INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A Framework for Reliable Multicast in the Internet

Michael Fuchs, Christophe Diot, Thierry Turletti and Markus Hofmann

N° ????

February 1998

THÈME 1

Rapport de recherche

ISSN 0249-6399

A Framework for Reliable Multicast in the Internet

Michael Fuchs, Christophe Diot, Thierry Turletti and Markus Hofmann

Thème 1 — Réseaux et systèmes
Projet RODEO

Rapport de recherche n° ???? — February 1998 — 117 pages

Abstract: This document presents the *Reliable Multicast Framing Protocol* (RMFP). RMFP is an ALF-based framework for protocols that can be integrated as *protocol profiles.* In this report two profiles are described: One for *Scalable Reliable Multicast* (SRM) and one for the *Local Group Concept* (LGC).

Additionally to the specifications of RMFP and the profiles the report describes the design and implementation of an object-oriented RMFP link library. The implementation contains an API and a SRM (Scalable Reliable Multicast) profile. Other profiles are planned to follow.

The key features of this implementation are the unified API to all protocol profiles implemented and a class interface to include implementations of new protocol profiles in an easy manner.

Key-words: ALF, Reliable Multicast, LGC, RMFP, SRM.

This report has been published previously as diploma thesis of Michael Fuchs at the University of Karlsruhe, Germany. Markus Hofmann has been the advisor for this work in Karlsruhe.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, B.P. 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : 04 92 38 77 77 - International : +33 4 92 38 77 77 — Fax : 04 92 38 77 65 - International : +33 4 92 38 77 65

RMFP: Encapsulation de protocoles de communication fiable pour l'Internet

Résumé : Ce rapport décrit le protocole RMFP (*Reliable Multicast Framing Protocol*). RMFP est un format d'encapsulation générique de protocoles de transmission multipoint fiable de données pour l'Internet. RMFP suit les principes ALF; il permet d'encapsuler plusieurs protocoles par le biais de profils spécifiques. Ce rapport décrit en plus de RMFP, deux profils particuliers: un profil pour le protocole SRM (*Scalable Reliable Multicast*) et un autre pour le protocole LGC (*Local Group Concept*).

A la suite de ces spécifications, ce rapport présente l'étude et la mise en œuvre d'une bibliothèque RMFP orienté-objet en C++. Cette implémentation contient l'API RMFP et un profil pour SRM; l'ajoût de profils supplémentaires est prévu par la suite. Le principal intérêt de cette bibliothèque RMFP est une interface de programmation unifiée pour n'importe quel protocole de transmission fiable de données facilitant l'ajoût de nouveaux profils de protocoles.

Mots-clés : ALF, Multicast Fiable, LGC, RMFP, SRM

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Task definition . . . 2

1.3 Outline . . . 2

2 Concepts **3**

2.1 IP Multicast . . . 3

2.2 Reliable Multicast . . . 4

2.3 Application Level Framing . . . 5

2.3.1 Features of ALF . . . 6

2.3.2 ADU Names . . . 7

2.4 Protocol Templates . . . 8

2.5 Protocol Frameworks . . . 8

3 RMFP **9**

3.1 Overview . . . 9

3.2 Mechanisms . . . 10

3.2.1 Error Control . . . 10

3.2.2 Hierarchical Naming with Objects . . . 13

3.2.3 Late-joining Receivers . . . 15

3.2.4 Automatic Profile Configuration . . . 16

3.2.5 External Modules . . . 17

3.3 The RMFP Specification . . . 17

3.3.1 General Aspects . . . 18

3.3.2 RMFP ADU Format . . . 19

3.3.3 RMFP Control Packet Format . . . 21

3.3.4 RMFP Session Packet Format . . . 24

4 The SRM protocol **27**

4.1 Mechanisms . . . 28

4.1.1 Loss Detection . . . 28

4.1.2 Feedback Suppression . . . 28
4.1.3 Repair Suppression . . . 29
4.1.4 Distance Estimation . . . 29
4.2 The SRM Profile . . . 30
4.2.1 SRM over RMFP . . . 30
4.2.2 The Packet Formats . . . 30

5 The Local Group Concept 37
5.1 Overview . . . 37
5.2 LGC as Protocol Profile . . . 38
5.2.1 Mechanisms . . . 39
5.2.2 Flows and Addresses . . . 42
5.2.3 The Data Flow . . . 43
5.2.4 The Control Flow . . . 43

6 Implementation 51
6.1 Overview . . . 51
6.2 Environment . . . 52
6.2.1 Interoperability . . . 52
6.2.2 Installation of the Library . . . 52
6.3 Design Guidelines . . . 53
6.3.1 Thread of Control . . . 53
6.3.2 The Minimal-copy Architecture . . . 54
6.3.3 Flow and Congestion Control . . . 54
6.4 The RMFP Library's Design . . . 55
6.4.1 The Classes . . . 56
6.5 The API . . . 60
6.5.1 Overview . . . 60
6.5.2 Upcalls . . . 61
6.5.3 Synchronization . . . 62
6.5.4 The Class Structure . . . 63
6.5.5 Upcall Methods of the `rmfpApplication` Class . . . 67
6.6 Example Application . . . 70
6.6.1 The Header File . . . 71
6.6.2 The Implementation File . . . 74

7 Evaluation 89
7.1 Measurements . . . 89
7.1.1 The Test Environment . . . 89
7.1.2 The Performance Tests . . . 90
7.1.3 The Tests with Late-joining Receivers . . . 93
7.1.4 Interpretation of the Results . . . 95
7.2 Aspects of ALF . . . 97

7.3 Aspects of the Implementation 98
7.3.1 Minimal-copy Architecture 98
7.3.2 Event Handling 100
7.4 The ADU Naming Concept 101
7.4.1 Semantic Reliability 102
7.5 Using Objects for semantic Reliability 103
7.5.1 Loss Detection 103
7.5.2 Parallel Transmission of Objects 104
7.5.3 Evaluation of the additional Overhead 104
7.5.4 Changes of RMFP and the Protocol Profiles 105
7.6 Related Work 106
7.6.1 RMF 106
7.6.2 The Generalized Data Naming Approach 107

8 Summary and Future Work **109**

A Glossary **111**

List of Tables

7.1 The systems used for the tests. 90
7.2 Test with one receiver. 91
7.3 Test with two receivers. 91
7.4 Test with three receivers. 92
7.5 Test with four receivers. 92
7.6 Late join of a single receiver. 93
7.7 Late-join of two receivers. 94
7.8 Late-join of two receivers. 94
7.9 Late-join of two receivers. 94

List of Figures

2.1 The IP-multicast address 3

3.1 The RMFP data packet (ADU) 19
3.2 The RMFP sender report packet 21
3.3 The RMFP receiver report packet 23
3.4 The RMFP session packet 25

4.1 The common SRM packet header 31
4.2 The SRM heartbeat control packet 32
4.3 The SRM NACK packet for scattered sequence numbers 32
4.4 The SRM NACK packet for spans 33
4.5 The SRM timestamp query packet 33
4.6 The SRM timestamp reply packet 34

5.1 The group hierarchy in LGC 38
5.2 The common control header 44
5.3 The free subpacket 45
5.4 The repair subpacket with span 46
5.5 The repair packet with a list of sequence numbers 46
5.6 The echo request packet 47
5.7 The echo reply packet 47
5.8 The diag packet 48
5.9 The acknowledgment packet 49

6.1 High-level structure 55

Chapter 1

Introduction

1.1 Motivation

In the last few years, the Internet has changed from a pure scientific network to the basis of the data communication in every-day life. The number of users grows still exponentially and has already reached the order of magnitude of tens of millions. Reasons for that development can be seen in the enhancement of both the underlying technology and user friendly interfaces like the World Wide Web.

The added spectrum and number of users introduce also new forms of communication into the Internet: Communication not just between two peers, but true group communication. Examples are the "broadcast" of radio and tv programs. Information providers send data automatically to all their clients instead of serving requests individually. The decreasing cost of bandwidth also makes audio and video conferences on the Internet economic even for private use.

The foundation for the group communication in the Internet is the IP-multicast service [8]. However, this service provides no reliability for the data transmission. This may be tolerable for some real-time applications like video and audio transmission, but other services like information distribution require a guaranteed delivery of data.

Today there are several protocols for reliable multicast transmission available; however, they differ in the service they provide, and it seems unlikely, that a single protocol can fulfill all the requirements of different applications.

This leads to the idea of protocol frameworks that integrate several protocols to provide adequate services for all applications. The *Reliable Multicast Framing Protocol* (RMFP) is such a framework, and its enhancement was the subject of this diploma thesis.

1.2 Task definition

The goal of this thesis has been the enhancement of RMFP. A theoretical analysis of basic concepts, like data naming, should be done. Specifications, so called *profiles*, for the integration of *Scalable Reliable Multicast* (SRM) and the *Local Group Concept* (LGC) into the RMFP framework should be enhanced and developed respectively. Furthermore, an implementation of the RMFP framework should demonstrate the practical feasibility of RMFP.

The RMFP framework has been implemented as a link library together with the SRM profile. So far, the library runs on the UNIX operating systems Solaris 2.5 and 2.6 and Digital Unix 4.0. Ports to other versions of UNIX should be fairly simple and are planned for the future.

The experience of the implementation was used to define more changes on RMFP, that will be part of the next version of RMFP.

1.3 Outline

In the second chapter basic concepts necessary to understand this work are presented. Described are IP-Multicast, reliable multicast and the concepts of protocol frameworks and protocol templates.

Chapter three contains the specification of RMFP. In the first part of this chapter, concepts like the hierarchical naming and the protocol mechanisms are discussed, and the rest of the chapter contains the detailed packet formats.

The next two chapters contain the specifications of the SRM and LGC profiles. Both chapters start with a small introduction to the protocols and continue with the specification of the mechanisms and the packet formats.

Chapter six presents the implementation of the RMFP library. Starting with the design and continuing with the specification of the API, this chapter ends with the detailed discussion of a small example application using the library.

The measurements and evaluation of the specifications and the implementation in chapter seven lead to suggestions for improvement and a new approach for data naming in transport protocols.

Finally chapter eight sums up the work and gives an outlook to future work.

Chapter 2

Concepts

2.1 IP Multicast

The IP-Multicast service has been introduced by Deering in 1989 [8]. According to the philosophy of the unicast IP service, the IP-Multicast service has a best-effort delivery model, i.e. it provides no reliability guarantees.

IP-Multicast uses a group-based addressing scheme. Every group is identified with an IP-multicast address. These addresses cover a subspace of all IP addresses. In the IP terminology, the address space between 224.0.0.0 and 239.255.255.255 are multicast addresses (so-called *class D* addresses, fig. 2.1).

The senders and receivers of a multicast group have a very loose relationship. Indeed, only the known group address defines this relationship. A sender can transmit packets to a multicast group in the same way as it sends unicast packets. The only difference is the use of a multicast address that identifies the group.

To receive multicast packets, a receiver has to *join* a multicast group explicitly. All packets send to that group are then delivered to the receiver, regardless of the sender.

This mechanism provides no possibility for the sender(s) to control the group membership. The IP-multicast service does not even provide the addresses or the number of the receivers in the group. The membership control and security issues are matter of upper layers.

The IP-multicast group addressing leads to a problem, when a new multicast group is to be formed. The standard mechanism today is to choose a multicast address randomly.

```
0             8              16             24           31
+---------------------------------------------------------+
|1|1|1|0|                                                 |
+---------------------------------------------------------+
```

Figure 2.1: The IP-multicast address

Since the usage of IP-multicast is still relatively small today, the probability of collisions is small (there are approximately 300 mill. IP-multicast addresses). However, there may be problems in the future, especially with multicast transport protocols, that require a number of multicast groups for a single session (e.g. protocols based on the Local Group Concept [14]). To reduce this problem, the IP-multicast service offers a mechanism to limit the spreading of sent multicast packets. Each packet carries a `time-to-live` field. The value of this field is set by the sender of the packet and can limit the scope of packets to the Local Area Network, to an institution (e.g. an university), a region (e.g. country) or a continent.

Similar to IP-unicast, usually applications don't use IP directly. They can use the User Datagram Protocol (UDP [21]), which enhances IP with the detection of bit-errors (by means of a checksum) and the *port* concept. In UDP-multicast ports provide a mechanism to multiplex and demultiplex several packet streams in the same group. However, UDP provides no reliability mechanisms to IP-multicast.

The protocol used for reliable unicast in the Internet, TCP, does not support the IP-multicast service, and thus cannot be used for multicasting.

2.2 Reliable Multicast

The IP-multicast service is sufficient for many applications that can tolerate packet loss. However, applications that require the guaranteed delivery of packets need additional mechanisms at the transport protocol level. Such applications are e.g. mass file distribution, web caching and conference tools like white-boards.

This requirement reliability adds much complexity to multicast. Partially, this complexity stems from very different application requirements that have to be provided by the transport protocol. A systematic list of such application requirements has been published in an Internet Draft [2]. The following list shows some of them that are important for the transport protocols.

Number of senders: Is it sufficient, if the protocol supports only one sender (mass file distribution), or does the application allow many participants of the multicast session to send (e.g. a white-board application).

Number of receivers: Many reliable multicast protocols scale not very well with the number of receivers. This can be caused by the control bandwidth, state information or CPU load that grow with the number of receivers.

Join policy: In unicast a TCP transmission can start, when a connection has been established between the two peers. In a multicast session, it is not clear, if all receivers are already in the group, when the transmission begins. Some applications may allow the members to join late and sometimes even to request all the information sent so far (*catch up*).

Slow receivers: A characteristic for multicast is the common progress of all receivers, that is, all the receivers get the same data. However, to ensure the common progress for the whole group, the sending rate has to be limited to the capacities of the slowest receiver. The different receiving capacities of the receivers is called *heterogeneity* of receivers and is caused by the network topology, network congestion and the processing capacity of the receivers. Some applications may want to remove too slow receivers, whereas others prefer to ensure, that all receivers get the data.

The group based addressing of IP-multicast requires also more security considerations than unicast, since every host connected to the Internet can send packets to any group and can join every group to receive the packets.

The application requirements and the different approaches to congestion control, security and other problems introduce many degrees of freedom to reliable multicast. Today it seems, that it is not possible to develop a transport protocol that can be configured to provide optimal service for all applications. However, there are a number of different protocols available, each of them providing an optimized service for some applications.

Protocol developers, who try to develop transport protocols that are as flexible as possible, often use a new architectural design principle, the Application Level Framing (ALF [4]).

2.3 Application Level Framing

The Application Level Framing (ALF) is an architectural design principle for data communication protocols and has been introduced by David Clark and David Tennenhouse in 1990 [4].

The most common used design principle today is the layering of the protocol stack. The implementation of each layer can be exchanged without changing the functionality of the neighbor-layers.

One key aspect of the layering is *encapsulation* of packets of one layer in packets of the next lower layer. The layers can also *segment* the data it has to transmit into several parts and transmit each part as a separate packet. Both encapsulation and segmentation are transparent to the upper layer. Each layer delivers the data in the same form as it has received it from its upper layer.

The ALF principle integrates the application layers and the transport layer and seeks a more flexible functional decomposition. The basic concept of ALF is the framing of the data packets at the application *level* into so-called ADUs (Application Data Units); the level terminology is used to compare the new architecture with the layered architecture. Throughout the integrated layers the ADUs are the unit of transmission. Segmentation, which was generally performed at the transport layer to provide data units in a size suitable for the network layer, is not longer performed, and it is up to the application to frame the packets in an adequate size for transmission.

The packet header of an ADU is visible at all levels of the integrated protocol stack. The encapsulation principle is given up and each field can be used at all levels, although it has to be clear, which functional module is responsible to set each field.

The goal of ALF is to provide flexibility and efficiency in the use of the network. However, this is reached in forcing the application programmer to implement a bigger part of the overall transport functionality.

2.3.1 Features of ALF

The following concepts are provided by ALF that give the application the flexibility to use the transport system in the most efficient way:

Out-of-order processing

Ordered delivery is a service that is provided by traditional transport protocols. The segmentation of the application data into transport layer PDUs however prevents the application and presentation layer to perform the presentation conversion and processing of PDUs received out-of-order. Although the position of the PDU in the PDU stream is known to the protocol, the application and presentation layer can only process data units known to them, not on the protocols PDUs.

ADUs are the data units provided by the application. They can be independently processed both at the application and the presentation level and are thus make out-of-order processing feasible.

The reason why out-of-order delivery of ADUs can be important to applications, is the possibility to have a steady CPU load at the receivers.

Steady processing at the receiver

When the transport protocol has to deliver the data ordered, packet losses lead to a bursty delivery of data. All packets successfully received but ordered behind a lost packet in the packet stream will be delayed for delivery until the loss is repaired. Then all packets received successfully will be delivered at once to the application. For applications that have high CPU requirements to process the received data, this can lead to idle cycles when waiting for loss recovery, and to overload when a burst of packets is delivered. For applications, that can structure their ADUs appropriately, out-of-order delivery helps the application to keep on processing, even if ADU losses occur.

However, applications that rely on an ordered ADU stream have to perform the ordering themselves by delaying the processing of out-of-order ADUs. On the other hand, it is easy for a protocol developer to provide this functionality outside the protocol level as some kind of link library or something equivalent.

Flexible reaction to lost data

To recover from packet loss, transport protocols normally buffer the PDUs when they are sent in a *retransmission buffer*. When a PDU gets lost, the sender can restore it from the retransmission buffer and send it again. This algorithm does not allow the application to influence the loss recovery, again caused by the segmentation of the application data into PDUs. Some applications, however, could benefit, if they could decide, how to react to packet loss. Some applications may be able to easily recompute the lost data, and thus saving the buffer space. Other application might want to sent different data to repair the previous loss or might even avoid retransmissions at all for data that is of no more use due to the delay!

The ALF principle gives the decision, of how to handle a received retransmission request to the application. Since the application knows about the contents of every transmission unit (the ADU), the application can use the optimal way to provide the requested data. Of course, the tradeoff for this flexibility is the requirement to the application programmer to implement the preferred way of handling the data himself.

2.3.2 ADU Names

If transport protocols are used that deliver the data ordered, the presentation conversion and application processing of the received data can be performed with the knowledge of the complete data stream up to the received packet.

The ALF principle allows the delivery of ADUs, when there are still missing ADUs that are positioned earlier in the ADU stream. This means, that the context necessary to process the ADU may be still incomplete. Without the context information the out-of-order delivery only shifts the task of reordering from the transport protocol to the application. This context information can only be provided by the sender of the ADU.

The solution suggested by the original ALF concept uses a special field in each ADU, the *ADU name*. The ADU name has to provide the context required by the receiving application to identify the data of every received ADU independent of the history of received ADUs.

This information is only relevant to the application. The protocol mechanisms, that detect the loss of packets still require some sequence number. Since this is already some context information, some applications may be able to do out-of-order processing even with an empty ADU name. An example would be a simple file transfer application: If the protocol uses byte-sequence numbers, this information is already sufficient to write the received data to the disk, even if there is still data missing with lower byte-sequence numbers. This example is very simple in that it assumes, that both the file name and the byte-offset of the first ADU is known a-priori to the receiver (or are transmitted in another way). A more realistic file transfer application could put the file name and also the byte-offset into the ADU name. The byte-offset of the data, relative to the start of the file, is required, since the sequence number of the first ADU of the file is not know a-priori by the receiver. The example program that explains the RMFP API in section 6.6 is just such a file transfer application.

The ADU naming in respect to the transport protocols reliability mechanisms will be subject to a more detailed analysis in section 3.2.1.

2.4 Protocol Templates

ALF introduces the integration of the protocol levels up from the transport level to the application level. This integration achieves more flexibility, but it requires more effort for the application programmers, since at least parts of the functionality provided by the layered protocol stack are moved to the application.

Some reliable multicast protocols are even just defined on the paper as *templates*. The term template has been introduced by Hofmann in [14] and describes a generic definition of protocol mechanisms. The Local Group Concept [14] is such a template. Another example is the *Scalable Reliable Multicast* SRM [9]. Especially SRM is designed to be implemented together with the application to be tailored to the application requirements. Since the cost of this approach is high for the application developer, both templates are also implemented as stand-alone protocols, that come as link-libraries and can be used easily by applications. The implementation of LGC is the *Local Group Multicast Protocol* (LGMP [24]) and one of the SRM implementations is GSRM [10]. However, such implementations limit again the flexibility.

2.5 Protocol Frameworks

Protocol frameworks are an attempt to integrate several protocol templates in a single implementation, e.g. a link-library. They provide a single interface to all protocols, so that the application programmer can easily change the used protocol without many changes to his code.

Frameworks are distinguished from a simple common interface in providing some functionality that are common to the integrated protocols, e.g. the network interface, packet formats and managing functionality.

At this time there are two frameworks in the development. One is the Reliable Multicast Framework (RMF [7]), and the other is the Reliable Multicast Framing Protocol (RMFP), that is subject of this report.

RMFP and RMF are compared in more detail in section 7.6.1.

Chapter 3

RMFP

3.1 Overview

The Reliable Multicast Framing Protocol is a framework for protocols. It is intended to give application developers an implementation of reliable multicast transport protocols. Also, protocol developers can use RMFP as framework for their implementations. This has the advantage, that parts of the transport protocol functionality are already implemented in RMFP and can be used. The API that comes with a RMFP implementation allows to use newly integrated protocols in existing applications without difficult changes in their networking code.

RMFP is an attempt to solve the problems that arise with reliable multicast: To provide application developers with flexible *and* complete transport protocol functionality. It is based on ALF (see section 2.3), since ALF provides the required flexibility. The ALF principle, however, is also responsible for the difficulties in developing protocol frameworks like RMFP: The tight integration of the transport protocol functionality, that is the foundation of ALF, rules out a simple common interface to integrate several protocols. The layered approach supports common interfaces in a much better way: One example is the BSD socket interface used in most networking applications in UNIX operating systems. This interface supports as different protocols as UDP (unreliable datagrams) and TCP (reliable and stream-oriented). Another example is the *Protocol Independent Interface* (PII [11]). Such common interfaces allow to configure the protocol specific functionality by a number of parameters, and the number of functions at the API is rather small.

If ALF is to be used, the application needs broader access to the protocols' internal structures and mechanisms: Access to packet headers, control of the retransmissions and flow and congestion control mechanisms are some examples. This leads to a broader API to the protocol functionality. And if several protocols use the same API, they also have to use similar structures internally. Since the applications have access to the packet headers, at least for the ADUs and some application relevant control packets, the protocols have even to use the same packet formats.

These common packet formats are the foundation of the RMFP specification. The specification also defines, how protocols can be integrated into the RMFP framework. The integrated protocols are called *protocol profiles*. For each profile there has to be a specification, that defines how the RMFP packet formats are used and what profile specific packets are introduced.

3.2 Mechanisms

The the next sections present the mechanisms that are either part of the RMFP specification or have been the foundation for the specification of the packet formats.

3.2.1 Error Control

Since RMFP is a framework for *reliable* multicast, the error control is the most important issue. RMFP itself provides no error control functionality, this is the task of the protocol profiles. However, since RMFP follows the ALF principle, some of the error control functionality has to be provided by the application.

- RMFP specifies the format of the ADUs. The sequence number field and the FEC and retransmissions flags of the ADU header are primarily provided for the protocol profiles to be used for error control.
- Any protocol profile has to be able to *detect* the loss of ADUs and to initiate the retransmissions. This includes the transmission of control information from a receiver that suffered a loss to some group member that can perform the retransmission.
- The application has to perform the *loss recovery*. When the protocol profile of a group member informs the application about a retransmission request of another group member, the application has to provide the retransmission data and has to resend the ADU.

The ALF principle described in section 2.3 introduces ADUs as common unit of transmission for all layers from the transport protocol up to the application. To enable the unordered delivery of ADUs each ADU has an ADU name assigned that identifies the ADU data in the application context independent of the history of received ADUs. This ADU name is of no meaning to the transport protocol. However, the transport protocol uses its own naming concept to perform loss detection and recovery – the sequence numbers.

The remainder of this section assumes, that only one sender is active in the regarded session. This assumption simplifies the problem in a way, but without limiting generality. If there are several senders in a session, each sender will mark its ADUs with his *source ID*. Each member of the session has its unique source ID, and all packets can be assigned to their sender. Although the following analysis treats only the case of sessions with a single sender, multiple senders in a session can be regarded as independent from each other, and the discussion corresponds to each sender respectively.

The Automatic Repeat Request mechanism

The ARQ mechanism is one of the two basic approaches to ensure reliable data transmission, and the only one that is 100% reliable. The other mechanism, Forward Error Correction (FEC), can only reduce the loss-rate. ARQ consists of two components: The loss detection and the loss recovery.

