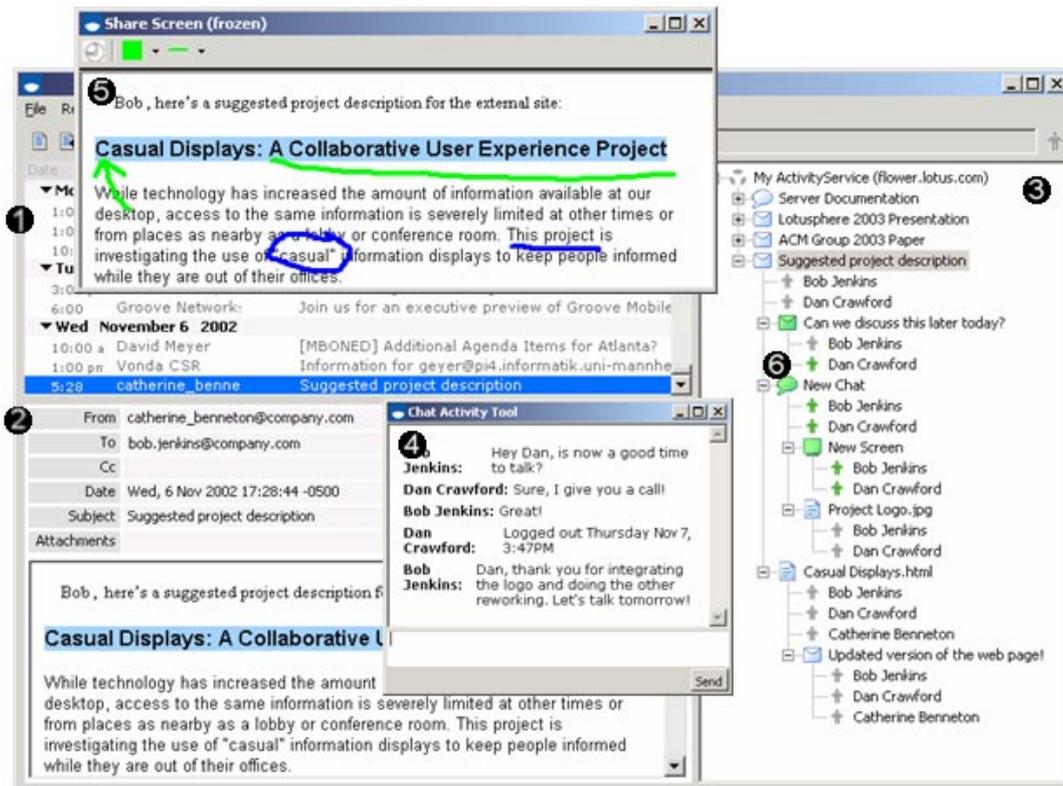


Figure 2.3: Instant Collaboration user interface

can also be created directly by selecting the desired type. Figure 2.3 shows a chat object (4) and an annotated screen share object (5). Attached to each object is a list of all members allowed to access it; they are identified by their email address (or alias). New members may join an object by invitation. All members currently active are marked by a green icon (6) in order to provide a quick overview. In case a new object is created or an existing object is modified, all members are updated eventually (see Section 6.7). As new objects are attached to existing ones, collaboration evolves into a multi-object activity with a dynamic group of participants.

We decided to realize Instant Collaboration with a replicated architecture in order to be independent from a certain infrastructure, and to achieve good responsiveness as well as high scalability [83]. Each site has a local database so that shared objects are persistent even when no application instance is running. The message exchange is based on states and events, and messages are encoded in XML [64] and distributed via TCP by the sender to all receivers. This is reasonable since usually groups are rather small with no more than ten participants, and the amount of data exchanged is small. Because the application state may change when some users are offline, this system design is challenging with respect to the consistency control mechanisms: All state changes that took place while a site was offline need to be propagated as soon as a communication is possible again so that a consistent state is reached. Appropriate algorithms are discussed in Sections 4.7.2 and 6.7.

2.5 Conclusions

The goal of distributed interactive applications is to facilitate human-human interaction with computer support. They allow users to access and modify shared data via a computer network. Even though the spectrum of distributed interactive applications is large, they have a common data model: The state of an application may change because of user actions or because of the passage of time. Only state updates due to user actions need to be propagated in the form of operations. And the state may be composed of independent objects that are either active or passive for a specific application instance.

This model allows us to discuss important issues independent from a specific application so that complex algorithms can be realized as generic services that may be employed by many different applications. In the remainder of this thesis, we concentrate on consistency control, support for late-joining participants, transport of operations, and group communication. Moreover, the feasibility of these algorithms is demonstrated on the basis of representative applications and simulations. In the next chapter, the shared whiteboard *mlb* is discussed in detail.

Chapter 3

The multimedia lecture board (mlb)

Shared whiteboards are one of the most prominent and most commonly used distributed interactive applications. They provide a shared workspace where users collaboratively create new documents, or present and edit existing documents [69]. Shared whiteboards are employed in many video conferencing and electronic meeting scenarios for sketching ideas, taking notes, exchanging information, presenting documents, and discussing items. For instance, in a teleteaching scenario the lecturer uses a shared whiteboard to present slides to the students that attend the session either locally, in a remote lecture room, or at home [57]. New slides can be created or existing slides can be annotated by all participants (including the students) using graphical objects such as freehand lines, text, and imported images.

Most shared whiteboards concentrate on synchronous collaboration. And in most cases, they do not support objects that change with the passage of time. They are often used together with other media and applications. For instance, in teleteaching scenarios audio and video are exchanged among the participating sites using appropriate software or hardware such as H.320 video conferencing systems, the Mbone tools vic [164] and rat [212], or MPEG-2 hardware codecs.

After analyzing existing shared whiteboards and their deficiencies in Section 3.1, the multimedia lecture board (mlb) is presented that was developed in the course of this thesis. First, an introduction to the mlb's basic whiteboard functionality is given in Section 3.2. Following, the data model and the architecture of the mlb are discussed in Section 3.3. In Section 3.4, presentation animations are described as an important feature for illustrating talks. The visualization of awareness information and tools to support multi-user collaboration integrated into the mlb are presented in Section 3.5. The mlb can also be used together with handheld devices, which facilitates novel usage scenarios (see Section 3.6). In Section 3.7, software

development issues and experiences with the mlb are discussed. The chapter is summarized in Section 3.8.

3.1 Related Work

One of the first whiteboards with a replicated architecture is wb [126]. It provides a simple drawing area where graphical objects and text can be created by all users. Because of its limited drawing functionality, wb is mostly used to annotate preauthored documents, which can be imported as postscript files. The only awareness information provided is a list of all participants together with information regarding the number of operations issued by each user and the time of the last activity. The wb follows the relaxed WYSIWIS principle and allows a user to view other pages than the active one. Even though the wb has a replicated state, it does not employ a consistency control mechanism but relies on the users to avoid concurrent changes. Communication is based on IP multicast and the reliability protocol SRM (see Section 2.3.3).

The successor of wb is the MediaBoard [249] that is part of the MASH project [185] at UC Berkeley. The goal of MASH is to integrate tools and applications for video conferencing in a joint framework with a common user interface. In addition to the basic whiteboard functionality of wb, the MediaBoard displays more awareness information and indicates which participant is responsible for the current state change. It also allows to form subgroups working on different slides. Another interesting feature is that the local operation history can be replayed by traversing older states. The MediaBoard has a replicated architecture and follows the ALF principle by integrating the reliable multicast protocol SRM into the application. This allows to repair packet loss concerning the active page with a higher priority than other loss. All operations are distributed in the form of events to the other instances of the MediaBoard, which requires an inefficient replay of the operation history in case a site needs to retrieve the current state (see Chapter 6).

The Authoring on the Fly (AOF) system of the University of Freiburg integrates different tools for the production, delivery, and postprocessing of teleteaching courseware [9]. The whiteboard AOFwb allows a lecturer to present slides to local and remote students [149]. AOFwb also supports presentation animations, offers a thumbnail view of all pages, and provides a telepointer so that the lecturer can point to certain areas of a slide. However, even though the whiteboard is shared, it is not collaborative, i.e., only the lecturer is able to issue state changes. Remote students need a special software to receive lectures (AOFrec). Presentations can be recorded locally together with audio and video in order to generate course material for offline learning. The recorded data streams can also be postprocessed with AOFedit (e.g., to cut speech pauses). The data model of AOF transforms all applications into the discrete domain,

i.e., continuous animations are sampled by storing their state periodically. Generally, all state changes are encoded as new full states in order to simplify random access to recorded data streams. This results in very high data rates.

The digital lecture board (dlb) [80] is a precursor of our mlb. It was developed by Geyer at the University of Mannheim for teleteaching scenarios [57]. Aside from creating discrete graphical objects, the dlb allows to display and manipulate 3D models described in VRML in the shared workspace [81]. The dlb also offers a variety of collaborative tools, which support the interaction among users and facilitate awareness: Voting and feedback tools allow to gather and aggregate opinions of all participants without relying on an audio channel [78], a hand-raising tool can be used to coordinate speakers, and an integrated chat tool provides an additional communication channel. Moreover, the dlb visualizes mouse movements within the shared drawing area with a telepointer. Following the relaxed WYSIWIS principle, users can prepare slides in a private workspace, privately annotate pages in the shared workspace, and view pages independent from other users. Users and resources can be managed by session and floor control [113]. The dlb has a replicated architecture, and its layered communication model is based on states and events. These are distributed via IP multicast and a separate reliability layer, the Scalable Multicast Protocol (SMP) [97], which employs the receiver-based ARQ mechanism of SRM [68]. Aside from the source-ordering of operations ensured by SMP, the dlb does not integrate any consistency control mechanisms but relies on the users or the floor control to prevent conflicting operations. This might be problematic in scenarios with frequent interactions. Messages can be encrypted using different algorithms [85]. On top of SMP, the dlb encodes operations with the Real-time Transport Protocol (RTP) [219] (see Section 7.1), which allows to record dlb sessions in synchronization with parallel audio and video streams [114].

In addition to the approaches presented above, most systems for tele-collaboration include simple whiteboards with basic shared drawing functionality such as Microsoft NetMeeting [167], the Modular Advanced Collaboration System (MACS) [17], mStar [176], SunForum [243], and Interactive Remote Instruction (IRI) [153]. Aside from shared whiteboards, non-shared electronic whiteboards supporting face-to-face meetings have received considerable attention from human-computer interaction research. For instance, Tivoli [193] and Flatland [180] are two systems, which provide rich interfaces for pen-based interaction (see Section 3.6).

In the remainder of this chapter, we present the multimedia lecture board (mlb) whose development was a major goal of this thesis. The mlb's design is inspired by our experience with the dlb and the AOFwb.

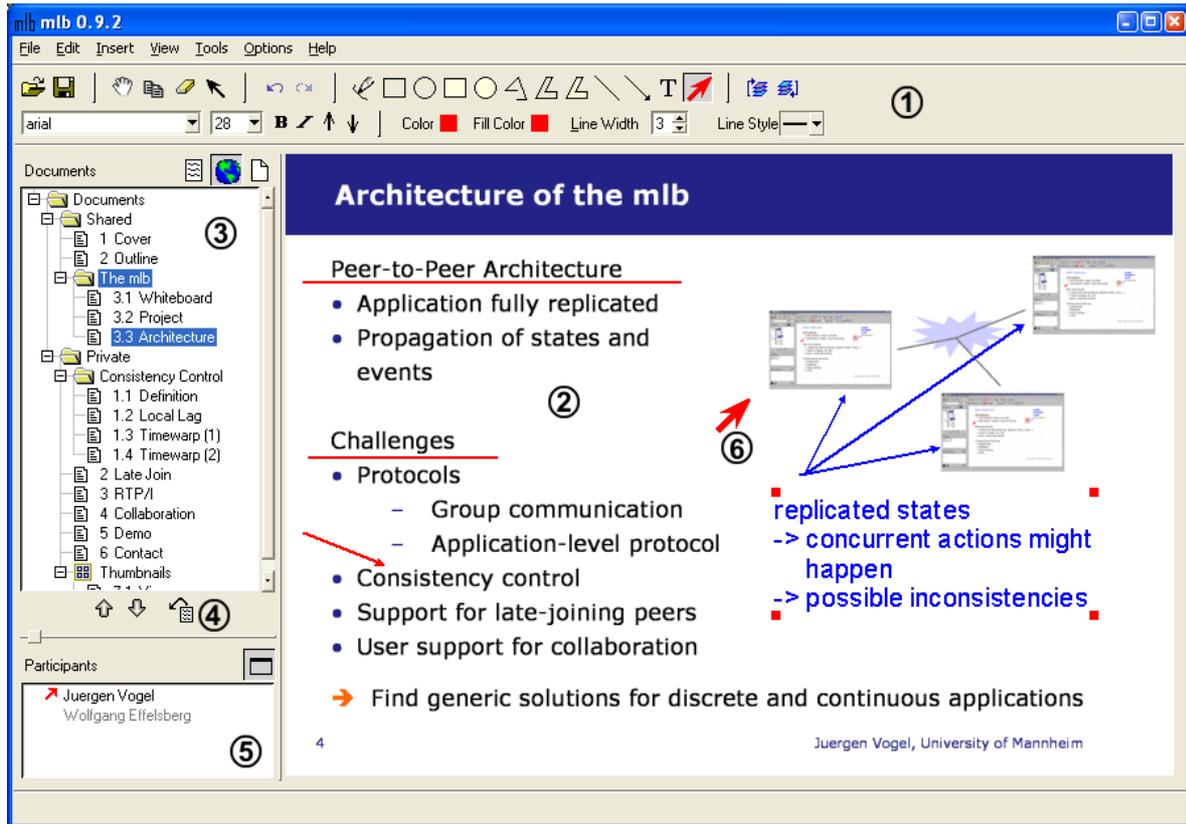


Figure 3.1: mlb user interface

3.2 Basic Features

The user interface of the multimedia lecture board (mlb) [259] is depicted in Figure 3.1: The tool bars (1) determine the next user action, the workspace (2) displays the page that is active for the local user, and the window pane to the left of the workspace may contain different tools and information such as the page hierarchy (3). In the following, we explain the mlb's features from the perspective of the user.

The mlb offers a large number of tools to create, modify, and present documents. A document has a hierarchy of chapters and pages (3), and each page may contain an arbitrary number of graphical objects from the set of freehand line, rectangle, oval, polyline, polygon, line, and arrow. Which object to create with which attributes is selected in the tool bar (1). Moreover, the mlb allows to create text objects using TrueType fonts, and is able to import external postscript and image files such as GIF, PNG, JPEG, BMP, etc. All objects can be deleted or modified, e.g., with respect to their color, size, or position. Hereby, documents can be prepared for later presentation or are created during a session. All user actions changing the state of an object can be undone and redone (see Section 4.8).

It is possible to open multiple documents at the same time; each document resides in its own workspace. But there is only one shared workspace, which is accessible to the local and all

remote users. The shared workspace is identified by a unique session address, and in case a user wants to join a session, he has to open a shared workspace with the appropriate address (see also Section 3.3). Generally, all actions within the shared workspace are visible for all participants, all users see the same active page (2), and all users are able to modify the state of the objects displayed. Thus, the shared workspace allows to create a joint document. All other documents are contained in private workspaces and are visible to the local user only. A private workspace can be employed to prepare a document, which is to be presented at a later point in time. The document tree in Figure 3.1 (3) shows one shared and one private workspace.

The mlb follows the relaxed WYSIWIS principle and allows a participant to annotate the shared workspace with private objects. These are visible only locally and can be used to take notes in a presentation. It is also possible to browse the pages of the shared workspace without disturbing the other session members. Moreover, the user interface of the mlb can be adapted to the likings of each user. For instance, the zoom factor of the pages can be selected, and the tool bars and tools, which should be visible, can be set. For the presentation of preauthored documents, a full screen mode exists, which is less distracting than the editing mode. All options regarding the user interface and other aspects such as session parameters are saved in a configuration file.

Large documents can be organized with the help of chapters. Like pages, chapters may be named individually. Since chapters may contain pages and other chapters, arbitrary hierarchies can be created (e.g., see Figure 3.1 (3)). A document hierarchy can be changed easily with drag and drop operations. The exchange of content among different workspaces is also possible via drag and drop: In case a page or a chapter is transferred into the shared workspace, all relevant state information is distributed to the other session members so that the relocated content becomes globally visible. When a participant leaves a session, the shared workspace is transformed into a private workspace so that no content is lost. By means of the late-join algorithm described in Chapter 6, a user can also rejoin a session at a later point in time and retrieve the current state of the shared document. This allows to assign certain tasks to independent teams and merge the results later. In a teleteaching scenario, the lecturer could access the ongoing work of the individual groups one after another.

The user can navigate within the document hierarchy using the mouse or the key bindings (page up, page down, etc.). If not in the local browsing mode, a change of the active page within the shared workspace is announced to all participants. The mlb also provides a thumbnail overview of all pages within a workspace. As depicted in Figure 3.2, the thumbnail overview contains a miniaturized version of all pages, and the user can activate a certain page by selecting the corresponding thumbnail with a mouse click. A special import button simplifies the presentation of a document and transfers the respective first page of a private



Figure 3.2: Thumbnail overview

workspace into the shared workspace (see Figure 3.1 (4)). The transferred page is also activated automatically for all participants. With this function, the network load is spread over a longer time span when compared to distributing all pages of a document to all participants at the start of the session.

All documents can be saved in an XML-compatible [64] format so that the content becomes persistent. The XML format basically allows to edit mlb documents with other applications. It is also possible to export mlb documents as HTML, which can be displayed by any web browser. Moreover, a document can be saved page-by-page in an image format. Documents created with other applications such as Microsoft PowerPoint [169] can be imported as postscript, PDF, or pagewise via one of the supported image formats. The latter possibility usually results in the best quality. Each page of the original document is then displayed as a background image on an mlb page and can be annotated using the graphical objects of the mlb. For easier handling, we developed various tools automating this conversion [259].

3.3 Data Model and Architecture

After describing important features from the user's perspective in the last section, the architecture and other technical aspects of the mlb are addressed. The mlb is implemented in C++ and Tcl/Tk, and the advantages of object orientation are exploited in the code. As indicated above, the mlb may hold several documents containing chapters, pages, and graphical objects. This is reflected in the class hierarchy of the mlb. As depicted in Figure 3.3, all containers and graphical objects are derived from a generic object class with important properties common to all classes. Text is modeled as a composite of `TextGroup`, `Text`, and `Character` objects: When the user begins a new text by placing the cursor on a page, a `TextGroup` object is created. This contains a `Text` object for each text element with homogeneous font

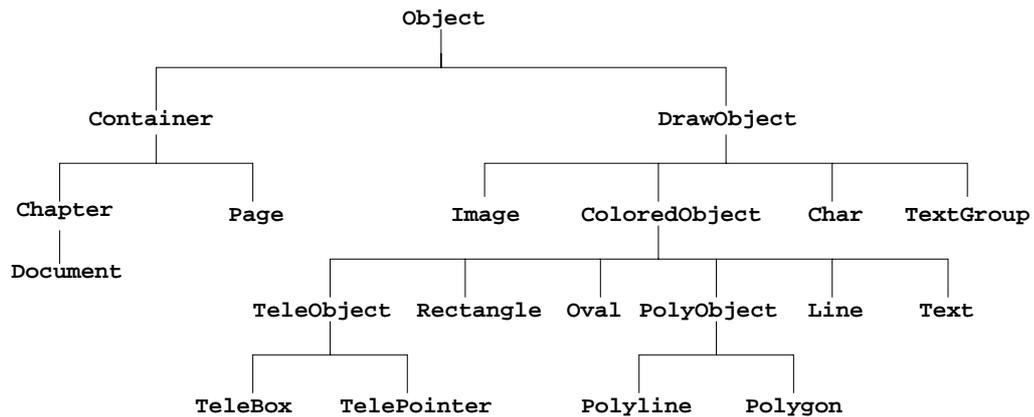


Figure 3.3: Class diagram for mlb objects

properties (size, color, etc.). Text objects hold a list of Character objects and are for data management purposes only.

3.3.1 Unique Identifiers

As discussed in Section 2.2, unique identifiers are required to reference objects and to relate operations to their target objects. There exist several alternatives for generating identifiers. A common approach is to select identifiers randomly from a certain range [219]. If this range is large and the number of selected identifiers small, the probability that two identifiers are identical is rather small. Nevertheless, identifiers may collide, and a mechanism is necessary to detect and resolve such collisions. For distributed interactive applications where the consistency of the shared state depends on operations being assigned unambiguously to their target object, the random approach is therefore not well suited [111].

An alternative approach for generating unique identifiers is to use the network layer or physical layer address of an application instance as a basis. An object identifier could then be created by using the MAC address plus an additional counter. However, it is by no means guaranteed that the address of an end-system is unique (e.g., some network interface cards allow to change the predefined MAC address), and the resulting object identifiers would be rather large and introduce a high overhead for the encoding of operations.

The last alternative is to assign identifiers in coordination among all participants such that collisions are precluded. For instance, a well-known server could manage the identifier namespace and be contacted when a new identifier is needed. This server could be responsible for multiple sessions, and each session could have several independent namespaces (e.g., one for containers and one for graphical objects). In a request, an application instance then provides the server with sufficient information to select the appropriate namespace. The server can also guarantee the persistence of identifiers so that an application instance could regain its

identifiers when it rejoins the session after it crashed. The scalability of this approach can be improved when application instances do not request identifiers individually but in ranges (e.g., 100 identifiers per request), and when several servers exist where each is responsible for a different area of the hierarchically structured identifier namespace. A prototype of this service was developed for the mlb, and the size of identifiers is set to 32 bit.

As long as an object resides within the same workspace, its identifier does not change. But when an object is transferred from a private to the shared workspace, it has to be assigned a new identifier in order to prevent a possible collision. For instance, consider a situation where a participant leaves the current session. Then the content of the shared workspace becomes a private document for this participant as discussed above. In case he rejoins the same session at a later point in time, his application instance retrieves the current state of this session's workspace (see Chapter 6). But the user's mlb also holds the older version of this state in a private workspace. In case content is transferred from this private workspace into the shared document, identifiers would collide. Thus, the mlb assigns new identifiers to all objects that are moved from a private to the shared workspace.

3.3.2 Display Order of Objects

Another important property that needs to be unique and identical among all mlb instances is the display order of graphical objects: The display order determines which object is displayed on top in case two objects overlap on a page. As is common for graphics applications, the mlb creates new objects above older ones. This display order might be changed by *raise* or *lower* operations: A lower (raise) operation places the concerned object below (above) of all other objects. Aside from graphical objects, a display order is also required for the ordering of chapters and pages within their container (see Figure 3.1 (3)). This hierarchy might be changed by placing an object at a new position, which can be anywhere in the tree. Because of these possibilities to change the display order, the order in which objects are created originally is not sufficient, and the display order has to be encoded explicitly as a part of each object's state.

For the mlb, the following algorithm was devised: The display order is represented by a 32 bit integer o (i.e., $0 \leq o \leq 2^{32} - 1$) where a higher value means a higher order. All objects within a container form a list, which is sorted ascending to o . Let $o_{max} := 2^{32} - 2$. In case a container is empty, the first object inserted is assigned the order $\frac{o_{max}}{2}$. If an object is appended to a non-empty list (i.e., a new object is created or a raise operation is issued), its order is set to the order of the current last object plus an offset o_{Δ} . In case the lower operation is issued for an object, it is given the order of the first object decremented by o_{Δ} . And if an object is placed between two other objects i and j with order numbers o_i and o_j , it is assigned the

order $\frac{o_i + o_j}{2}$. We propose to set $o_\Delta := 2^{16}$, which gives enough clearance to append and insert objects in most cases.

However, two problems might occur: First, the display order namespace might not allow to place an object at the front or the back of the list or between two other objects. In this case, the namespace is depleted, and our mechanism reallocates the namespace by assigning new ordering numbers to all objects, placing the middle object in the middle of the 32 bit namespace. But in our experience, the namespace for order numbers is large enough so that a reallocation is almost never necessary. Second, two ordering numbers may collide when two participants create new objects at the same time. Then the consistency control algorithm described in the next chapter determines which object should be created first, and the other object's display order is set to the next available order o . This is possible since the consistency control algorithm ensures that all operations are executed in the same order at all mlb instances. Thus, in case the display order of an object needs to be recalculated locally, the algorithm will reach the same result at all sites.

3.3.3 Data Model

Because of the reasons discussed in Section 2.1, the mlb has a replicated architecture. The mlb only allows state changes due to user actions and therefore is a discrete interactive application. The creation of a new container or graphical object is propagated as a state. All other changes within the shared workspace are encoded as events or cues. Cues are used for intermediate state updates, which are followed by an event. For instance, in case the user moves a graphical object to a new position, the intermediate coordinates are encoded as cues. Only the object's final position is an event. This distinction allows to handle intermediate changes differently than final changes (see below). In order to save network resources, the user can define the percentage of intermediate local changes that should be propagated as cues. From our experience with capturing mouse movements, a cue rate of only ten percent of all mouse events is sufficient for a smooth visualization.

All cues and events are encoded and interpreted absolutely in order to prevent dependencies within the operation history. For instance, instead of encoding a "change size" operation as "increase size by 10 percent", the event defines the final size. The most interesting object in this respect is text: While a certain character can be deleted by an absolute operation that references the character object's identifier, an index has to be defined when inserting a new character. Most shared text editors interpret insert operations relatively since the intention of the user is to place the character at a distinct position in a word, and the corresponding index depends on the insert and delete operations that were executed earlier [138, 238] (please refer to Sections 4.3 and 4.8 for a more detailed discussion). Since such dependencies within

mlb protocol
RTP/I
SMP
UDP
IP multicast

Figure 3.4: Protocol stack

the operation history would complicate the handling of operations, the mlb interprets insert character operations absolutely, i.e., a character is always inserted at the index encoded in the corresponding operation. Because text objects are rarely edited by multiple users at the same time, this simplification has no negative effect in practice.

Since all users can individually adjust the zoom factor of their local view on the shared workspace, all pixel coordinates contained in states or state updates need to be transcoded to standardized coordinates. When interpreting a received operation, an application instance then applies its local zoom factor.

The mlb displays exactly one page at a certain point in time so that only the graphical objects of this page are active objects (see Section 2.2). Moreover, the current page and all chapters in the document tree up to the root are active objects.

3.3.4 Communication Model

The mlb manages the message exchange among the application instances participating in a session with the communication protocols depicted in Figure 3.4: Group communication is currently provided by IP multicast, but we are planning to switch to application-level multicast in the future (see Chapter 8). Reliable transport of data is provided by the Scalable Multicast Protocol (SMP), which was originally developed for the dlb at the University of Mannheim [97]. SMP is a receiver-based ARQ protocol and is realized as an independent protocol layer with its own packet header (see Section 2.3.3). But at the same time, it is integrated as a library into the mlb and offers a rich interface, which allows to influence the service level: The mlb transmits operations as ADUs, which can be handled independent from other ADUs in case they concern independent parts of the shared state. Thus, the source-ordering function of SMP is disabled for the mlb, and ADUs are ordered by the mlb's consistency control algorithm instead (see Chapter 4). Moreover, SMP allows to transmit data unreliably, which is used for cues: Since cues either do not change the application state or are overwritten by a following event (which is propagated reliably), their loss does not have a severe impact on the application. SMP also offers a basic flow control mechanism for adapting the send rate to the user's setting. Congestion control is not part of SMP.

The Real-time Protocol for distributed Interactive applications (RTP/I) was developed by Mauve et al. at the University of Mannheim [159]. It captures the common aspects of distributed interactive applications, as defined by the data model presented in Section 2.2. For instance, operations are classified into the categories of state, delta state, event, and cue. Such information allows to design algorithms as generic services. RTP/I is realized as a library, which offers direct access to its functionality. As suggested by the ALF approach, the application can reuse the header fields of RTP/I ADUs for other purposes. RTP/I was strongly influenced by the mlb and its complex, hierarchical state. It will be discussed in detail in Chapter 7. RTP/I also makes it possible to record and replay mlb sessions together with audio and video streams (see Section 7.4.3).

Finally, user actions are encoded as ADUs by the mlb protocol, which defines how states and state updates of all objects supported by the mlb are propagated [257]. This protocol is general enough to be reused for other shared whiteboards. It is presented in Section 7.5.2.

3.4 Presentation Animations

A presentation animation allows to predefine a sequence of actions that change the application state. The user may then traverse this sequence stepwise. For instance, a presentation animation could organize the content of a slide into multiple steps, each step showing a certain amount of text and graphical objects. Thus, presentation animations are a powerful way to structure presentations and to visualize processes. They are widely used in single-user presentation software such as PowerPoint [169] and OpenOffice [186] and are also valuable for shared whiteboards. Presentation animations can either be user-controlled or time-controlled. In the first case, the user determines when the next state change should happen, e.g., when unveiling the next line of text by a mouse click. Time-controlled animations change the state of an object with the passage of time, e.g., when moving an object along a predefined path. For the mlb, only user-controlled presentation animations were realized in order to keep the application discrete.

For each page, a presentation animation can be defined using the interaction window depicted in Figure 3.5 that is displayed on the tool pane of the mlb's user interface (see Figure 3.1). An animation consists of an unlimited sequence of steps. In each step, an arbitrary set of objects can be revealed (+) or hidden (-). The same object may occur in multiple steps. To execute an animation, the user has to activate the presentation mode. In this mode, the next animation step is triggered by clicking the mouse or typing a key. When the last step of a page is reached, the next user action automatically activates the following page.

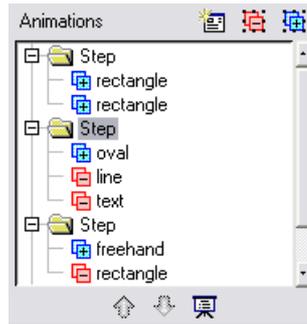


Figure 3.5: Presentation animation

The mlb file format supports presentation animations so that they are saved (restored) when storing (opening) a document. An animation itself is not part of the shared workspace, and only the user who created an animation (or loaded a document with animations) can see the steps in the animation window of Figure 3.5. However, the animation is visible to all participants of a session. The mlb’s data model facilitates a straightforward implementation of presentation animations: An animation is executed by issuing “show” and “hide” events for the objects concerned when the next step is activated.

3.5 Collaboration and Awareness

Aside from the functionality for editing shared and private documents, the mlb offers a variety of tools to coordinate the collaboration among the participants and to provide awareness information. As discussed in Section 2.3.1, these services are vital for multi-user scenarios where social protocols such as eye contact and gestures are captured insufficiently or not at all by the audio and video channels.

3.5.1 Visualization of Participant Information

The mlb provides a lightweight session control where a session basically is open to everyone where users may join and leave at any time, and where all users have the same rights [107]. Under these conditions, it is especially important for the users to have access to information about the other participants. The mlb displays a list of all participants in its tool pane (see Figure 3.1 (5)). When selecting a certain participant, the information window depicted in Figure 3.6 is opened. The canonical name (CName) uniquely identifies each participant and is composed of the user’s login name and the IP address of the end-system. All participant information is exchanged periodically with RTP/I among the session members (see Section 7.3.2); this is similar to the RTCP protocol of standard RTP [219].

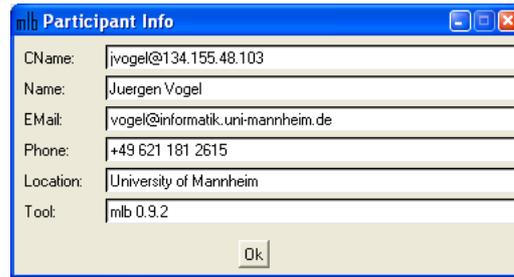


Figure 3.6: Participant information window

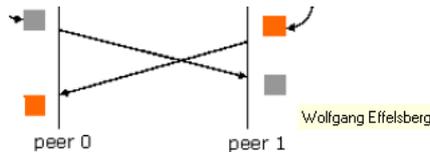


Figure 3.7: Indicating the origin of an operation



Figure 3.8: Awareness information

In order to simplify the interaction among participants, the mlb does not employ floor control. This implies that any user can modify the shared state, and it is not always obvious who is responsible for a certain change. For this reason, the user may activate a tracking mechanism to display the name of the responsible participant in a temporary tool-tip window next to the modified object. In Figure 3.7, the user “Wolfgang Effelsberg” just created the lower right rectangle.

Active participants are also highlighted in the participant list: As indicated in Figure 3.8, the participant “Juergen Vogel” just issued an operation, which is visualized by displaying his name in black. In case a participant stays inactive for a certain period of time, the color of his name degrades (“ages” [11]) to light gray in several steps. The participant list also visualizes other awareness information. For instance, users that left the session or were disconnected due to a technical failure are marked: In Figure 3.8, the participant “Wolfgang Effelsberg” is marked as disconnected by the icon in front of his name. This visualization is more noticeable than deleting the user’s name from the list when the session has many members. The user may clear the list from disconnected participants.

3.5.2 Telepointer

For graphical user interfaces, the mouse is a vital input device to control the application. Telepointers are used to visualize local mouse actions that would otherwise be invisible for remote participants [59]. Telepointers therefore provide important awareness information to remote users [103]. A telepointer can be realized as a single shared pointer or as multiple pointers [181]. In the first case, the local user shares the system’s mouse pointer with all remote participants: As long as the local user moves the mouse, he has control over the

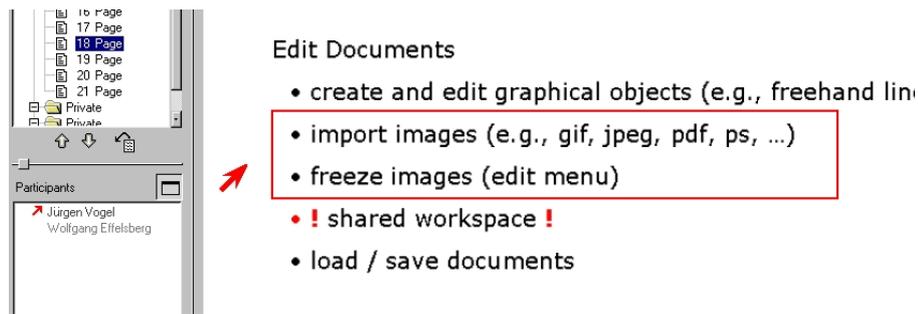


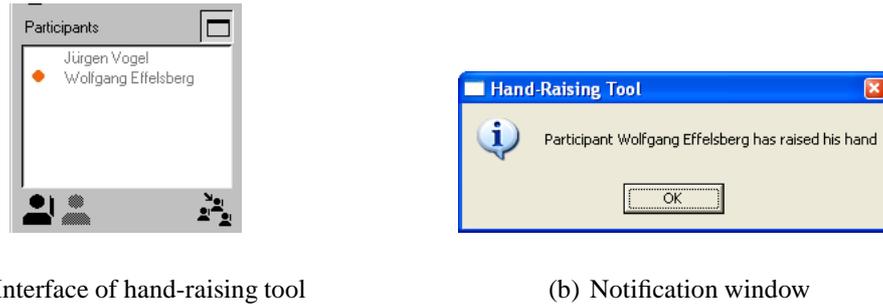
Figure 3.9: Telepointer and telebox

system's mouse pointer. But while he is inactive, the mouse pointer acts as telepointer, which now visualizes the mouse movements of a certain remote user. This remote user can be the currently active participant or the present floor holder in case the application has a floor control mechanism. But this seamless transition between local pointer and telepointer can be rather distracting for the user. The shared pointer approach also allows only one telepointer for all participants.

Thus, the multiple pointer concept is employed for the mlb where each participant controls its own telepointer. In order to distinguish pointers, which are visible simultaneously, users can choose individual colors for their pointers [78]. As depicted in Figure 3.9, each participant with an active telepointer is also marked with a small telepointer icon in the participant list. Since this icon is displayed in the telepointer's color, the participant list gives a quick overview about which telepointer belongs to which participant.

The user activates the telepointer by selecting the appropriate button in the tool bar as depicted in Figure 3.1 (1). The mlb supports telepointers only in the shared drawing area, i.e., mouse movements outside that area are not visible to remote users, e.g., when selecting a menu item. While this limits the amount of awareness information, Geyer argues that it is reasonable for applications that follow the relaxed WYSIWIS paradigm and that allow users to select an individual screen layout [78]. Otherwise, it might not be possible to display a telepointer at its correct position, or the telepointer's path might be confusing ("jumping telepointer") [181]. The mlb also provides a telebox to frame a certain area of the shared workspace [44]. It is assigned the same color as the participant's telepointer (see Figure 3.9). Other forms of tele-objects are conceivable, e.g., freehand lines.

Even though telepointers and teleboxes are completely integrated into the mlb's user interface, internally they are managed as an independent distributed interactive application. For each active telepointer (telebox), the shared state contains one object whose attributes define its color, position, etc. [258]. The first position of a pointer movement is transmitted as a state, intermediate positions as cues (unreliably and with the selected cue rate, see Section 3.3.4), and the last position as an event. Telepointer and telebox positions are propagated in stan-



(a) Interface of hand-raising tool

(b) Notification window

Figure 3.10: Hand-raising tool

standardized coordinates as described above. All telepointer operations are transmitted via RTP/I, SMP, UDP, and IP multicast in an independent session. Since each participant controls his own object, no consistency control mechanism is necessary.

3.5.3 Hand-Raising

Coordinating multiple session members in group discussions or teleteaching sessions can be difficult with the limitations of the audio and video channels that are usually encountered in electronic meetings: For larger groups, there is not enough screen space to have a video window open for every remote participant. Also, there usually is only one audio channel which requires some discipline to prevent that participants disturb each other.

To overcome this limitation, the mlb offers a hand-raising tool, which allows to coordinate participants explicitly and was inspired by the dlb [78]. Its user interface is integrated into the participant list as shown in Figure 3.10(a). For instance in Figure 3.10, the user “Wolfgang Effelsberg” has a question and requests attention by pressing the “raise hand” button. This is indicated to all session members by a flashing orange icon next to the requester’s name in the participant list (see Figure 3.10(a)), and (if desired) by the dialog window shown in Figure 3.10(b). Since multiple users are able to raise their hand, the right to speak is granted by selecting a user and pressing the “give control” button. The called participant is then notified by a dialog window. A user may also cancel his earlier request with the “lower hand” button. Please note that due to the lack of a true floor control mechanism, the hand-raising tool is actually not able to assign resources to a certain participant but signals requests only.

The hand-raising tool is also designed as a distributed interactive application of its own [256]. Its state is rather simple with a list of all participants that currently raise their hand. There exist three different events according to the possible actions described above (“raise hand”, “lower hand”, and “give control”). States and events are exchanged via the same protocols as the mlb traffic. Again, a consistency control algorithm is not required.



Figure 3.11: Chat window

3.5.4 Chat

The mlb provides a chat tool as an additional communication channel, which allows an easier exchange of text messages than with the mlb’s shared workspace. The chat is also useful in situations where the audio channel fails or for discussing issues without interrupting an ongoing presentation (“sidetalks”) [78]. Figure 3.11 depicts the user interface of the chat tool. A history of all messages is displayed below the entry fields for the local user. The chat tool is realized as an independent application with its own session. Its state contains a single object that holds a limited history of the last messages typed. The default value of 20 messages can be changed by the user. The only event possible is to send a new message. A consistency control mechanism is required to determine the correct order of all messages (see Chapter 4).

3.5.5 Voting and Feedback

An important task in group discussions is to make decisions about certain topics, e.g., when voting about different design possibilities or when solving a problem in a lecture. To do this via the audio channel in sessions with a large audience is rather complicated and time-consuming. A decision support tool that is designed for polling the opinions of all participants therefore is a vital element of collaborative systems [190], which also provides rich awareness information [125, 216].

Like the dlb, the mlb integrates two decision support tools: A voting tool and a feedback tool [78]. The user interface of the feedback tool is displayed in the tool pane of the mlb. It allows to continuously evaluate the current session in terms of different (user-defined) features. For instance, Figure 3.12(a) initiates a survey of the perceived audio quality of the individual session members. An overview of all surveys defined for the current session is given in Figure 3.12(b) where the user may change the parameters of a topic or disable it temporarily (like the “Video Quality” survey in Figure 3.12(b)). A user can vote using the respective slider in Figure 3.12(c). The accumulated result for all participants is then

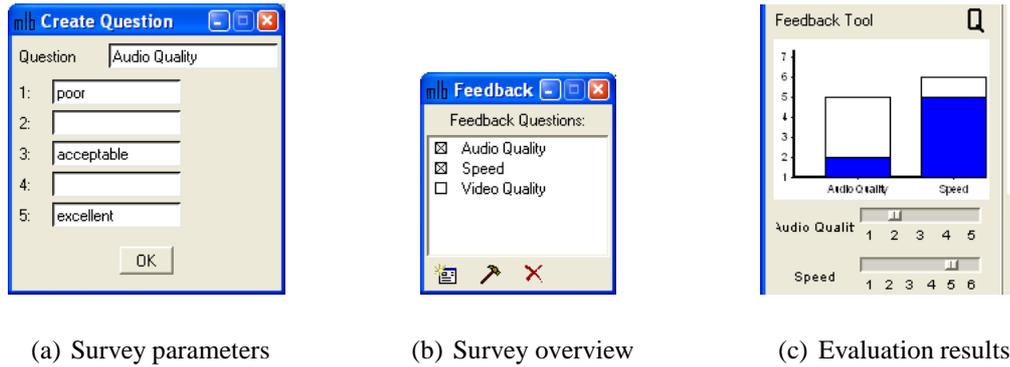


Figure 3.12: Feedback tool

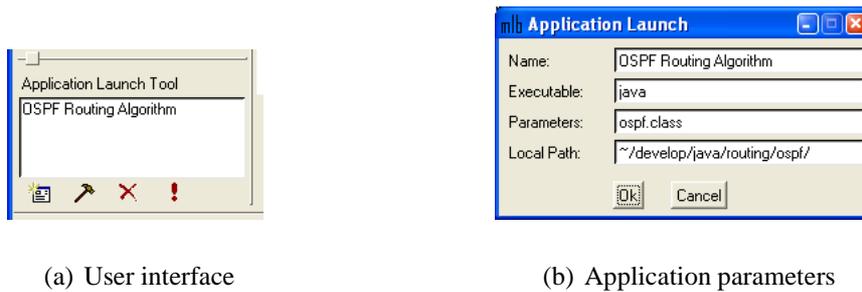
displayed as a bar chart, which gives a quick overview and allows the users to react quickly to changes.

The voting tool allows to evaluate both quantitative (i.e., multiple choice) and qualitative (i.e., text) questions via a separate user interface, which gives more detailed information than the feedback tool. Both the feedback and the voting tool are independent distributed interactive applications, which form sessions of their own [255]. Their states maintain an object for each topic, including the collected answers. The message exchange and the consistency control are managed as in the mlb.

3.5.6 Application Launch

The application launch tool can be used to start external applications synchronously at all sites. Its purpose is to allow the integrated presentation of multimedia elements to all participants without leaving the mlb environment. For instance, a video clip or an animation could be displayed together with a presentation. Figure 3.13(a) shows the user interface of the application launch tool, which is integrated into the tool pane of the mlb (see Figure 3.1). The details of a specific application are depicted in an extra dialog. For instance, Figure 3.13(b) gives the parameters of a Java animation. The user starts an application at all sites by selecting the appropriate entry and pressing the “execute” button. This requires that the application to start is installed at all sites and that the local paths (see Figure 3.13(a)) refer to the application’s directory. Aside from starting the application, the user has no additional control over the individual instances (as he would have with application sharing).

Like the other collaborative tools described above, the application launch tool is a separate discrete distributed interactive application [253]. Its state consists of a list of external applications together with their parameters whereas the local paths of the applications are not a



(a) User interface

(b) Application parameters

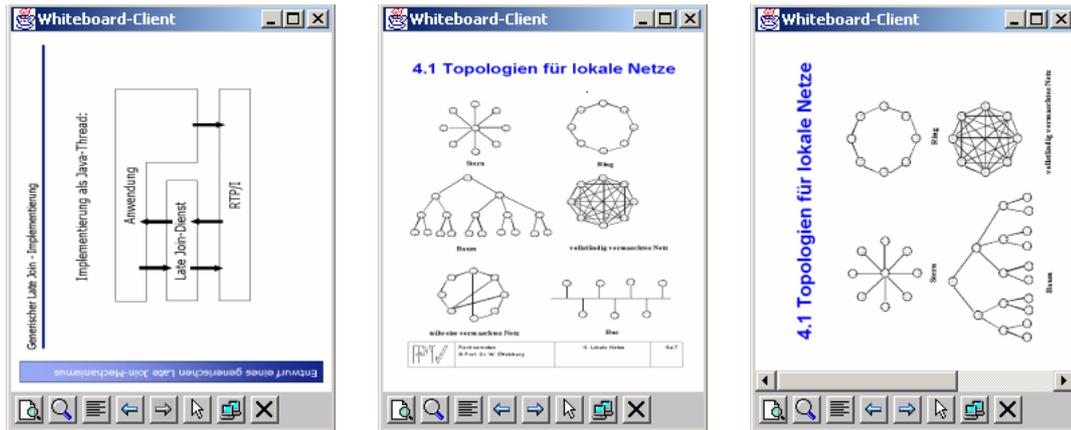
Figure 3.13: Application launch tool

part of the shared state. The tool uses the same consistency control algorithm and network protocols as the mlb.

3.6 Handheld Devices and Pen-Based Interaction

The mlb was mainly developed for a standard PC with a regular monitor as output device, and keyboard and mouse as input devices. However, the mlb can also be employed together with large screen systems such as the LiveBoard [60] or with small handheld devices, e.g., pocket PCs [168]. While the mlb can be executed unmodified on large screen systems (i.e., the large screen is just a different output device for a PC), this is not possible for handheld devices since these lack the required resources with respect to processing power, memory space, as well as screen size and resolution. Thus, a lightweight *pocket mlb* with a limited functionality was developed in this thesis.

Due to their limited resources, handheld devices are mainly used to support collaboration in face-to-face scenarios where additional communication channels such as audio and video are not required [179]. The handheld device gives access to the shared workspace of a distributed interactive application such as the mlb. Figure 3.14 depicts the user interface of the pocket mlb, which is able to display the active page. In order to optimize the usage of the limited screen size, pages are shown in full screen either in a landscape mode (see Figure 3.14(a)) or in a portrait mode (see Figure 3.14(b)) depending on their orientation. Alternatively, pages can be zoomed by a factor of 2 (see Figure 3.14(c)). In addition to displaying the active page, the pocket mlb can also visualize the document hierarchy, and it allows the user to change the active page. Moreover, the user can control a telepointer, which is visible to all session members. In the future, we plan to extend this functionality such that the pocket mlb is able to control animations, and to create and modify private and shared annotations. Then, the handwriting recognition software provided by the mobile device can be employed for editing text objects.



(a) Landscape mode

(b) Portrait mode

(c) Zoomed portrait mode

Figure 3.14: Pocket mlb user interface

Aside from the pocket mlb, other collaborative service for handheld devices were developed at the University of Mannheim [160]. As Scheele et al. demonstrate, a voting tool can be used successfully for integrating exercises into lectures with a large number of students [216].

Because of the limited resources of mobile devices, the pocket mlb is designed with a client-server architecture where all costly tasks such as the management of the shared state and the scaling of graphical objects to the screen size of a handheld are executed by the server. The implementation of this architecture is based on the Wireless Interactive Learning Devices (WILD) system, which was developed at the University of Mannheim [160]. WILD connects the mobile clients over a Wireless LAN with the server (using TCP). In case of the pocket mlb, this server is a regular mlb instance that collects all requests from the attached pocket mlb instances, updates them, and forwards operations to and from other mlb instances via the communication model presented above. For remote mlb participants, the actions of pocket mlb instances therefore seem to come from the server mlb instance.

The human-computer interaction with both handheld devices and large screen systems is not based on the traditional input devices of mouse and keyboard. Instead, pen-based interaction is employed [165]. Due to the advances in handwriting recognition software, pen-based interaction is also popular with the notebook-sized Tablet PCs [170]. Besides transforming pen strokes into text, freehand drawings can also be interpreted as graphical objects, which can be classified, selected, grouped, and modified as proposed by Saund et al. in [215]. Moreover, pen-based interaction allows to control the application with so-called gestures: A gesture is a small graphical object that has to be drawn in a certain way. When the application detects a gesture, the corresponding command is invoked. Usually, gestures consist of a single stroke and are iconic so that it is easy to associate them with the underlying command [151]. For instance, the user could delete a graphical object by crossing it out, or select an object by cir-

cling it. The main advantages of gestures are that they are easy to use, and that they save the screen space that would otherwise be needed for menus and buttons. Additionally, commands could also be invoked by interpreting text objects as demonstrated by Rojas et al. in [213]. Integrating such features into the mlb and the pocket mlb remains an issue for future work.

3.7 The mlb as a Software Project

The mlb is the successor of both the AOFwb [149] and the dlb [80]. Its development was a joint project of the Universities of Freiburg and Mannheim that was sponsored from October 1999 to December 2001 in the project “ANETTE” [7] by the DFN-Verein [45]. Core elements of the mlb such as data and communication model, shared workspace, presentation animations, collaborative services, and pocket mlb were developed in the course of this thesis. Since Summer 2001, the mlb is used regularly in lectures and seminars at several Universities.

The mlb is implemented in C++ and Tcl/Tk [246]. It uses several libraries, which were also developed at the University of Mannheim (SMP, RTP/I, late-join, consistency control), plus the external libraries ImageMagick [124] to process images, and FreeType [72] to display TrueType fonts. Without these external libraries, the entire mlb comprises approximately 92,000 lines of code. The mlb is executable on both Linux and Windows platforms, and its code is published as open source under the GNU General Public License [88]. Downloads and a user manual are available at [259].

3.8 Conclusions

The multimedia lecture board (mlb) is a distributed interactive application for presenting and editing documents in teleconferencing scenarios. The mlb allows to open one shared and multiple private documents, supports various graphical objects, has a flexible user interface, and is able to import and export documents. Presentations can be structured with simple animations. The shared state of the mlb consists of a hierarchy of chapters, pages, and graphical objects. All messages are exchanged over IP multicast, the reliable transport protocol SMP, and the application-level protocol RTP/I.

Aside from the basic shared whiteboard functionality, the mlb provides several tools to coordinate the synchronous collaboration of multiple participants: A telepointer to visualize mouse movements, a hand-raising tool to serialize requests to speak, a chat tool for discussions, a voting tool for evaluation questions, and a tool to launch external applications. Moreover, awareness information about the session members and their actions is displayed.

In particular, the visualization of information in the participant list that was devised in this thesis proved to be effective without distracting the user. We also presented a lightweight version of the mlb for handheld devices (pocket mlb).

In the following chapters, the mlb is used as our main example application for the generic services and protocols developed in this thesis. In the next chapter, we discuss mechanisms for keeping the local state copies of distributed applications consistent.

Chapter 4

Consistency Control

Distributed interactive applications often employ a replicated architecture where each user runs an equal instance of the application, and each instance maintains a local copy of the application state (see Section 2.1). Local user actions updating this shared state need to be transmitted to all remote application instances so that these can modify their local state copies accordingly. For instance, when a participant modifies the position of a graphical object on a shared whiteboard page, the new coordinates have to be transmitted to all session members.

But without taking special precautions, it cannot be guaranteed that all users perceive the identical state after such operations have been exchanged, even if all operations were successfully delivered to all application instances. The main problem is that the transmission of an operation O is subject to a certain network delay: While O can be executed at once at the originating site so that the response time is zero, it takes some time to transmit it over a network to the other instances. Thus, the notification time for O is larger than zero. This causes two problems: First, for continuous interactive applications the exact execution time of an operation determines the resulting state. For instance, when changing the direction of a moving object, the object's new position depends on the point in time at which the change in direction was executed. If the distributed system does not take this into account, two sites with different network delays might not reach the same state.

Second, the propagation delay can result in different orderings of operations [162]. For instance, consider the situation depicted in Figure 4.1 where two participants of a shared whiteboard session change the color of a rectangle at almost the same time. Participant 1 changes the rectangle's color to black with operation O_1 , participant 2 to gray with O_2 . Given the network delays indicated in Figure 4.1, the same rectangle is displayed in gray for the first participant and in black for the second participant after both actions have been executed because the execution orders of O_1 and O_2 are different at the two sites.

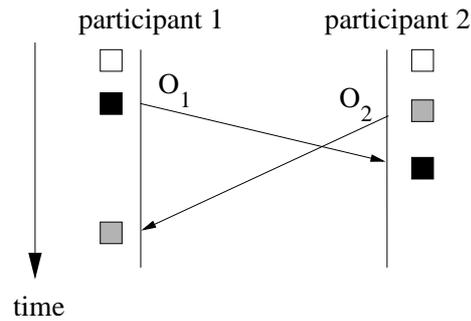


Figure 4.1: Network delay results in different operation orders

Such problems can be prevented when the application employs an appropriate *consistency control* mechanism to keep the local state copies of all participants synchronized. Simplified, consistency is achieved when all participants eventually see the same application state, e.g., both participants in Figure 4.1 finally see either a gray or a black rectangle.

In the following section, the task of such a consistency control algorithm is investigated and formalized. General design alternatives are discussed in Section 4.2, and in Section 4.3 existing approaches are analyzed. In the subsequent sections, the concepts of local lag, timewarp, and state request are presented and combined to a generic consistency control service, which is applicable to both discrete and continuous interactive applications. This service is evaluated by means of experiments with the *mlb* and other applications in Section 4.7. Specifically for discrete applications, possibilities to undo operations without endangering the application's consistency are discussed in Section 4.8. The chapter is concluded in Section 4.9.

4.1 Consistency Criteria

Before discussing different consistency control mechanisms for distributed interactive applications, possible relations among the operations that are issued in the course of a session are formalized, and several consistency criteria are defined. First, discrete applications are examined where the shared state is changed by user interactions only. Following, the considerations are generalized for continuous applications, which also allow state changes due to the passage of time.

4.1.1 Discrete Interactive Applications

The shared state of a discrete interactive application is changed by user interaction only. Depending on the communication model of the application, user actions are propagated either as events or delta states, which are applied to the current state, or the current state is replaced with a new full state (also see Sections 2.2 and 4.2). We denote events, delta states, and states

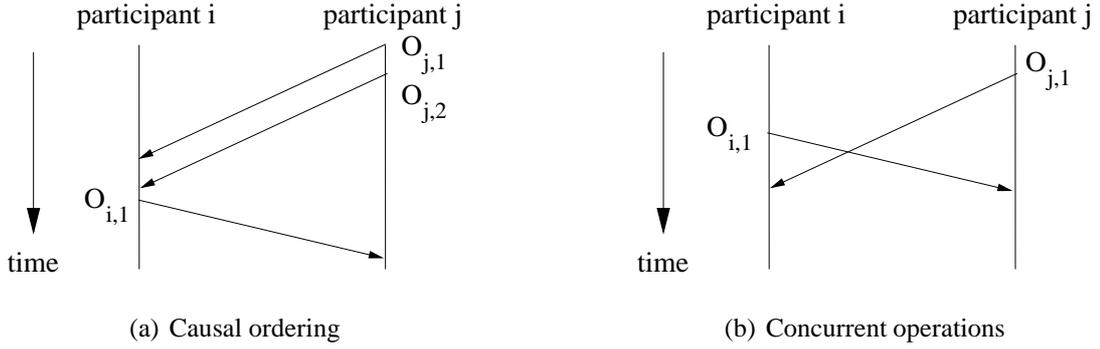


Figure 4.2: Relations among operations

as *operations*. In order to simplify the discussion, it is assumed that the shared application state is not partitioned. This does not impose a restriction since partitions are independent and each operation has a distinct target object. When applying a set of operations $\{O_k\}$ to the local state S_i of an application instance i , the resulting state S'_i depends on the execution order of these operations (see Figure 4.1) [162]. Thus, one important aspect of consistency concerns the order of operations.

Operations have certain relations, which need to be taken into account when establishing an order. One such relation is the *cause-effect order* [209, 162, 238]. For example, consider a participant first creates a rectangle with the operation O_1 and then changes its color with O_2 . Then O_2 must always be executed after O_1 . To express this, the **causal ordering** relation “ \rightarrow ” is defined by Lamport [146]:

Definition 4.1 Given two operations $O_{i,a}$ and $O_{j,b}$ generated at the sites i and j , then $O_{i,a} \rightarrow O_{j,b}$ iff (1) $i = j$ and $O_{i,a}$ was issued before $O_{j,b}$, or (2) $i \neq j$ and $O_{i,a}$ was executed before $O_{j,b}$ is issued, or (3) there exists an operation $O_{k,c}$ such that $O_{i,a} \rightarrow O_{k,c}$ and $O_{k,c} \rightarrow O_{j,b}$.

The different cases are illustrated in Figure 4.2(a) where $O_{j,1} \rightarrow O_{j,2}$ and $O_{j,2} \rightarrow O_{i,1}$. In case $O_{i,a} \rightarrow O_{j,b}$, $O_{j,b}$ is called *dependent* on $O_{i,a}$. And when neither $O_{i,a} \rightarrow O_{j,b}$ nor $O_{j,b} \rightarrow O_{i,a}$, $O_{i,a}$ and $O_{j,b}$ are **concurrent** (see Figure 4.2(b)). In case concurrent operations affect the same attributes of the state, we denote them as **conflicting**, e.g., O_1 and O_2 in Figure 4.1 are conflicting. Based on the causal ordering relation, the consistency criterion of **causality** is defined by Sun et al. [238]:

Definition 4.2 An application provides causality iff $\forall O_{i,a}, O_{j,b}$ where $O_{i,a} \rightarrow O_{j,b}$, $O_{i,a}$ is executed before $O_{j,b}$ at all sites.

The causal order defines only a partial ordering on all dependent operations, which is not sufficient to reach identical states at all application instances since concurrent operations are

not included yet. Thus, the consistency criterion of **convergence** is defined by Ellis and Gibbs [58]:

Definition 4.3 *Starting from the identical initial state S^0 , an application provides convergence iff $S_i = S_j \forall i, j$ after the same set of operations has been executed at all sites.*

While the causality criterion aims at the execution order of operations during user interactions, convergence concerns the application's state after all operations that were issued in a certain period of time have been exchanged and executed. This has two implications: First, convergence allows states to differ at a certain point in time when not all operations have been successfully received by all instances. Second, convergence does not demand that the operations are actually executed in any specific order. Convergence only guarantees that all sites determine the same state provided that they have received all operations required to do so.

Thus, we are also interested in whether the state of an application is *correct*, i.e., whether the state is the same as the one that will be reached if there is no propagation delay and all sites execute all operations in the order they are issued. In order to give a formal correctness criterion, a virtual perfect site P is defined such that P receives all operations $O_{i,a}$ immediately and executes them in the order they were issued (in case two operations $O_{i,a}$ and $O_{j,b}$ are issued at the same physical time, P can use an additional tie-breaker such as the sites' identifiers to determine a distinct order). The virtual perfect site P therefore always has the state a non-distributed application would have. **Correctness** can then be defined as follows:

Definition 4.4 *Starting from the identical initial state S^0 , an application provides correctness iff $S_i = S_P \forall i$ after the same set of operations has been executed at all sites.*

As for convergence, correctness applies only to those sites that received all necessary operations. In case a site received all operations, we denote a correct state as **complete**. Since correctness is the stronger criterion, establishing correctness implies that convergence is also achieved.