Loss detection
Even if the application does not need any ordering of the data, the protocol will use some kind of sequence numbers to assign an order to its ADU stream. Normally, this order corresponds to the sending order. Losses are detected by means of gaps in the sequence number space. The actual algorithm can reside at the receiver (receiver-based loss detection) or at the sender (sender-based loss detection). The loss detection algorithm uses some state information, the history of ADUs already received successfully, and computes the necessary information to do the loss recovery: The sequence numbers of the lost ADUs.

- The receiver-based algorithm detects the lost ADUs at the receiving protocol instance, that will encode and transmit the sequence numbers of the lost ADUs in control packets. The addressee of those control packets is a group member that can retransmit the ADU. The encoding is mostly done in form of *spans* to reduce the necessary bandwidth. The addressee of the control packets (in a unicast transmission this is the sender) can then compute the sequence numbers of the lost packets without other information.
- The sender-based algorithm requires the receiving protocol instance to send *positive acknowledgments* to the sender for every packet received. Since the receiver does not keep state information about the received ADUs, it cannot compute the gaps in the sequence number space. Also, the lack of ADU history prevents the receiver from encoding the positive acknowledgments into spans (spans implicitly encode the sequence numbers of the lost ADUs, and this is the receiver-based scheme). The addressee of the positive acknowledgments (generally the sender of the ADUs) uses its information about the transmitted ADUs and the state information of the already acknowledged ADUs to determine the sequence numbers of the ADUs to retransmit.

In the remainder of the report only the receiver-based approach will be considered, since at least for multicast, the sender-based approach has several disadvantages (a comparison of the receiver- and sender-based approach can be found in [20]):

- To detect the gaps, the history of lost and received ADUs has to be available. If the sender has to do this, the number of receivers would be limited by the senders capacity in keeping this state information.

- since the sender has to track the history of all ADUs at all receivers, it has to process the control packets from all receivers. With many receivers, the sender will suffer the so-called *ack-implosion*. This is an overload of the sender by processing the control packets. Some receiver-based protocols use the so-called *NACK suppression* mechanism to prevent the overload of control packets. A receiver that suffered a loss, does not need to send a control packet with lost ADU information, if another receiver has done so before for the same ADU. If the retransmission for the first request is transmitted, both receivers will receive it.

The SRM protocol (see chapter 4) uses a receiver-based mechanism with NACK suppression to free the senders completely from management tasks for special error control state information and to avoid the ack-implosion.

The LGC protocol (see chapter 5) uses a combined approach. To prevent the nack-implosion at the sender, LGC builds a tree structure with the sender as source. The control packets are not sent directly to the sender, but are gathered at the inner nodes (group controllers) of the tree. Thus, the sender and each of the group controllers has to process the control packets of the (limited) number of its children.

Loss recovery

In ARQ lost packets are retransmitted. For unicast transmission, the sender of the retransmissions will always be the original sender. For multicast transmission, receivers that have successfully received a given PDU can also retransmit that PDU to the receivers that suffered a loss of that PDU. An example protocol is SRM [9], where every group member is involved in loss recovery.

Forward Error Control

FEC reduces the loss-rate in sending redundancy information additionally to the useful data. The encoding takes a block of n ADUs and computes a given number k of redundancy packets. The n+k packets form a transmission group. If the packets of a transmission group are sent, it is sufficient to receive any subset of size n of the transmission group to reconstruct the original n ADUs. However, if more then k packets of the transmission group get lost, the losses cannot be repaired. Thus, FEC can only reduce the packet loss-rate. An introduction to FEC can be found in [23].

FEC can be combined with ARQ to so-called *hybrid ARQ*. This mechanism is especially useful for reliable multicast, since it can effectively reduce the overall loss-rate and thus retransmissions, too. An investigation of hybrid ARQ has been presented in [19].

There are several possibilities to use FEC in RMFP:

1. The usage of FEC within RMFP transparent for the protocol profile, i.e. as some layer under the profile [16] could improve the behavior of all profiles. The effects of such a transparent FEC mechanism have been investigated in [16] and [19].

2. FEC can be implemented as a mechanism of a protocol profile.

3. The application can implement the FEC mechanism. It is possible, to have this done by some standard module provided by a RMFP implementation (see 3.2.5).

ALF and loss recovery

According to the ALF principle it's the application that has to manage the retransmission data. In RMFP the protocol profiles have the task to detect the losses and inform the application about the need of retransmissions. The application then provides the retransmission data. However, the protocol profiles use the *sequence numbers* to identify ADUs, whereas the application requires the *ADU name* to identify the ADUs. This leads to the need for a mapping between the protocols sequence numbers and the ADU names.

The retransmissions of ADUs can only be performed by group members that have the ADU either sent themselves or received already successfully. Since the complete ADU contains both the sequence number and the ADU name, the mapping information required to provide the retransmission data is already available at the retransmitting group member. The member can map the sequence number to the ADU name and then the ADU name to the retransmission data. Depending on the management of the retransmission data, the mapping may also be performed directly from sequence number to retransmission data.

The RMFP specification doesn't specify, if the mapping from sequence number to ADU name should be performed already at the protocol profile or at the application; this decision is implementation dependent. The implementation of RMFP developed together with this thesis delegates this task to the protocol profile.

3.2.2 Hierarchical Naming with Objects

Additionally to the sequence number field and the ADU name there is another field in the ADU header to support the mechanisms to identify the data carried in the ADUs: The object ID field. It can be used to optimize the transmission overhead caused by the ADU name.

E.g. the file transfer application presented in section 2.3.2 puts the name of the file into the ADU name field of each ADU. If the file name includes some path name, the file name can become considerably big. This file name, however, doesn't change for all the ADUs belonging to the file; only the byte-offset field varies from ADU to ADU.

The object ID field can reduce the bandwidth required by the ADU name. Each file name used during the transmission is mapped onto a unique object ID. The file name can then be omitted in the ADU name. The problem with this approach is the transmission of the mapping information of object ID to ADU name that is required at the receivers to process the ADUs. It can be transmitted in one of the ADUs of the file in the ADU name field or separately as session information. In the example, the first approach has the disadvantage, that all ADUs of the file can only be processed, when the ADU with the file name in the ADU name field has been received successfully. The other approach has the disadvantage, that the session packet has to be transmitted reliably, since the ADUs of a file are only useful, if the file name is known.

How the object ID field is used is up to the application. It has to find the optimal way to suit its requirements and to optimize the used bandwidth.

Another issue is the relationship between objects and sequence numbers. Three possibilities are suggested:

1. The object ID is independent to the sequence number field and is only used by the application. The ADUs are sequenced relative to the start of the session and are not influenced by the object ID. This is suitable for applications that require *all* ADUs to be received reliably. This is the mechanism defined at the specification of RMFP and the SRM profile.

2. The sequence numbers are computed relative to the objects and the object IDs are sequenced. If the objects are transmitted one after the other, i.e. the ADUs of several objects are not interleaved, every two ADUs can be compared in respect to their sending order.

   To reorder the ADUs and to detect ADU losses at the receiver, the object IDs and sequence numbers are compared hierarchically: Since the objects are transmitted sequentially, the sending order of two ADUs can be computed out of the object ID, if the object IDs of the ADUs differ. If both ADUs belong to the same object, the sequence number decides about the order. The loss detection is more difficult than with the first sequencing approach:

   - Lost ADUs *within* an object are detected by gaps in the objects sequence number name-space.
   - Objects lost in total are detected by gaps in the object ID name-space.
   - If the first or last ADUs of an object are lost, the start-of-object/end-of-object flags are used to detect the losses.

   These mechanisms are sufficient to be able to detect all possible ADU losses, although, in the third case, it is not always possible to determine the number of all lost ADUs and their sequence numbers. The coding of negative acknowledgments for retransmission requests must be performed as spans.

   The problematic loss of ADUs around object boundaries (i.e. the loss of ADUs carrying start-of-object/end-of-object flags) imposes the constraint on the transmission order of objects: The transmission of an object must be completed (by an ADU carrying the end-of-object flag) before the first ADU of the next object (i.e. an object with an object ID incremented by one) can be sent. This limits the usability of this approach for applications that want to transmit several objects *simultaneously*, e.g. a whiteboard application. Such applications require the next model.

3. The sequence numbers are computed relative to the objects, i.e. every object has its own sequence number space, but there is no ordering relation between the ADUs of different objects. This requires, however, that all control information has to refer to

each object independently, too. In section 7.4.1 a concept is presented that is based on this model of sequencing. It allows the receiver application to decide, which objects have to be received reliably (*semantic reliability*). Another very general approach of how this can be done is described in [22].

3.2.3 Late-joining Receivers

An important problem for reliable multicast is the synchronization of late-joining receivers. In general, applications may require to allow receivers to join an ongoing session. Such receivers have to figure out, at which point of the ADU stream they start with the receipt of data.

The following discussion assumes, that the ADUs are sequenced relative to the session and not relative to the objects (see section 3.2.2), since this is the method used in the current specification of RMFP.

With the use of sequence numbers RMFP imposes a total order on each sender's ADU stream, but only a partial order on the combined ADU stream of all senders. Therefore the synchronization of joining receivers has to be performed independently for each sender in the session. In the rest of this section this is assumed without further notion.

In the rest of the section the term *initial sequence number* refers to the sequence number of the packet with the lowest sequence number that a receiver processes. A receiver keeps information about the initial sequence number for each sender independently. Similarly, the *highest-sequence-number-sent* is the highest sequence number used by the sender. For a receiver, this is actually the highest sequence number *seen* from a given sender so far.

There are several solutions:

- The receiver uses the ADU with the lowest sequence number it receives. It won't ask for retransmissions for any ADU with a lower sequence number.
- The senders transmit synchronization points as session information. Those synchronization points are sequence numbers within their ADU stream, that are determined by the application and are useful in the application context. A joining receiver that receives such information, can ask for retransmission of all ADUs starting at this synchronization point.

It is up to the application to decide, which style of receiver synchronization to use. Consequently, the RMFP supports both. The senders transmit the information of the style to use and if necessary the current synchronization point within the sender report packets (section 3.3).

RMFP defines following behavior at a joining receiver:

1. The receiver has no information yet. This means that the receiver has not yet received any information about the sequence numbers sent by the sender.

   ADU received: The sequence number of this ADU is used as an initial sequence number.

Highest-sequence-number-sent received: This information is carried e.g. by a SRM heartbeat control packet. The next sequence number is used as the initial sequence number.

Synchronization point received: The receiver takes the synchronization point as the first sequence number of the ADU stream from the sender. Since the sender report packet carrying the synchronization information also carries the highest-sequence-number-sent, the receiver can ask for retransmission for all ADUs starting with the synchronization point's sequence number and up to the highest-sequence-number-sent.

2. The receiver is synchronized without synchronization point received. The receiver is already synchronized due to a received ADU or highest-sequence-number-sent information.

   ADU received: If the ADU's sequence number is lower than the present initial sequence number for that sender, the initial sequence number is set back to the ADU's sequence number and missing packets starting with this sequence number are requested for retransmission.

   Highest-sequence-number-sent received: It should be greater than the already known initial sequence number, which has no impact on the synchronization. If it is not, which could happen in case of out-of-order receipt of control packets, this information is discarded.

   Synchronization point received: If the synchronization point's sequence number is greater than or equal to the initial sequence number, the information is regarded as obsolete. Otherwise, the initial sequence number is set back to the received sequence number and the missing packets are requested for retransmission.

3. The receiver has already received a synchronization point. This implies, that the synchronization process is already finished. Received synchronization information is not considered anymore at all, and ADUs with lower sequence numbers than the used synchronization point are discarded.

Because of the finite sequence number space, there are problems with the described synchronization algorithm. To ensure proper operation the synchronization process has to be stopped after a defined span of sequence numbers has been seen by a receiver (again independently for each sender). In the implementation the size of the span is a quarter of the sequence number space. At this point the receiver assumes that it is fully synchronized.

3.2.4 Automatic Profile Configuration

One of the foundations that provide flexibility in RMFP are the different protocol profiles. The protocol profiles have differing characteristics and the optimal protocol profile depends on the scenario, i.e. the number of group members, the number of senders etc. (see section

2.2). If it is clear at the development of an application, that one of the protocol profiles is a good choice for all envisioned scenarios for the application, the application can always use that profile and every group member always knows this profile, when it joins.

However, for some applications it might prove useful to support several protocol profiles, depending on the scenario. The information of the profile has to be distributed to all members. RMFP provides a mechanism for joining members to be configured automatically by received sender control packets (section 3.3). However, this mechanism only works correctly, if the senders of a session agree about the profile. RMFP provides no mechanism to deal with conflicts, if members of the same group use different profiles.

3.2.5 External Modules

Some of the standard functionality of other transport protocols have been omitted in RMFP to allow the applications to use the transport functionality in a more flexible way. However, many applications could use the standard functionality. To simplify the use of RMFP it is possible to use some implementations of this functionality as external modules. Some possible modules are the following:

Retransmission buffer: According to the ALF principle the application is responsible to manage the retransmission data. This brings flexibility, but many application programmers might want to use the classical mechanism, a simple buffer indexed by the sequence numbers.

Reordering module: ALF explicitly introduces the unordered delivery of received ADUs. Applications, that do not require the flexibility and performance of that mechanism or are not even capable to process the ADUs out-of-order could be implemented simpler, if they could rely on ordered delivery.

3.3 The RMFP Specification

In this section the packet formats of RMFP are specified. This specification is based on an Internet draft [6] that has been modified and extended. The implementation of RMFP refers to the following specification.

RMFP specifies three types of packets:

1. The ADUs are the data packets.
2. The *sender and receiver control packets* carry some session information.
3. The *session packet header* allows the free definition of new session packets that can be defined on demand by the application.

The protocol profiles that provide the reliability can define own control packets. Those profile specifications are not part of the RMFP specification, but are defined separately. Two profile specifications for SRM and LGC can be found in sections 4 and 5.

3.3.1 General Aspects

Network environment

Since one of the basic principles of RMFP is the tight integration of the protocol functionality into the application, RMFP was developed to use the UDP/IP protocol [21]. On many operating systems like UNIX, the usage of raw IP is currently only possible within the kernel or for applications with special permissions and would thus prevent the tight integration into the application or general usage.

As a consequence, the packet formats do not contain checksum fields, since this service is already provided by UDP.

Since there are systems using different byte-orders, there is a network byte-order defined to ensure interoperability between different hardware platforms. The network byte-order used at RMFP is the same as used within the IP protocol stack. RMFP provides the reordering, if necessary, for the fields of the packet headers.

This specification suggests an addressing scheme for the different packet types: For each of the three packet types – ADU, control and session – RMFP uses the same IP multicast address, but different UDP ports. Since all packets can be identified due to their type field, they could be well sent on the same IP multicast address/UDP port. However, such an approach can lead to inefficiencies at the buffer management, since the type of a received packet can only be retrieved after the packet has been copied into the application buffers. That's why RMFP relies on UDP to multiplex/demultiplex the three flows.

Some protocol profiles may need to use more addresses and/or ports or cannot even use the global multicast groups in which every group member takes part. However, the profile developers should seek to be as compliant as possible to this suggestion to reduce profile specific differences at the API.

This specification requires the application to provide a single address/port pair for the session, the *session address* and the *session port*.

- The *data flow* (all the ADUs) is assigned to the session address/session port.
- The *control flow* (sender and receiver report packets as well as the profiles' control packets) is assigned to the session address/session port + 1.
- The *session flow* (all application defined session packets) is assigned to the session address/session port + 2.

System environment

To avoid problems with alignment, all packet fields are naturally aligned, e.g. all two-octet sized fields are placed on even addresses. The packets themselves are assumed to be four-octet aligned.

```
0                   8                  16                  24               31
+-------------------+-------------------+--------------------------------------+
| V  |P|R|F|S|E|X   |   Payload Type    |                Length                |
+-------------------+-------------------+--------------------------------------+
|                                Source ID                                     |
+---------------------------------------+--------------------------------------+
|            Sequence Number            |               Object ID              |
+-------------------+-------------------+--------------------------------------+
|    Name Length    |                         ADU Name                         |
+-------------------+----------------------------------------------------------+
|                                ADU Name                                      |
+------------------------------------------------------------------------------+
|                                 ......                                       |
```

Figure 3.1: The RMFP data packet (ADU)

3.3.2 RMFP ADU Format

The RMFP ADUs have the header format shown in fig. 3.1. The intention in designing this format was to include enough information to be sufficient for the different protocol profiles, but to keep the overhead small.

The fields have following meaning:

Version(V): 2 bits
This field identifies the version of RMFP.

Padding(P): 1 bit
If the padding bit is set, the packet contains one or more additional padding octets at the end, which are not part of the payload. The last octet of the padding contains a count of how many octets should be ignored.

The padding bytes keep all the ADUs four-byte aligned.

Retransmission (R): 1 bit
This bit, when set, indicates that the ADU is being retransmitted.

Forward Error Correction (F): 1 bit
This bit, when set, indicates that FEC is used.

Start of Object (S): 1 bit
This bit, when set, indicates that the ADU is the first one of an object.

End of Object (E): 1 bit
This bit, when set, indicates that the ADU is the last one of an object.

Exceptional Handling (X): 1 bit
This bit is free for use by the application. It is not processed at RMFP or any profile and is intended to allow the application to mark ADUs that should be treated in a unusual way.

Payload Type: 8 bits

This field is intended to serve the application in a similar way as the payload type field in RTP [25] does. The application can use this field to indicate the type of the payload. Some values of this field are used to indicate control or session packets used by RMFP and the profiles and may not be used for application purposes. Following values are so far defined:

201: Sender report packets.

202: Receiver report packets.

203: Session packets.

205: SRM control packets.

206: LGC control packets.

The application can use the other values freely, however, it is possible that other values above 200 may be used by other profiles, or added functionality in future versions of RMFP.

Length: 16 bits

This field identifies the length of the packet in 32 bits minus one, including the header and any padding. To count in multiples of four octets avoids an alignment check. This algorithm has been introduced by RTP [25].

It can be used to combine several ADUs into one UDP packet. In a compound UDP packet only the length fields allow the detection of the ADU boundaries.

When several ADUs (original and retransmitted) are concatenated within one UDP packet, the original ADUs should all be placed at the beginning of the UDP packet so that receivers that do not encounter losses can just drop the tail of the retransmitted ADUs without processing it.

Source ID: 32 bits

This field identifies the source. The source IDs are generated randomly similar to the SSRC field in RTP to avoid collisions between several members.

Sequence Number: 16 bits

The sequence number is an ADU counter. It is incremented by one for each ADU sent. It can be used to detect ADU losses and calculate loss rates.

The exact semantics of the sequence number is determined by the protocol profile. It is possible to count the sequence number starting with the first ADU sent and incrementing it for each ADU throughout the session. Another possibility would be to use the sequence number object-relative, i.e. each object has its own counter assigned starting at zero for its first ADU (see section 3.2.2).

0	8	16	24 31
V |P|R|F|S|E|X	Payload Type	Length	
Source ID			
Profile ID	L|S|V|	reserved	
Base object ID		Base sequence number	
Current object ID		Highest sequence number	
Application-specific extensions			

Figure 3.2: The RMFP sender report packet

Object ID: 16 bits
This field identifies the object carried in the packet. How this field can be used is discussed in section 3.2.2.

Name Length: 8 bits
This field specifies the length in octets of the following ADU Name. 0 is a valid value, indicating that no explicit ADU name is available.

ADU Name: variable
The ADU name is used by the application to identify an ADU in the application context. The contents of this field are completely transparent to RMFP and the protocol profiles.

The length of the ADU name can be between 0 and 255 octets. There can be unused octets to ensure proper alignment (32bit) within the ADU header.

This field can contain the information to identify both the object and the position within the object of the ADU, e.g. the filename and the byte-offset for ADUs in a file transfer application. However, the application can also use the object IDs and sequence numbers to identify objects and ADUs.

3.3.3 RMFP Control Packet Format

RMFP control packets include sender report packets and receiver report packets. Those packets can be used by the senders and receivers respectively to transmit session information.

Sender Report packet

Sender report packets are sent periodically by the sender and contain information about the current sending state. They can help to configure new joining receivers and provide information to detect tail losses. The structure of the header is shown in fig. 3.2.

Version(V): 2 bits
This field identifies the version number.

Padding(P): 1 bit
If the padding bit is set, the packet contains one or more additional padding octets at the end which are not part of the payload. The last octet of the padding contains a count of how many octets should be ignored.

Since the actual header is already aligned, the padding flag is only necessary, if an application specific extension is included in the packet.

SR Type: 5 bits
This field has no interpretation by RMFP and can be used by the application, e.g. to transmit extra information like an `end of transmission` indication. It might also be used to denote the type of the application specific extension.

Payload Type: 8 bits
This field is set to 201 for sender report packets

Header Length: 16 bits
This field specifies the length of the packet in multiples of 32 bits minus one.

Source ID: 32 bits
This field identifies the sender.

Profile: 8 bits
Indicates the type of the protocol profile used. It is used together with the LSV, first object ID and lowest sequence number fields to configure late joining receivers (section 3.2.4). A receiver, that wants to join a session and does not know a-priori which protocol profile is used, can wait for receipt of a sender report packet and configure its protocol profile according to this field.

Lowest Sequence Valid (LSV): 2 bit
These bits define the interpretation of lowest sequence number field:

00: The sequence number of the first ADU sent by the sender in this session.

01: The sequence number of some position in the transmission that can be used to synchronize.

10: No valid information. The sender provides no special help to synchronize. The new receiver should synchronize its join on the first ADU it receives.

If the lowest sequence number fields is valid, a late-joining receiver can ask for retransmission back to the indicated sequence number. The sender can choose the value of this field appropriately to mark some logical boundery in the ADU stream (see also section 3.2.3).

First Object ID: 16 bits
The object ID for late-joining receivers to synchronize.

0	8	16	24 31
V |P| RC	Payload Type	Length	
Source ID			
Sender's source ID 1			
Fraction Lost	reserved	Highest Sequence Number	
......			
Sender's source ID N			
Fraction Lost	reserved	Highest Sequence Number	
Application-specific extension			

Figure 3.3: The RMFP receiver report packet

Lowest Sequence Number: 16 bits
The sequence number for late-joining receivers to synchronize.

Current Object ID: 16 bits
This field and the highest sequence number field are used to indicate the current state of the sender. The receivers can use this information to detect tail-losses.

Highest Sequence Number: 16 bits
This information comes together with the current object ID and is the sequence number of the last ADU sent. It is used to detect tail-losses.

Receiver Report packet

Receiver report packets are sent periodically by the receivers to give feedback on congestion and packet losses. They contain some receive statistics for each sender. The format of this packet type is shown in fig. 3.3.

The fields have the following meaning.

Version(V): 2 bits
This field contains the version number.

Padding(P): 1 bit
The padding bit is used to force alignment of the packet. It is used in the same way as in the sender report packet.

Report Block Count(RC): 5 bits
The RC denotes, how many report blocks are contained in this packet. Each report block consists of a sender's source ID, a fraction lost field and a highest sequence number field.

Payload Type: 8 bits
This field is set to 202 for receiver report packets.

Header Length: 16 bits
This field specifies the length of the packet in multiples of 32 bits minus one.

Source ID: 32 bits
This field identifies the sender of this packet (a receiver).

Senders Source ID N: 32 bits
Each report block has this field. It denotes the sender of the following fraction lost and highest sequence number fields.

Fraction Lost: 8 bits
The fraction of packets lost since last receiver report, expressed as a fixed point number with the binary point at the left edge of the field. Fraction lost is the loss rate seen by the receiver in respect to the sender identified by the previous `sender's source ID` field. The information may be used for congestion control or error recovery (FEC) by the sender.

Highest Sequence Number: 16 bits
This field indicates the highest sequence number received from the corresponding sender so far.

3.3.4 RMFP Session Packet Format

The session packets are used to enable group members to easily exchange session information. RMFP defines a very light-weight approach, that merely supports the sending and receiving of unreliable data, that is marked as session information. Thus the RMFP just defines the protocol header and provides the transmission and receipt of such packets. There are no special packets defined for some specific use, this is up to the application. Session packets can be used e.g. to support the following functions:

Remote configuration: A sender can transmit configuration parameters to configure other members. This mechanism is only used to transmit parameters. The application has the responsibility to use the parameters to configure the protocol.

Support at joining a session: A member joining a session has to be informed about the current state of the session. It could use a special session packet to issue some status request packet, and the senders can answer to that packet with some status reply session packets.

Application level reliability mechanisms: An applications might want to enhance some profile's reliability mechanism, e.g. to implement a retransmission mechanism on the object level (i.e. for objects consisting of several ADUs).