Now the question arises how it can be determined whether a local execution order of operations observes causality, convergence, or correctness. For this purpose, **state vectors** [58, 239] are defined based on Lamport's work on logical clocks [146]: A state vector SV is a set of tuples $(i, SN_i), i = 1, \dots, n$ where i denotes a certain application instance, n is the number of instances, and SN_i is the current sequence number of i . When i issues an operation O , SN_i is incremented by 1 (starting with 0), and the new state vector is assigned to O as well as to the state that results after applying O . Let $SV[i] := SN_i$. If we assume that the scenario given in Figure 4.2(a) starts with an initial state, the operations

would have the following state vectors: $SV_{O_{j,1}} = \langle (i, 0), (j, 1) \rangle$, $SV_{O_{j,2}} = \langle (i, 0), (j, 2) \rangle$, and $SV_{O_{i,1}} = \langle (i, 1), (j, 2) \rangle$.

Sun et al. show how causality can be tested with the help of state vectors [238]: Let SV_{O_i} be the state vector of an operation O_i issued at i and SV_j the state vector at site j the time O_i is received. Then O_i can be executed at site j when (1) $SV_{O_i}[i] = SV_j[i] + 1$ and (2) $SV_{O_i}[k] \leq SV_j[k] \forall k \neq i$. This means that prior to the execution of O_i all other operations that O_i causally depends on have been received and executed by j . If this is the case, O_i is *causally ready* and can be applied to the current state. Note that local operations are causally ready by definition, i.e., they can always be executed immediately which results in an optimal response time for the user. But if O_i is not causally ready, it needs to be buffered until all necessary operations have arrived and have been executed, increasing the notification time.

Causality establishes a partial order on all dependent operations. For discussing convergence and correctness, a global order is needed that also takes concurrent operations into account. As proposed by Sun et al. [238], such a global order can also be defined on the basis of state vectors:

Definition 4.5 *Let O_i and O_j be two operations generated at sites i and j , SV_{O_i} and SV_{O_j} the state vectors of O_i and O_j , and $sum(SV) := \sum_k SV[k]$. Then $O_i < O_j$, iff (1) $sum(SV_{O_i}) < sum(SV_{O_j})$, or (2) $sum(SV_{O_i}) = sum(SV_{O_j})$ and $i < j$.*

In the second case, when the state vector sums are equal, the site identifiers are used as tie-breakers, but other tie-breakers are also conceivable. Correctness can now be achieved when all operations are executed as defined by this global order. In [241], Sun et al. prove that if $O_i \rightarrow O_j \Rightarrow O_i < O_j$. Thus, an operation sequence ordered by $<$ observes causality as well. For the situation given in Figure 4.1, the global order would result in the same ordering of the “change color” events at both sites, leaving both copies of the rectangle either gray or black depending on the tie-breaker. However, this does not imply that all operations actually have to be executed in that global order. It is only required that the final state is identical to the state that would have been reached by executing the operations in that order. This means that it is perfectly legal for a consistency control mechanism to execute operations in different orders at the individual sites as long as the resulting states comply with the correctness criterion and with causality preservation.

Note that all consistency criteria discussed above do not consider the semantics of operations. Even when the application’s state fulfills the syntactic criteria of consistency or correctness, it might not meet the user’s expectations. In the example from above, the application does not know whether the rectangle should be gray or black from the user’s perspective. Instead, the rectangle’s color is determined by a formal ordering relation. This issue will be discussed in more detail in Chapter 5.

4.1.2 Continuous Interactive Applications

In addition to user-initiated state changes, replicated continuous applications allow state changes due to the passage of time, which do not require the exchange of update information. This implies that each instance of the application has access to a physical clock, which can be used to measure the progress of time [6]. In a real world system, these local clocks will never be fully synchronized, even when employing mechanisms such as the Network Time Protocol (NTP) [171] or GPS. The term *time* is used to refer to one specific reading of one specific local clock of a certain site. Due to limited synchronization, this reading may not be reached simultaneously (in the sense of a common reference clock) by all local clocks [184]. However, a possible offset between local clocks does not affect the application's consistency, which is shown in this section.

Because of time-related state changes, consistency in the continuous domain is not only about finding a correct sequence of operations and ensuring 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 looks as if the operations had been executed at the *correct point in time*. In the following, we define the terms consistency and correctness for continuous applications [161]. Please note that all consistency criteria and consistency control algorithms for continuous interactive applications are also valid in the discrete domain.

Let $S_{i,t}$ be the state that the application instance i holds at the time t , and O_{i,t^o,t^*} an operation issued by i at time t^o and that is to be executed at time t^* . For now, it is assumed that $t^o = t^*$ (in Section 4.4, we will discuss that it may make sense to set t^* to a value greater than t^o). It is assumed that the resolution of the local clock is sufficiently high so that no two operations can be issued at the same time at the same site, and therefore an operation is uniquely identified by i and t^o . If this is not the case, an additional local counter can be used to distinguish operations with identical values for i and t^o . The set of all operations O_{i,t^o,t^*} issued during a session is called *operation history* H . Based on the execution times t^* of the operations, the **physical time order** is defined by Mauve [155, 260]:

Definition 4.6 *Given two operations O_{i,t_i^o,t_i^*} and O_{j,t_j^o,t_j^*} , then $O_{i,t_i^o,t_i^*} < O_{j,t_j^o,t_j^*}$ iff (1) $t_i^* < t_j^*$ or (2) $t_i^* = t_j^*$ and $i < j$.*

As in the case of the state vector order, the sites' identifiers are used as tie-breakers. A **consistency** criterion can then be defined as follows [161]:

Definition 4.7 *A continuous distributed interactive application provides consistency iff at any time t for all sites i, j that received all operations $O_{k,t^o,t^*} \in H$ with execution times $t^* \leq t$ the states $S_{i,t}$ and $S_{j,t}$ are identical.*

As in the definition of convergence for discrete applications, sites that have not yet received all necessary information do not affect the question whether the application ensures consistency or not. But if at any time t all sites have received all operations, then the state at all sites at that time must be identical. It should be noted that consistency is completely independent of the synchronization of the distinct local clocks. The consistency criterion only requires that at the same reading of the local clocks the same state is reached and not necessarily at the same reading of a common reference clock.

Like in the discrete domain, the consistency criterion does not demand that the operations are actually executed in any specific order or that they are executed at their predefined time t^* . Again, we define a virtual perfect site P to determine whether the state is also correct, i.e., whether the state is the same as the one that would have been reached by executing all operations in the physical time order and at their given execution time. P receives all operations $O_{i,t^o,t^*} \in H$ by t^* at the latest and therefore always has the state that a single non-distributed application would have when processing H . **Correctness** is defined as follows:

Definition 4.8 *A continuous distributed interactive application provides correctness iff at any time t for all sites i that received all operations $O_{k,t^o,t^*} \in H$ with execution times $t^* \leq t$ the states $S_{i,t}$ and $S_{P,t}$ are identical.*

As for consistency, correctness is independent of clock synchronization and applies only to those sites that received all necessary operations.

Both the consistency and the correctness criterion make no statement about the time during that a site has not yet received all operations that it would need to calculate the current state. It is therefore possible that consistency-related, transient artifacts occur even if the correctness criterion is met by the application. For instance, let O_{i,t^o,t^*} change the direction of a moving object. In case the site j receives O_{i,t^o,t^*} after t^* , this object will be in a wrong position. This wrong position is also seen by the user. Thus, j needs to adjust the object's position in order to fulfill the correctness criterion. The application might do this by either changing the position abruptly (i.e., the object jumps to its new position), or by interpolating a new heading of the object toward the correct position, which would prolong the inconsistency but might be more pleasant for the user.

We denote a situation where a site j did not execute an operation O_{i,t^o,t^*} by the time t^* as **short-term inconsistency** at site j . This means that the states of at least two instances differ for a certain amount or time. Ideally, short-term inconsistencies should not occur during a session. But if such an inconsistency happens, the application needs to repair it with an appropriate consistency control mechanism in order to comply with the consistency or correctness criterion.

Causality is another important criterion, which was introduced for discrete applications and makes sure that dependent operations are executed in the correct order. Observing causality is also important in the continuous domain: For instance, it prevents that an application tries to execute an operation on an object that does not exist yet. Moreover, it ensures that remote users can understand the flow of operations. But at the same time, if $O_{i,t_i^o,t_i^*} \rightarrow O_{j,t_j^o,t_j^*}$ and site k receives O_{j,t_j^o,t_j^*} first, it needs to delay O_{j,t_j^o,t_j^*} until O_{i,t_i^o,t_i^*} arrives. This might cause a short-term inconsistency for both operations. An application might therefore decide to observe causality only for certain operations in order to prevent short-term inconsistencies (e.g., when the target object does not exist yet).

4.2 Design Considerations: Hard State vs. Soft State

Hard state and soft state are two fundamental synchronization approaches, and a distributed system can adjust its own mechanism between them. In the following, the effects of both approaches on the application's robustness and on the propagation delay of state changes are examined, and their overhead and complexity are discussed.

The *soft state* approach was first introduced by Clark in [32] and is used by a number of protocols such as RTP [219], RSVP [16], and SAP [108]. It works as follows: One or more session members periodically announce the current state of the application. Participants discover new or updated parts of the shared state by these announcements, i.e., there are no explicit notifications for state changes, and all messages are exchanged in the form of states. The time span between two announcements is called report interval. An application instance saves the time of the last announcement for each object of the shared state, and in case it does not receive any data for several intervals, the concerned object times out and is deleted from the application's state. All messages are transmitted unreliably, and packet loss is repaired by the next state transmission.

The main advantage of the soft state approach is its low complexity since all possible failures are handled implicitly by the periodic state transmissions, and there is no need for additional mechanisms to repair packet loss, to initialize late-joining participants, and to repair inconsistencies due to disordered or outdated operations. This simplicity also makes the application very robust. However, depending on the report interval and the packet loss rate, it might take a long time until state updates are propagated to all session members. This might result in frequent short-term inconsistencies. Also, repeated packet loss might even cause false time-outs and the removal of objects that still exist. Moreover, due to the lack of feedback, the originator of a state update does not know if and when the other session members received his update. Another severe drawback is the potentially high network load generated by the periodic transmissions of the complete shared state, especially for complex distributed inter-

active applications that tend to have large states. For instance, consider an mlb session where the same slides would be transmitted again and again.

The alternative to soft state is the *hard state* approach where new objects, state updates, and the deletion of objects are propagated explicitly and immediately. State changes and deletes are encoded preferably in the form of events (or cues) so that only essential information is distributed. All messages are transmitted reliably, and the application or the underlying protocols need to employ a mechanism to detect and repair packet loss. Moreover, explicit mechanisms are needed in order to handle late-join situations as well as disordered and out-dated operations. Since the application needs to address all possible error situations, and all these mechanisms need to interoperate, the hard state approach often results in a complex architecture [78, 83, 142, 154, 249, 259]. But when compared to the soft state approach, the propagation delay for state updates is expected to be much less because of the instantaneous and reliable announcement of operations. This increases the responsiveness of the application so that user interaction is more direct, and at the same time it decreases the probability for short-term inconsistencies. Furthermore, since only data is transmitted that is actually needed by remote application instances to update their state, the resulting network load is expected to be considerably less when compared to soft state.

The performance of the soft state approach can be improved by adopting elements from the hard state approach. As Ji et al. show in [129], the propagation delay and the probability for short-term inconsistencies can be lowered by announcing state changes explicitly in addition to the periodic state transmissions. Alternatively, newer information can be reported more frequently than older information [66, 203]. But for distributed interactive applications, the biggest drawback of soft state approaches is the high network load. In the remainder of this chapter, we therefore focus on the hard state approach as it is employed by the mlb, TeCo3D, Instant Collaboration, and the Spaceshooter game (see Section 2.4 and Chapter 3). As mentioned above, this implies that the application has to use explicit error recovery mechanisms. Reliable transport of operations is beyond the scope of this thesis, and we assume that the application integrates an appropriate reliability mechanism (see Sections 2.3.3 and 7.2). In the next sections, mechanisms to achieve consistency for discrete and continuous applications are discussed.

4.3 Related Work

The state of a distributed interactive application can change either because of the passage of time or because of user interactions, which are distributed as operations. Operations might be received in different orders or after their scheduled execution time. Thus, the application

needs to employ a mechanism that establishes consistency according to the criteria discussed in Section 4.1.

Consistency control mechanisms can be classified as either pessimistic or optimistic. *Pessimistic* mechanisms prevent concurrent operations and allow only a single participant to issue state changes at a certain point in time. This can be done by a locking or a floor control algorithm where exclusive access rights have to be obtained before the participant is allowed to change the state of an object [91, 178, 138, 240]. Alternatively, operations can be serialized through a distinct controlling instance: All operations changing the state of an object are first sent to a specific site, which orders operations and distributes them to all session members [162]. This approach is used by the collaborative virtual environment MASSIVE-3 [92]. Pessimistic approaches therefore reduce the consistency control challenge to the ordering of operations that origin from a single source. Even though this technique is very effective, it has a major drawback: Because of the explicit or implicit exchange of access rights, natural collaboration among participants is restricted. And for continuous applications, the problem of outdated operations remains.

Optimistic approaches, on the other hand, allow users to issue concurrent operations and seek to repair any short-term inconsistencies that emerge in an efficient way [58, 238]. While these mechanisms support natural collaboration among users, existing approaches tend to be complex, domain-dependent, and expose the user to frequent short-term inconsistencies. In the following, operational transformation, serialization, object duplication, and dead reckoning are examined.

Operational transformation was first introduced by Ellis and Gibbs in [58] and has evolved into one of the most important consistency control mechanisms for discrete applications [237, 238]. The basic idea of operational transformation is that an application instance i immediately executes all operations that are causally ready. But in order to establish correctness, each remote operation $O_{j,b}$ is transformed into another operation $O'_{j,b}$ before it is executed so that $O'_{j,b}$'s effect on the current state S_i is the same as the effect of $O_{j,b}$ on S_j at the time $O_{j,b}$ was issued. This means that i changes all operations such that the resulting state is identical to the one that would be calculated by the virtual perfect site P . One important property of this scheme is that it considers relative operations (see Section 2.2). For instance, let $O_{j,b}$ be an operation deleting the two characters “bc” in a text string “abcd”. The application would encode this operation as “delete 2 characters starting at index 1”. But due to concurrent operations, i 's state might change until $O_{j,b}$ is received, e.g., when another user inserts a character “x” at the beginning of the text string so that the new state of i would be “xabcd”. The operational transformation would then adjust the index encoded in $O_{j,b}$ such that the new position of “bc” is considered before executing $O_{j,b}$ so that $S_i = “xad”$ is reached. This transformation scheme requires that the application stores the operation history, and that

all operations carry information about the state that was valid when they were issued (e.g., in the form of a state vector). It also requires clear and simple semantics of the application so that the transformations can be computed automatically.

Operational transformation is a robust consistency control mechanism, which allows a site to repair short-term inconsistencies on the basis of local information. It works best for applications employing relative state changes (e.g., text editors). But the transformation rules tend to be very complex [238], and their correctness is difficult to verify. Moreover, it is not granted that appropriate transformation rules can be found for all applications, especially for continuous applications.

With *serialization* [133, 241], all application instances execute all operations in a distinct order, e.g., in the total order on the basis of state vectors as given in Definition 4.5. Since a site i does not know which operations will be received, it cannot anticipate this order. Instead, an operation is executed as soon as it is causally ready. As for operational transformation, all operations are stored in a history H . In case i receives an operation O_j that would violate the total order when appended to H , a short-term inconsistency has occurred, and the correct order of the operations needs to be restored. This can be achieved by the following “undo/do/redo” algorithm presented by Sun et al. in [238]: All operations $O_k \in H$ with $O_k < O_j$ are undone, then O_j is executed, and all operations are redone that were undone in the first step. Afterwards, the history of i is identical to H of the virtual perfect site. Thus, serialization achieves syntactic correctness as defined above¹.

Like operational transformation, serialization is able to repair inconsistencies on the basis of local information. Moreover, local operations can be applied immediately to the current state resulting in a low response time for the user. However, it requires that the application is able to undo all operations, which is especially difficult for continuous applications like the multi-player game presented in Section 2.4.2. For example, consider a situation where the effects of a shot need to be undone (also see Section 4.8).

Another optimistic consistency control mechanism is *object duplication* (or multi-versioning) as proposed by Sun et al. in [235, 236] for shared graphic editors. The basic idea is to handle two conflicting operations O_i and O_j that change the same attributes of the shared state by duplicating the concerned object and by applying O_i to one duplicate and O_j to the other. Thus, multiple versions of the same object might be created and displayed simultaneously. The different versions of an object are generated such that they incorporate the maximum set of non-conflicting operations. The participants can either select a certain version or keep them

¹Note that the serialization scheme described here is not related to the concept of serialization for databases [63]: Here, the application’s state depends on the execution order of operations, and the user is not shielded from concurrent modifications.

all. Object duplication establishes causality and convergence in the sense that all application instances create the same object versions, but obviously correctness is not achieved. Drawbacks to this approach are a complex management of multiple versions of the same object, and a confusing effect for the user if too many different versions exist (e.g., when subsequent operations create successive versions). Furthermore, for continuous interactive applications this approach seems to be problematic. For example, consider a networked computer game in which object duplication would result in two representations of the same object.

Dead reckoning is commonly employed for consistency control in continuous applications such as distributed virtual environments and battlefield simulations [229, 224]. It uses a combination of state prediction and state transmission: Each object of the shared state has a single controlling application instance, e.g., for a plane the instance of the pilot. State prediction means that all sites are able to calculate state changes caused by the passage of time locally [188], e.g., the path of a flying plane. Only the controlling instance is allowed to issue operations, e.g., when the pilot changes the direction of his plane. These operations modify the state of the affected object such that it differs significantly from the predicted state. Thus, the controlling instance has to notify all participants by propagating the updated state. Following the soft state approach, operations are transmitted unreliably as states, and packet loss is repaired by periodic announcements. Each site is only responsible for the objects it controls. For example, a collision between two objects that an application instance does not control is not discovered by that instance. Instead, the controlling site transmits an update when it detects the collision.

In order to apply the consistency and correctness criteria to dead reckoning, we say that a site j has received an operation O_{i,t^o,t^*} if it has received a state including the effect of that operation. With this clarification it can be shown that the consistency criterion is fulfilled by dead reckoning: If at time t two sites i and j have received sufficient state updates so that they know of all operations with $t^* \leq t$, they will have the same state. However, dead reckoning cannot guarantee correctness because it transmits states instead of events, and the transmissions are unreliable. Thus, it is not possible for an application instance that receives a state update to determine how that state came to be. This in turn may lead to an incorrect state at the controlling site. For example, consider a situation where two planes A and B approach each other. At some point in time, the instance controlling plane A receives a state update for B that puts B past A. If some preceding updates were lost, there is no way for the controlling site of A to determine how B got to this position and whether the two planes collided or not. The virtual perfect site P will not have that problem since it receives all operations reliably and in time. Such problems may not only occur due to packet loss, but also when the controller of an object receives information about the interaction between two objects so late that it was not taken into account when calculating its own state.

The main advantage of the dead reckoning approach is its low complexity, which originates from the concepts of a single controlling application and the soft state approach. Thus, as Lin and Shab prove in [150], dead reckoning is highly scalable and is employed successfully for large simulations and distributed virtual environments. At the same time, dead reckoning does not provide correctness, might cause a high network load if states are large, and limits user interaction by the single controlling site.

As we have seen, all consistency control algorithms discussed above have some severe drawbacks or are applicable to a certain class of applications only. In the following sections, a generic consistency control service is proposed, which was developed in this thesis. It can be employed for all discrete and continuous applications and uses a combination of algorithms to ensure correctness: First, *local lag* reduces the number of short-term inconsistencies. Second, *timewarp* repairs inconsistencies exceeding the time span covered by local lag. Third, *state requests* repair inconsistencies exceeding the time span covered by timewarp. The common design principle of these algorithms is that all application instances execute all operations in the same order (and at the correct execution time in the case of continuous applications).

4.4 Local Lag

The task of a consistency control algorithm is to handle operations O_{i,t^o,t^*} that are received after their scheduled execution time t^* or that are received in a wrong order. Such inconsistencies are caused by the network delay: While an operation O_{i,t^o,t^*} that is issued by the participant i at the time t^o can be executed by i in time even when $t^* = t^o$, all remote instances will receive that operation later than t^o due to the network delay. Figure 4.3 (a) depicts such a situation for the operation $O_{i,1,1}$ with $t^* = t^o$. In case a site j receives O_{i,t^o,t^*} at $t > t^*$, a short-term inconsistency has occurred at site j , either because the local state has changed in the meantime due to the passage of time or because concurrent operations might have been received and executed in an order that is wrong when considering O_{i,t^o,t^*} . In both cases, the current state $S_{j,t} \neq S_{j,t^*}$, and applying O_{i,t^o,t^*} to $S_{j,t}$ would raise an inconsistency. In Figure 4.3 (a), participants j and k experience such a short-term inconsistency.

Even if it is assumed that a short-term inconsistency will eventually be repaired by a mechanism ensuring the consistency or the correctness criterion, it has a negative impact: First, the user perceives an inconsistent state and might act on its basis. Second, the application has to calculate the correct state, which might consume significant computational resources. Finally, when the application displays the corrected state, it might be considerably different from the (wrong) state visible before, causing artifacts such as jumping objects. Thus, it is desirable to prevent short-term inconsistencies.

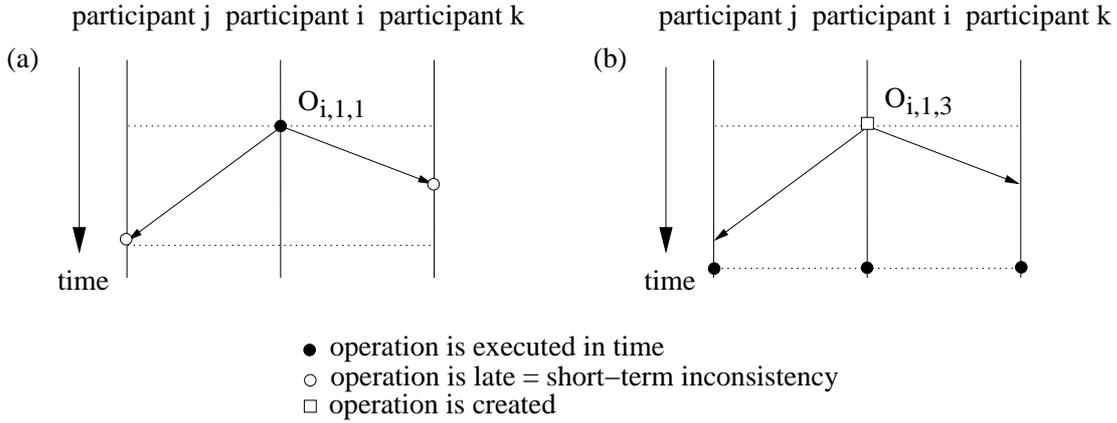


Figure 4.3: Equalization of the operation delay

The length I of a short-term inconsistency depends on the offset between the local clocks of the originating and receiving sites, the transmission delay, and on how much earlier an operation was issued (t^o) than executed (t^*). It is calculated for site j and operation O_{i,t^o,t^*} as follows:

$$I_j(O_{i,t^o,t^*}) = d(i, j) - (T_i^* - T_j^*) - (t^* - t^o), \quad (4.1)$$

where T_k^* denotes the reading of a common reference clock at the time the local clock at site k reaches t^* , and $d(i, j)$ denotes the (unidirectional) network delay from i to j . If I_j is negative, then the operation has not caused a short-term inconsistency. The longer a short-term inconsistency persists, the more distracting it is for the users, and the more likely it is that users will issue operations on basis of a state suffering from a short-term inconsistency.

Cristian proposes to prevent short-term inconsistencies by shifting the execution time t^* of an operation into the future [36], i.e., $t^* > t^o$ for O_{i,t^o,t^*} . The time span $t^* - t^o$ gained is then used to distribute O_{i,t^o,t^*} to all application instances. In the optimal case, distribution is completed before t^* is reached, allowing all instances to execute O_{i,t^o,t^*} at the correct time. For instance, in Figure 4.3 (b) both participants j and k receive $O_{i,1,3}$ before its execution time is reached and simply buffer the operation until then. As long as this is the case for all operations, they will be executed in the physical time order (see Definition 4.6), and the resulting local states will be correct. Because of the artificial delay introduced by the originating site, Mauve denotes this approach as *local lag* [155]. Local lag is related to the bucket synchronization mechanism of the multi-player game MiMaze [77, 47].

The concept of local lag is not only useful for continuous interactive applications where it prevents that operations are outdated immediately after they were issued, but is also beneficial for discrete interactive applications. Here, the time span gained is used to sort concurrent operations in the correct order (e.g., in the state vector order, see Definition 4.5) before their

execution. Thus, local lag prevents a complex reordering of concurrent operations after their execution as would be the case with the serialization mechanism (or a costly transformation of operations).

Choosing the right value for the local lag is not easy. On the one hand, a large time span $t^* - t^o$ is desirable to increase the time for distributing an operation to all destinations so that short-term inconsistencies can be prevented. At the same time, $t^* - t^o$ also represents the response time of the application (see Section 2.3.1): The local user issues an operation at t^o , but its effects become only visible at t^* . If this response time exceeds a certain threshold, the user will notice the artificial delay, and the application might feel unnatural. Thus, there exists a trade-off between the goals of achieving a low response time and minimizing the number of short-term inconsistencies [161]. This trade-off was first identified by Mauve [155] and is reflected in Equation 4.1.

The other factor that determines whether an operation O_{i,t^o,t^*} causes a short-term inconsistency at the receiver is the clock offset between the local clocks of the sender i and the receiver j that is given by $T_i^* - T_j^*$ in Equation 4.1. This offset might even compensate the network delay $d(i, j)$ between i and j so that a response time of zero could be achieved for $t^* = t^o$ without raising a short-term inconsistency. However, this works only in one direction. As soon as the previous receiver j becomes the sender, short-term inconsistencies will occur because of the clock offset, which now prevents that any operation arrives at i in time. Thus, the clock offset cannot be used to prevent short-term inconsistencies.

In [155] and [161], Mauve investigates the trade-off between low response time and low probability for short-term inconsistencies: A compromise has to consider the expected average network delay and the maximum tolerable response time for user actions. The average network delay depends on the locations of the participants, the quality of the network path that the exchanged operations traverse, and the employed network protocols. Typical network delays are: Less than 1 ms for a LAN, 20 ms within a European country, 40 ms within a continent, and 150 ms for a world-wide session. Depending on the application scenario and the operation issued, the maximum tolerable response time lies between 50 ms and 300 ms [155, 187, 223, 251]. For a given application, it is advisable to conduct a series of psychological experiments to investigate an adequate value. In the ideal case, the tolerable response time is larger than the expected network delay, so that a local lag value can be selected that suppresses most short-term inconsistencies. In case the clocks of the participating sites are not synchronized, the average clock offset should also be included in the local lag value.

The main advantage of local lag is that the probability for short-term inconsistencies is reduced significantly, so that the execution of a heavy-weight repair mechanism is limited to

exceptional situations where the local lag is not sufficient for distributing an operation to all session members in time, e.g., when the network is congested. This is achieved by a slight increase in the response time that might not even be noticeable for the user. A positive side-effect is that varying network delays are leveled for the remote sites: Park and Kenyon find that users can adapt better to the effects of a higher but constant notification delay than to one that changes continuously [189].

Implementation of the local lag concept is straightforward and can be done independent from a specific application in the form of a generic service as we demonstrate in [260] (see Section 7.4.1): The service maintains a queue for all operations that is sorted by the physical time ordering relation. If the execution timestamp t^* of an operation O_{i,t^o,t^*} is reached (and O_{i,t^o,t^*} is causally ready), the service notifies the application to execute O_{i,t^o,t^*} . Note that the local lag concept leads to a new programming paradigm. Traditionally, the functionality triggered by a local event is as follows: Execute the event and display the new state, then create and distribute the corresponding operation. With local lag, this changes to: Create and distribute the operation, insert the operation together with received operations into the local lag queue, wait until its execution time is reached, then calculate and display the new state. Thus, with local lag remote and local operations are no longer distinguished once an operation was created and enqueued.

Even though local lag can reduce the number of short-term inconsistencies significantly, they can still occur when the time span gained is not sufficient to distribute an operation to all participants. This might happen if the regular network delay between session members that are far apart is too high, the network suffers from congestion and/or repeated packet loss, or when the application allows only short response times for the local user. Indicative of a possible inconsistency is the receipt of an operation O_{i,t^o,t^*} with $t^* < T_C$ where T_C is the current time of the receiver. In the next section, timewarp is presented as an efficient repair mechanism for those inconsistencies that cannot be prevented by local lag.

4.5 Timewarp

With the timewarp algorithm, which was first described by Jefferson [128], an application instance can repair inconsistencies that are not covered by local lag on the basis of a local operation history. It is related to the serialization approach in the sense that each site executes all operations in a distinct order (see Section 4.3). In the following, first the basic timewarp algorithm is presented and then various improvements for continuous and discrete applications are proposed.

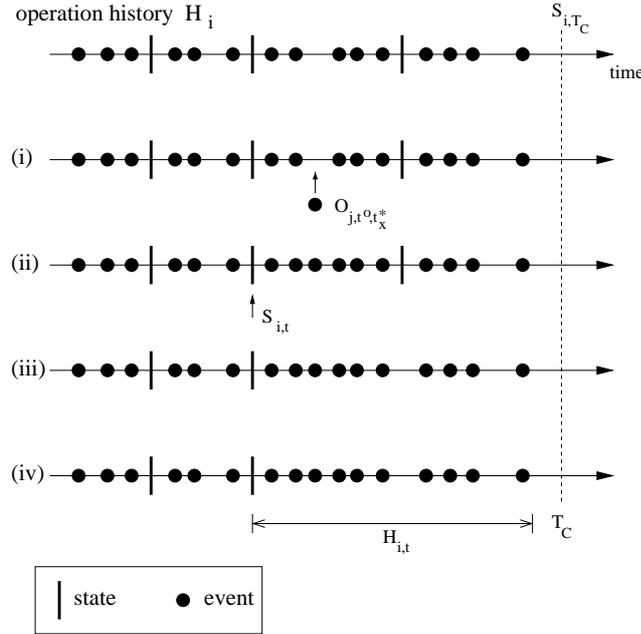


Figure 4.4: The basic timewarp algorithm

4.5.1 The Basic Timewarp Algorithm

The timewarp algorithm requires each application instance i to store all local and remote operations in a history H_i , which is ordered by a certain ordering relation (e.g., by the physical time order or the state vector order). Moreover, i periodically saves a snapshot of the current state S_{i,T_C} in H_i where T_C denotes the current time.

In case i receives an operation O_{j,t^o,t_x^*} with the execution time $t_x^* < T_C$, a short-term inconsistency has occurred, which is repaired by a *timewarp* as described by Mauve [155]: First, O_{j,t^o,t_x^*} is inserted into H_i at the correct position (see Figure 4.4 (i)). Then, the first state $S_{i,t} \in H_i$ is determined with $S_{i,t} < O_{j,t^o,t_x^*}$, and the state of the application instance i is set back to $S_{i,t}$ (ii). All states $S_{i,t'}$ with $S_{i,t} < S_{i,t'}$ are possibly inconsistent and are therefore deleted from H_i (iii). Finally, the operation sequence $H_{i,t}$ with all operations that follow $S_{i,t}$ is executed in a fast-forward mode until T_C is reached and the application can continue processing at normal pace (iv). More formally, $H_{i,t}$ contains all operations $\{O_{j,t^o,t^*} \in H_i \mid S_{i,t} < O_{j,t^o,t^*} \wedge t^* \leq T_C\}$ and is ordered according to t^* . To avoid a distracting effect, only the final state of the application should be visible to the user.

Instead of deleting outdated states $S_{i,t'}$ as described above, they could also be updated while calculating the new current state. This concept is also known as trailing states [37, 161] and has the advantage that H_i will contain more states as possible starting points for future timewarps.

The timewarp algorithm ensures that all application instances execute all operations in the same order and therefore fulfills the correctness criterion of Definition 4.4 as proven by Mauve et al. in [161]. But the timewarp algorithm also has two major drawbacks: First, the application must support the execution of operations in a fast-forward mode, which might be difficult to implement for some continuous applications and which might consume a significant amount of processing resources. Second, storing the history of exchanged operations together with (potentially large) state snapshots requires adequate memory space. In the following, several improvements are proposed, which address these drawbacks.

4.5.2 Round-Based Timewarp

The performance of the timewarp algorithm can be improved considerably when it is executed round-based: Instead of treating each operation individually, the application collects all operations during a certain period of time T . The operation with the smallest execution time t_x^* that has arrived during T and that is due for processing then determines the starting state $S_{i,t}$ and the sequence $H_{i,t}$ for calculating the new current state. $S_{i,t}$ is either the state calculated in the last round or an older state so that a timewarp is required. Thus, at most one timewarp per T is executed, independent of the number of late-arriving operations. As we show in [161], this reduces the complexity of the timewarp algorithm from $O(n^3)$ to $O(n^2)$ where n is the number of participants. Moreover, it prevents that a timewarp delays the execution of subsequent operations, which might trigger additional timewarps.

The round-based timewarp is well-suited for continuous applications that often update the state displayed to the user in a certain frequency (e.g., 25 updates per second). T can be chosen such that this update frequency is matched when considering the estimated processing time for one timewarp. If T is sufficiently small, the user will not be able to notice that operations are not executed at the exact time t^* . In Section 4.7.3, experimental results for this approach are given.

Round-based timewarp is also applicable for discrete applications. Additionally, it might be the case that the local state of a site does not need to be changed during an update interval T , either because there are no new operations or because a late operation nevertheless does not trigger a timewarp. The latter case is discussed in the next section.

4.5.3 Filtered Timewarp for Discrete Applications

Since a timewarp can be costly in terms of processing power, it is desirable to execute a timewarp only when absolutely necessary. In this thesis, an approach was devised to reduce the number of timewarps significantly with the help of application-level knowledge [260]: In

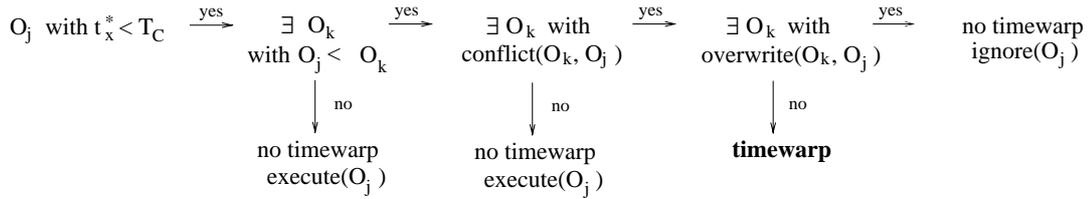


Figure 4.5: Decision algorithm for timewarp with filtering

the discrete domain, there exist quite a few cases where a timewarp is not required, and a site i can either ignore a late-arriving operation O_{j,t^o,t_x^*} (short: O_j) or execute it immediately without a timewarp. The algorithm deciding on how to treat O_j is depicted in Figure 4.5: First, it is checked if there exist executed operations O_{k,t^o,t_x^*} (short: O_k) with $O_j < O_k$. If not, O_j is the last operation so far and can be executed without endangering correctness and without a timewarp. Second, in case there is a set of operations $\{O_k \mid O_j < O_k\}$, for each O_k it is checked if it *conflicts* with O_j . As defined above, two operations are conflicting if they change the same aspect of the shared state.

In order to decide whether two operations are conflicting, the application has to provide an appropriate function $\text{conflict}(O_k, O_j)$. For instance, events changing the objects on a shared whiteboard page conflict if they target the same object and the same attribute (e.g., color, size, position). Figure 4.6 outlines this function for the mlb. The mlb maintains an operation history for each container object (i.e., page or chapter) so that operations that target element objects of a container and also affect the state of the container can be discovered as conflicting (e.g., operations changing the display order of graphical objects). The conflict function for the mlb decides that two operations are conflicting if: (1) They target the same object and change aspects of the object that are identical or dependent on each other, (2) the two objects share the same parent and the stacking order or visibility information of the objects is changed, (3) both objects change their parent to the same new parent, or (4) the operations change the active object (e.g., the single page currently displayed is marked as active). It is important to realize that an application may start with a very simple conflict function returning `true` in almost all situations. Later on, this function may be improved to prevent more timewarps. For the mlb, even a relatively simple conflict function already reduces the number of timewarps significantly.

If no conflicting operation O_k is discovered, O_j can be executed immediately without a timewarp (see Figure 4.5) since the execution order of O_j and the sequence $\{O_k\}$ has no impact on the resulting state. Finally, if there exists at least one conflicting operation O_k , it is checked if it *overwrites* the effects of O_j . This means that the state that would be reached after executing O_j and O_k is identical to the state that is reached by executing O_k only. If there is at least one such O_k , then O_j can be ignored, and no timewarp is necessary. As for determining conflicting operations, the application has to provide an appropriate function $\text{overwrite}(O_k, O_j)$

conflict(O_k, O_j)

1. $\text{object}(O_k) = \text{object}(O_j)$
 - O_k is state or O_j is state
 - O_k is event with type delete
 - O_k and O_j are events
 - $\text{type}(O_k) = \text{type}(O_j)$
 - $\text{type}(O_k)$ and $\text{type}(O_j) \in \{\text{raise, lower}\}$
2. $\text{object}(O_k) \neq \text{object}(O_j)$ and $\text{parent}(O_k) = \text{parent}(O_j)$
 - O_k and O_j are state
 - $O_{j/k}$ is state and $O_{k/j}$ is event with $\text{type} \in \{\text{raise, lower, change parent}\}$
3. $\text{object}(O_k) \neq \text{object}(O_j)$ and $\text{parent}(O_k) \neq \text{parent}(O_j)$
 - O_k and O_j are events with types = change parent and $\text{target}(O_k) = \text{target}(O_j)$
4. O_k and O_j are events with types = set active

Figure 4.6: Conflicting operations**overwrite**(O_k, O_j)

1. $\text{object}(O_k) = \text{object}(O_j)$
 - O_k is state and O_j is event
 - O_k is event with type delete
 - O_k and O_j are events
 - $\text{type}(O_k) = \text{type}(O_j)$
2. O_k and O_j are events with types = set active

Figure 4.7: Overwriting operations

deciding whether one operation O_k overwrites another operation O_j . Figure 4.7 specifies the overwrite function for the mlb. For instance, let O_j and O_k be events changing the color of an object to gray and black, respectively. Then the object's color will be black after both O_j and O_k have been executed. Even if a timewarp were executed, the user would miss the fact that the object was gray for a certain period of time, since only the final state at T_C is displayed.

Note that the set of overwriting operations is a real subset of the set of conflicting operations, meaning that not all conflicting operations are overwriting operations. For example, let O_j be a state creating a new object on a whiteboard page p and O_k be an event changing the stacking order of another object on p . Then O_j and O_k conflict regarding the display order of all objects belonging to p , but O_k does not overwrite O_j .

We denote this stepwise testing of the necessity of a timewarp as *filtered timewarp* algorithm. It is able to significantly reduce the number of timewarps for a discrete application. In Section 4.7.1, promising experimental results are presented for the mlb. However, its use is more difficult for continuous applications due to the fact that here operations are valid only at their given execution time. The execution of a late-arriving operation O_j therefore generally requires a timewarp. Ignoring O_j is possible, but deciding whether the effect of O_j would have been completely overwritten is generally more complex than in the discrete domain.

4.5.4 Restricting the Size of the Operation History

Besides its computational complexity, the timewarp algorithm also might have a high demand on memory space for storing the operation history. The basic timewarp algorithm requires that all operations of a session are kept indefinitely. Moreover, depending on the application,

in particular the periodic state snapshots might be large. Thus, we devised three mechanisms to limit the size of the operation history: First, long operation sequences may be replaced by semantically equivalent operations. Second, operations that are known to have been successfully delivered and executed by all session members can be deleted from the history [83]. Third, operations that are received exceedingly late are handled by a different repair mechanism [260].

Unlike states and events, cues do not need to be stored in the operation history since they do not change the application's state or are followed by an event. In some cases, the application might also be able to find a shorter semantical representation for an operation sequence that occurred in the session without losing vital information. For instance, when creating a free-hand line on an mlb page, each point added by the user is propagated immediately in order to achieve a good responsiveness (see Section 2.3). But for later usage, this operation sequence can be substituted with a single state containing the complete freehand line. This shortens the history significantly.

The second possibility to reduce the size of the operation history is to remove an operation as soon as it is announced to be executed by all session members [83]. In this case, a site can assume that no concurrent operations are under way, which could trigger a timewarp. Such announcements could either be exchanged explicitly or be derived from the state vectors of operations: Let site i receive an operation O_j from site j with state vector SV_{O_j} . Then $SV_{O_j}[k]$ denotes the sequence number of the last operation that j received from k , i.e., j implicitly acknowledges all older operations of k . Under the condition that O_j is causally ready, there can be no short-term inconsistencies triggered by operations from j that are concurrent to operations from k with $SN_k \leq SV_{O_j}[k]$. This means that from j 's perspective i can remove all operations of k with $SN_k \leq SV_{O_j}[k]$ from its history. Once i has received similar acknowledgments from all participants, old operations can be deleted. This process can be sped up by periodically exchanging status messages with the current state vectors of a session member.

The third approach is to restrict the time span T_H covered by the operation history H_i by deleting all operations from H_i that are older than $T_C - T_H$ [260]. This has two implications: (1) The range of the timewarp is limited, meaning that an inconsistency caused by an operation O_{j,t^o,t_x^*} arriving exceedingly late with $t_x^* < T_C - T_H$ cannot be repaired by time-warping. Extending T_H will increase the probability that an inconsistency can be repaired by a timewarp but will consume more memory space. T_H should therefore be chosen by the application in order to fine-tune this trade-off. For the mlb, T_H is initially set to 180 seconds and can be changed by the user. Such a large value for T_H will make it very unlikely that the operation history is insufficient. (2) The history does not start from the beginning of the session. In order to execute a timewarp, the history needs to contain at least one state $S_{i,t}$

for each object with a timestamp $t = T_C - T_H$. In case the trailing state strategy is not used by the application (see Section 4.5.1), a new state for each object should be inserted into the history at least every $\frac{T_H}{2}$. If a finer granularity is needed, the application might insert states more frequently.

The last two mechanisms to restrict the size of the operation history bear the risk that a short-term inconsistency cannot be repaired by timewarp anymore. For instance, a new participant might late-join an ongoing session, obtain the current state, and issue an operation in parallel to the operation of another member (also see Chapter 6). In this case, the second approach might fail since participants clearing their history have no knowledge about the late-joining application instance. The third approach fails when an operation is received extremely late, e.g., when recovering from a partitioned network. If the local repair of a short-term inconsistency is not possible, the application has to switch to a different mechanism and request the concerned state from the other session members. Such a state request mechanism is presented in Section 4.6.

4.5.5 Timewarp as a Generic Service

The timewarp algorithms described above were realized as an application-independent service in this thesis [260] (see Section 7.4.1). One task of the service is to manage the operation history once operations have passed through the local lag service and were executed by the application. The history is sorted, older operations are deleted, and new states are inserted as described above. In case a short-term inconsistency occurs, the timewarp service decides on the necessary actions as depicted in Figure 4.5 by applying the `conflict` and `overwrite` functions provided by the application. Should a timewarp be necessary, the service delivers the starting state $S_{i,t}$ and the operation sequence $H_{i,t}$. The application executes those like regular operations.

4.6 Requesting Full States

In case a short-term inconsistency cannot be repaired locally, the concerned application instance i needs to request the current state from another session member in order to restore the correctness of the application. A state request raises two questions: First, who should send a state as reply, and second, how is it guaranteed that this state received is correct? A state request mechanism that addresses these questions was developed in this thesis [260].

The first problem is a typical task for a multicast feedback mechanism: Because of the application's replicated architecture, there is no preselected server and a state request can be served

by a number of session members. But only one answer (feedback) is actually needed. Since it is a priori unknown which sites are able to respond to a request (e.g., an instance might suffer from a known inconsistency), state requests are distributed via multicast to all session members, and an appropriate site to serve the request is selected dynamically by a multicast feedback mechanisms [74]. Here, the *exponential feedback raise* algorithm of Nonnenmacher and Biersack [183] is used: The basic idea is that each member able to answer the request sets a feedback timer; the timer values are exponentially distributed. If this timer expires, the state is sent by multicast. Should the answer of another site be received before the own state is transmitted, the local timer is canceled. This prevents multiple answers.

The second problem is caused by the fact that a site j that might want to answer a state request is not able to guarantee the correctness of its own state S_j . For example, there might be a late-arriving operation en route to j . A state should therefore carry sufficient information that allows to check whether it is complete and correct (e.g., in the form of a state vector). The requesting site i can then be provided with a correct state by *iterative state transmissions* or by *iterative state requests*.

With *iterative state transmissions*, all sites k compare S_j to their local state S_k . The comparison can lead to four main results: (1) S_j includes the same operations as S_k and nothing needs to be done. (2) S_j includes all operations that are included in S_k as well as some additional operations. In this case, k has missed some operations and should adopt the received state S_j . (3) S_k includes all operations that are part of S_j as well as some additional operations. In this case, j has sent an incorrect or incomplete state, and k will send its own state to repair this problem (using feedback suppression with exponential timers as described above). (4) Both states contain operations that are not included in the other one. In this case, both states suffer from short-term inconsistencies. Now it is checked which state contains more operations, using the sender identifiers as a tiebreaker if the number of operations is equal. If S_k contains fewer operations than S_j , then k discards its local state and adopts S_j . If S_k contains more operations than the received state, then S_k is transmitted using the appropriate feedback suppression. We expect this case to be very rare in practice since it is likely that another site l possesses a complete state that it will distribute in the request process.

After a limited number of iterations, this algorithm will result in all sites having the same state. If there is no single participant who has received all operations, the overall result is a consistent state across all site that misses some operations. This is acceptable since it can only happen because of the exceptional situation that a network is partitioned longer than covered by the operation history. In this situation, it seems reasonable to keep the state of the partition where most changes have been executed and adapt the state of the other partition(s).

The second possibility to deliver a correct state to the requesting site i is the *iterative state requests* approach. Here, the responsibility for checking the state S_j sent in the reply lies entirely with i : i compares the state vector (or other adequate meta-data) of S_j to information gathered from received operations and from session messages, which are exchanged periodically among all session members. If this check leads to the result that S_j is incomplete or incorrect, it is discarded, and another request round is initiated. This is repeated until the check is successful. The iterative state request approach might take longer to complete than iterative state transmission. But if it is assumed that under normal conditions a high percentage of session members hold a consistent state, and that sites suffering from a known inconsistency do not answer to state requests, one would expect only very few cases where a repeated state request is actually necessary.

For both approaches it is advisable to prevent new actions by i until it is verified that i 's state is correct in order to prevent operations on the basis of an inconsistent state. Both the iterative state transmission and the iterative state request mechanism can be managed autonomously by a generic consistency control service. The application needs to provide only functions for retrieving and setting the state of objects (see also Section 7.4.1).

4.7 Experimental Results

In the following, the consistency control mechanisms of local lag, timewarp, and state request are discussed and employed for the mlb, for Instant Collaboration, and for the Spaceshooter game. Furthermore, results from experiments and simulation studies are presented.

4.7.1 Results for the mlb

The discrete application mlb employs the generic consistency control service, which combines local lag, timewarp, and state request in the form of iterative state transmission. In our experience of nationwide mlb sessions, a conservative local lag value of 100 ms is sufficient to prevent practically all short-term inconsistencies without having a negative impact on the user. Even though some users notice the higher response time, they usually hold the system's local software performance responsible for it. Before executing an operation, its state vector is checked whether it is causally ready. If not, the operation is delayed until the missing operations have been received. For handling timewarps, the mlb uses a combination of the round-based and the filtered timewarp. For each chapter or page, an independent operation history is managed as described in Section 4.5.4. These improvements to the timewarp algorithm allow to execute a timewarp almost in real-time, with typical processing times clearly below 50 ms on an average PC where the major time is spent for rendering the updated state.

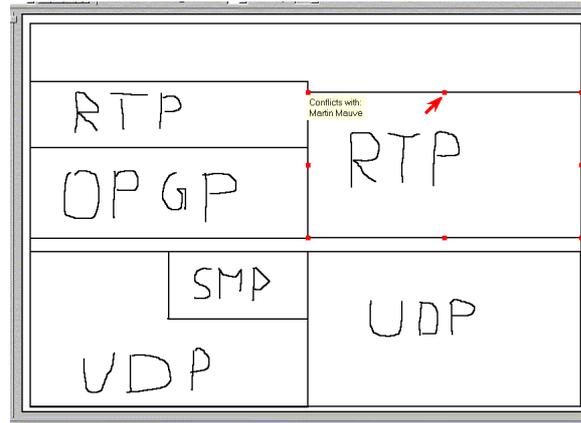


Figure 4.8: mlb experiment

In order to test the filtering approach for preventing timewarps, we conducted a series of experiments for two participants collaborating with the mlb [260]. The experimental setup is a worst-case scenario where each operation received causes a short-term inconsistency (by introducing an artificial delay) and therefore is a potential candidate for a timewarp. The task for the two participants was to outline a protocol stack as shown in Figure 4.8. By using polylines rather than text, the number of operations and objects created during the experiment is very high.

The prime source of timewarps in this scenario is the creation of objects such as the polyline segments that are used for the text: When the creation of two objects overlaps in time, the corresponding operations are potential candidates for a timewarp since the display order (foreground/background) of one of the objects could be wrong (see Figure 4.6). Another potential source of conflicting operations is the resizing and positioning of the boxes. When both participants are working on the same box, conflicting operations may happen.

The results of three runs of the experiment are shown in Table 4.1. The second column shows the total number of operations that were received late by both participants, which is identical to the total number of operations. In a more natural environment with an adequate amount of local lag, it is expected that only those operations that are dropped by the network and have to be retransmitted will arrive late. The third column shows the number of operations that did not cause a timewarp since there was no executed operation with a greater timestamp than the late-arriving operation (see Figure 4.5). Column four shows the number of operations that did not cause a timewarp because they were not in conflict with operations with a larger timestamp. The fifth column indicates the number of operations that were overwritten and therefore did not lead to a timewarp. Finally, the sixth column shows how many timewarps took place. Overall, only 0.3% to 1% of the late arriving operations actually caused a timewarp, which demonstrates the efficiency of our approach.

run	late operations	no later operations	no conflict	overwrite	timewarps
1	687	310	323	47	7
2	515	193	295	25	2
3	594	246	314	31	2

Table 4.1: Timewarp performance

4.7.2 Results for Instant Collaboration

In this thesis, a consistency control service was also developed for Instant Collaboration (see Section 2.4.3). Here, local lag is not employed because of two reasons: First, Instant Collaboration contains only discrete applications which means that local lag would be useful only to coordinate concurrent operations. But concurrent operations are rather unlikely in typical usage scenarios, and almost all consistency-related problems are caused by the interleaved phases of synchronous and asynchronous collaboration. This aspect will be discussed in detail in Section 6.7. Second, one major design goal was to keep the architecture as simple as possible without relying on synchronized local clocks at the participating sites [83].

The consistency control mechanism of Instant Collaboration is based on state vectors, and operations are cached until they are causally ready to be executed [83]. Correctness is achieved with the round-based timewarp algorithm together with filtering. The size of the operation history is limited by state vector analysis as described in Section 4.5.4. These mechanisms were evaluated in different scenarios for synchronous and asynchronous collaboration that reflect the activities of a typical work week of five days for small peer groups of two or three members [83] (also see Section 6.7.2). In total, three objects are created per work day, and each participant issues 50 state changes on the average per day. The simulated scenarios differ with respect to the time spans that a participant spends online or offline. Operations that cannot be delivered because some participants are offline are cached until the receivers can be reached again (please refer to Section 6.7.1 for details). The longer the time span is that some sites spend offline, the more operations need to be cached, and the higher is the probability that timewarps are triggered when these operations are exchanged later on.

When all participants collaborate synchronously over the entire simulation time, a timewarp can happen only in the rare case where two or more operations targeting the same shared object are issued concurrently. For a scenario with two participants, a total number of five timewarps are triggered with an average number of 6.4 operations that need to be executed in order to regain a correct state. When both participants work online for only 80% of the simulated time, a timewarp becomes much more likely since concurrent actions can now happen over the whole time span where at least one participant is offline. Consequently, a total number of 27 timewarps occur with 6.7 operations to be executed on the average. When the time that sites spend online is further reduced to 50%, 118 timewarps happen with an average of 13.2 operations to be executed. In the last scenario, the two participants worked

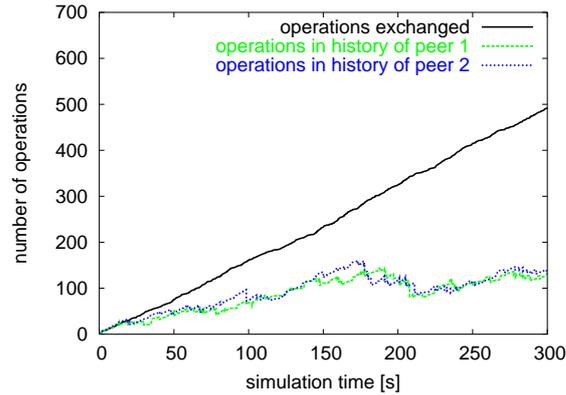


Figure 4.9: Size of the operation history

synchronously only for 20% of the simulated time. Since in this case many shared objects and subsequent operations are actually created while being offline, the total number of timewarps decreases to 44 with an average of 9.5 operations to be executed.

The effect of the algorithm to reduce the size of the operation history can be seen in Figure 4.9, which depicts the total number of operations exchanged in comparison with the actual size of the history of the two participants, assuming an online time of 80%. In this scenario, our algorithm is able to limit the size of the histories to approximately 25% of the total number of exchanged operations. For the other scenarios, this number lies between 20% and 40%. These savings seem to be only marginal but can be explained by the number of objects, each with its own history.

4.7.3 Results for the Spaceshooter Game

The concepts of local lag and round-based timewarp are also used to ensure the correctness of the shared state for the multi-player game presented in Section 2.4.2. In order to evaluate the properties of these mechanisms and their effects on the user, several experiments were conducted [161].

In a first scenario, two human players competed with each other, and the trade-off between a low response time and a low probability for short-term inconsistencies was investigated by choosing different values for the local lag and the network delay. A local lag below 120 ms was not noticed at all by the test persons. In case the average network delay exceeds the local lag value, short-term inconsistencies occur. Their impact is the higher, the larger the difference between network delay and local lag is because the starting state used in a timewarp dates back further. Once this difference exceeds approximately 140 ms, users were able to recognize visual artifacts like abrupt changes in the position of the spaceships.

In the second experiment, a high number of users n was simulated in order to analyze the performance of the round-based timewarp algorithm. Besides verifying that the duration of one timewarp increases by $O(n^2)$ with the number of participants, the experiments also indicate that an optimized implementation can support up to 150 participants on a standard PC when updating the state 25 times per second [161].

4.8 Undo of Operations

To undo operations is an important feature allowing users to take back incorrect or accidentally issued actions [199]. Furthermore, Gordon et al. argue that with undo users can adopt a trial and error strategy and easily explore alternative solutions [90]. Even though it might be possible for a user to undo the effects of a former operation manually by creating new operations, an application-supported undo is far more efficient; therefore most applications have built-in undo functions (e.g., Microsoft Office applications). For distributed interactive applications, undo is especially important since multiple users may issue concurrent and conflicting operations on the shared state of the application, and a user might not always be able to foresee the actions of remote participants. Similarly, it should also be possible to redo operations that were undone before.

But at the same time, providing undo and redo for distributed interactive applications is challenging since operations of different users are interleaved and may depend on each other. For instance, consider the following situation: User i deletes an object with the operation O_i and user j concurrently modifies the state of that object with O_j such that $O_i < O_j$. Then user i undoes his own operation O_i . Now the question arises how this undo is to be handled: Should the object, once it is restored, have the state that was valid when user i deleted it, or should the modification of user j be included in the restored state? Because of these semantic dependencies among interleaved operations of different users, the task of undoing and redoing operations is problematic for distributed interactive applications: The application does not understand the intentions of the user when issuing an undo operation and can therefore not decide on the desired result. Nevertheless, from the user's perspective an automatic undo mechanism is desirable.

Undoing operations for continuous applications bears an additional challenge since the shared state might not only have changed due to operations but also due to the passage of time in the time span between issuing an operation and undoing it. But in this section, we concentrate on discrete applications. First, the concepts of undo and redo are introduced more formally, and fundamental design alternatives for undo algorithms are discussed. Then, existing approaches are examined. Following, the undo mechanism designed in this thesis for discrete applications is presented, and its properties are analyzed.

4.8.1 Formalization of Undo and Redo

There are two different notions of undo and redo: A strict one, which closely observes the consistency criteria, and a relaxed one, which tries to achieve the user's intention. In the *strict* definition of undo, the desired effect of undoing an operation O_i is to reach the state that would have been calculated if O_i never had been executed, i.e., O_i and its effects are to be eliminated from the operation history. In the example described above, this would mean that the undo operation should recreate the object, and the effect of j 's operation should be reflected in the object's state. And redoing O_i should result in the state reached by applying O_i in the first place, i.e., redoing an operation O_i means to undo the undo of O_i .

In the *relaxed* meaning of undo, the object is restored to the state that it had before the operation O_i to be undone was executed. This means that all operations O_j on the object following O_i in the operation history are ignored. Thus, only the perspective of the user that issues the undo operation is considered: In the example, user i wants to restore the object to its former state. Since $O_i < O_j$, i is not even aware of j 's operation, and O_j should not be reflected in the recreated object.

Recent literature seems to prefer the more formal notion of undo [28, 200, 211, 233, 234], but both notions have their advantages and disadvantages. While the strict undo provably fulfills the consistency criteria, the relaxed undo is significantly easier to handle for an algorithm. As discussed above, it is not obvious which of the two meets the desired semantics of the application better.

4.8.2 Design Considerations for Undo Algorithms

Besides the question of which notion of undo an algorithm should follow, there exist a number of design choices that need to be taken into account, namely which undo mode to use, whether relative operations need to be supported, and how undo and redo operations are encoded.

An undo mechanism can be realized in different modes, which determine the next operation to be undone. In a *local undo* scheme, a participant can only undo his own operations whereas *global undo* includes the possibility to undo remote operations as well [3]. Undoing a remote operation might be confusing for the user when he is not aware of the other participant's intention, and is also likely to disturb the remote participant. Orthogonal to the decision whether the undo mode should be local or global are the modes of linear and selective undo: If operations can be undone only in the reverse order in which they were originally executed (i.e., following the reverse order of the operation history), we speak of *linear undo*. Alternatively, with *selective undo*, arbitrary operations might be chosen as the next operation to be undone [252]. In case global and linear undo are combined, the next operation to be undone

depends on the local history of an application instance and might be different at different sites.

Selecting an appropriate undo mode is application-specific, and an undo algorithm should be able to undo any operation O_i that is determined to be undone, independent of how O_i is actually chosen [233]. It is also possible for an application to support different undo modes, which can be selected by the user, e.g., by providing different buttons for local and global undo.