<table>
<tr><td>0</td><td>8</td><td>16</td><td>24 31</td></tr>
<tr><td>V |P|R|F|S|E|X</td><td>Payload Type</td><td colspan="2">Length</td></tr>
<tr><td colspan="4">Source ID</td></tr>
</table>

Figure 3.4: The RMFP session packet

The packet format is shown in fig. 3.4.

Version(V): 2 bits
This field identifies the version number.

Padding(P): 1 bit
The padding bit is used to force alignment of the packet. It is used in the same way as in the sender report packet.

Flags: 5 bits
The usage of this field is defined by the application. It could be used e.g. to identify different types of session packets.

Payload Type: 8 bits
This field is set to 203 for RMFP session packets.

Length: 16 bits
This field specifies the length of the packet in multiples of 32 bits minus one, including the header and any padding.

Source ID: 32 bits
This field identifies the sender of the session packet. It is calculated like the length field of the ADU.

Chapter 4

The SRM protocol

The *Scalable Reliable Multicast* has been introduced as a protocol template[1] in [9]. It is designed according to the ALF principle, in that it leaves much of the protocol functionality at the application. It defines a set of algorithms, that can be used to achieve eventually reliable multicast functionality.

The algorithms of this protocol template have been designed to be efficient, robust and scalable to very large networks and sessions. The framework has been prototyped in wb, a distributed white-board application [17], and has been extensively tested on a global scale with sessions ranging from a few to more than 1000 participants.

In SRM all members of a multicast session, both senders and receivers join a single, global IP-multicast group. To take part in an SRM multicast session, the members just have to join the IP-multicast group. They can then receive all the data transmitted and can take immediately part in the reliability mechanisms. There is no member in a multicast session that has a special role, and every member can join and leave the session at any time.

To achieve the necessary reliability with such a loose relationship between the members, the reliability mechanism are receiver-based (see section 3.2.1). All receivers have to detect lost ADUs by means of gaps in some sequence number space and have to issue retransmission requests (*Negative ACKnowledgments*). Every other group member that receives a retransmission request can perform the retransmission, if it has already received the requested ADU.

To avoid a flooding of the network with NACK packets, if many members have lost the same ADU, a special NACK suppression mechanism is applied: Before sending the NACK for a given ADU, the members wait a random time. If they receive a NACK from another member for that ADU during that period, they backoff the delay for the NACK exponentially.

[1] In its original specification [9] SRM has been called a *framework*. In this document the term "framework" has another meaning, and thus here SRM is called a *template*.

A similar mechanism is applied to suppress duplicate retransmissions. A member, that is able to answer a retransmission request, waits for a random time before sending the retransmission. If the member receives the requested packet during the delay, it cancels the retransmission.

4.1 Mechanisms

The mechanisms defined in this section refer to the SRM protocol profile for RMFP.

4.1.1 Loss Detection

ADU losses are always detected by the receivers. To allow the detection of losses, the ADUs are ordered in some way, e.g. by sequence numbers. The receivers can detect losses by means of gaps in the sequence number space of the received ADUs.

This mechanism is not sufficient to detect losses of the ADUs sent last (so-called *tail-losses*). SRM senders use special control packets, so-called heartbeats, that are used to distribute the sequence number of the last ADU sent, if there has been no ADU transmission for a while.

4.1.2 Feedback Suppression

If a receiver A has detected the loss of an ADU, it schedules a retransmission request (NACK). The actual transmission of the request is delayed randomly. The request timer is chosen from the uniform distribution on

$[C_1 * d(S,A), (C_1 + C_2) * d(S,A)]$ seconds,

where $d(S,A)$ is host A's estimate of the one-way delay to the original source S of the missing data and C_1 and C_2 are constants. As specified in [9], $C_1 = C_2 = 2$. When the request timer expires, host A sends a request for the missing data, and doubles the request timer to wait for the repair.

The distance based delay makes it probable, that the request timers of receivers close to sender of the lost ADU expires earlier than the request timer of receivers further away. Thus, the NACK of such a receiver can reach the other receivers that have experienced the loss of the same ADU before their request timer expire. They will react on the receipt of the NACK in backing of their request timers exponentially and calculate their new request timers randomly. If the current request timer has been chosen from the uniform distribution on

$2^i * [C_1 * d(S,A), (C_1 + C_2) * d(S,A)]$,

then the backed-off timer is randomly chosen from the uniform distribution on

$2^{i+1} * [C_1 * d(S,A), (C_1 + C_2) * d(S,A)]$,

However, to prevent multiple backoffs for duplicate NACKs, after a backoff received NACKs lead to no new backoffs, until half of the current request delay has passed.

To prevent unusable high delays after repeated packet losses, i should also limited to a small integer (e.g. 5).

4.1.3 Repair Suppression

Every group member that receives a NACK for an ADU it has received successfully can send the retransmission ADU. To avoid, that every group member that has received the ADU and the NACK sends a retransmission, the retransmissions are delayed by a similar mechanism as used to suppress retransmission requests.

When a member B receives a request from a member A that the member B is able of answering, member B sets a repair timer to a value from the uniform distribution on

$[D_1 * d(A,B), (D_1 + D_2) * d(A,B)]$

D_1 and D_2 are constants. As specified in [9], $D_1 = D_2 = log10(G)$, where G is the current number of members in the session. When this repair timer expires, then host B multicasts the repair. The delay for the the retransmissions is dependent on the distance between A and B. This makes it probable, that the repair timers of members close the the member that lost the ADU expire earlier than the repair timers of of members more distant. If other members that are waiting for their repair timers to expire receive the retransmission, they will cancel their repair timers.

In order to prevent duplicate requests from triggering a responding set of duplicate repairs, the member B that sent or received a retransmission for a given ADU ignores requests for that ADU for $3 * d(S,B)$ seconds after sending or receiving a retransmission for that ADU. S is the original sender of the ADU or the sender of the first request.

4.1.4 Distance Estimation

For the overall performance of SRM it is necessary to estimate the distance between all pairs of members [2]. The distance between two members is estimated with a simple algorithm measuring the round-trip delay (RTT):

If a member A wants to measure the RTT to the other members, it multicast a *timestamp query* control packet. The packet contains a timestamp indicating when the packet has been sent. If another member B receives the timestamp query from member A, it will answer with a *timestamp reply* packet containing the timestamp received from A and the delay at B between receipt of the query and the transmission of the reply. If member A receives this query, it can estimate the RTT with the following formula:

$d(A,B) = (TRR - TQS - DEL)/2,$

where TRR is the time the timestamp reply has been received, TQS is the time the timestamp query has been sent and DEL is the delay at member B. TQS and DEL are obtained from the timestamp reply.

[2] This is the limiting factor for the group size in SRM

4.2 The SRM Profile

This section contains the definition of the SRM protocol profile for RMFP. It is based on an Internet draft [3]. The definitions of that draft have been modified to fix several bugs and fit to the current specification of RMFP.

The SRM profile implementation described in this report is designed according to this profile definition.

4.2.1 SRM over RMFP

The SRM profile uses the flows defined by RMFP (see section 3.3.1. The session flow is not used by SRM (it is up to the application to send/receive the session packets), whereas the data and the control flow are used by SRM.

The data flow is composed of the original ADUs and the retransmission ADUs. The ADUs use the format specified by RMFP (section 3.3.2). The original ADUs have the retransmission bit set to 0, whereas the retransmission ADUs have this bit set to 1.

The control flow is composed of the RMFP report packets (section 3.3.3) and the SRM control packets (nacks, time-stamp queries and replies, heartbeat). The RMFP report packets are used according to the RMFP specification. To provide the SRM functionality, four new control packets are defined by SRM:

- The Heartbeat packets are sent by the sources to indicate the sequence number of their last transmitted packet.

  This packet type is introduced although the sender report packets provide the same functionality, because the overhead of heartbeat packets is relatively small in comparison to the sender report packets, and their sending frequency is determined by the SRM specification. The sending of sender report packets, however, is initiated by the application.

- The NACK packets that are sent by receivers to request the retransmission of one or several packets.

- The timestamp request and timestamp reply are sent to estimate the distance (in time) between the members of a group.

All packets of the control flow, i.e. the RMFP report packets and the SRM control packets, can be concatenated to fill up UDP packets.

4.2.2 The Packet Formats

The common SRM control header

All SRM control packets share a common header (fig. 4.1). Since the header is the same for all SRM control packets, several of the SRM control packets can be concatenated after one instance of the common header.

0	8	16	24	31
V |P| CC	Payload Type	Length		
Source ID				

Figure 4.1: The common SRM packet header

The common header is derived from the first 8 octets of the report packets. They are distinguished from them by means of the payload type (=205 for SRM control packets).

The RR Type and SR type fields of the report packets are used as *chunk count* (CC). This field indicates the number of SRM control packets concatenated.

To decompose a concatenated SRM control packet, each control subpacket uses the first five bits (after the common header) as a `subtype` field, that identifies the type of the control subpacket.

Version (V): 2 bits
: Identifies the version of RMFP.

Padding (P): 1 bit
: This flag is not used, since all SRM control packets are 32 bit aligned by definition.

Chunk Count (CC): 5 bits
: The number of SRM control packets concatenated with this header.

Packet type (PT): 8 bits
: Contains the constant 205 to identify this as an SRM control packet.

Packet length (length): 16 bits
: The length of this control packet is calculated with the formula defined by RMFP: It is measured in 32-bit words minus one, including the header and all SRM control sub-packets concatenated after this common header.

The Heartbeat

Generally, an ADU loss is detected by finding a gap in the sequence space. However the last ADU may be dropped. So each sender sends a low-rate, periodic message (a heartbeat) announcing the highest sequence number sent. The sender report (SR) packets are used to achieve this functionality. Additionally, a special heartbeat packet (fig. 4.2) is defined to provide a fast reaction when a sender stops or interrupts a data transmission. It contains following fields:

SubType (ST): 5 bits
: The type of the control subpacket. For heartbeats $ST = 00000$.

0	8	16	24	31
ST = 0	unused	Sequence Number		

Figure 4.2: The SRM heartbeat control packet

0	8	16	24	31
ST = 1	Count	reserved		
Sender's Source ID				
Sequence Number		Sequence Number		
......				

Figure 4.3: The SRM NACK packet for scattered sequence numbers

Sequence number: 16 bits
The sequence number of the last ADU sent.

Since heartbeats are important to detect lost packets, three heartbeat packets are sent after each original data transmission. Those three heartbeats are sent respectively T, $2 * T$ and $8 * T$ seconds after the emission of the last ADU. In the implementation, T is set to one second, but it could depend on the application (T should be small for interactive applications) and/or the number of members of the group. This mechanism is reseted every time an ADU is sent.

The Negative Acknowledgment (NACK)

There are two types of NACK packets defined: One is optimal when the sequence numbers to acknowledge negatively are not continuous, the other is used for a whole span of sequence numbers.

A – NACK FOR SCATTERED SEQUENCE NUMBERS
The header format is shown in fig. 4.3. The following fields are defined:

Sub Payload Type (ST): 5 bits
The type of the control subpacket. In this case, $ST = 00001$.

Count: 11 bits
$(Count + 1)$ defines the number of NACKS (sequence numbers) in this packet.

Senders Source ID: 32 bits
The source ID of the original sender from whom we have lost a packet.

Sequence number: 16 bits
The sequence number(s) of the lost packet(s).

0	8	16	24	31
ST = 2 |	Count		Sequence Number	
		Sender's Source ID		

Figure 4.4: The SRM NACK packet for spans

0	8	16	24	31
ST = 3 |		reserved		
		Timestamp		

Figure 4.5: The SRM timestamp query packet

B – THE NACK PACKET FOR SPANS
The header format of this packet is defined in fig. 4.4. The following header fields are defined:

Sub Payload Type (ST): 5 bits
The type of the control subpacket. In this case, $ST = 00010$.

Count: 11 bits
$(Count + 1)$ defines the number of consecutive missing ADUs.

Sequence number: 16 bits
The sequence number of the first lost ADU.

Sender's Source ID: 32 bits
The source ID of the original sender from whom the ADUs have been lost.

It should be noted, that although the nack packet with spans is defined, SRM doesn't support the accompanying mechanism well: Even if big gaps are detected, each packet gets its own NACK timer assigned. Even when several of those timers expire at the same time, it is not likely that there will be many spans of continuous sequence numbers.

Both type of nack packet are sent according to SRM's algorithms (see 4.1.2).

The timestamp queries

The timestamp queries are used in conjunction with the timestamp replies to compute host-to-host delay. The packet format is shown in fig. 4.5. The following fields are defined:

ST: 5 bits
$ST = 2$ defines a timestamp query

timestamp: 32 bits
The timestamp is composed of the 32 middle bits of a 64 bits NTP timestamp. The 16 first bits encode the seconds and the later 16 bits encode the fraction.

0	8	16 24 31
ST = 4	Count	reserved
Sender's Source ID 1		
Last Timestamp received (LTR_1)		
Delay since last Timestamp Request ($DLTR_1$)		
......		
Sender's Source ID N		
Last Timestamp received (LTR_N)		
Delay since last Timestamp Request ($DLTR_N$)		

Figure 4.6: The SRM timestamp reply packet

Timestamp queries are multicast to the whole group:

- Just after joining a SRM session.
- every five RMFP control interval (the RMFP control interval is computed as specified according to the RTCP interval defined in the RTP specification [25]).

The timestamp replies

The timestamp replies are sent as response to a received timestamp query for distance estimation purposes. The packet format is shown in fig. 4.6. The timestamp reply can contain several sub-chunks, each as a response for a received timestamp query. The following fields are defined:

ST: 5 bits
$ST = 3$ defines a timestamp reply

Count: 11 bits
The number of timestamp reply chunks that are contained in this timestamp reply packet.

$Sender'sSourceID_n$**:** 32 bits
The source ID of the sender of the corresponding timestamp query.

LTR_n **(last time-stamp received from** $Sender_n$**):** 32 bits
The timestamp field of the last timestamp query from $Sender'sSourceID_n$

$DLTR_n$ **(Delay since last time-stamp request from** $Sender_n$**):** 32 bits
The delay, expressed in 1/65536 seconds, since the last timestamp query has been received from $Sender'sSourceID_n$.

Note:

- It might be more efficient to unicast those replies directly to the source of the corresponding timestamp request. In fact, the information of the timestamp replies are only useful to the source of the corresponding timestamp request.
- Since the distance estimation is crucial for the setting of the timers and thus for the performance of SRM, it is suggested to smooth the estimated values.

Chapter 5

The Local Group Concept

This chapter describes the Local Group Concept (LGC) [14] and specifies a protocol profile to integrate LGC into RMFP. A stand-alone implementation of LGC is available at the university of Karlsruhe as the Local Group Multicast Protocol (LGMP) [24]. LGMP and the LGC profile for RMFP differ in some aspects, since the specification of LGMP was not available, when the LGC profile has been developed. Another source of differences are the different protocol architectures: LGMP is designed following the layered protocol architecture as a stand-alone transport protocol, whereas the LGC profile as part of RMFP follows the ALF principle (see section 2.3).

5.1 Overview

LGC defines algorithms to enable scalable, reliable multicast (one-to-many-communication). These mechanisms can be used to define a reliable multicast transport protocol.

LGC tries to reduce the implosion problem by distributing the processing of receiver status information. The basic principle of LGC is the definition of a hierarchy of subgroups. In each subgroup a Group Controller (GC) is responsible to process the status reports from its members. It then computes a single control message with the necessary state information of its subtree (the hierarchy of subgroups below the GC) and forwards it to the multicast-sender or a higher-level GC. Thus the sender (which is also the highest-level GC) and the GCs process the receiver status information in parallel. The processing load for each GC is limited with the number of members in its subgroup. Each receiver can become a GC. An example for such a groupstructure is showed in fig. 5.1.

Each GC, except for the sender, is member of two groups: It belongs as a simple member to a group controlled by a higher-level GC and it will send its control information to this higher-level GC. Additionally, the GC is also member of the group it controls. The GCs are thus the links between the different levels of group hierarchy.

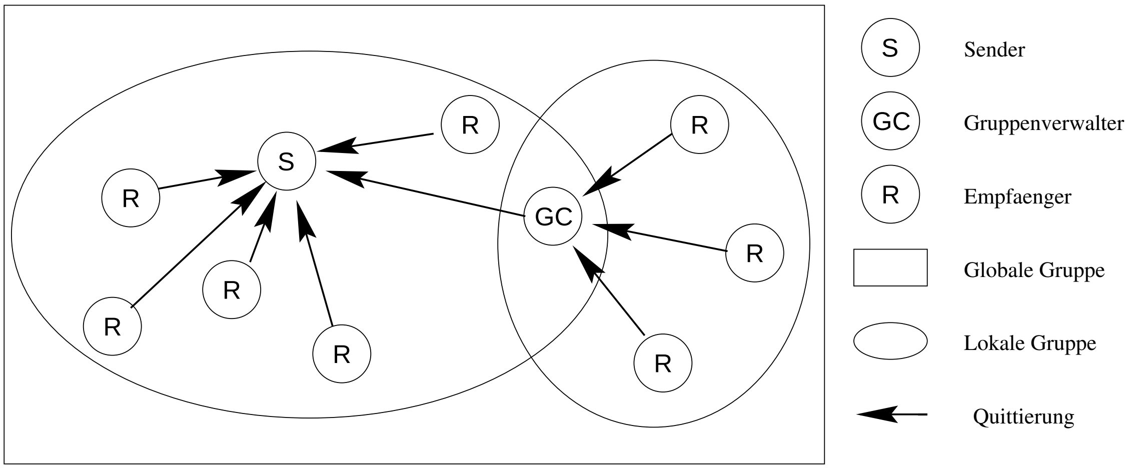

Figure 5.1: The group hierarchy in LGC

The group structure also enables the local recovery of packet losses: If a loss is detected in a group, the GC retransmits the packet itself or asks the group members that received the packet to perform the retransmission. Only if the whole group has lost the packet, the GC sends a retransmission request to the higher-level GC.

A key element to the efficient use of LGC is the group structure. Since network conditions (e.g. by congestion) and session membership (join and leave) are dynamic, the group structure has to be adaptable. The functionality to establish an appropriate group structure is provided by the Dynamic Configuration Protocol (DCP). The DCP is not part of LGC, but is used by LGC. An implementation of DCP is available at the University of Karlsruhe, Germany [13].

The mechanisms of LGC and DCP are fault-tolerant, i.e., failing hosts, whether GC or receivers, don't endanger the reliability of transmission to other group members.

5.2 LGC as Protocol Profile

This section discusses LGC in respect to the integration into RMFP. This means, that design issues deliberately left implementation dependent in the LGC specification have been adapted to RMFP and its environment.

5.2.1 Mechanisms

Joining and leaving local groups

Local groups are joined on the initiative of the receiver and are similar to the IP-multicast model: The session members choose a GC with the help of DCP and begin to send their periodic status control packets to that GC. When a GC receives a control packet at its unicast port from a session member that was not registered in its local group, it will add the member to its group.

Similarly, when a GC receives no control packets from one of its members for a given time-span, it will remove the group member from the local group. To avoid the erroneous removal of a member, the time-span has to be long enough to cover several lost packets in sequence. It is suggested to set the time-span to be bigger than three times the RTT between the member and GC (see section 5.2.1).

Loss detection

Each receiver or group controller detects packet losses through gaps in the sequence space. The sequence numbers are obtained from received ADUs and sender report packets. The GC additionally detects lost packets by analyzing acknowledgment packets from its group members.

Acknowledgment scheme

Every receiver and GC propagates state information periodically to their superior GC or to the sender (representing the root GC) in the form of *acknowledgment* packets. These packets are used to indicate the state of receipt of the group member, and, if the group member is a GC itself, the state of receipt of the tree below the group member. Each ADU has this state information assigned, so the actual acknowledgment packet consists of the list of state information for every ADU.

A group member M transmits its control information to its GC with the period $TACK_M$. To avoid redundant retransmissions, $TACK_M$ has to be bigger than RTT_M, the round-trip time between M and its GC:

$RTT_M < TACK_M < RTT_M + e, e > 0$

LGC distinguishes four different types of acknowledgments:

- positive ACKnowledgment (ACK)
- Negative ACKnowledgment (NACK)
- Semi-Positive ACKnowledgment (SPACK)
- Semi-Negative ACKnowledgment (SNACK)

Receivers use only ACKs and NACKs. A GC can send all four forms of packets to its own GC to indicate the state of receipt regarding a given ADU:

- An ACK indicates that ALL group members have received the ADU, including the GC (the sender of the acknowledgment) itself and all receivers in the subtree.
- A NACK indicates that NO group member has positively acknowledged the ADU so far and the GC itself has lost the ADU, too.
- A SPACK indicates that the GC itself has received the ADU, but other members of its group or further down the tree have not (or at least have not acknowledged positively yet).
- A SNACK indicates that the GC itself has lost the packet, but other members of its group have received it.

Loss recovery

LGC defines two different operation modes for loss recovery. One, the load-sensitive mode, is optimized to reduce the bandwidth used, and the other, the delay sensitive mode, to is optimized for minimal delays.

Load-sensitive mode: During the time interval $TRETR_G$ a GC G collects the reports from its group members. Every ADU acknowledged as not received (by NACKs) or detected as lost at the GC is scheduled for loss recovery. When the timer $TRETR_G$ expires, the GC starts the loss recovery process:

- If the GC itself has received a requested packet, it multicasts it to the local group or unicasts it to the members that suffered the loss, if the number of retransmission requests is below some threshold value.
- If it has not received the packet, but any of its group members have, it will unicast a REPAIR request to that member. If this repair algorithm is not successful after three times, the GC will acknowledge the packet negatively in its next acknowledgment packet to its higher-level GC and thus requesting a retransmission.
- If neither the GC nor any of the group members have received the packet, the GC will acknowledge the missing packet negatively in the next acknowledgment to its higher-level GC and thus requesting a retransmission.

The setting of the $TRETR_G$ timer is important to avoid duplicate retransmissions: If the timer expires too fast, positive acknowledgments as reaction to the receipt of a retransmitted ADU may not have made their way back to the GC in time, and thus the ADU is retransmitted again. LGMP uses following formula:

$TRETR_G = \max_{M \in localgroup}\{TACK_M\} + e_2$
$= \max_{M \in localgroup}\{RTT_M + e_1\} + e_2, e_1, e_2 > 0$

During the next $TRETR_G$ interval, retransmission requests for ADUs that are in the recovery process do not lead to loss recovery actions to avoid unnecessary retransmissions. However, ADUs lost at the GC are processed again until they are received.

The generation of acknowledgment packets depends not on earlier error recovery action, but only on the latest received acknowledgment packet of each group member and the GC's own state of receipt.

Delay-sensitive mode: In addition to the periodically transmitted acknowledgment packets receivers transmit acknowledgment packets immediately when they detect a loss. They also reset their TACK timer in that case.

When a GC receives a retransmission request (a NACK or SNACK) for an ADU, it tries to satisfy the request immediately:

- If the GC has the requested ADU, it will multicast it to its group.
- If the GC does not have the requested ADU, but it has already received a positive acknowledgment (a ACK or SPACK) from group members, it multicasts a REPAIR packet to the group.

  The members that have received the ADU react on the REPAIR packet scheduling the ADU for retransmission. The retransmission is delayed for a random time-span, that is dependent on the distance of the group member to the GC. This mechanism is used to suppress redundant retransmissions of the ADU. The mechanism uses the same formula to set the retransmission timers as the SRM profile (section 4.1.3).
- If the GC does not have the requested ADU and it has not received any positive acknowledgment (ACK or SPACK) from its group, it immediately acknowledges the ADU negatively to its superior GC and resets its TACK timer.

When the GC does loss recovery actions for an ADU, an *ignore* timer with the duration of the GC's TACK interval is set up for that ADU. Until the timer expires, further retransmission requests for the ADU don't lead to additional loss recovery actions. This ignore timer has the same function as the retransmission timer of the load-sensitive mode and is set in the same way. However, in the load-sensitive mode there is only one retransmission timer for all ADUs, whereas in the delay-sensitive mode every ADU is timed separately.

Host failure detection and recovery

LGC uses, like SRM, a receiver-initiated model of the group membership, and the group membership itself is only relatively loose and bases on the IP-multicast group model. Thus, LGC's mechanisms are fault-tolerant in respect to the receivers and GCs.

Receiver GCs detect the failure (or dropping out off the group) of members by means of timeouts of the *alive* timers. Every member has a alive timer assigned. Every time a GC receives a packet from a member, that members alive timer is reset. The duration $TALIVE_G$ of the timer is set dependent on the RTT between the GC and the member M:

$TALIVE_M > 3 * RTT_M$

If a member M is quiet for a longer duration than $TALIVE_M$, it is removed from the GCs member table.