An important issue that needs to be considered when designing an undo algorithm is whether the application employs relative operations (see Section 2.2). If this is the case, then undoing an operation O_i also affects all operations that follow O_i in the operation history, and an undo algorithm must include appropriate measures to compensate side-effects. For instance, let “ab” be the current state of a shared text editor. First, user i issues the operation O_i to insert the character “1” at position 3 at the end of the current text, then user j issues O_j to insert “2” at position 2 so that the new state is “a2b1”. When i undoes O_i , the position of “1” has changed in the meantime, and the naive undo operation “delete character at position 3” would lead to the incorrect state “a21” (whereas the expected state would be “a2b”).

The last design consideration concerns how undo and redo operations are propagated to all session members. Basically, there exist two possibilities: If an undo is encoded *semantically*, it contains all information necessary to change the current state such that the effect of O_i on the state can be undone. For instance, consider a shared whiteboard session where the user i changes the color of a rectangle from green to blue with O_i . Then the undo would contain the original color green of the rectangle, and a remote application would only need to apply the undo operation to its current state to execute the undo. This also implies that the undo operation would be handled by the application (and its consistency control mechanism) in exactly the same way as any other operation, e.g., an undo operation carries an independent state vector.

In contrast, a *syntactic* undo identifies the operation O_i that is to be undone (e.g., by O_i 's sender identifier and sequence number), and a site receiving such an undo needs to calculate the necessary steps to execute the undo locally. Thus, a syntactic undo introduces a special operation into the data exchange model of the application and can be recognized as such. The advantage of syntactic undo is that it is more flexible, e.g., when application instances have different operation histories. However, this scheme also introduces dependencies into the operation history that need to be considered by the application and by generic services operating on the data model of distributed interactive applications (see Sections 2.2 and 7.4).

Irrespective whether an undo is realized syntactically or semantically, we denote the undo operation for O_i as O_i^{-1} , and the redo operation as $(O_i^{-1})^{-1}$, which is not necessarily identical

to O_i . For instance, in the strict meaning of undo, only the effects of $(O_i^{-1})^{-1}$ on the current state are required to be the same as executing O_i at its original position in the history.

4.8.3 Related Work

In [12], Berlage proposes a framework for selective undo in distributed interactive applications. An undo operation O_i^{-1} is derived semantically from the original operation O_i without considering any relative operations O_j that might follow O_i in the history, and O_i^{-1} is executed on the current state. Thus, the approach follows the relaxed concept of undo. The user can select the operation to be undone in a list containing textual descriptions for the operations of the history (e.g., “change color of rectangle 23 to green”), i.e., both local and remote operations can be undone. This list can also be searched with regular expressions. Operations that currently cannot be undone are not listed. For instance, in case a later operation has deleted the target object. For applications that depend heavily on relative operations such as text editors [238], ignoring dependent operations is not a viable approach and might result in severe inconsistencies. Another drawback is that selecting the operation to be undone via text-based descriptions might be difficult for the user, especially in extensive histories. For instance, objects are named by their internal identifier (“rectangle 23”), which has no meaning to the user. From the user’s perspective, it might actually be faster to undo the operation manually instead of searching the operation list.

The concept of operational transformation was introduced to realize consistency control especially for applications with relative operations (see Section 4.3). Operational transformation can also be extended to facilitate undo and redo functionality in a strict meaning. Such mechanisms are proposed in a number of articles that differ with respect to the undo modes supported and with respect to their technical details [200, 211, 234]. The common idea for undoing O_i by operational transformation is to first create an undo operation O_i^{-1} that is the direct inverse of O_i , e.g., if $O_i =$ “insert character 1 at position 3” then $O_i^{-1} =$ “delete character 1 at position 3”. In a second step, O_i^{-1} is transformed against all operations O_j following O_i . The new undo operation $O_i^{-1'}$ is then applied to the current local state. All undo operations need to be propagated syntactically since the local operation histories of the participating sites might be ordered differently so that each site has to transform the undo operation individually. The most powerful and generic undo mechanism is the one presented by Sun in [234]. It supports all possible undo modes and is also able to handle conflicting operations. However, adding undo and redo capabilities to operational transformation increases the algorithm’s complexity considerably.

The consistency control mechanism of object duplication (see Section 4.3) is extended by Chen and Sun in [28] to provide undo and redo. Object duplication handles conflicting oper-

ations by creating multiple versions obj_k of the same object obj . Thus, undoing an operation O_i might delete a certain version obj_k of obj if O_i is the last operation in the operation sequence of obj_k conflicting with other versions of obj . In this case, the concerned versions are merged. Likewise, when redoing an operation, versions might be restored. The undo/redo operation itself is encoded syntactically in order to keep the communication overhead low. A severe drawback of the object duplication approach is that undo or redo might have complex side-effects that create, delete or merge object versions. These effects might be rather difficult to understand for the user.

4.8.4 Undo of Operations for the mlb

Since the mlb does not employ relative operations, an undo algorithm was designed based on the main ideas of the selective undo presented by Berlage in [12]. It complies to the relaxed notion of undo: An undo operation O_i^{-1} is executed irrespective of any other operations O_j with $O_i < O_j < O_i^{-1}$. Moreover, undo and redo operations are encoded semantically in order to avoid dependencies within the operation history of a session. From the user's perspective, O_i^{-1} restores those attributes of the state changed by O_i to the state that was valid before O_i was issued. For instance, when undoing a delete operation, the object is restored to the latest state that was visible to the user, and any concurrent operations modifying this state are ignored. For this purpose, the application creates O_i^{-1} for all operations O_i at the same time that O_i is issued by the user² such that O_i^{-1} contains all information necessary to undo O_i . For instance, when O_i deletes an object, O_i^{-1} would be the entire current state of the object, and if O_i changes the color of an object to green, O_i^{-1} would encode a state change to the object's former color.

All undo operations are stored in an undo list. In case the user undoes an operation O_i , the appropriate undo operation O_i^{-1} is retrieved and issued like any other operation, i.e., it is assigned a current state vector and/or a current timestamp. The application instances handle O_i^{-1} like a regular operation, i.e., it is added to the operation history and might trigger a timewarp if it is received too late. At the time the user invokes the undo command, a redo operation $(O_i^{-1})^{-1}$ is created such that it is able to restore the state that was valid when O_i^{-1} is issued. All redo operations are managed in a redo list that is emptied as soon as the user issues the first operation manually after a series of undo commands. Likewise, when the redo command is invoked, the corresponding undo command is added to the undo list. The mlb maintains separate undo and redo lists for each document page.

²Please note that due to local lag the state of the application might be different at the time t^o an operation O_{i,t^o,t^*} is issued and the time t^* it is executed.

Even though all possible undo modes are supported, this approach works best with the combination of local and linear undo/redo. This allows to provide users with an interface that is easy to understand and that resembles the functionality of well-known single-user applications. Like the consistency control algorithms presented earlier in this chapter, the management of undo and redo lists can be realized in an application-independent way on the basis of operations.

Since our approach follows the relaxed definition of undo, we will now examine its consequences on the example of representative operation sequences and demonstrate that the resulting shared state is reasonable. Some of these operation sequences are also known as “undo puzzles” and are widely used to analyze undo algorithms [200, 211, 234]. Let i and j be two collaborating users, O_i and O_j two operations issued by i and j modifying the same attribute of an object obj , O_i^\dagger and O_j^\dagger operations that delete obj , and O_i^{-1} , O_j^{-1} , $O_i^{\dagger-1}$, and $O_j^{\dagger-1}$ the corresponding undo operations.

Example 1: Consider the operation sequence $O_i < O_j < O_i^{-1}$. O_i^{-1} resets obj to the state it had before the sequence was executed and therefore undoes O_j at the same time. This is acceptable from the i 's perspective since the undo was invoked to restore the object's original state.

Example 2: $O_i^\dagger < O_j < O_i^{\dagger-1}$. Following the relaxed notion of undo, this restores obj to its original state and ignores the modification of O_j , which is acceptable from i 's perspective since i never saw the effects of O_j and would be confused if obj is not in the expected original state. From j 's perspective, this means that the effects of O_j that were visible before are also undone, which might be confusing.

Example 3: $O_i < O_j^\dagger < O_i^{-1}$. In this case, O_i^{-1} cannot be applied since obj does not exist anymore and O_i^{-1} holds only information about attributes that were changed by O_i . But in order to restore obj , O_i^{-1} would need to encode the complete state, which is not a viable alternative: Since O_j^\dagger and O_i^{-1} could be concurrent operations, i does not know when O_i^{-1} would need to be an event and when a state. Thus, a state would always be required, increasing the memory space for storing the undo and redo lists and also the network load. Instead, the undo algorithm ignores O_i^{-1} if obj does not exist. This is reasonable when the user i already saw that obj was deleted but might be confusing if O_j^\dagger and O_i^{-1} are concurrent.

Example 4: $O_i^\dagger < O_j^\dagger < O_i^{\dagger-1}$. Similar to Examples 1 and 2, obj is recreated despite the double delete operation. This meets i 's expectation that wants obj to be undeleted. In comparison, the operational transformation approach described by Sun in [234] would restore obj only when all delete operations are undone.

In order to use this undo algorithm for the mlb, all operations on the mlb state are designed such that they are absolute (see Section 3.3.3). Among all graphical objects, text is the most critical in this respect. Current group text editors such as DistEdit [138] and REDUCE [238] model operations to insert and delete characters relatively, i.e., operations identify the indices where characters are to be inserted or deleted and adjust these indices by means of operational transformation in case they were changed by other operations. For the mlb, a different approach is used: Each character is treated like a regular graphical object with its own unique identifier. When undoing an insert character operation, the corresponding delete operation can therefore be assigned unambiguously to its target character, even when this has changed its position in the meantime. In contrast, undoing a delete operation restores the character at its original index, which might not be what the users intended. For instance, let *obj* be “hello world”, $O_i = \text{“delete e”}$, $O_j = \text{“insert my at index 0”}$, and $O_i < O_j < O_i^{-1}$. Then our undo scheme would determine O_i^{-1} as “insert e at index 1” and the resulting state as “mey hlllo world”. For the mlb, this is nevertheless acceptable since in typical mlb sessions text objects are rather small and not edited concurrently by several users. Moreover, this problem does not exist with linear local undo when operations are only issued by a single user. Also, the situation described above might be problematic semantically but does not cause an inconsistency. The major advantage of this approach is that it substantially simplifies the management of operations and the history for the application as well as for generic services operating on the data stream of the application (see Chapter 7).

To employ this undo mechanism for continuous applications would be rather difficult since O_i^{-1} would have to take the state changes into account that happened because of the passage of time between O_i and O_i^{-1} . These changes do not only affect the target object *obj* itself, but might also influence other objects that have interacted with *obj*, e.g., in a collision of moving objects. A better solution for the continuous domain might therefore be syntactic undo operations, which delete the original operation from the history: O_i^{-1} is dated back to the execution time of the original operation O_i , deletes (or neutralizes) O_i , and triggers a timewarp in order to calculate the updated current state. Removing O_i from the history might have severe side-effects on operations following O_i so that the updated state differs significantly from the former state. Please note that all application instances would need to execute a timewarp so that such an undo operation is heavyweight. Another drawback is that an additional operation type is necessary, which needs to be considered by generic services. These issues require further investigation.

Summing up, undo and redo are powerful commands, which allow a user to access and manipulate past states of the application. This is especially important for distributed interactive applications allowing concurrent operations since it is difficult for a user to predict the side-effects of his local operations that are interleaved with the actions of remote participants.

Thus, it is much more likely that the resulting state does not meet the expectations and intentions of the users when compared to a single user application. In the next chapter, other possibilities besides the explicit undo and redo commands are presented to visualize the sources and the effects of such concurrent operations together with alternative operation sequences.

4.9 Conclusions

Replicated interactive applications need to synchronize the local copies of the shared state managed by the application instances. The consistency of these state copies is endangered by the concurrent exchange of operations. For continuous applications, the passage of time is another source of possible inconsistencies when operations are received after their scheduled execution time. Thus, the application needs to employ an appropriate consistency control mechanism.

In this chapter, different consistency criteria were introduced, which need to be observed by discrete and continuous applications: *Causality* ensures that necessary preconditions are fulfilled before an operation can be executed, and *correctness* guarantees that the state of all instances is identical to the state of a virtual perfect site, which receives all operations without network delay. Correctness can be achieved by executing all operations in the state vector order (for discrete applications) or the physical time order (for continuous applications).

Following, the concepts of soft state and hard state were discussed, and after analyzing related work, a generic consistency control service was presented. This service uses a combination of consecutive consistency control algorithms to achieve correctness. In a first step, local lag eliminates a large percentage of short-term inconsistencies by voluntarily delaying the local execution of operations. Short-term inconsistencies not prevented by local lag are repaired by timewarp, which uses the local operation history to recalculate the current state. With the round-based timewarp and the filtered timewarp, two improvements to the basic algorithm were presented. Moreover, possibilities to restrict the size of the local operation histories were discussed. Due to this restriction, it is possible that an inconsistency cannot be repaired by timewarp anymore. Even though this occurs only in exceptional situations, the consistency control service employs a novel state request mechanism as a last resort. The feasibility and the good performance of all of these algorithms were demonstrated on the example of three applications.

In the last section, the possibility to undo and redo operations was identified as an important issue for distributed interactive applications; it is closely related to consistency control. An undo mechanism for discrete applications was presented, which is used for the mlb. This mechanism seeks to observe the intentions of the local user when issuing an undo or redo

command. In the next chapter, other possibilities for the user to analyze concurrent operations and to explore alternative solutions and operation sequences are presented.

Chapter 5

Visualization of Conflicting Operations

The combination of local lag, timewarp, and state request presented in the last chapter is an effective way to establish consistency for distributed interactive applications. Under regular conditions, most short-term inconsistencies are prevented by local lag. And in the rather unlikely case that local lag is insufficient and an operation is received at a site after its scheduled execution time, a timewarp restores the correct shared state. From a technical point of view, the timewarp algorithm is robust, scalable and economical with respect to its consumption of resources. And in the worst case, when sufficient history data is no longer available, requesting state information from other session members allows the local application instance to retrieve the current state.

However, from the user's perspective, the execution of a timewarp might obstruct the collaboration among users or cause visual artifacts [91]. For example, in Figure 4.1, two concurrent operations O_1 and O_2 change the color of the same object. If we assume that $O_1 < O_2$, then the timewarp algorithm with filtering ignores O_1 at site 2 and determines the object's color to be gray at both sites (see Section 4.5.3). While this is efficient, it also means that the participant at site 2 is not even aware of the fact that the first user also intended to change the object's color. Instead of finding a consensus between the two users about the correct state, the timewarp algorithm favors the participant whose operation is determined to be last in the operation history.

Another critical situation is shown in Figure 5.1: The user at site 2 intends to raise the white rectangle to the foremost position while the user at site 1 wishes to have the following display order from bottom to top: White rectangle, gray rectangle, black circle. Even though the operation O_2 issued at 1 lies before O_1 , at site 1 O_1 is executed first. Thus, when O_2 arrives at 1, a timewarp ensures the correct order of objects (gray rectangle, white rectangle, black circle). Obviously, the final consistent state of this sequence violates the original intention

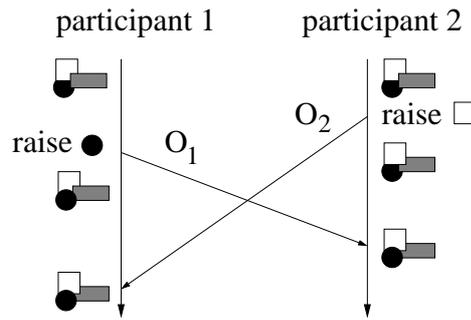


Figure 5.1: Concurrent operations O_1 and O_2 trigger a timewarp at site 1

of user 1. And even though at site 2 no timewarp is executed, the intention of user 2 is also violated. This effect of concurrent operations is the more disturbing for the users the later O_2 arrives, and the more operations and users are involved.

For continuous applications, a late operation O_{i,t^o,t^*} may cause a consistency-related visual artifact even when the history does not contain an operation conflicting with O_{i,t^o,t^*} : Let O_{i,t^o,t^*} modify the direction of a moving object obj . Then the position of obj has changed since the scheduled execution time t^* , and the new position of obj that is calculated by a timewarp might differ from its current position so that obj “jumps” to the corrected coordinates.

In all these cases, the timewarp algorithm displays the final, correct state. But as indicated, this might be confusing for the users since they do not know how this state came to be. Thus, from the user’s perspective, it would be desirable to have more insight into the effects of the timewarp algorithm and into the effects of concurrent operations in general. In this chapter, we therefore propose to provide information about conflicting operations and to show the effects of serialization in order to help users to understand the intentions of others. The users may then employ some social protocol to agree on the shared application state.

First, we analyze the popular concept of intention preservation and show that it is not sufficient for our purpose. Then, semantic conflicts of operations are discussed in Section 5.2, and important design options for visualization mechanisms are explored in Section 5.3. Related work is analyzed in Section 5.4. Following, a novel technique to facilitate collaboration and awareness by visual analysis of the operation history is presented in Section 5.5. The chapter is concluded in Section 5.6.

5.1 Intention Preservation

A consistency criterion that addresses conflicts in concurrent operations is defined by Sun et al. in [238] with **intention preservation**: A distributed interactive application achieves intention preservation when executing an operation O_i on the current shared state has the same

effect as executing O_i on the state that was valid at the time O_i was issued. This means that the original intention of the user is preserved even when the state of some application instances was modified by concurrent operations in the meantime. Intention preservation mainly aims at relative operations. For instance, most multi-user text editors use relative insert and delete operations identifying the position of the concerned characters (e.g., “insert u at index 1 of string ct”). These positions might have changed due to concurrent operations so that executing the original operation would lead to an unintended state. Operational transformation is a consistency control mechanism that complies with intention preservation by changing the referenced indices of text operations such that the desired effect is achieved [238] (see Section 4.3).

However, intention preservation is only successful as long as concurrent actions are not semantically opposed, e.g., when changing the meaning of a sentence. Moreover, for absolute operations it is not possible to preserve the intentions of all users. For instance, in Figure 4.1 the rectangle can be either black or gray. The approach of object duplication seeks to overcome this limitation by creating multiple versions of the same object in the case of conflicts [236]. But it offers no support for merging different versions and finding a joint state (also see discussion in Section 4.3).

In the following, we concentrate on serialization as the consistency control algorithm and on semantic conflicts of operations where the users alone are able to determine the desired state.

5.2 Semantic Conflicts in Operations

While many distributed interactive applications allow all participants to access and modify the objects within the shared workspace at any time, it may happen that users issue operations that *conflict semantically*. This might lead to a state that does not meet the users’ expectations and that may be difficult to interpret. More precisely, two operations O_{i,t_i} and O_{j,t_j} created at the sites i and j at the times t_i and t_j conflict semantically in case they modify interdependent aspects of the shared state. For instance, let O_{i,t_i} change the position of a rectangle and O_{j,t_j} delete the same rectangle. Then the two participants obviously disagree about the desired state of that rectangle. A critical situation occurs when such operations are issued within a brief time span so that the users are not aware of each others’ actions. This means that O_{i,t_i} and O_{j,t_j} are not necessarily concurrent (see Section 4.1.1). Instead, the critical time span $t_{cr} = |t_i - t_j|$ depends on the reaction time of the users. For the mlb, preliminary experiments indicated that t_{cr} is approximately one second when the concerned slide is completely visible on the screen. The application should therefore visualize all semantic conflicts within t_{cr} in an appropriate way so that the users are aware of the conflict and are able to find a compromise.

Semantic conflicts can be discovered when checking received operations against the operation history using a conflict function similar to the one given in Figure 4.6 for shared whiteboards¹. This should be done for all operations and not only in case a timewarp is triggered. For instance, both users in Figure 5.1 should be made aware of the emerged conflict so that they will be able to find an agreement about the objects' order.

5.3 Design Considerations

Semantic conflicts in user actions can be discovered using predefined functions such as the one given in Figure 4.6, and they could be resolved syntactically by a consistency control mechanism such as timewarp where the final shared state is calculated on the basis of certain rules. However, the application itself is in general not able to determine the desired state. This would require the computer to understand the motivations and intentions of the users, which is not possible. Instead, a distributed interactive application should provide information about any detected conflicts so that the users are aware of them and are able to resolve them. Thus, our main goal is to visualize conflicting operations and the effects of the timewarp algorithm. For this purpose, we must determine which information should be visualized, and how that information should be visualized.

First of all, it should be indicated if a timewarp was triggered or if a conflict occurred so that the user is aware of this situation. Then the user needs information about all the operations that are involved in a particular conflict: Which objects are modified and how? And where is their place in the shared workspace? Which participants are responsible for the operations? Moreover, since single operations are usually not sufficient to present “the big picture”, additional information about the general course of action might be necessary, e.g., about the task that a user is currently working on. In this context, information about related operations that were issued earlier or by different participants might also be useful. Finally, since a timewarp might change the order of operations within the history, it is interesting to know about alternative states that would be reached with a different sorting criterion and that might be preferred by the session members.

Now the question arises in which form the relevant information about conflicting operations should be visualized. Again, there are different possibilities concerning the placement of information (where), the representation (how), the trigger (when), and the duration (how long): First, data can be displayed within the shared workspace, e.g., next to the concerned

¹A function to check for semantic conflicts would differ in some details from the one given in Figure 4.6, e.g., if both operations are events deleting the same object this would not be considered to be a semantic conflict since both users have the same *intention*.

objects. While this provides a direct reference, it might be distracting or hide the content of the workspace in case there is not enough screen space. Alternatively, information can be shown in a separate window. This prevents that the workspace becomes cluttered but makes it more difficult to relate awareness information to their objects or actions.

Second, information can be represented in different ways as discussed by Tam in [244]. Most common are graphic representations such as icons or simple objects. Information can then be encoded using different colors (e.g., mark objects that are changed by conflicting operations in red), shapes (e.g., show a delete event as cross), and sizes (e.g., display the names of conflicting participants in a larger font size). The nuances and order within a single category also expose information. For instance, there might be only two different colors to mark participants as either involved in a conflict (e.g., red) or not (e.g., blue), or a continuous graduation from black to light gray may indicate the recency of a conflict with an aging effect (see Section 3.5.1). However, these graphical representations also require that the user acquaints himself with the respective meanings, which is increasingly demanding with a rising number of information categories. As an alternative, information can also be encoded as textual descriptions, which are easy to understand but might require more time to take in. In addition to static visualizations, animations are effective in demonstrating complex circumstances and in catching the user's attention. Aside from animating single pieces of information (e.g., operations or participants), the operation history itself can be animated, which allows the user to review the (complete or partial) course of action of a certain time span.

Third, information can be displayed automatically or on the explicit request of the user, which is a tradeoff between effort and distraction. And fourth, the data might be visible permanently or only for a short period of time. While a temporary visualization releases screen space, vital information might disappear too quickly.

When designing a visualization mechanism, some general issues have to be considered: First, the mechanism should be easy to use without imposing too much overhead since participants would switch to other information retrieval strategies otherwise, e.g., explicit audio communication. Second, while all necessary information should be displayed, an information overload where the amount of information has a negative impact on the performance has to be prevented [182]. Since this issue depends on the user, the mechanism should be adaptable to the user's expertise by setting the level of detail (beginner vs. expert). In this context, the tradeoff between displaying fine-grained information about the individual operations on the one hand and about the general context on the other hand is especially challenging. Third, the information should be available when the user wants to access it, including the possibility to review conflicts that occurred at some time earlier. Finally, a user should be able to analyze information and explore alternative states locally without disturbing remote participants.

5.4 Related Work

Providing the users of a distributed interactive application with awareness information about simultaneous and possibly conflicting state changes has been identified as a vital aspect, which determines the success of such an application [51, 59, 91]. However, most existing work focuses on general awareness information about past and present changes and does not address the specific effects of consistency control mechanisms. In the following, we discuss some existing approaches, which serve as a basis for the mechanisms presented in the next sections.

The concept of visualizing the operation history was first introduced for single-user applications where it proved to be effective and easy to use. For instance, Chimera [143] illustrates the evolution of the application's state in a separate window by a series of state snapshots, each displayed as a small thumbnail. In Interlocus [109], such a snapshot history is the interface to the undo functionality so that the user can select the state to be restored.

The explicit visualization of the operation history is also a promising technique for distributed interactive applications. A first approach was presented by Beaudouin-Lafon and Karsenty in [11] where the user is provided with a VCR-like interface. In [87], the history itself is represented as a timeline, which is enriched with meta-information next to those events that are of particular interest to the user. For instance, leaving and joining participants are marked by specific icons on the timeline. Thus, the potentially large history of a session becomes searchable so that the user has quick access to important information. Such indexing can be realized with different levels of detail and different types of meta-information (e.g., explicitly created vs. derived from other data) [82].

Different methods to display past state changes of a shared drawing area in general are investigated by McCaffrey in [163]: A change index describes the operation history textually (e.g., "Alice creates an oval"), a trailing function shows the path of moved objects, and a replay mechanism repeats the course of action during a certain time span. The aim of these methods is to provide information about how the current state came to be in case the user did not keep track of the joint editing process or wants to review it. This is also known as *change awareness* [244]. McCaffrey finds the replay mechanism to be most successful when controlled via the change index [163]. However, the index might become obscure when the history is large. In [244], Tam uses a combination of techniques to visualize changes in a shared drawing area, including a thumbnail overview, which is always visible in a separate window, color codings and textual descriptions within the shared workspace, and a replay mode. Information is given concerning the changed object, the operation, and the author. The system allows the user to investigate changes in various levels of detail, e.g., in the thumbnail overview all modified objects are marked, and during replay minor changes can

be omitted. Since the replay always starts at the beginning and does not offer a fast-forward mode, it might take a long time to analyze a certain history.

A mechanism to manage multiple, alternative operation histories is presented by Edwards in [56]: Instead of automatically merging the operations of multiple users into a single, linear operation history by serialization, the operations form a directed graph with different paths, which may split and join anytime. Each node of this operation graph reflects a different application state, and the path from the initial state to a certain node represents the history of that particular state. The graph itself is displayed so that the user can navigate on it and explore the different versions of the state. The current position in the graph determines the state that is shown in the main application window. Thus, the user has direct access not only to past states but also to alternative versions, similar to the object duplication approach discussed in Section 4.3. Moreover, the user is allowed to change the operation graph at arbitrary positions by issuing new operations. Depending on the editing mode, the graph is split such that one branch reflects the modified state and the other preserves the old state. The inserting mode keeps the number of paths and integrates the change into all subsequent parts of the graph. Semantically, this means a direct manipulation of the past. Alternatively, different paths might be joined in order to create a common state reflecting the changes of multiple users. When new operations are issued, the operation graph is analyzed with respect to possible conflicts [55]. For instance, a delete operation that is inserted somewhere in the graph might cause conflicts with subsequent operations targeting the deleted object. Edwards discusses strategies to handle these conflicts by discarding the conflicting operation or by splitting the affected path [55].

While this approach is very flexible and allows to explore and preserve different versions of the shared state, the operation graph becomes rather complex after a few operations of different users. Keeping track of these multiple “realities” is difficult, especially since users can work on different parts of the graph (i.e., different points in time) simultaneously. Inserting operations at earlier positions of the history graph is likely to introduce additional conflicts instead of resolving them. Finally, the system lacks specific aids to merge conflicting paths into a single and non-conflicting shared state, which should be the ultimate goal of a distributed interactive application.

5.5 Visualization of Conflicting Operations

The goal of this chapter is to introduce a visualization mechanism that helps the user to understand how the shared state of a distributed interactive application evolved, especially in cases where operations of different participants conflict or where an operation is received so late that a timewarp is required. Such a visualization might support the user in changing the

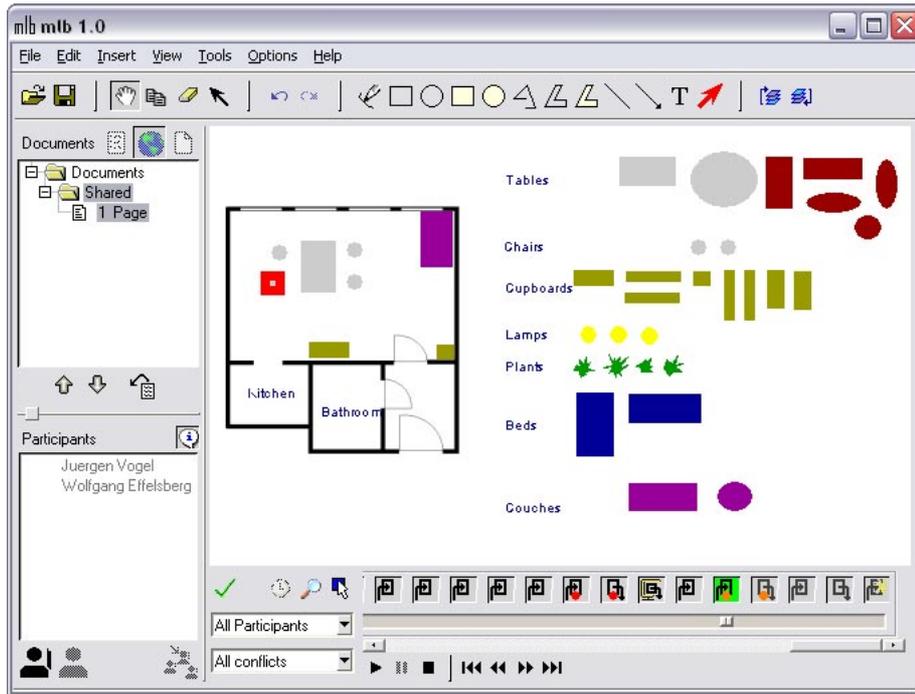


Figure 5.2: mlb with visualized operation history

content of the shared workspace to the semantically desired state and in coordinating future actions with other session members.

For the timewarp algorithm, each application instance needs to store at least recent parts of the operation history (see Section 4.5). As discussed above, the history contains important awareness information: An explicit visualization of the history is an effective way to show the course of action of the respective participants and the correlation among their operations. We therefore propose to provide the information discussed in Section 5.3, i.e., who changed which object by which operation conflicting with which other operations, by means of a graphical representation of the operation history, which the user can access and analyze if needed. Since this information is already available at each site, such a mechanism causes only little overhead for the application.

A prototype for the visualization of the operation history was designed in the course of this thesis and was integrated into the mlb. While the mlb is a discrete application, most of the presented mechanisms are also applicable to the continuous domain. As depicted in Figure 5.2, the history is visible below the shared workspace of the mlb. The interface offers rich information about operations, objects, and participants, provides a replay function, and allows the user to explore past actions and alternative states. In the following, these features are discussed in detail.

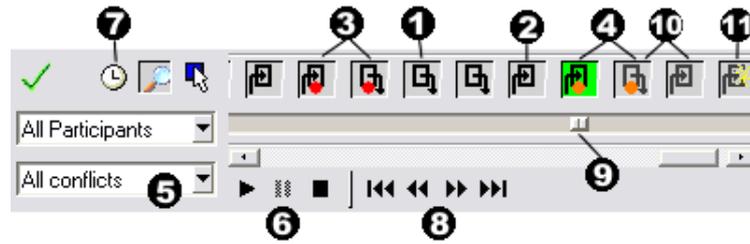


Figure 5.3: Visualization of the operation history

5.5.1 Representing the Operation History

The history is represented as a timeline, which holds an icon for each operation (see Figure 5.3). The operations are displayed in the order of their execution times: The most recent operation is shown on the right, and when a new operation is executed, it is appended to the timeline. An operation icon encodes different information depending on its shape, color, and background: Operations issued by the local user have an outgoing arrow (1), while all remote operations show an incoming arrow (2). Operations that are part of a conflict sequence are marked by a dot that is either red (3) or orange (4) in order to distinguish among different conflict sequences. In (3), one remote and one local operation conflict semantically. In case the application detects such a conflict, the visualization of the history is updated immediately, allowing the user to keep track of all changes. In addition, conflicts are also indicated within the shared workspace by attaching a tool-tip window, which contains the names of the conflicting participants next to the concerned object (compare Figure 3.7). Moreover, the names of the involved session members are highlighted in red in the mlb's participant list, degrading back to light gray (compare Figure 3.8). As discussed in Section 5.2, all conflicts are visualized for operations whose execution times lie within a certain time span. By default the most recent conflict is displayed, but the user can also choose to see all conflicts (5) or a conflict that occurred at a certain time (see Figure 5.4 (1)).

When selecting an operation icon that is part of a certain conflict sequence, the entire sequence is marked so that the history can be examined by the user. Also, the corresponding object is highlighted in the shared workspace so that the user can explore which object belongs to which event (see Figure 5.2). Vice versa, all icons targeting a certain object are highlighted by a white background when the object is selected.

5.5.2 Exploring Past States

Our visualization scheme includes a replay function for reviewing the past course of action. It is controlled with a VCR-like interface (see Figure 5.3): After pressing play (6), operations are replayed in the order given by the history, and the appropriate state is displayed in the

workspace window. The replay can run either in the original time lapse or in a fast-forward mode (7). The skip buttons (8) change the current position in the history. The start time for a replay is limited by the oldest operation that is held by an application instance. The operation that was executed last is marked by a green background (4 left). Moreover, the target object of the last operation executed is highlighted in the shared workspace, and the participant that issued this operation is named in a tool-tip window next to that object (this feature can be disabled since it might be distracting). All events not yet executed are displayed with gray icons (10), instead of black ones (2).

The user can also browse through the history by means of a slider (9). The slider's position marks the last operation executed. When the slider is moved, the content of the workspace is updated accordingly. In preliminary experiments, the slider-controlled history browsing was found to be very effective and allows a fast access to past states and other awareness information.

Exploring the operation history has only effects on the local view and does not disturb remote users. While a past state is displayed, objects cannot be created or modified. All remote operations received in the meantime are appended to the history, but their effect is not visible until the replay reaches their scheduled execution time. An operation is marked as new (11) until it is executed for the first time so that the local user is aware of remote users' actions. After the latest operation was executed, the local user is able to modify the state again.

The application can realize the replay mechanism either by the timewarp algorithm or by the undo functionality. In the first case, a timewarp is executed when a state that is older than the current one is to be displayed (e.g., when moving the slider to the left or when jumping to the history's beginning). When moving forwards in time, the respective next state can be calculated by applying the next operation to the current state at the correct execution time. This execution time is determined on the basis of the current time and the offset between the execution times of the current and the next operation (divided by the fast forward factor if necessary). The advantages of the timewarp-based approach are that it is easy to implement since the required algorithms are already available, and that it is applicable to all applications. However, executing a timewarp is costly in terms of processing power, which might be critical when browsing through the history.

For the mlb, it was therefore decided to implement the replay with the undo and redo functionality described in Section 4.8.4: When moving backward in time, the last operation executed is undone by applying the appropriate undo operation, and when moving forward in time, operations are redone. Jumping to a position (e.g., to the end of the history) is realized by executing an entire sequence of (undo) operations. While this approach requires that the application is able to undo all operations (as it is the case for the mlb), it achieves a very

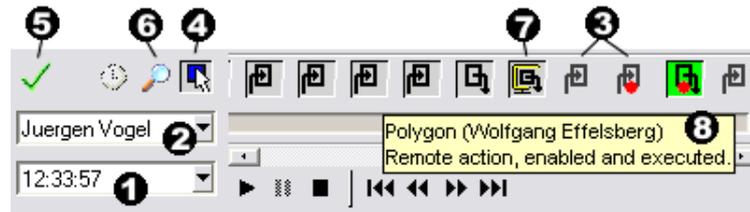


Figure 5.4: Exploring alternative states

good performance resulting in smooth state updates even when skimming quickly through the history.

5.5.3 Exploring Alternative States and Changing the Current State

A user might not only be interested in how the current state came to be but also in alternative states. For instance, when several users issue concurrent and conflicting operations, it is useful to understand the intentions of a certain user. For this purpose, a dominating user can be selected as depicted in Figure 5.4 (2). This disables the operations of all other users, starting from the operation executed last. In the situation shown in Figure 5.4, some remote operations are disabled, indicated by a flat icon (3) while enabled operations are sunken. Starting the replay or browsing the history will now apply enabled operations only to the workspace. Thus, the workspace shows the evolution of an alternative state. Similarly, the evolution of certain objects selected in the workspace can be tracked while all other objects remain unchanged.

Moreover, single operations can be disabled or enabled by switching to the change mode (4) and clicking on the icons. In case the execution time of a disabled operation lies before the time of the current state, the workspace is updated immediately to reflect the modified history. As described above, this can be realized either by executing a timewarp on the basis of the changed history or, as in the case of the mlb, by undoing the concerned operations. In some cases, it might also be necessary to disable subsequent operations together with the one disabled by the user. For instance, when disabling a create operation, all other operations for the same object need to be disabled as well.

Disabling certain operations affects only the local view, i.e., the current shared state is not affected. But it might happen that the user creates a state that she actually prefers. For instance, she might not have issued a certain event if she had known about the modifications of another user, e.g., in the situation given in Figure 4.1, she might not have changed the object's color. Or perhaps operations were issued on the basis of a state that suffered from a short-term inconsistency (see Section 4.4). For such cases, a convenient way to alter the shared state is provided: The state displayed in the user's local view can be finalized by

pressing the apply button (5). Then, the local state becomes the new shared state for all users and is propagated to all sites.

Again, this mechanism is realized using the undo functionality of the application: For each disabled operation, the corresponding semantic undo operation is created and distributed to all session members (see Section 4.8). Thus, the original operation history is not modified but new operations are appended to the history, become visible in the representation, and can be analyzed and undone if desired. As for undo, only disabled local operations can be finalized. In other words, selective local undo is realized via the visualization of the operation history. While being straightforward for discrete applications, this mechanism might be rather difficult (or even impossible) to realize in the continuous domain. Implementing the propagation of a modified state via timewarp instead would require to notify all remote sites about the events to be undone so that these can execute a timewarp on the changed history. This is considerably more complex than the semantic undo approach and has the severe drawback that all sites need to store the same parts of the event history. And in order to redo undone events at a later point in time, undone events cannot be removed from the operation history.

5.5.4 Summary of Visualization Techniques

Summing up, icons indicate whether an event is local or remote, conflicting or not, is executed or not, is the last event executed, is enabled or not, and is selected or not. Although it would be possible to encode even more information into an icon such as the operation's type (e.g., create, move, etc.) or the responsible user, this would increase the complexity of the representation. Instead, detailed textual descriptions for each operation are provided via a tool-tip window (see Figure 5.4 (8)).

Besides giving an overview, the operation history allows to replay past actions and to explore alternative states as described above. The information contained in the history can be analyzed in a time-based, object-based, or participant-based fashion. When the user interacts with the representation of the history, the (local) content of the workspace is updated immediately. This connection between the history window and the workspace window is intensified by highlighting the corresponding parts when an object or an operation icon is selected.

5.5.5 Lessons Learned and Discussion

The properties of the visualization scheme devised in this thesis were evaluated in preliminary experiments with two participants. The users' task in these experiments was to furnish a small apartment by placing furniture from a given set (see Figure 5.2). In this setting, it is likely that the participants move the same piece of furniture concurrently and raise a conflict. By

introducing artificial network delays, conflicts could be triggered explicitly. The experiments indicate that the visualization mechanisms provide sufficient information for the users to notice and analyze conflicts. Especially the slider-based browsing of the operation history together with the possibility to review the operations of a certain participant proved to be very efficient. In this context, we also plan a feature that allows to save different versions of the application's state so that the user can quickly compare alternative states.

However, some shortcomings were discovered as well. First, the user has to acquaint himself with the different visualizations. In an earlier prototype, the history was displayed only when a conflict occurred or when requested by the user. But we discovered that a permanent view of the history together with immediate updates and tool-tip descriptions of icons accelerated the user's learning process.

When the remote users continue to issue operations, analyzing the history might be too slow so that the local user permanently lags behind. An analysis might also be difficult when multiple operations conflict. Furthermore, changing the shared state by disabling operations is problematic when several users do this at the same time. In such situations, it is likely that new conflicts emerge, and that the resulting state violates the users' intentions once more. In order to lower the probability for repeated conflicts, it is not allowed to disable remote operations when finalizing a state. The risk for new conflicts might also be reduced by indicating which users are currently reviewing the history so that users can coordinate.

In sessions with many members and a high activity, the operation history might become rather large, making its visualization difficult to analyze². While the size of the operation history can be reduced significantly by replacing longer operation sequences as described in Section 4.5.4, it might still be too large. Thus, we provide an overview mode (see Figure 5.4 (6)) where homogeneous operation sequences are subsumed and displayed as a single icon (7). A sequence is homogeneous when all operations target the same object, do not conflict, and originate all either from the local user or from remote participants. For instance, subsequent move operations to place an object are combined into a single icon.

5.6 Conclusions

We believe that the success of a consistency control mechanism for distributed interactive applications depends on both technical and human factors: Besides satisfying the formal consistency and correctness criteria, being scalable and easy to implement, the resulting behavior of the application as well as the calculated shared state must meet the expectations of

²By maintaining separate histories for each page, the mlb lessens this problem (see Section 4.7.1).

the user. In this chapter, we therefore discussed semantic conflicts in the actions of different session members. In order to provide awareness information about operations, objects, participants, and about the effects of the consistency control algorithm, a visualization scheme for the operation history was devised. This graphical representation also allows to replay past actions and to explore alternative states. With this information, the user can detect and analyze past conflicts, and may be able to prevent future ones. This approach is independent of a specific application and can be implemented as a generic service.

The visualization of conflicting operations led us into a novel research area: The semantic analysis of the data streams of distributed interactive applications. The meta-information extracted from such data streams could also be used to identify interaction patterns among users (e.g., who interacted with whom for how long), to create general session statistics (e.g., number of members and duration), and to index and access recorded sessions (e.g., search for the modifications of a certain participant), which is mostly time-based up to date [115] (see Section 7.4.3).

In the future, we plan to conduct a more thorough evaluation of different conflict scenarios in order to address the shortcomings of our approach as discussed above. Such user studies could also help to quantify the benefits of the visualization mechanisms with parameters such as task completion time, success, user behavior, and user satisfaction [102]. Moreover, it would be interesting to employ our visualization techniques for other distributed interactive applications. Besides continuous applications, applications that support asynchronous collaboration might benefit in particular since conflicts are much more likely when the propagation delay of operations is not close to real-time (see Sections 2.4.3 and 4.7.2). This aspect is also discussed in the next chapter where algorithms to update application instances that missed parts of the operation history are presented.

Chapter 6

Support for Late-Joining Session Members

Distributed interactive applications often support dynamic groups where users may join and leave at any time (see Section 2.3.1). But a participant joining an ongoing session has missed the data that has previously been exchanged by the other session members. It is therefore necessary to initialize the application instance of the late-comer with the current shared state. Only thereafter the user is able to participate in the session. For example, a user late-joining an mlb session did not receive the actions that lead to the creation of the document page currently displayed, and the application instance of the late-joining member needs to be provided with the current state of that page. An initialization is also required for recovery purposes in case of hardware or software failures (e.g., temporary network outages).

The late-join problem is challenging in particular if the state of a distributed interactive application is large and complex. Most existing distributed interactive applications implement some form of support for late-comers without further investigation of alternatives and consequences. As we shall show, this may lead to high network and application loads as well as to consistency problems, which could be prevented with an appropriate late-join mechanism.

In the following, the late-join problem for distributed interactive applications is investigated in detail. A generic solution which is scalable, robust, and flexible is proposed: Scalability and robustness are reached by a distributed algorithm in combination with group communication. In order to be flexible, a policy model allows a late-joining site to structure and optimize the initialization process according to its specific needs. Our late join algorithm is applicable to discrete and continuous interactive applications; as in the previous chapters, these differ mainly with respect to their consistency requirements.

The remainder of this chapter is structured as follows: First, the design space for late-join algorithms is discussed by means of a general classification scheme devised in this thesis. In Section 6.2, existing work is outlined. Following, a novel late-join algorithm is proposed in Section 6.3. Possibilities to ensure the consistency of the shared state in late-join situations are examined in Section 6.4. The properties of this algorithm in comparison to design alternatives are evaluated in simulation studies in Section 6.5. How the late-join algorithm can be used by different applications is demonstrated in Section 6.6. In Section 6.7, the late-join algorithm is extended to applications that allow synchronous as well as asynchronous collaboration. The discussion of late-join algorithms is concluded in Section 6.8.

6.1 Design Space of Late-Join Algorithms

The task of a late-join algorithm is to initialize the application instance of a late-joining participant with an appropriate amount of data. For easier discussion, we denote the late-joining participant as *late-join client* and any site providing information to the late-join client as *late-join server* [84]. The role of a late-join server may be assigned dynamically by the late-join algorithm, and an application instance might be both late-join client and server for different parts of the shared state.

More specifically, a late-join algorithm needs to address the following questions: Who provides information to the late-join client, what type and amount of data is used for the initialization, when does the transmission of information take place, and how is that data distributed? After identifying important criteria that should be observed, these questions are investigated in detail.

6.1.1 Design Criteria

A late-join should lead to a *consistent state* at the late-joining site as defined in Section 4.1. This state should enable the late-join client to actively participate in the ongoing session. Although a distributed interactive application has to employ a consistency control mechanism in any case, it might be necessary to include some supplemental algorithms for late-join situations. Furthermore, the algorithm should be *robust* against possible failures in the initialization process.

For a late-coming participant, it is desirable that the time span between joining a session and being able to interact with the application is short. An optimization of the *initialization delay* is particularly important if the state of the application is complex.

The transmission of data due to a late-join causes additional network traffic. Therefore another design goal is to minimize the *network load* during an initialization. This is particularly important for sessions with many members where in general only a small fraction of all members will actually need the transmitted information. Finally, the late-join algorithm itself should cause a low *application load* in terms of memory and processing power, especially for those application instances acting neither as late-join client nor as server. As some of these design goals might be contradictory, a late-join algorithm has to find appropriate compromises.

6.1.2 The Source of Late-Join Data

Distributed interactive applications employ a peer-to-peer architecture where each application instance stores sufficient information about the application's shared state (see Section 2.1). For the initialization of a late-join client this implies that data in general could be provided by a number of session members. Thus, a late-join algorithm needs to specify the set of late-join servers, and, should there be more than one possible server, determine the server that is responsible for a certain initialization process. The set of potential late-join servers can either be predefined or determined on demand during the session.

A straightforward approach would be to select a single site as late-join server that is well-known to all participants of a session. This late-join server could either be a regular application instance (e.g., the participant that initiated a session) or a specialized site that is part of a supporting infrastructure. The main advantages of this approach are that the server does not have to be selected in the ongoing session, meaning also that there is no additional delay for selecting a server, and that the application and network load for the other sites remain unaffected. However, a single server has the typical drawbacks of centralized systems (e.g., potential performance bottleneck, single point of failure, see Section 2.1) and also increases the initialization delay for late-join clients with a high end-to-end delay to that server.

An alternative approach is that multiple late-join servers share the responsibility for delivering data to late-join clients. Possible servers could either be all sites holding a certain piece of information or a subset thereof, and servers could either be a regular site or a specialized instance. Such a distributed algorithm is able to spread the network and application load for the initialization of late-join clients and is more robust against failures. However, the late-join algorithm now has to determine the site(s) that handle(s) a certain initialization process. Also, the initialization delay for the late-join client might increase depending on the selection mechanism.

6.1.3 The Extent of Late-Join Data

The task of a late-join algorithm is to provide a late-comer with sufficient information about the current shared state of the application so that the concerned user is able to participate in the ongoing session. One possibility is that the late-join client receives all data that is necessary to calculate the complete state. This keeps the late-join process simple. But in case the application's state is large, the resulting initialization delay as well as network and application load for client and server might be high. These could be reduced considerably when the late-join client is initialized only with those parts of the shared state currently visible to the user. For example, a late-joining participant of a shared whiteboard session would be supplied only with the active page. When employing such a partial initialization, the late-join algorithm has to determine whether any remaining parts of the shared state need to be delivered, and, if so, in which order and when. In the shared whiteboard example, the late-join client would need additional data when a page is activated that was created before he joined the session. Depending on the application and the session, it can also occur that a late-join client never needs a complete copy of the shared state. In order to allow such decisions for a late-join algorithm, objects need to be classified with respect to their priority and content. In the following, we denote the decision when to provide the late-join client with which parts of the application's state as *late-join policy* [261].

In addition to determining which parts of the application's shared state are necessary for an initialization, a late-join algorithm also needs to specify in which form that data is to be provided. The different possibilities are depicted in Figure 6.1: Initialization information can be transferred either by a replay of the operation history, by a copy full state, or by a combination of state transmission and a short list of subsequent operations.

A replay of the operation history would require the late-join client to calculate the current state from the original sequence of operations, starting either from an empty or a well-known initial state (see Figure 6.1 (i)). The main advantage of this approach is that it can easily be implemented in an application-independent way since a replay does not require any specific knowledge about the operation sequence (see also Sections 6.2, 7.4.3, and 4.5). However, a replay of the operation history also has several drawbacks: First, it is inefficient since a large part of the transmitted information may no longer be relevant for the current state. For example, an image on a shared whiteboard page that has been deleted later is not relevant anymore. In general, it is more efficient to transfer state information than to transmit all operations that have lead to that state. When editing a text it makes sense to transmit the state of the text instead of all insert and delete operations that lead to the current state. This becomes even more important when the overhead for packet headers is taken into account.

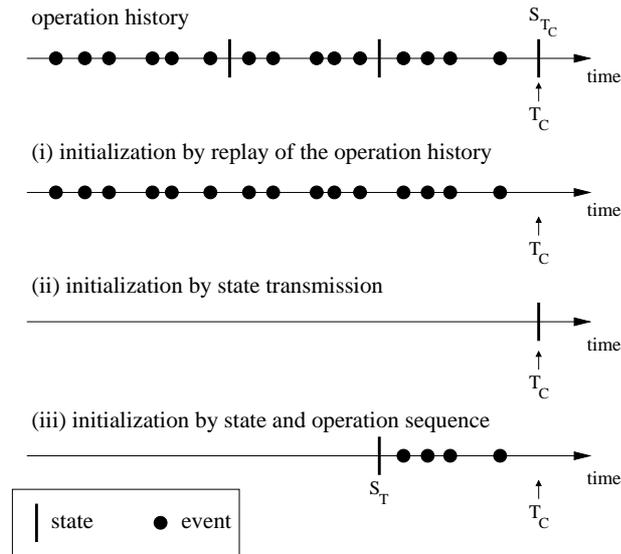


Figure 6.1: Different types of late-join data

Second, the application either has to be able to reconstruct every operation that has ever been transmitted since the beginning of the session, or all operations need to be buffered indefinitely. This is clearly not acceptable for a large number of applications.

Third, for continuous applications the state may not be reconstructible from a simple replay of operations since operations are only valid at their given execution time. Thus, the application would need an algorithm such as timewarp in order to reconstruct the state of such an application from outdated operations. This is not the case for all continuous applications.

Under these considerations, a replay of the operation history for initialization purposes is only appropriate in cases where the application needs a complete history to fulfill other tasks as well, such as a local replay functionality.

Initializing the late-join client with a copy of the current state S_{T_C} is in general the most efficient way with respect to the transmission delay of the data, the network load, and the application load of the client (see Figure 6.1 (ii)). The biggest challenge of this approach lies in ensuring that the initializing state is consistent (see Section 6.4).

The last alternative is a mixture of state transmission and a replay of a small part of the operation history (see Figure 6.1 (iii)): Instead of the current state, an older state S_T is used as a starting point, followed by a sequence of operations that were issued between the time S_T was valid and the current time T_C . This approach can be used to improve a late-join algorithm with respect to consistency issues and will be discussed in detail in Section 6.4.

In the following, we denote any initialization information that is provided to the late-join client as *late-join data*, and the request for such information as *late-join request*.

6.1.4 Distribution of Late-Join Data

The exchange of data between late-join client and server can either be realized via a direct point-to-point connection or by means of group communication [84]. A unicast connection ensures that no other session member is involved in the late-join process, thus minimizing the overall network and application load. On the other hand, unicast connections limit the number of clients a single application instance can serve at a certain point in time. In a worst case scenario, clients would have to wait for a free connection, which increases the initialization delay. In the case of failures, a unicast connection might also be blocked for a certain amount of time before it is available for other clients. Moreover, unicast prevents that more than one late-join client can profit from a particular transmission of late-join data. Finally, the maintenance of a unicast connection generates an overhead that is significant if the amount of data to be transferred is small.

These drawbacks can be avoided or reduced when distributing late-join data via group communication (see Section 2.3.3). At the same time, group communication implies that data is received by all or a subset of all session members, including sites that are not interested in that data. To limit the overall network and application load, it might therefore be necessary to restrict the number of receivers of late-join data by introducing additional communication groups. This aspect will be discussed in Section 6.3.3.

6.2 Related Work

One possibility to realize late-join functionality is to reuse the loss recovery mechanism of the application's reliable transport service: From the protocol's perspective, a late-joining application instance has missed all data packets since the beginning of a session. Repairing this "loss" then provides the late-join client with a replay of the entire application-level operation history, which allows to reconstruct the complete shared state. For instance, the reliable multicast protocol SRM (see Section 2.3.3) supports transport-level late-joins [67]. SRM uses a timer-based feedback mechanism to select a site that is responsible for the transmission of a certain packet [68] so that all session members holding a certain piece of information can act as late-join server. A main advantage of this approach is its robustness: As long as one site has the required information, a late-comer will be able to join the session. Furthermore, the approach is application-independent, and an existing implementation can be used for different applications. For instance, the shared whiteboard MediaBoard [249] integrates SRM (see Section 3.1). However, providing late-join data by transport protocol functionality has some severe drawbacks as pointed out above: It leads to a high initialization delay as well as high

network and application loads. Also, the entire packet history must be kept until the end of the session.

Alternatively, a late-join algorithm can reside on the application level. Here, the distinct advantage is that application knowledge can be used to optimize the late-join process. Application-level approaches can be classified into centralized and distributed. A centralized late-join algorithm requires that a single site exists that is able to act as late-join server for the entire shared state. A late-joining instance may contact this server, which will in turn deliver the relevant state information. Examples where a centralized server is used for late-join purposes are the Notification Service Transport Protocol (NSTP) [191] and the JASMINE system [222].

NSTP provides a generic client-server infrastructure for distributed applications and operates on an abstract data model of states and events [191]: Each participant connects to the central notification server managing the shared state. State changes are sent to the server, which determines a global order among concurrent events and updates all clients. The state of the application can be partitioned into several sets of objects, the so-called places. A client receives only events for those places it has registered for. For the notification server, an object is an abstract attribute-value pair. An event changing the state of an object contains the new value for a certain attribute. Thus, the server can manage the state of an application using the information exposed in the abstract data model, without needing any application-specific knowledge. A late-joining participant has to decide which places it is interested in and receives the state of all corresponding objects from the server. Thus, when compared to the transport-level approach from above, the knowledge about the application's data model improves the performance of the late-join algorithm significantly.

JASMINE is a generic framework for collaborative Java applets and also employs a client-server architecture [222, 221]. Besides collaborative services such as session management, the server provides event distribution similar to NSTP. But unlike NSTP, the server interprets events and maintains the state of the applets, i.e., the server implements the full application-level logic. As a consequence, the server can initialize late-joining participants with the current state by handing over serialized Java objects. The late-join client has to wait until the initialization is completed before it can participate in the session. Depending on the size of the shared state, the resulting initialization delay might be high.

The main advantage of a centralized solution at the application level is its simplicity with respect to the handling of consistency-related issues. At the same time, a single state server results in the typical disadvantages of all centralized solutions (see Section 2.1): Main problems are the existence of a single-point-of-failure (lack of robustness), and the high application load for the server, which might become a bottleneck. Moreover, for applications with a

replicated architecture, it seems to be inappropriate to introduce a centralized server just for late-join purposes.

Alternatively, multiple sites are assigned the role of a late-join server for different sub-sets of objects. This is done by the collaborative virtual environment MASSIVE-3 [92], which employs a peer-to-peer architecture. In MASSIVE-3, a virtual world is structured into interest regions, and an application instance traces only the objects of the regions it participates in. Each interest region is managed by a unique and well-known application instance. This so-called *master* process designates a global order to all operations changing the state of the region of interest it manages. Other sites that issue an operation either have to distribute the operation via the master process or become master for the corresponding region of interest themselves. A late-join client contacts the master process of each region of interest it wants to participate in. Thus, the initialization is limited to parts of the application's state that are relevant for the client. Since the master is well-known, a mechanism to select a late-join server is not necessary. The late-join server creates an initializing operation sequence composed of state and event information.

Partitioning the application's state into regions of interest and assigning a master for each region reduces the danger that a single late-join server becomes a bottleneck. However, it does not increase robustness since a late-join server may still fail or become overloaded if multiple late-joining participants are interested in the same objects at the same time. Thus, we believe that a solution to the late-join problem should not rely on dedicated late-join servers at all.

In distributed late-join approaches, many sites are able to take the role of a late-join server for a given piece of the shared state. Thus, the failure of single instances does not compromise the initialization of late-comers. An example is the Network Text Editor (NTE) [106]. In NTE, text documents are subdivided into lines. If a line is changed (e.g., a character is inserted), the complete state of the line is transmitted via unreliable multicast to all participants. Due to the included redundancy, it is likely that packet loss is repaired via the next state transmission, and a retransmission of a lost state is not necessary. But in some cases, this redundancy approach is not sufficient to guarantee the eventual delivery of all events, for instance, if the last change to a line is lost. In these cases, periodic session messages including information about the current state are used to detect and request lost data. This packet loss repair mechanism is also used to handle a late-join situation: Since state changes concerning the active objects (lines) are distributed via state transmissions, late-comers are supplied with the active text areas very fast. The client detects other lines by analyzing the periodic session messages and requests them in analogy to the loss scenario.

But the transmission of an object's state after a state change can only be justified if the network connection suffers from a high loss rate and if the state itself is rather small. Otherwise, the resulting overhead would place a high burden on both the network and the application instances. On the other hand, multicast transmission of states means that a single state is able to repair more than one packet loss, and it can initialize more than one late-join client at the same time.

The main benefits of distributed late-join approaches at the application level are robustness and scalability as well as the usage of application-level knowledge to avoid the drawbacks of the transport-level approaches. One problem, though, is the lack of reusability. For example, it would be quite difficult to tailor the late-join functionality implemented for NTE for the use with a virtual 3D world or for a distributed Java applet. In the following section, we present our distributed, application-level late-join algorithm that is based on the data model presented in Section 2.2 and therefore can be used by all distributed interactive applications.

6.3 The Late-Join Algorithm

The analysis of existing work leads to interesting insights: To be efficient, a late-join algorithm must make use of application-level knowledge. A replay of all transmitted operations is generally not an acceptable solution. In order to be robust and scalable, the late-join algorithm should not contain any centralized elements. Finally, the implementation of a late-join algorithm should be flexible so that it can easily be used with different applications.

From the perspective of a late-join client, a distributed late-join algorithm is usually composed of the following three steps: (1) The first task for a late-join client is to determine the point in time at which the late-join data of a given object should be requested. If certain objects are more important than others, the client should be able to prioritize them accordingly. (2) In the second step, a late-join server needs to be selected for each late-join request. (3) Finally, the late-join data must be distributed to the late-join client(s).

For the realization of this algorithm, we solve the following four key problems: Selection of sites, use of late-join policies, distribution of late-join data, and consistency control.

6.3.1 Selection of Application Instances

The late-join algorithm operates in a distributed fashion without dedicated late-join servers. Late-join data can therefore be provided by any session member holding a consistent copy. Since it is likely that several session members qualify for being late-join servers, a certain site

has to be selected. Similarly, multiple late-comers may wish to request the same late-join data at the same time. This too should be coordinated so that request implosions are prevented.

Scalable selection of at least one session member from a potentially large group is a common problem in group communication and is known as scalable multicast feedback [183] (see Section 4.6). A feedback algorithm must guarantee that at least one participant is selected. The number of participants selected should be low - ideally it would be exactly one. Finally, the algorithm should be scalable in terms of group size, and introduce only a small delay.

Here, the exponential feedback raise algorithm of Fuhrmann and Widmer [74] is used for the selection of application instances in late-join situations. The selection process is triggered by sending a message via multicast to all candidates (e.g., a late-join request for the selection of a server). Upon reception of that message, each candidate i picks a random number $x \in [0; 1)$. Now two cases are distinguished: (1) If $x < \frac{1}{N}$, where N is the number of candidates, i is selected and acts immediately. (2) If $x \geq \frac{1}{N}$, an exponentially distributed back-off timer is set with running time $t = T_{max}(1 + \log_N x)$ where T_{max} is the maximum desired latency until at least one member must be selected. If a back-off timer expires, the respective member has been selected. In both cases, a selected site transmits a reply packet to the group. All sites receiving this information cancel their own back-off timers. In the ideal case, only one reply is sent. As the analysis of Fuhrmann and Widmer in [74] shows, the exponential feedback raise algorithm has a good behavior with respect to the expected number of selections and the expected selection latency. This was also verified in our own simulations (see Section 6.5).

6.3.2 Late-Join Policies

In general, it is not necessary for an application instance to be initialized with the entire shared state. A prerequisite for such a partial initialization is that the application's state is partitioned into independent objects (see Section 2.2). For each object, a late-join client can then decide when the data for that object should be requested, using its application-specific knowledge. The use of such policies ensures that the generic late-join service can be easily adapted to the needs of the application. Existing late-join approaches lack this ability.

For instance, in a shared whiteboard session a late-coming site typically needs all active objects immediately (i.e., objects belonging to the page currently displayed). In contrast, the state of all passive objects (i.e., objects belonging to currently invisible pages) are usually required only when they become active again. Having this knowledge, an application could decide to request the late-join data of active objects only and postpone the requesting of passive pages to a later point in time. One major advantage of this approach is that the amount of data transmitted immediately is limited, which also minimizes the initialization

delay. Furthermore, an initialization process is spread over a longer period of time, which also spreads the network and application load.

In order to select different policies for individual objects, a site has to explore the name space of objects when joining a session. Besides information about which objects exist, some application-level knowledge about each object is needed to decide about an appropriate policy, i.e., if an object is active or passive and which part of the shared state it represents. For example, the RTP/I protocol presented in Chapter 7 provides such meta data.

For our late-join service, a number of policies are defined for requesting objects [261]: *No late-join*, *immediate late-join*, *event-triggered late-join*, and *network-capacity-oriented late-join*. Policies can be predefined or selected dynamically during a session on different levels of granularity: individually for each object or for object types, and for all or for individual application instances. They may also be changed during a session, depending on the situation.

1. *Late-join policy: No late-join*

The *no late-join* policy is chosen by a site to indicate that it is not interested in a certain object. In a distributed virtual environment, it could be used for objects that the user will never be able to see. This policy reduces the overall amount of information that is required for an initialization.

2. *Late-join policy: Immediate late-join*

A site may choose the *immediate late-join* policy for objects that are currently active. Late-join data for these objects is then requested immediately. The likelihood that several clients may profit from a single transmission of late-join data is rather low with this policy.

3. *Late-join policy: Event-triggered late-join*

For a number of scenarios, it might be reasonable to request an object only if it is the target of an event. As illustrated above, a shared whiteboard might request pages with the immediate late-join policy if they are currently active. Other objects could be requested at the time they become the target of an “activate page” event. This request strategy is implemented in the *event-triggered late-join* policy. Besides deferring requests until the data is actually needed, this policy also significantly increases the likelihood that multiple late-comers benefit from a single transmission (if multicast is used for data distribution) since all clients will encounter the triggering event at the same time. Thus, all sites who join a session between two successive events for a certain object will benefit from a single transmission of the initialization information.

Figure 6.2 shows the finite state machine for the event-triggered late-join policy. When the late-join service learns about the existence of an object *obj*, the machine is in its initial state `object discovered`. At the time a policy is chosen, the late-join state changes to

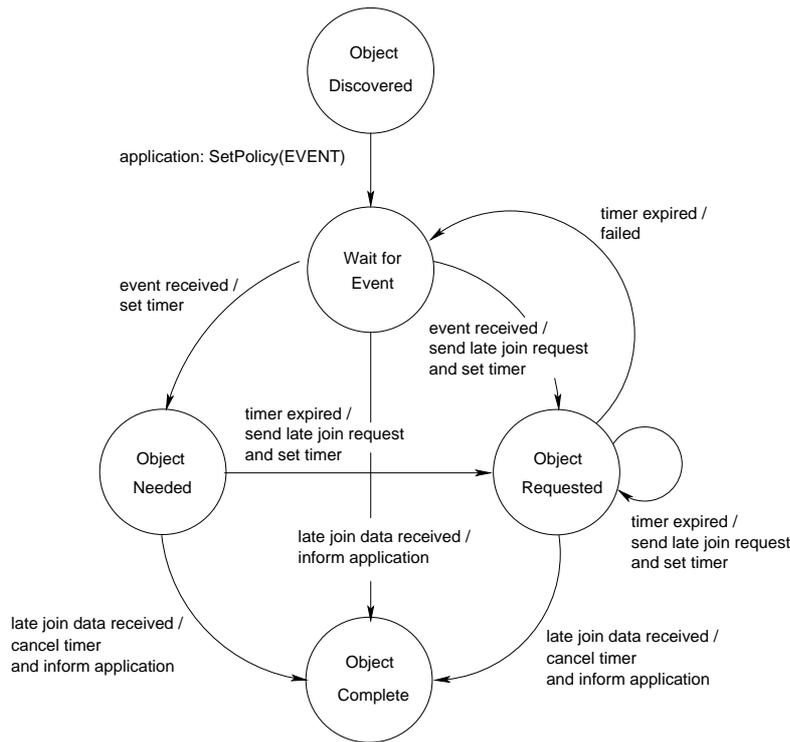


Figure 6.2: Event-triggered late-join

`wait for event`. While in this state, the late-join service checks all incoming operations whether they target `obj`. In case sufficient information for the initialization of `obj` is received, it is delivered to the application, and the late-join state becomes `object complete`.

When an event for `obj` is received while in the `wait for event` state, a site is selected to send a request by means of the exponential feedback raise algorithm. This is necessary since there might be more than one client requiring late-join data for `obj`, in particular when using the event-triggered policy. Based on the results of the selection process, a client either sends the request at once (`object requested`) or sets a back-off timer and makes the transition to `object needed`. If the late-join data for `obj` is received while the timer is running, a transition to `object complete` is performed. Otherwise, when the timer expires, a request is transmitted, another timer is set, and the late-join state changes to `object requested`. The additional timer is for reliability purposes and protects the transfer of data in case the application does not use a reliable transport protocol. When repeated late-join requests for state information fail, the application is informed.

4. Late-join policy: Network-capacity-oriented late-join

For objects where late-join data is not immediately required, the *network-capacity-oriented late-join* policy may be chosen. With this policy, the late-join service monitors the incoming and outgoing network traffic. Only if this traffic falls below a threshold set by the application, late-join data is requested. Similar to the event-triggered late-join, requests of several late-

join clients can be clustered so that a single transmission of initialization data may serve multiple clients.

Other policies are conceivable and can be integrated easily into our late-join service. Naturally, all late-join policies delaying the transmission of initialization information increase the probability that the last participant that possesses that data leaves the session. Most distributed interactive applications discussed earlier apply the immediate late-join policy. The MediaBoard additionally orders all requests according to their priority for the user [249].

6.3.3 Distribution of Late-Join Data

After a late-join request is issued and an appropriate server is selected, the late-join data needs to be transmitted to the late-join client(s). As discussed above, this data should be distributed by means of group communication (i.e., application-level or IP multicast). In the following, we investigate the implications when using either one, two, or three multicast groups for the exchange of late-join requests and data [262].

6.3.3.1 Variant 1: One Multicast Group

The easiest solution is to transmit late-join data to the same multicast group (the so-called *base group*) as the regular session data. All applications presented in Chapter 6.2 choose this approach. Its main benefit is its low complexity. But at the same time, all sites and not only late-join clients will receive late-join data. This may result in large amounts of data being delivered to application instances that do not need it.

6.3.3.2 Variant 2: Two Multicast Groups

Alternatively, a separate late-join multicast group for the transmission of late-join information can be established [76, 84]. This group is denoted as *client group*, and all late-join clients are members. Requests still have to be distributed via the base group in order to find a late-join server. But the late-join data is then sent to the client group only.

As soon as a client does not expect to receive further late-join information, it should leave the client group. However, preliminary simulations have shown that depending on the application, the late-join policy model, and the user behavior it is very unlikely that a late-join client will ever complete the late-join process for all objects. For example, consider a shared whiteboard session where a set of slides is presented. Even if the lecturer jumps back to an older slide every once in a while, it is unlikely that all slides are presented more than once in a session. Therefore, a client should leave the client group when it has not requested or received any late-join information for a certain period of time. Should the client discover at

a later point in time that it needs the late-join data of an object (e.g., an older slide has been reactivated), it simply rejoins the client group.

Restricting the receivers of late-join information to application instances that probably need the data is expected to reduce both network and application load for the entire group. However, it introduces additional costs for the management of the client group.

6.3.3.3 Variant 3: Three Multicast Groups

Distributing late-join state requests over the base group as described above has two drawbacks: First, requests are received and processed by each session member. This places a burden on both the network and the end-systems. Second, if a large number of sites receive a request, there is a higher probability that the selection process will pick more than one application instance as a late-join server, even with the exponential feedback raise algorithm. Because a transmission of late-join data is costly for the application (e.g., it must extract the current state of an object) and the network, a major goal for a late-join algorithm is to minimize the number of duplicate server selections.

Distribution of requests can be restricted by using a third multicast group: If a limited number of potential late-join servers form an additional multicast group, requests can be transmitted directly to this *server group*. If a server has been selected for a certain request, he sends an acknowledgment to the server group in order to suppress other selections and transmits the requested late-join data to the client group. Provided that the participants of the server group are chosen well, they can provide all late-comers with the desired data while the majority of application instances is not involved in the server selection process. Because fewer sites receive a request, it is to be expected that network and application load due to requests and duplicate data can be reduced.

This approach raises two questions: First, who should be a member of the server group? And second, what happens if a request fails because no server for the requested object is present in the server group? The latter problem can be solved by a second request round: If no server can be found in the server group, the same request is forwarded to the base group. While this introduces an extra initialization delay, it guarantees that requests are eventually answered.

The first question is more complex, and there are a number of criteria, which need to be considered for an algorithm that decides upon the membership in the late-join server group: (1) Ideally, each object for which information is likely to be requested should have a server in the late-join server group. This reduces the initialization delay for late-join clients. (2) The size N^{SG} of the server group should be as small as possible. The smaller N^{SG} , the less network traffic is generated, and the fewer sites are involved in late-join management. A small N^{SG} also decreases the likelihood that more than one late-join server will reply to a

request. (3) The number of join and leave operations should be small for the server group since each of these operations is associated with overhead at the network layer (e.g., multicast routing). (4) The burden to act as a late-join server should be shared by all sites. (5) And the algorithm should cause only little computation and communication overhead.

In the following, three different approaches are considered to decide which sites should participate in the late-join server group: Distributed, isolated, and application-controlled.

Distributed membership management

In distributed membership management, all sites exchange information about their capabilities to act as a late-join server so that an optimal set of servers can be determined. For instance, sites that are able to provide late-join server functionality for many objects should be preferred as members of the server group. The main drawback of the distributed membership management is its complexity: Application instances need to exchange and process supplemental information, which may lead to a significant overhead, especially for large sessions. For this reason, distributed membership management is not taken into account for our late-join service.

Isolated membership management

Isolated membership management seeks to avoid additional messages and processing overhead by using local information: Each site decides on its own whether it should join or leave the late-join server group. A site leaves the server group if it has not answered any requests for a certain amount of time t_l ¹:

$$t_l = T_m \left(\gamma \frac{O_p}{O} + (1 - \gamma) \frac{R_a}{R} \right) \quad (6.1)$$

where T_m is the average group membership time (provided by the application), and O_p denotes the number of objects the site can provide as a late-join server. O_p is set in relation to the total number of objects O present in the session. The intention is that sites that are able to serve a large percentage of all objects should stay longer in the server group. R_a is the number of late-join requests that actually have been answered by the server. This number is set in relation to the number of requests R that could have been answered. The lower this percentage, the less important is the server. Finally, $\gamma \in (0; 1)$ trades the importance of the number of available objects against the number of answered requests.

This approach seeks to build a late-join server group with a small number of “powerful” servers. But it cannot guarantee that a server is found for each request. Thus, there must also be a way to let sites (re-)join the server group. If a request remains unanswered in the

¹Please note that an application instance also leaves the server group when the user leaves the session.

server group, the late-join client initiates a second request round in the base group. All sites that are able to become a late-join server for the requested object use a feedback mechanism to decide who will actually join the server group: In order to select a preferably powerful server, the *biased* exponential feedback raise algorithm of Widmer and Fuhrmann [270] is used that extends the feedback mechanism discussed in Section 6.3.1. In addition to the random value $x \in [0; 1)$, the back-off timer t now also depends on the number of objects a candidate possesses:

$$t = T_{max}(\gamma(1 - \frac{O_p}{O}) + (1 - \gamma)(1 + \log_N x)). \quad (6.2)$$

Thus, the more objects a site i can serve the shorter is t . As Widmer and Fuhrmann show in [270], this increases the probability that a powerful server is selected significantly. Moreover, the selection latency is reduced when compared to unbiased feedback. The selected server then transmits an acknowledgment to the base group, enters the server group, and sends the requested late-join data.

If a second round for a certain late-join request is necessary, the initialization delay for the late-join client increases. For distributed interactive applications where a low initialization delay is crucial, the application can also define a minimum group size N_{min}^{SG} for the late-join server group, i.e., servers are allowed to leave only as long as the current group size N^{SG} is higher than N_{min}^{SG} . This results in a higher probability that a request can be answered without a second round.

Application-controlled membership management

In some cases, the application may want to decide explicitly who should join the server group rather than leaving this decision to the late-join algorithm. For example, when the application employs a floor control mechanism, only the floor holder may be selected to transmit the late-join data for an object. Since there is only one candidate for joining the server group, the late-join service allows to specify a site that should immediately enter the server group if a request targets a certain object. In this case, the site would leave the server group as soon as it loses all floors.

6.4 Consistency Control

The late-join algorithm has to ensure that the state of a late-join client is consistent after the initialization. Depending on the consistency control mechanism that the application uses for the exchange of “regular” operations (see Chapter 4), additional measures might be necessary to ensure consistency in late-join situations.

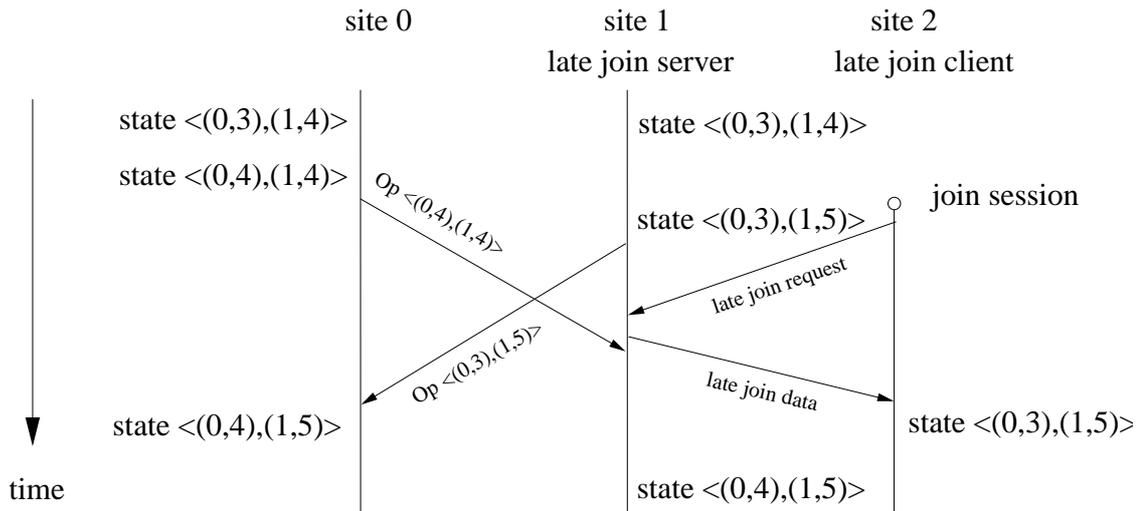


Figure 6.3: Inconsistency in a late-join situation

One option for a distributed interactive application is to employ a pessimistic consistency control mechanism such as floor control. In this case, there exists a dedicated application instance for each object, which controls user-initiated state changes and determines a global order for the operation sequence. Thus, an obvious approach is to select the controlling site as the late-join server for a specific object. If a late-join server delivers after the late-join data all operations that have been issued during the initialization and that were not covered yet, no supplemental consistency mechanisms are necessary, independent of how late-join data is distributed².

In the following, we focus on applications that repair inconsistencies on the basis of local information (e.g., with operational transformation or timewarp), and that allow concurrent operations. But as discussed in Section 4.6, a site is not able to guarantee the consistency of its local state. Consider the example of a discrete application with three sites as shown in Figure 6.3: At the beginning, only sites 0 and 1 are members of the session, and both have the same state specified by the state vector notation $\langle(0, 3), (1, 4)\rangle$, i.e., the current sequence numbers of sites 0 and 1 are 3 and 4, respectively (see Section 4.1). Then site 0 issues an operation with the state vector $\langle(0, 4), (1, 4)\rangle$. Before that operation is received, site 1 also changes the state ($\langle(0, 3), (1, 5)\rangle$), and site 2 joins the session. The late-join request of site 2 is answered by site 1 even before the operation $\langle(0, 4), (1, 4)\rangle$ is received. In this example, late-join data is provided in the form of a state. The late-join client now holds the state $\langle(0, 3), (1, 5)\rangle$, and even when he discovers and/or receives the missing operation $\langle(0, 4), (1, 4)\rangle$, the inconsistency cannot be repaired since the client lacks the required information to do so.

²In the continuous domain, it could happen that the late-join client receives an event too late. But since this can happen for regular events also, the application has to implement a repair mechanism (such as the timewarp algorithm presented in Section 4.5) for this occasion anyway.

A late-join algorithm must therefore provide a mechanism to discover this problem and guarantee a consistent initial state. After such a state has been established at the client, the regular consistency control mechanism will take over, using this state as its starting point. In the following, mechanisms to establish consistency are devised for the different options to deliver late-join data as described in Section 6.1.3.

6.4.1 Initialization by Replay of the Operation History

In case a late-join client is initialized by replaying the operation history, the application needs to calculate the consistent state from that operation sequence in a fast forward mode (see Figure 6.1 (i)). A first condition is a complete replay. The late-join client can detect and request missing operations, e.g., by the sequence numbers of operations.

For discrete applications, the correct order in which operations have to be executed must be observed to reach a consistent state. Should the client receive operations out of order either because the underlying transport protocol does not ensure a source order of packets or because the late-join server did not have a consistent history, the correct order of that sequence needs to be restored, e.g., by serialization.

In the continuous domain, operations are only valid at a certain point in time. Thus, the application has to reconstruct the state from outdated operations. For instance, with the timewarp algorithm the application jumps to the execution time of each operation before processing it. As long as outdated operations can be handled, no additional measures are necessary for late-join situations.

When initializing a late-join client with a replay of the operation history, the method of data distribution (i.e., using one, two, or three multicast groups) has no effect on eventual consistency.

6.4.2 Initialization by State Transmission

With respect to the initialization delay for the late-join client as well as to the network and application load, the most efficient way to provide initialization information are object states (see Figure 6.1 (ii)). But since a late-join server cannot guarantee that his local state copies are consistent, additional measures are mandatory, depending on the method of data distribution³.

³If we assume that under normal conditions a high percentage of all instances holds a consistent state and that sites that suffer from a known inconsistency (e.g., repeated packet loss) do not answer to state requests, one would expect only few cases where such measures are actually necessary.

6.4.2.1 Iterative State Transmissions

When distributing late-join states via the base group (as in the first variant proposed in Section 6.3.3), inconsistent states can be discovered if all sites check the states they receive against their local state copies. This can either be done by comparing the states themselves or by using supplemental meta data identifying which operations are included in that state, e.g., state vectors as defined in Section 4.1. This comparison can lead to the same results as with the iterative state transmission described in Section 4.6: In case a site receives a state that differs from its own, it either adopts the received state or sends its own. After a limited number of iterations this will result in all participants (including the late-join client) having a consistent state.

The main benefit of the iterative state transmission scheme is that inconsistencies can be repaired in a short period of time, and that after execution of the algorithm all sites hold a consistent copy of the application's state. On the other hand, each session member has to compare its local state to each received state, which might be costly on processing power. And iterative state transmission may result in a high network load.

6.4.2.2 Iterative State Requests

In case late-join states are transmitted via a special multicast group in which only late-join clients participate (as in variants two and three described in Section 6.3.3), the iterative state transmission would fail since states are received exclusively by late-join clients. Even if there exist some clients that already hold parts of the shared state, consistency of the application can be guaranteed only if all sites perform the state comparison. Thus, the iterative state request mechanism described in Section 4.6 is used where the late-join client is responsible to check whether or not a received state is consistent. For this purpose, each initial state has to carry meta-data about all operations included, e.g., a state vector. The late-join client compares this meta-data with the information given in session messages, which are sent periodically to the base group by all participants. If this check indicates that the late-join client has received an inconsistent state, then that state is discarded and requested again. If necessary, this is repeated until the check is successful.

Compared to the iterative state transmission approach, discovery of inconsistencies by periodic session messages and iterative state requests might take longer until an inconsistency is discovered and repaired.

6.4.3 Initialization by State and Operation Sequence

One major advantage of consistency control mechanisms such as operational transformation and timewarp is that they allow a participant to repair inconsistencies on the basis of local information and do not require additional communication. As a prerequisite, each site has to store a certain amount of data in the form of an operation history. In order to enable a late-join client to repair possible inconsistencies locally, this information would have to be transferred to the client. But as pointed out in Section 6.1.3, a replay of the complete operation history is highly inefficient for most applications. On the other hand, providing a copy of the current state means that the late-join client will not be able to repair inconsistencies locally in situations like the one depicted in Figure 6.3. Instead, the client relies on receiving data from other sites until the extent of information that he stores locally is sufficient for the regular consistency control mechanism to take over. In the case of timewarp, this point is reached when the client does not receive any operations with an execution time that lies before the earliest state stored in the operation history.

The probability that a late-join client is able to repair inconsistencies locally can be increased by combining the approaches of operation history replay and state transmission: Instead of transmitting the current state, the server initializes the client with an older state S_T where T lies before the current time T_C , and a sequence of operations $\{O_t | t \geq T\}$ (see Figure 6.1 (iii)). This provides the late-join client with a basis for the consistency control mechanism. The smaller T is, the higher is the probability that a client is able to repair inconsistencies locally, and additional state transmissions are not necessary. But at the same time, a small T also increases the initialization delay as well as the network load. However, under regular network conditions with a low packet loss rate, remote operations are usually received in a very small time span. Thus, it is likely that this approach will reduce the number of iterative state transmissions even when the difference $T_C - T$ is small.

Even situations like the one shown in Figure 6.3 can now be repaired by the late-join client without an additional state transmission if $SV_{S_T} < \langle (0, 3), (1, 4) \rangle$: The client discovers that there is a gap in the sequence numbers, e.g., by periodic session messages or by a subsequent operation of the same sender. To repair the inconsistency, it is now sufficient to request the missing operation and to restore the correct order of operations.

6.5 Simulation Results

The generic late-join service described above is implemented in C++ (see Section 7.4.2) and realizes the dynamic source model where each site may be a late-join server. Late-join data is currently provided in the form of states, but that could easily be changed to the

other data distribution strategies. Requests for late-join data are managed via the flexible late-join policy model. For deciding on an appropriate policy, meta-data as described in Section 6.3.2 is exchanged periodically among all sites. For the delivery of late-join data, all three variants with either one, two, or three multicast groups for the communication between clients and servers are implemented. The composition of the server group for the third variant is determined by the isolated membership management together with the adaptive server timeout.

Based on this implementation, we developed a simulation toolkit that allows us to analyze different scenarios with respect to the design criteria discussed in Section 6.1.1 [262]. In the following, a shared whiteboard session and an online game scenario are simulated. In order to base the simulations on realistic end-to-end delays, network topologies are generated with the Georgia Tech Internetwork Topology Models (GT-ITM) [22] toolkit. The topologies use the transit-stub method, which defines a two-level network with transit domains as the network's backbone and stub domains that host the end-systems. Edges between nodes are placed by the random model, and the generator's option to introduce extra transit-stub or stub-stub edges is disabled. All application instances are located on the edge of the network, and all inner nodes act as routers. Data packets are routed according to the Dijkstra algorithm [46] on the basis of the end-to-end delays. All packets are transmitted reliably⁴.

The simulations are event-based, and each site periodically generates an action from the set $\{join\ session, leave\ session, create\ new\ object, send\ operation, activate\ object, deactivate\ object, do\ nothing\}$ with predefined probabilities. The time span between two actions of the same site is randomized. Depending on the action, data is exchanged with the appropriate end-to-end delays, and late-join situations might occur. In order to reduce uncertainties that influence the outcome of a certain simulation setting, actions are generated only once for each scenario and stored in a file, so that each run can use the same sequence of actions as input. The results for a certain scenario that are discussed below represent the average of ten simulation runs with the same action file.

First, the impact of the distribution model on the design criteria is analyzed for the shared whiteboard and the online game scenario. Following, the effects of late-join policies are studied. The last set of simulations examines how different sets of parameters for the composition of the late-join server group in the third variant of late-join data distribution influences the simulation results.

⁴In case the application does not use a reliable transport protocol, the late-join algorithm can repair packet loss by repeating the server selection process (see above). This will increase the initialization delay.

6.5.1 Distribution of Late-Join Data

6.5.1.1 Shared Whiteboard Scenario

For the simulation, a shared whiteboard session is set up with a maximum of 100 participants. The network topology is chosen such that the end-to-end delays are distributed evenly between 3 ms and 480 ms, with an average delay of 230 ms. This allows us to select a maximum running time T_{max} of 500 ms for the exponential feedback timers (see Sections 6.3.1 and 6.3.3.3).

The model of a shared whiteboard is based on the experience gained with the mlb. The application's state is structured hierarchically (see Section 3.3). At a certain point in time, the same page is visible to all users. Thus, only a small subset of all objects is actually active at any given time. Operations can be issued only for those active objects. A good strategy for a late-join client in this scenario is to request the state of the current page (i.e., all active objects) with the immediate late-join policy. All other objects are not requested unless they become active.

Periodically, each application instance generates an action as described above with predefined probabilities. The time span between two actions of the same site is randomized between 1.5 s and 4.0 s. The probabilities for the individual actions are chosen such that a realistic session with a high user activity is reproduced. The probability to create a new page is set to 0.0005 and the probability to create a new graphical object to 0.018. During the simulated time of 15 minutes, this leads to 14 pages created, and a total of 572 graphical objects. On 11 occasions, an older page is reactivated (corresponding to a probability of 0.0002), and 3,300 times the state of an object is modified (corresponding to a probability of 0.12).

Typically, a shared whiteboard group is relatively stable during the presentation, with only a few members joining or leaving. Therefore, the simulated session starts with 80 members, which is also the minimum group size throughout the session (i.e., an instance is only allowed to leave if there are more than 80 session members remaining). After initiation, there is a dynamic phase while additional participants join with a relatively high probability; towards the end, members leave more frequently. This behavior was reflected by exponentially decreasing the join probability (starting with 0.02), and by exponentially increasing the leave probability (ending with 0.001). During the simulation, 20 participants join the ongoing session and 16 leave the session early. Almost all joins happen in the first 4 minutes of the session, but one occurs at 6.5 minutes, and two more after approximately 10 minutes.

For the third variant, the minimum size of the server group N_{min}^{SG} is set to 3. The meaning of this setting is discussed in Section 6.5.3.

During the simulation, a total of 1,055 situations occur where a late-join client needs the initialization data of a certain object (see Figure 6.4(a)). Since the feedback mechanism described in Section 6.3.1 reduces the number of duplicate late-join requests, only 791 requests are actually transmitted for the third variant with three multicast groups for data distribution⁵.

As depicted in Figure 6.4(b), these requests lead to a total of 80,690 received requests for the first, 68,530 for the second, and 14,270 for the third variant. These numbers include the requests used for the processes of server and client selection. When comparing variants one and two, the total number of received late-join requests is higher for the first variant since the feedback mechanism selects more clients to send a request when employing only one multicast group (not shown in Figure 6.4(a)): As an estimation for N (see Section 6.3.1), the current size of the base group N^{BG} is used for the first variant whereas the size of the client group N^{CG} is used in case of the second variant. Since $N^{BG} > N^{CG}$, the running times of the feedback timers are higher for the first variant, which decreases the probability that a client is selected early and suppresses other requests. In case of the third variant, the total number of received requests is much lower since requests are transmitted to the much smaller server group (at the first attempt).

Consequently, the total number of initializing states sent in response to the received requests is much smaller for the third variant (800, see Figure 6.4(c)) when compared to variants one (2,260) and two (2,063). The dramatic effect when introducing a multicast group for late-join clients becomes visible when analyzing the total number of late-join states received by all sites (see Figure 6.4(d)): While the first variant leads to 201,750 received states, only 22,230 and 7,750 states are received in case of the second and third variant, respectively. Since states might be large and their handling costly in terms of processing power, the number of transmitted and received states has a significant impact on the performance of the late-join algorithm. Summing up, together with the number of received late-join requests, the overall network load due to late-joins is by far the smallest for the variant with three multicast groups.

Figure 6.4(e) shows the distribution of the average initialization delay for a late-join client, which is measured as the time span between discovering that late-join data is needed for an object and receiving that data. The initialization delay is mostly caused by the double selection process via the feedback algorithms described above: First, a late-join client is selected to send a request, then a server is selected to send the initialization data. The time to extract and transmit the data are equal for all variants and therefore not considered. As pointed out earlier, the feedback timers run slightly longer for the first variant than for the variant with two groups. Thus, the resulting average initialization delays are 780 ms for the

⁵This number does not include the requests of a second round, which might be necessary because no server was found at the first attempt.

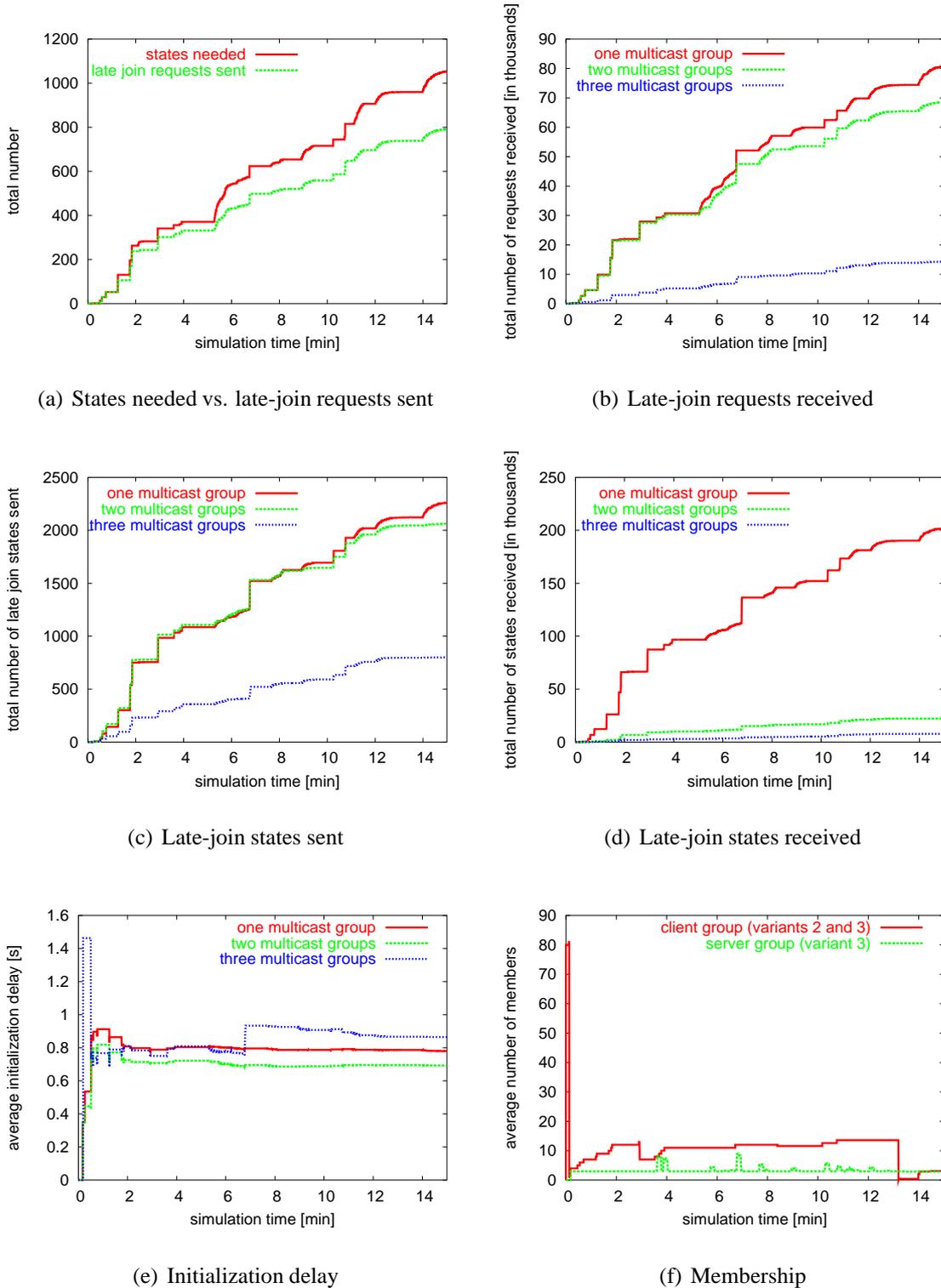


Figure 6.4: Simulation results for the shared whiteboard scenario

first variant compared to 693 ms for the second variant. The third variant causes a higher average delay of 865 ms since it is possible that no appropriate server is found in the server group and an additional request round becomes necessary. After the (initially empty) server group is formed, the average initialization delay of the third variant is comparable to the one of the first variant. Only when a participant joins after 6.5 minutes, appropriate servers cannot be found within the server group at the first attempt since our timeout mechanism led to a small server group. Thus, additional request rounds are necessary to find a server, which increases the average initialization delay, as depicted in Figure 6.4(e). That new sites join the server group due to the second request round can also be seen in Figure 6.4(f).

The composition of the groups for late-join clients and servers is depicted in Figure 6.4(f): Late-join activity is high at the beginning of a session. At first, many participants join in a short period of time, and late-join requests cause some members to join the server group as well. Since in this early phase of the session there are only few objects, and all of them are active, the first wave of clients soon has received the complete application's state and leaves the client group. Later on, sporadic late-joins occur so that the size of the client group increases slowly. Late-joins and switching of the active page (which may trigger sites to join the client group in this scenario) on the one hand and client timeouts as well as clients leaving the session on the other hand balance one another, so that the client group is relatively stable now. The size of the server group fluctuates around the predefined minimum of three members, which indicates that many requests can be answered by the current group members.

The application load induced by the late-join algorithm is mostly caused by four tasks. (1) Process incoming late-join requests (i.e., select a site): As already discussed, the third variant results in the smallest number of received requests and therefore in the smallest application load due to request processing. (2) Extract and send the late-join data for a certain object if selected as the server: Again, the third variant causes the lowest number of object states sent. (3) Discard unneeded packets: As shown above, with the third variant by far the lowest number of states are received in total. (4) Manage additional communication groups: The application load caused by the management of multicast groups can be approximated by the total number of join and leave operations. The base group has to be managed in all variants and is not considered. During the simulation, there are a total of 188 joins and leaves for the client group, and 805 joins and leaves for the server group (one third occurring in the first two minutes of the simulation). The overall application load depends highly on the specific application. But when considering the significant reduction of received late-join traffic, it can be expected that the additional cost for group management is negligible and that the third variant results in the smallest total load.

To conclude, using separate multicast groups for the transmission of late-join requests and the distribution of late-join data results in a significant reduction of the total network load.

Depending on the processing costs for the handling of requests and responses, these variants also have a smaller application load despite the overhead for group management. But at the same time, the initialization delay for late-join clients increases slightly when a separate server group is used.

6.5.1.2 Online Game Scenario

In the second scenario, an online game is simulated with a maximum of 150 participants. Here the underlying network topology results in end-to-end delays between 2 ms and 436 ms with an average of 197 ms. As in the last scenario, T_{max} is set to 500 ms. In our model of an online game, each participant has an individual view of the application's state, with a set of active objects that may be different for each participant. Thus, a site may receive events for passive objects. A late-join client requests all objects that are active for itself by the immediate late-join policy and all other objects by the event-triggered policy.

The activity in an online game is expected to be much higher than in a shared whiteboard scenario, and the time span between two successive actions of an application instance is randomized between 100 ms and 800 ms. The probability to create a new object is set to 0.005, which leads to a total number of 228 objects created during the simulated time of 180 s. The probability to change the state of an object is set to 0.15, resulting in 6,930 events.

One main characteristic of the online game scenario is that the composition of a session is much more dynamic than in the shared whiteboard scenario, with a continuous late-join activity and a large variance in the number of session members. The minimum number of session members is therefore set to 75, while the join probability of 0.005 and the leave probability of 0.0005 remain constant throughout the simulation. This leads to 154 joins (i.e., 79 joins after the simulation started) and 27 leaves during the simulated time.

These settings cause a total of 4,550 situations where a late-join client needs the initialization data for a certain object (see Figure 6.5(a)). Due to the suppression of duplicate requests as described in Section 6.3.1, only 3,400 late-join requests are issued.

As in the first scenario, the average initialization delay for the late-join client is larger for the first variant (906 ms) than for the second (695 ms, see Figure 6.5(e)). For approximately 50 percent of all requests, the third variant does not find an appropriate server in the server group, making a second round necessary. As a consequence, the average initialization delay for the third variant is 994 ms. These initialization delays seem to be rather high, especially for a continuous interactive application, but can be explained by the high end-to-end delays of our simulated network topology, which lengthen the selection process (see above). Moreover, the rate of successful requests in the server group is lower than in the first scenario because each participant is interested in a different set of objects, and because there is a high rate of joining

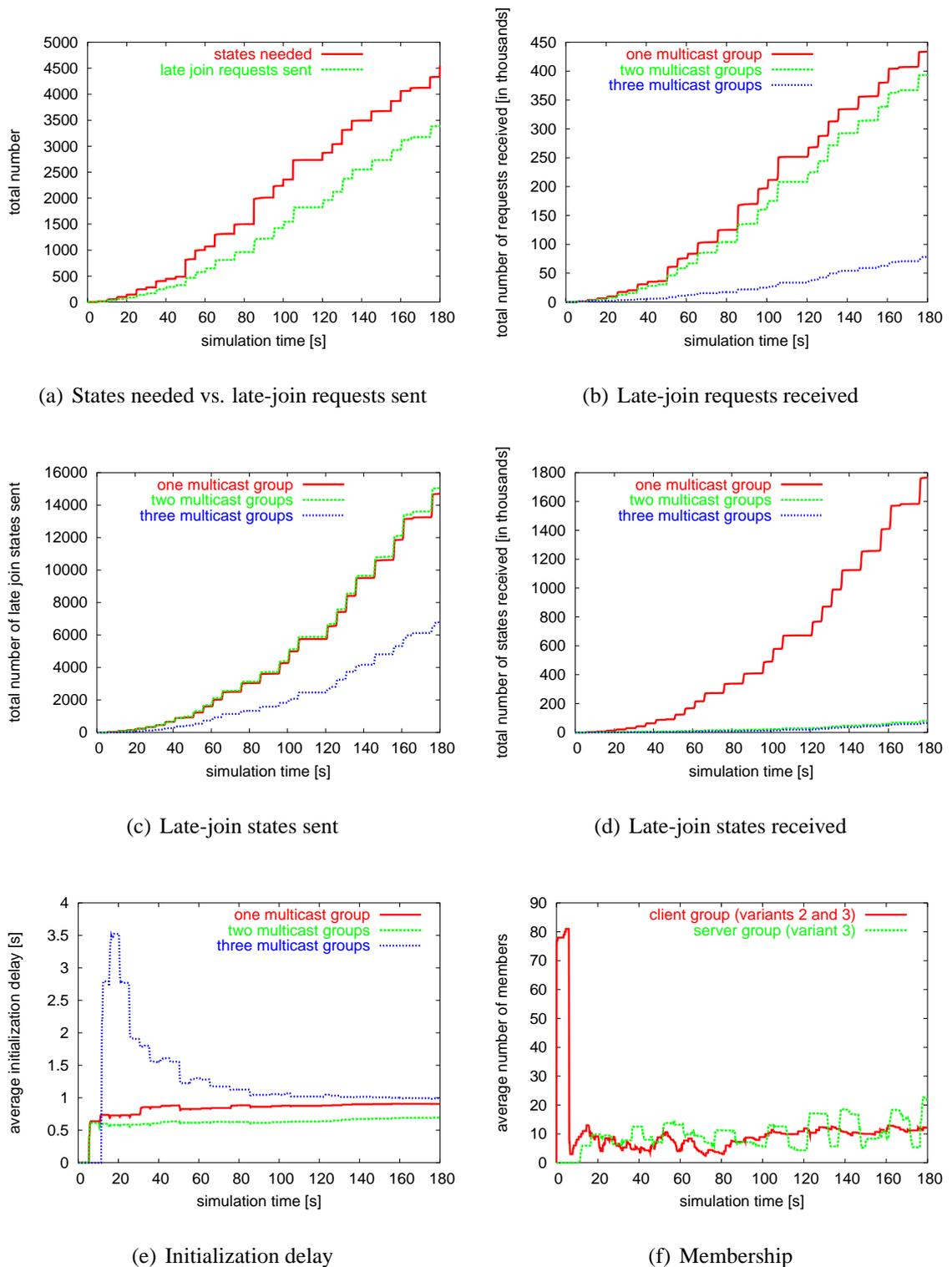


Figure 6.5: Simulation results for the online game scenario

and leaving sites. In a real world setting of a typical online game, we expect much lower end-to-end delays and therefore lower initialization delays. The dynamic group compositions of the server and the client group can also be seen in Figure 6.5(f). In contrast to the whiteboard scenario, the client group is never empty, and the server group is rather instable.

The total network load due to late-join activities is measured by the total numbers of received states and requests. Since the first variant distributes all late-join data via the base group, it produces a high network load with a total of 1.77 million states received by all sites (see Figure 6.5(d)) and 0.43 million requests received (see Figure 6.5(b)). Variants two and three transmit states over a client group; they reduce the number of received states to 79,300 and 64,660 respectively. The server group of the third variant cuts the number of received late-join requests to 78,200. These numbers show that the total network traffic related to the support of late-joining participants can be dramatically reduced by additional multicast groups.

Concerning the application load, with variant one a total number of 14,700 states are transmitted, and variant two leads to 15,050 transmissions whereas only 6,790 states are transmitted with the third variant (see Figure 6.5(c)). The management of the client group has to handle 3,980 join and 3,970 leave operations in versions two and three. The dynamics of the server group of the third variant was evident with 3,509 join and 3,487 leave operations. It can be concluded that depending on the application the third variant is expected to produce the lowest overall load, even though the proportion of the group management costs are higher than in the whiteboard scenario.

To sum up, introducing additional multicast groups for late-join clients and servers saves a significant amount of application and network load. These savings are lower for applications with higher join and leave rates of session members like in the gaming scenario. The higher initialization delay for late-join clients might be unacceptable for some continuous applications but are acceptable for applications with lesser demands. In our experience, users tolerate a higher initialization delay if the initialization takes place when joining a session for the first time (up to 2-3 seconds including the data transmission time). During a session, much lower delays are required in order to prevent consistency-related artifacts (up to 300 ms, see Section 4.4). Variant 2 with a separate client group but without an additional server group could therefore be a good fit for continuous interactive applications.

6.5.2 Late-Join Policies

The flexible late-join policy model allows a client to request only those parts of the shared state that are necessary to immediately participate in an ongoing session. In order to analyze the effects of this policy model, we compare the effects of the event-triggered and the immediate late-join policy in a simulated whiteboard scenario where only the variant with three

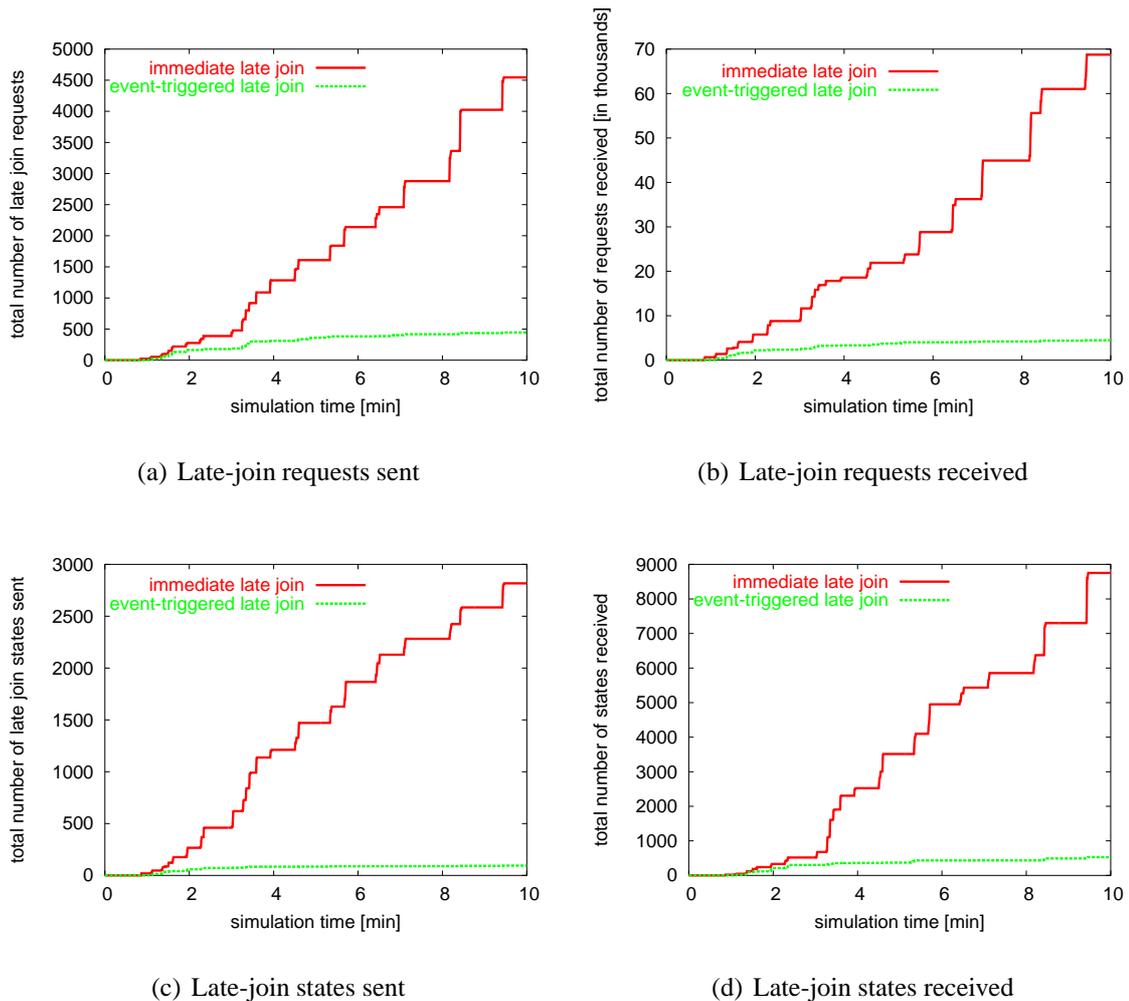


Figure 6.6: Simulation results for different late-join policies

multicast groups for data distribution was considered⁶. The network topology is the same as in Section 6.5.1.1. During the simulated time span of 10 min, 34 pages and 210 graphical objects are created, and 1,002 events are issued. The performance of the event-triggered late-join policy depends highly on the probability that passive objects are reactivated. Here, a relatively high rate is selected with eight occasions on which an older page becomes active again. For continued late-join activity, the whiteboard session starts with 50 members, and 46 participants join during the simulation. Members are not permitted to leave.

In the first scenario, a late-join client imitates the behavior of many existing approaches and requests the state of all objects immediately (see Section 6.2). As depicted in Figure 6.6(a), a total of 4,545 requests are issued (including all cases where a second request round is executed). In the second scenario, only the active objects are requested immediately and all others on demand by the event-triggered policy. This reduces the total number of requests

⁶Our simulation toolkit uses a simple network model that does not take link bandwidths into account. Simulating the network-capacity-oriented late-join policy was therefore not possible.

significantly to 445. Consequently, the network load due to the reception of requests is much higher for the first policy: A total number of 68,760 requests are received by all sites (see Figure 6.6(b)). With the event-triggered policy, only 4,470 requests are received. Even more important with respect to the network load is the dramatic reduction in the number of states sent and received: From 2,817 to 95 states sent (see Figure 6.6(c)), and from 8,753 to 529 states received (see Figure 6.6(d)).

The event-triggered late-join policy also helps to limit the application load caused by managing the groups for late-join clients and servers. When using the immediate late-join policy only, 2,322 join/leave operations for the client group and 2,369 join/leave operations for the server group occur. For the event-triggered late-join policy, these numbers decrease to 142 joins/leaves for the client group and 246 joins/leaves for the server group.

The presented simulation results show that the policy model is capable to significantly reduce both the network traffic and the application load caused by the initialization of late-join clients. Even in scenarios where a higher percentage of the shared state is active or where objects are reactivated more frequently, we expect substantial savings. At the same time, the average initialization delay increases only by 12% for the event-triggered policy in this scenario. Similar results were obtained when simulating the different policies for the online game scenario.

6.5.3 Composition of the Late-Join Server Group

The composition of the late-join server group has a direct influence on the performance of the late-join algorithm that uses three multicast groups for data distribution. One important factor is the minimum size of the server group N_{min}^{SG} (see Section 6.3.3): In case the server group has many members due to a high N_{min}^{SG} , the probability for finding an appropriate late-join server in the first request round is relatively high. This limits both the initialization delay and the number of requests sent. But at the same time, a larger server group possibly increases the network load caused by the number of received requests and the number of sent and received states due to duplicate server selections. To investigate this correlation, the shared whiteboard scenario of Section 6.5.1.1 was simulated for different N_{min}^{SG} . Figure 6.7(c) shows that the average initialization delay indeed decreases with an increasing group size: From 933 ms for $N_{min}^{SG} = 0$ to 865 ms for $N_{min}^{SG} = 3$, and to 731 ms for $N_{min}^{SG} = 10$.

The total number of requests received is by approximately 2,260 requests higher for $N_{min}^{SG} = 0$ (see Figure 6.7(a)) since a second request round is more likely. But as depicted in Figure 6.7(b), a lower value for N_{min}^{SG} also decreases the total number of received states from 8,970 to 7,750. And the number of received states is the main factor determining the total network load of the late-join algorithm.

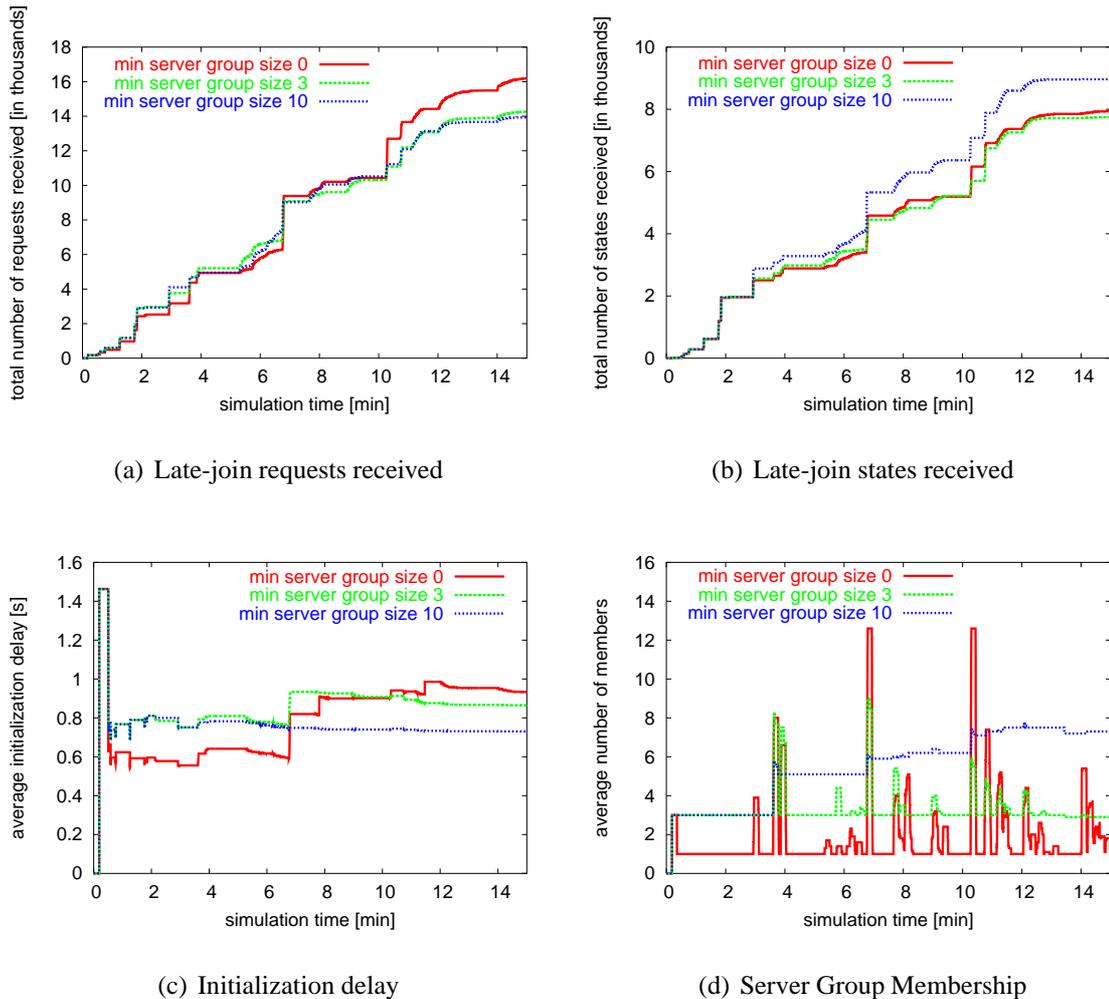


Figure 6.7: Simulation results for different minimum server group sizes

With respect to the application load, a small value for N_{min}^{SG} leads to a rather instable server group with frequent joins and leaves of group members. Figure 6.7(d) demonstrates that the fluctuation is much higher when setting the N_{min}^{SG} to 0 or 3 when compared to $N_{min}^{SG} = 10$.

Summing up, for continuous applications that require a low initialization delay for late-join clients, it is appropriate to define a minimum size between 10 and 20 percent of all participants for the late-join server group. For all other applications, a small minimum size greater than zero is sufficient. In the scenario presented here, setting N_{min}^{SG} to 3 seems to be the best choice.

Another component determining the composition of the server group and therefore influencing the performance of the late-join algorithm is the adaptive server timeout as defined by Equation 6.1. The goal of this timeout is that sites that can serve a great portion of the application's state and that have frequently acted as late-join server remain in the server group longer than others that are less important. In order to evaluate the efficiency of this approach, another simulation study for our standard whiteboard scenario is conducted with $N_{min}^{SG} = 3$.

In this study, the adaptive timeout approach is compared to one that uses a random timeout value within the same range of values. While the initialization delays are almost identical, the network load caused by the number of received requests increases by 500 for the random timeout approach because no appropriate late-join server can be found at the first attempt. This also increases the number of received states significantly by 1,050. These numbers indicate that the adaptive server timeout mechanism is able to improve the overall performance of the late-join algorithm.

6.6 Using the Late-Join Algorithm in Sample Applications

The generic late-join service was implemented as a library on the basis of the RTP/I protocol and can easily be integrated into distributed interactive applications (see Section 7.4.2). The service realizes the flexible policy model for initializing late-join clients by object states. For the exchange of late-join requests and states, either two or three multicast groups can be chosen. In the following, it is discussed how the late-join service can be used by different applications.

The application state of the 3D collaboration tool TeCo3D is modeled as one single object that is requested by the immediate late-join policy. In order to minimize the initialization delay, only one additional communication group for the late-join clients is used. Since TeCo3D uses a strict floor control mechanism to serialize operations [156], the dynamic server selection scheme is not used but the current floor holder is picked as late-join server [261].

The mlb structures its application state hierarchically where a page holds a set of graphical objects, and pages are arranged into chapters and documents. For the user, exactly one page is visible at a certain point in time; the hierarchy of pages is also shown in a separate window (see Figure 3.1). A late-join client requests the active objects of the current page by the immediate late-join policy, and all other objects when they become the target of an event. The periodic session messages of RTP/I carry enough meta-data for the client to infer the page hierarchy (see Section 7.3.2). The page hierarchy allows late-joining users to change the active page before its state is actually available. The late-join communication is realized over three multicast groups since the initialization delay is not crucial here. Late-join servers are selected dynamically, and we propose to set the minimum size of the server group to one for small sessions (up to ten session members) and to three for bigger sessions. Late-join data is provided in the form of an older state and an operation sequence.

The collaborative tools integrated into the user interface of the mlb (telepointer, voting, feedback, hand raising, chat, and application launch) are all similar in that their state is rather small, and that all objects are active. For telepointers, no late-join mechanism is employed

since each telepointer action starts with a state transmission so that late-join clients are updated implicitly (see Sections 3.5.2 and 4.2). All other tools use the service and employ the immediate late-join policy only. In cases where the application state is composed of only one object (i.e., hand raising and chat), the request process can be started as soon as a participant enters a session. All tools use three multicast groups for the transmission of late-join data.

6.7 Support for Late-Joiners in Instant Collaboration

The previous sections focused on synchronous collaboration where all participants are connected at the same time via a network, and operations are distributed immediately among all sites. Under regular conditions without network or software failures, late-joins occur only when a user enters a session for the first time, and the corresponding application instance is in its initial state.

In contrast, distributed interactive applications that also support *asynchronous* collaboration allow a user to change the application's state independent from others, even when some participants are disconnected. In this case, operations need to be exchanged when the concerned sites are online again at a later point in time. Instant Collaboration is an example for such an application, which was introduced in Section 2.4.3. In Instant Collaboration, objects of the shared state are persistent even when all users are offline [79].

In a scenario where sessions have no predefined end, potential late-join situations occur when a user switches from asynchronous to synchronous collaboration: The user might have changed the application's state while working offline, or he might have missed operations from other participants. Depending on the application and the user behavior, the local state copies held by the participating sites may always diverge to a certain degree, and late-joins may happen frequently. The biggest challenge for a late-join algorithm in such an environment is that information might be available only at a small percentage of all session members, in the extreme only at the originator of that information, and during a limited time span.