There is no special mechanism for a receiver to leave a group explicitly.

Group controller If a GC leaves the session, whether by failure or intentionally, this is detected by the DCP by means of a timeout. The DCP notifies the members of that GC to join another GC. The necessary parameters like addresses are provided by the DCP.

In this situation it is possible that active session members are temporarily not registered in the group structure until they join another group. Thus it is possible that ADUs are positively acknowledged by all session members registered in groups, but unregistered members have not received them yet.

To avoid unrecoverable packet loss in this situation the sender application has to wait to free the retransmission buffers. To reduce the error recovery time, it is useful to delay the release of the retransmission buffers also at the GCs or even at simple receivers. The sender multicasts a *free* packet on the sessions control flow. If a session member changes the GC, it has the guarantee, that every GC has a copy of all data packets it possibly misses.

The management of the retransmission buffers is up to the application. Thus the free packet is initiated by the sender application and evaluated by the receiver applications.

5.2.2 Flows and Addresses

Due to the hierarchical structure of LGC original and retransmission data is always sent on different flows:

- The sender transmits the original data on an IP multicast address known to all session members (this is the RMFP data flow).
- Some retransmissions are multicast to the local groups. Each local group has its own multicast IP address.
- The other retransmissions are unicast to the receiver with the loss.

Control packets to the global group are sent on the RMFP control flow. Other controls are sent to the local groups or are unicast to single members.

The RMFP session packets are not processed by the LGC profile and are thus not further discussed in this LGC profile definition.

Since LGC needs more than just global groups for data, control and session information, as they are defined by RMFP, more addresses/ports are used. In the following there is a list of all address/port pairs used by a LGC member:

- *global address, RMFP data port*: Set up before the session. All original data is sent to this address/port. This is the session address/session port of RMFP.
- *global address, RMFP control port (= RMFP data port + 1)*: All control packets sent to the whole group are sent on this address/port. This address/port is set up by RMFP.
- *global address, RMFP session port (= RMFP data port + 2)*: All session packets sent to the whole group are sent on this address/port. This address/port is set up by RMFP.
- *global address, dcp port (= RMFP data port + 4)*: Used only by DCP. All DCP packets are sent to this address/port.
- *local group address, local group data port*: Each session member sets up its own local group address when joining the session. It will be used as address of the local group, if the session member becomes a GC. All retransmission packets for the local group are sent on this address/port.
- *local group address, local group control port (= local group data port + 1*: All control packets destined for the local group are sent on this address/port.
- *unicast address, unicast data port*: All retransmission data destined for a single session member are sent to its unicast address/data port.
- *unicast address, unicast control port (= unicast data port + 1*: All control packets destined for a single session member are sent to its unicast address/control port.

The local group address/port pairs and the unicast ports of the GCs are distributed to the session by the DCP. The GC knows the unicast addresses of its members from UDP.

5.2.3 The Data Flow

LGC uses the ADU format of RMFP without modifications. It uses the ADU fields in the same way as the SRM profile (see section 4.2.1).

5.2.4 The Control Flow

The control flow is composed of the RMFP control packets (section 3.3.3) and the LGC control packets (*free*, *repair*, *echo*, *diag* and *acknowledgment* packets).

- The free packets are used to indicate that a receiver can free its retransmission buffer up to the contained sequence number. Due to the ALF principle they are actually not profile controlled. The buffer management is up to the application, and so this mechanism is also application controlled.

<table>
<tr><td>0</td><td>8</td><td>16</td><td>24</td><td>31</td></tr>
<tr><td>V |P| CC</td><td>Payload Type</td><td colspan="3">Length</td></tr>
<tr><td colspan="5">Source ID</td></tr>
</table>

Figure 5.2: The common control header

- The repair packet is sent from a GC to a receiver (unicast transmission) to demand the retransmission of the specified ADUs from that receiver.
- The echo packets are used to calculate round-trip times between session members.
- The diag packets are sent to indicate special conditions.
- The acknowledgment packets are used to indicate the status of ADUs. If the acknowledgment packet is transmitted from a receiver, each ADU is whether positively or negatively acknowledged by an ACK and NACK respectively. If the acknowledgment packet is transmitted from a GCT, the referred ADUs could be received by the whole group (positive ACKnowledgment (ACK)), by some group members including the GC (Semi-Positive ACKnowledgment (SPACK)), by some group members excluding the GC (Semi-Negative ACKnowledgment (SNACK)) or by no group member at all (Negative ACKnowledgment (NACK)).

Every control packet shares the same header, and several control packets may be concatenated and sent in a single UDP packet with only one occurrence of this common header (at the beginning of the UDP packet). The format of the common header is shown in fig. 5.2. The following fields are defined:

Version (V): 2 bits
Identifies the version of RMFP.

Padding (P): 1 bit
The padding flag is not used for the control packets, since all control packets are aligned by definition.

Chunk Count (CC): 5 bits
The profile specific bits contain the number of control packets contained in this aggregate LGC control packet.

Packet type (PT): 8 bits
Contains the constant 206 to identify this as an LGC control packet.

Packet length (length): 16 bits
The length of this LGC control packet in multiples of 32-bit words minus one, including the header and all control packets concatenated by this header. The calculation algorithm is the same as for the RMFP length fields.

0	8	16	24	31
ST = 0	unused	Sequence Number		

Figure 5.3: The free subpacket

Source ID: 32 bit
The source ID of the sender of this aggregate control packet.

The free packet

The sender application initiates the sending of a free packet to the session members to allow the members to free their retransmission buffers. It should be computed in a way so that every member has acknowledged all sequence numbers prior to the given one positively. The format of the free packet is shown in fig. 5.3. The following fields are defined:

Subtype (ST): 5 bits
The type of this control packet. $ST = 00000$ denotes a free packet

Sequence number: 16 bits
Every packet with a sequence number less or equal may be removed from the retransmission buffer.

The repair packet

If a GC detects that it has lost an ADU, and it has information that at least one of its members has received the ADU, it tries to initiate the retransmission of the packet within its group. The GC will send a repair packet containing the sequence number of the lost packet.

A GC sends a repair packet to its group or to a single group member, depending on the mode of operation. If operating in delay-sensitive mode, the packet is sent to the group and if operating in load-sensitive mode, the packet is sent to a single group member.

A repair packet asks the receiving group member to perform a retransmission of the requested ADUs (indicated as a range of sequence numbers). If a member receives a repair packet on its unicast port (load-sensitive mode), it will retransmit the ADUs immediately. If it receives the repair packet on the group port, it will delay the retransmission randomly in a similar way like SRM: Since the repair packet is multicast to the group, every member schedules the indicated packets for repair. The member with the shortest delay will perform the retransmission. The other group members will cancel the retransmission, when they receive the requested packets.

There are two different types of repair packets, that have different encodings of the sequence numbers to retransmit:

0	8	16	24	31
ST = 1 |	count	Sequence Number		

Figure 5.4: The repair subpacket with span

0	8	16	24	31
ST = 2 |	Count	Sequence Number		
Sequence Number		Sequence Number		
......				

Figure 5.5: The repair packet with a list of sequence numbers

A – REPAIR PACKET WITH SPAN
The repair packet with spans is used to request a series of continuous sequence numbers. The format of the packet is shown in fig. 5.4. The following fields are defined:

Subtype (ST): 5 bits
The type of the part. $ST = 00001$ denotes a repair packet with span.

Count: 11 bits
Number of ADUs to be retransmitted in sequence.

Sequence number: 16 bits
The sequence number of the first ADU of the sequence to be retransmitted.

B – REPAIR PACKET WITH SINGLE SEQUENCE NUMBERS
This kind of repair packet are used to request one or more sequence numbers for retransmission, that are not continuous. The format of the packet is shown in fig. 5.5. The following fields are defined:

Subtype (ST): 5 bits
The type of the control packet. $ST = 00010$ denotes a repair packet with single sequence numbers.

Count: 11 bits
Number of sequence numbers contained in this packet.

Sequence numbers: 16 bits
Each sequence number denotes an ADU that is to be retransmitted.

0	8	16	24	31
ST = 3 |	reserved			
Timestamp				

Figure 5.6: The echo request packet

0	8	16	24	31
ST = 4 |	Count	reserved		
Sender's Source ID 1				
Last Timestamp received (LTR_1)				
Delay since last Timestamp request ($DLTR_1$)				
......				
Sender's Source ID N				
Last Timestamp received (LTR_N)				
Delay since last Timestamp request ($DLTR_N$)				

Figure 5.7: The echo reply packet

The echo packets

The echo packets are used to determine the RTT between hosts. Each host has to know the RTT to its GC. A GC has also to know the RTT to each of its group members. There are two forms of echo packets:

A – The echo request packet

The echo request packet is used to request an echo reply packet. Those packets are used to estimate the RTT between a GC and its members. The packet format is shown in fig. 5.6. The following fields are defined:

ST: 5 bits
 $ST = 00011$ defines an echo request.

Timestamp: 32 bits.
 The timestamp is composed of the 32 middle bits of a 64 bits NTP timestamp. The 16 first bits encode the seconds and the later 16 bits encode the fraction.

B – The echo reply packet

The echo reply packet is sent as an answer to an echo request packet. The format of the packet is shown in fig. 5.7. It has following fields:

0	8	16	24	31
ST = 5	Code	Code specific		

Figure 5.8: The diag packet

ST: 5 bits
$ST = 00100$ defines an echo reply packet

Count: 11 bits
The number of timestamp reply trunks that are contained in this echo reply packet.

$SourceID_n$**:** 32 bits
The sender of the request corresponding to this reply trunk.

LTR_n **(last time-stamp received from** $sender_n$**):** 32 bits
The timestamp field of the corresponding request from $sender_n$

$DLTR_n$ **(Delay since last echo request from** $sender_n$**):** 32 bits
The delay, expressed in 1/65536 seconds, since the request corresponding to this reply trunk was received from $sender_n$.

To reduce network load the echo packets are sent together with acknowledgment packets, when sent in direction from member to GC.

The RTT is estimated the first time, when a new member joins a group. The member will add an echo request to its first acknowledgment packet. In this case, the GC answers this request immediately with an echo reply packet and encloses its own echo request. The new member will then answer the GCs request immediately.

The RTT is calculated in the same way as at the SRM profile (section4.1.4). When t_1 is the sending time of the request, t_2 is the receipt time of the reply, and d is the delay between receipt of the request and sending of the reply, then the RTT is estimated with:
$RTT = (t_2 - t_1 - d)$

The diag packet

The diag packets are used to indicate special conditions like errors. The format of the diag packet is shown in fig. 5.8. The packet has following fields:

SubType (ST): 5 bits
The type of the part. $ST = 00101$ denotes a diag packet.

Code: 11 bits
Diagnostic code, describes the special condition that lead to the diag.

0	8	16	24	31
ST = 4 |	Count	Sequence Number		
T T| Span	T T| Span	T T| Span	T T| Span	
		......		

Figure 5.9: The acknowledgment packet

Length: 16 bits
: Length of condition specific information following the subtype header. It is computed in steps of four octets (alignment).

The following diagnostic codes are defined:

1: Data not available
: A GC or a receiver indicates that requested data is not available.

The acknowledgment packet

LGC distinguishes four different types of acknowledgments:

- positive ACKnowledgment (ACK)
- Negative ACKnowledgment (NACK)
- Semi-Positive ACKnowledgment (SPACK)
- Semi-Negative ACKnowledgment (SNACK)

Since a receiver or GC has to acknowledge every packet in some way (contrary to e.g. to SRM, which uses only NACKs), it is most efficient to have only one packet type for acknowledgments containing the information for every ADU in the sequence space of interest. The sequence space is divided into spans. Each span denotes a range of sequence numbers, starting with the highest sequence number of the last span +1. The highest sequence number of the span is computed by adding the number of ADUs of the span.

The format of the acknowledgment packet is shown in fig. 5.9. The following fields are defined:

SubType (ST): 5 bits
: The type of the part. $ST = 00110$ denotes an acknowledgment packet

Count: 11 bits
: Number of spans.

Sequence number: 16 bits
: All ADUs with sequence numbers less or equal are positively acknowledged, and the ADU with the next sequence number is not positively acknowledged.

TT: 2 bits
Identify the acknowledgment type of the ADUs denoted by the following span:

00: ACK

01: SPACK

10: SNACK

11: NACK

Span: 6 bits
Number of packets in the span.

The packet is in always 32 bit aligned. If the number of spans is not a multiple of four, some octets might be unused.

Chapter 6

Implementation

6.1 Overview

This chapter describes the implementation of RMFP. The implementation provides the functionality of RMFP and contains also an implementation of the SRM profile.

One of the basic principles of RMFP is the tight integration of the protocol functionality into the application. Thus, an implementation of RMFP within the operating system's kernel or as a special process is not applicable. Therefore this implementation is designed as a link library and the complete protocol code runs within the address space of the application. The programming interface (API) of the RMFP library consists of method calls and method upcalls.

As a consequence of this decision, the only network protocol usable is UDP/IP. The Internet protocol stack is the most common protocol stack today and available on almost all machines, from PC up to mainframes.

Since the protocol was not to be implemented in the kernel and should be used without constraints for the applications, UDP was the only possible protocol. Raw IP would have required *root* permissions for the applications on Unix machines, while TCP cannot be as bases for reliable multicast at all.

The programming language for the library is C++. The object-oriented approach for the implementation is useful for specifying interfaces, a feature that was used both at the API and at the internal interface to the profiles. This reduces the effort to add new profile implementations. Additionally, C++ enables the use of templates and dynamic binding, both features used within the implementation, and delivers executables with good performance.

Besides of the RMFP library, there has been another library developed. This *Toolkit* library contains general data types that are used within the RMFP library.

6.2 Environment

The library was developed mainly on workstations with the Solaris operating system version 2.5 of SUN. During the testing of the implementation it has been verified, that the library also works correctly under the operating systems Digital Unix 4.0 and Linux 2.0.33. Digital Unix is a product of Digital Equipment Corporation (DEC) and runs on the 64-bit Alpha architecture. Linux is a public domain Unix system various systems. The platform used with Linux for the tests has been a PC.

The compiler used under Solaris has been the CC of the SUN-Workshop developing environment. On both Digital Unix and Linux GCC, the public domain C and C++ compiler of the Free Software Foundation (FSF), has been the choice.

6.2.1 Interoperability

All three platforms have different memory architectures. This means, that the same data structures have different memory representations. The SUN platform (SPARC) has a memory word size of 32 bits, and the bytes in such a word are ordered in the big-endian manner. DEC's Alpha platform has a word size of 64 bits and uses the little-endian manner and the PC platform has a word size of 32 bits and little-endian byte order.

The proper interoperation of instances of the protocol library running on different platforms requires a common format of the packets. All fields have the same size on all architectures, and the byte-order has to be the same. RMFP uses the standard byte-order used on the Internet, i.e. the big-endian byte-order. Thus, protocol instances on little-endian platforms have to change the byte-order before sending and after receiving the packets.

6.2.2 Installation of the Library

The library comes as C++ source code and is organized in several directories. The `ToolkitLib` directory contains the Toolkit library, and the `src` directory contains the RMFP library and the SRM profile code. The two test applications are located in the `app` directory: `app97` checks the functionality of the RMFP library and sends dummy data; `ft` is a simple file-transfer application. The ft application is shown later as example in section 6.6. When the libraries are built, they are put into the `lib` directory. The binaries of test applications are put into the `bin` directory.

To allow the compilation of the source code for several platforms in the same source directories, all system dependent files are put into separate subdirectories, that have the name of the system (solaris, du, linux). The system dependent file types are the dependencies files (ending in `.d`), the object files (ending in `.o`), the libraries (ending in `.so`) and the binaries.

To build the library, the the source code has to be extracted from the archive inside the directory designated to hold the source tree. This directory is called the *RMFP root* directory in the rest of this section.

Before building the libraries and the binaries, the makefiles have to be checked, if the `PROJDIR` variable is set correctly. It has to contain the path to the RMFP root directory. The makefiles that have to be checked are the system dependent Makefiles in the directories containing the source code for the different libraries and applications. Those system dependent makefiles have all the name `Makefile.<system>` and in each source directory is a makefile for each supported system.

To build the distribution the GNU make utility has to be available. It is assumed, that it has the name `gmake`. It is called with the name of the system as parameter in the root directory, e.g. `gmake solaris`.

The single parts of the distribution can also be built separately. E.g. to build just the Toolkit library for Solaris, the necessary command is `cd ToolkitLib ; gmake -f Makefile.solaris`.

To use the libraries, it is necessary to include them into the `LD_LIBRARY_PATH` variable. The example for Solaris:

```
setenv LD_LIBRARY_PATH <RMFP-Root-Dir>/lib/solaris
```

To use the binaries, the `PATH` variable has to be changed (again for Solaris):

```
setenv PATH ${PATH}/bin/solaris
```

6.3 Design Guidelines

The primary goal of a RMFP implementation is to provide applications with a flexible and efficient transport system. The solution chosen for RMFP is the tight integration of the transport protocol into the application. This is done according to the ALF principle (section 2.3). Many aspects of an RMFP implementation, however, are not specified by RMFP. This section describes some important design guidelines that have been applied.

6.3.1 Thread of Control

User space implementations of transport protocols are often implemented as own processes (e.g. SandiaXTP [26] and XTPlite [15]). Other implementations are designed as link libraries and share a single process with the application, but use *multithreading* [18]. Threads, sometimes called *light-weight processes*, enable the parallel execution of functions in a single process. An example of such a multithreaded protocol implementation is LGMP [24].

Transport protocols implemented as processes or with multithreading allow the temporal decoupling of the protocol's and the application's operations. This simplifies the use of the transport protocol's API.

However, this decoupling limits also the control the application has over the protocol. The threads of the transport protocol perform their operation without the temporal synchronization with the application.

The intended tight integration of RMFP into the application lead to the decision to do without multithreading in the library. The application has to give the control to the library, so that the protocol operations are performed. The library gives information to the application to enable exact timing and to avoid polling. Another possibility is provided to allow the application to give the control completely to the RMFP library. The control flow will be returned in an event model. A description about these mechanisms at the API is given in section 6.5.3.

6.3.2 The Minimal-copy Architecture

One of the bottlenecks of protocol stacks is the handling of the data [4]. This can be either copying the data and processing the data, e.g. to compute checksums and to do the presentation conversion. The copying of the data is usually necessary to move the data into and out of the operating system kernel. The copying of the data is usually also performed at the interface between the protocol stack and the application to simplify the interaction. For some protocol stacks, the two locations of copying fall together. E.g. the TCP/IP protocol stack is mostly implemented in the kernel and many applications don't use further protocol layers or have them integrated in the application. If parts of the protocol stack are implemented in the user space, it is normally necessary to copy the data twice.

Another source of copying operations are buffers in the protocol stack. E.g. retransmission buffers are used at the transport protocol, to keep the data after having been sent to perform loss recovery.

This implementation of RMFP tries to avoid unecessary copying and processing of the transmitted data at the transport level. The ADUs are framed by the application in its buffers, and the send operations of the RMFP library copies them directly into the kernel address space. The received ADUs are copyied out of the kernel address space into buffers provided by the application.

Furthermore, the ADUs are not copied into any buffers in the RMFP library. Following the ALF principle, the buffering of the ADUs for retransmission, ordering and similar tasks that are generally part of the transport protocol are performed by the application. The application has more possibilities to optimize these tasks, since it knows about the *semantics* of the ADUs.

6.3.3 Flow and Congestion Control

The implemented mechanisms to achieve flow and congestion control are only intented to support the application. Flow and congestion control depend heavily on the application requirements and have to be controlled by the application. For that reason, the RMFP specification does not include flow or congestion control except for the definition of a congestion signal (the `lossrate` field of the receiver's report packets), but leaves it to the application or profiles to adopt the sending rate.

The minimal-copy architecture of the library would make it also difficult to implement a buffering mechanism at the library that is necessary to delay the transmission of ADUs, when the ADU sending rate of the application is too high.

The design of the library leaves it to the application to decide, if and when ADUs are to be sent. If retransmissions are due, the library issues an upcall to the application and the application can perform the retransmission right away, but it is also possible that the application does not do the retransmission at all or delays it. The sending of the sender's and receiver's report packets is also initiated by the application. Thus, using the lossrate information that is part of the receiver's report packets as congestion signal, the application has the ability to control the rate of ADUs and RMFP control packets.

The open question is the behavior of the profiles. The profiles define their own control packets and control their transmission. If some kind of flow and congestion control is to be used also for those control packets, it is up to profile developpers to design it. If there is no such kind of control implemented, the only possibility to slow down the sending rate of controls is indirectly by slowing down the ADU rate.

The SRM profile does not implement functionality to control the sending rate of SRM control packets, since SRM defines only an application controlled rate-control mechanism and makes no suggestions about further flow and congestion control mechanisms.

6.4 The RMFP Library's Design

In an abstract view of the RMFP library, there can be three parts distinguished (see fig. 6.1):

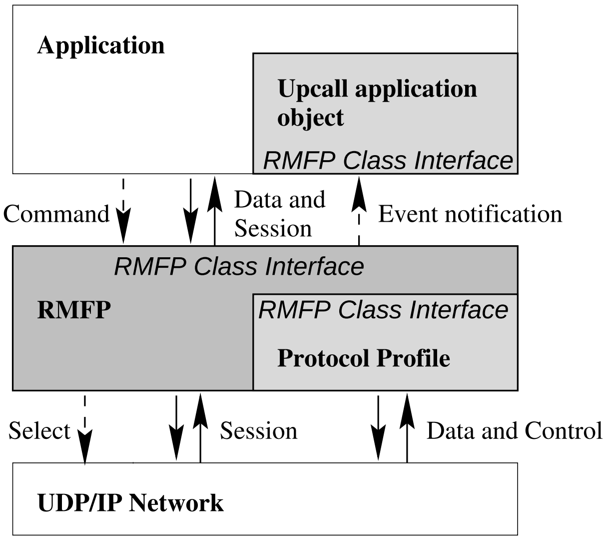

Figure 6.1: High-level structure

The RMFP framework: This provides the functionality defined by the RMFP specification (section 3) and includes the interface specifications both to the application and to the profiles.

The upcall application object: This object is used to execute the upcalls from the library. The interface of this class is specified by the RMFP part of the library, but the functionality has to be implemented by the application.

The protocol profiles: All profiles derive their main classes from interface classes specified by the RMFP part of the library. Thus all profiles share the same interface. The protocol developpers have to implement their protocol's functionality below this interface.

6.4.1 The Classes

The following description of classes is separated into three categories:

1. The RMFP classes that include the packet formats and the classes specifying the interfaces to the application and the protocol profiles.
2. The SRM classes that implement the functionality of the SRM protocol profile. Most of the SRM classes are derived from RMFP's classes specifying the profile interface.
3. The Toolkit classes. These classes form a separate class library that implement basic functionality, e.g. abstract data types. The Toolkit classes are usable not only with the RMFP library, but also in other programs. For that reason they have been put into a separate library.

The RMFP classes

The Session class: The `rmfpSession` class is the main part of the API. The application instantiates an object of this class for each session it wants to join. The session object is initialized by a instance of the `rmfpSessionPars` class and a pointer to a `rmfpApplication` instance. During the session, the application will call the methods of the session object to issue commands, and the session object will call methods of the `rmfApplication` object to notify the application of events (upcall model).

This class can be regarded as the heart of RMFP, since it controls the communication with the application and the profiles and contains the control loop that initiates every action.

There are some auxiliary classes:

- The `rmfpNetControl` class provides some functionality for the use of filedescriptors.
- The `rmfpMemberTable` class provides the management for group members for both the application and the profiles. It uses a combination of hashtable and linked list data types.

- The `rmfpEventHandler` class dispatches upcalls. Every function that wants to issue an upcall, has to send an event to this event handler. The event handler will invoke the specified upcall method in the `rmfpApplication` object.

The Application class: The `rmfpApplication` class can not be instantiated itself, but is used as an interface specification. The application has to derive an own application class and to implement the interface methods. The library will call the methods of this class as upcalls to the application. Each session is associated with exactly one `rmfpApplication` object.

The Profile class The `rmfpProfile` class is intended as a interface class for the profiles. Each profile to be integrated into RMFP has to derive an own class and implement the specified methods. The `rmfpSession` object can than interact with each possible profile in the same way. Thus, the integration of a new profile doesn't need much changes at the `rmfpSession` class.

The derived classes of this class are intended to be the main classes of the profile implementations.