In the following, a robust late-join algorithm is proposed for distributed interactive applications allowing both synchronous and asynchronous collaboration. Again, we concentrate on applications where access to the shared state is unrestricted and users are allowed to issue operations any time. This novel algorithm is integrated into the Instant Collaboration prototype and tested in simulated long-term sessions.

One peculiarity of Instant Collaboration is that its communication model is based on point-to-point connections, not multicast. Thus, we decided to distribute late-join information via unicast as well. We are aware that this introduces a performance penalty and does not scale well with the number of application instances (see Section 6.1.4). However, a recent user

study by Muller et al. showed that most sessions with Instant Collaboration have only few members, with typical averages between three and twelve [177], so that the overhead is acceptable. A promising alternative would be to distribute data via application-level multicast as described in Chapter 8.

6.7.1 Late-Join Algorithm

For distributed interactive applications that allow synchronous as well as asynchronous collaboration, the late-join algorithm plays a central role and needs to be designed carefully. We will now examine the design options presented in Sections 6.1 and 6.3 specifically for such applications and discuss the decisions made for Instant Collaboration.

The first design option concerns the extent of late-join data and determines at what time a late-join client should be provided with what parts of the shared state, and in which form that data is to be delivered. As in the synchronous domain, active objects should be delivered with the highest priority so that a late-join client is able to participate in an ongoing session as quickly as possible. If participants switch frequently between synchronous and asynchronous collaboration, and the number of participants working synchronously is rather small, it is advisable to employ the immediate late-join policy with a lower priority for all other objects, too. The goal of this approach, which we use for Instant Collaboration [83], is to minimize the extent to which the local state copies differ on average. The performance penalty in terms of network and application loads when using the immediate late-join policy instead of the event-triggered policy in the Instant Collaboration scenario is significantly lower than for the multicast settings described in Section 6.5.2 because the unicast distribution model here prevents that a single transmission of late-join data can be utilized by several late-join clients. Thus, clients need to be initialized individually anyway. Unicast also prevents that data is received by sites that are not interested, and eliminates the need for feedback raise mechanisms to select application instances (possibly lowering the initialization delay).

The most efficient way to provide late-join data to others are object states. However, in the dynamic environment encountered here, it is likely that the local state of a site that is selected as a late-join server is not up to date. Thus, several state transmissions might be necessary until a late-join client reaches a consistent state (see Section 6.4). Better suited is therefore a replay of the operation history, which enables a client to repair possible inconsistencies locally with an appropriate consistency control mechanism. Under the condition that all missing operations will be delivered eventually to the late-join client, this approach is very robust. Since in many late-join situations the client already holds a large part of the operation history, only the missing operations need to be exchanged, which reduces the initialization delay as well as the network and application load. For Instant Collaboration, the missing

parts of the operation history can be identified by comparing the state vectors of the late-join client and the server. The client then integrates these operations into its operation history by applying the timewarp algorithm (see Section 4.7.2).

Late-join clients that join a session for the first time have an empty initial state. Instead of replaying the complete operation history to those clients, the initialization can be started with an old state S_T and a subsequent replay of later operations $\{O_t | t \geq T\}$ (see Section 6.4.3). The starting state S_T should be correct and complete (see Section 4.1) so that the combination of state and operation replay is sufficient for the client to reach a correct state. Whether a state S_T from the operation history of the server j qualifies as a starting point can be determined by comparing its state vector SV_{S_T} with the vectors SV_i of all other sites i : If $SV_{S_T}[k] \leq SV_i[k] \quad \forall k, i \neq j$, S_T contains the effects of all operations scheduled before T and is a valid starting point. In practice, this implies that $T_C - T$ might be large.

When a user of Instant Collaboration joins a session (i.e., accesses one of the activities he is participating in), the late-join client contacts all sites with whom it collaborates one by one and sends them the current state vectors of the shared objects belonging to that session. Comparing those state vectors with its own, an application instance can decide which operations from its history need to be sent to the late-join client and vice versa (if any). Thus, the late-join client can obtain missed operations from any instance participating in a shared object, not only from the original source of an operation. This is especially useful when that source is offline at the time the late-join client connects. Additionally, it increases the robustness of the system and distributes the burden of initializing clients over all sites.

Instead of contacting only a single site, the late-join client contacts all sites with whom it collaborates in order to quickly discover operations missing in the client's history. This lowers the time span until a consistent state is established and increases the robustness of the system. To further increase the probability that all missed operations are received, information about the current state of objects is also exchanged whenever a user joins an activity. Both techniques also increase the application and network loads, but do not pose a severe scalability problem since the number of session members per activity is typically small (see above).

In Instant Collaboration, a whole set of sessions (activities) is managed, and each session has a distinct set of members (see Section 2.4.3). Thus, there exist situations where a late-join client is not able to trigger the initialization process: Consider the case that a new activity is created (or the client is invited to join an existing activity), and either the user creating that activity or the user being invited is offline. So when reconnecting, the late-join client does not know about the new activity and can therefore not ask for the transmission of missed operations. Even worse, it can occur that the client does not know the other members of this activity if they did not collaborate before. Thus, in case a user misses the creation of an

activity or the invitation to an existing activity, the responsibility for the initialization must be with the originator of that action (i.e., the late-join server). Should client and server connect for the transmission of any other data, the missed operations can be delivered. The late-join server also tries to contact the client periodically (depending on the current application and network load).

The probability for quickly updating a late-join client can be increased by placing the responsibility not solely on the original source but on all members of an activity. The more members an activity has, the higher is the likelihood that a site possessing the concerned data is working synchronously with the late-join client. For this purpose, the user who created an activity or invited new members informs all available sites that the relevant operations could not be delivered to one or more sites. All application instances are then assigned the task to transmit the operations to the late-join client either by making regular contact or by probing periodically. They stop as soon as they receive a state vector from the client that indicates that it possesses the required information. In case the late-join client receives the same operations more than once, it simply ignores them.

6.7.2 Simulation Results

For the Instant Collaboration prototype, we implemented the distributed caching algorithm for the exchange of late-join data as described above [83]. In order to evaluate its performance, different scenarios for the synchronous and asynchronous collaboration of three users were simulated. In each scenario, the activities of a typical work week with five days are reproduced randomly. During one simulated workday, the following actions take place: A total of three activities are created, each user views the data of an activity fifteen times (i.e., opens and closes the view window of an activity fifteen times), and each user modifies the state of an activity ten times. The simulated scenarios differ with respect to the time span that each user is spending online or offline. For easier handling, each workday is simulated in 60 seconds. Because of the limited number of operations in the scenarios, we insert a state snapshot every five operations into the history (see Section 4.5).

While a user is working offline, operations originating from or targeting that application instance cannot be delivered immediately. Instead, the algorithms described above transmit the missed operations when the user reconnects. Figure 6.8(a) shows the times users are collaborating synchronously in a scenario where the three users spend 80% of the simulation online. The number of operations that actually need to be cached because the receivers are unreachable is depicted in Figure 6.8(b). The late-join algorithm manages to update sites as soon as they are online again. The curves include those operations that are cached by all instances that were online at the time they were issued. On average, site 1 stores 5.3

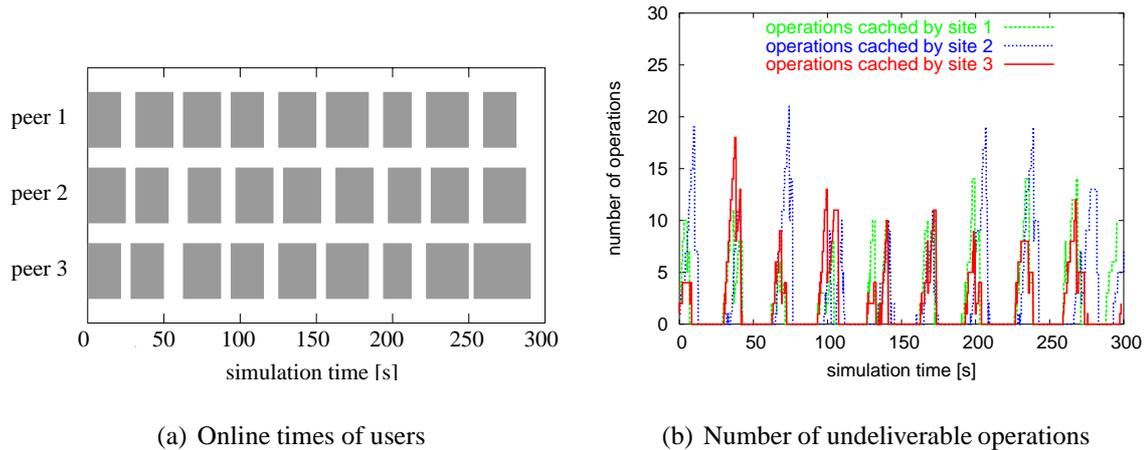


Figure 6.8: Simulation results for interleaved synchronous and asynchronous collaboration

operations (site 2: 7.2, site 3: 5.3) for the other instances, and it takes about 3.9 s (site 2: 4.4 s, site 3: 4.0 s) between generating and delivering a certain operation. These numbers increase considerably for a scenario where users are online for 50% of the simulation time: Site 1 caches on average 25.4 operations (site 2: 15.3, site 3: 22.5) and missed operations are transmitted 22.7 s (site 2: 13.5 s, site 3: 20.5 s) after they were issued. In a last scenario, users are online for only 20% of the simulation time, and the first users meets the others only rarely. Consequently, site 1 stores on average 107.4 operations for 60.6 s (site 2: 80.4 operations for 22.8 s, site 3: 66.4 for 27.6 s).

These numbers show that local state copies might diverge to a considerable degree when users are working offline for a long period of time. The algorithm for the distributed caching of missed operations most likely will alleviate this problem if the number of members of a shared object increases. However, its performance also depends on the work patterns of the activity members. In the worst case, local states of a shared object might diverge for a very long time, e.g., if two users are alternating between being offline and online: Each user works on a state that is incomplete and then, after eventually merging the operation history, they might see an unexpected result. This reflects a major problem of this approach: While the state is now correct for both users, the algorithm cannot guess what their actual intention was when they were independently modifying the shared objects. The algorithm only makes sure that both can see the same result. However, for the user it is difficult to understand how this state came to be. We therefore believe that it is crucial to provide conflict visualization mechanisms such as the ones described in Chapter 5 to assist the user in this task. For example, the application could animate parts of the history to visualize the sequence of operations that led to the current state.

There are also two technical solutions that might alleviate the aforementioned problem: Adding additional background servers, which cache operations and update late-join clients, would help to decrease the time between the resynchronization of states. When sites are connected to the system, they can send operations to the caching server if the collaborating users are currently offline. When the other sites connect, they first contact the caching server to collect missing operations. The drawback of this approach is that it requires additional infrastructure that needs to be maintained. Another technical approach could be to put more semantics into the consistency algorithm itself: The ordering of operations by state vectors is based on sequence numbers, i.e., when people work offline for long periods of time, the system only looks at the sequence numbers to restore a correct state. But the sequence numbers do not reflect when operations actually took place. If two users work offline at different times, the order of operations could be prioritized by the recency of the state change, which would probably help users to better understand the resulting state after merging the operation histories. This could be achieved either by using globally synchronized clocks as a means for sorting (which again requires infrastructure) or, more elegantly, by using a modified state vector scheme that incorporates local time into the numbering.

6.8 Conclusions

Distributed interactive applications often support dynamic groups where users may join and leave at any time. But a participant joining an ongoing session has missed all data previously exchanged by the other session members. Thus, the application needs to employ a late-join algorithm that initializes the late-joining site with the current state. In this chapter, various possibilities for the design of late-join algorithms were discussed. While most existing distributed interactive applications integrate some kind of late-join mechanism, it was shown that a carelessly designed algorithm may cause high initialization delays for the late-join client, leads to high application and network loads, and might also raise inconsistencies.

Thus, a novel late-join algorithm was proposed, which is scalable and robust due to its replicated approach and group communication. The algorithm was realized as a reusable service by employing a generic model for distributed interactive applications in combination with flexible late-join policies. However, the basic concepts presented here can also be used as a basis for application-specific solutions to the late-join problem. Moreover, it was thoroughly analyzed how consistency of the shared application state can be reached in late-join situations.

By simulating different scenarios, it was shown that a carefully designed late-join algorithm significantly reduces the application and network load. Furthermore, the simulation provided insights on how to best distribute late-join data to the clients. It was demonstrated that appli-

cations with a stable group membership such as shared whiteboards will benefit considerably from two additional multicast groups: One for the transmission of state information to late-join clients and one for the transmission of state requests to the potential servers. In contrast, very dynamic and time critical applications (i.e., networked computer games) are likely to best use only one additional multicast group for the transmission of late-join data.

The late-join algorithm was integrated into existing applications. For TeCo3D, only one additional multicast group is used in order to minimize the initialization delay. For the mlb, the variant with a client and a server group is chosen. Both applications use the policy model to request late-join data for active objects at once and data for all other objects when receiving events.

For distributed interactive applications facilitating synchronous and asynchronous collaboration on the shared state, late-join situations occur frequently, and the local copies of the shared state held by the participating sites might diverge to a considerable degree. A novel late-join algorithm was devised for such applications and was integrated into Instant Collaboration. Simulation studies indicate that the distributed caching of temporarily undeliverable operations is well-suited to handle late-join situations. Besides achieving consistency in such a dynamic environment, one important issue for a late-join algorithm is to give sufficient feedback to the user when updating states.

Chapter 7

RTP/I - An Application-Level Protocol for Distributed Interactive Applications

Distributed interactive applications have a common data model: They have a shared state, which might be structured into a hierarchy of objects and can be changed by operations or by the passage of time. This model allows us to discuss and design algorithms for problems common to many distributed interactive applications independent from a specific application, e.g., consistency control (see Chapter 4) and support for late-joining session members (see Chapter 6). These algorithms can also be implemented in an application-independent way if the information required for their realization is separated from the other application data and exposed such that it can be accessed from outside the application. For instance, a late-join algorithm would only need information such as the type of an operation (e.g., state or event) and its target object, but not the actual meaning of the application's state (e.g., shared whiteboard pages) or certain operations (e.g., change color of a circle).

A standardized network protocol is well-suited for exposing such universal information [38, 219, 225]. We therefore propose the application-level protocol RTP/I for distributed interactive applications [158, 159]. RTP/I is based on the common data model, and its main goal is to convey information about the application, the exchanged operations, and the ongoing session. This information can be used to implement common algorithms for distributed interactive applications in the form of generic services. The same late-join service could then be employed by a networked computer game and a shared whiteboard. Thus, applications can reuse these services instead of implementing the same functionality over and over again, as it is currently the case for distributed interactive applications.

Another design goal for RTP/I is to provide basic protocol functionality to the application, such as fragmenting data packets and informing the application about the participants present in a session.

The design of RTP/I was strongly influenced by a variety of applications [80, 142, 154] and generic services [115, 260, 261] developed at the University of Mannheim. In particular, the mlb with its complex application state and demands on consistency control, support for late-joining session members, and recording functionality had a great impact on RTP/I.

In the next section, existing approaches for the standardized transmission of application data are examined. In Section 7.2, design options for a generic protocol for distributed interactive applications are analyzed, and it is discussed which functionality should be included in such a protocol. The RTP/I protocol is then presented in Section 7.3. In Section 7.4, it is demonstrated how generic services can be realized with the information provided by RTP/I. The integration of RTP/I into an application is discussed in Section 7.5. Section 7.6 concludes this chapter.

7.1 Related Work

The basis for RTP/I is the Real-Time Transport Protocol (RTP) [219]. RTP is a generic protocol framework for distributed *non-interactive* applications, i.e., applications that do not allow user interactions such as tools for transmitting videos in real-time [164]. RTP consists of two parts: A data protocol and a control protocol (RTCP). The data protocol is used for the distribution of application packets (e.g., a video stream), which are framed with the standardized RTP header. This packet header holds information that is common to all streaming applications such as an identifier for the sender of a data packet, a sequence number to order the packets of a source and to identify lost packets, and a timestamp denoting when the encoded data was sampled.

The control protocol RTCP maintains important meta-data using a soft state approach (see Section 4.2). Besides announcing information about the participants (e.g., names and email addresses), its main task is to monitor the quality of data reception at the individual receivers with different parameters (e.g., loss rate), and to report the current measurements back to the sender. According to this data, the application might adapt the encoding of the media stream. The bandwidth available for these RTCP reports is limited to a certain percentage of all session traffic so that RTP scales well with respect to the number of session members. At the same time, this might result in a high lag until important information is delivered.

The information encoded in the RTP header and the meta-data provided by RTCP is sufficient to realize generic services such as a recorder for video conferences [119].

Many video-conferencing programs [164, 212] and applications for streaming audio and video [202, 210] base their media communication on RTP. In order to adapt RTP to different applications, profiles and payload types can be defined. A *profile* captures common aspects of a whole class of applications. For instance, the profile for audio and video conferences gives general recommendations for the encoding of audio and video streams (e.g., packet send rate) and defines values for some RTP header fields [218]. The encoding of a certain media type is specified in a *payload type* definition, e.g., for the transmission of MPEG-4 audio and video streams [137].

Despite the fact that RTP is very flexible, it is not well-suited for distributed *interactive* applications, since the data models for interactive and non-interactive applications differ substantially. A major difference is that a non-interactive application continuously substitutes the current state with a new one since the application is not able to calculate subsequent states by itself as time goes by, as it is the case in the interactive domain. In addition, the data of a non-interactive application is not structured into independent objects, and there is only a single data source per media stream. Other arguments against RTP include that not many interactive applications would need the RTCP quality measurements and that RTP allows collisions of participant identifiers (even though there is a recovery scheme). Perkins and Crowcroft discuss the usage of RTP with distributed interactive applications more thoroughly in [195].

The Reliable Multicast Framing Protocol (RMFP) [38] and the Reliable Multicast Framework (RMF) [40] are both protocol frameworks for reliable multicast, and they can be adapted to the specific reliability requirements of an application. Moreover, RMFP and RMF contain some protocol functionality that is in general useful for distributed interactive applications, e.g., RMFP supports structured shared states [38]. However, they are not well-suited as a generic protocol since they do not address some important aspects such as the classification of operations, information for consistency control mechanisms, or timing of operations.

The Simple Object Access Protocol (SOAP) [225] is a generic and flexible protocol for the exchange of structured messages (i.e., packets with application data) in various distributed environments ranging from electronic commerce to multi-user conferencing. SOAP defines the format of messages [226], the encoding of data types [227], message exchange patterns such as request-response and Remote Procedure Call (RPC) [227, 228], as well as processing rules for the message handling at the application instances [226]. The message format is based on the Extensible Markup Language (XML) [64], and SOAP concentrates on the syntax of messages and leaves the particular semantics to the applications. Even though SOAP is not specifically designed for a certain underlying protocol, it is mostly run over HTTP/TCP or SMTP/TCP [225].

SOAP messages contain a header and a body for the application payload. This splitting allows the development of generic services that are located at so-called “intermediate nodes” [225] on the path of a message from the sender to the receiver. Intermediate nodes process the header only, execute their service (e.g., encrypt or log the message), and then forward the (possibly modified) message to the next intermediate node or to the final receiver.

SOAP includes several interesting design principles and could be used to define a protocol framework for distributed interactive applications. However, it has some major drawbacks: First, XML is text-based, and messages need to be parsed before they can be decoded. This results in a large overhead with respect to packet size and processing costs when compared to a binary protocol. Additional overhead is introduced by information about XML syntax specifications that is included in the packets.

To conclude, existing protocol frameworks are not well-suited for distributed interactive applications. At the same time, they provide an insight into design principles and help to identify important features that a generic protocol for distributed interactive applications should have. In the next section, these design considerations are examined in more detail.

7.2 Design Considerations for a Protocol Framework

A generic protocol for distributed interactive applications should capture the common aspects of these applications and reveal sufficient information for the development of generic services. At the same time, it should follow a minimalistic approach and provide only such protocol functionality that is useful to all applications. In case one or more applications require additional features, the protocol needs to be adaptable and extendable, e.g., by means of profiles and payload type definitions as in RTP [219] or by extension headers as in SOAP [226]. This flexibility requires that the protocol is designed as an open framework. In the following, the core functionality of a protocol framework for distributed interactive applications is discussed, and it is investigated which information should be exposed by this framework.

7.2.1 Architecture and ADU-Related Information

A protocol framework for distributed interactive applications should operate on the basis of ADUs as they are the smallest piece of information that can be interpreted by the application [33] (see Section 2.3.3).

The generic protocol should be independent of the underlying transport and network protocols that an application might employ. In particular, different types of **reliability** should be supported: The first possibility is that the application chooses a strict layering approach

where the reliability mechanism is provided transparently by a separate protocol, e.g., TCP. The main advantage of this approach is a structured design of the application's communication system and the possibility to reuse existing reliability protocols. However, stand-alone protocols for reliable data transport also establish a source order on all packets originating from the same sender. This ordering is too strict for applications with a state that is structured into independent objects and might cause unnecessary delays when an operation is missing from a sequence targeting different objects. Such artificial delays could even result in inconsistencies for continuous applications.

Alternatively, the reliability mechanism can be combined with other application-level functionality following the concepts of ILP and ALF (see Section 2.3.3 and [33]). In this case, the reliability mechanism can use the same header fields that are also needed for other functions (e.g., sequence numbers, identifiers for objects and participants, etc.). Besides this efficiency gain, the main advantage of the ILP approach is that application-level knowledge can be used to adapt the reliability mechanism to the nature of the application data that is to be transmitted as described by Mauve and Hilt in [157]. For instance, operations should be transferred quickly and with a high level of reliability by protecting their transmission with a FEC scheme. In contrast, application-data units (ADUs) that are not state-changing could be distributed unreliably. Moreover, application-level knowledge can be applied to the ordering of ADUs.

Another important aspect for distributed interactive applications is **consistency control** (see Chapter 4). The protocol framework should allow an application to employ any pessimistic or optimistic consistency control mechanism but also provide information that is needed by those mechanisms, e.g., information about the order of operations. In case the application employs a pessimistic consistency control algorithm (e.g., locking), sequence numbers can be used to establish a (partial) order of operations per sender and per object. If the application uses an optimistic algorithm, concurrent operations originating from different sources are possible, and a single sequence number is not sufficient. Instead, operations could be ordered by state vectors. For continuous applications, the execution time of operations has to be considered additionally by a consistency control algorithm. Thus, ADUs should carry timing information in the form of physical clock readings. These timestamps can also be used for inter-stream synchronization when the distributed interactive application is combined with other (interactive or non-interactive) applications, e.g., in a video conference scenario.

Many issues that need to be addressed for distributed interactive applications require to request (parts of) the shared state in a structured way. For instance, the consistency control mechanism proposed in Chapter 4 uses **state requests** as a fallback solution in case the local operation history is not sufficient to repair a short-term inconsistency. Moreover, the late-join algorithm proposed in Chapter 6 initializes a late-joining application instance by request-

ing appropriate state information. Thus, the protocol framework should offer a standardized ADU for requesting the state of certain objects. This allows generic services to request state information or to answer state requests without needing to interpret the actual application data. Since requests differ with respect to their urgency (e.g., a state request for resynchronization is more important than the request of an archiving service), it should be possible to assign appropriate priorities to state requests.

The native data units of a protocol framework for distributed interactive applications are ADUs. Depending on the application and the ADU type, these might be rather large. For instance, the state of an image that is located on a shared whiteboard page might have several hundred kbytes. But the links of a network are only able to handle packets that do not exceed a certain size, the Maximum Transmission Unit (MTU). In case a packet exceeds the MTU of a link on the way from the sender to the receiver, it is split into smaller units. Such **fragmentation** of a large ADU on the network layer has the negative effect that the loss of a single fragment means that the whole ADU is discarded by the transport layer [135]. Thus, the protocol framework should fragment large ADUs on the application-layer so that the employed reliability mechanism can manage the loss of missing ADU fragments. The size of the smallest MTU supported over an end-to-end connection can be determined by ICMP packets [172].

In order to relate ADUs unambiguously to their sender and to their target object, unique **identifiers** are required (see Section 3.3.1). RTP uses random numbers to identify participants [219], which is not a suitable solution for distributed interactive applications since such identifiers may collide, and changing the identifier in case of a collision as in RTP might raise severe difficulties, e.g., for the consistency control mechanism. A straight-forward approach for generating unique identifiers is to use the network layer or physical layer address of a site as a basis. However, this results in large identifiers and packet headers. Alternatively, identifiers can be assigned by a well-known server that coordinates the identifier namespace and prevents collisions. The server can be responsible for multiple sessions where each has several independent namespaces, e.g., one for participants and one for objects. The scalability of this approach can be improved when application instances request identifiers in ranges, and when there exist several servers where each is responsible for a different area of the hierarchically structured identifier namespace. A prototype for such a service was developed in the mlb project.

Summing up, a generic protocol for distributed interactive applications should frame ADUs with a header that contains sufficient information for the core functionality of the protocol and for the development of generic services: A classifier for the ADU type, a sequence number for ordering and fragmenting ADUs, a counter for ordering fragments, identifiers for the ADU's source and the target object, a timestamp for ordering and timing ADUs,

a state vector for consistency control, a priority for ranking ADUs, and reliability-related information if necessary.

7.2.2 Session-Related Information

Aside from this ADU-specific data, the protocol framework should also provide information that is related to the session (so-called “meta-data” [158]). There are two categories of session-related information that are useful to many distributed interactive applications and generic services and that should be included in the protocol framework: Information about participants of a session and information about the application’s shared state.

In order to facilitate collaboration, distributed interactive applications seek to establish a certain degree of awareness among the users (see Sections 2.3.1 and 3.5). For instance, users should be aware of other session members, their status, and their actions. Therefore, the protocol framework should realize a light-weight session control such as the one proposed for RTP [219] to announce joining or leaving session members and to provide the names, email addresses and locations of users. More complex information such as statistics about a participant’s activities or different access rights of participants should not be included in the basic protocol framework but be managed by separate generic services, e.g., by a floor control service [156].

The other important category of meta-data concerns the shared state of the application itself: For many generic services it is vital to know which objects are present in a session without depending on the reception of ADUs targeting these objects. For instance, the generic late-join service needs to learn about all objects present when joining an ongoing session. Besides the pure existence of an object, further information might be necessary that describes the object’s properties. For example, this information can be used by the application to determine an appropriate late-join policy (see Section 6.3.2). More precisely, it should be announced whether a certain object is active or not, i.e., whether it is currently displayed by one or more application instances (see Section 2.2). Moreover, some description concerning the type of the object and its role in the application’s state is needed, e.g., whether the object is a shared whiteboard page or a graphical object, and on which page this graphical object is located. This *application-level name* depends on the application and might not be mandatory in all cases, e.g., when similar information can also be derived from the object’s identifier.

Both types of session-related information comply with our data model for distributed interactive applications that was introduced in Section 2.2: They have a replicated state (e.g., the list of objects that are currently active), and this state might change in the course of the session (e.g., a formerly passive object is activated). Thus, mechanisms for the propagation of states

and state changes for meta-data are necessary. Moreover, a consistency control algorithm is required that synchronizes all local copies of the meta-data.

As discussed in Section 4.2, consistency control mechanisms either seek to establish consistency with a *hard state* or a *soft state* approach. In the first case, consistency of the shared state is enforced, which includes that state changes are announced immediately and reliably. The main benefits of the hard state approach are that the propagation delay for operations is low, and that the sender of a state or a state update knows when all participants received this operation. However, it is also very complex and requires that explicit measures be taken to prevent and repair inconsistencies, e.g., in case a late-join occurs. Alternatively, consistency can be reached with a soft state approach where periodic state (re-)transmissions are used to distribute state changes and to substitute possibly inconsistent data. This approach is very simple and robust and requires no explicit error recovery mechanisms. For instance, late-joining participants are updated by means of the next state transmission. However, the propagation delay of operations might be rather large depending on the packet loss rate, the probability for concurrent actions, and the retransmission frequency. It also consumes more bandwidth than the hard state approach, even though the shared state for meta-data is fairly limited in size.

Considering the low complexity of the soft state approach and the small and rather stable shared state, we propose to employ the soft state approach for session-related information. It can be improved in two ways: First, the propagation delay for state changes can be decreased by shortening the report interval that lies between two successive state transmissions for the updated state. Second, the bandwidth used for the distribution of meta-data can be limited to a certain percentage of the total bandwidth available as proposed in [219] for RTP.

7.3 The RTP/I Protocol

The Real-Time Protocol for distributed Interactive applications (RTP/I) is based on the design considerations of the last section [111, 156, 158, 159]. Following the design principle of RTP, it consists of two parts: A data protocol for the transmission of ADUs and a control protocol (RTCP/I) for the exchange of session-related information. Both parts are run over separate channels. The application is free to choose the actual transport protocol (e.g., UDP over IP multicast).

RTP/I was developed at the University of Mannheim, based on a joint project of Mauve [156] and Hilt [111]. A first version was published by Mauve et al. in [158]. In this section, the latest version of RTP/I is described, which was thoroughly modified to address the challenges

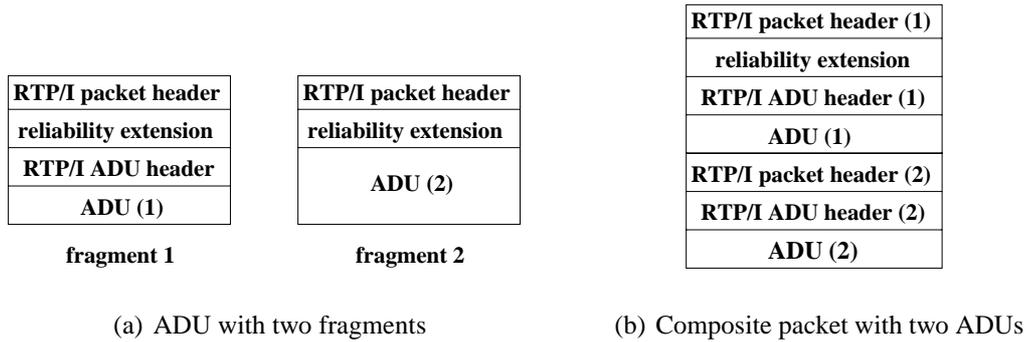


Figure 7.1: Structure of RTP/I ADUs

discovered when designing the mlb, the generic late-join service, and the consistency control service.

7.3.1 The RTP/I Data Transfer Protocol

The task of the RTP/I data transfer protocol is to frame application data units and to fragment large ADUs if necessary. As given by the data model for distributed interactive applications (see Section 2.2), five different ADU types are distinguished: States, delta states, events, cues, and state requests. Each type contains specific information that is to be exposed by RTP/I. Because ADUs might be fragmented, this information is arranged into two independent headers: An RTP/I packet header for each fragment containing all data necessary for the fragmentation process, and an RTP/I ADU header carrying all information that is analyzed once an ADU is complete. This two-level structure was devised in the course of this thesis. Figure 7.1(a) depicts an ADU that is transported in two fragments. In case an application-level reliability mechanism is employed that needs additional information, a reliability extension header follows each RTP/I packet header (since reliability has to be established on the basis of fragments). The design of an appropriate reliability mechanism was discussed by Mauve and Hilt in [157], and its realization is an issue for future work.

An application might also generate subsequent ADUs that are very small, e.g., when a user changes the position of a graphical object in a series of mouse events. In this case, it is more efficient to aggregate some ADUs into an RTP/I composite packet, which is transported en bloc to the receivers. As shown in Figure 7.1(b), there is only one reliability extension header per composite packet.

7.3.1.1 Packet Header

The RTP/I packet header is identical for all ADU types and contains the information depicted in Figure 7.2. The first two bits define the version V of the RTP/I protocol. The type field

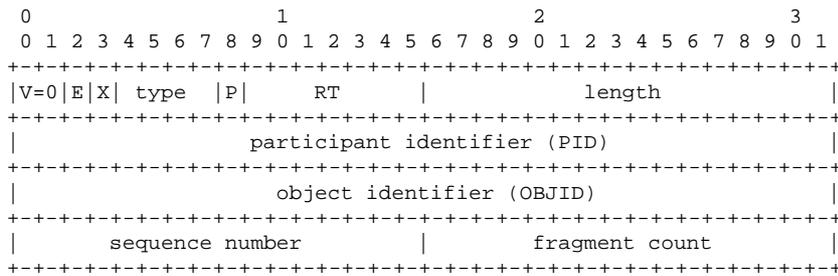


Figure 7.2: RTP/I packet header

classifies the ADU type. The used reliability mechanism is identified by the reliability type RT field, and the X bit indicates whether the packet header is followed by an reliability extension header. The length field gives the size of an ADU in bytes (excluding the 20 byte packet header) and allows to split composite packets. In case the packets need to be aligned to certain block sizes (e.g., for an encryption algorithm), an appropriate amount of padding bytes can be attached at the end. Then, the padding bit P has to be set, and the last padding byte added holds the total number of padding bytes.

The ADUs have separate sequence number namespaces per sender, per target object and per ADU type so that these four header fields are necessary to associate a fragment to its respective ADU. Individual fragments are numbered by the fragment count (starting with 0), and the last fragment is recognized by the end bit E.

The participant identifier PID belongs to the ADU's sender, and the object identifier OBJID determines the target object¹. Both identifiers have to be unique and persistent in a session and might be generated by an identifier service as described in Section 7.2.1.

7.3.1.2 States

A state ADU contains all data that is necessary for the application to create a certain object. It is used when a new object is introduced, when an inconsistency needs to be repaired, or when a late-joining participant needs to be initialized. Some applications might also announce state changes by transmitting the updated state of an object if states are small or a soft state approach is chosen (see Sections 2.2 and 4.2).

The common header for RTP/I state ADUs is depicted in Figure 7.3. The payload type field PT defines how the attached payload is encoded by the application. In case a profile needs to include additional information, it can use the optional profile information field PI. The priority PRI indicates how receivers have to react: A state with a priority of 3 must be adopted by all receivers (i.e., an existing local state is discarded) and is used for newly created objects and for resynchronizations. States with a priority of 0 can be ignored. For example,

¹Objects are denoted as *sub-components* in RTP/I [158].

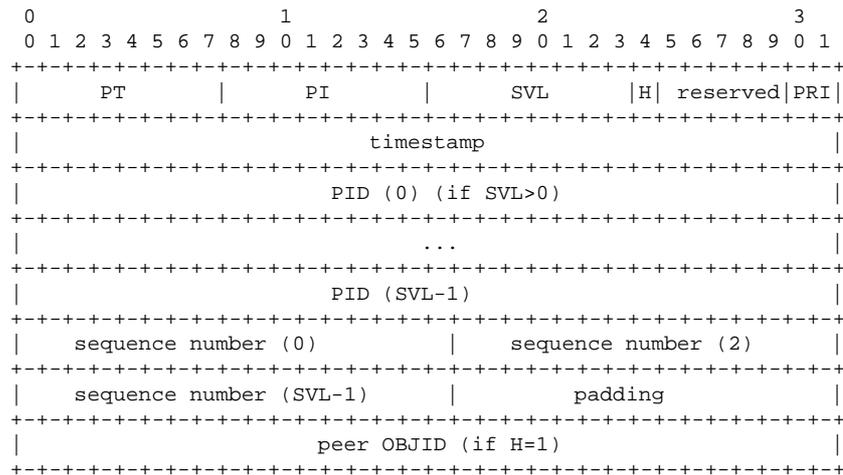


Figure 7.3: RTP/I state ADU header

late-join data has a priority of 0 since it is only relevant to late-join clients. The timestamp denotes the physical time at which the state was extracted by the sender.

When the application employs an optimistic consistency control mechanism and allows concurrent operations, an application instance cannot guarantee that the extracted state of an object is consistent (also see Sections 4.6 and 6.4). For instance, there might be one or more operations that should have been executed before this operation according to their timestamp but that were not incorporated into the sender's operation history yet. In order to discover such conditions, the state header was extended to include a state vector as defined in Section 4.1. State vectors must not be encoded in the ADU's payload since sequence numbers are set by RTP/I and need to be generally accessible by generic services (see Section 7.4). The length of the state vector that is included in a state header is given by the SVL field.

Similar to the mlb, many distributed interactive applications have a complex shared state that is structured into a hierarchy of objects. These relationships among objects need to be established by the application when executing a state ADU. Thus, state ADUs have to carry information about object relationships, which can be represented by the identifiers of the concerned objects. Since RTP/I identifiers must not be encoded in the payload of ADUs, the state header offers the possibility to include the identifier of a single peer object (indicated by the hierarchy bit H). For instance, an object hierarchy can be encoded by specifying each object's parent.

7.3.1.3 Delta States

Delta states encode all changes to the state of an object that have been issued since a certain state snapshot S_T was taken. The state of an object that is reflected by a delta state can be derived by applying the delta state to S_T . A delta state header therefore carries a PID and a sequence number to identify S_T and is otherwise identical to a state header.

7.3.1.4 Events

Events change the current state of the application and therefore have an ADU header that is identical to the state header except that there is no need for priorities (see Figure 7.3). The timestamp is interpreted differently and denotes the event's execution time. In case the application employs the local lag approach as described in Section 4.4, the timestamp includes the local lag value. The peer object identifier is needed only for events changing an object's relationship, e.g., when the pages of a shared whiteboard are resorted.

The state vector might be used to order operations and to check whether the event is causally ready for execution (see Section 4.1), e.g., whether the target object of a move event already exists. Alternatively, events can be ordered by their timestamp, their participant identifier and sequence number. And the application might check semantically whether an event is causally ready, e.g., whether the target object already exists before executing an event.

7.3.1.5 Cues

Cues are user actions that have no effect (or only a temporary effect) on the application's shared state but that should be propagated nevertheless in order to increase the user's awareness about each other's actions and to increase the application's presentation quality (see Section 2.2). The two most important cue categories are informal messages and intermediate state changes. An example for the first category are temporary awareness hints that notify a user about the actions of remote participants, e.g., "W. Effelsberg is idling for ten minutes". Such messages do not change the application's state and are propagated as cues. In the second category, a series of actions changes the application's state continuously until a final state is reached. For instance, moving a graphical object within a 3D world generates a sequence of intermediate object positions until the user places the object at its final position (see Section 3.3.3). In order to keep the application's shared state consistent, only this last position would need to be propagated as event. But transmitting the intermediate positions as cues additionally visualizes the object's path to remote users.

The header of a cue ADU is identical to the event ADU header. The main motivation to distinguish between events and cues is that these can be handled differently. For instance, cues do not need to be transmitted reliably while a lost event would endanger the application's consistency. Cues that change the application's state must therefore be followed by an event defining the final state change.

7.3.1.6 State Requests

State request ADUs offer a standardized way to request the state of a certain object. The RTP/I state request ADU is depicted in Figure 7.4, i.e., there is no application-specific pay-

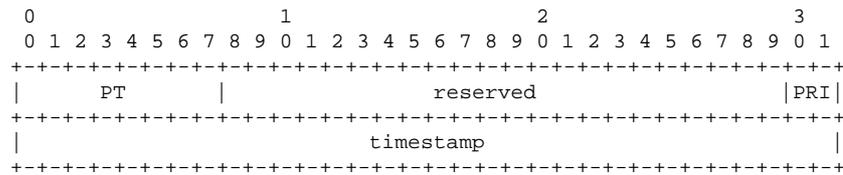


Figure 7.4: RTP/I state request ADU

load. The priority indicates how urgent the request is: A priority of 3 is used for requests that need to be answered immediately (e.g., to repair an inconsistency), 2 is for requests where short delays can be tolerated (e.g., in a late-join situation), 1 is for requests where an answer is not vital (e.g., for passive objects), and requests with a priority of 0 can be ignored when the network or application load does not allow an answer. Which application instance should answer to a state request is not defined by RTP/I but has to be determined by the application, e.g., by a feedback mechanism as described in Section 6.3.1.

7.3.2 The RTP/I Control Protocol (RTCP/I)

The RTP/I Control protocol (RTCP/I) manages the exchange of meta-data about the participants and the application's state in a session. As proposed in Section 7.2.2, RTCP/I follows the soft state approach where meta-data is updated periodically. Similar to RTCP, the bandwidth consumed by RTCP/I is limited to approximately 5% of the bandwidth available for RTP/I by applying the following algorithm: Each application instance periodically reports its own meta-data, and the time span between two reports, the so-called *report interval*, is inversely proportional to the number of session members and the average report size, i.e., the larger the session and the meta-data, the less frequent a report is sent. Ideally, the report interval calculated at each participant is the same. The resulting network load is spread evenly by randomizing the individual send times as described by Floyd and Jacobson in [66].

In order to limit the propagation delay for important state changes (e.g., when a new member joins the session), Raman and McCanne propose to shorten the report interval for such updates so that they are transmitted with a higher priority [203]. At the same time, the next report for older information is delayed, so that the overall bandwidth consumed remains constant. In case some meta-data is not reported for several report intervals (e.g., 5 intervals), it is first marked as timed out and later deleted.

7.3.2.1 Participant Information

Each participant reports certain information about himself by sending a so-called RTCP/I source description ADU. As shown in Figure 7.5, the header of the source description ADU has a fixed size of 8 bytes and contains the protocol version, the type of the RTCP/I ADU, the length of the ADU in bytes (including any padding bytes, see Section 7.3.1.1), and the

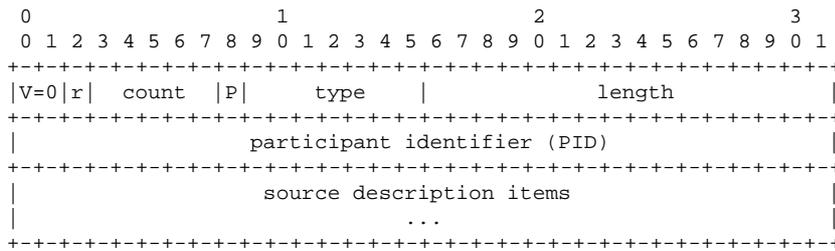


Figure 7.5: RTCP/I source description ADU

sender’s participant identifier. The header is followed by a list of source description items where each item holds a certain piece of information such as the user’s name, phone number, and email address [159]. The number of items is given by the count field, and each item starts with one byte denoting its size.

The so-called “canonical name” (cname) of a participant is the most important source description item: The cname can be used to identify the same participant in case he attends several RTP/I (and RTP) sessions at the same time. While the participant identifier is unique in a session, the cname should be unique across sessions. Typically, a cname has the syntax “user@host” where the host could be the end-system’s IP address or name (e.g., jvogel@pi4.informatik.uni-mannheim.de).

A joining participant immediately sends a source description ADU containing the cname in order to propagate this state change quickly. Members that leave a session send an explicit “bye” ADU. In case the “bye” ADU is lost, the participant information times out after several report intervals.

7.3.2.2 Application State Information

The second category of session-related information that is exchanged periodically among all session members describes the shared state of a distributed interactive application: The RTCP/I object report ADU contains a list of all objects that are known to the reporting application instance. This information allows to explore the application’s state without relying on the individual object states, which are usually significantly larger than the meta-data. The header of an object report ADU is identical to the source description header (see Figure 7.6). For each object, the sender indicates whether it is active locally (A), and whether the peer object identifier is included (H). The optional application-level name together with the peer object identifier can be used by the application to specify the role of the object in the shared state. For instance, the mlb uses the object type (page, rectangle, ...) as the application-level name.

In order to save bandwidth, a participant does not report objects that were announced previously by other session members and that have the same properties of activeness, application-

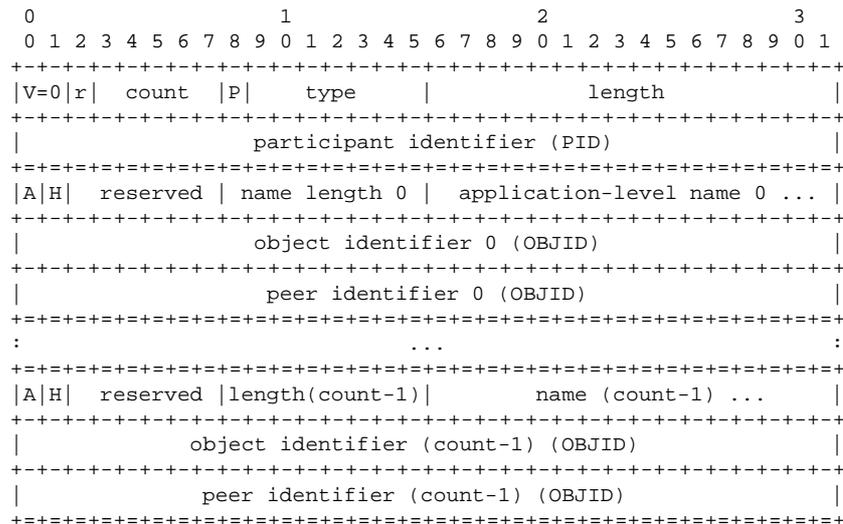


Figure 7.6: RTCP/I object report ADU

level name, and peer object as its local object copy. Thus, if an object is active for at least one participant, it is also reported as active in each interval. In the ideal case, each object will be reported only once per interval. As proposed above, changes to an object's properties will be reported with a higher priority.

7.4 Generic Services

The information exposed by the RTP/I protocol framework can be used to implement algorithms for distributed interactive applications as generic services. Thus, complex functionality needs to be developed only once and can be integrated with little effort into any application that bases its communication system on RTP/I.

A generic service can be designed at different levels of abstraction: If it uses only information of the core RTP/I protocol, it is valid for all applications. Alternatively, generic services can also be devised specifically for a profile or for certain payload types. In this thesis, generic services for consistency control and for the support of late-joining participants were devised. Moreover, a service for the recording of sessions was developed by Hilt [115]. These services are independent of a specific profile or payload. Other examples that can be realized for RTP/I are floor and session control, encryption, and data mining in archived sessions. In Section 7.5, it is discussed how the communication system of a distributed interactive application can be based on RTP/I.

Like all distributed interactive applications, a generic service has a shared state that might change in the course of a session. As a consequence, generic services need to exchange state updates and follow either a soft state or a hard state approach for achieving consistency. For instance, a generic floor control service needs to announce access rights, a recording service

```
Functions of class RTPI.ConsistencyControl:
    void InsertRtpiAdu(RTPIADU adu)
    void SetInsertStateFrequency(Time interval)
    void SetStorageTime(Time time)

Functions of interface RTPI.ConsistencyControlFeedback:
    void ExecuteRtpiAdus(RTPIADU_List adus)
    RtpiAdu GetObjectState(ObjId object)
    boolean Conflict(RTPIADU adu1, RTPIADU adu2)
    boolean Overwrite(RTPIADU adu1, RTPIADU adu2)
```

Figure 7.7: Consistency control library API

needs to signal the playback speed [111], and a late-join service needs to select application instances to join the late-join server group (see Section 6.3.3). We propose to use a separate communication channel for generic services, which is more flexible than integrating those messages into either RTP/I or RTCP/I since it does not impose any specific reliability or consistency control mechanism on the services [261]. A more detailed discussion on this topic is conducted by Hilt [111] and Walling [266].

7.4.1 Generic Consistency Control Service

In Chapter 4, a consistency control mechanism for distributed interactive applications was presented, which is based on local lag, timewarp, and state request. The information provided by RTP/I allows to realize this mechanism as a generic service [260], which is optimized for discrete applications. It is implemented in C++ and was successfully integrated into the mlb.

The generic consistency control service manages one operation history per object and orders operations according to their state vector or timestamp. As depicted in Figure 7.7, the application hands over all local and remote ADUs to the class `RTPI.ConsistencyControl` without executing them. Before an ADU is inserted into the history, the service checks whether the ADU is causally ready or not. This check can be performed on the basis of an ADU's state vector or, if events do not carry state vectors, with application-level semantics. If an ADU is not causally ready, it is delayed until the ADUs it depends on have arrived. Otherwise, it is integrated into the operation history.

When the execution time of an ADU lies in the future due to local lag, the service can make use of the time gained and reorder the operation history if necessary. Once their execution time is reached, ADUs are delivered to the application via the function `ExecuteRtpiAdus` of the `RTPI.ConsistencyControlFeedback` interface. In case an ADU is received out of order or too late and the checks for conflicting and overwriting ADUs described in

Section 4.5.3 lead to the result that a timewarp is necessary, the service calculates the appropriate operation sequence that will repair the inconsistency, and delivers this sequence to the application. Cues do not trigger a timewarp but are discarded in such a case. In order to limit the size of the operation sequences that restore the consistent state, the service periodically asks the application for state snapshots where the insertion frequency is determined by the application. The application also restricts the size of the operation histories by defining a time limit for storing ADUs. In case a timewarp cannot be executed due to this limitation, the generic service initiates the state request mechanism described in Section 4.6 where the state request has the highest priority of 3. The state received in return also has a priority of 3 and is handed over to the application via `ExecuteRtpiAdus`.

7.4.2 Generic Late-Join Service

The late-join algorithm presented in Section 6.3 is another example for an RTP/I-based generic service. The late-join service is implemented as a library in both Java [261] and C++ [262], and it is integrated into the applications TeCo3D [156] and mlb.

In addition to the application group with three channels for RTP/I, RTCP/I, and the generic services, the late-join service introduces either one extra group for the late-join clients or two extra groups for clients and servers (each with one channel for RTP/I and RTCP/I). All RTP/I and RTCP/I ADUs that are received by a participant via the application group are analyzed by the late-join service. For instance, a received event might start the state request mechanism for an object if the event-triggered late-join policy was selected for this object. Received RTCP/I object reports are used to explore the application's shared state, and the application might select an appropriate late-join policy for an object by its application-level name and peer object identifier. In case an application instance has to be selected to enter the late-join server group, both selection request and acknowledgment are sent via the generic services channel. The number of session members that are needed to calculate the feedback timers are estimated from the RTCP/I participant reports.

The late-join data to initialize the clients is transmitted via the RTP/I channel of the client group (depending on the consistency control mechanism, in the form of a single state ADU or, like for the mlb, as a sequence of state and event ADUs). For the second variant of the late-join algorithm (see Section 6.3.3), state requests are sent directly through the RTP/I channel of the application group, and for the third variant requests are distributed via the RTP/I channel of the late-join server group. Late-join state requests have a priority of 2, and states sent in response with the lowest priority (0) since they are only relevant for late-join clients and can be ignored by others.

Functions of class `RTPI_LateJoin`:

```
void ReceiveRtpiAdu(ADU adu)
void ReceiveRtcpiAdu(ADU adu)
void SetPolicy(ObjId object, LJ_Policy policy)
void setResponsibility(ObjId object, bool responsible)
```

Functions of interface `RTPI_LateJoinFeedback`:

```
RTPI_ADU_List GetLateJoinData(ObjId object)
LJ_Policy GetPolicy(ObjId object, ObjId peer, string name)
void LateJoinFailed(ObjId object)
void ReceiveLateJoinData(ObjId object, RTPI_ADU_List adus)
```

Figure 7.8: Late-join library API

Figure 7.8 depicts the interfaces between application and late-join service. The application provides all incoming traffic of the application group via the functions `ReceiveRtpiAdu` and `ReceiveRtcpiAdu` of the class `RTPI_LateJoin` to the late-join service. This class also offers the possibility to set the late-join policy for an object and to select the application-controlled group membership for an object (see Section 6.3.3). The application has to implement the `RTPI_LateJoinFeedback` interface. In case an application instance is selected as the server for a certain object, the late-join service retrieves the appropriate initialization information from the application via the function `GetLateJoinData`. The application is also informed when a new object is discovered, a late-join request failed after several attempts, and when late-join data is received.

In Section 6.7, a late-join algorithm for distributed interactive applications was presented that facilitates asynchronous collaboration. It was implemented for the Instant Collaboration system, but it could just as well be implemented as generic service for RTP/I: The operation history and the distributed caching of operations that could not be delivered to their destinations can be managed on the basis of the object-specific state vectors that are included in the state and event headers. In order to identify and request operations that are missing in its local history, an application instance needs to notify other session members about its local state. For this purpose, participants have to announce their local state vectors. These announcements could either be reported via the generic services channel, be integrated into the RTCP/I participant reports, or be included in RTP/I state request ADUs. The last two possibilities could be realized in an RTP/I profile for asynchronous applications.

7.4.3 Generic Recording Service

A third generic service on the basis of the RTP/I protocol framework was developed by Hilt for the recording and playback of RTP/I sessions [111, 115, 116]. The basic idea is that the

recorder attends an RTP/I session and stores the operation history created from the received ADUs together with the session-related information gathered from RTCP/I. During replay, the recorder then generates an appropriate operation sequence from the stored history by adjusting the timestamps to the current physical time. The recorder also exchanges the object and participant identifiers from the original ADUs so that they will not collide with identifiers present in the session. Such collisions would happen in case the recorded stream were played back with the original identifiers to the same session it was captured from, or in case the same sequence is played back several times to a session. The application instances receiving the played back operation sequence handle these ADUs just like regular data. Like the other generic services, the recorder is able to operate on RTP/I information only and does not need to interpret the ADUs' payloads [115].

The major challenge when designing such a recording service is random access during playback [111], i.e., when a start time for the playback is requested that lies after the session start time. Before an application instance is able to decode a playback sequence, it needs to be initialized with appropriate states for all objects that are present in the sequence: In general, the recorder cannot assume that the objects are in such a state that the operation sequence can be directly applied to them or that they even exist already. Thus, the playback sequence has to include an initializing state for all objects that are active or become active during the playback. The initializing state of an object can be retrieved from the recorded operation history and preferably has a timestamp close to the requested playback time. It is followed by a complete sequence of delta states and events. By adding states into the operation history, fine-grained random access can be accomplished so that the part of the playback sequence that lies before the actual playback time is small. Thus, the recorder periodically issues state requests with a priority of 0.

Figure 7.9 gives an example for such a random access [111]. The topmost sequence shows parts of the original operation history. The play time of the random access is denoted as T_p . At that time, the two objects O_1 and O_3 are active. Thus, the recorder has to generate an initialization sequence for these two objects before the actual playback starts. For discrete applications, only the order of operations needs to be considered and not their timing. The initialization sequences for O_1 and O_3 can therefore be replayed in a fast forward mode (see Figure 7.9 (i)). Afterwards, the recorder plays back the remaining parts of the operation history in real-time. This approach is not possible for continuous applications since here the execution time of operations in relation to the timeline has to be considered. As depicted in Figure 7.9 (ii), the recorder has to start the real-time playback with the earliest state of the active objects (here with S_1).

The generic recording service was designed and implemented by Hilt for the Interactive Media on Demand (IMoD) system [112, 116]. With IMoD, several RTP/I and RTP sessions can

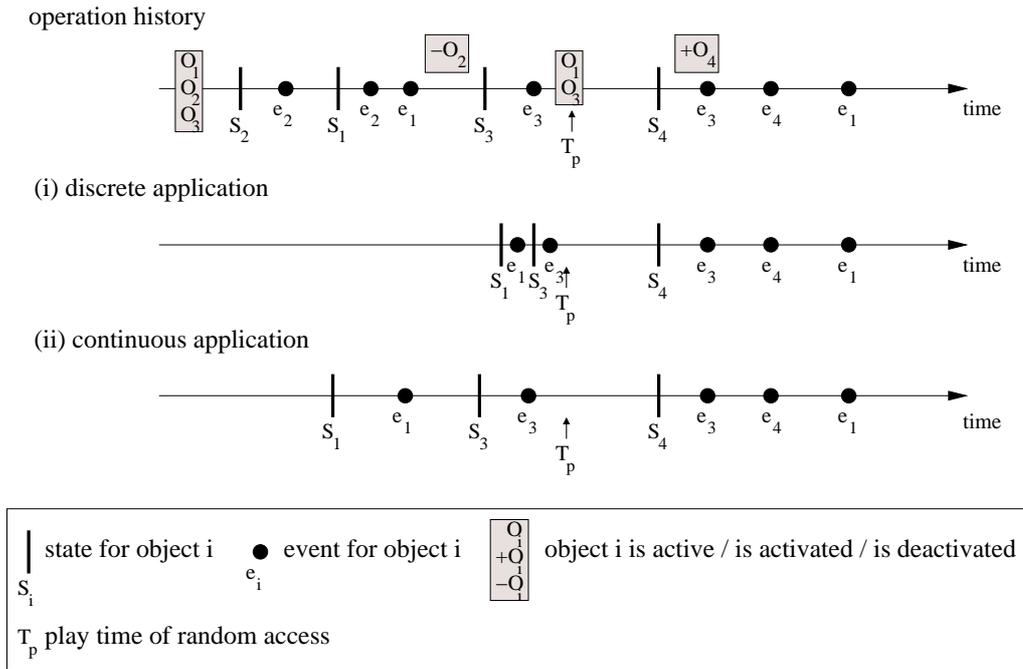


Figure 7.9: Random access for discrete and continuous applications

be recorded (played back) in parallel. The IMoD system was successfully tested with the mlb, TeCo3D, and the Spaceshooter game. At the University of Mannheim, it is used to record lectures and to produce Computer Based Training (CBT) units for e-learning [25, 117, 217]. The CBT includes the original mlb presentation slides with annotations as well as recorded audio and video streams.

7.5 Transport of Application Data with RTP/I

After discussing generic services, now the RTP/I C++ library itself is presented. It was developed in the scope of this thesis in cooperation with the IMoD project [111]. Furthermore, the payload type definition for shared whiteboards is presented. Aside from the mlb, three other distributed interactive applications use RTP/I: TeCo3D [154], the Spaceshooter game [161], and the Java Remote Control tool for sharing Java animations [142]. These three applications are based on the Java library of RTP/I [156], which is fully interoperable with our C++ library.

7.5.1 The RTP/I Protocol Library

The main functions of the RTP/I C++ library are given in Figure 7.10. In order to be independent of the other protocols that the application might employ, incoming and outgoing data are forwarded explicitly to the library via the class `RTPI_Rtpi`. In case the appli-

Functions of class `RTPI_Rtpi`:

```
RTPI_Buffer_List CreateRtpiPackets(RTPI_ADU_List adus)
RTPI_ADU_List HandleReceivedRtpiData(RTPI_Buffer buffer)
void HandleReceivedRtcpData(RTPI_Buffer buffer)
RTPI_Participant_List GetParticipantList()
RTPI_Object_List GetObjectList()
void AddObject(RTPI_Object object)
void DeleteObject(ObjId object)
void SetObjectActiveness(boolean active)
```

Functions of interface `RTPI_Feedback`:

```
void SendRtcpData(RTPI_Buffer buffer)
void ParticipantChanged(RTPI_Participant)
void ObjectChanged(RTPI_Object)
```

Figure 7.10: RTP/I library API

cation wants to send data, it provides the function `CreateRtpiPackets` with a list of ADUs after defining application-level header fields such as the object identifiers. The library fills in the other header fields, fragments or compounds the ADUs and returns a list of ready-to-send bit strings. Data received via the RTP/I channel is rebuilt into ADUs by the `HandleReceivedRtpiData` function. Incoming RTCP/I traffic is analyzed by the library, which notifies the application via the `RTPI_Feedback` interface in case either the participant- or the session-related meta-data has changed. This information can also be accessed anytime by the application via the class `RTPI_Rtpi`, and can also be changed (e.g., when an object is created or deleted).

7.5.2 Payload Type Definition for Shared Whiteboards

The RTP/I payload type definition for shared whiteboards was developed for the `mlb` but is also valid for other whiteboards. A detailed specification can be found in [257].

The state of a shared whiteboard is usually structured into a hierarchy of objects, i.e., a shared whiteboard document consists of chapters containing pages, and pages containing graphical objects. In order to allow a fine-grained handling, the objects of the application state are modeled as independent RTP/I objects. The tree hierarchy of objects is represented by referencing each object's parent container via the peer object field of RTP/I ADUs. Moreover, the ordering of objects within their container has to be considered since chapters and pages are listed in a certain order, and graphical objects have a display order (see Section 3.3). This object order is encoded in the payload of RTP/I ADUs.

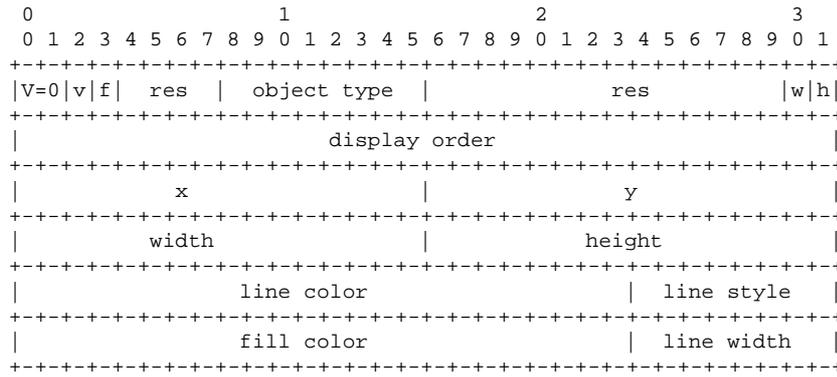


Figure 7.11: Rectangle state ADU

New objects are created by transmitting an RTP/I state ADU with priority 3 to all session members. Figure 7.11 depicts the payload for the state of a rectangle with object type 4. The first two bits denote the version of the payload type definition. The visibility bit *v* indicates whether the object is currently visible. For instance, mlb presentation animations are controlled via the visibility bit. The rectangle is defined by the (x,y)-coordinates of its upper left corner and its width and height in pixels. The *w* and *h* bits are set in case width or height are negative. Other attributes define the line color, the fill color (if the rectangle is filled as indicated by the *f* bit), the line style and the line width. The payload also includes the display order of objects.

Aside from rectangles, the following objects are defined [257]: session, chapter, page, and group as containers, and oval, line, polyline, polygon, text, and image as graphical objects.

Single state changes are transmitted as RTP/I events, and a series of connected actions is sent as a sequence of cues with one closing event, e.g., when a graphical object is moved to a new position. In order to limit the application and network load for such sequences, the user can define the frequency of cue transmissions. In our experience, a cue rate where every third mouse movement causes a transmission results in a very smooth illustration, and transmitting every tenth value is still sufficient. Figure 7.12 depicts the payload for a move operation where (x,y) define the new coordinates of the moved object. Aside from move operations, the following state-changing actions are defined in [257] (depending on the object type): delete, change visibility, change size, change line width, change line style, change line color, change fill color, move point, add point, close polygon, change type, change font, insert characters, delete characters, change name, set active, raise, lower, and change parent.

The mlb uses the participant information provided by RTCP/I to display a list of session members and to indicate the originator of an operation (see Section 3.5). State-related information is used to define the set of active objects, i.e., the page that is currently displayed and all graphical objects placed on that page. The application-level name is set to the object’s type. For chapters and pages, the application-level name also holds their name and order

0										1										2										3																			
V=0										res										object type										operation type										res									
										x																				y																			

Figure 7.12: Move event or cue ADU

number. This data allows late-join clients to display the complete table of contents so that a late-joining user can select a page that was not recovered yet.

As discussed in Section 3.3.4, the mlb transmits RTP/I ADUs via the reliable multicast protocol SMP, i.e., the reliability mechanism is not integrated into RTP/I as proposed in Section 7.2, which is an issue for future work.

In addition to the specification for shared whiteboards, payload types are defined for telepointers [258], for hand-raising tools [256], for voting and feedback tools [255], for chat tools [254], and for application launch tools [253]. In the case of the mlb, all these tools are integrated into one common user interface, but their shared state is managed independently via separate RTP/I sessions.

7.6 Conclusions

In this chapter, the application-level protocol framework RTP/I was presented that is based on the data model for distributed interactive applications. RTP/I exposes the common aspects of these applications and therefore allows the development of generic services. Such services can then be used easily by any application that bases its communication on RTP/I.

RTP/I consists of two parts. The data transfer protocol frames application data into states, delta states, events, cues, and state requests, and provides access to important information such as sender, target object, and state vector. Large ADUs are fragmented if necessary. The second part is the RTP/I control protocol, RTCP/I, which maintains meta-data about the participants of a session and the shared state of the application itself.

RTP/I is an open framework, which can be adapted to the specific needs of an application. It does not force the application to use a certain transport protocol, reliability mechanism or consistency control algorithm but supports the realization of such mechanisms. If needed, additional functionality might be added to the RTP/I framework, either by means of an extension header or by a profile definition.

Moreover, three generic services for RTP/I were presented, a consistency control service, a late-join service, and a recording service. The first two services were developed in the con-

text of this thesis and were successfully integrated into the mlb. The recording service was developed in a parallel PhD project. Finally, the RTP/I C++ library and the payload type definition for shared whiteboards were discussed. Even though the payload type definition was developed for the mlb, it represents a common basis on which different shared whiteboard tools could implement their interoperation.

Chapter 8

Application-Level Multicast for Distributed Interactive Applications

Distributed interactive applications have a replicated architecture, and all messages need to be transported from their originating site to all other application instances by some form of group communication. With respect to this message exchange, applications often have specific demands. For instance, many synchronization algorithms require data to be transmitted reliably so that some mechanism to repair packet loss is needed. Another factor are bandwidth requirements: For instance, the data stream emitted in an mlb session is typically about 1 kbyte/s per site. Finally, the propagation delay among the sender and the receivers is especially important for distributed interactive applications. For continuous applications, an operation O_i is only valid at a certain point in time so that O_i should be delivered by then. And a low delay means that it is less likely that concurrent operations occur that could lead to a short-term inconsistency (see Section 4.4), and that users can collaborate in a natural way. Distributed interactive applications are *delay-sensitive*.

In the Internet, two techniques exist for group communication: *IP multicast* and *Application-Level Multicast* (ALM). The properties of both alternatives are discussed in Section 8.1, and it is shown that IP multicast suffers from various difficulties, which prevents it from being widely deployed. ALM is a promising alternative. In Section 8.2, the requirements for an ALM routing algorithm are analyzed, and fundamental issues of multicast routing are discussed in Section 8.3. Existing ALM approaches are presented in Section 8.4. Following, a novel ALM routing algorithm is proposed, which optimizes the propagation delays on the basis of application-level priorities and network characteristics, which qualifies this approach especially for delay-sensitive applications (see Section 8.5). Then, the performance of this routing algorithm is demonstrated in simulation studies in Section 8.6. An operational routing protocol is introduced in Section 8.7. Section 8.8 concludes the chapter.

8.1 Group Communication

Let us consider the sample session shown in Figure 8.1(a) with four application instances on four end-systems i , which are connected via a network of three routers r_i . Note that Figure 8.1(a) depicts only those nodes that are relevant in our sample and that they represent only a small part of the entire network. The first possibility to realize group communication in this setting is to use multiple unicast connections: As shown in Figure 8.1(b), the sender 1 is connected directly to each receiver and has to transmit three copies of a certain piece of information. While being simple, this causes a high network load due to the duplicate packets, and it also burdens the sender with the maintenance of multiple network connections. Depending on the group size, the resources available might not be sufficient, i.e., when participants have only a limited capacity for outgoing and incoming network traffic. Thus, group communication on the basis of point-to-point connections among all senders and receivers is not a viable solution. Alternatives are IP multicast and application-level multicast.

8.1.1 IP Multicast

IP multicast provides efficient group communication in the Internet. First, a short introduction to the basic architecture of IP multicast is given. Subsequently, the current situation of IP multicast and the difficulties of the architecture are discussed.

8.1.1.1 The Architecture of IP Multicast

IP multicast offers group communication functionality at the IP layer. A certain communication group, or *multicast group*, is identified by a class D IP address. Each participant of such a group maintains only one network connection to the group and does not need to know about the other members. Moreover, sites may join or leave a multicast group at any time. An end-system i communicates over the Internet Group Management Protocol (IGMP) [20] with the multicast edge router of i 's LAN about the membership of i in a certain multicast group.

All messages sent to the IP group address are distributed within the network via a *multicast tree* as illustrated in Figure 8.1(c): A packet is duplicated at those routers r_i that represent the last possible branchings on the paths from the sender to the receivers, e.g., r_1 and r_2 in Figure 8.1(c). In the ideal case, packets traverse the same physical link only once. Like all IP traffic, delivery of multicast packets is unreliable and best-effort.

Each IP multicast router needs to determine all outgoing links for an incoming multicast packet. In contrast to unicast IP, this decision cannot be made on the basis of the IP address alone since an IP multicast address is not tied to a certain set of receivers. Thus, routers that

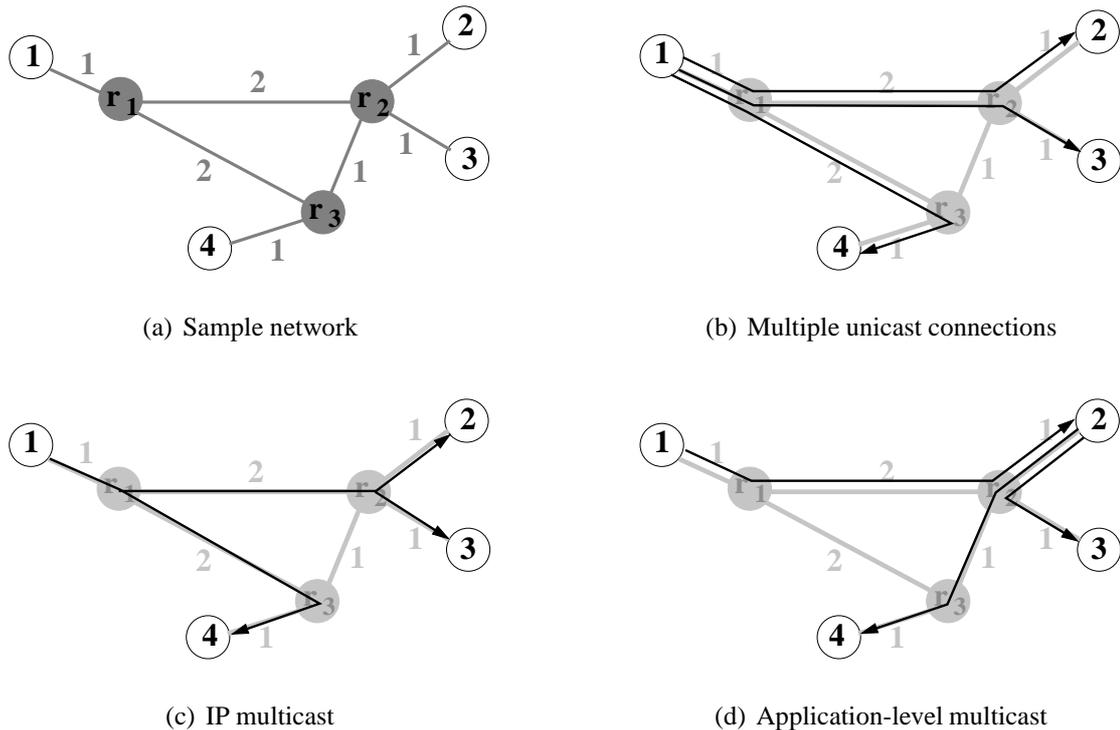


Figure 8.1: Group communication

are nodes of a multicast distribution tree are required to manage some state information about the members of each multicast group.

Currently, there exist a number of routing protocols for the realization of IP multicast. Different protocols are used for *intra-domain routing* and for *inter-domain routing*. Intra-domain routing can be classified into two types according to their tree building approach. Protocols of the first type construct a distribution tree for each sender of a multicast group. These *sender-specific trees* can be calculated such that the network load and the end-to-end delays for the receivers are optimal. But a severe drawback is that routers have to manage a large amount of membership information for groups with many senders (as is the case for most distributed interactive applications). Examples of protocols with sender-specific distribution trees are the Distance Vector Multicast Routing Protocol (DVMRP) [265], Multicast Open Shortest Path First (MOSPF) [174], and Protocol Independent Multicast-Dense Mode (PIM-DM) [5].

Intra-domain multicast routing protocols of the second class lessen the management overhead for the routers significantly by constructing a single *shared tree* per multicast group. However, the resulting end-to-end delays and the network load might be higher than for sender-specific trees. Representatives of this protocol class are Protocol Independent Multicast-Sparse Mode (PIM-SM) [62] and Core-Based Trees (CBT) [10].

For multicast traffic crossing domain borders, inter-domain routing protocols such as the Multicast Source Discovery Protocol (MSDP) [65] or the Border Gateway Multicast Protocol (BGMP) [247] are used. They allow the interoperability between different intra-domain protocols. In addition, they limit the negative effects of some intra-domain routing protocols such as the periodic flooding of the network by DVMRP and PIM-DM.

The protocols described so far make IP multicast a best-effort service where data packets are delivered as well as the current network conditions allow. For applications requiring guarantees about delivery parameters such as packet loss rate, bandwidth, and end-to-end delays [274], *Quality of Service* (QoS) infrastructures were proposed: IntServ (Integrated Services) [15] and DiffServ (Differentiated Services) [13] are two examples. IntServ realizes QoS by reserving resources individually for each data stream [16, 43], which requires state information and a complex logic in all routers. DiffServ is less complicated and does not allow the explicit reservation of resources. Instead, a data stream is assigned one of three service levels, and routers prioritize data packets that have a higher level. Thus, DiffServ facilitates only relative QoS.

8.1.1.2 Discussion

Despite the fact that the work on IP multicast started almost 20 years ago [41], it is not yet deployed on a large-scale, and its architecture suffers from various problems. One fundamental issue is that multicast routers have to maintain state information for each group. This is contrary to the end-to-end design principle of unicast IP where routers are kept as simple as possible in order to achieve high performance and to keep the cost for the network infrastructure low. In contrast, IP multicast requires “smart” routers. Moreover, the state management restricts the architecture’s scalability with respect to the number of multicast sessions that can be handled simultaneously.

Another structural problem is the addressing scheme used by IP multicast: Group addresses can be chosen individually by the multicast group (i.e., by the user or by the application) so that address collisions are possible. To prevent this, there is a number of proposals for the allocation of unique addresses: The session directory SDR [105] maintains a global list of existing multicast sessions and their addresses. With GLOP [166], blocks of multicast addresses are allocated statically. And the Multicast Address Allocation Architecture (MAL-LOC) [248] facilitates a dynamic selection of addresses. None of these schemes has gained unanimous acceptance. As long as multicast address allocation is not handled exclusively by a single mechanism, the duplicate selection of addresses cannot be excluded.

The concept of open multicast groups implies that a sender cannot control who receives the data. Similarly, it is not possible to prevent malicious senders from interfering with a

multicast session, e.g., by flooding the session in a denial-of-service attack. These problems together with the addressing issues can be solved with Source-Specific Multicast (SSM) as first proposed by Holbrook and Cheriton in [118]. Here, a multicast session is bound to the single sender, and the sender's IP address is part of the session address. Thus, session addresses are unique. A receiver joins the session by subscribing to the session address. On the basis of this address, the path from the receiver to the sender can be determined easily so that a distribution tree is constructed by reverse path forwarding. In addition, the sender can control who is attached to the distribution tree. Other senders are not permitted by definition. However, SSM is only applicable for applications with exactly one sender and is therefore not well-suited for distributed interactive applications. Moreover, routers still have to maintain state information.

The number of routing and address allocation protocols for IP multicast as listed above shows how complex the architecture is. Together with the demand that the different protocols must be interoperable, the administration of IP multicast is very difficult. Furthermore, not all routers are multicast-enabled. These routers can be traversed by multicast traffic only via tunnels encapsulating multicast data within unicast packets. These tunnels need to be configured manually, which increases the administrative overhead even more. This situation is expected to improve only when IPv4 will be replaced with IPv6, which offers native multicast support and integrates IGMP into ICMP [42, 34]. In the current Internet, the MBone (Multicast Backbone) is an overlay network of multicast-capable routers [61].

Aside from routers that are not multicast-capable, multicast deployment is also hindered at the network's edge: Dial-in connections with modems, ISDN, or DSL do not support multicast. Also, many firewalls prohibit multicast traffic and therefore prevent end-systems from participating in multicast sessions. To overcome these limitations, a tunnel between the end-systems and a multicast-enabled gateway is necessary (e.g., the dial-in gateway proposed by Kuhmünch in [141]).

From the perspective of the Internet Service Providers (ISPs), IP multicast poses economic questions in addition to the technical challenges. The high administrative effort required for IP multicast also results in high deployment costs for an ISP. In addition, billing users or neighbor ISPs for traversing multicast traffic is complicated since the network resources consumed by in- and outgoing traffic are not the same. Diot et al. discuss management issues with IP multicast in detail in [48].

IP multicast provides best-effort delivery of data to an arbitrary number of receivers. But many applications also need additional transport layer functionality such as reliability, source ordering, restricted access to multicast sessions (i.e., session control), as well as flow and congestion control for group communication. Since this functionality is considerably more

complicated to realize than for unicast, solutions are still under investigation. It also seems that the demands of different applications are too diverse for a single generic multicast transport protocol.

To conclude, IP multicast suffers from a vicious circle, which is difficult to break: The current multicast architecture in the Internet has various technical and administrative problems preventing a faultless flow of multicast traffic. As a consequence, only few users and applications actually employ multicast. But as long as there are no “killer applications” with a high number of users, there is no pressure to improve and deploy multicast on a large scale.

8.1.2 Application-Level Multicast

A promising alternative to IP multicast is Application-Level Multicast (ALM) [30]: The key idea is to use the end-systems as nodes in a multicast distribution tree. As depicted in Figure 8.1(d), the end-systems are interconnected via unicast, and packets are duplicated at the application layer when necessary: 2 copies all incoming traffic from 1 to the outgoing links to 3 and 4. The construction and the maintenance of the distribution tree are also handled at the application level without any support from the network.

8.1.2.1 Discussion

ALM eliminates several key problems of IP multicast: From the router’s perspective, ALM is equivalent to unicast. Thus, it is not necessary for a router to maintain additional state information for group communication, and they remain very efficient for point-to-point traffic. While routers are kept simple, the logic for multicast is pushed to the network’s edge which complies with a fundamental design principle that was vital for the success of today’s Internet. Furthermore, ALM can be deployed immediately without any changes to the existing network infrastructure and without any extra administrative overhead.

The membership in an ALM session can be managed by the application. In Instant Collaboration, only application instances that were invited previously are allowed to join a multicast session [83]. Similarly, it is possible to explicitly define the rights to send or receive data.

An address collision can be prevented easier than with IP multicast since sites are attached explicitly to the distribution tree and are identified by their unicast address. The namespace of the group addresses themselves is determined by the application (or the ALM protocol) and can be significantly larger than the one given by class D IP addresses (32 bit in IPv4). Also addresses might be unique by definition, e.g., by including the sender’s IP address as in SSM [118].

In general, dial-in connections such as ISDN or DSL are able to handle ALM traffic as well as regular unicast traffic. Firewalls might be problematic when certain port numbers are disabled, or when UDP is used as transport protocol, which is often not allowed to pass firewalls.

But ALM also has some major drawbacks when compared to IP multicast: An ALM distribution tree consumes more network resources than the one that would be built by IP multicast. In Figure 8.1(d), the physical link between 2 and r_2 is used three times. Moreover, when receivers are not connected directly to the sender, packets are transmitted over more physical links than with IP multicast, increasing the end-to-end delay. Inner nodes of the ALM tree need to process and forward packets so that the application load of these sites becomes higher. And the time consumed for processing packets at the end-systems increases the propagation delays even more. At the same time, ALM is more efficient than group communication with direct unicast connections from the sender to each receiver since several physical links are traversed only once. In other words, ALM is somewhere in the middle between IP multicast and n point-to-point connections.

An ALM protocol for the communication of a distributed interactive application needs to take care of various aspects: Besides the core functionality for building and maintaining the distribution tree (i.e., routing), reliable and ordered transport of data [127], flow and congestion control [250], as well as membership management [70] have to be considered. For all these aspects, generic solutions are desirable so that distributed interactive applications can choose an appropriate ALM protocol without having to design and implement the same functionality repeatedly. In the following, we concentrate on the routing functionality of ALM protocols.

8.2 ALM Routing for Distributed Interactive Applications

Existing approaches for ALM routing focus on network characteristics (e.g., latency) to construct the multicast distribution tree. Their aim is to limit the impairment of this tree when compared to the optimal tree that would be realized by IP multicast. As long as those network characteristics remain constant and no changes in the set of session members occur, all packets from a sender will take the same paths towards the destinations. This approach is well-suited when all packets have to be delivered to all receivers with the same priority, e.g., in a multi-destination file transfer.

However, a number of applications exist where the priority of a packet may be different for the individual receivers. For instance, in a multi-player game the actions of a player are important for competing players close by, and these players should receive operations with

a very low delay. Other players may be able to tolerate a higher delay. Low end-to-end delays are also essential for concurrency control mechanisms (see Section 4.4): The lower the delay, the lower is the probability for concurrent operations for short-term inconsistencies. Here, low delays are particularly important for sites interacting with each other directly (e.g., modifying the properties of the same object on a shared whiteboard page). Furthermore, a packet's priority may change over time for some or all receivers. For example, if sensor data is transmitted by a sender, this data may typically have a low priority for all receivers unless an extreme sensor reading occurs, which requires the transmission of a packet with very low latency to some receivers. Traditional tree routing algorithms are unable to handle such situations, they do not take application-level priorities into account.

Thus, a novel routing algorithm is designed in this thesis, which combines application-level priorities and network characteristics when building a multicast distribution tree [263]. Since multicast routing is handled at the application level, integrating application knowledge into the routing decision is straightforward and introduces little overhead. The general idea of this approach is to allow the sending application to assign a priority to each pair of packet and receiver. The higher the priority, the more direct will the path be that the packet takes towards its destination. But as we will see later, the cost for reduced latency is higher consumption of network resources. Thus, the key challenge is to find an algorithm that also takes this tradeoff into account.

In Section 8.3, basic algorithms for building distribution trees are discussed, and metrics to determine the quality of a given tree are presented. Then, existing ALM approaches are outlined in Section 8.4. The algorithm for the construction of multicast distribution trees that take application-level semantics into account is described in Section 8.5. Section 8.6 contains an evaluation of the presented algorithm by means of simulation. Last, an ALM protocol is introduced, which is based on this routing algorithm in Section 8.7.

8.3 Distribution Trees and Metrics

Given that the underlying network is not partitioned, each participant of a session is able to connect to all other members via unicast. From the application-layer's perspective, the graph of participants and unicast connections is fully connected. Figure 8.2(b) depicts this graph for the sample session of Figure 8.2(a). Note that Figure 8.2 shows only those end-systems and routers that participate in the sample multicast session; the entire network might be much larger. For easier discussion, we use the term “graph” for the application-level network of unicast connections among end-systems and the term “network” for the underlying physical network (of end-systems and routers).

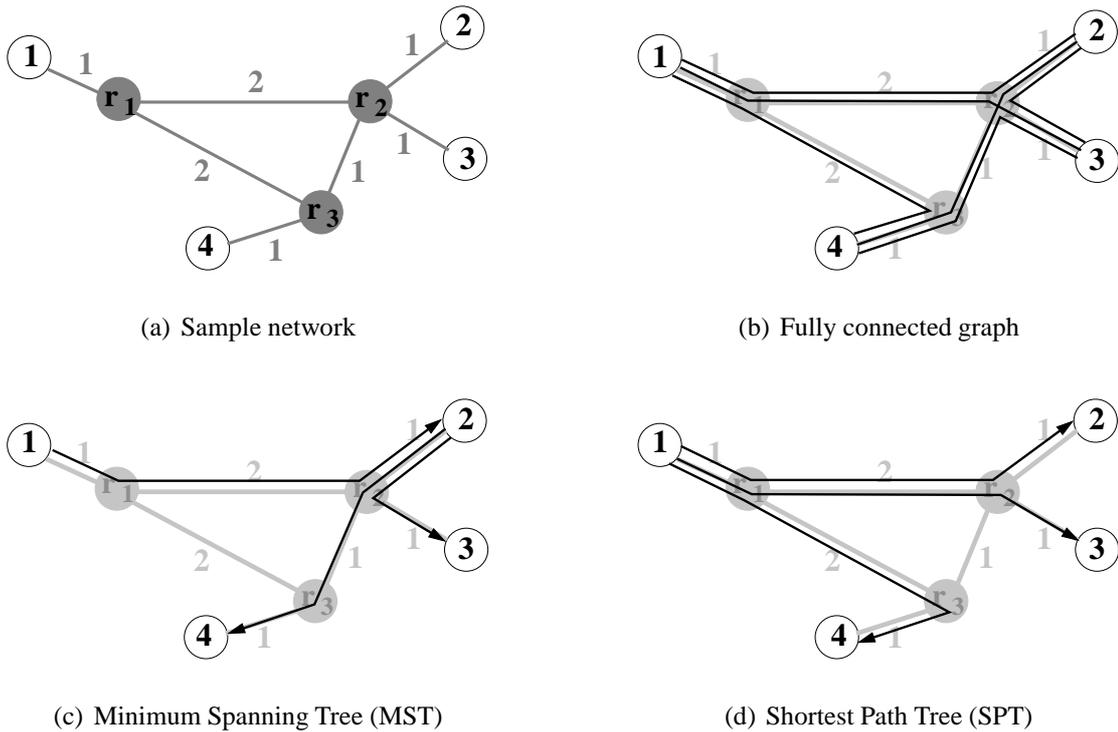


Figure 8.2: Distribution trees

Moreover, the following notations are used: Let $G = (V; E)$ be a fully connected, directed, and weighted graph where $V = \{i\}$ denotes the set of nodes and $E = \{e_{ij}\}$ the set of edges. V contains only those end-systems that participate in the considered multicast session. We define the node $s \in V$ as source (root) and the remaining nodes $R = V \setminus \{s\}$ as receivers. The directed edge e_{ij} connects the end-system i with j . Each edge has a certain weight $w(e_{ij}) > 0$. Depending on the application, different properties of an edge can be used as weights (e.g., bandwidth, latency, loss rate, etc). For distributed interactive applications, the end-to-end delays are especially important so that we concentrate on these as weights (see discussion in Section 4.4). For instance, in Figure 8.2(c), the weight of the edge e_{12} is 4.

The task for an ALM routing algorithm is then to select a sufficient subset of E for a multicast distribution tree T so that a sender can reach all receivers, i.e., T connects all application instances in V . In the following, we focus on source-specific trees. A distribution tree T is therefore given by $T \subset E$ with (1) $\forall j \in R \exists e_{ij} \in T$ and (2) T does not contain any cycles. This tree should be constructed such that the resulting end-to-end delays and the network resources consumed are optimized. In the remainder of this section, we discuss metrics measuring the quality of T and also present basic tree-building algorithms.

8.3.1 Tree Height and Fanout

Basic characteristics of a distribution tree are height and fanout. The **height** of a tree is defined as the maximum number of nodes that are traversed on the path from the sender to a receiver. For the tree depicted in Figure 8.2(c), the height is 2. A high tree might result in high end-to-end delays for receivers lower in the tree. The **fanout** of a node i denotes the number of outgoing links. For 2 in Figure 8.2(c), the fanout is 2. Increasing the average fanout for T decreases its height when the number of nodes remains constant. The maximum possible fanout for T depends on the application and on the available resources (e.g., bandwidth for outgoing network traffic).

8.3.2 Resource Usage and Minimum Spanning Tree

The **link stress** measures how many copies of a specific packet are transmitted over the same physical link of the network. For instance, the link stress of $1, r_1$ in Figure 8.2(d) is 3. Usually, the maximum and the average link stress over all physical links used by T are examined. Since a higher link stress implies that more network resources are consumed, both maximum and average link stress should be as low as possible for T .

The link stress alone does not capture how many network resources are actually absorbed by T . Thus, we define the **resource usage** C_R as the product of link stress and link weight, summed over all physical links of the network used for T . This sum is equivalent to the sum of the weights of all edges in T since the weight of a physical link is implicitly contained in the weight of all edges using this link. For instance, in Figure 8.2(c) the resource usage of T in the underlying network is $w(1, r_1) + w(r_1, r_3) + 3w(r_2, 2) + w(r_2, 3) + w(r_2, r_3) + w(r_3, r_4) = 9 = w(e_{12}) + w(e_{23}) + w(e_{24})$. C_R is therefore given by:

$$C_R(T) = \sum_{e_{ij} \in T} w(e_{ij}). \quad (8.1)$$

When minimizing C_R , the distribution tree T is a *Minimum Spanning Tree* (MST). An MST for our application-level graph of four end-systems with 1 as sender is depicted in Figure 8.2(c). If the graph G is undirected, the MST can be calculated with the well-known algorithms of Kruskal [140] and Prim [201]. For those graphs, the MST is well-suited as a shared distribution tree since an MST algorithm selects the same set of edges for T independent of the sender. An algorithm for directed graphs is proposed by Edmonds [54].

8.3.3 End-to-End Delays and Shortest Path Tree

Before a packet reaches a receiver $t \in R$, it traverses the path q_t from the sender s to t as defined by the distribution tree T : $q_t = \langle e_{sj_1}, \dots, e_{j_{n-1}j_n}, e_{j_nt} \rangle$ with $e_{ij} \in T$. This path determines the end-to-end delay $d(t)$ experienced by t :

$$d(t) = \sum_{e_{ij} \in q_t} w(e_{ij}). \quad (8.2)$$

The total delay for the distribution of a packet from the source to all receivers is measured by the **cumulative end-to-end delay** C_D :

$$C_D(T) = \sum_{t \in R} d(t). \quad (8.3)$$

Optimization (i.e., minimization) of C_D leads to a distribution tree T that is a *Shortest Path Tree* (SPT). The SPT can be calculated with the algorithm of Dijkstra [46]. Since G is a fully connected graph, the SPT is a tree where all receivers are linked directly to the sender s . Figure 8.2(d) shows the application-level SPT for the example with 1 as sender. Commonly, this would be regarded as “normal” unicast rather than application-level multicast. With respect to C_D , the SPT is optimal: $C_D = 11$ in Figure 8.2(d), as opposed to $C_D = 17$ for the MST in Figure 8.2(c). But at the same time, it causes a very high consumption of network resources: $C_R = 14$ in Figure 8.2(d) compared to $C_R = 9$ in Figure 8.2(c). Furthermore, building an SPT is not possible when the sender’s bandwidth is not sufficient to serve all receivers simultaneously¹.

Another measure that evaluates the end-to-end delay experienced by a receiver t is the **Relative Delay Penalty** RDP :

$$RDP(t) = \frac{d(t)}{w(e_{st})}. \quad (8.4)$$

$RDP(t)$ compares the end-to-end delay that is achieved for t to the smallest possible delay, which is equal to the unicast delay from s to t . $RDP(t)$ therefore assesses the optimality of the path q_t where $RDP(t) = 1$ is the optimum. For the MST distribution tree depicted in Figure 8.2(c), the RDP for the receiver 3 is 1.5. By definition, for SPT distribution trees $RDP(t) = 1 \forall t \in R$.

¹This could also be the case for some inner nodes of an MST but is less likely.

8.4 Related Work

We will now examine existing application-level multicast protocols and discuss whether their algorithm to construct a distribution tree is applicable in the scenario where a delay-sensitive application wants to define receiver-specific priorities on a per-packet basis. In Section 8.6.5, simulation results are compared. Typically, ALM distribution trees are built on the basis of path characteristics such as end-to-end delays, available bandwidth and packet loss rates. Besides building a stable and robust tree, the main goal is to minimize the additional routing overhead compared to native IP multicast.

Yoid [70] manages two independent topologies for the exchange of data. Control messages and other data that needs to be delivered with high reliability are transmitted over a mesh, i.e., nodes might receive duplicates. For the distribution of regular application data, Yoid creates a shared multicast tree: Each node i selects another node j as parent, preferably such that the delay between i and j is low. Receivers gather the set of possible parents by periodic control messages and explicit queries. An initial list can be obtained from a so-called rendezvous host during the bootstrap phase. Aside from the network delays, the maximum fanout of a potential parent is considered in the choice of a parent node. Because the initial list of possible parents is usually incomplete and the fanout of nodes is constrained, the resulting distribution tree may be suboptimal. As a consequence, nodes periodically ping other session members in order to find a better parent and to ultimately approximate an MST. In case a node changes its parent, special care has to be taken in order to prevent cycles that would partition the tree. An alternative method to select a parent node in Yoid is provided for the transfer of large data files: Nodes connect to the parent that caches the largest amount of data. This algorithm makes use of application-level knowledge but completely neglects network characteristics.

Other examples of tree-building ALM protocols are the Application-Level Multicast Infrastructure (ALMI) protocol [194], the Host Multicast Tree Protocol (HMTP) [276], the Banana Tree Protocol (BTP) [110], and Overcast [127]. In ALMI, a session control server centrally calculates and maintains the distribution tree that is an MST on the basis of end-to-end delays. In order to allow reconfigurations of the tree during an ongoing session and to absorb short-time partitions, each node of the tree caches the most recent data packets for a certain amount of time. All other tree-building approaches mentioned above form self-organizing distribution trees where nodes select an appropriate parent, and they all implement mechanisms for integrating new members, detecting loops and partitioning, and for optimizing the tree by rearrangement. Unlike the other protocols, Overcast builds sender-specific trees instead of a single shared tree. The main criterion for the distribution tree in Overcast is to maximize throughput for each receiver, which qualifies Overcast mainly for the transfer of bulk data.

The main advantage when constructing the distribution tree by parent selection is the low complexity of the local optimization process. Thus, such a routing protocol scales well with the number of participating nodes. However, in many cases local optimization is not able to determine the optimal tree with respect to end-to-end delays or consumed network resources. In [110], Helder and Jamin therefore present several mechanisms to improve the parent-selection scheme. Depending on a cost function that defines whether a node should switch its current parent, the resulting distribution tree approximates either an MST or an SPT. For example, switching to a parent with a low cost path to the source will converge to an SPT. Since it might take a long time until the approximation converges to the optimum tree, this mechanism seems not to be suitable for a highly dynamic environment where priorities might change on a per-packet basis.

Another approach to build distribution trees is the Topology Aware Grouping (TAG) algorithm [144]. Here a node selects its parent such that both share a large portion of their path to the sender on the underlying network. A new node traverses the tree starting from the sender until an appropriate parent is found. The topology of the underlying network is inferred either by tools such as traceroute or by topology servers [71]. The ALM tree built by TAG is source-specific, and data originating from other session members is distributed via the root of the tree. This introduces a severe performance penalty for distributed interactive applications where most of all session members send data.

With TMesh [267], Wang et al. propose to add additional links to an ALM tree. These shortcuts reduce the number of hops on the way from a sender to the receivers, and TMesh seeks to optimize the average end-to-end delays for the whole group and builds a rather stable tree. Thus, TMesh seems to be not flexible and fast enough to facilitate delay optimization for certain receivers in an environment where priorities change dynamically.

Instead of constructing a tree directly, Narada [30] employs a two-step process: First, a mesh is built among the participating end-systems. Then, Narada runs a distance vector protocol with latency and bandwidth [29] as the routing metrics on top of the mesh. The resulting tree is a sender-specific SPT based on the underlying mesh. Thus, the routing protocol of Narada does not facilitate priority-based routing. As in Yoid, the mesh increases the robustness of Narada against failures of network or nodes. The mesh is also used to manage the otherwise independent source-specific distribution trees. But the crucial factor in this approach is the quality of the mesh that must balance the number and the characteristics of the used unicast links. If there are too many links in the mesh, the resulting distribution topology will resemble a star of unicast connections from the sender to all receivers. As in Yoid, joining end-systems obtain a list of current session members by a bootstrap mechanism and connect to one or more listed nodes. Then, members periodically add links to the mesh that improve the overall routing performance and remove links that are rarely utilized by a distribution tree.

Like Narada, Gossamer [26] also employs the tree-over-mesh approach where the mesh is constructed in order to minimize latencies of the distribution tree. The number of connections a node can maintain at a certain point in time is explicitly restricted with Gossamer in order to take bandwidth limitations into account.

Approaches where application-level semantics are used for routing can be found in the area of content delivery networks: The common idea of Bayeux [277], Chord [232], and Content Addressable Networks (CAN) [205] is to realize a scalable lookup service for objects (e.g., files, end-systems, etc.) where the responsibility for managing the object space is shared equally among a network of peer nodes. The multi-hop lookup path for a target object (e.g., the receiver of a message) is determined on the basis of certain properties of the (hash-generated) destination address. Thus, the application-level semantics can be applied when assigning an object's key, i.e., the semantics are rather static. For example, in Bayeux the current node uses the i -th digit of an object's key to resolve the next hop towards the destination. In contrast to the previously discussed ALM routing protocols, these content delivery networks base their routing decisions (almost) exclusively on application semantics. Consequently, the resulting distribution tree may be very inefficient with respect to end-to-end delays and link stress [277, 206]. In [207], Ratnasamy et al. therefore improve the CAN algorithm by incorporating some knowledge about the underlying network topology: Each node determines its geographical area by measuring its round-trip time to some well-known nodes (so-called "landmarks"). The address space is then allocated such that the latency of the next hop is low when routing along the lookup path. Assuming that the end-to-end delay between two nodes sharing a certain prefix of their IP addresses is low, [75] takes this concept one step further and selects the node with the IP address closest to the hashed object key as the next hop.

Summing up, existing approaches for ALM routing do not offer the desired functionality of data delivery with dynamic per-packet and per-node priorities. But they successfully address other important aspects of an ALM protocol such as scalability, interoperability with IP multicast, reliability and robustness, bootstrapping, and data naming. In the next section, we therefore concentrate on an appropriate routing algorithm for the scenario with delay-sensitive applications.

8.5 Priority-Based Application-Level Multicast Routing

An ALM routing algorithm builds a distribution tree by connecting the end-systems with unicast links. The resulting tree should use the resources of the underlying network efficiently, which requires some knowledge about the network topology. Since on the application level there is no direct access to topology information, observable parameters (e.g., latency) may be used to deduce a certain amount of knowledge (see Figure 8.3): When node 1 has a high

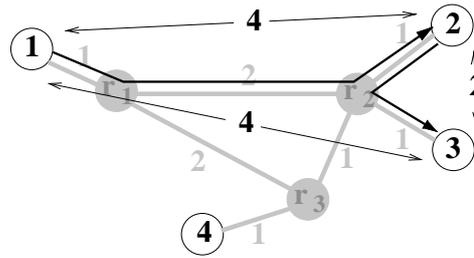


Figure 8.3: Joint path to distant receivers

delay to both nodes 2 and 3, and 2 has a low delay to 3, it is likely that the edge e_{12} shares a significant portion of the physical link with the route e_{13} . Other, more sophisticated possibilities to deduce the network's topology from the application level include tools such as traceroute, landmark routers with well-known locations [207, 71, 220], and methods analyzing one or several network parameters that can be measured at the end-systems (e.g., loss patterns [148, 208]). In the following, we concentrate on unicast latencies as the network parameter determining the ALM tree².

In this context, we consider the Minimum Spanning Tree (MST) and the Shortest Path Tree (SPT) (see Section 8.3). The MST optimizes the resource usage C_R of the multicast tree but the path length is not taken into account and can cause very long end-to-end delays. Hence, using an MST is only reasonable when end-to-end delays are not an issue (e.g., for non-interactive data dissemination). When building an SPT from the unicast delays, the distribution tree will consist of separate unicast connections from the sender to each receiver. With respect to the end-to-end delays C_D , the SPT is optimal but causes a very high consumption of network resources. Furthermore, building an SPT is not possible when the sender's bandwidth is not sufficient to serve all receivers simultaneously.

8.5.1 Introducing Application-Level Semantics

The aim of this chapter therefore is to construct application-aware distribution trees that balance the characteristics of MST and SPT: For each packet-receiver pair, the application may provide a priority. Depending on this priority, the forwarding path of the packet should gradually change from the MST path to the SPT path so that the end-to-end delays for receivers with a high priority are optimized at the cost of a higher consumption of network resources. For nodes with a lower priority, the delay is less important so that paths can be used that consume less resources.

²We will ignore that unicast routing protocols may give suboptimal routes and assume that the underlying unicast routing algorithm causes direct paths to a node to be shorter than any indirect path over intermediate nodes.

In order to find an algorithm with this property, first the two metrics of resource usage C_R and cumulative end-to-end delay C_D as defined in Equations 8.1 and 8.3 are combined by using one common application priority for the entire distribution tree. The optimization of the combined metric allows the gradual transition from MST paths to SPT paths as the application priority increases. In a second step, we generalize the metric such that one priority may be given for each destination. Its optimization leads to a tree where every path from the sender to a destination changes from the MST path to the SPT path. Finally, an efficient algorithm is presented, which provides a very good approximation for the optimal distribution tree with respect to the last metric.

Let $p \in [0; 1]$ be the application's priority with which it wants to deliver data: 1 means that the end-to-end delay for all receivers should be as low as possible, 0 denotes no special delay requirements. A balancing cost function C can then be defined as follows:

$$C(T) = (1 - p) C_R(T) + p C_D(T). \quad (8.5)$$

Figure 8.4 visualizes the effect of p when building the optimum distribution tree according to C for a sample ALM session. The participants of the session are numbered from 1 to 6, while intermediate routers of the underlying network appear as unmarked nodes. The corresponding table contains the pairwise end-to-end delays. Let node 2 be the sender. The resulting distribution trees that are optimal with respect to C are depicted in Figure 8.5. When p is increased, nodes further away move up in the tree, reducing the end-to-end delays to the sender, until for $p = 1.0$ a star-like SPT is reached. As can be seen from the graphs, the number of possible trees for a small overlay network with only 6 nodes is very limited.

Following, the cost function C is generalized for the case of individual per-receiver priorities where information may be of high importance to some receivers (and should therefore be delivered on a direct path) and of lower importance to other receivers. Let $p : V \rightarrow [0; 1]$ be the per-node priorities for a sender s . They can easily be integrated into C_D , defining the cost function C_D^p :

$$C_D^p(T) = \sum_{t \in R} p(t) d(t). \quad (8.6)$$

Integrating the per-node priorities into C_R is more difficult since the costs are calculated over the edges of the tree and not per receiver. However, in an MST, the relevant cost for a receiver is the weight of the edge over which it is connected to the rest of the tree. Consequently, the priority of a node can be assigned to this edge. This leads to the cost function C_R^p :

$$C_R^p(T) = \sum_{e_{ij} \in T} (1 - p(j)) w(e_{ij}). \quad (8.7)$$

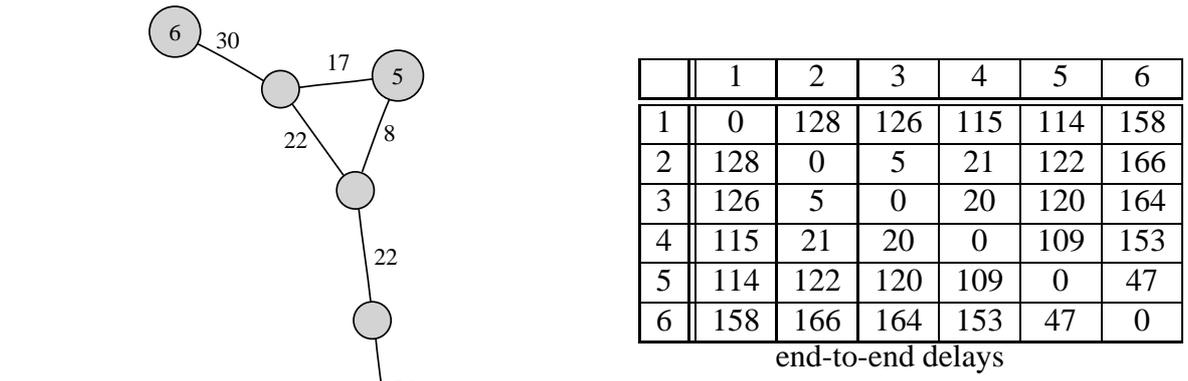


Figure 8.4: Example graph

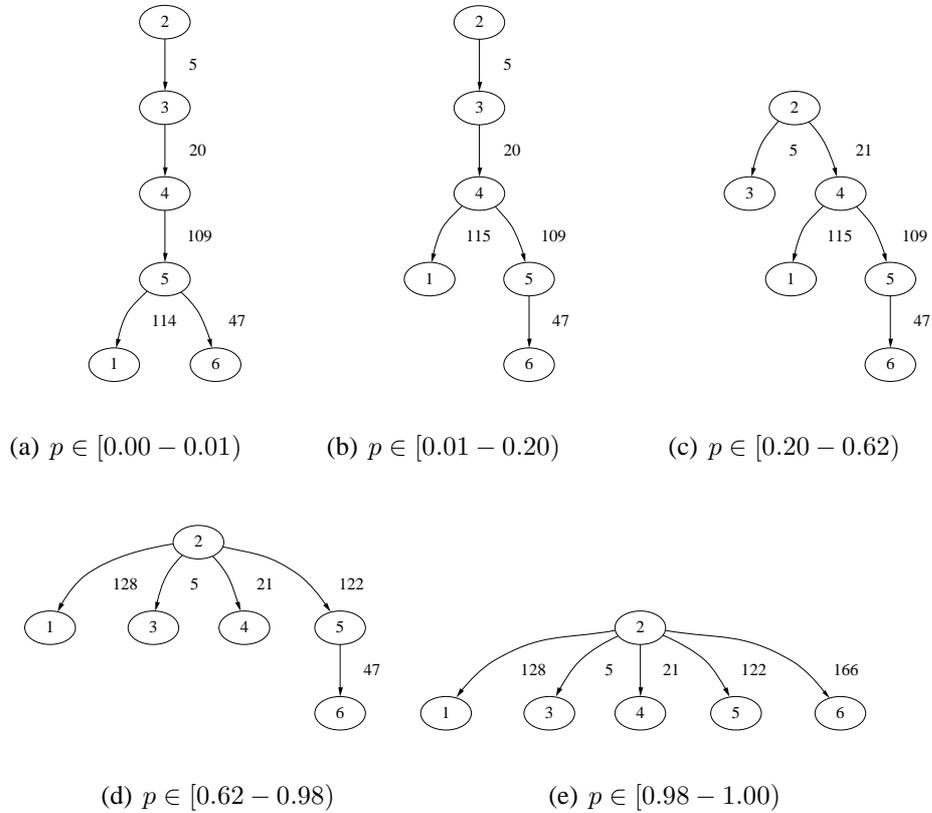


Figure 8.5: Optimal distribution trees

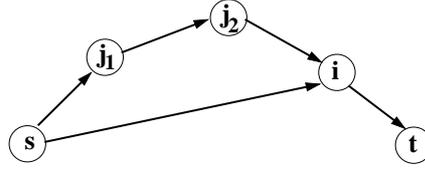


Figure 8.6: Approximated path

The total costs C are then defined as

$$C(T) = C_R^p(T) + C_D^p(T). \quad (8.8)$$

Note that C specializes to C_D if $\forall v p(v) = 1$, and to C_R if $\forall v p(v) = 0$. This means that the node priorities determine the structure of the minimum cost tree with the extremes SPT and MST.

8.5.2 The Priority-Based MST Algorithm

Direct optimization of this cost function is computationally complex so that an algorithm that constructs a distribution tree T according to C would not be well-suited for an ALM protocol. However, it is possible to approximate C such that the tree that is optimal with respect to the approximation can be calculated by an MST algorithm. For this purpose, the cost function needs to be based solely on the tree's edge weights, and not on complete paths to individual receivers.

Therefore, C_D^p must be approximated: The general idea is to split the complete path q_t from the sender s to a receiver t into the last edge of the path e_{it} and the path of all previous edges $\langle e_{sj_1}, e_{j_1j_2}, \dots, e_{j_ni} \rangle$. Now, the cost of the path from s to i is approximated with the cost of the direct edge e_{si} so that $w(e_{si})$ is a lower bound for the actual path costs (see Figure 8.6). This leads to a simplified approximate formulation for the global costs C :

$$\begin{aligned} C(T) &= \sum_{e_{ij} \in T} (1 - p(j))w(e_{ij}) + \sum_{t \in R} p(t) \sum_{e_{ij} \in q_t} w(e_{ij}) \\ &\approx \sum_{e_{ij} \in T} (1 - p(j))w(e_{ij}) + \sum_{t \in R, i \in V: e_{it} \in T} p(t)(w(e_{si}) + w(e_{it})) \\ &= \sum_{e_{ij} \in T} w(e_{ij}) + p(j)w(e_{si}). \end{aligned}$$

The last equality follows from the property that a spanning tree of a graph has the same number of edges as there are target nodes in the graph. Consequently, both sums are calculated over the same set of edges. In order to minimize the approximated costs C , an MST algorithm

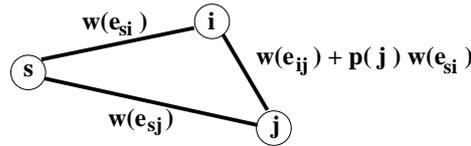


Figure 8.7: Modified edge weights

can be applied to the graph G with modified weights. The new weights w' are set to

$$w'(e_{ij}) = w(e_{ij}) + p(j)w(e_{si}). \quad (8.9)$$

The calculation of $w'(e_{ij})$ is illustrated in Figure 8.7. With increasing $p(j)$, indirect links e_{ij} to the target node j will become more expensive, and eventually such links will be removed from the distribution tree and be exchanged against more direct paths.

We denote the approximating cost function as \tilde{C} :

$$\tilde{C}(T) = \sum_{e_{ij} \in T} w'(e_{ij}). \quad (8.10)$$

In order to calculate the optimum distribution tree for \tilde{C} , a directed MST algorithm has to be applied as it is not a priori known in which direction data is distributed over the edge, and the costs for the two directions may differ. Algorithms to construct MSTs in directed graphs have been described in [31, 54]. We call the combination of modified edge weights and directed MST computation *Priority-based directed minimum Spanning Tree* (PST) algorithm [263]. Pseudo-code for the PST algorithm is presented in the next section.

8.5.3 Pseudo-Code for the PST Algorithm

Figure 8.8 gives the pseudo-code to compute the PST on a graph $G = (V; E)$ for a sender s with priority function p . First, the weights $w'(e_{ij})$ of the directed graph are calculated as given by Equation 8.9. Second, the directed minimum spanning tree is determined according to the algorithm published by Edmonds [54]. This algorithm was originally designed to construct a branching T with maximum total costs \tilde{C} on the basis of G : A *branching* is a directed graph without cycle where each node has at most one incoming edge, i.e., a branching is not necessarily connected. Thus, to build an MST, we define all weights w' to be negative and ensure that the branching T contains $|V| - 1$ edges (maximizing \tilde{C} with negative weights is equal to minimizing \tilde{C} with positive weights).

The basic idea of Edmond's algorithm is to calculate an initial graph T by selecting for each node $i \in R$ the incoming edge with maximum costs. In case T contains a cycle Z , it is broken up by exchanging an edge within Z with an appropriate edge from outside Z . Once

- (1) Compute weights $w'(e_{ij})$ for all edges in E :
- $\forall i, j, i \neq j : w'(e_{ij}) = -(w(e_{ij}) + p(j)w(e_{si}))$
- (2) Compute the directed minimum spanning tree with source s on G :
- Discard all edges $e_{is} \in E$ that target the source node s .
 - \forall nodes $i \in V, i \neq s$: select the edge $e_{ji} \in E$ with maximum weight $w'(e_{ji})$. Let E' be the set of selected edges.
 - While $T := (V; E')$ contains a cycle $Z := (W; F), W \subset V, F \subset E'$ do
 - Find the edge $e_{kl} \in F$ with minimum weight $w'(e_{kl})$.
 - Modify the weight w' of each edge $e_{ij} \in \{e_{ij} | i \in V \setminus W, j \in W\}$:
 $w'(e_{ij}) := w'(e_{ij}) + w'(e_{kl}) - w'(e_{h(j)j})$,
 with $h(j) \in W$ being the predecessor node with edge $e_{h(j)j} \in F$.
 - Select the edge $e_{mn} \in \{e_{ij} | i \in V \setminus W, j \in W\}$ with maximum weight $w'(e_{mn})$, and set $E' := E' \cup \{e_{mn}\} \setminus \{e_{h(n)n}\}$.
 - Build a new graph T by contracting all nodes $i \in W$ into a pseudo-node φ :
 $V := V \setminus W \cup \{\varphi\}$. Modify E and E' by replacing all edges e_{ij} with tail node $i \in W$ or head node $j \in W$ by $e_{\varphi j}$ or $e_{i\varphi}$, and delete edges $\{e_{ij} | i, j \in W\}$.
 Create new weights w' accordingly.
 - Replace all pseudo-nodes $\varphi \in V$ and the corresponding edges in E' by the original nodes and edges. T represents the directed MST with root s .

Figure 8.8: Pseudo-code for the computation of the PST

Z is broken up, the nodes that were part of Z are replaced by a so-called “pseudo-node” φ in order to prevent that the edges within φ are exchanged in future computational steps. The process of exchanging edges is repeated until there are no more cycles in T .

As proven by Camerini, the complexity of Edmond’s algorithm is $O(n^2)$ where n is the number of nodes in V [23]. Theoretically, there exist n^{n-2} different distribution trees on G per sender [24], i.e., when all nodes send data there are n^{n-1} trees in total. However, the simulations and the experimental results indicate that the actual number of trees in a certain scenario is fairly limited [8, 263]. In Section 8.7.4, mechanisms are discussed to limit the number of trees required in a session.

Figure 8.9 shows two sample PSTs for the ALM scenario with four session members. In Figure 8.9(a), the sender 1 assigns a high priority to node 4 and low priorities to 2 and 3. The initial graph calculated by the PST algorithm is a branching with edges $\{e_{32}, e_{23}, e_{14}\}$ that includes the cycle $\{e_{32}, e_{23}\}$. In the next step, e_{42} is determined to be the edge with maximum weight targeting the cycle. Thus, e_{32} is replaced with e_{42} , and the algorithm terminates since T does not contain any more cycles. Note that only node 4 with the highest priority has a direct connection to the sender. As the sender’s priority for 2 increases to 0.4, 2 is also connected directly to 1 (see Figure 8.9(b)).

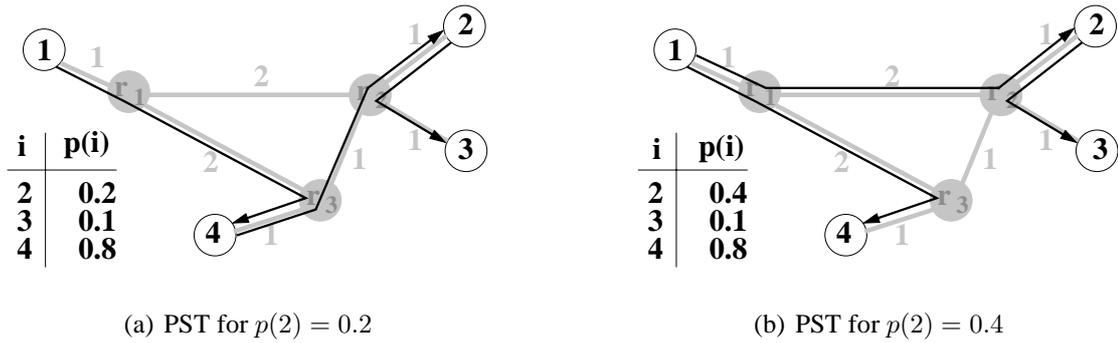


Figure 8.9: Priority-based distribution trees

8.5.4 Considering Constraints for Distribution Trees

In case end-systems have a network connection with limited bandwidth (e.g., modem or ISDN), the PST algorithm might calculate distribution trees that are not an applicable solution because one or more nodes would not be able to handle the induced traffic. This could also happen when the application has strict demands on the end-to-end delays, and packets that are delivered with a delay exceeding a certain threshold are invalid for the application. An algorithm that takes such restrictions into account does not only have to find a distribution tree T on G such that the cost function C as defined in Equation 8.8 is optimized. It also needs to observe constraints such as (1) $d(t) \leq d_{max} \forall t \in R$ in case of an maximum allowable delay d_{max} for a certain scenario, or (2) $b \cdot |\{e_{ij} \in T \mid i = k \vee j = k\}| \leq b_{max}(k) \forall k \in V$ when b denotes the bandwidth consumed for one incoming or outgoing connection, and $b_{max}(k)$ is the maximum bandwidth available at node k .

One possible solution to this novel optimization problem is to approximate the cost function C with \tilde{C} as defined in Equation 8.10 by applying the modified weight function to the original graph (see Section 8.5.2), and by computing the directed MST under consideration of the constraints. Calculating such a constrained minimum cost tree is also known as the Constrained Steiner Minimum Tree (CSMT) problem [86]. Since the CSMT problem is NP-complete [132], a number of heuristics were proposed to approximate the CSMT in polynomial time [139, 204, 214, 242]. Designing and evaluating such an algorithm in order to construct a PST so that restrictions of the available bandwidth or of the maximum allowable delay are taken into account is an issue for future work. For the remainder of this chapter, we concentrate on the unrestricted PST algorithm as described above.

In the next section, simulation results for the PST algorithm are discussed and compared to results of existing approaches. Thereafter, an ALM protocol on the basis of the PST algorithm is presented and analyzed in Internet experiments.

8.6 Simulation Results

The performance of the PST algorithm is evaluated with a basic network simulator, which was developed for this purpose in the course of this thesis. The simulator is event-based and allows packet-level data distribution on arbitrary network topologies. A network topology is characterized by a set of nodes connected via edges with a certain delay. Other factors such as bandwidth, router load, and packet loss are not considered. All network topologies are generated with the Georgia Tech Internetwork Topology Models (GT-ITM) [22] toolkit. The topologies use the transit-stub method, which defines a two-level network with transit domains as the network's backbone and stub domains hosting the end-systems. Edges between nodes are placed using the random model, and the generator's option to introduce extra transit-stub or stub-stub edges is disabled. All end-systems are located on the edge of the network, and all inner nodes act as routers. Unicast connections among end-systems are determined by the shortest path algorithm of Dijkstra [46].

In the following, the characteristics of the PST algorithm are compared to the delay-based MST and SPT approaches on the basis of a realistic application scenario. Comparing the results of PST and MST is especially interesting since many existing approaches seek to construct ALM distribution trees in the shape of MSTs (see Section 8.4).

8.6.1 Simulation Setup

Realistic event patterns to determine application priorities for the simulations are generated by tracing the multi-player game presented in Section 2.4.2. All user actions (e.g., accelerating, turning, shooting, etc.) together with timestamps and information about the current game state are recorded for game sessions with six and eighteen players. In the simulation, each action leads to one packet, which needs to be distributed from the source to the other application instances.

The application-level priorities $p(i) \in [0; 1]$ used for the PST algorithm are based on the relative positions between the spaceships and their orientations (see Figure 2.2). If the spaceship i of a player is in shooting range of another player's ship s , the end-system s of s sets $p(i)$ to 1. We define that i is in shooting range of s if the distance between i and s is less than the maximum range of the laser beam and s is oriented in such a way that it can hit i after conducting at most one turn operation. For players j outside the shooting range of s , p is calculated depending on their geographic distance $d(s, j)$ on the game field: $p(j) = 1 - \frac{d(s, j)}{d_{max}}$ where d_{max} is the maximum distance possible.