The Member class: By the `rmfpMember` class, the library offers both the application and the profiles management functionality for the group membership. The application can ask for the list of group members and thus avoid implementing this functionality. Additionally this class provides the foundation for the member management at the profiles. They can derive this class to provide the functionality they need. Thus, a single member management can be used for the application and the profiles.

The Event classes: Events are used inside the RMFP library to initiate upcalls to the application. Each upcall corresponds to an event class, and all event classes are derived classes of the `rmfpEvent` class. To initiate an upcall, an object of the corresponding event class is instantiated and sent to the `rmfpSession` object's `rmfpEventHandler` instance. The event handler will then issue the upcall.

The ADU-Container class: The `rmfpADUContainer` class is part of the API. ADU containers are used to simplify the access to ADUs. On the net an ADU is represented simply as a bytestream, and the receive operations on the sockets simply copy the whole ADU into a plain buffer. The ADU container can split this buffer into the logical components of an RMFP ADU: The header, the name, the data and the padding.

ADU containers are also used to send data. The application can put the parts of an ADU independently into a container.

The ADU-Entry class: The `rmfpADUEntry` is another interface class for the profiles and have to be derived by the profiles. The ADU entries store the ADU information necessary for the profile, like the ADU name, or the original header. They are used for error control purposes.

The Packet classes: RMFP defines several packet types. Each of those packets is represented by a corresponding class allowing access to the fields.

- The `PacketHeader` class defines the common part of all packets used by RMFP and its profiles. All other packet types derive from this class.
- The `rmfpADUHeader` class defines the header of the ADUs. This class extends the common `PacketHeader` by sequence number and object id. The ADU name is not part of this class, but is defined extra.
- The classes for the report packets (`rmfpSRHeader and rmfpRRHeader`). The two kind of report packets have each their own corresponding class that is derived from the common `PacketHeader`.

SRM profile classes

The class hierarchy for the SRM profile is shaped by the RMFP classes designed as profile interface. They are derived and extended to provide the functionality of the SRM protocol.

The Profile class: The `srmProfile` class is derived from `rmfpProfile` and is the controlling class of the profile. It keeps the hash table with the adu entries (`srmADUTable`), performs the send and receive operation on the net and represents the main interface to the RMFP part of the library.

The Member classes: The `srmMember` class is derived from `rmfpMember`. It stores all the necessary member information SRM needs to operate, e.g. round-trip times, source IDs, etc.

The `srmMember` class is not instantiated itself, but further derived and diversified to the `srmRemote` and `srmSelf` classes. The instances of those classes represent the state of the local session endpoint, i.e the own instance of the RMFP session object, and the state of the remote members of the session, respectively.

The class uses another class to provide the management of the ADU sent by the represented member:

- The `srmADUTable` class keeps track of all ADU entries (see below). The entries themselves are stored independently of their source in a hash table kept by the SRM profile instance. This class simply keeps track of the sequence number range of the ADUs from the member. Thus, the ADU-Entries can be easily accessed by the member object.

The ADU-entry class: The `srmADUEntry` class is derived from `rmfpADUEntry`. It tracks all necessary information for single ADUs, including the state. This class is responsible for initiating NACK packets and for asking the application for retransmission data.

The Packet classes: For all RMFP packet classes exist corresponding versions in SRM. They add no data fields, but a method to process received packets. There are also classes for the SRM specific control packets.

The `process` methods of these classes allow the recursive decomposition of the received composite control packets: To limit overhead bandwidth, RMFP and SRM allow the hierarchical concatenation of control packets into a single UDP packet. A UDP packet can contain several receiver report, sender report and SRM composite control packets. The composite SRM control packets consist again of a concatenation of SRM control subpackets.

The Control-assembler class: The `srmControlAssembler` classes provide the functionality to concatenate several RMFP and SRM control packets into a single UDP packet for transmission. In a first step, requests in form of SRM events are gathered, that request the construction of control packets. In the second step, the requests are processed and the hierarchically structured UDP packets are constructed.

The Event classes: The derived classes of `srmEvent` have nothing to do with RMFPs event classes. In the SRM profiles implementation they are mainly used to indicate the need to send a control packet of some kind. Since SRM and RMFP offer the possibility to send composite control packets, it is useful not to send requested packets immediatly, but to store the request (in form of the events) and defer the construction of the composite control packets.

The Toolkit classes

The Error class: The `Error` class defines a number of error codes together with their descriptions. If this class is used, e.g. to carry the return status of a function call, an descriptive text can be printed, without much effort.

The Number template class: The `Number` template class provides unsigned integers with the notion of wrap-arounds. Such numbers are used for the `Seq` type (sequence numbers, 16bit) and for the `Time32` class (32bit).

The Time32 template class: This class is used to provide all the timing information for RMFP and within the RMFP library. It complies to the timestamp fields of the RMFP specification, i.e. a timestamp is 32 bit wide. The upper 16 bit specify the seconds, the lower 16 bit the fractions of a second. This leads to a range of about 18 hours before a wraparound occurs, with a precision of approximatly 60 micro seconds. Because of the limited range, the instances of the `Time32` class are not suited to carry absolut timestamps for a longer duration then half the range, and are comparable only within this limits. For the use within the protocol to calculate roundtrip times and timeouts the range is big enough.

The Timer class: The `Timer` class provides the management functionality necessary to wait for several timouts at the same time. It uses the `Time32` class to calculate the timepoints. It does not use the operating systems `alarm` function, but checks for expired timeout only when polled.

Expired timeouts lead to the invocation of upcall methods, that are registered at the Timer during the insertion of an alarm.

To avoid inefficient, frequent polling to check for expired timeouts, the timer can be asked for the next timeout scheduled. The controlling instance can then use the systems timing facilities to wait. Within the RMFP library this is done with the `select` system call.

The HashTable template class: The `HashTable` template class implements a simple hash table. It is possible to define own `HashKey` classes, that are used to index the hash table.

The InAddr class: This class is used as a wrapper class around the Internet address classes.

The SocketIO class: The `SocketIO` class is a wrapper class to hide the systems socket interface.

The Network class: The `Network` class is used to abstract several sockets into a single functional unit. E.g. the LGC profile uses global groups, local groups and unicast to send data and control information. The `Network` class can combine e.g. all the sockets that are used to receive data. Thus the profile does not have to distinguish between different sockets while receiving data.

6.5 The API

6.5.1 Overview

The RMFP implementation is designed as a C++ class library. This object-oriented approach leads to a different API concept compared to common C language function interfaces.

To use the transport protocol functionality, the application has to *instantiate* class objects of several classes defined by RMFP. One class of RMFP (the *rmfpApplication* class) has to be derived by the application. The application has to implement the virtual functions specified by this class to receive the *upcalls* (see section 6.5.2).

When the necessary objects are instantiated and configured, the application can call methods of RMFP's class objects. RMFP and its profiles call the methods of the application's `rmfpApplication` object to notify the application about events.

To leave a multicast session the application has to destruct the class objects associated with that session.

The API defines several ways to synchronize the protocol functionality with the application's thread of control (see section 6.5.3). These mechanisms are necessary, since the RMFP implementation uses the same thread of control as the application. The application and the RMFP library share that single thread and the API provides the means to change the "owner" of the thread in a useful manner.

As a consequence of the class interface the complexity of the API is bigger compared to a more common C language interface, e.g. as it is used with the LGMP protocol [24]. However, this complexity is necessary to provide the full flexibility possible by the ALF concept.

The source code of an example application is commented in section 6.6.

6.5.2 Upcalls

Upcalls are a simple design principle at programm internal interfaces [5]. They are suited for programms consisting of several modules that communicate via interfaces and have more complicated communication patterns than simple, immediatlely returning function calls. An example is an application using a communication library, like the RMFP library.

In this example, the superior module, the application, calls functions of the inferior module, the communication library. However, the result of a simple function call of the library can require several independent of complex reactions by the application. An example with the RMFP library is the receipt of control packets: If the application is notified about the receipt of a control packet (by an active socket), it will call a *process* method of the session object representing the connection end point. The results of this call can be diverse: The application might have to retransmit some ADU, it might have to process a receiver or sender report packet, or maybe nothing is to do, if the reaction to the control packet is limited to the protocol profile. In RMFP the problems are even worse, since a received UDP packet can contain several RMFP and profile control packets that all require reaction of the application.

In the normal function call model, the information about the required action has to be traded from the inferior module (communications library) to the superior module (application) as return values of the function call. The application has then to process those return values and do the required operations.

The upcall model puts some of the functionality in deciding how to react to the inferior module. At the initializiation, the superior module configures the inferior module with a number of reaction functions, one for each class of expected reaction notifications. When the superior module gives the control to the inferior module to do some operations, the inferior module can call those upcall functions to notify the superior module of required reaction. If the superior module finishes the handling of this upcall, it returns the control to the inferior module, which can do more upcalls, or can return the control finally back to the superior module.

In the example of the RMFP library with a received control packet, the application calls the library's *process* function, when the control socket is active. The library then parses the received UDP packet. For each contained control packet, that requires the reaction of the application, an upcall is issued to the application. An example is a received sender report.

Upcalls are even more useful in RMFP than the previous example shows, since in normal use, the applications give the control completely to the library. To be able to make some progress, the application informs the library about conditions, when it wants to be invoked. This can be activity at some sockets or some timers set by the application. Each of the described events trigger an upcall function.

In the design and implementation phases of the application, the developers have to know about the syntax of the upcalls. In the C++ programming language they can be specified as class definitions with virtual methods. The application programmer derives a new class of this virtual class and implements the virtual methods to react on upcalls.

This is also the way the RMFP upcall interface has been designed. The virtual class that specifies the upcall interface is the *rmfpApplication* class. The application programmer derives a new class from this class and implements the upcall methods.

The application creates an instance of the new upcall class and intializes the connection endpoint (an object of the *rmfpSession* class) with this object. The library will call the methods of this object as upcalls.

Additionally RMFP uses a second, independent upcall mechanism for the use of its timer. If the application requires some timing, it can use the timer of the RMFP library (actually the timer is part of the *Toolkit* library that comes with RMFP). If a timer is set, the application has also provide an object of a class derived from the class *TimerObject*. If the timer expires, the timer will call an upcall method of this TimerObject.

6.5.3 Synchronization

The design of the RMFP implementation to use only a single thread shared with the application requires a practical mechanism to give the thread of control to the module that needs it. A multi-threaded approach would allow both the application and the protocol library to run in parallel. Threads that wait for some event can go to *sleep* to avoid wasting CPU load.

With the single threaded approach the scheduling of the CPU time has to be implemented at the API of the library: The application as the superior module has to be able to control the thread of control in some way. This means, that the library must not keep the control for a longer period, if the application does not wants this.

The solution applied at the RMFP library has been the event driven model: Every action at the library or at the application is realized as the reaction of an event. The scheduling of the events has to be managed with the help of the operating system, and if no events are available, the process with both the application and the RMFP library is put to sleep. This has to be done in a single operation. On UNIX operating systems this is generally the *select* system call. This system call allows the application to specify a number of *sockets* and a duration. Sockets are the endpoints for communications in the UNIX environment (interprocess communications, UDP, TCP, access to the filesystem, etc.). A call to the select system call will put the calling process to sleep until one of the specified sockets becomes active (e.g. an UDP socket, when an UDP packet has been received) or the specified duration runs out. Since the sockets can be used for all forms of communications of the program with the environment, every incoming event can be noticed with the select call. Internal events caused by timers can be realized with the duration parameter.

The RMFP library needs such an event handler to operate in an efficient way, i.e. to do without constant polling in an endless loop that wastes CPU time doing nothing. It provides two possibilities:

1. The application can do the event handling. To be able to process the events (e.g. incoming packets or timers) at the library, the API provides a method to retrieve the sockets used by RMFP and the profiles and a method to get the timepoint of the next timout due. With this information, the application can implement its own event handler. This the more flexible approach for the application, but has the disadvantage of requiring a more complex interaction at the API.

2. The RMFP library implements already a simple event handler. Again the API provides the functionality to hand over the sockets and the timeouts, but this time in the direction from the application to the library. If an event is due for the application, the event handler invokes the appropriate upcall method at the application.

   This approach is the preferred way, since the necessary event handler is already available at the library and can be used in an easy way at the application

6.5.4 The Class Structure

The `rmfpSession` class

Method: Constructor
PARAMETERS:
`rmfpSessionPars *pars:` Data structure containing the parameters for the session object.
RETURN VALUE: *none*
DESCRIPTION: Constructor for a session object. The pars argument contains all the necessary information to join a session: The sessions address, the profile to use, the profile specific `rmfpProfilePars` argument to configure the selected profile, etc.

Method: Destructor
PARAMETERS: *none*
RETURN VALUE: *none*
DESCRIPTION: Destructor. Closes all sockets and release the allocated memory.

Method: sendADU
PARAMETERS:
`rmfpADUContainer *adu:` Contains the the ADU to send.

Return Value: `Error`: The status of the operation.

Description: An ADU is sent to the session. Within the ADU container, both the header and the data field have to be present. In the header, the object ID and the object start/end flags have to be set appropriatly. The other fields are initialized be RMFP and the profile. After the operation, the ADU container will contain the ADU as it has been sent.

Method: getADU

Parameters:

- `void *_buffer:` The address of the buffer, where the received data should be put.
- `int *_b_size:` The size of the buffer. On return, the variable will hold the number of bytes received.
- `rmfpADUContainer *_adu:` An empty ADU container, to hold the received ADU. It will use the _buffer to keep the parts of the ADU.
- `InAddr *_from:` _from should point to an (empty) InAddr object. It is filled with the Internet address of the sender of the ADU.

Return Value: `Error`: Indicates, if the receive operation was successfull. There are several possible causes for failure during normal operation:

- Own ADU received. This will happen every time, an ADU has been sent, since the multicast socket is operated with the `MC_LOCAL_LOOP` flag. This means, that all packets sent will be received on the same address.
- Duplicate ADU received. Happens, when a retransmitted ADU is received again.
- Buffer to small.

Description: This method should be invoked after the `rmfpApplication::ADURcvd` upcall method was called. It is essential for the application to react immediately, since the received UDP packets stay in the sockets receive buffer, until this method is called. So to avoid buffer overflow, which results in additional packet loss, this method should be called directly from the upcall function itself.

Method: loop

Parameters:

- `rmfpTimeVal *_dur:` The maximal duration, the library takes control. If the _dur point to zero valued timeval, this call is just polls at the network, satisfy expired timers and returns immediately after. If the _dur point to `NIL` the library gets the control for infinite time (it can be stopped by a `wakeup` call.

RETURN VALUE: `Error`: The status of the operation.
DESCRIPTION: This method is the library's main loop. It can take the control from the application, for a specified or infinite time. The application code is then only invoked by upcalls on protocol activity, or when a timeout expires in the library that was set by the application.

Method: wakeup
PARAMETERS: *none*
RETURN VALUE: *none*
DESCRIPTION: This method cancelles a previous call to loop. E.g., it can be called from an upcall function, when the application wants to leave the session.

Method: sendSR
PARAMETERS:
- `void *_ext=NULL`: An optional extension, that is sent along with the control packet.
- `int _ext_size=0`: The size of the extension.

RETURN VALUE: `Error`: The status of the operation.
DESCRIPTION: This function triggers the transmission of a sender's report packet. The packet itself is created by the library, although some of the fields depend on values set by the application previously.

Method: sendRR
PARAMETERS:
- `void *_ext=NULL`: An optional extension, that is sent along with the control packet.
- `int _ext_size=0`: The size of the extension.

RETURN VALUE: `Error`: The status of the operation.
DESCRIPTION: This function triggers the transmission of a sender's report packet. The packet itself is created by the library, although some of the fields depend on values set by the application previously.

Method: controlInterval
PARAMETERS: *none*
RETURN VALUE: `rmfpTimeVal`: The current delay suggested between two receiver's report packets.
DESCRIPTION: Calculates the current delay between two receiver's report packets with the formula used at RTCP [25]. This is only a suggestion. The application can decide to send the packets with a different frequency, or maybe not at all.

Method: setBase

PARAMETERS:

`OID oid:` The object ID used as base from now on.

`Seq seq:` The sequence number used as base from now on.

RETURN VALUE: *none*

DESCRIPTION: This function sets a new *base*. It used at a sender to inform receivers about suitable synchronization points. This information is later put automatically into the sender's report packets. E.g. a sender could send SR packets with the object ID and sequence number of the first packet it sent. This would enable receivers at the beginning of the session to detect the loss of the first packet(s).

NOTE: Due to wrap arounds in the sequence number space the base sequence number will become invalid after some time. It is up to the sender to ensure proper progression or invalidation of the base used in the session.

Method: invalidateBase

PARAMETERS: *none*

RETURN VALUE: *none*

DESCRIPTION: This function invalidates the *base*. This can be e.g. to abandon the synchronization point at the beginning of the session. Receivers joining later cannot ask for retransmissions up to the beginning of the session, but have to synchronise up from the first received ADU other sequence number information (e.g. the highest sequence number field of the sender's report packet.)

Method: getSessionPars

PARAMETERS: *none*

RETURN VALUE: `rmfpSessionPars*`: Pointer to the sessions parameter object.

DESCRIPTION: Returns the current parameters object.

Method: getProfilePars

PARAMETERS: *none*

RETURN VALUE: `rmfpProfilePars*`: The parameters currently used at the profile. The actual type of this object depends on the used profile, e.g. `srmProfilePars`.

DESCRIPTION: Returns the profiles current parameters object.

Method: getTimer
PARAMETERS: *none*
RETURN VALUE: `rmfpTimer*`: Pointer to the the timer instance used by the session (each session object has its own timer).
DESCRIPTION: The access to the sessions timer enables the application to set timeouts and enter the sessions `loop`. If an application timeout expires, the library will call the applications upcall function.

Method: getMemberList
PARAMETERS: *none*
RETURN VALUE: `rmfpMemberList*`: A pointer to a `llist<rmfpMember>` object containing all known members of the session.
DESCRIPTION: To use the member management functionality of the RMFP library, the application has direct access to all member instances of the session. The `llist` class is defined in the toolkit library.

Method: getFDs
PARAMETERS:
- `int *_fds`: A field to take the filedescriptors.
- `int *_num`: The size of the field. On return, *_num is set to the number of filedescriptors actually used.

RETURN VALUE: `Error`: The status of the operation.
DESCRIPTION: When the application does not want to enter the session's `loop`, maybe because there are also other filedescriptors to wait for, the application can use this function to get all filedescriptors used by the session. Together with the possibility to get the next timeout due, the application can implement it's own loop function. The session will the only be polled, when data has arrived or timeouts have expired.

6.5.5 Upcall Methods of the `rmfpApplication` Class

Method: ReceiverReportRcvd
PARAMETERS:
- `rmfpRRHeader *header_p`: Pointer to the received control header.
- `int h_size`: The size of the header.
- `void *ext_p`: Points to the application extension, if received.
- `int ext_size`: Size of the received application extension.

RETURN VALUE: `Error`: The status of the operation.
DESCRIPTION: Called by the session, whenever a receiver's report packet is received.

NOTE: The pointers to the information are only valid during the upcall, and the buffers are freed upon return of the control to the library automatically.

Method: SenderReportRcvd

PARAMETERS:

`rmfpSRHeader *header_p`: Pointer to the received control header.
`int h_size`: The size of the header.
`void *ext_p`: Points to the application extension, if received.
`int ext_size`: Size of the received application extension.

RETURN VALUE: `Error`: The status of the operation.
DESCRIPTION: Called by the session, whenever a sender's report packet is received.
NOTE: The pointers to the information are only valid during the upcall, and the buffers are freed upon return of the control to the library automatically.

Method: SessionPacketRcvd

PARAMETERS:

`rmfpSessionPacket *s_packet`: A pointer to the session packet.
`int s_size`: The size of the session packet.

RETURN VALUE: `Error`: The status of the operation.
DESCRIPTION: Called by the application, whenever a session packet is received.
NOTE: The pointers to the information are only valid during the upcall, and the buffers are freed upon return of the control to the library automatically.

Method: ProfileConfigured

PARAMETERS:

`rmfpProfileType p_type`: The type of the profile used in the session.

RETURN VALUE: `Error`: The status of the operation.
DESCRIPTION: This upcall indicates, that the automatic profile configuration is completed. This happens, whenever a sender's report is received, since the sender report packets indicate the profile used. As long as the configuration phase is not completed, no data packets are received and no data can be sent.
The profile is configured first with default parameters, but the application has the possibility to change the parameters with the `setProfilePars` command.

Method: MemberJoined

PARAMETERS:

`InAddr *addr:` The address of the new member.
`SourceID *src:` The source ID of the new member.
`rmfpMember *member:` The member instance of the new member.

RETURN VALUE: `Error`: The status of the operation.
DESCRIPTION: Called when a packet is received from a member, that was not known so far at session.

Method: MemberLeft
PARAMETERS:
`InAddr *addr:` The address of the member.
`SourceID *src:` The source ID of the member.

RETURN VALUE: `Error`: The status of the operation.
DESCRIPTION: The profile detected, that a member left the session. For protocol profiles with a loose session semantics like SRM this happens after an inactivity timer expires, but other protocols may have more precise information about leaving members.

Method: ADURcvd
PARAMETERS: *none*
RETURN VALUE: `Error`: The status of the operation.
DESCRIPTION: This function indicates the arrival of ADUs. The library design tries to avoid every copying of ADU data within the library, and thus the library cannot read the ADUs out of the sockets before the application provides the buffers. When this function is called, the library has detected activity at a data socket. The application is now expected to call the `getADU` method of the session to read the data into the application buffers.

Method: ADULost
PARAMETERS:
`SourceID src:` The source ID of the original sender of the lost ADU.
`InAddr *addr:` The address of the original sender.
`Seq seq:` The sequence numer of the lost ADU.

RETURN VALUE: `Error`: The status of the operation.
DESCRIPTION: Under some conditions ADUs can get lost completely, even in a reliable protocol. This can happen e.g. when a sender drops out unexpectedly, or the network is down for a lengthy period. This call indicates, that the profile gives up the hope to finally get the specified ADU, although this could still happen.

Method: RetransmissionReq

PARAMETERS:

`InAddr *addr`: The address of the original sender of the ADU.

`rmfpADUContainer *adu`: Contains the original header and name when the ADU to retransmit was received or sent.

RETURN VALUE: `Error`: The status of the operation. The application should return `Error::good`, if it can deliver the requested data, or `Error::data_not_available`.

DESCRIPTION: This upcall is made, when the profile wants to retransmit an ADU. The application doesn't have to provide the data, but it has to indicate this fact with the return code. If the application can provide the data, it just puts it into the ADU container and call the session's `retransit` method.

Method: FreeInd

PARAMETERS:

`SourceID src`: The source ID of the sender of the ADUs to free.

`InAddr *addr`: The address of the sender.

`Seq seq`: All ADUs with sequence numbers less or equal to seq may be freed.

RETURN VALUE: `Error`: The status of the operation.

DESCRIPTION: Some profiles like the LGC profile may provide the functionality to indicate, if all members of a session have received some data packets, and issue some `free` command to the whole group, indicating that the retransmission buffers can be released.

Method: Exception

PARAMETERS:

`Error error` The error code.

RETURN VALUE: `Error`: The status of the operation.

DESCRIPTION: Something went wrong inside the library. This upcall indicates some error condition. The exception doesn't have to mean, that something fatal to the transmission has happened, but it is mostly used to indicate bugs within the library.

6.6 Example Application

This section contains an example application to demonstrate, how applications can use the RMFP library. The application is a simple multicast filetransfer utility to transmit a single file from one sender to a number of receivers. The constraint to a single sender and a single file is not caused by the RMFP library, but would complicate the application source code without showing more of the functionality of the RMFP libraries interface.

The source code of the application is separeted into a C++ header file and a C++ implementation file. The application uses another class, the `SpanList`, that provides the functionality to track the sequence numbers of the ADUs sent or received respectively. This class is not explained here, since it has nothing to do with the RMFP API directly. The single binary can be used both as sender and receiver, depending on command line options.

The filetransfer application demonstrates additionally some features of ALF:

1. The retransmission data is not buffered in the memory, but the application retrieves it from the file itself. This is only possible, since the application has to provide the retransmission data.
2. The received ADUs are processed out-of order, i.e. they are written immediately to the file. The ADU Name carries the byte-offset.
3. The application implements a simple rate control algorithm that includes both original ADUs and retransmitted ADUs.