A typical distribution of priorities for a game session with six players is depicted in Figure 8.10. Priorities close to 1 are common because the objective is to score points by shooting

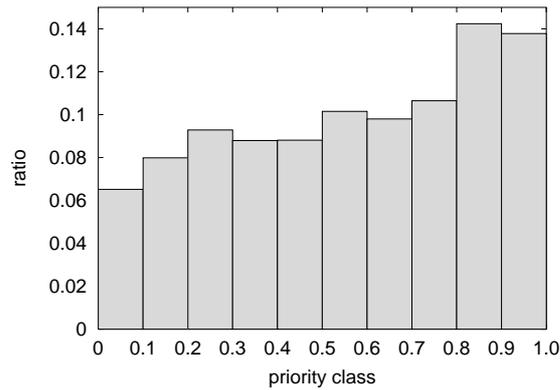


Figure 8.10: Distribution of application priorities

at other players so that players will cluster together instead of spreading out evenly on the game field.

8.6.2 Simulation Results for Six End-Systems

The first simulation scenario is based on a game session with six players. The session lasted for 140 seconds, and during that time span a total of 2,630 events were issued. The priority distribution is depicted in Figure 8.10. Figure 8.4 shows the underlying network topology with end-to-end delays between 5 ms and 166 ms and an average of 100 ms.

At first, the fanout and the height of the distribution trees built by the SPT, MST, and PST algorithms, respectively, are analyzed. Since an SPT connects all receivers directly to the sender, the fanout is 5 for the sender, and the height of all trees is 1. For the MSTs, the maximum fanout is 3 with an average of 1.4 for the inner nodes of the tree (leaf nodes have a fanout of 0 and are not considered here). The average height of the trees is 2.1. The maximum fanout of PSTs is 5 (i.e., the PST is an SPT), and the average fanout for inner nodes of the trees is 1.6, which is only slightly larger than the fanout of MSTs. The average height of PSTs is 1.7.

The delay properties of a specific distribution tree can be measured using the costs C_D^p as defined by Equation 8.6. Figure 8.11(a) depicts the Cumulative Distribution Function (CDF) of C_D^p for the SPT, the MST, and the PST, respectively. By definition, the SPT routing algorithm results in the best distribution of C_D^p , with 90% of all trees having a C_D^p of less than 440 ms. However, the difference between SPT and PST is comparatively small (12 ms at 90%), meaning that the end-to-end delays in the distribution trees constructed with the PST algorithm are on the average only marginally higher for high-priority receivers than the delay on the direct paths. In comparison, MSTs have significantly higher values for C_D^p . The receiver-specific end-to-end delays $d(t)$ (see Equation 8.8) resulted in the following 99% confidence intervals

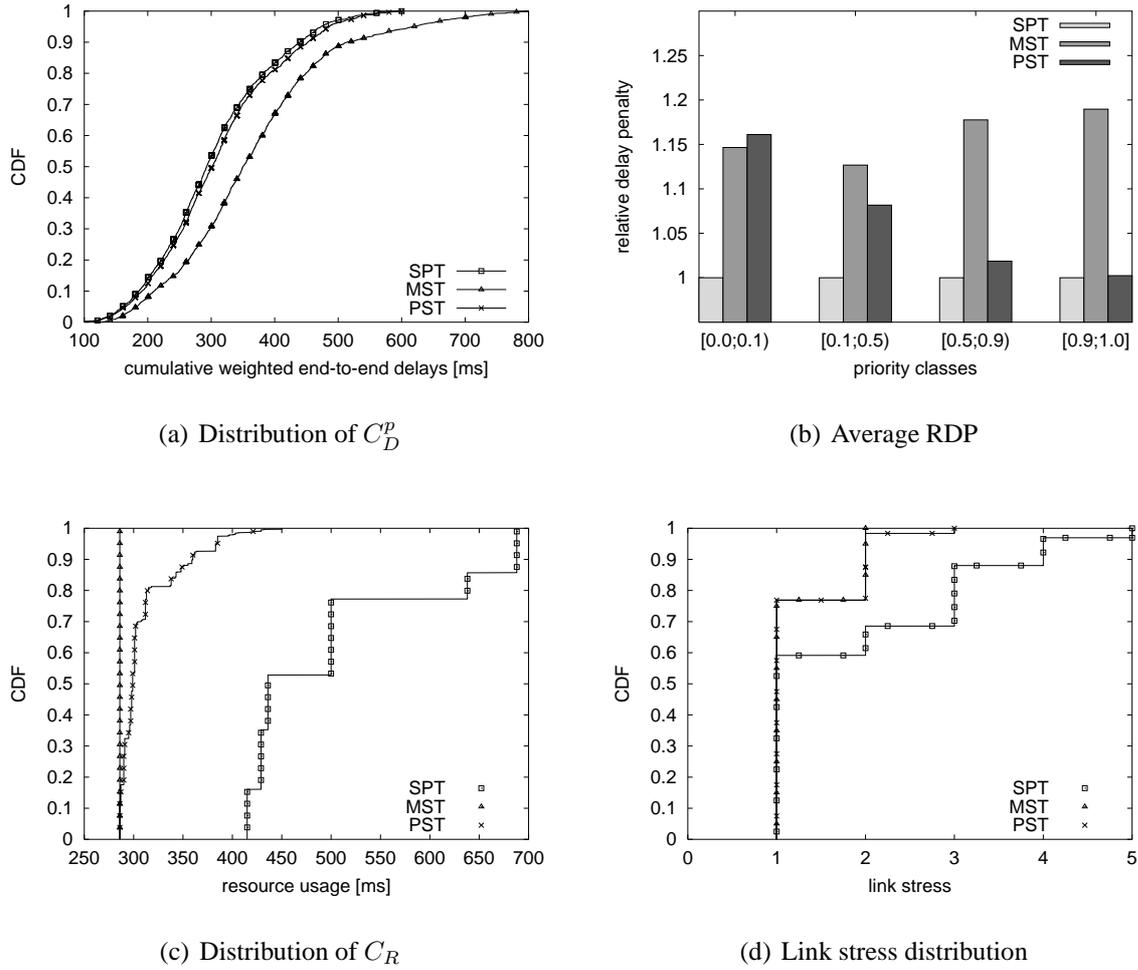


Figure 8.11: Simulation results for 6 end-systems

for this simulation scenario: SPT [98.8; 101.2], MST [118.1; 121.3], and PST [103.9; 106.4]. This also shows that the PST realizes end-to-end delays that come close to the optimum.

The relative delay penalty RDP as defined by Equation 8.4 is a measure for the optimality of the end-to-end delay and compares the actual end-to-end delay of a receiver t to the smallest possible delay (i.e., the unicast delay from s to t). Figure 8.11(b) shows the average RDP values for different priority classes. By definition, for the SPT distribution trees, RDP is 1 for all receivers. The variation of the RDP for MSTs is random. In case of the PST algorithm, the RDP decreases continuously with increasing application-level priorities from 1.16 for receivers t with $p(t) \in [0.0; 0.1)$ to 1.002 for t with $p(t) \in [0.9; 1.0]$. Thus, a delay close to the unicast latency can be achieved for nodes with a high priority. The maximum range of the average RDP is relatively small (0.16) since only six end-systems participated in this simulation scenario, and the distribution trees have paths with at most three hops.

The network load caused by a certain tree can be measured using the resource usage metric C_R as defined by Equation 8.1. C_R takes into account that more than one copy of a packet

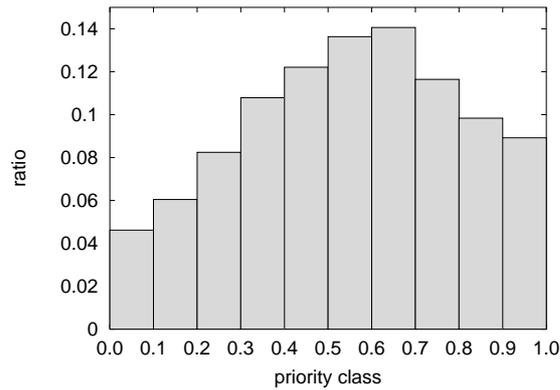


Figure 8.12: Distribution of application priorities

may be sent over the same physical link. The distribution of C_R for the simulation scenario is depicted in Figure 8.11(c). The MST algorithm always selects the same set of edges e_{ij} for its trees, independent of the source node. Thus, C_R is constant at 286 ms, which is at the same time the lower bound for C_R . 70% of all distribution trees built by the PST algorithm have a C_R between 286 ms and 307 ms, which is close to the optimum and far better than the values obtained by SPT. Thus, the optimization of end-to-end delays for certain application instances by PSTs causes only a slight increase in the resource usage when compared to MSTs.

Link stress is another indicator for the network overhead caused by an ALM tree. MSTs result in the lowest link stress with 77% of all distribution trees having a link stress of 1 and a maximum link stress of 2, as shown in Figure 8.11(d). Distribution trees constructed by the PST algorithm come close to these values with the only difference that 1.7% of the trees have a link stress of 3. The link stress for the star-shaped SPT topologies lies between 1 and 5, and only 60% of the trees have a link stress of 1.

8.6.3 Simulation Results for Eighteen End-Systems

For the second simulation scenario, a more complex network topology was created with 42 routers, 80 links, and 18 end-systems participating in a virtual game session. The delays among end-systems lie between 16 ms and 268 ms with an average value of 145.5 ms. During the session's duration of 104 seconds, a total of 6,564 events were issued by all players. With 18 players, the spaceships are spread out more evenly over the game field, which results in the application priorities shown in Figure 8.12.

All distribution trees built by the SPT algorithm have a fanout of 17 at the sender and a height of 1. The maximum fanout of all MSTs is 4 with an average value of 2.0, and the average height of all trees is 3.6. For PSTs, the maximum fanout is 12, and the average fanout of all inner nodes is identical to the one of the MSTs with 2.0. This means that on average the

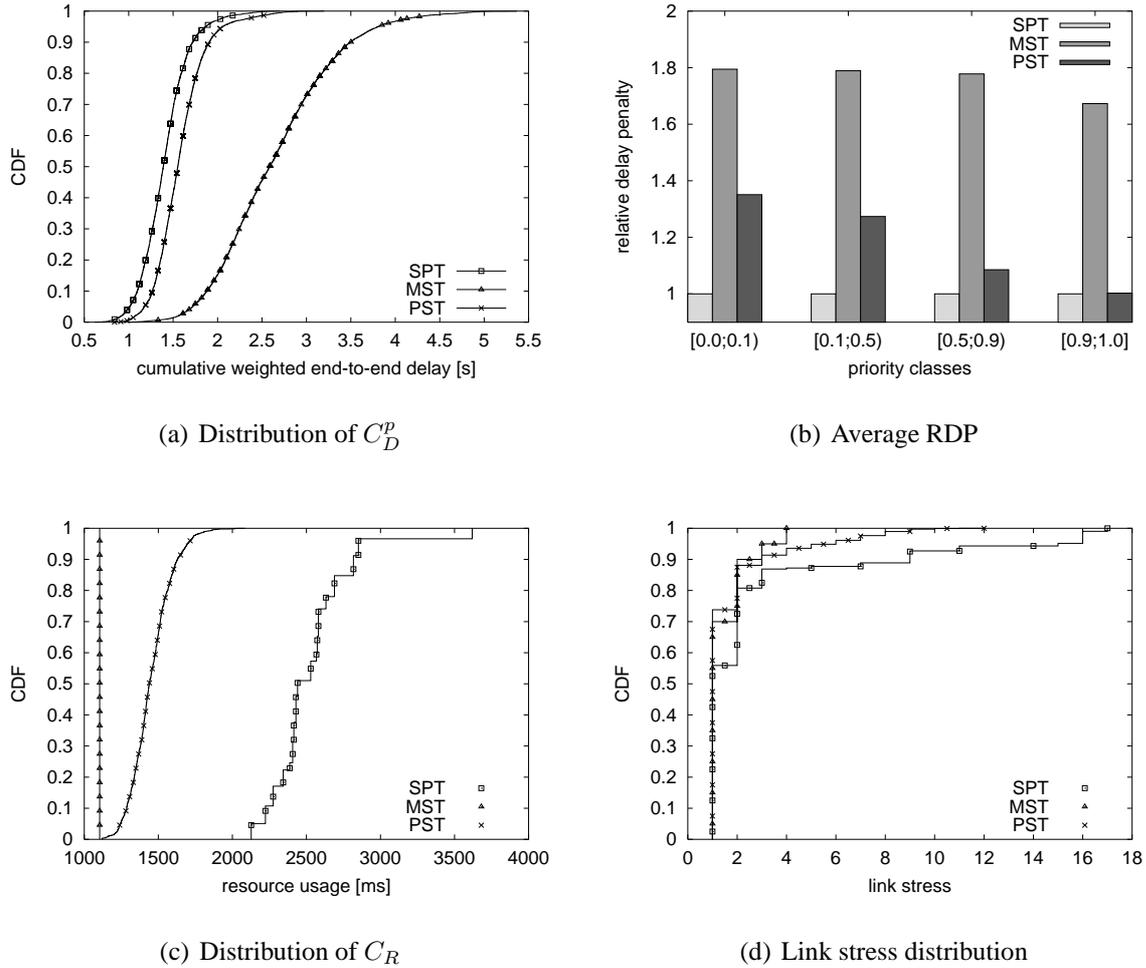


Figure 8.13: Simulation results for 18 end-systems

bandwidth consumed for outgoing network traffic is approximately the same for MSTs and PSTs. This is also indicated by the average height of 3.0 for all PSTs.

The distributions for C_D^p are depicted in Figure 8.13(a). Because of the increased complexity of the ALM trees (with up to 8 hops on paths of the PST), the difference in C_D^p between SPT and PST is larger (1,721 ms to 1,906 ms at 90%). However, the PST achieves a good optimization of the latency from the source node to receivers with a high priority when compared to the values of C_D^p for the MST algorithm (3,496 ms at 90%). The receiver-specific end-to-end delays resulted in the following 99% confidence intervals: SPT [148.4; 149.2], MST [281.5; 283.8], and PST [175.2; 176.5]. Again, the PST algorithm comes close to the optimum values of SPTs. The optimization of end-to-end delays becomes also visible in the average RDP values for nodes with different priority classes (see Figure 8.13(b)): For the PST algorithm, the RDP decreases from 1.35 to 1.002, which is close to the optimum. This is a significant improvement when compared to MSTs, even for receivers in the lowest priority class.

At the same time, priority-based minimum spanning trees cause a higher network load than MSTs as can be seen in Figure 8.13(c). It shows the resource usage distributions for the three tree-building algorithms: 90% of all PSTs have a resource usage that is up to 50% higher than C_R of the MSTs. But shortest path trees have a resource usage that is by far larger. As in the first simulation scenario, the MST algorithm generates the lowest link stress, with 90% of all distribution trees having a link stress of at most 2 and a maximum stress of 4 (see Figure 8.13(d)). The values for the PST algorithm are only slightly larger with 90% of all multicast trees having a link stress of at most 3 and a maximum link stress of 12. In comparison, the link stress of the SPT trees has a value of 9 at 90%, and the maximum link stress is 17.

Summing up, the simulation results show that the PST algorithm optimizes the end-to-end delay for receivers for which the sender has a high application-level priority. Even delays for end-systems with a lower priority are better in most cases than those achieved with multicast trees built by the MST algorithm. At the same time, the increase in network load is kept at a tolerable level.

8.6.4 Introducing Uncertainty

All simulation results discussed above are calculated under the condition that the application always has full knowledge about the actual end-to-end delays. In a real network, delays fluctuate (depending on router load), and measurements give approximations only. Thus, simulations for the PST algorithm were also conducted when the measured delays differ from the real values up to a certain percentage δ : For the calculation of $w'(e_{ij})$ as defined in Equation 8.9 a random value for $w(e_{ij})$ is picked from $[w(e_{ij}) - \delta w(e_{ij}), w(e_{ij}) + \delta w(e_{ij})]$. The simulation setup is identical to the one for 18 end-systems, as described above. The simulation results presented in the following are determined for the SPT, MST and PST algorithms under the condition that the real delays are known, and for the PST algorithm when δ is either 0.2 or 0.5.

For $\delta = 0.2$, the cumulative end-to-end delay of the PST algorithm C_D^p degrades only by 128 ms at 90% when compared to the value for $\delta = 0$, and by 637 ms for $\delta = 0.5$ (see Figure 8.14(a)). Even for this high level of uncertainty, the C_D^p of the PST algorithm is still superior to the one of the MST algorithm. The relative delay penalty RDP is also mostly unaffected by smaller measurement errors and increases only slightly from 1.002 to 1.04 for $\delta = 0.2$ in the highest priority class. A maximum error of 0.5 has more effect on RDP , which rises to 1.23.

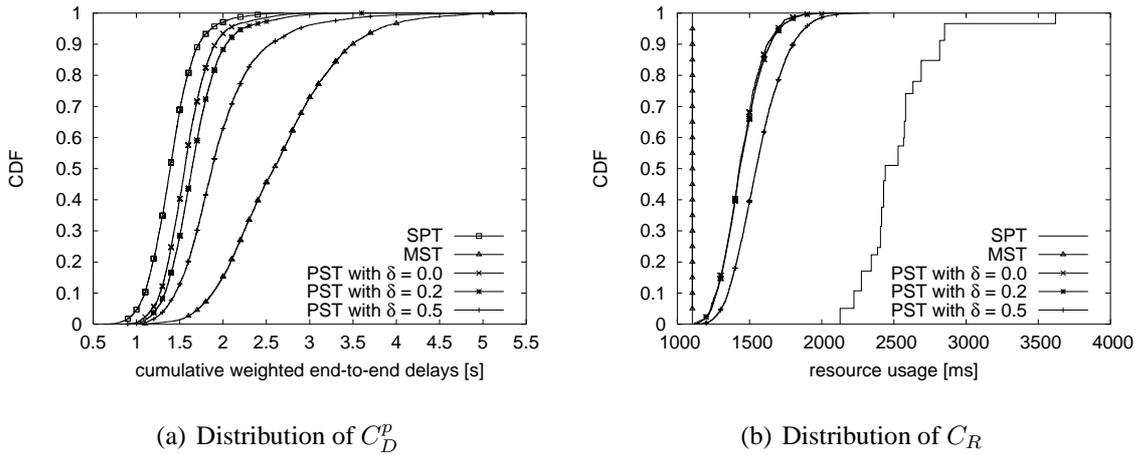


Figure 8.14: Simulation results for scenario with uncertain delays

As depicted in Figure 8.14(b), the resource usage distribution for $\delta = 0.2$ is almost identical to the one for $\delta = 0$. For $\delta = 0.5$, C_R degrades only slightly. These simulation results indicate that the PST algorithm is fairly robust against inaccurate knowledge of delays.

8.6.5 Comparison of Simulation Results

In the following, the simulation results of the PST algorithm are compared with the results of some of the ALM routing protocols discussed in Section 8.4. Even though the respective simulation scenarios differ with respect to the underlying network topologies, the physical link latencies, the number of routers and end-systems, and the application data distributed via the ALM trees, the simulation results allow a coarse assessment.

Narada is a representative of the tree-over-mesh approaches, and simulation results are given in [30]. Chu et al. measure the relative delay penalty in the form of a *90-percentile value* RDP_{90} . RDP_{90} is defined as that value of RDP where the cumulative distribution function CDF reaches 90% (i.e., 90% of all measured values for RDP are lower than RDP_{90}). For a group of 18 end-systems and depending on the simulation's setting, RDP_{90} of Narada lies between 2.1 and 2.5 [30]. This is almost identical to RDP_{90} for the MST algorithm determined above. In comparison, RDP_{90} for the PST algorithm is much smaller with a value of 1.5 (not shown in Figure 8.13(b)). When considering application priorities, the advantage of the PST algorithm becomes even clearer (see Figure 8.13(b)). The simulation results for link stress and resource usage of Narada are also similar to the results for the MST algorithm as discussed above.

HMTP builds a shared distribution tree by parent selection, and in [276] Zhang et al. give simulation results for different scenarios. For a small multicast session with approximately

20 end-systems, the value of RDP_{90} is 3.5, which is much higher than for Narada or the PST algorithm. The link stress is given only for a group of 100 members and is therefore not comparable to the other simulation results.

TAG constructs sender-specific trees on the basis of information about the network topology. For small multicast groups, this results in an average RDP of 1.25 [144]. In comparison, the average RDP achieved by the PST algorithm is slightly lower with 1.16. For application instances with a high priority, an RDP close to the optimum of 1 can be realized with PSTs.

The main goal of routing algorithms for large-scale content distribution networks such as Bayeux and CAN is to limit the application's management overhead and to achieve high scalability with respect to the number of participating nodes. The RDP_{90} for a group of 4,096 is 4.25 for Bayeux, while the high link stress induced is comparable to the one of an SPT [277]. Combining such a scheme with some information about the network topology as proposed in [207] cuts the average RDP to 2.1 for groups with 100 members and to 3.2 for 4,096 members.

Analyzing the simulation results of existing approaches shows that the PST routing algorithm is able to achieve a good performance for all session members in terms of end-to-end delays and usage of network resources. For receivers with a high application-level priority, end-to-end delays close to the optimum can be realized.

8.7 The PST Protocol

The simulation analysis indicates that the PST routing algorithm is well-suited for delay-sensitive applications where priorities are important. In the following, the PST Protocol (PSTP) is presented, which is based on the PST algorithm. PSTP was developed in the course of this thesis, and an early version was described in [8]. Again, we concentrate on the routing aspects when designing PSTP. The protocol has to handle the following tasks: (1) Determine the distribution trees according to the PST algorithm, (2) transport application data along those trees, (3) gather the information necessary for building PSTs, and (4) integrate joining session members and manage leaving members.

Aside from routing, there exist a number of functions that could be integrated into PSTP and that would be useful for distributed interactive applications. In particular, the transport of data is unreliable, and there is no specific ordering of packets. Furthermore, the protocol does not have any mechanisms for flow or congestion control. These functions could also be designed to take application-level priorities into account. For example, the reliable delivery of data packets with a high priority could be ensured by FEC (also see Section 7.2.1). And data with a low priority could be dropped first in case a node does not have sufficient bandwidth or

```

Functions of class PSTP_Socket:

void join(IPAddress member, int port)
void leave()
void send(byte[] data)
void send(byte[] data, IPAddress receiver)
void setPriority(IPAddress member, double priority)
MemberList getMembers()

Functions of interface PSTP_Feedback:

void sessionStatus(int status)
void receive(byte[] data, IPAddress sender)
void memberJoin(IPAddress member)
void memberLeave(IPAddress member)

```

Figure 8.15: PST protocol library API

when the network is congested. The realization of such functionality remains an issue for future work.

8.7.1 Application Interface

The PST protocol is implemented as a Java library [8]. The class `PSTP_Socket` depicted in Figure 8.15 provides a socket-like interface to the protocol's functionality and can be used easily by any distributed interactive application. An application instance joins a certain ALM session by providing the unicast IP address and port number of a bootstrap node that already is a session member. How such an address can be obtained is not considered here. One possibility are well-known nodes that maintain a session directory with a (partial) list of current members [89, 131].

Data can be sent either to the whole multicast group or to a single session member that is identified by its IP address. The application-level priority for a certain receiver can be changed by the application anytime. In case the application does not assign a priority for one or more members, a default priority of 0 is used in order to save network resources. By calling the `getMembers` function, the application is provided with information about all members of the multicast group (e.g., current priority, end-to-end delays for unicast and for current PST, etc.). Data received by a session member is provided to the application via the `PSTP_Feedback` interface shown in Figure 8.15. The application is also notified when errors occur (e.g., the bootstrap node is unreachable), or when members join or leave the session.

Selecting appropriate priorities $p(i)$ for the different receivers depends on the number of users and their behavior, on the available resources, and on the application itself. If resources are

limited or if the multicast group is rather large, the priority distribution should be determined such that low priorities are much more likely than high priorities. In this case, the resource usage of the priority-based distribution tree is expected to approximate the usage of an MST. For the multi-player game presented in Section 2.4.2, priorities are selected depending on the proximity of players on the field and the range of the laser beam (see Section 8.6.1). In case of the mlb, a similar approach could be used so that high priorities are assigned between group members currently interacting with each other or working on the same area of a slide. Alternatively, the consistency control mechanism of the application could be supported by choosing priorities accordingly. If the application employs local lag to reduce the probability for short-term inconsistencies, receivers with a unicast delay close to the local lag could be assigned high priorities, while the delay optimization for receivers with a larger difference between unicast delay local lag is less important. In the case of Instant Collaboration, high priorities could be assigned to receivers that actively modify the content of an activity, mid-range priorities to participants that observe the activity's content passively, and low priorities to session members that have not joined the activity.

It would also be possible to use different priorities on the basis of ADU types: States and events are delivered with a high priority since they change the application's shared state, while cues are sent with a low priority (see Sections 2.2 and 7.3.1). This distinction could be fine-grained and depend on the type of event (e.g., high priority for delete events) or the targeted object (e.g., high priority for events targeting an mlb page itself).

8.7.2 Basic Protocol Functionality

In the following, basic functions of the PST protocol are presented, and it is discussed which transport protocol to employ, how members join or leave a session, and how the end-to-end delays $w(e_{ij})$ between end-systems are determined.

PSTP establishes unicast connections between two nodes with UDP [196]. While this implies that either PSTP or some other application-level protocol has to implement mechanisms for reliable data delivery, source ordering and flow and congestion control, it also allows PSTP to quickly set up and terminate connections. A low overhead for connection management is particularly important since a distribution tree might change on a per-packet basis. Thus, TCP with its three-way handshake and its four-way close [198] is not well-suited in this scenario. In addition, using UDP as a transport protocol would allow PSTP to distribute data with IP multicast when that is available (e.g., in a LAN). Employing scoped IP multicast is common practice with ALM protocols [70, 27, 276]. Its integration into PSTP remains an issue for future work.

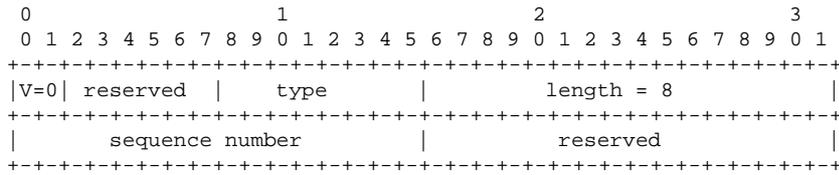


Figure 8.16: PSTP packet for types join, leave, and ping

All nodes participating in a PSTP session are identified by their IP address and port number. This increases the robustness of the protocol when compared to the alternative of using a separate and smaller participant identifier, which would require an address mapping mechanism. However, it comes at the cost of consuming more memory space and bandwidth. We denote the tuple of IP address and port number as *node identifier*. In a network environment where NAT (Network Address Translation [230]) or DHCP (Dynamic Host Configuration Protocol [52]) may manipulate the IP address of end-systems, the uniqueness of the PSTP node identifier cannot be guaranteed. This remains an issue for further investigation.

To join a session, a node *n* sends the join packet shown in Figure 8.16 directly to the bootstrap member *b*. The IP addresses and the individual port numbers of *n* and *b* are encoded in the IP and UDP headers respectively. The sequence number of the packet is used to detect out-of-bound packets, and the length field holds the total length of the packet in bytes. As a response, *b* sends a list of all known members *i*. This list contains the identifiers and the end-to-end delays $w(e_{bi})$ of these nodes *i*. The new member *n* then iteratively contacts each node *i* it discovers with a join packet. All gathered end-to-end delays form the delay matrix needed for the PST algorithm. In case a node leaves the session, a leave packet is sent to all other members. Again, an SPT is used for the distribution to ensure a quick update of the ALM group.

Besides the application-level priorities $p(i)$, the unicast delays $w(e_{ij}) \forall i, j \in V$ need to be determined in order to calculate PST distribution trees. These delays are measured by the periodic exchange of ping messages among all nodes: The sender *i* of a ping packet (see Figure 8.16) locally stores the current timestamp in $T_{ping}(j)$, together with the address of the recipient *j* and the sequence number of the ping packet. *j* replies instantaneously with a pong packet containing the sequence number of the corresponding ping packet (see Figure 8.17). As soon as *i* receives this reply, $w(e_{ij})$ can be calculated as $\frac{T_C - T_{ping}(j)}{2}$ where T_C denotes the current time. Since delays may fluctuate, the measurements for $w(e_{ij})$ are smoothed exponentially.

All members exchange lists of their local delay measurements periodically so that each node can maintain the global delay matrix. This soft state approach increases the robustness of the protocol and is also more efficient than explicit notifications when a value changes (also see Section 7.3.2). In order to save network resources, the delay lists are distributed via an MST.

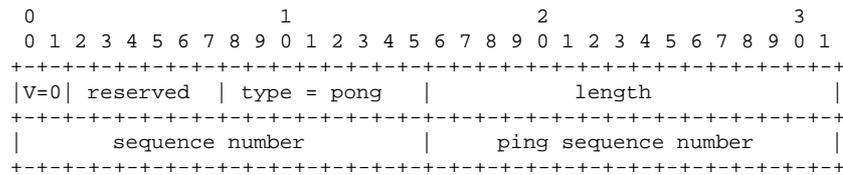


Figure 8.17: PSTP pong packet

In addition to the periodic exchange, delay lists are also provided to new session members in reply to their join message. The packet used for the delay exchange is depicted in Figure 8.19 and will be explained in the next section after discussing the routing process.

The periodic ping messages also allow to detect members that left the session due to node or network failures. In case a node did not react to a ping message for several times, it is marked as dead, and after a few more unsuccessful pings it is removed from the list of members.

8.7.3 Efficient Topology Distribution

The PST routing algorithm can be implemented either in a centralized or a distributed fashion. In the latter case, each node decides independently on which links a received packet should be forwarded. But this requires that each node maintains the matrix of all application-level priorities in addition to the matrix of the end-to-end delays. Since those priorities might change quickly, this might not be a viable solution. And each node would have to calculate the sender-specific PST for each incoming packet.

Alternatively, the routing is centralized, and a sender distributes the locally calculated PST to the other nodes. Now each node has to keep track of its local priorities and the delay matrix only, and packets are forwarded on the basis of information provided by their source. While this approach generates additional network traffic for the distribution of the ALM trees, it reduces the overhead for information management and tree calculation considerably and also saves the network traffic that would be necessary to exchange the priorities. Furthermore, inconsistent information will not result in routing loops. But in comparison to a fully distributed algorithm such as the parent selection approaches presented in Section 8.4, PSTP is less scalable with respect to the number of nodes and is therefore not well-suited for scenarios with a large number of participants.

In PSTP, the sender encodes a distribution tree in the data packets itself. As depicted in Figure 8.18, a data packet contains additional fields for the sender's identifier since the headers of UDP and IP refer to the node of the last hop only. The subsequent field gives the total number of receivers r . Thereafter comes a list of all receivers $i \in R$: For each i , the identifier and the position of the node's parent p are given. The parent position is an integer ranging from 0 (for the root) to r and points to the position of p in the node list. This information allows to

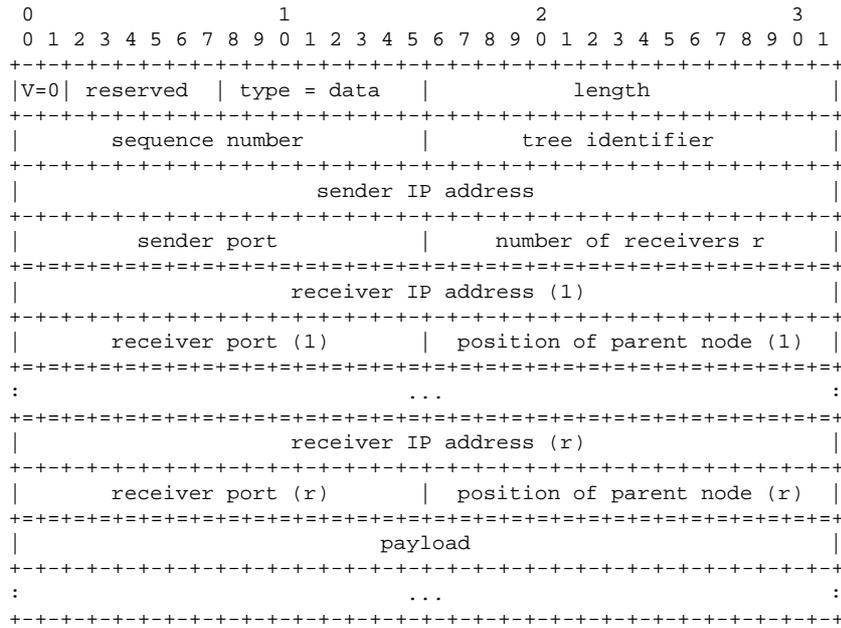


Figure 8.18: PSTP data packet

rebuild the tree: The list itself is sorted such that nodes higher up in the tree are encoded first. In this way, a node can remember its own position when traversing the list and determine which of the following nodes point to this position. These are the node’s children to whom the packet needs to be forwarded. The encoding with a list of nodes and their parents is the most efficient way to represent arbitrary tree structures [35].

When forwarding a data packet, a node *i* can create a new, smaller packet header by omitting all nodes listed before *i* since nodes do not need to know the complete tree. However, the first list entry (with position 0) denotes the original source of the packet and needs to be encoded in all packets.

Providing the distribution tree in every packet increases the routing’s robustness but is not very efficient because each node consumes 8 bytes in the packet header. It is therefore proposed to assign an identifier to each tree on the first occasion it is used (see Figure 8.18). All nodes receiving a packet with an encoded tree store their list of children together with the identifiers for the root and the tree. For subsequent packets, the sender provides the identifier of the appropriate tree only, and the receivers act on the basis of their cached routing information. In case a node receives a packet with an unknown tree identifier, it notifies the sender. The missing tree is then provided with the next data packet to be distributed over that tree.

A certain tree might need to be replaced with a new version when the end-to-end delays, the application-level priorities or the set of group members change. The replacement is performed by including the updated tree in the next data packet. Since the number of trees for a certain sender is limited by the identifier’s namespace, it may also happen that there is no identifier left to encode a new tree. In this case, the identifier of an existing tree is overwrit-

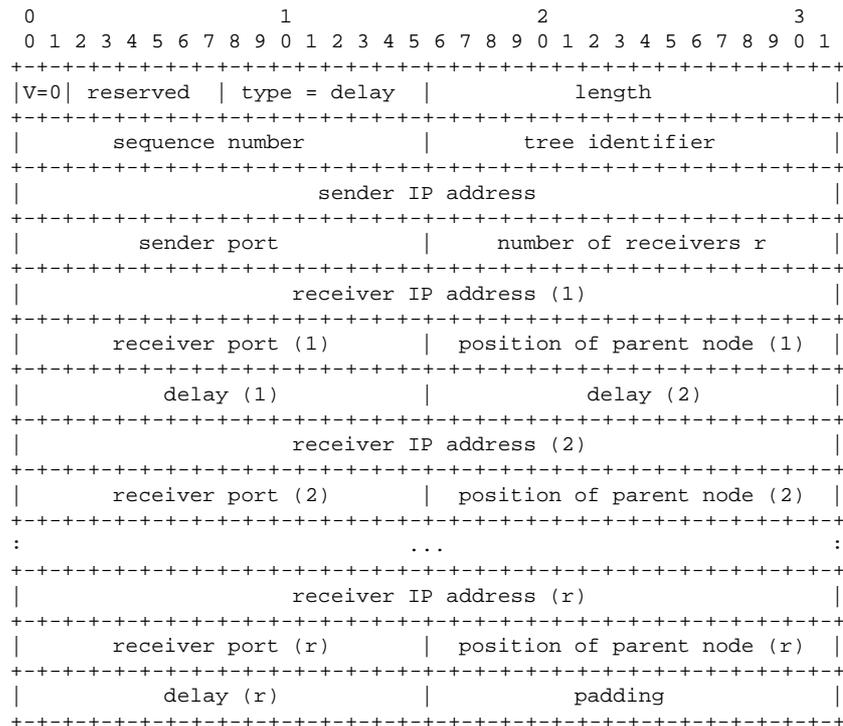


Figure 8.19: PSTP delay packet

ten, preferably one that is rarely used. In order to restrict the amount of memory that nodes need for storing topology information, the current version of PSTP allows a maximum of 256 different trees per sender.

Besides data packets, delay packets used for the exchange of the measured unicast delays carry topology information (see last section). As depicted in Figure 8.19, the delay fields are interleaved with the topology list in order to simplify the decoding. The distribution tree used for delay packets is always an MST, and shortening the node list as described above is not employed.

8.7.4 Maintenance of the Distribution Tree

The change of end-to-end delays and in particular the change of application-level priorities require that new PSTs are calculated and distributed continuously. Calculating a PST is costly in terms of processing time for the sender, and distributing a tree consumes a considerable amount of bandwidth and memory space at the inner nodes. Thus, ways to limit the overall number of distribution trees for a session need to be considered.

Two cases where an update of a PST might be omitted can be distinguished³: First, the alterations in priorities or delays have no effect on the distribution tree at all, and the newly calculated tree T' is identical to the former tree T . Ideally, this can be discovered without having to compute T' completely and comparing T' with T . Second, the changes result in a new distribution tree but the improvements in terms of the cost function defined above are marginal. In this case, the overhead that would be generated does not justify an update.

The first case is now investigated in more detail. A PST T does not change in the following cases: (1) When the cost $w(e_{ij})$ of a link that is not in T increases, (2) when the priority $p(j)$ for a receiver j connected directly to the sender increases, and (3) when the priority $p(j)$ for a receiver j that is a leaf node decreases. Furthermore, a change in receiver priorities or unicast delays may be too small to cause a tree change. An increase in link delay on a direct link between sender and receiver may cause the receiver to be connected through an indirect link (corresponding to a priority decrease). An increase in the delay of an indirect link may cause a node to be connected directly (corresponding to a priority increase). Similar considerations hold for a delay decrease on direct or indirect links. When computing a directed MST, it is possible to record for each step of the algorithm by how much the cost of a link has to increase before it is excluded from the distribution tree, or by how much the cost of a link has to decrease before it will be included in the tree. With these considerations, rebuilding the tree can be limited to the cases where the tree structure will change.

When a number of unicast delays or priorities are modified simultaneously, recomputing the whole tree is reasonable. But if only a single parameter changes, adjusting the existing tree may be less costly. Let us assume that the cost of a single link increases sufficiently to cause a change in the distribution tree. Two cases have to be distinguished: (1) $w(e_{ij})$ increases for a $i \in R$, and (2) $w(e_{sj})$ increases for the sender s . In the first case, $w'(e_{ij})$ is updated and j is connected to the rest of the tree via a less expensive link. However, the link costs for all nodes in the tree below j as well as the tree structure remain unaffected. Because links are asymmetric, it may be the case that it is now less expensive to connect i via e_{ji} , and so on. Hence, the direction of links on the path from j to s has to be reversed as long as the costs w' in the direction towards the sender are less expensive than the link costs in the opposite direction.

In the second case, when the cost of a link e_{sj} from the sender increases, this modification will also increase the costs of $w'(e_{jk}) \forall k \in R \setminus \{j\}$. For all k with $e_{jk} \in T$, it is necessary to check whether the node can be connected to the rest of the tree via a less costly link (i.e., the

³Note that some of the improvements discussed in this section are only possible because the overlay graph is fully connected and because the relative weight increase on the last hop of an indirect path is based on the weight of the link from the sender to the start of the last hop link and not on the complete path to the receiver.

	Oslo	UM1	UM2	UM3	Berkeley	ADSL
Oslo	-	19	18	18	98	50
UM1	19	-	1	0	85	40
UM2	18	1	-	0	84	40
UM3	18	0	0	-	84	45
Berkeley	98	85	84	84	-	95
ADSL	50	40	40	45	95	-

Table 8.1: Matrix of measured end-to-end delays [ms]

rest of the tree may grow “into” the region with the increased link costs). The tree parts below k will not be affected. Thus, in both cases only very limited parts of the tree will change.

The same calculations can be applied when link costs decrease. Moreover, priority changes affect the costs of all incoming links of a node but since only one of these links can be in the current distribution tree, the above statements are even valid for altered priorities.

Even when the changes of end-to-end delays or application-level priorities result in a new distribution tree T' , the improvement of T' with respect to the metrics and cost functions discussed in Sections 8.3 and 8.5 might be marginal when compared to the original tree T . Considering the overhead associated with the introduction of T' , it might therefore be more efficient to continue distributing data packets via T although T is no longer optimal.

In order to evaluate the potential of such an approach, a simple heuristic was defined and applied to the simulation scenario of the online game with 18 participants (see Section 8.6.3): A new PST is generated only when at least one priority $p(i)$ changes by more than a certain threshold γ_p when compared to the value of $p(i)$ that was valid the last time a new tree was introduced. Changes in the end-to-end delays are not considered here since the delays are constant in the simulation scenario. For $\gamma_p = 0.01$, only 0.5% of all tree calculations can be saved with this heuristic (compared to $\gamma_p = 0$), while $\gamma_p = 0.05$ and $\gamma_p = 0.1$ lead to substantial savings of 14.6% and 32.4% respectively. At the same time, the effect of the heuristic on the *RDP* distribution is negligible, even for $\gamma_p = 0.1$ [8]. This indicates that the number of distribution trees that are actually needed in a certain scenario can be reduced significantly by an appropriate heuristic, without causing any severe performance penalties.

Another heuristic that could be employed for this purpose is based on the cost function C , as defined in Equation 8.8: The sender replaces T with the new tree T' only if $\frac{C(T')}{C(T)}$ lies below a certain threshold $\gamma_C < 1$. However, this requires the sender to calculate T' in any case while the first heuristic can be applied before the PST algorithm is executed.

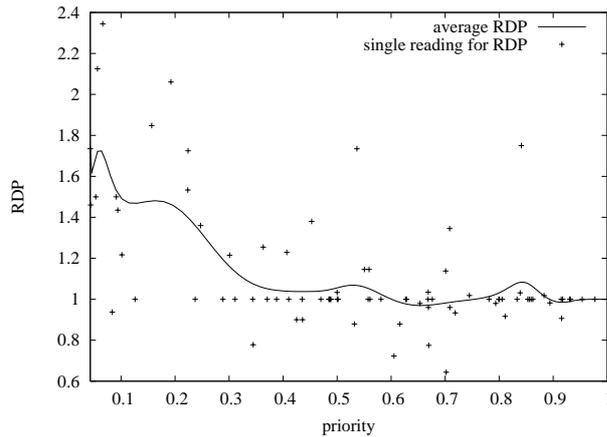


Figure 8.20: Experimental results for RDP

fanout	1	2	3	4	5
Oslo	0.64	0.32	0.04	0.00	0.00
UM1	0.56	0.31	0.13	0.01	0.00
UM2	0.50	0.22	0.25	0.04	0.00
UM3	0.19	0.35	0.24	0.14	0.08
Berkeley	0.64	0.35	0.01	0.00	0.00
ADSL	0.26	0.68	0.07	0.00	0.00

Table 8.2: PST fanout distribution

8.7.5 Experimental Results

Based on the online game scenario with six players (see Section 8.6.2), the performance of the fully implemented PST protocol was measured in an Internet environment [8]: Three nodes UM1, UM2, and UM3 are located in the same LAN at the University of Mannheim, another node is also placed in Mannheim but accesses the Internet via ADSL (which usually causes high end-to-end delays), one node joins the session from the University of Oslo in Norway, and the last node is located at the University of California in Berkeley, USA. This setting resulted in the matrix of average end-to-end delays $w(e_{ij})$ that is shown in Table 8.1. The exchange of application data and the application-level priorities $p(i)$ are determined by the same event file as in the simulations.

Figure 8.20 depicts the measured values for $RDP(i)$ depending on $p(i)$ for all packets originating at the ADSL node. As expected from the simulation results discussed above, the average $RDP(i)$ decreases with increasing $p(i)$ and is approximately 1 for the highest priorities. Single readings of $RDP(i)$ below 1 can be explained by minor fluctuations in the delays measured for the LAN end-systems: Delays among UM1, UM2 and UM3 are very small (see Table 8.1) so that slight variations of the latency's absolute value have a major impact on relative metrics such as RDP . The same explanation holds true for the high readings of RDP for high priorities as shown in Figure 8.20.

The experimental results for the fanout distribution are listed in Table 8.2. While the PSTs of the end-systems that are located within the LAN have a fanout of up to 5 (i.e., the PST is an SPT), in 99% the PSTs rooted at Berkeley have a fanout of no more than 2 since its links to the other nodes are rather costly. This proves that the PST algorithm is able to limit the usage of network resources in the experiment.

All application instances together emitted a total of 2,630 packets with application data in this scenario, which lead to an overall of 13,150 packets received by the members. The packet

headers and control messages of PSTP add up to a total of 224 kbytes received. Thus, the protocol overhead is about 17% when assuming that data packets carry 100 bytes of payload on the average. In case the average payload size is 300 bytes, this overhead is reduced to 6%.

For the experiment, the simple heuristic to reduce the number of distribution trees is employed: A new tree is calculated only when at least one value $p(i)$ changes by $\gamma_p = 0.1$, or when a delay changes by at least 5 ms (see Section 8.7.4). This heuristic saves up to 80% of all tree calculations at each member and results in an average of 20 different trees per sender, which is by far less than the theoretical maximum of 6^4 trees.

8.7.6 PSTP and RTP/I

The application-level protocol framework RTP/I for distributed interactive applications was introduced in Chapter 7. By exposing generic information about the exchanged data and the ongoing session (e.g., whether a packet is a state or an event), RTP/I allows the implementation and reuse of application-independent services such as the late-join service presented in Chapter 6. In addition, RTP/I provides basic protocol functionality such as packet fragmentation and light-weight session control to the application. The design of RTP/I is basically independent of the underlying transport and routing protocols. Currently, all applications based on RTP/I employ UDP and IP multicast [142, 154, 161, 259]. However, RTP/I could also be combined with an ALM protocol such as PSTP.

Even though RTP/I and PSTP could be operated separately following a strict layering approach, there exist quite a few interesting relationships between both protocols so that a collaboration according to the ILP/ALF approach [33] might increase the overall efficiency of the application's communication system. In the following, possibilities for the interoperation of RTP/I and PSTP and their design implications are discussed.

As can be seen from the packet definitions given in Figures 7.2 and 8.18, the packet headers of RTP/I and PSTP each include some fields with similar tasks. The sequence number field is identical for both protocols, and using a common field would save 16 bits for each packet. Note that the sequence number is also needed for other purposes such as source ordering, reliability, flow and congestion control. Other fields with the same function are the ones for identifying the source of a packet. While PSTP uses a tuple of IP address and port number with a total of 48 bits as a node identifier, a unique 32 bit participant identifier is employed by RTP/I. Both types of identifiers have their advantages and disadvantages. The node identifier is larger but does not need any additional mapping as with the participant identifier. However, IP address and port number might not be unique in a network environment with NAT or DHCP. In any case, using a common header field for identifying session members would save either 32 or 48 bit per packet. Aside from the saved network bandwidth, combining the

packet headers of PSTP and RTP/I could reduce the costs for packet processing due to saved copy operations.

RTP/I integrates a mechanism for fragmenting ADUs that do not fit into a single transport packet in order to prevent IP-level fragmentation (see Section 7.3.1). Since a PSTP header might be rather large because of an included distribution tree, either the fragmentation mechanism of RTP/I should take this into account and adjust the fragment size accordingly, or packets could be fragmented by PSTP instead. Alternatively, the topology information of PSTP could be distributed in separate packets, preferably with RTCP/I.

One important issue that a transport protocol for distributed interactive applications needs to address is the reliable delivery of data. An appropriate reliability mechanism could make use of the knowledge gained from both RTP/I and PSTP. A reliable multicast protocol could offer different levels of reliability depending on the type of data (e.g., state or cue) and the application-level priority. High-priority packets could then be sent with a FEC scheme while the loss of low-priority packets would be repaired by ARQ. Furthermore, the tree structure of PSTP could facilitate a reliable transport protocol. For instance, inner nodes of the distribution tree could cache packets for a certain amount of time so that nodes suffering from packet loss could contact their parent nodes first. In order to have a stable topology, it seems to be reasonable to use MSTs in such a design. Levine and Garcia-Luna-Aceves give an introduction to tree-based reliable multicast protocols in [147].

In case the application employs synchronized clocks, the end-to-end delays needed for the PST algorithm could be measured using the timestamp fields of RTP/I data packets. Such delay measurements could also support other protocol functions and generic services. For instance, the consistency control service presented in Chapter 4 could determine an appropriate value for the local lag depending on the maximum end-to-end delay that occurs in a session. Furthermore, the ping and delay exchange mechanisms of PSTP as described in Section 8.7.2 might be integrated into RTCP/I that already maintains important session data (see Section 7.3.1). However, the integration of the delay measurements into RTP/I would require RTP/I to be aware of the reliability mechanism employed by the application, which might influence the measurements (e.g., when using packet retransmissions). RTCP/I could also be solely responsible for the management of session members and implement the functionality of PSTP for joining and leaving members.

As already mentioned in Section 8.7.1, the application-level priorities determining the PST distribution trees might be influenced by RTP/I: First, RTP/I data packets that change the application's shared state such as events and states as well as high-priority state requests should be distributed with a high priority in order to minimize the propagation delays. In contrast, cues and low-priority states and state requests would be assigned a low application-

level priority in order to save network resources. Second, soft state information transmitted periodically by RTCP/I is preferably distributed via MSTs only. One-time announcement packets (e.g., member join) might be assigned a higher priority.

8.8 Conclusions

Distributed interactive applications allow a group of users to collaboratively change the shared state of the application. Because of their replicated architecture, all operations originating at a certain application instance need to be transported from their source to all other instances by means of an efficient group communication protocol. In the Internet, IP multicast realizes group communication with direct support from the network routers. By installing a source-specific distribution tree among the end-systems with the routers as inner nodes, the lowest possible end-to-end delays and a minimum usage of network resources can be achieved. However, the overall architecture of IP multicast is very complex, with a multitude of protocols and several administrative and technical design flaws. Thus, IP multicast is not widely deployed.

A promising alternative to IP multicast is application-level multicast where a distribution tree is constructed via unicast connections among end-systems. Router support is therefore not needed. Like IP multicast, existing ALM routing protocols seek to build distribution trees such that the propagation delay perceived by the receivers as well as the network load is minimized. However, they focus on an optimization with respect to the whole distribution tree. This might result in very high delays for some of the receivers. But there are distributed interactive applications where the fast delivery of data is more important for some session members than for others. For example, when two users are manipulating the same part of the application's state they should receive each other's operations as quickly as possible.

Thus, a novel, priority-based routing algorithm for ALM was developed in the course of this thesis. Its main contribution is that it allows an application to influence the path that a packet takes from the sender to a receiver by specifying a *priority* for each packet-receiver pair: As the priority is increased, the routing tree changes gradually from an MST to an SPT. The PST algorithm for building such a distribution tree was realized by approximating the optimal tree for a cost function that combines end-to-end delays, network resource usage and application-level priorities. Thus, the PST algorithm is a generalized tree-building algorithm that includes MSTs and SPTs as its extremes. The simulation results for a realistic online game scenario indicate that the PST algorithm constructs multicast trees with end-to-end delays that are close to the optimum for receivers with a high priority while the total network load increases only slightly when compared to an MST. It was also discussed how the PST algorithm can

be adapted so that constraints (e.g., maximum bandwidth available at the nodes) can be taken into account.

On the basis of this novel routing algorithm, the PSTP protocol was implemented, which can be employed easily by any distributed interactive application. Its main tasks are the delay measurements needed for the PST algorithm, and the distribution of data packets according to the distribution trees that are calculated at the packets' sources. An efficient encoding of PSTs was proposed where identical trees are referenced in the packet header rather than including the complete tree in each packet. The protocol overhead of PSTP was also lowered by limiting the number of trees used in a session and by limiting the number of tree calculations at the senders. PSTP was successfully tested in the Internet, confirming the simulation results. Open issues are further efficiency optimizations, the consideration of constraints, the interoperation of PSTP and RTP/I, the integration of other transport protocol functionality, and tests with other distributed interactive applications (e.g., Instant Collaboration).

Chapter 9

Conclusions and Future Work

9.1 Conclusions

The Internet has a significant impact on human-human communication and allows users that are situated at different locations to collaborate independent from space and time. On the basis of the Internet, distributed interactive applications provide a rich communication platform by letting multiple users share and modify multimedia content. Even though the variety of distributed interactive applications is large and ranges from distributed virtual environments for synchronous communication to software engineering groupware for asynchronous collaboration, these applications comprise common design principles and challenges: Each user runs a local instance of the application, and each instance maintains a copy of the shared application state. This state may change by events and with the passage of time. Events need to be distributed from their source to all other instances in order to keep all local state copies synchronized.

This basic data model allows us to address important issues of distributed interactive applications such as collaboration management, consistency control and the communication model in an application-independent way. Since most distributed interactive applications are considerably more complex than single-user applications, generic solutions for these issues are especially important in order to simplify their design, implementation, and verification.

Shared whiteboards are a prominent example for distributed interactive applications and are used for presenting and editing documents in electronic meeting scenarios. A major goal of this thesis was the development of the shared whiteboard *mlb*. In addition to its presentation functionality, the *mlb* integrates several tools to support collaboration and awareness among multiple users and can also be employed together with handheld devices. The *mlb* was the

first application with a hierarchical state to verify the generic data model, the RTP/I protocol, and the algorithms for consistency control, late-join, and session recording.

Since each instance of a distributed interactive application maintains a local copy of the shared state, synchronization mechanisms are required to keep these copies equal to a certain extent. First, it might happen that operations are received in different orders or after their scheduled execution time. In this case, a consistency control algorithm is needed, which enforces the formal criteria of causality, convergence, or correctness. Second, late-joining application instances need to be initialized with the current shared state. In this thesis, algorithms were devised that address both issues.

A generic consistency control service for discrete and continuous applications was presented, which combines the three algorithms of local lag, timewarp, and state request to achieve correctness. The processing costs of the basic timewarp algorithm were reduced by means of a round-based execution and a filtering approach for discrete applications. Moreover, it was discussed how the memory space for storing the operation history can be limited. The feasibility and the good performance of the generic consistency control service were demonstrated using the mlb, the Spaceshooter game and Instant Collaboration as examples.

To undo or redo an operation is an important feature for distributed interactive applications, which allows to cancel unintentional actions and to explore alternative states. Because of possible side-effects and dependencies within the operation history, an undo algorithm has to consider consistency issues. A straightforward undo scheme was proposed, which meets the expectations of the local participant and is based on semantically encoded undo and redo operations. This approach is compatible with our consistency control service. It was successfully implemented for the mlb.

Local lag and timewarp together establish a correct application state by serializing the operation history, and they assume that the operations executed last best reflect the state desired by the users. However, this is not necessarily the case when users issue operations that affect the same aspects of the state concurrently or in a short period of time. Here, the users might not even be aware of each other's actions, and the ordering of operations together with the resulting state are more or less random. Since the application lacks the ability to resolve such semantic conflicts, a novel visualization mechanism was introduced, which allows the users to review the operation history and to analyze semantic conflicts in past operations. This tool can also be used to explore alternative states so that the participants themselves may resolve conflicts. A prototype was integrated into the mlb and showed promising results in experiments.

Another synchronization mechanism is required when the application allows participants to join an ongoing session at any time. The late-joining application instance then needs to be

initialized with the current state. The late-join problem was thoroughly investigated, and it was demonstrated that a carelessly designed algorithm results in a high initialization delay for the late-join client, leads to high application and network loads, and might cause inconsistencies. In contrast, the late-join algorithm designed in this thesis uses additional communication groups and a flexible policy model to significantly reduce the application and network loads. The efficiency was verified in simulation studies. The late-join algorithm was also implemented as a generic service, and the different late-join policies allow to adapt the service to the needs of a specific application. Moreover, it was discussed how consistency can be achieved in late-join situations depending on the extent and distribution of initialization information.

The synchronization problems of consistency control and late-join are especially challenging for applications supporting synchronous and asynchronous collaboration, e.g., Instant Collaboration. Such applications allow to modify the state even when some session members are offline. Thus, late-join situations occur frequently, and the state copies held by the individual application instances might diverge to a considerable degree before updates can be exchanged. For this dynamic scenario, a synchronization algorithm was proposed, which seeks to quickly complete the local operation histories of the individual sites by exchanging missing operations. Consistency can then be established with the timewarp algorithm. This synchronization mechanism was implemented for Instant Collaboration and analyzed in simulation studies.

In order to implement generic services like the consistency control service or the late-join service, basic information needs to be exposed so that it becomes generally accessible. For this purpose, the application-level protocol RTP/I was introduced. RTP/I frames the network traffic of a distributed interactive application with different operation types and provides information about the shared state as well as the participating users. The implementations of the generic services presented in this thesis are based on RTP/I and can therefore be integrated easily into different applications. This was demonstrated for the mlb.

All messages originating from a certain application instance need to be distributed to all other sites by some means of group communication. Since IP multicast is not widely deployed in the Internet due to technical and administrative flaws, application-level multicast is a promising alternative where the end-systems form a multicast distribution tree on the basis of multiple unicast connections. In this thesis, a novel ALM routing algorithm was developed, which incorporates application-level knowledge into the tree-building process. The PST algorithm balances the properties of shortest path trees and minimum spanning trees by optimizing the end-to-end delays for receivers with a high application-level priority under consideration of the induced network load. These properties were demonstrated in simulation studies based on a network game and in Internet experiments with the operational PST protocol.

9.2 Scientific Contributions

The main scientific contributions of this thesis are as follows:

- The timewarp consistency control algorithm was improved significantly with a filtering mechanism and techniques for reducing the size of the operation history [260, 83].
- Two algorithms for requesting a consistent full state from remote sites were devised [260].
- The algorithms of local lag, timewarp, and state request were combined to a generic consistency control service for distributed interactive applications [260, 161].
- The problem of undoing operations for distributed interactive applications was formalized, and an undo mechanism for discrete applications was designed.
- A powerful visualization technique based on a timeline representation of the operation history was developed, which for the first time gives the user detailed feedback about conflicting actions and allows to review the evolution of the application's shared state.
- The initialization of late-joining session members was identified as a major challenge for distributed interactive applications, and a novel and generic late-join service was proposed [261, 262]. As shown in extensive simulations, this late-join algorithm is a major improvement when compared to existing approaches. Moreover, it was discussed how eventual consistency can be achieved in late-join situations.
- Using the example of Instant Collaboration, the problems of consistency control and late-join were investigated for applications with blended synchronous and asynchronous collaboration, and a novel synchronization algorithm was designed to meet the specific challenges in this domain [83].
- A multicast routing algorithm was proposed, which allows for the first time to incorporate application-level knowledge into the routing process and which is a generalization of two well-known tree-building algorithms [263]. In simulation studies and in a protocol implementation, it was proven that this algorithm performs well.
- The development of the application-level protocol RTP/I was advanced in many respects, including protocol design issues, generic services, and payload type definitions [116, 159, 253, 254, 255, 256, 257, 258].
- The state-of-the-art shared whiteboard mlb was developed, which offers many interesting features for collaborative environments [259]. Aside from Instant Collaboration

and the Spaceshooter game, the mlb was the main testbed for the algorithms and protocols mentioned above.