6.6.1 The Header File

The header file contains the class definition of the `appFileTrans` class. This class provides the main functionality for data communications. It implements both the active functionality of the implementation (e.g. the sending of ADUs or report packets) and the reactive functionality (the upcalls). The class is derived from `rmfpApplication`, the class that specifies the upcall interface, and from `TimerObject`. The latter class provides the interface to the timer upcalls. The timer of the RMFP implementation is used to control the active functionality of the application.

```
#ifndef _APPFILETRANS_H_
#define _APPFILETRANS_H_

#include <stdlib.h>
#include <stdio.h>

// The ToolkitLib's headers
#include <GeneralTypes.H>      // Some simple types (ints, BOOLEAN, etc.)
#include <Error.H>             // Defines error types and debugging facilities
#include <Timer.H>             // Necessary to use the timer in the library.

// RMFP's headers
#include <rmfp_Types.H>        // Some defines, types, functions
#include <rmfp_Application.H>  // The virtual upcall class
#include <rmfp_Session.H>      // The session class. Interface to the library
#include <rmfp_ADUContainer.H> // Container class for the ADUs
#include <srm_Profile.H>       // Defines the srmProfilePars class

#include "span.H"              // Simple management functionality for
```

```
                              // the sequence numbers of received ADUs.

#define DEFAULT_TTL 16         // The default multicast TTL
#define PACKET_DATA_SIZE 100   // Data bytes per packet
#define BUFFER_SIZE 1000       // Buffer size
#define ADU_DELAY .5           // Sending delay of the original ADUs

#define BW 1000.0              // Bytes per sec., including retransmissions

typedef u_int32 ByteOffset ;
```

The definition of the `appFileTrans` class:

```
class appFileTrans : public rmfpApplication , public TimerObject {
protected:
  rmfpSession   *session ;  // "connection end point"
  rmfpSessionPars s_pars ;  // To configure the session object
  srmProfilePars p_pars ;   // To configure the SRM profile

  SourceID source_id ;      // Own source id
  char *address;            // IP address as string
  int port;                 // UDP port number of mc group
  int ttl;                  // Time-to-live to send
  FILE *F;                  // The file to read/write
  SpanList span ;           // Keeps track of the ADUs

  rmfpADUContainer rac ;    // ADU container used for sending
                            // and receiving
  rmfpADUHeader head ;      // ADU Header used in sending
                            // operations
  rmfpADUName   name ;      // ADU Name
  char adu_name_str[256] ;  // To initialize and read out the adu name.
  char buffer[BUFFER_SIZE] ;// Buffer for send and receive.
```

Each `TimerHandle` represents a configured timeout. If a timer expires, one of the parameters of the upcall method invoked at the `TimerObject` (in this case, the `appFileTrans` object) is the corresponding `TimerHandle`. The `TimerObject` can use this information to distinguish between several possible timeouts.

```
  // Some variables for the timers
  TimerHandle send_handle ;     // To time the sending of the ADUs
  TimerHandle recv_handle ;     // To time the receiver (reset the bucket)
  TimerHandle sr_handle ;       // To time the sending of SRs
  TimerHandle rr_handle ;       // To time the sending of RRs

  // For the sender ;
  OID           send_oid ;      // Present object ID for sending
  ByteOffset    byte_offset ;   // Byteoffset for the present ADU
```

```
  int           file_sent ;     // Flag == 1, if last ADU of file has been sent

  // For the receiver
  int flag_start ;              // Have we already received an ADU?
  char file_name[256] ;
  SourceID file_sender ;
  Seq seq_low ;                 // The lowest sequence number of the file
  Seq seq_high ;                // The highest sequence number of the file
  int flag_low ;                // seq_low set?
  int flag_high ;               // seq_high set?
```

The filetrans application implements a simple leaky bucket rate control algorithm: The "bucket" is filled in regular intervals, controlled by a timer with the allowed bandwidth for one interval. Every time an ADU (original and retransmission) is transmitted, the size of the ADU is removed from the bucket. When the bucket is empty, no ADUs can be sent anymore, until the bucket is refilled.

```
  // Rate control
  int bucket ;
```

The function definitions:

```
public:
  appFileTrans(char *fn, char *addr, int p, int t) ;

  rmfpSession *getSession() {return session;}
  void setSession(rmfpSession *_s) {session = _s;}
```

The alarm- and interval-callback methods are the upcall function of the `TimerObject` class. The method `alarmCallback` provides all of the active functionality of the application. The `intervalCallback` method is not used.

```
  void alarmCallback (TimerHandle _th, ClientDataPtr client_data) ;
  void intervalCallback (TimerHandle _th,
                               ClientDataPtr client_data) ;
```

The rest of the methods are the upcalls inherited from the `rmfpApplication` class.

```
  Error ReceiverReportRcvd(rmfpRRHeader *header_p, int  h_size, void *ext_p,
                           int  ext_size) ;
  Error SenderReportRcvd(rmfpSRHeader *header_p, int h_size, void *ext_p,
                         int ext_size);
  Error SessionPacketRcvd(PacketHeader *s_head, void* ext, int ext_size) ;
  Error ProfileConfigured(rmfpProfileType p_type) ;
  Error MemberJoined(InAddr *addr, SourceID src, rmfpMember *member);
  Error MemberLeft(InAddr *addr, SourceID src );
```

```
  Error ADURcvd() ;
  Error ADULost(SourceID src, InAddr *addr, Seq seq) ;
  Error RetransmissionReq(InAddr *addr, rmfpADUContainer *adu) ;
  Error FreeInd(SourceID src, InAddr *addr, Seq seq) ;
  Error Exception(Error error ) ;
} ;

#endif _APPFILETRANS_H_
```

6.6.2 The Implementation File

This file contains the `main` function and the method definitions of the `appFileTrans` class.

```
#include <GeneralTypes.H>
#include <Error.H>
#include <InAddr.H>
#include <Timer.H>

#include <rmfp_Types.H>
#include <rmfp_PacketHeader.H>
#include <rmfp_ADUContainer.H>
#include <rmfp_Print.H>
#include <rmfp_Session.H>
#include <srm_Types.H>
#include <srm_Profile.H>

#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>

#include "filetrans.H"

static int SR_DELAY = 5 ;       // Duration betw. two sender reports (in sec.)
static int RR_DELAY = 5 ;       // Duration betw. two recv. reports (in sec.)
static int SESSION_DELAY = 20 ; // Duration betw. two session packets (in sec.)

const char help_string[] =
"filetrans (-s <filename> | -c) address port [ttl]" ;

int send_mode ;                 // Indicates, if instance is sender or recv.
char *filename ;

char* address ;
int port ;
int ttl = 1 ;                   // Default TTL is one (can be overridden)
```

The main function is called by the run-time system. It reads the command-line parameters and initializes the `appFileTrans` object. It then gives the control entirely to the RMFP libraries event loop.

```
int main(int argc, char *argv[]) {
  rmfpTimeVal now ;
  now.now() ;                    // Sets the timeval {\tt now} to current time.
  rmfpTimeVal timer_setting = 1 ;
```

Parse the command line. The expected syntax is:

For senders: `filetrans -s <filename> <address> <port> [<TTL>]`

For receivers: `filetrans -c <address> <port> [<TTL>]`

```
  if (argc < 4) {
    cout << help_string << endl ;
    exit(1) ;
  }
  if (!strcmp(argv[1], "-c")) {
    send_mode = 0 ;
    address = argv[2] ;
    port = atoi(argv[3]) ;
    if (argc > 4) ttl = atoi(argv[4]) ;
    else ttl = 1 ;
  }
  else if (!strcmp(argv[1], "-s")) {
    send_mode = 1 ;
    if (argc < 5) {
      cout << help_string << endl ;
      exit(1) ;
    } else {
      filename = argv[2] ;
      address = argv[3] ;
      port = atoi(argv[4]) ;
      if (argc > 5) ttl = atoi(argv[4]) ;
      else ttl = 1 ;
    }
  }
  else {
    cout << help_string << endl ;
    exit(1) ;
  }
```

If the RMFP and Toolkit libraries are compiled with the debugging functionality, the global variables `debug_level`, `debug_modes` and `debug_lossrate` can be set to control the debut output.

1. `debug_level` can be set to an integer between 0 and 3, where 3 is the most verbose level.
2. `debug_modes` is a bitset, and each bit represents a code module (i.e. a C++ source file including the header). If the bit of a module is set, debug information for this module is printed. The *ToolkitLib/Error.H* header file contains the module-bit mapping.
3. `debug_lossrate` is a variable of type `double`. Valid values are in the interval $[0, 1]$. The RMFP library induces an artificial lossrate according to this variable. 0 means no losses and 1 means, that all packets are lost.

```
  debug_level = 3 ;
  debug_modes = 0xffffffff & ~db_timer ;
  debug_lossrate = 0.25 ;
  cout << "Debug Modes: " << hex << debug_modes << dec << endl ;

  Error                 err ;
  rmfpSession           *session ;
```

Create and initialize the appFileTrans object and enter the event loop. The session object is created by the appFileTrans object's creation.

```
  appFileTrans          app(filename, address, port, ttl) ;
  session = app.getSession() ;

  cout << "### Entering session loop indefinitely" << endl ;
  err            = session->loop(0) ;

}
```

The constructor of the `appFileTrans` class. The internal variables are set, the `rmfpSessionPars` object is intialized, a `rmfpSession` object is created and the timers are set.

```
appFileTrans::appFileTrans(char *filename, char *addr, int p, int t) {

  rmfpTimeVal now ;
  now.now() ;

  InAddr target_address ;

  source_id = now.getTime() ;    // A new, random source ID for this instance.

  address = addr;
  port = p;
  ttl = t;
```

```
  cout << "Using SourceID " << hex << source_id << dec << endl ;

  target_address.setAddr(address) ; // Initialize the InAddr instance.
  target_address.setPort(port) ;
```

Initialize the `rmfpSessionPars` object. This object contains the parameters the RMFP library needs to initialize a new `rmfpSession` object, the class representing a connection end point.

```
  s_pars.setType(srm) ;                    // The profile type
  s_pars.setSessionAddr(target_address) ;  // The session's multicast address
  s_pars.setScope(ttl) ;                   // The TTL value for sending
  s_pars.setMCOptions(RMFP::mc_local_loop | RMFP::mc_reuse_address) ;
                                           // The socket option for multicast
  s_pars.setSourceID(source_id) ;          // The source ID
  s_pars.setBaseObjectID(send_oid) ;       // The initial object ID
  s_pars.setProfilePars(&p_pars) ;         // The parameters for the profile
  s_pars.setApplication(this) ;            // The upcall object (this object)
  s_pars.setMemberTableSize(100) ;         // Size of the hashtable for members

  session = new rmfpSession(&s_pars) ;
  bucket = int(BW * ADU_DELAY) ;           // For the rate control

  if (send_mode) {                         // Initialeze the sender spec. vars.
    // Open the file to read
    if((F = fopen(filename, "r")) == NULL) {
      cout << "Cannot open file " << filename << endl;
      exit(1);
    }
    // Variables
    send_oid = 0 ;      // Object ID of the file, default value.
    byte_offset = 0 ;   // Byte counter for sent data
    file_sent = 0 ;     // File not yet sent
    // Session init
    session->setBase(send_oid, 1000) ;   // First sequence number is 1000
```

The senders use two timers: The `sr_handle` is the timer for sending sender report packets and the `send_handle` is the timer for sending ADUs.

```
    // Timer
    sr_handle = session->getTimer()->insertAlarm(this, 0, now + SR_DELAY) ;
    send_handle = session->getTimer()->insertAlarm(this, 0, now + ADU_DELAY) ;
  } else {
    // Variables
    file_sender = 0 ;   // Will be set to the source ID of the sender
    file_name[0] = 0 ;  // Empty file_name
```

```
    flag_start = 0 ;    // Is set, if an ADU has been received already.
    flag_low = 0 ;      // Is set, if the first ADU of the file has been rcvd.
    flag_high = 0 ;     // Is set, if the last ADU of the file has been rcvd.
```

The receivers use two timers: The `rr_handle` to time the sending of receiver report packets, and the `recv_handle` to reset the bucket for the rate control. The receiver needs the rate control for the retransmissions.

```
    // Timer
    rr_handle =
      session->getTimer()->insertAlarm(this, 0, now + rmfpTimeVal(RR_DELAY)) ;
    recv_handle =
      session->getTimer()->insertAlarm(this, 0, now + rmfpTimeVal(ADU_DELAY)) ;
  }
}
```

The `intervallCallback` method is not used, but has to be implemented, since it is a virtual method of the `TimerObject` base class.

```
void appFileTrans::intervalCallback (TimerHandle _th,
                                    ClientDataPtr client_data) {
  cout << "### Interval callback (not used)" << endl;
}
```

The `alarmCallback` method is invoked every time a timer expires for this `appFileTrans` object. This method controls the sending of all packet types.

```
void appFileTrans::alarmCallback (TimerHandle _th,
                                    ClientDataPtr client_data) {
  Error err ;

  rmfpTimeVal   now ;
  now.now() ;              // Set the {\tt now} variable to the current time.
```

If the expired timer has been the `sr_handle`, send a sender report packet and reset the timer.

```
  if(_th == sr_handle) {
    // Send a sender report packet
    cout << "<APP> Sending SR..." ;
    err = session->sendSR(0xFF) ;
    if (err.isGood()) {
      cout << "ok" << endl ;
    } else {
      cout << "failed: " << err << endl ;
```

```
    }
    sr_handle = session->getTimer()->insertAlarm(this, 0, now+SR_DELAY) ;
    if (sr_handle == 0) {
      cout << "*<APP> can't set new sr_handle" << endl ;
    }
  } // sr_handle
```

If the expired timer has been the `rr_handle`, send a receiver report packet and reset the timer.

```
  else if (_th == rr_handle) {
    // send a receiver report packet
    cout << "<APP> Sending RR..." << endl ;
    err = session->sendRR() ;
    if (err.isGood()) cout << "<APP> ... RR ok" << endl ;
    else cout << "<APP> ... SR failed: " << err << endl ;
    rr_handle = session->getTimer()->insertAlarm(this, 0, session->nextRR()) ;
  } // rr_handle
```

If the expired timer has been the `recv_handle`, reset the bucket for the rate control mechanism.

```
  else if (_th == recv_handle) {
    // receiver timeout -> reset the bucket for rate control (retransmissions!)
    bucket = int(BW * ADU_DELAY) ;
  } // recv_handle
```

And finally, if the expired timer has been the `send_handle`, an ADU is sent.

```
  else if (_th == send_handle) {
    // sender sends an ADU
    int lenData;
    bucket = int(BW * ADU_DELAY) ;   // reset the rate control bucket
```

If the file has been sent completely, do nothing.

```
    if (!file_sent) {
```

Read the data into the `buffer`.

```
      // We have to set the position, since a retransmission request may have
      // changed the file pointer.
      if (fseek(F, byte_offset, SEEK_SET)) {
        perror("***<APP SND> Error fseek") ;
        exit(1) ;
      }
      lenData = fread(buffer, sizeof(char), PACKET_DATA_SIZE, F);

      cout << "<APP SND> file: read " << lenData << " bytes at pos "
           << byte_offset << endl;
```

Set the fields of the ADU header that are under control of the application: The E, S and F bits (the X bit is not used), the object ID and the payload type (the payload type is not used, but should be lower then 200 to avoid misinterpretations of the ADU as some kind of control packet).

```
if (lenData != PACKET_DATA_SIZE) {
  head.setE(TRUE) ;
  file_sent = 1 ;          // This is the last ADU.
} else {
  head.setE(FALSE) ;
}
if (byte_offset == 0) {
  head.setS(TRUE) ;        // Set the Start bit for the first ADU
} else {
  head.setS(FALSE) ;
}
head.setF(FALSE) ;
head.setObjectID(send_oid) ;
head.setPT(66) ;           // The payload type is actually not used.
```

The first eight bytes of the ADU name are set to the byte offset, coded as ASCII string in readable format. This coding is not efficient, but demonstrates, how this field can be used. Another possibility for the encoding would be to write the binary format of the byte offset into four bytes of the ADU name.

The second part of the ADU is set to the file name.

```
sprintf(adu_name_str, "%08x%s", htonl(byte_offset), filename) ;
name.setNameString(adu_name_str) ;
```

The `rac` variable is an object of the `rmfpADUContainer` class. This class is used to carry a complete ADU that is composed of the header, the ADU name and the data.

```
rac.storeHeader(&head) ;
rac.storeName(&name) ;
rac.storeData(buffer, lenData) ;

cout << "<APP SND> Sending ADU ..." ;
err = session->sendADU(&rac) ;
cout << " Seq: " << head.getSeq() << endl ; // The seq is initialized
                                            // during the send operation.
cout << "<APP SND> \t" << head << endl ;
if (!err.isGood()) {
  cout << "*<APP SND> Error sending packet" << endl ;
}
```

After the ADU has been sent, the sequence number of the ADU has to be registered at the `SpanList`. This information is used later, to check if retransmission requests refer to valid ADUs.

```
      span.insert(head.getSeq()) ;      // Memorize the ADU
      byte_offset += lenData;           // Adjust the offset
      bucket -= rac.getTotalSize() + UDP_HEADER_SIZE ; // Spill bucket
    } // if {!file_sent}
```

Reset the timer for the next ADU.

```
    send_handle = session->getTimer()->insertAlarm(this, 0, now + ADU_DELAY) ;
    if (send_handle == 0) {
      cout << "***<APP SND> can't set new alarm_handle" << endl ;
      exit(1) ;
    }
  } // send_handle

  else {
    cout << "***<APP SND> Illegal timer handle" << endl ;
    exit(1) ;
  }
}
```

This is the upcall method, when a receiver report packet has been received. The packet is not used.

```
Error appFileTrans::ReceiverReportRcvd( rmfpRRHeader *header_p,
                                         int   h_size,
                                         void  *ext_p ,
                                         int   ext_size ) {
  cout << "### ReceiverReportRcvd" << endl ;
  return Error() ;
}
```

This is the upcall method, when a sender report packet has been received. Receiver applications use this packet to check, if they have already received the first ADU (in terms of the sequence number) of the file. Other fields of this packet are used also by the SRM protocol profile to detect losses of the last ADUs. The retransmission requests for packet lost are automatically sent by the protocol profile.

```
Error appFileTrans::SenderReportRcvd(rmfpSRHeader       *header_p ,
                                     int                h_size ,
                                     void               *ext_p ,
                                     int                ext_size ) {
  cout << "### SenderReportRcvd" << endl ;
  if (!send_mode) {
    if (!flag_low) {
```

```
      seq_low = header_p->getLSeq() ;
      flag_low = 1 ;
    }
  }
  return Error() ;
}
```

This is the upcall method, when a session packet has been received. This packet type is not used in this application.

```
Error appFileTrans::SessionPacketRcvd(PacketHeader *s_head,
                                        void* ext, int ext_size) {
  cout << "### SessionPacketRcvd" << endl ;
  return Error() ;
}
```

This is the upcall method, when the protocol profile has been configuered automatically by means of a received sender report packet. a receiver report packet has been received. This mechanism could be used at the creation of the `rmfpSession` object, but is not used in this application.

```
Error appFileTrans::ProfileConfigured(rmfpProfileType   p_type) {
  cout << "### ProfileConfigured" << endl ;
  return Error() ;
}
```

This is the upcall method, when a new member has joined the group, i.e. a packet has been received from that member.

```
Error appFileTrans::MemberJoined(InAddr *addr ,
                                   SourceID     src ,
                                   rmfpMember   *member){
  cout << "### MemberJoined, SourceID: " << hex << src << dec
       << " Addr: " << *addr << endl ;
  return Error() ;
}
```

This is the upcall method, when a member has left the group, i.e. no packet has been received from that member for a certain timespan.

```
Error appFileTrans::MemberLeft(InAddr   *addr ,
                                SourceID      src ) {
  cout << "### MemberLeft, SourceID: " << hex << src << dec
       << " Addr: " << *addr << endl ;
  return Error() ;
}
```

This upcall method is called, whenever the data socket becomes active. This upcall method should always call the `rmfpSession::getADU` method to read the data out off the socket buffers. However, this upcall does not always mean, that a valid ADU has been received. ADUs originally sent by this instance, or retransmitted ADUs already received before are discarded by the protocol profile. In this case, the `rmfpSession::getADU` method returns with an error.

```
Error appFileTrans::ADURcvd() {
  int size = sizeof(buffer) ;
  InAddr from ;
  SourceID src ;
  char fn[256] ;
  int bo = 0 ;
  int ferr ;

  // The ADU has to be read from the socket, even if we are the sender and
  // ignore it.
  Error err = session->getADU(buffer, &size, &rac, &from) ;

  if (send_mode) return Error() ;  // Senders don't care about received ADUs

  if (!err.isGood()) {          // Error in getADU
    cout << "<APP> ADU received, but discarded (own or duplicate)" << endl ;
    return Error() ;
  }
  if (!rac.getName()) {         // We need the ADU name to deal with the ADU
    cout << "<APP> \tno name" << endl ;
    return Error() ;
  }

  cout << "### <APP> ADURcvd" << endl ;
  cout << "<APP> Data received from: " << from << endl ;
  cout << "<APP> \theader: " << (*rac.getHeader()) << endl ;

  src = rac.getHeader()->getSourceID() ;
```

Check the ADU name format and extract file name and byte offset.

```
  if (sscanf(rac.getName()->getString(), "%8x%s", &bo, fn) < 2) {
```

```
  cout << "***<APP> Illegal ADU name: "
       << rac.getName()->getString() << endl ;
  exit(1) ;
}
bo = ntohl(bo) ;      // Adjust the byte order of the offset field

if (!flag_start) {
  // This is the first ADU received -> set the file_name
  cout << "<APP> Opening file " << fn << " to write." << endl ;
  strcpy(file_name, fn) ;
  file_sender = src ;
  if((F = fopen(fn, "w")) == NULL) {
    cout << "***<APP> Cannot open file " << fn << " : " << strerror(errno)
         << endl;
    exit(1);
  }
  flag_start = 1 ;
} else {              // This is NOT the first ADU received
  // Check, if the file of the ADU matches the file of the old ADUs
  if (file_sender != src) { // Check sender
    cout << "*<APP> Wrong sender: " << hex << src << "  required: "
         << file_sender << dec << endl ;
    return Error() ;
  }
  if (strcmp(file_name, fn)) { // Check file name
    cout << "*<APP> Wrong file: " << fn << endl ;
    return Error() ;
  }
}

// If this is the last ADU of the file, we have to store the sequence number
if (rac.getHeader()->getE()) {        // Test the end-of-object flag
  seq_high = rac.getHeader()->getSeq() ;
  flag_high = 1 ;
}

// Everything ok, write the ADU to the file
cout << "<APP> Writing data to file position " << bo << endl ;
if (fseek(F, bo, SEEK_SET)) { perror("***<APP> Error fseek") ; exit(1) ; }
if (!(ferr = fwrite(rac.getData(), 1, rac.getDataSize(), F)))
  if (errno != 0) {           // really an error
    perror("***<APP> Error fwrite") ;
    exit(1) ;
  }
if(fflush(F)) {
  perror("***<APP> Error fflush") ;
```

```
  }
  cout << "<APP> " << ferr << " bytes written to file" << endl ;

  // Keeping track of the received ADU
  span.insert(rac.getHeader()->getSeq()) ; // Memorize the ADU for retrans.
  cout << "<APP> Span: " << span << endl ;

  // Check, if the file is complete
  if (flag_low && flag_high) {
    Span *sp = span.head ;
    if ((sp->low == seq_low) && (sp->high == seq_high)) {  // file complete
      cout << "<APP> File " << file_name << " is complete!" << endl ;
      exit(0) ;         // That's it, terminate.
    }
  }
  return Error() ;
}
```

This upcall method is invoked by the profile, if for some reason an ADU loss cannot be repaired.

```
Error appFileTrans::ADULost(SourceID    src ,
                             InAddr           *addr ,
                             Seq              seq ) {
  cout << "### ADULost" << "\n" ;
  return Error() ;
}
```

This upcall function is called, whenever the protocol profile wants to retransmit an ADU. The ADU Container in the parameter list contains the original header and ADU name of that ADU. The application should provide the data to complete the ADU. The application does not have to provide the data, but can return from this upcall with an error notification (`Error::data_not_available`), if it does not want to retransmit the ADU for some reason.

```
Error appFileTrans::RetransmissionReq(InAddr *addr, rmfpADUContainer *adu) {
  Error err ;
  int len_data ;

  cout << "### RetransmissionReq" << "\n" ;

  // Check, if we have the data. If not, this is a real protocol or application
  // bug.
  if (!(span.check(adu->getHeader()->getSeq()))) {
    cout << "<APP RTR> ADU (" << adu->getHeader()->getSeq()
```

```
        << ") not available" << endl ;
    return Error(Error::data_not_available) ;
  }

  // Read the ADU name
  char fn[256] ;
  int bo = 0 ;
  if (sscanf(adu->getName()->getString(),
             "%8x%s", &bo, fn) < 2) {
    cout << "*<APP RTR> Illegal ADU name: " << adu->getName()->getString()
         << endl ;
    return Error() ;
  }
  bo = ntohl(bo) ;

  // Seek the file-pos and read the data.
  if (fseek(F, bo, SEEK_SET)) {perror("***<APP> Error fseek") ; exit(1) ;}
  (len_data = fread(buffer, sizeof(char), PACKET_DATA_SIZE, F)) ;
  if (len_data == 0) {
    if (errno) {
      perror("***<APP RTR> Error fread") ;
      cout << "***<APP RTR>\ttried to read at position " << bo << endl ;
      exit(1) ;
    } else {
      cout << "***<APP RTR> No data read from disk. Pos: " << bo << endl ;
      exit(1) ;
    }
  } else if (len_data == -1) {
    cout << "***<APP RTR> fread returns -1, Error: "
         << strerror(errno) << endl ;
    exit(1) ;
  }

  cout << "<APP RTR> read " << len_data << " bytes at pos " << bo << endl ;

  // Put the data into the ADU container
  adu->storeData(buffer, len_data) ;
```

If the bucket is empty, no ADUs are allowed to be sent. Return with an error.

```
  // Empty bucket
  bucket -= adu->getTotalSize() + UDP_HEADER_SIZE ;
  if (bucket < 0) {
    cout << "<APP RTR> bucket empty" << endl ;
    return Error(Error::data_not_available) ;
  }
```

```
  cout << "<APP RTR> RTR Head: " << *(adu->getHeader()) << endl ;
  return Error() ;
}
```

This upcall is provided for protocol profiles that provide functionality to free retransmission buffers. Since the retransmission buffers are located at the application, RMFP provides an upcall to notify the application about ADUs, that won't be requested for retransmission by the protocol profile anymore. A protocol profile with that functionality is LGC (see section 5).

```
Error appFileTrans::FreeInd(SourceID    src ,
                             InAddr           *addr ,
                             Seq              seq ) {
  cout << "### FreeInd from" << (*addr) << endl << "\tSourceID: "
       << src << "\tSeq: seq" << endl ;
  return Error() ;
}
```

Serious problems in the RMFP library are signalled with this upcall. The errors are normally not fatal, but may cause (or indicate) problems in the communication.

```
Error appFileTrans::Exception(Error error) {
  cout << "### Exception: " << error.typeString() << "\n" ;
  return Error() ;
}
```

Chapter 7

Evaluation

In this chapter the RMFP specification and implementation are analyzed and enhancements to both are suggested.

7.1 Measurements

To test the performance of the RMFP library, some tests have been carried out in the local network of the Institute of Telematics at the University of Karlsruhe. Due to lack of time, the tests were simple and included only up to five group members. The tests had following objective:

- The performance of the RMFP library in a local network.
- The behavior of late-joining receivers.

7.1.1 The Test Environment

The test application

The tests have been performed with a modified version of the file-transfer application (see section 6.6):

- The command line interface has been enhanced to set the delay between the transmission of ADUs and the ADU data size. The delay does not strictly determine a certain rate; after sending an ADU, the next ADU is scheduled to be transmitted after this delay. If error control or other activities delay the sending of an ADU, this additional delay is not corrected for the next ADUs. Thus, the rate derived from the delay parameter is only an upper bound of the actual rate.
- The receivers measure the time between the arrival of the first packet and the last packet and calculate the average throughput. This calculation, however, only reflects

the data. Header overhead, duplicates and control packets are not counted. Thus the *used* bandwidth on the network in total is higher than the indicated value.

The network environment

The network environment consisted of five hosts (table 7.1).

Name	Processor	Operating System	Network Interface	Role
Walapai	Alpha 21164A (600 MHz)	Digital Unix 4.0	FDDI	sender
Mohave	Alpha 21164A (600 MHz)	Digital Unix 4.0	FDDI	receiver
Cocopah	Alpha 21064A (233 MHz)	Digital Unix 4.0	FDDI, Ethernet	receiver
Moon	UltraSPARC-I (143 MHz)	SUN Solaris 2.6	Ethernet	receiver
Mond	UltraSPARC-I (143 MHz)	SUN Solaris 2.6	Ethernet	receiver

Table 7.1: The systems used for the tests.

The LAN consisted of an FDDI ring and an Ethernet segment. They were interconnected at the *Cocopah*. This system acted as IP-multicast router.

7.1.2 The Performance Tests

The task for each test was the complete transmission of a 1 MByte file. The results of the measurements are always in the view of the receivers, which reflects the characteristics of SRM. A SRM sender has no notion of the receivers. It will transmit all original ADUs, even if no receivers are in the multicast group. For the performance tests, all receivers had joined the IP-multicast group before the sender started the transmission. Each receiver measured the time between the receipt of the first ADU received successfully and the ADU that completed the file.

A single receiver

These tests were carried out with the rate control disabled (i.e. with the delay parameter set to zero). The results in table 7.2 show, that up to a packet size of 2000 bytes the sender was the bottleneck: For 1000, 1400 and 2000 bytes/ADU the packet rate was almost constant around 170 packets per second. Bigger packet sizes, however, lead to losses causing a much lower average transmission rate.

Two receivers

With two receivers the situation was nearly the same as for a single receiver (see table 7.3). However, the receiver *Cocopah* got already problems at a packet size of 2000 bytes. The tests with this packet size also show two important phenomenons:

ADU size bytes	Kbytes/s at Mohave		
1000	171	172	169
1400	240	241	245
2000	342	339	340
3000	107	98	98
4000	81	125	138
8000	71	80	77

Table 7.2: Test with one receiver.

1. In every of the three tests at the beginning of the transmission Cocopah received a number of packets without problems. But with first losses, the error control mechanisms lead to further losses, instead of helping to repair them. The loss recovery could only be completed after the sender stopped sending original ADUs.
2. In the third test, the error control activity lead to problems at Mohave, too. After starting to process the control packets from Cocopah, Mohave began to loose packets, too.

ADU size	Kbytes/s at					
bytes	Mohave			Cocopah		
1000	173	175	174	173	175	174
1400	243	244	240	242	242	240
2000	370	366	269	79	77	66
4000	66	81	78	53	60	63

Table 7.3: Test with two receivers.

Three receivers

The tests with a packet size of 1000 bytes (see table 7.4 lead again to the same results as before: almost no losses, and the few losses could be repaired. The bottleneck was the sender.

At 1400 bytes/ADU the collapsing problem can be seen again: In the second test, all systems received with the full rate of 243 Kbyte/s. This shows, that under conditions with no losses all receivers are capable to process this rate. In the third test, Mond collapsed and the average throughput at this system dropped to 12 KByte/s. In this test, Mohave had no problems, but Cocopah got problems, too.

The tests with 2000 and 4000 bytes/ADU have been carried out with a throttled rate at the sender. For the tests shown in the table an optimal value for the ADU delay parameter was used. The rates were just high enough to allow Mond to repair a few occasional losses without collapsing.

ADU size bytes	Kbytes/s at Mohave			Cocopah			Mond		
1000	174	174	175	174	175	175	174	175	175
1400	126	243	256	122	243	88	98	243	12
2000	206	205	207	180	205	207	206	206	207
4000	157	157	158	157	157	158	158	158	158

Table 7.4: Test with three receivers.

Four receivers

The tests with this rate show a surprising low throughput at all ADU sizes. The values of table 7.5 are results of experiments with an optimized sending rate. Higher rates lead to collapses at the receivers Moon and Mond. It is not clear, why the results differ that much from the results with just one of them: The two SUNs have the same hardware and the same network connection; in a loss-free transmission almost no control traffic is generated by the two systems that could cause an overload.

An explanation could be, that the both systems use the *Network File System* to write the received data to the file system. This means, that the data received at the SUNs is send on the same Ethernet segment to the file server of the institute. Thus, the load on the Ethernet could be the limiting factor, especially in collapsing scenarios with an extreme high error control bandwidth.

All performance tests with those two receivers together showed the same problems.

ADU size bytes	Kbytes/s at Mohave			Cocopah			Mond			Moon		
1000	54	54	55	54	52	54	53	54	54	52	53	53
1400	61	60	60	59	60	54	60	58	54	60	60	53
2000	63	63	64	62	64	64	61	63	64	62	64	63
4000	87	87	83	86	87	77	86	87	24	87	86	31

Table 7.5: Test with four receivers.

7.1.3 The Tests with Late-joining Receivers

The RMFP library allows late-joining receivers to request ADUs they have missed before joining the group. In the tests carried out to analyze the behavior this mechanism the same 1 MByte file has been transmitted completely, before one or several receivers joined the session late. In this scenario all ADUs sent to those receivers were retransmissions and the measurements reflect the throughput of those retransmitted ADUs – again counting only the actually used data, without counting packet headers, duplicate ADUs and control packets.

In these tests, the sender has always been Walapai, and the ADU size was fixed to 1400 byte (this is near the optimum for Ethernet).

Late-join of a single receiver

The results in the first table 7.6 show the late join of Mohave. The throughput of the tests varied due to the probabilistic mechanisms in SRM's error control, but are always around 50 KByte per second.

Under the same conditions, both Mond and Moon showed a much slower throughput.

Finished receivers	Kbytes/s at Mohave		
none	37	48	49
Cocopah	41	47	47
Cocopah, Mond	51	40	67
Cocopah, Mond, Moon	50	52	45
	Mond		
Mohave, Cocopah, Moon	9	8	10
	Moon		
Mohave, Cocopah, Mond	8	8	7

Table 7.6: Late join of a single receiver.

Late-join of two receivers

In table 7.7 the tests were carried out with Mohave and a second late-joining receiver. The results for Mohave with Cocopah, Mond and Moon as second late-joining receiver are similar to the tests with one late-joining receiver, although Mohave had a lower throughput together with Mond and Moon. This could result from increased processing of control packets from the two SUNs. Cocopah proved to be a bit slower than Mohave. This indicates the dependency of the throughput on the processing capacities of the systems, since Mohave is a faster machine than Cocopah (see table 7.1).

Finished receivers	Kbytes/s at					
	Mohave			Cocopah		
none	52	59	51	36	37	35
	Mohave			Mond		
Cocopah	44	46	43	9	8	12
	Mohave			Moon		
Cocopah	44	46	43	9	8	12
Cocopah, Mond	37	40	38	10	9	8

Table 7.7: Late-join of two receivers.

Late-join of three receivers

The results of the tests with the three late-joining receivers Mohave, Mond and Moon are shown in table 7.8. The Throughput at Mond and Moon stayed the same, whereas Mohave's throughput decreased again, probably caused by the control packets of the two SUNs.

Finished receivers	Kbytes/s at								
	Mohave			Mond			Moon		
Cocopah	27	30	29	7	6	8	8	8	10

Table 7.8: Late-join of two receivers.

Late-join of four receivers

The results shown in table 7.9 show nothing new.

Finished receivers	Kbytes/s at											
	Mohave			Cocopah			Mond			Moon		
none	43	46	40	20	21	18	8	7	6	8	7	7

Table 7.9: Late-join of two receivers.

7.1.4 Interpretation of the Results

The behavior of SRM

1. SRM specifies no flow control mechanism, but only rate control. This is the result of the very loose relationship of the group members in SRM; the sender does not have to know about the receivers, and there is no special feedback information from the receivers to the senders specified. The retransmission requests can be satisfied by every group member that has already received the requested ADUs.

   The lack of the flow control makes SRM to be a bad choice for an application like the file-transfer application used for the tests, since file-transfer works best, if the data can be transmitted as fast as the network and the receiver capacities allow. With SRM, only a fixed sending rate can be used. In this situation the file-transfer application has to implement the flow control itself.

   RMFP introduces the Receiver Report packets to provide application controlled feedback. Applications with need for flow control can use these packets, but this can lead to feedback implosion at all members, since all members will receive and process those packets. The file-transfer application doesn't implement a flow control mechanism, but only rate control.

   The lack of flow control was the main reason for the instability of transmission in the tests: If no losses occurred, all receivers were fast enough to process all ADUs. Even losses of a few ADUs, however, lead to the collapse of the receiver, that could be resolved only after the sender finished the transmission of original ADUs.

2. If the number of ADUs lost is high, the error control mechanism of SRM produces much CPU load. The reason for this problem is the individual treatment of each ADU lost: Each ADU needs its own timer to time the transmission for retransmission requests or the retransmission of the ADU respectively. This produces long timer chains (timers are usually implemented as a linked list sorted by the expiration time) with time consuming list traversal, when timers are set or deleted. The processing of a retransmission request containing the sequence numbers of many ADUs can thus slow down the member.

3. If the number of ADUs lost is high, SRM also produces a big amount of control traffic: Since for every ADU the exact time of the transmission of a retransmission request is randomized and individual, a big number of lost ADUs, even if they are consecutive, produces many control packets.

4. The individual treatment of lost ADUs and the individual timing of their retransmission requests lead to the relatively low bandwidth during the tests with late-joining receivers: Instead of requesting the retransmission of all ADUs in a single control packet coded as span, many control packets were created.

5. The setting of the timers does not depend on the number of ADUs to be retransmitted. If a retransmission request is sent, the timer is backed off (i.e. doubled) and if the requested ADU could not be received after this time, the request is sent again. However, with a big number of ADUs lost, most of the ADUs cannot be retransmitted until the timer expires again and the retransmission request is repeated. This leads to an enormous amount of control traffic at the time a receiver joins-late and tries to catch-up previous ADUs.

   Furthermore, the backing off of the request timer several times leads eventually to very long timer settings. If many ADUs get lost originally, for some ADUs the delays between the retransmission requests can get long (up to minutes). Caused by this it takes similarly long until all ADUs are finally retransmitted. This effect can be seen at the late-join of receivers and after a collapse of a receiver: When most ADUs have been received successfully, it takes a long time with no activity at all, until the very last ADUs are eventually requested and then retransmitted.

   As a solution to the problem of extreme long request timers, the implementation of the SRM profile limits the number of back-offs. For the problem of the control overflow one solution would be to make the timers dependent on the number of lost ADUs.

The list shows, that especially the timing of the retransmission requests can create problems, if receivers suffer high losses and especially in the case of late-joins, where many consecutive ADUs are requested.

The implementation

A good estimator for the performance of the implementation is the throughput, that can be achieved if no losses occur. The results of the performance tests show, that the RMFP library could send up to approximately 170 ADUs per second on the sender Walapai, with a bandwidth of up to 370 KBytes/s. This is not much compared to other protocols. However, the CPU load, observed with the *top* utility during the tests, has been rather low (10-20%) during the sending without error control. This indicates, that the sources of delay could be the result of the file read operation: The application always reads the data in the size of the ADU data from the file and does not read the file completely before starting to send. The file, however, is located on the file server and has to be accessed via NFS. However, due to lack of time, this could not be verified before finishing this report.

The problem of the collapse is probably mainly caused by the SRM profile. However, the implementation can be optimized to offer a better performance in this situation: The list realizing the timer chain could be replaced by a (balanced) tree and the management of the ADU Entries, i.e. the class instances containing the state information (ADU header, timer handles, etc.) could be optimized. A general approach to improve the performance of the library would be an optimized memory management.

7.2 Aspects of ALF

The ALF principle requires the application to frame the ADUs. These ADUs are then the data packets for the transport protocol, and the network protocol has to transport them. RMFP is designed for the Internet and uses UDP/IP as network protocol. UDP and IP, however, have a limited packet size of 64 KByte. This maximum IP packet size, however, is usually too big for the link layer, and thus IP can `fragment` the packets into smaller packets for the transmission over the link layer. The data, that fits into a single link layer packet depends on the link layer. E.g. Ethernet (IEEE 802.3) allows a maximum data size of 1454 Bytes [28].

Fragmentation differs from *segmentation* in that the fragments itself are transmitted as IP-packets and can be routed at the network layer themselves. Segmentation and reassembly would have to be performed at the endpoints of each link.

If IP has to split an IP packet into several fragments, all fragments have to be received, so that the original IP-packet can be assembled again. However, losses occur on single fragments, and thus the loss-rate for fragmented IP packets is higher than the loss-rate for IP packets transmitted at once. The loss-rate of a fragmented IP packet can be calculated out of the loss-rate of the fragments (i.e. the loss-rate of IP packets transmitted in single link layer packets):

$P_{IP} = 1 - (1 - P_f)^n$,

where P_{IP} is the loss-rate of a fragmented IP packet, P_f is the loss-rate of a single fragment (i.e. the loss-rate of IP packets transmitted at once) and n is the number of fragments per IP packet. For small loss-rates and a small number of packets the loss-rate can be approximated to

$P_{IP} \approx n \cdot P_f$

For small loss-rates and a small number of fragments per IP packet, the increase in the loss-rate is not critical. However, the loss-rates in the *multicast backbone* (MBONE) that is formed of the multicast capable systems in the Internet are comparatively high [12], and thus fragmentation in multicast should be avoided.

The tests described in section 7.1 did not show this problem, since the loss-rate in the LAN is very low. The sometimes high losses at the receivers were caused by overload at the receivers and not by the network, where fragmentation would increase the loss-rate.

Since each ADU corresponds to an IP packet, the application has to be aware of the smallest of the *Maximum Transmission Units* (MTU) of all the links the IP packet has to travel through the network. The MTU is the maximum data size that a single link layer packet can carry.

This responsibility is a difficult problem for the application, since it is not even possible in the current version of IP (IPv4) to query this minimum path MTU. Thus, the application has to make a guess.

The practical limits of the ADU size and the difficulties in calculating an optimal value are a severe problem of the ALF approach, since the application data may be of another structure. This can make it difficult for some applications to use the advantages of out-of-order processing. Such applications have to reorder the received ADUs prior to processing them.

7.3 Aspects of the Implementation

7.3.1 Minimal-copy Architecture

The advantage of the minimal-copy architecture

The minimal-copy architecture (section 6.3.2) has been developed to limit the copying of the data in the protocol stack. The repeated copying of the data at the interfaces between the layers has been identified as one of the possible bottlenecks in high-speed applications [4]. However, the copying of the data at the interfaces of the layers has been an important part of the isolation of single protocol layers.

The problems with the minimal-copy architecture

The use of the minimal-copy architecture requires a tight synchronization of the application and the protocol library. Since all ADU buffers belong to the application, all protocol operations requiring such buffers (send, retransmit, receive) have to be invoked by the application and have to be finished, when the method call returns. Otherwise a more complex buffer management would be needed that allows the RMFP library to control application buffers for a longer period.

The send, retransmit and receive operations that are invoked by the application access themselves the socket interface. They copy the ADUs to send directly out of the application buffer into kernel buffers or ADUs received from the kernel buffers into the application buffers. The operating system, however, may not be able to execute these operations immediately. In such a situation, it offers two operation modes:

- In the *blocking* mode, the system calls return only, when they can be executed. E.g. the `receive` system call returns only, if data is available. If no data is available at the time this function is invoked, the return is delayed until data is actually received. This can block the process (or thread in a multi-threading environment) forever.
- The *non-blocking* mode guarantees, that the system calls return immediately, even if they cannot be executed. They indicate this event with an error code as return value.

The requirement for the send, retransmit and receive RMFP library calls to return immediately can only be satisfied with the non-blocking mode. This means however, that the application has to deal with the errors of the system calls at the socket interface.

There is another problem with the receive operation of the RMFP library (the `getADU` call): The event handler of the library detects, when an ADU has been received. It then will invoke the `ADUreceived` upcall of the application. The application has immediately to call the `getADU` method of the session object to free the receive buffers at the operating system. Otherwise ADUs can get lost if the the buffer suffers an overflow.

The call to `getADU`, however, might not be successful: The received ADU can be without value, since it has been sent by this application instance itself or might be a duplicate. The protocol profile detects this situation, but the application has again to deal with possible error return values of the `getADU` call.

The send operation of the library (the `sendADU` method of the session class) calls directly the socket interface's `sendmsg` system call. This is the reason, that the control for the timing to send ADUs is completely at the application. This again means, that flow, rate and congestion control algorithms have to be implemented at the application and cannot be part of the RMFP library.

Evaluation of the minimal-copy architecture

To evaluate the benefits of the minimal-copy architecture it is necessary to compare the behavior of two protocol implementations, that differ only in this aspect. However, due to the lack of time, this has not been possible in this work.

The question is, if the copying operations are really a bottleneck in reliable multicast transport. Although it is easily possible today to use networks with a bandwidth above 100 MBit/s in unicast communication, it is unlikely that the same rates can be reached in reliable multicast, even if all systems are connected to a high-speed network. Some reasons are:

- Reliable multicast has higher processing demand for transmission control. Depending on the protocol this is the case at least at some of the group members. E.g. the LGC protocol (section 5) requires some members, the group controllers, to process the control packets of the members of their subgroup and to control the retransmissions in their group. The SRM protocol sends all control packets to the global multicast group, requiring *every* member to process *all* control packets. Sender based approaches have the processing bottleneck at the senders, and FEC based protocols require most of the CPU time to encode and decode the redundancy packets.

- One key aspect of multicast is that all receivers make the same progress in receiving the data. Only then the retransmissions can be kept low. This leads to the situation, where the slowest receiver and the slowest network link become the bottleneck for the progress of the whole group. If the communication is to fast for some receivers, some form of error control mechanism has to applied, requiring again more CPU load at the involved systems.

In the view of ALF the minimal-copy architecture is no problem, since a tight control of the transport protocol by the application is desired. Thus the application should be able to control the exact timing of sending and receiving operations anyway. In fact, together with the concept of ALF another concept, the *Integrated Layer Processing* (ILP [4]) is introduced.

ILP tries to minimize operations that require the processing of the transport data, including the copying operations. ILP has been investigated in the last years e.g. in [1].

This point is another aspect of the tradeoff of increased application complexity in favor of flexibility and performance that is a key aspect of ALF.

7.3.2 Event Handling

The event handling between the RMFP library and the application is an important aspect of the API. To be efficient, the delivery of events has to be controlled by the operating system: A process or thread waiting for events can be put into the *waiting* mode. This means, that the operating system does not schedule the process/thread, until an event for that thread occurs.

In a multi-threading design it is no problem, to have several sources of events that are controlled by separate modules. An example with the RMFP library would be an application that is member in several sessions, manages additional files and reacts on input events that are generated e.g. by the X-Window system. The application would use an own thread for each event generating source.

With the conventional design, i.e. each process has only a single thread, the application can always wait only for a single event source. Thus only events from that source would wake up the process again. The only solution to that problem is a single event handler that provides the event handling for all event sources.

The system call used to put a process/thread to sleep to wait for an event is the `select` function. Every source of input, e.g. network sockets and files, are associated to *file-descriptors* that can be checked with the `select` function. Additionally a duration can be specified. After that time the `select` function returns, even if no events have been received. This enables the combination of waiting for input events and timing.

The RMFP library uses such an event handler, that is based on the `select` function (see section 6.5.3). However, it uses one event handler for each session, which leads to the described problem. One solution for the application programmer would be to use a thread for each session. For single-threaded applications the only solution is to use the event handler of one of the `rmfpSession` objects. The RMFP library's API provides the functionality to ask the session object for the used file-descriptors and the next timeout. With this information the application can register these file-descriptors and a new timeout at the session object with the active event handler.

A suggestion for the next revision of the RMFP library is to provide an extra event handler class as a module. The session objects could be configured to use such an event handler object that is controlled by the application. This would ease the design and implementation of applications and is a clearer approach.

7.4 The ADU Naming Concept

The ADU naming is necessary to identify the ADUs at the receivers, both at the protocol profiles providing the reliability and at the applications to process the ADUs in the application context. This section discusses the ADU naming concept of RMFP and the profiles and what mechanisms are supported with this concept.

The protocol profiles and the application generally need different identifiers for ADUs: The profile's reliability mechanisms need some kind of sequencing information for loss detection and recovery. The applications need some information about the ADU in the application context, e.g. a file name and byte offset. The ADU format of RMFP (see section 3.3.2) has three fields concerning ADU naming:

The sequence number: The protocol profiles use this field to sequence the ADUs, so that lost ADUs can be detected.

The object ID: The object ID is used by the application to reduce overhead. It is always set by the application, since the application has to decide, to which object[1] an ADU belongs. However, the object ID can also be used at the protocol profile, if the sequencing mechanism includes the objects (see section 3.2.2). The profile specifications of SRM and LGC and the implementation of the SRM profile, however, use the *plain* sequencing, i.e. all ADUs are sequenced according to their sending order, regardless of the object ID provided by the application.

The ADU name: This field is only used by the application, to carry the context information necessary to process the ADU at the receiver (see section 2.3.2).