The realization of the applications, algorithms, and protocols discussed in this thesis was also challenging from a software engineering perspective. The implementation of distributed algorithms, network protocols, user interfaces, application logic, and simulations comprise a total of 114,000 lines of code. For the mlb, 92,000 lines of code were programmed in C++, Tcl/Tk, and Java, including 5,000 lines for the pocket mlb, 5,000 lines for the consistency control algorithms and the conflict visualization, 6,000 lines for the late-join library, and 17,000 lines for the RTP/I library. The mlb has left the status of a research prototype and is widely used in practice now. The complete program code of the mlb is available at [259] as open source under the GNU General Public License [88]. The simulation studies of the late-join algorithms were conducted in C++ and Tcl/Tk with 2,000 lines of code. The communication and synchronization module of Instant Collaboration was developed in Java with 12,000 lines of code. Finally, the simulation studies of the PST routing algorithm and the implementation of the PST protocol add to another 8,000 lines of code in C++ and Java. All software developed is executable on Linux and Windows platforms.

9.3 Future Work

Even though many important aspects of distributed interactive applications were discussed in this thesis, there remain several issues for future research.

The visualization of semantically conflicting operations as it was realized in this thesis has some limitations that need to be addressed. While browsing the operation history is a powerful technique, it also interrupts the current task of a user. In addition to the tool-tip windows, other real-time visualization mechanisms therefore need to be investigated. The visualized operation history might also be difficult to analyze when multiple conflicts occur in a short period of time, many participants are involved, or the history is large. The representation of large operation histories could be improved with user-defined filters, e.g., to show only operations targeting a certain object. In order to allow a structured handling of conflicts, it might be useful to integrate explicit mechanisms for resolving conflicts, e.g., by means of voting. All techniques proposed need to be evaluated thoroughly for different scenarios and applications, including continuous and asynchronous. Moreover, the semantic analysis of operations might also be useful for other tasks, e.g., for searching in archived sessions.

In the course of this thesis, consistency control and the handling of late-join situations proved to be especially challenging for applications that combine synchronous and asynchronous forms of collaboration. Since here the local state copies might diverge to a considerable

degree before updates can be exchanged, and semantic conflicts are likely when merging longer operation sequences, the resulting state might not meet the users' expectations. This problem could be alleviated by restricting the users' ability to modify data while being offline, by different update mechanisms (e.g., by an additional infrastructure for caching operations), or by employing tools for visualizing and resolving conflicts.

Establishing awareness about the actions of remote users is an important feature for the mlb. Aside from the awareness tools already integrated, the awareness about changes within the shared workspace could be increased by drawing the user's attention to important actions such as deleting an object, e.g., by appropriate animations as proposed by Gutwin in [98]. As a consequence, it would be less likely that such actions are overlooked. Moreover, actions outside the shared workspace could be described textually (e.g., a status message could indicate that a participant is working within a private workspace) or represented graphically (e.g., the local interaction with user interface widgets such as buttons and menus could be represented symbolically for remote participants [100]). While awareness information implicitly improves the coordination and collaboration among users, explicit mechanisms such as floor and session control would be required for sessions with many participants or in teleteaching scenarios. These mechanisms could be realized as generic services.

The functionality of the pocket mlb could also be extended in several areas: Controlling presentation animations and the ability to annotate slides would be useful features in face-to-face scenarios. Text objects could be created with the handwriting recognition provided by handheld devices. Furthermore, the human-computer interaction could be simplified by gestures.

The RTP/I protocol plays a central role for implementing the algorithms developed in this thesis in the form of generic and reusable services. Additional services on the basis of RTP/I are conceivable, and new applications might require that RTP/I is modified or extended. Furthermore, an adaptable reliability mechanism needs to be integrated. The ultimate goal is to establish RTP/I as an Internet standard.

The focus when designing the ALM protocol PSTP was on the routing functionality. Other functions that could be added under the consideration of application-level priorities are reliability as well as flow and congestion control. Moreover, the selection of appropriate priorities for different applications needs to be investigated. Another important issue is to further reduce the computational complexity and to improve scalability. One solution might be to cluster adjacent (with respect to latencies and priorities) end-systems and to construct local PSTs. Finally, it is planned to extend the PST routing algorithm so that restrictions with respect to the available resources are taken into account when calculating a multicast distribution tree.

Bibliography

- [1] Abbott, K. R. and Sarin, S. K. *Experiences with Workflow Management: Issues for the Next Generation*. In: Proc. ACM CSCW, Chapel Hill, NC, USA, pages 113–120, October 1994.
- [2] Abdel-Wahab, H., Guan, S., and Nievergelt, J. *Shared Workspaces for Group Collaboration*. In: IEEE Communications Magazine, Vol. 26, No. 11, pages 10–16, 1988.
- [3] Abowd, G. D. and Dix, A. J. *Giving Undo Attention*. In: Interacting with Computers, Vol. 4, No. 3, pages 317–342, 1992.
- [4] Ackermann, M. S. and Starr, B. *Social Activity Indicators: Interface Components for CSCW Systems*. In: Proc. ACM UIST, Pittsburgh, PA, USA, pages 159–168, November 1995.
- [5] Adams, A., Nicholas, J., and Siadak, W. *Protocol Independent Multicast-Dense Mode (PIM-DM): Protocol Specification (Revised)*. Internet Draft, IETF, draft-ietf-pim-dm-new-v2-03.txt, February 2003.
- [6] Allen, J. F. *Time and Time Again: The Many Ways to Represent Time*. In: International Journal on Intelligent Systems, Vol. 6, No. 4, pages 341–355, 1991.
- [7] ANETTE. *Applications and Network Technology for Teleteaching*. DFN–Projekt TK602-VA/T 102.1, URL <http://www.informatik.uni-mannheim.de/informatik/pi4/-projects/ANETTE/anetteIndex.html>, 2000.
- [8] Arnold, M. Application-Level Multicast-Routing unter Berücksichtigung von Anwendungsprioritäten (in German). Master’s thesis, Praktische Informatik IV, University of Mannheim, Germany, June 2003.
- [9] Bacher, C., Müller, R., Ottmann, T., and Will, M. *Authoring on the Fly: A New Way of Integrating Telepresentation and Courseware Production*. In: Proc. ICCE, Kuching, Malaysia, pages 89–96, December 1997.
- [10] Ballardie, A. *Core Based Trees (CBT Version 2) Multicast Routing - Protocol Specification*. Internet Request For Comments, IETF, RFC-2189, September 1997.
- [11] Beaudouin-Lafon, M. and Karsenty, A. *Transparency and Awareness in a Real-Time Groupware System*. In: Proc. ACM UIST, Monterey, CA, USA, pages 171–180, November 1992.

- [12] Berlage, T. *A Selective Undo Mechanism for Graphical User Interfaces Based on Command Objects*. In: ACM Transactions on Computer-Human Interaction, Vol. 1, No. 3, pages 269–294, 1994.
- [13] Blake, S., Black, D., Carlson, M., Davies, E., Wang, Z., and Weiss, W. *An Architecture for Differentiated Services*. Internet Request For Comments, IETF, RFC-2475, December 1998.
- [14] Borghoff, U. M. and Schlichter, J. H. *Computer-Supported Cooperative Work*. Springer, Berlin, Heidelberg, New York, 2000.
- [15] Braden, R., Clark, D., and Shenker, S. *Integrated Services in the Internet Architecture: An Overview*. Internet Request For Comments, IETF, RFC-1633, June 1994.
- [16] Braden, R., Zhang, L., Berson, S., Herzog, S., and Jamin, S. *Resource ReSerVation Protocol (RSVP) - Version 1 Functional Specification*. Internet Request For Comments, IETF, RFC-2205, September 1997.
- [17] Brand, O., Mahalek, W., Sturzebecher, D., and Zitterbart, M. *MACS: A Flexible and Scalable Collaborative Environment*. In: Proc. ED-MEDIA, Freiburg, Germany, June 1998.
- [18] Brand, O., Petrak, L., Sturzebecher, D., and Zitterbart, M. *Supporting Tele-Teaching: Visualization Aspects*. In: Journal of Network and Computer Applications, Special issue on support for flexible e-learning on the WWW, Vol. 23, No. 4, pages 339–355, 2000.
- [19] Byers, J., Luby, M., Mitzenmacher, M., and Rege, A. *A Digital Fountain Approach to Reliable Distribution of Bulk Data*. In: Proc. ACM SIGCOMM, Vancouver, Canada, pages 56–67, September 1998.
- [20] Cain, B., Deering, S., Kouvelas, I., Fenner, B., and Thyagarajan, A. *Internet Group Management Protocol, Version 3*. Internet Request For Comments, IETF, RFC-3376, October 2002.
- [21] Callas, J., Donnerhackle, L., Finney, H., and Thayer, R. *OpenPGP Message Format*. Internet Request For Comments, IETF, RFC-2440, November 1998.
- [22] Calvert, K. L., Doar, M. B., and Zegura, E. W. *Modeling Internet Topology*. In: IEEE Communications Magazine, Vol. 35, No. 6, pages 160–163, 1997.
- [23] Camerini, P., Fratta, L., and Maffioli, F. *A Note on Finding Optimum Branchings*. In: Networks, Vol. 9, pages 309–312, 1979.
- [24] Cayley, A. *A Theorem on Trees*. In: Quartely Journal of Pure and Applied Mathematics, Vol. 23, pages 376–378, 1889.
- [25] CBT Multimedia Technology 02/03, LS Praktische Informatik IV, University of Mannheim, Germany. URL http://www-mm.informatik.uni-mannheim.de/veranstaltungen/ws20022003/mm_uli/, 2004.

-
- [26] Chawathe, Y. Scattercast: An Architecture for Internet Broadcast Distribution as an Infrastructure Service. Ph.D. thesis, University of California, Berkeley, CA, USA, December 2000.
- [27] Chawathe, Y., McCanne, S., and Brewer, E. A. *RMX: Reliable Multicast for Heterogeneous Networks*. In: Proc. IEEE INFOCOM, Tel Aviv, Israel, pages 795–804, March 2000.
- [28] Chen, D. and Sun, C. *Undoing Any Operation in Collaborative Graphics Editing Systems*. In: Proc. ACM SIGGROUP, Boulder, CO, USA, pages 197–206, September 2001.
- [29] Chu, Y., Rao, S. G., Seshan, S., and Zhang, H. *Enabling Conferencing Applications on the Internet using an Overlay Multicast Architecture*. In: Proc. ACM SIGCOMM, San Diego, CA, USA, pages 55–67, August 2001.
- [30] Chu, Y., Rao, S. G., and Zhang, H. *A Case For End-System Multicast*. In: Proc. ACM SIGMETRICS, Santa Clara, CA, USA, pages 1–12, June 2000.
- [31] Chu, Y. J. and Liu, T. H. *On the Shortest Arborescence of a Directed Graph*. In: Science Sinica, Vol. 14, pages 1396–1400, 1965.
- [32] Clark, D. *The Design Philosophy of the DARPA Internet Protocols*. In: Proc. ACM SIGCOMM, Stanford, CA, USA, pages 106–114, August 1988.
- [33] Clark, D. and Tennenhouse, D. *Architectural Considerations for a New Generation of Protocols*. In: Proc. ACM SIGCOMM, Philadelphia, PA, USA, pages 200–208, September 1990.
- [34] Conta, A. and Deering, S. *Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification*. Internet Request For Comments, IETF, RFC-2463, December 1998.
- [35] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms* (Second Edition). MIT press, Cambridge, MA, USA, September 2001.
- [36] Cristian, F. *Understanding Fault-Tolerant Distributed Systems*. In: Communications of the ACM, Vol. 32, No. 2, pages 56–78, 1991.
- [37] Cronin, E., Filstrup, B., Kurc, A. R., and Jamin, S. *An Efficient Synchronization Mechanism for Mirrored Game Architectures*. In: Proc. NetGames, Braunschweig, Germany, pages 67–73, April 2002.
- [38] Crowcroft, J., Vicisano, L., Wang, Z., Gosh, A., Fuchs, M., Diot, C., and Turlitti, T. *RMFP: A reliable multicast framing protocol*. Internet Draft, IETF, draft-crowcroft-rmfp-02.txt, March 1998.
- [39] Crowley, T., Milazzo, P., Baker, E., Forsdick, H., and Tomlinson, R. *MMConf: An Infrastructure for Building Shared Multimedia Applications*. In: Proc. ACM CSCW, Los Angeles, CA, USA, pages 329–342, October 1990.

- [40] DeCleene, B., Bhattacharaya, S., Friedman, T., Keaton, M., Kurose, J., Rubenstein, D., and Towsley, D. *Reliable Multicast Framework*. White Paper, March 1997.
- [41] Deering, S. and Cheriton, D. *Host Groups: A Multicast Extension to the Internet Protocol*. Internet Request For Comments, IETF, RFC-966, December 1985.
- [42] Deering, S. and Hinden, R. *Internet Protocol, Version 6 (IPv6) Specification*. Internet Request For Comments, IETF, RFC-2460, December 1998.
- [43] Delgrossi, L. and Berger, L. *Internet Stream Protocol Version 2 (ST2) Protocol Specification - Version ST2+*. Internet Request For Comments, IETF, RFC-1819, August 1995.
- [44] Dermler, G. and Froitzheim, K. *JVTOS - A Reference Model for a New Multimedia Service*. In: Proc. IFIP HPN, Liege, Belgium, pages 183–197, December 1992.
- [45] DFN-Verein. URL <http://www.dfn.de>, 2004.
- [46] Dijkstra, E. *A Note on Two Problems in Connexion with Graphs*. In: Numerische Mathematik, Vol. 1, pages 269–271, 1959.
- [47] Diot, C. and Gautier, L. *A Distributed Architecture for Multiplayer Interactive Applications on the Internet*. In: IEEE Network Magazine, Vol. 13, No. 4, pages 6–15, 1999.
- [48] Diot, C., Levine, B. N., Lyles, B., Kassem, H., and Balensiefen, D. *Deployment Issues for the IP Multicast Service and Architecture*. In: IEEE Network, Vol. 14, No. 1, pages 78–88, 2000.
- [49] Dommel, H.-P. and Garcia-Luna-Aceves, J. J. *Floor Control for Multimedia Conferencing and Collaboration*. In: ACM/Springer Journal on Multimedia Systems, Vol. 5, No. 1, pages 23–38, 1997.
- [50] Dourish, P. *Using Metalevel Techniques in a Flexible Toolkit for CSCW Applications*. In: ACM Transactions on Computer-Human Interaction, Vol. 5, No. 2, pages 109–155, 1998.
- [51] Dourish, P. and Belotti, V. *Awareness and Coordination in Shared Workspaces*. In: Proc. ACM CSCW, Toronto, Ontario, Canada, pages 107–114, November 1992.
- [52] Droms, R. *Dynamic Host Configuration Protocol*. Internet Request For Comments, IETF, RFC-2131, March 1997.
- [53] Ducheneaut, N. and Bellotti, V. *E-Mail as Habitat: An Exploration of Embedded Personal Information Management*. In: ACM Interactions, Vol. 8, No. 5, pages 30–38, 2001.
- [54] Edmonds, J. *Optimum Branchings*. In: J. Research of the National Bureau of Standards, Vol. 71B, pages 233–240, 1967.

-
- [55] Edwards, W. K. *Flexible Conflict Detection and Management in Collaborative Applications*. In: Proc. ACM UIST, Banff, Alberta, Canada, pages 139–148, October 1997.
- [56] Edwards, W. K. and Mynatt, E. D. *Timewarp: Techniques for Autonomous Collaboration*. In: Proc. ACM SIGCHI, Atlanta, GA, USA, pages 218–225, March 1997.
- [57] Effelsberg, W. and Geyer, W. *Tools for Digital Lecturing - What We Have and What We Need*. In: Proc. BITE, Maastricht, Netherlands, pages 151–173, March 1998.
- [58] Ellis, C. A. and Gibbs, S. J. *Concurrency Control in Groupware Systems*. In: Proc. ACM SIGMOD, Portland, OR, USA, pages 399–407, May 1989.
- [59] Ellis, C. A., Gibbs, S. J., and Rein, G. L. *Groupware - Some Issues and Experiences*. In: Communications of the ACM, Vol. 34, No. 1, pages 38–58, 1991.
- [60] Elrod, S., Bruce, R., Gold, R., Golderg, D., Halasz, F., Janssen, W., Lee, D., McCall, K., Pedersen, E., Pier, K., Thang, J., and Welch, B. *LiveBoard: A Large Interactive Display Supporting Group Meetings, Presentations, and Remote Collaboration*. In: Proc. ACM SIGCHI, Monterey, CA, USA, pages 599–607, May 1992.
- [61] Eriksson, H. *MBone: The Multicast Backbone*. In: Communications of the ACM, Vol. 37, No. 8, pages 54–60, 1994.
- [62] Estrin, D., Farinacci, D., Helmy, A., Thaler, D., Deering, S., Handley, M., Jacobsen, V., Liu, C., Sharma, P., and Wei, L. *Protocol Independent Multicast-Sparse Mode (PIM-SM): Protocol Specification*. Internet Request For Comments, IETF, RFC-2362, June 1998.
- [63] Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L. *The Notions of Consistency and Predicate Locks in a Database System*. In: Communications of the ACM, Vol. 19, No. 11, pages 624–634, 1976.
- [64] Extensible Markup Language (XML) 1.0 (Second Edition). W3C Recommendation, available at <http://www.w3c.org/TR/REC-xml>, October 2000.
- [65] Fenner, B. and Meyer, D. *Multicast Source Discovery Protocol (MSDP)*. Internet Request For Comments, IETF, RFC-3618, October 2003.
- [66] Floyd, S. and Jacobson, V. *The Synchronization of Periodic Routing Messages*. In: IEEE/ACM Transactions on Networking, Vol. 2, No. 2, pages 122–136, 1994.
- [67] Floyd, S., Jacobson, V., Liu, C., McCanne, S., and Zhang, L. *A Reliable Multicast Framework for Light-Weight Sessions and Application Level Framing*. In: IEEE/ACM Transactions on Networking, Vol. 5, No. 6, pages 784 – 803, 1997.
- [68] Floyd, S., Jacobson, V., McCanne, S., Liu, C., and Zhang, L. *A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing*. In: Proc. ACM SIGCOMM, Cambridge, MA, USA, pages 342–356, August 1995.
- [69] Fluckiger, F. *Understanding Networked Multimedia: Applications and Technology*. Prentice Hall, London, UK, 1995.

- [70] Francis, P. *Yoid: Extending the Internet Multicast Architecture*, April 2000. Unrefereed report, available at <http://www.icir.org/yoid/docs/yoidArch.ps.gz>.
- [71] Francis, P., Jamin, S., Jin, C., Jin, Y., Raz, D., Shavitt, Y., and Zhang, L. *IDMaps: A Global Internet Host Distance Estimation Service*. In: IEEE/ACM Transactions on Networking, Vol. 5, No. 9, pages 525–540, March 2001.
- [72] FreeType. URL <http://www.freetype.org>, 2004.
- [73] Friedrich, M. Entwurf und Implementierung einer Beispielanwendung für die Synchronisation in verteilten interaktiven Anwendungen mit kontinuierlichen Zustandsänderungen (in German). Master's thesis, Praktische Informatik IV, University of Mannheim, Germany, 2001.
- [74] Fuhrmann, T. and Widmer, J. *On the Scaling of Feedback Algorithms for Very Large Multicast Groups*. In: Special Issue of Computer Communications on Integrating Multicast into the Internet, Vol. 24, No. 5-6, pages 539–547, 2001.
- [75] Garces-Erice, L., Ross, K. W., Biersack, E. W., Felber, P. A., and Urvoy-Keller, G. *Topology-Centric Look-Up Service*. In: Proc. NGC, Munich, Germany, pages 58–69, September 2003.
- [76] Garcia-Molina, H. and Spauster, A. *Ordered and Reliable Multicast Communication*. In: ACM Transactions on Computer Systems, Vol. 9, No. 3, pages 242–271, 1991.
- [77] Gautier, L. and Diot, C. *Design and Evaluation of MiMaze, a Multi-player Game on the Internet*. In: Proc. IEEE ICMCS, Austin, TX, USA, pages 233–236, June 1998.
- [78] Geyer, W. Das digital lecture board – Konzeption, Design und Entwicklung eines Whiteboards für synchrones Teleteaching (in German). Ph.D. thesis, Department for Mathematics and Computer Science, University of Mannheim, Germany, 1999.
- [79] Geyer, W. and Cheng, L.-T. *Facilitating Emerging Collaboration through Light-Weight Information Sharing*. In: Proc. ACM CSCW, New Orleans, LA, USA, pages 221–230, November 2002.
- [80] Geyer, W. and Effelsberg, W. *The Digital Lecture Board - A Teaching and Learning Tool for Remote Instruction in Higher Education*. In: Proc. ED-MEDIA, Freiburg, Germany, June 1998.
- [81] Geyer, W. and Mauve, M. *Integrating Support for Collaboration-unaware VRML Models into Cooperative Applications*. In: Proc. IEEE ICMCS, Florence, Italy, pages 655–660, June 1999.
- [82] Geyer, W., Richter, H., Fuchs, L., Frauenhofer, T., Davijavad, S., and Poltrock, S. *A Team Collaboration Space Supporting Capture and Access of Virtual Meetings*. In: Proc. ACM SIGGROUP, Boulder, CO, USA, pages 188–196, September 2001.
- [83] Geyer, W., Vogel, J., Cheng, L.-T., and Muller, M. *Supporting Activity-centric Collaboration through Peer-to-Peer Shared Objects*. In: Proc. ACM SIGGROUP, Sanibel Island, FL, USA, pages 115–124, November 2003.

-
- [84] Geyer, W., Vogel, J., and Mauve, M. *An Efficient and Flexible Late Join Algorithm for Interactive Shared Whiteboards*. In: Proc. ISCC, Antibes, France, pages 404–409, July 2000.
- [85] Geyer, W. and Weis, R. *The Design and the Security Concept of a Collaborative Whiteboard*. In: Elsevier Computer Communications, Vol. 23, No. 3, pages 233–241, 2000.
- [86] Gilbert, E. N. and Pollack, H. O. *Steiner Minimal Trees*. In: SIAM Journal on Applied Mathematics, Vol. 16, No. 1, pages 1–29, January 1968.
- [87] Ginsberg, A. and Ahuja, S. *Automating Envisionment of Virtual Meeting Room Histories*. In: Proc. ACM Multimedia, San Francisco, CA, USA, pages 65–75, November 1995.
- [88] GNU's Not Unix. URL <http://www.gnu.org>, 2004.
- [89] Gnutella and Limewire. URL <http://www.limewire.com>, 2004.
- [90] Gordon, R., Leeman, G., and Lewis, C. *Concepts and Implications of Undo for Interactive Recovery*. In: Proc. ACM Annual Conference, Denver, CO, USA, pages 150–157, October 1985.
- [91] Greenberg, S. and Marwood, D. *Real Time Groupware as a Distributed System: Concurrency Control and its Effect on the Interface*. In: Proc. ACM CSCW, Chapel Hill, NC, USA, pages 207–217, October 1994.
- [92] Greenhalgh, C., Purbrick, J., and Snowdon, D. *Inside MASSIVE-3: Flexible Support for Data Consistency and World Structuring*. In: Proc. ACM CVE, San Francisco, CA, USA, pages 119–127, September 2000.
- [93] Greif, I., editor. *Computer-Supported Cooperative Work: A Book of Readings*. Morgan Kaufmann, San Francisco, CA, USA, 1988.
- [94] Groove Networks. URL <http://www.groove.net>, 2004.
- [95] Grudin, J. *CSCW: History and Focus*. In: IEEE Computer, Vol. 27, No. 5, pages 19–26, 1994.
- [96] Grudin, J. *Groupware and Social Dynamics: Eight Challenges for Developers*. In: Communications of the ACM, Vol. 37, No. 1, pages 92–105, 1994.
- [97] Grumann, M. *Entwurf und Implementierung eines zuverlässigen Multicast-Protokolls zur Unterstützung sicherer Gruppenkommunikation in einer Teleteaching-Umgebung (in German)*. Master's thesis, Praktische Informatik IV, University of Mannheim, Germany, 1997.
- [98] Gutwin, C. *Workspace Awareness in Real-time Distributed Groupware*. Ph.D. thesis, University of Calgary, Canada, 1997.

- [99] Gutwin, C. and Greenberg, S. *Support for Group Awareness in Real-time Desktop Conferences*. In: Proc. Second New Zealand Computer Science Research Students' Conference, Hamilton, New Zealand, April 1995.
- [100] Gutwin, C. and Greenberg, S. *Design for Individuals, Design for Groups: Tradeoffs Between Power and Workspace Awareness*. In: Proc. ACM CSCW, Seattle, WA, USA, pages 207–216, November 1998.
- [101] Gutwin, C. and Greenberg, S. *Effects of Awareness Support on Groupware Usability*. In: Proc. ACM SIGCHI, Los Angeles, CA, USA, pages 511–518, April 1998.
- [102] Gutwin, C. and Greenberg, S. *The Effects of Workspace Awareness Support on the Usability of Real-Time Distributed Groupware*. In: ACM Transactions on Computer-Human Interaction, Vol. 6, No. 3, pages 243–281, 1999.
- [103] Gutwin, C. and Penner, R. *Improving Interpretation of Remote Gestures with Telepointer Traces*. In: Proc. ACM CSCW, New Orleans, LA, USA, pages 49–57, November 2002.
- [104] Hagsand, O. *Interactive Multiuser VEs in the DIVE system*. In: IEEE Multimedia, Vol. 3, No. 1, pages 30–39, 1996.
- [105] Handley, M. *Session Directories and Scalable Internet Multicast Address Allocation*. In: Proc. ACM SIGCOMM, Vancouver, Canada, pages 105–116, September 1998.
- [106] Handley, M. and Crowcroft, J. *Network Text Editor (NTE) – A Scalable Shared Text Editor for the MBone*. In: Proc. ACM SIGCOMM, Cannes, France, pages 197–208, September 1997.
- [107] Handley, M., Crowcroft, J., and Bormann, C. *The Internet Multimedia Conferencing Architecture*. Internet Draft, IETF, draft-ietf-mmusic-confarch-00.txt, February 1996.
- [108] Handley, M., Perkins, C., and Whelan, E. *Session Announcement Protocol*. Internet Request For Comments, IETF, RFC-2974, October 2000.
- [109] Hayashi, K. and Tamaru, E. *Information Management Strategies Using a Spatial-Temporal Activity Structure*. In: Proc. ACM SIGCHI, Pittsburgh, PA, USA, pages 182–183, May 1999.
- [110] Helder, D. and Jamin, S. *End-host Multicast Communication Using Switch-tree Protocols*. In: Proc. CCGRID, Berlin, Germany, pages 419–424, May 2002.
- [111] Hilt, V. *Netzwerkbasierte Aufzeichnung und Wiedergabe interaktiver Medienströme (in German)*. Ph.D. thesis, Department for Mathematics and Computer Science, University of Mannheim, Germany, June 2001.
- [112] Hilt, V. *Interactive Media on Demand (IMoD) System*. URL <http://www.informatik.uni-mannheim.de/informatik/pi4/projects/IMoD/>, 2004.
- [113] Hilt, V. and Geyer, W. *A Model for Collaborative Services in Distributed Learning Environments*. In: Proc. IDMS, Darmstadt, Germany, pages 364 – 375, September 1997.

-
- [114] Hilt, V., Geyer, W., and Effelsberg, W. *A New Paradigm for the Recording of Shared Whiteboard Streams*. In: Proc. MMCN, San Jose, CA, USA, pages 154–164, January 2000.
- [115] Hilt, V., Mauve, M., Kuhmünch, C., and Effelsberg, W. *A Generic Scheme for the Recording of Interactive Media Streams*. In: Proc. IDMS, Toulouse, France, pages 291–304, October 1999.
- [116] Hilt, V., Mauve, M., Vogel, J., and Effelsberg, W. *Interactive Media on Demand: Generic Recording and Replay of Interactive Media Streams*. In: Proc. ACM Multimedia (Technical Demonstration), Ottawa, Canada, October 2001.
- [117] Hilt, V., Schremmer, C., Kuhmünch, C., and Vogel, J. *Erzeugung und Verwendung multimedialer Teachware im synchronen und asynchronen Teleteaching (in German)*. In: WI Schwerpunktheft "Virtuelle Aus- und Weiterbildung", Vol. 43, No. 01/2001, pages 23–33, 2001.
- [118] Holbrook, H. W. and Cheriton, D. R. *IP Multicast Channels: EXPRESS Support for Large-scale Single-source Applications*. In: Proc. ACM SIGCOMM, Cambridge, MA, USA, pages 65–78, August 1999.
- [119] Holfelder, W. *Interactive Remote Recording and Playback of Multicast Videoconferences*. In: Proc. IDMS, Darmstadt, Germany, pages 450–463, September 1997.
- [120] Hudson, S. E. and Smith, I. *Techniques for Addressing Fundamental Privacy and Disruption Tradeoffs in Awareness Support Systems*. In: Proc. ACM CSCW, Cambridge, MA, USA, pages 248–257, November 1996.
- [121] Huitema, C. *The Case for Packet Level FEC*. In: Proc. PfHSN, Sophia Antipolis, France, pages 109–120, October 1996.
- [122] Huitema, C. *Routing in the Internet (2nd Edition)*. Prentice Hall, Upper Saddle River, NJ, USA, 2000.
- [123] IEEE Computer Society. *IEEE standard for information technology - protocol for distributed interactive simulations: Entity information and interaction*. IEEE Standard 1278-1993, New York, NJ, USA, 1993.
- [124] ImageMagick. URL <http://www.imagemagick.org>, 2004.
- [125] Isaacs, E. A., Morris, T., and Rodriguez, T. K. *A Forum for Supporting Interactive Presentations to Distributed Audiences*. In: Proc. ACM CSCW, Chapel Hill, NC, USA, pages 405–416, October 1994.
- [126] Jacobson, V. *A Portable Public Domain Network Whiteboard*. Xerox PARC Viewgraphs, April 1992.
- [127] Jannotti, J., Gifford, D. K., Johnson, K. L., Kaashoek, M. F., and O'Toole, J. W. *Overcast: Reliable Multicasting with an Overlay Network*. In: Proc. OSDI, San Diego, CA, USA, pages 197–202, October 2000.

- [128] Jefferson, D. R. *Virtual Time*. In: ACM Transactions on Programming Languages and Systems, Vol. 7, No. 3, pages 404–425, 1985.
- [129] Ji, P., Ge, Z., Kurose, J., and Towsley, D. *A Comparison of Hard-state and Soft-state Signaling Protocols*. In: Proc. ACM SIGCOMM, Karlsruhe, Germany, pages 251–262, August 2003.
- [130] Johanson, R., editor. *Groupware: Computer Support for Business Teams*. The Free Press - Macmillan, New York, NJ, USA, 1988.
- [131] JXTA. URL <http://www.jxta.org>, 2004.
- [132] Karp, R. M. Complexity of Computer Computations, chapter "Reducibility among combinatorial problems", pages 85–103. Plenum Press, New York, NJ, USA, 1972.
- [133] Karsenty, A., Tronche, C., and Beaudouin-Lafon, M. *GroupDesign: Shared Editing in a Heterogeneous Environment*. In: The Journal of the Usenix Association, Vol. 6, No. 2, pages 167–195, 1993.
- [134] Kasera, S. K., Kurose, J., and Towsley, D. *Scalable Reliable Multicast Using Multiple Multicast Groups*. In: Proc. ACM SIGMETRICS, Seattle, WA, USA, pages 64–74, June 1997.
- [135] Kent, C. A. and Mogul, J. C. *Fragmentation Considered Harmful*. In: Proc. ACM workshop on Frontiers in computer communication technology (ACM SIGCOMM), Stowe, VT, USA, pages 390–401, August 1987.
- [136] Kermode, R. *Scoped Hybrid Automatic Repeat reQuest with Forward Error Control (SHARQFEC)*. In: Proc. ACM SIGCOMM, Vancouver, Canada, pages 278–289, August 1998.
- [137] Kikuchi, Y., Nomura, T., Fukunaga, S., Matsui, Y., and Kimata, H. *RTP Payload Format for MPEG-4 Audio/Visual Streams*. Internet Request For Comments, IETF, RFC-3016, November 2000.
- [138] Knister, M. J. and Prakash, A. *Issues in the Design of a Toolkit for Supporting Multiple Group Editors*. In: The Journal of the Usenix Association, Vol. 6, No. 2, pages 135–166, 1993.
- [139] Kompella, V. P., Pasquale, J. C., and Polyzos, G. C. *Multicast Routing for Multimedia Communication*. In: IEEE/ACM Transactions on Networking, Vol. 1, No. 3, pages 286–292, 1993.
- [140] Kruskal, J. B. *On the Shortest Spanning Subtree of a Graph and the Travelling Salesmen Problem*. In: Proceedings of the American Mathematical Society, Vol. 7, pages 48–50, 1956.
- [141] Kuhmünch, C. *A Multicast Gateway for Dial-In Lines*. In: Proc. ECMAST, Madrid, Spain, pages 441–455, May 1998.

-
- [142] Kuhmünch, C., Fuhrmann, T., Schöppe, G., and Herrmann, D. W. *Java Teachware - The Java Remote Control Tool and its Applications*. In: Proc. EDMEDIA, Freiburg, Germany, June 1998.
- [143] Kurlander, D. and Feiner, S. *A History-Based Macro by Example System*. In: Proc. ACM UIST, Monterey, CA, USA, pages 99–106, November 1992.
- [144] Kwon, M. and Fahmy, S. *Topology-Aware Overlay Networks for Group Communication*. In: Proc. NOSSDAV, Miami, FL, USA, pages 127–136, May 2002.
- [145] Lacher, M. S., Nonnenmacher, J., and Biersack, E. W. *Performance Comparison of Centralized Versus Distributed Error Recovery for Reliable Multicast*. In: IEEE/ACM Transactions on Networking, Vol. 8, No. 2, pages 224–238, 2000.
- [146] Lamport, L. *Time, Clocks, and the Ordering of Events in a Distributed System*. In: Communications of the ACM, Vol. 21, No. 7, pages 558–565, 1978.
- [147] Levine, B. N. and Garcia-Luna-Aceves, J. J. *A Comparison of Known Classes of Reliable Multicast Protocols*. In: Proc. IEEE ICNP, Columbus, OH, USA, pages 112–121, October 1996.
- [148] Levine, B. N., Paul, S., and Garcia-Luna-Aceves, J. J. *Organizing Multicast Receivers Deterministically by Packet-Loss Correlation*. In: Proc. ACM Multimedia, Bristol, UK, pages 201–210, September 1998.
- [149] Lienhard, J. and Maass, G. *AOFwb - A New Alternative for the MBone Whiteboard wb*. In: Proc. ED-MEDIA, Freiburg, Germany, pages 391–398, June 1998.
- [150] Lin, K. C. and Schab, D. E. *The Performance Assessment of the Dead Reckoning Algorithms in DIS*. In: Simulation, Vol. 63, No. 5, pages 318–325, 1994.
- [151] Long, A. C., Landay, J. A., Rowe, L. A., and Michiels, J. *Visual Similarity of Pen Gestures*. In: Proc. ACM SIGCHI, Amsterdam, Netherlands, pages 360–367, April 2000.
- [152] Malkin, G. *RIP Version 2*. Internet Request For Comments, IETF, RFC-2453, November 1998.
- [153] Maly, K., Abdel-Wahab, H., Wild, C., Overstreet, C. M., Gupta, A., Abdel-Hamid, A., Ghanem, S., and Zhu, X. *IRI-h, A Java-Based Distance Education System: Architecture and Performance*. In: ACM Journal of Educational Resources in Computing, Vol. 1, No. 1, pages 1–15, 2001.
- [154] Mauve, M. *TeCo3D: A 3D Telecooperation Tool Based on VRML and Java*. In: Proc. MMCN, San Jose, CA, USA, pages 240–251, January 1999.
- [155] Mauve, M. *Consistency in Continuous Distributed Interactive Media*. In: Proc. ACM CSCW, Philadelphia, PA, USA, pages 181–190, December 2000.
- [156] Mauve, M. *Distributed Interactive Media*. Ph.D. thesis, Department for Mathematics and Computer Science, University of Mannheim, Germany, August 2000.

- [157] Mauve, M. and Hilt, V. *An Application Developer's Perspective on Reliable Multicast for Distributed Interactive Media*. In: ACM Computer Communication Review, Vol. 30, No. 3, pages 28–38, 2000.
- [158] Mauve, M., Hilt, V., Kuhmünc, C., and Effelsberg, W. *RTP/I - Toward a Common Application-Level Protocol for Distributed Interactive Media*. In: IEEE Transactions on Multimedia, Vol. 3, No. 1, pages 152–161, 2001.
- [159] Mauve, M., Hilt, V., Kuhmünc, C., Vogel, J., Geyer, W., and Effelsberg, W. *RTP/I: An Application-Level Real-Time Protocol for Distributed Interactive Media*. Internet Draft: draft-mauve-rtpi-00.txt, 2000.
- [160] Mauve, M., Scheele, N., and Geyer, W. *Ubiquitous Computing in Education*. In: Proc. COMCON, Crete, Greece, June 2001.
- [161] Mauve, M., Vogel, J., Hilt, V., and Effelsberg, W. *Local-lag and Timewarp: Providing Consistency for Replicated Continuous Applications*. In: IEEE Transactions on Multimedia, Vol. 6, No. 1, pages 45–57, 2004.
- [162] Mayer, E. Synchronisation in kooperativen Systemen (in German). Ph.D. thesis, Department for Mathematics and Computer Science, University of Mannheim, Germany, 1994.
- [163] McCaffrey, L. *Representing Change in Persistent Groupware Environments*. Technical report, Grouplab Report, Department of Computer Science, University of Calgary, Canada, 1998.
- [164] McCanne, S. and Jacobson, V. *vic: A flexible Framework for Packet Video*. In: Proc. ACM Multimedia, San Francisco, CA, USA, pages 511 – 523, November 1995.
- [165] Meyer, A. *Pen Computing: A Technology Overview and a Vision*. In: ACM SIGCHI Bulletin, Vol. 27, No. 3, pages 46–90, 1995.
- [166] Meyer, D. and Lothberg, P. *GLOP Addressing in 233/8*. Internet Request For Comments, IETF, RFC-2770, February 2000.
- [167] Microsoft NetMeeting. URL <http://www.microsoft.com/windows/netmeeting/>, 2004.
- [168] Microsoft Pocket PC. URL <http://www.pocketpc.com>, 2004.
- [169] Microsoft PowerPoint. URL <http://www.microsoft.com/office/powerpoint/>, 2004.
- [170] Microsoft Windows XP Tablet PC Edition. URL <http://www.tabletpc.com>, 2004.
- [171] Mills, D. L. *Network Time Protocol (Version 3) Specification, Implementation, and Analysis*. Internet Request For Comments, IETF, RFC-1305, March 1992.
- [172] Mogul, J. and Deering, S. *Path MTU Discovery*. Internet Request For Comments, IETF, RFC-1191, November 1990.

-
- [173] Morse, K. L., Bic, L., and Dillencourt, M. *Interest Management in Large-Scale Virtual Environments*. In: *Presence Teleoperators and Virtual Environments*, Vol. 9, No. 1, pages 52–68, 2000.
- [174] Moy, J. *MOSPF: Analysis and Experience*. Internet Request For Comments, IETF, RFC-1585, March 1994.
- [175] Moy, J. *OSPF Version 2*. Internet Request For Comments, IETF, RFC-2328, April 1998.
- [176] mStar. URL <http://www.cdt.luth.se/mstar/>, 2004.
- [177] Muller, M., Geyer, W., Brownholtz, B., Wilcox, E., and Millen, D. R. *One Hundred Days in an Activity-centric Collaboration Environment*. In: *Proc. ACM SIGCHI, Vienna, Austria, April 2004*.
- [178] Munson, J. and Dewan, P. *A Concurrency Control Framework for Collaborative Systems*. In: *Proc. ACM CSCW, Cambridge, MA, USA, pages 278–287, November 1996*.
- [179] Myers, B. A., Stiel, H., and Gargiulo, R. *Collaboration Using Multiple PDAs Connected to a PC*. In: *Proc. ACM CSCW, Seattle, WA, USA, pages 285–294, November 1998*.
- [180] Mynatt, E. D., Igarashi, T., Edwards, W. K., and LaMarca, A. *Flatland: New Dimensions in Office Whiteboards*. In: *Proc. ACM SIGCHI, Pittsburgh, PA, USA, pages 346–353, May 1999*.
- [181] Nakajima, A. *Telepointing Issues in Desktop Conferencing Systems*. In: *Computer Communications, Vol. 16, No. 9, pages 603–610, 1993*.
- [182] Neuwirth, C. M., Chandhok, R., Kaufer, D. S., Erion, P., Morris, J., and Miller, D. *Flexible Diff-ing in a Collaborative Writing System*. In: *Proc. ACM CSCW, Toronto, Ontario, Canada, pages 183–195, November 1992*.
- [183] Nonnenmacher, J. and Biersack, E. W. *Scalable Feedback for Large Groups*. In: *IEEE/ACM Transactions on Networking, Vol. 7, No. 3, pages 375 – 386, June 1999*.
- [184] Oberweis, A. *Zeitstrukturen für Informationssysteme (in German)*. Ph.D. thesis, Department for Mathematics and Computer Science, University of Mannheim, Germany, 1990.
- [185] OpenMash. URL <http://www.openmash.org>, 2004.
- [186] OpenOffice. URL <http://www.openoffice.org>, 2004.
- [187] Pantel, L. and Wolf, L. *On the Impact of Delay on Real-Time Multiplayer Games*. In: *Proc. NOSSDAV, Miami, FL, USA, pages 23–29, May 2002*.
- [188] Pantel, L. and Wolf, L. *On the Suitability of Dead Reckoning Schemes for Games*. In: *Proc. NetGames, Braunschweig, Germany, pages 79–84, April 2002*.

- [189] Park, K. S. and Kenyon, R. V. *Effects of Network Characteristics on Human Performance in a Collaborative Virtual Environment*. In: Proc. IEEE Virtual Reality, Houston, TX, USA, pages 104–111, March 1999.
- [190] Patrick, A. *User-Centered Design of an MBone Videoconference Polling Tool*. Report, Version 3.3, CRC, Ottawa, Canada, March 1998.
- [191] Patterson, J. F., Day, M., and Kucan, J. *Notification Servers for Synchronous Groupware*. In: Proc. ACM CSCW, Cambridge, MA, USA, pages 122–129, November 1996.
- [192] Paul, S., Sabnani, K. K., Lin, J. C., and Bhattacharyya, S. *Reliable Multicast Transport Protocol (RMTP)*. In: IEEE Journal on Selected Areas in Communications, Vol. 15, No. 3, pages 407–421, 1997.
- [193] Pedersen, E. R., McCall, K., Moran, T. P., and Halasz, F. G. *Tivoli: An Electronic Whiteboard for Informal Workgroup Meetings*. In: Proc. ACM SIGCHI, Amsterdam, Netherlands, pages 391–398, April 1993.
- [194] Pendarakis, D., Shi, S., Verma, D., and Waldvogel, M. *ALMI: An Application-Level Multicast Infrastructure*. In: Proc. USITS, San Francisco, CA, USA, pages 49–60, March 2001.
- [195] Perkins, C. and Crowcroft, J. *Notes on the Use of RTP for Shared Workspace Applications*. In: ACM Computer Communication Review, Vol. 30, No. 2, pages 35–40, 2000.
- [196] Postel, J. *User Datagram Protocol*. Internet Request For Comments, IETF, RFC-0768, August 1980.
- [197] Postel, J. *Internet Protocol*. Internet Request For Comments, IETF, RFC-0791, September 1981.
- [198] Postel, J. *Transmission Control Protocol*. Internet Request For Comments, IETF, RFC-0793, September 1981.
- [199] Prakash, A. and Knister, M. J. *Undoing Actions in Collaborative Work*. In: Proc. ACM CSCW, Toronto, Ontario, Canada, pages 273–280, November 1992.
- [200] Prakash, A. and Knister, M. J. *A Framework for Undoing Actions in Collaborative Systems*. In: ACM Transactions on Computer-Human Interaction, Vol. 4, No. 1, pages 295–330, 1994.
- [201] Prim, R. C. *Shortest Connection Networks and Some Generalizations*. In: Bell System Technology Journal, Vol. 36, pages 1389–1401, 1957.
- [202] QuickTime Generic RTP Payload Format. URL <http://www.developer.apple.com/quicktime/icefloe/dispatch26.html>, 2004.
- [203] Raman, S. and McCanne, S. *A Model, Analysis, and Protocol Framework for Soft State-based Communication*. In: Proc. ACM SIGCOMM, Cambridge, MA, USA, pages 15–25, August 1999.

-
- [204] Ramanathan, S. *Multicast Tree Generation in Networks with Asymmetric Links*. In: IEEE/ACM Transactions on Networking, Vol. 4, No. 4, pages 558–568, 1996.
- [205] Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Shenker, S. *A Scalable and Content-Addressable Network*. In: Proc. ACM SIGCOMM, San Diego, CA, USA, pages 162–171, August 2001.
- [206] Ratnasamy, S., Handley, M., Karp, R., and Shenker, S. *Application-Level Multicast using Content Addressable Networks*. In: Proc. NGC, London, UK, pages 14–29, November 2001.
- [207] Ratnasamy, S., Handley, M., Karp, R., and Shenker, S. *Topologically-Aware Overlay Construction and Server Selection*. In: Proc. IEEE INFOCOM, New York, NJ, USA, June 2002.
- [208] Ratnasamy, S. and McCanne, S. *Inference of Multicast Routing Trees and Bottleneck Bandwidths using End-to-end Measurements*. In: Proc. IEEE INFOCOM, New York, NJ, USA, pages 353–360, March 1999.
- [209] Ravindran, K. and Samdarshi, S. *A Flexible Causal Broadcast Communication Interface for Distributed Applications*. In: Journal on Parallel and Distributed Computing, Vol. 16, No. 2, pages 134–157, 1992.
- [210] RealOne Player. URL <http://www.real.com>, 2004.
- [211] Ressel, M. and Gunzenhäuser, R. *Reducing the Problems of Group Undo*. In: Proc. ACM SIGGROUP, Phoenix, AZ, USA, pages 131–139, November 1999.
- [212] Robust Audio Tool (rat). URL <http://www-mice.cs.ucl.ac.uk/multimedia/software/rat/>, 2004.
- [213] Rojas, R., Knipping, L., Raffel, U., and Friedland, G. *Elektronische Kreide: Eine Java-Multimedia-Tafel für den Präsenz- und Fernunterricht (in German)*. In: Springer Informatik Forschung und Entwicklung, , No. 16, pages 159–168, 2001.
- [214] Salama, H., Reeves, D. S., and Viniotis, Y. *The Delay-Constrained Minimum Spanning Tree Problem*. In: Proc. ISCC, Alexandria, Egypt, pages 699–793, July 1997.
- [215] Saund, E., Mahoney, J., Fleet, D., Larner, D., and Lank, E. *Perceptual Organization as a Foundation for Intelligent Sketch Editing*. In: Proc. AAAI Spring Symposium, Palo Alto, CA, USA, pages 118–125, March 2002.
- [216] Scheele, N., Mauve, M., Effelsberg, W., Wessels, A., and Fries, S. *The Interactive Lecture - A New Teaching Paradigm based on Ubiquitous Computing*. In: Proc. CSCL, Poster Session, Bergen, Norway, June 2003.
- [217] Schremmer, C. and Hilt, V. *A Systematic Approach to the Automatic Conversion of a "Live" Lecture into a Multimedia CBT Course*. In: Proc. NLT, Bern, Switzerland, pages 126–134, August 1999.
- [218] Schulzrinne, H. and Casner, S. *RTP Profile for Audio and Video Conferences with Minimal Control*. Internet Request For Comments, IETF, RFC-3551, July 2003.

- [219] Schulzrinne, H., Casner, S., Frederick, R., and Jacobson, V. *RTP: A Transport Protocol for Real-Time Applications*. Internet Request For Comments, IETF, RFC-3550, July 2003.
- [220] Shavitt, Y., Sun, X., Wool, A., and Yener, B. *Computing the Unmeasured: An Algebraic Approach to Internet Mapping*. In: Proc. IEEE INFOCOM, Anchorage, AK, USA, pages 1646–1654, April 2001.
- [221] Shirmohammadi, S., de Oliveira, J. C., and Georganas, N. D. *Applet-Based Multimedia Telecollaboration: A Network-Centric Approach*. In: IEEE Multimedia Magazine, Vol. 5, No. 2, pages 64–73, 1998.
- [222] Shirmohammadi, S., Saddik, A. E., Georganas, N. D., and Steinmetz, R. *JASMINE: A Java Tool for Multimedia Collaboration on the Internet*. In: Multimedia Tools and Applications, Vol. 19, No. 1, pages 5–28, 2003.
- [223] Shneiderman, B. *Response Time and Display Rate in Human Performance with Computers*. In: ACM Computing Surveys, Vol. 16, No. 3, pages 265–285, 1984.
- [224] Singhal, S. and Zyda, M. *Networked Virtual Environments Design and Implementation*. Addison Wesley, Upper Saddle River, NJ, USA, 1999.
- [225] SOAP Version 1.2 Part 0: Primer. W3C Recommendation, available at <http://www.w3c.org/TR/soap12-part0/>, June 2003.
- [226] SOAP Version 1.2 Part 1: Messaging Framework. W3C Recommendation, available at <http://www.w3c.org/TR/soap12-part1/>, June 2003.
- [227] SOAP Version 1.2 Part 2: Adjuncts. W3C Recommendation, available at <http://www.w3c.org/TR/soap12-part2/>, June 2003.
- [228] SOAP Version 1.2 Usage Scenarios. W3C Recommendation, available at <http://www.w3c.org/TR/xmlp-scenarios/>, July 2003.
- [229] Srinivasan, S. *Efficient Data Consistency in HLA/DIS++*. In: Proc. ACM WSC, Coronado, CA, USA, pages 946–951, December 1996.
- [230] Srisuresh, P. and Egevang, K. *Traditional IP Network Address Translator*. Internet Request For Comments, IETF, RFC-3022, January 2001.
- [231] Stefik, M., Bobrow, D., Foster, G., Lanning, S., and Tartar, D. *WYSIWIS Revised: Early Experiences with Multiuser-Interfaces*. In: ACM Transactions on Office Information Systems, Vol. 5, No. 2, pages 147–167, 1987.
- [232] Stoica, I., Morris, R., Karger, D., Kasshoek, M. F., and Balakrishnan, H. *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*. In: Proc. ACM SIGCOMM, San Diego, CA, USA, pages 149–160, August 2001.
- [233] Sun, C. *Undo Any Operation at Any Time in Group Editors*. In: Proc. ACM CSCW, Philadelphia, PA, USA, pages 191–201, December 2000.

-
- [234] Sun, C. *Undo as Concurrent Inverse in Group Editors*. In: ACM Transactions on Computer-Human Interaction, Vol. 9, No. 4, pages 309–361, 2002.
- [235] Sun, C. and Chen, D. *A Multi-version Approach to Conflict Resolution in Distributed Groupware Systems*. In: Proc. IEEE ICDCS, Taipei, Taiwan, pages 316–325, April 2000.
- [236] Sun, C. and Chen, D. *Consistency Maintenance in Real-Time Collaborative Editing Systems*. In: ACM Transactions on Computer-Human Interaction, Vol. 9, No. 1, pages 1–41, 2002.
- [237] Sun, C. and Ellis, C. *Operational Transformation in Real-Time Group Editors: Issues Algorithms, and Achievements*. In: Proc. ACM CSCW, Seattle, WA, USA, pages 59–68, November 1998.
- [238] Sun, C., Jia, X., Zhang, Y., Yang, Y., and Chen, D. *Achieving Convergence, Causality Preservation and Intention Preservation in Real-Time Cooperative Editing Systems*. In: ACM Transactions on Computer-Human Interaction, Vol. 5, No. 1, pages 63–108, 1998.
- [239] Sun, C. and Maheshwari, P. *An Efficient Distributed Single-Phase Protocol for Total and Causal Ordering of Group Operations*. In: Proc. IEEE HiPC, Trivandrum, India, pages 295–300, December 1996.
- [240] Sun, C. and Susic, R. *Optional Locking Integrated with Operational Transformation in Distributed Real-Time Group Editors*. In: Proc. ACM PODC, Atlanta, GA, USA, pages 43–52, May 1999.
- [241] Sun, C., Yang, Y., Zhang, Y., and Chen, D. *Distributed Concurrency Control in Real-Time Cooperative Editing Systems*. In: Proc. of the Asian Computing Science Conference, Singapore, pages 85–95, December 1996.
- [242] Sun, Q. and Langendörfer, H. *Efficient Multicast Routing for Delay-Sensitive Applications*. In: Proc. PROMS, Salzburg, Austria, pages 452–458, October 1995.
- [243] SunForum. URL <http://www.sun.com/desktop/products/software/sunforum>, 2004.
- [244] Tam, J. R. Change Awareness in 2D Graphical Workspaces. Master’s thesis, Department of Computer Science, University of Calgary, Alberta, Canada, 2002.
- [245] Tanenbaum, A. S. Computer Networks (4th Edition). Prentice Hall, Upper Saddle River, NJ, USA, 2002.
- [246] Tcl/Tk. URL <http://www.scripatics.com>, 2004.
- [247] Thaler, D. *Border Gateway Multicast Protocol (BGMP): Protocol Specification*. Internet Draft, IETF, draft-ietf-bgmp-spec-05.txt, June 2003.
- [248] Thaler, D., Handley, M., and Estrin, D. *The Internet Multicast Address Allocation Architecture*. Internet Request For Comments, IETF, RFC-2908, September 2000.

- [249] Tung, T. L. MediaBoard. Master's thesis, University of California, Berkeley, CA, USA, 1998.
- [250] Urvoy-Keller, G. and Biersack, E. W. *A Multicast Congestion Control Model for Overlay Networks and its Performance*. In: Proc. NGC, Boston, MA, USA, pages 141–147, October 2002.
- [251] Vaghi, I., Greenhalgh, C., and Benford, S. *Coping with Inconsistency due to Network Delays in Collaborative Virtual Environments*. In: Proc. ACM VRST, London, UK, pages 42–49, December 1999.
- [252] Vitter, J. S. *US&R: A New Framework for Redoing*. In: IEEE Software, Vol. 1, No. 4, pages 39–52, 1984.
- [253] Vogel, J. *RTP/I Payload Type Definition for Application Launch Tools*. Technical Report TR-01-018, Department for Mathematics and Computer Science, University of Mannheim, Germany, 2001.
- [254] Vogel, J. *RTP/I Payload Type Definition for Chat Tools*. Technical Report TR-01-011, Department for Mathematics and Computer Science, University of Mannheim, Germany, 2001.
- [255] Vogel, J. *RTP/I Payload Type Definition for Feedback Tools*. Technical Report TR-01-014, Department for Mathematics and Computer Science, University of Mannheim, Germany, 2001.
- [256] Vogel, J. *RTP/I Payload Type Definition for Hand-Raising Tools*. Technical Report TR-01-010, Department for Mathematics and Computer Science, University of Mannheim, Germany, 2001.
- [257] Vogel, J. *RTP/I Payload Type Definition for Shared Whiteboards*. Technical Report TR-01-005, Department for Mathematics and Computer Science, University of Mannheim, Germany, 2001.
- [258] Vogel, J. *RTP/I Payload Type Definition for Telepointers*. Technical Report TR-01-009, Department for Mathematics and Computer Science, University of Mannheim, Germany, 2001.
- [259] Vogel, J. *multimedia lecture board (mlb)*. URL <http://www.informatik.uni-mannheim.de/informatik/pi4/projects/mlb/>, 2004.
- [260] Vogel, J. and Mauve, M. *Consistency Control for Distributed Interactive Media*. In: Proc. ACM Multimedia, Ottawa, Canada, pages 221–230, October 2001.
- [261] Vogel, J., Mauve, M., Geyer, W., Hilt, V., and Kuhmüch, C. *A Generic Late Join Service for Distributed Interactive Media*. In: Proc. ACM Multimedia, Los Angeles, CA, USA, pages 259–268, November 2000.
- [262] Vogel, J., Mauve, M., Hilt, V., and Effelsberg, W. *Late Join Algorithms for Distributed Interactive Applications*. In: ACM/Springer Multimedia Systems, Vol. 9, No. 4, pages 327–336, 2003.

-
- [263] Vogel, J., Widmer, J., Farin, D., Mauve, M., and Effelsberg, W. *Priority-Based Distribution Trees for Application-Level Multicast*. In: Proc. NetGames, Redwood City, CA, USA, pages 140–149, May 2003.
- [264] 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, URL <http://www/vrml.org/Specifications/>, 1997.
- [265] Waitzmann, D., Partridge, C., and Deering, S. *Distance Vector Multicast Routing Protocol*. Internet Request For Comments, IETF, RFC-1075, November 1988.
- [266] Walling, C. Entwicklung eines generischen Signalisierungsprotokolls für interaktive Medien (in German). Master's thesis, Praktische Informatik IV, University of Mannheim, Germany, 2000.
- [267] Wang, W., Helder, D., Jamin, S., and Zhang, L. *Overlay Optimizations for End-host Multicast*. In: Proc. NGC, Boston, MA, USA, pages 154–161, October 2002.
- [268] Weis, R., Vogel, J., Effelsberg, W., Geyer, W., and Lucks, S. *How to Make a Digital Whiteboard Secure – Using JAVA-Cards for Multimedia Applications*. In: Proc. IDMS, Enschede, Netherlands, pages 185–198, October 2000.
- [269] Whittaker, S. and Sidner, C. *Email Overload: Exploring Personal Information Management of Email*. In: Proc. ACM SIGCHI, Vancouver, BC, Canada, pages 276–283, April 1996.
- [270] Widmer, J. and Fuhrmann, T. *Extremum Feedback for Very Large Multicast Groups*. In: Proc. NGC, London, UK, pages 56–75, 2001.
- [271] Widmer, J. and Handley, M. *Extending Equation-based Congestion Control to Multicast Applications*. In: Proc. ACM SIGCOMM, San Diego, CA, USA, pages 275–286, August 2001.
- [272] Widmer, J., Mauve, M., and Damm, J. P. *Probabilistic Congestion Control for Non-Adaptable Flows*. In: Proc. NOSSDAV, Miami, FL, USA, pages 13–21, May 2002.
- [273] Wittmann, R. and Zitterbart, M. *Multicast Communication: Protocols, Programming, and Applications*. Morgan Kaufmann, San Francisco, CA, USA, 2000.
- [274] Wolf, L. C., Griwodz, C., and Steinmetz, R. *Multimedia Communication*. In: Proceedings of the IEEE, Vol. 85, No. 12, pages 1915–1933, 1997.
- [275] Yahoo! Messenger. URL <http://messenger.yahoo.net>, 2004.
- [276] Zhang, B., Jamin, S., and Zhang, L. *Host Multicast: A Framework for Delivering Multicast to End Users*. In: Proc. IEEE INFOCOM, New York, NJ, USA, June 2002.
- [277] Zhuang, S. Q., Zhao, B. Y., Joseph, A. D., Katz, R., and Kubiawicz, J. *Bayeux: An Architecture for Scalable and Fault-tolerant Wide-area Data Dissemination*. In: Proc. NOSSDAV, Port Jefferson, NY, USA, pages 11–20, June 2001.