These fields are used by applications, RMFP and the protocol profiles to provide the required functionality. Following mechanisms are specified in the SRM and LGC profiles and implemented in the RMFP library's SRM module:

Loss detection by the protocol profile: The loss detection mechanisms use only the sequence number field. All ADUs of a single sender have a total order assigned and lost ADUs are detected by means of gaps in this single sequence space.

Application controls the retransmission data: According to the ALF principle the management of the retransmission data is up to the application. A problem is, that retransmission requests encode only the sequence numbers of lost ADUs and not the ADU names. In the implementation the SRM profile of a group member that wants to do the retransmission maps this sequence number onto the ADU name of the requested ADU. Thus the application can use the ADU name to provide the requested data (see section 3.2.1).

[1] These objects have nothing in common with *class objects* of some object-oriented programming language, but refer to the structure of the application's transmission data

Identification of ADUs in application context: The ADU name provides the necessary context information, so that a receiver application can process the ADU. This ADU name may not be necessary for all applications, since the sequence number and object ID are sufficient for them. However, most applications can profit from this field (section 2.3.2).

Hierarchical naming with objects: If the application uses objects consisting of several ADUs, the object ID field allows to reduce transmission overhead (see section 3.2.2).

However, there is one mechanism important for reliable multicast, that is not possible with the specified SRM and LGC profiles and the implementation: *Semantic reliability.*

7.4.1 Semantic Reliability

When the receiving instance of a conventional transport protocol detects a loss, a retransmission request is automatically sent. In multicast, however, there are applications, that don't need the reliable *receipt* of all data. E.g. a white-board application needs only to receive the data of the pages currently viewed. Data for other pages is not used, and automatic retransmission requests for unused data waste bandwidth and processing capacity. For such applications it is useful, if session members that suffer losses can control the reaction to such a loss – whether or not to send a retransmission request.

This mechanism is called semantic reliability [2], since the decision depends on the data, that got lost.

The problem of semantic reliability is, that the application needs the *ADU name* of a lost ADU to decide how to react. The protocol, however, can only compute the sequence numbers of lost ADUs, since lost ADUs are detected by means of gaps in the sequence number space.

This *mapping information* necessary to provide the ADU name corresponding to the known sequence number has to be transmitted separately from the ADU. The mapping information has not to be transmitted reliably, since it is only used to *avoid* unnecessary retransmission requests: If it is not available for an ADU detected as lost, the protocol can transmit the retransmission request automatically to be sure that no required ADU is lost permanently.

There are two possible approaches to transmit the mapping information:

1. The mapping information is transmitted in some session or control protocol. To increase the reliability for the mapping information, it can be secured for example by a simple FEC mechanism: The mapping information for a given ADU is transmitted in several subsequent session/control packets.

   The disadvantage of this approach is the necessity, that the mapping information has to be transmitted for each ADU, which itself produces some overhead.

2. ADUs contain the mapping information for other ADUs. E.g. each ADU carries the mapping information of its predecessor.

The overhead necessary for this approach is smaller than for the previous one, since no extra control packets have to be transmitted.

Both approaches have the disadvantage of requiring additional bandwidth for the control information. However, for applications, where many receivers each need only a small portion of the transmitted ADUs, the number of retransmission requests and retransmissions can be reduced effectively. Especially if the packet loss-rate is not high, it is unlikely, that both the ADU and the packet with the mapping information get lost, and thus almost all loss recovery action for not needed ADUs can be avoided.

The next section presents an approach based on the application objects to reduce the bandwidth overhead caused by the mapping information.

7.5 Using Objects for semantic Reliability

The concept of semantic reliability imposes considerable overhead to the communication to transmit the information necessary to map the sequence number to the ADU names for lost ADUs. This overhead is necessary to be able to decide the reliability requirement for each ADU *independently*. However, the reliability requirements of application data is in many cases not that independent at the ADU level, because the size of ADUs is limited due to the constraints of the network packet size. E.g. a file transfer application will typically segment each file into several ADUs, and the information of a file is typically only useful for the receiver, if it is complete. Thus, all ADUs belonging to a given file share the same reliability requirement, i.e. the reliability requirement of the file.

In RMFP the application uses objects to represent data structures that don't fit into a single ADU. Thus, an efficient approach to reduce the overhead is to use the objects as the units of semantic reliability. The mapping information consists of the protocols object ID and an *object* name. This object name has to be chosen by the application to be sufficient for the receivers to decide about the reliability of the object. In the file-transfer example this object name would be the file name.

An advantage of this approach is, that the mapping information does not need to waste any additional bandwidth at all, since the object name is important for the receiver application anyway to process the ADUs. It is generally part of the ADU name of each ADU, or has to be provided by some session protocol (see section 3.2.2). If it is part of the ADU name, the receipt of a single ADU of an object is sufficient to decide about the reliability for all other ADUs of this object.

This object based approach works better with the increasing number of ADUs per object, since at the beginning of an object the mapping information might not be available fast enough to prevent retransmission requests for lost ADUs not required.

7.5.1 Loss Detection

Although the definition of objects makes the support for semantic reliability feasible, it creates new problems for the loss detection. Generally transport protocols discover losses as gaps in the plain sequence number space. This approach, that is also part of the specifications of the SRM and LGC profile, allows a receiver only to compute the sequence numbers of lost ADUs. However, to be able to do decide to which object a lost ADU belongs the object ID is required, too.

This requires, that the object ID is part of the sequencing information. This cannot be done with plain sequencing, only with object relative sequencing (see section 3.2.2). Lost ADUs are detected by gaps in the object relative sequence number space. The following mechanisms are necessary to ensure correct loss detection and recovery:

- The so-called *heartbeat* mechanism to detect tail-losses has to be modified. Since every object has its own sequence number space, special control information about the highest-sequence-number-sent has to be transmitted for each object. With this heartbeat information receivers can detect the loss of ADUs, even when no ADU with a higher sequence number has been received.

- During a session the number of transmitted objects may grow on and on. Thus, without some kind of garbage collection, the amount of information to indicate the highest-sequence-number-sent grows linearly with the number of objects sent. To limit the size of the heartbeat packets the application has to *close* an object explicitly, when it wishes to stop transmitting ADUs for that object. In this event a close packet is transmitted containing similar information as the heartbeat packets. But the contained sequence number is indeed the sequence number of the last ADU of the object. Future heartbeats won't contain information about this object anymore. That's why this information has to be transmitted to the receivers *reliably*.

  If a sender closes an object, this means only, that the sender stops sending new *original* ADUs for this object. The ADUs of this object can still be requested for retransmission.

7.5.2 Parallel Transmission of Objects

Some applications require the interleaved transmission of ADUs of several objects, called *parallel object transmission*. An example is the white-board application: An user can use several pages at the same time, sending ADUs for several pages interleaved. Typically a page will be the unit for the reliability requirement: When the page is displayed, all its ADUs have to be received reliably, and if the page is not displayed, its ADUs are not used at all. Thus, each page is represented by an object.

The suggested mechanism of sequencing all objects independently supports this mechanism.

7.5.3 Evaluation of the additional Overhead

The introduction of objects was motivated by the idea of reducing the used bandwidth in avoiding unnecessary retransmissions. This is an optimization of loss *recovery*. On the other side, objects create new overhead caused by the more complex loss *detection* that is necessary. The overall bandwidth required depends on the application's transmission characteristics.

The overhead created by the loss detection mechanism for objects has its origin mainly in new or more complex packets (see section 7.5):

1. Every time a protocol profile transmits sequencing information in a control packet, additionally to the sequence number the object ID has to be transmitted. For packet types encoding several sequence numbers as *spans*, this has the further disadvantage that each span has to be constrained to a single object. This makes the encoding more space consuming.

2. The *heartbeat* packets produce overhead that is proportional to the number of objects in transmission.

   This overhead can be reduced by heuristics selecting the objects, that contribute to the heartbeat packet, e.g. by means of the duration since the last ADU sent of each object.

3. The overhead produced by the *close* packets is inversely proportional to the number of ADUs per object, since the close information has to be transmitted just once (however reliably) for every object.

The list shows, that the overhead introduced by objects is mainly dependent on two aspects: The number of ADUs per object, and the number of objects in transmission. Both parameters are application dependent.

This extra overhead shows, that applications that don't use semantic reliability work best with plain sequencing. If the mechanisms for semantic reliability are used for such applications, the extra bandwidth can be avoided almost completely, if the application transmits all ADUs with the same object. However, in that case the application couldn't benefit of the hierarchical ADU naming with objects, that has been introduced to reduce transmission overhead, too.

7.5.4 Changes of RMFP and the Protocol Profiles

To support semantic reliability, protocol profiles have to change the mechanism to detect tail-losses as described in section 7.5.1.. Necessary is the change of the *heartbeat* control packets and the introduction of *close* control packets.

Both packet types are very similar: Both contain pairs of object ID and highest sequence number for several objects. The heartbeats contain the pairs for all objects in transmission, and the close packets contain the last sequence number for objects for which the application indicated, that no new ADUs are transmitted.

Whereas the heartbeat packets are transmitted unreliably and periodically, the close packets are transmitted only once for every object and have thus to be transmitted reliably. However, normally there is no mechanism to transmit control packets reliably, since they are themselves used to provide reliability.

A possibility to achieve reliability for the close packets is to transmit them similar to ADUs belonging to an object. This object is reserved for reliable control packets and is always *open* (i.e. it cannot be closed). Thus the close packets transmitted in this object are not needed to control this object.

The normal reliability mechanisms apply to this object, with the exception, that all receivers have to receive the packets of this object reliably. The applications have no access to this object, the object ID for this object is reserved by the protocol profile (or RMFP).

Close packets are only used by senders, and thus don't require receivers to become senders of reliable packets. So even protocols allowing only a single sender can use this mechanism.

7.6 Related Work

7.6.1 RMF

The Reliable Multicast Framework (RMF) is another work in progress about protocol frameworks for reliable multicast. The latest publication about RMF is an Internet draft [7].

RMF tries to provide a structure consisting of packet formats and multicast session state information, that allows the implementation of specific protocols through RMF.

RMF uses self-identifying packets and session state information to simplify the receivers:

- The session state information at the receivers is configured by a session protocol. An example for this information is the address to which acknowledgment packets are sent.
- Self-identifying data packets contain information, how the receivers should respond to them: By positive or negative acknowledgments, with multicast or unicast, etc.

The idea of this mechanism is to build *universal receivers*, i.e. a unique receiver protocol implementation for all multicast protocols. Thus a receiver application does not have to care about the used protocol. The sender can even change the mechanism during the session by changing the control information in the data packets.

Comparison to RMFP

A comparison of RMFP and RMF is difficult, since the specification of RMF is not yet completed. It is not clear, which integration effort for protocols is necessary and which protocols can be supported and which not.

However, following points give an overview about some key issues:

- The approach of RMF provides more functionality for the protocols than RMFP does for the protocol profiles. In RMFP the protocol state machines of the profiles have to be implemented for every protocol profile. RMF tries to provide at least for the receiver a generic protocol state machine, that can be configured by a session protocol.
- RMF is designed as transport protocol in the layered protocol architecture, whereas RMFP uses ALF.

A more thorough comparison can be done, when the development of the two frameworks has made further progress.

7.6.2 The Generalized Data Naming Approach

In [22] Steve McCanne and Suchitra Raman have proposed a data naming scheme that is similar to the suggested use of objects to provide semantic reliability in RMFP (see section 7.5).

They introduce hierarchically ordered *containers*. The leaf containers are similar to the objects introduced by RMFP, whereas the inner nodes of the container tree contain other containers. The *names* of the containers that are necessary for the application to process the ADUs are transmitted separately to the ADUs in *bind* packets. This is similar to the mechanism RMFP supports for the object names (see section 3.2.2).

The idea and the use of the leaf containers are very similar compared to the objects in RMFP suggested for semantic reliability (section 7.5):

- All leaf containers are independently sequenced.
- A *bind* session packet carries the mapping information of the *container descriptor* (RMFP: object ID) and the *container name* (RMFP: object name).

Both approaches face the common problem: The ADUs are sequenced relative to the start of their object, and thus the control information transmitted by the sender to enable the detection of tail-losses at the receivers has to provide the highest sequence number for every object.

The suggestion for RMFP tries to limit the amount of this information by requiring the application to close completed objects. McCanne/Raman allow an arbitrary number of objects with their container hierarchy:

- Every container has a signature. For leaf containers consisting of ADUs, this is just the highest sequence number. For containers consisting of containers, the signature is calculated with a hash function out of the signatures of the child-containers.
- A special *Session Announcement Protocol* (SAP) is used between the sender and the receivers to detect the tail-losses: The sender periodically transmits an *update* packet that contains the signatures of the root-container and its child-containers.
- Every receiver calculates the signatures of all containers it has received, too. On receipt of the update packet it compares the signature in the update packet with the signature it has calculated for the root container in his view as receiver.
- If there is a discrepancy between the sender's and receiver's signatures of the root-container, the receiver compares the signatures of the child-containers. Since the root-containers signatures didn't match, one or more of the child-containers signatures can't match, too.

- The receiver sends a *query* packet, asking for the signatures of the child-containers with the discrepancies.
- The sender or another member with the requested information responds with a new update packet, this time with the signatures of the requested containers and their child-containers.
- This update/query cycle is repeated, until the signatures of leaf containers are transmitted in the update packets. These signatures consist of the sequence numbers of the last ADU sent for those containers.
- With this information, the receiver can finally request the retransmission for the lost ADUs.

The problem of this approach is the possibly long convergence time, until the receivers finally locate a loss. The way suggested by McCanne/Raman to reduce this time is to use heuristics at the senders of the update packets to include the information that is probably required.

The General Data Naming is currently still under development. A more complete comparison of this approach with the approach suggested for RMFP in section 7.5 requires the implementation of both approaches and an evaluation in respect to their suitability to the requirements of existing applications.

Chapter 8

Summary and Future Work

This work features the enhancement of the Reliable Multicast Framing Protocol (RMFP). This protocol is designed to be a framework for reliable multicast transport protocols that can be integrated as so-called protocol profiles. RMFP is not designed to be located at the transport layer in the conventional layered protocol architecture, but uses the Application Level Framing (ALF) architecture to provide more flexibility and efficiency to the applications. ALF is designed to integrate the protocol layers up from the transport layer into the application to provide maximal flexibility. RMFP is designed to provide the advantages of ALF as a protocol library to simplify the use of the transport protocols.

The starting-point of this work have been early specifications of RMFP and a SRM profile. These specifications have been enhanced, design errors have been corrected, and a hierarchical naming concept based on objects has been developed for RMFP. Additionally to the SRM profile a new protocol profile for the Local Group Concept (LGC) has been developed.

To demonstrate the functionality of RMFP and to experiment with its features, RMFP and the SRM profile have been implemented as a protocol library. This implementation uses the object-oriented paradigm to define the programming interfaces; internally for the profiles, and externally for the API.

Together with test applications, the RMFP library has been tested, and the functionality of the concepts could be shown. As a major result, it could be shown, that it is possible to create ALF based protocols as link libraries, although ALF was designed to integrate the protocol as close into the applications as possible.

The analysis of the implementation and the tests lead to a number of suggestions for the future enhancment of RMFP, concerning the design of the RMFP library, the API, some implementation aspects and considerations about RMFP and SRM themselves.

A key result of the evaluation of RMFP, SRM and the RMFP library has been the development of a new naming concept to provide a new feature to applications: Semantic reliability. With this concept, based on the objects used in RMFP, receivers in a multicast session can decide about the reliability requirements for the data, not only the senders. Suggestions have been presented, how to integrate this new concept into RMFP.

The next steps in the development process are to integrate the new naming concept into the specifications and to enhance the implementation with these features. More protocol profiles have to be implemented to show, that RMFP is really flexible enough to support many protocols with different characteristics. With this upcoming implementation more tests have to be carried out, to analyze RMFP and the integrated protocol profiles more thoroughly as it has been possible during this work.

If the development of RMFP goes on, it can help to provide practical solutions for the numerous problems that are inherent reliable multicast.

Appendix A

Glossary

ADU The Application Data Unit (ADU) is introduced within the ALF design principle ([4]). It replaces the Protocol Data Unit (PDU) as the unit for data processing within a transport protocol. In contrast to the PDU concept, the ADUs are framed at the application level and not at the protocol level. This enables out-of-order processing of received data at the application, since every ADU can be identified independently within the application.

ALF Application Level Framing was first introduced by David Clark and David Tennenhouse in [4]. It is a design principle for transport protocols suggesting tight integration of the protocol into the application, the use of Application Data Units (ADUs) and out-of-order processing of received data.

ARQ Automatic Repeat Request is the general term for loss recovery mechanisms using upstream control information to request retransmissions.

FEC Forward Error Correction is the general term for loss recovery using redundancy information. The sender generates redundancy data from the original data, and the receivers can use this redundancy information to reconstruct lost original data. FEC cannot guarantee 100% reliability.

Hybrid loss recovery Loss recovery performed by mechanism combining both ARQ and FEC.

LGC The Local Group Concept (LGC) [14] is a set of protocol mechanisms to enable the reliable multicast transmission of data in a point-to-multipoint transmission. Like SRM ([9]) it is not a complete protocol specification, but a protocol template.

Member A session member is any instance that takes part in a multicast session. A member is always identified by its access point to the multicast session at the transport protocol. Thus, a member can be an application process, but not an endsystem.

In general it is also possible, that one application process has several access points and thus plays the role of several members itself.

Multicast group This term is used according to the IP multicast service. A multicast group is defined by its Internet group address [8].

Multicast port This term is used according to the UDP/IP multicast service. A multicast port specifies some kind of subchannel in a multicast group. The UDP packets send to a multicast group will always be sent one a given port and have to be received on this port again. The UDP protocol [21] is responsible for the multiplexing/demultiplexing of the packets of a multicast group.

Object The term object is used context dependent with two meanings:

- In the context of the *object-oriented implementation* of RMFP, the term object denotes the *instance* of a *class* like used in the definitions of object-oriented programming languages.
- In the context of the *protocol mechanisms*, an object is a logical aggregation of ADUs, that share some common aspect. This commonality may be defined by the application or the transport protocol.

PDU This is the abbreviation for Protocol Data Unit, a term introduced with the OSI reference model (a introduction to this can be found in [27]. The PDUs are constructed by a protocol layer and are the packets in the view of this layer. At the next lower layer, these packets can be segmented and will generally be encapsulated in the PDUs of that layer.

Protocol profile A reliable multicast protocol integrated into RMFP. A profile features the mechanisms defined by the given protocol within the framework defined by RMFP.

Protocol template Protocol templates are descriptions of protocol mechanisms, in contrast to protocol specifications. While a protocol specification is sufficient to implement compatible protocol instances, a protocol template doesn't specify all details of a protocol, e.g. the packet formats. Protocol templates are intented to be implemented together with a given application to allow a maximum of flexibility and performance.

Receiver A receiver is a session member that receives data. This does not exclude that it also can be a sender. The term receiver is used to stress the role of the member in the context given.

RMFP The Reliable Multicast Framing Protocol. Defines a framework for reliable multicast transport protocols.

Sender A sender is a session member that sends data. This does not exclude that it also can be a receiver. The term sender is used to stress the role of the member in the context given.

Session: The usage of the term *session* in this document follows the definition in [2]:

A session is a happening or gathering consisting of flows of inrormation related by a common description that persists for a non-trivial time (more than a few seconds) such that the participants (be the humans or applications) are involved and interested at intermediate times. Sessions may be defined recursively as a super-set of other sessions.

In this document a session is defined more technically as a multicast communication consisting of dependent components. This differs at the different layers of hierarchy in the protocol architecturs: Starting at the network layer, a session can be identified with a multicast group address. The transport layer can limit its scope regarding network session to some ports of a multicast group, but also combine several groups to a session at this layer. At the application layer, several transport layer sessions may be combined to an application layer session.

E.g. a conferencing application identifies a *conference* as a session, while at the transport layer there are several sessions each for *audio, videao* and *whiteboard*. At the network the transport layer session may share again one Network layer session, i.e. a single multicast group.

SRM The Scalable Reliable Multicast protocol. It provides group communication within a loose session semantics basing on the IP multicast service [8]. In its original specification [9] it was not defined as a complete protocol specification, but more like a protocol template, i.e. it described the protocol mechanisms, but not details like packet formats.

Bibliography

[1] Bengt Ahlgren, Mats Bjoerkmann, and Per Gunningberg. The Applicability of Integrated Layer Processing. Submitted to the Seventh IFIP Conference on High Performance Networking (HPN'97), 1997.

[2] P. Bagnall, R. Briscoe, and A. Poppitt. Taxonomy of Communication Requirements for Large-scale Multicast Applications. Internet Draft: draft-taxonomy, November 1997.

[3] Claude Castelluccia and Thierry Turletti. RMFP Profile for SRM. internal draft, November 1996.

[4] D. Clark and D. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *Proceedings of ACM SIGCOMM '90*, pages 201–208. ACM, September 1990.

[5] David Clark. The Structuring of Systems Using Upcalls. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 171–180. Association for Computing Machinery, Oakland, CA, December 1985.

[6] J. Crowcroft, Z. Wang, A. Ghosh, and C. Diot. RMFP: A Reliable Multicast Framing Protocol. Internet Draft: draft-crowcroft-rmfp-01, March 1997.

[7] B. DeCleene, J. Kurose, D.Towsly, et al. RMF: A Transport Protocol Framework for Reliable Multicast Applications. Internet Draft: draft-decleene-rmf-01, 1997.

[8] Steve Deering. Host Extensions for IP Multicasting. RFC 1112, August 1989.

[9] Sally Floyd, Van Jacobsen, Steven McCanne, Ching-Gung Liu, and Lixia Zhang. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing, November 1995.

[10] GSRM Homepage. WWW: http://research.ivv.nasa.gov/RMP/GSRM.html, 1997.

[11] GlobalCast Communications, Inc. *PII: Protocol Independent Interface*, August 1997.

[12] M. Handley. An Examination of MBone Performance. http://buttle.lcs.mit.edu/mjh/mbone.ps, 1997.

[13] Markus Hofmann. LGC Homepage. WWW: http://www.telematik.informatik.uni-karslruhe.de/ lgc.

[14] Markus Hofmann. Enabling Group Communication in Global Networks. In *Proceedings of Global Networking '97, Calgary, Alberta, Canada*, June 1997.

[15] Markus Hofmann and Claudia Schmidt. A Flexible Protocol Architecture for Multimedia Communication in ATM-Based Networks. In *Proceedings of 18th Biennial Symposium on Communications, Kingston, ON, Canada*, pages 297–300, June 1996.

[16] C. Huitema. The case for packet level FEC. In *Proceedings of IFIP 5th International Workshop on Protocols for High Speed Networks (PfHSN'96)*. INRIA, Sophia Antipolis, France, October 1996.

[17] Van Jacobsen. Multimedia Conferencing on the Internet. Tutorial 4, SIGCOMM '94, 1994.

[18] Bil Lewis and Daniel J. Berg. *Threads Primer, A Guide to Multithreaded Programming.* SunSoft Press, Prentice Hall, 1995.

[19] Joerg Nonnenmacher, Ernst Biersack, and Don Towsley. Parity-Based Loss Recovery for Reliable Multicast Transmission. In *Proceedings of ACM SIGCOMM '97*. ACM, September 1997.

[20] S. Pingali, D. Towsley, and J. Kurose. A Comparison of Sender-Initiated and Receiver-Initiated Reliable Multicast Protocols. In *SIGMETRICS'94*, 1994.

[21] J. Postel. User Datagram Protocol. RFC 768, August 1980.

[22] Suchitra Raman and Steven R. McCanne. General Data Naming and Scalable State Announcements for Reliable Multicast. Technical report, Computer Science Division (EECS), University of California, June 1997.

[23] Luigi Rizzo. On the feasibility of software FEC. Technical report, Universita di Pisa, January 1997.

[24] Jochen Schlick. Implementierung eines Multicast Protokolls im Internet. Master's thesis, University of Karlsruhe, Germany, October 1997.

[25] Henning Schulzrinne, Stephen Casner, Ron Frederick, and Van Jacobsen. RTP: A Transport Protocol for Real-Time Applications. RFC 1889, January 1996.

[26] W. Timothy Strayer, Simon Gray, and Raymond E. Cline. An Object-Oriented Implementation of the Xpress Transfer Protocol. Technical report, Sandia National Laboratories, California, 1995.

[27] Andrew S. Tanenbaum. *Computer Networks.* Prentice Hall, Englewood Cliffs, N.J. 07632, 1992.

[28] Martina Zitterbart. *Hochleistungskommunikation. Band 1: Technologie und Netze.* R. Oldenburg Verlag, Muenchen, 1995.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers lès Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot St Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, B.P. 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399