Wireless Media Multicast with Predictable Reliability in the Network-Integrated Multimedia Middleware

Yongtao Shuai

Submitted at : 2th May 2011
Reviewer : Prof. Dr.-Ing. Thorsten Herfet
2nd Reviewer : Prof. Dr. Philipp Slusallek
Advisors : Dr. Marco Lohse
: M.Sc. Manuel Gorius
Master's Thesis for
Yongtao Shuai

Wireless Media Multicast with Predictable Reliability in the Network-Integrated Multimedia Middleware

The Internet Protocol (IP) evolves to a universal envelope for digital data transport. Even though it is an asynchronous transmission mechanism, it becomes increasingly important for the delivery of real-time services such as digital audiovisual media. Usually, those applications require a predictable delivery time but can tolerate a specific residual error. This paradigm has to be incorporated into the design of future IP based transport protocols.

The bidirectional characteristic of available IP networks enables adaptive error coding, as a receiver is able to send feedback messages in order to signal its reception quality to the sender. With the generalized architecture on adaptive hybrid error correction (AHEC) the Telecommunications Lab has developed a versatile basis for adaptive error correction on packet level. Coding parameters are determined analytically based on a statistical model in order to serve a certain delivery time and a predictable reliability level to the application.

The Network-Integrated Multimedia Middleware (NMM) provides a software for creating networked multimedia applications based on the idea of transparently distributed flow graph. The architecture allows for integrating new streaming protocols.

In particular, the tasks to be solved are the following:

• Evaluate the support of IP multicast in currently available 802.11n hardware. Examine and refine the configuration of the corresponding devices for optimized multicast transfer.
• Elaborate characteristic parameter sets for the AHEC architecture operating in various IEEE 802.11 multicast scenarios. Consider different system parameters such as the multicast group size as well as the data rate of the multimedia stream under variable reception qualities.
• Integrate the available implementation of the AHEC protocol stack into the NMM architecture, for both directly connecting components in a distributed flow graph using the idea of a Parallel Binding (already available in NMM), as well as in a client/server streaming approach.
• Set up a wireless demonstration environment and evaluate your implementation with the multicast delivery of real-time multimedia streams.

The facilities include the NMM in an integrated software and hardware setup as well as several connections to digital TV broadcast networks. Appropriate hardware for the wireless multicast setup will be provided. Coding parameters for the AHEC architecture are obtained via an available analysis framework.

Tutors: Marco Lohse, Manuel Gorius

Supervisor: Prof. Dr.-Ing. Th. Herfet

1 http://www.nt.uni-saarland.de/publications/
Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, ..............................................  ......................................................
(Datum / Date)  (Unterschrift / Signature)
Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass die vorliegende Arbeit mit der elektronischen Version übereinstimmt.

Saarbrücken,……………………………..

(Datum / Date) (Unterschrift / Signature)

Statement in Lieu of an Oath

I hereby confirm the congruence of the contents of the printed data and the electronic version of the thesis.

Saarbrücken,……………………………..

(Datum / Date) (Unterschrift / Signature)
Abstract

Wireless media multicast in the Network-Integrated Multimedia Middleware (NMM) is not feasible due to the unreliability of IEEE 802.11 multicast. An Adaptive Hybrid Error Correction (AHEC) scheme has been proposed and also a protocol stack based on it, called Predictably Reliable Real-time Transport (PRRT), has been developed. However, the parameters of AHEC are determined analytically based on a statistical model. The practicability and the online adaptivity still need to be explored. The PRRT supports reliable multicast and is convenient to use. Thus in this thesis we integrate the PRRT into the NMM architecture and evaluate the AHEC parameters with the PRRT within NMM in various multicast scenarios.
Acknowledgments

I would like to thank ...

Prof. Dr.-Ing. Thorsten Herfet for supervising this thesis.

Prof. Dr. Philipp Slusallek for reviewing this thesis.

Manuel Gorius for his continuous and invaluable encouragement and support in all aspects.

Dr. Marco Lohse for his professional and competent advice and assistance.

Dr. Michael Repplinger for his valuable suggestions and technical support.

the team of Telecommunication Lab for their support and assistance, especially Jochen Miroll and Zakaria Keshta.

my parents and friends for their endless love, encouragement and support, especially Jing and Xiaoqi.
# Contents

1. Introduction 19
   1.1. Related Work ............................................. 20
   1.2. Problem Description ..................................... 22
   1.3. Contributions .......................................... 22
   1.4. Outline .................................................. 23

2. Foundations : Integration of PRRT into NMM 25
   2.1. PRRT .................................................... 25
   2.2. NMM ..................................................... 28
   2.3. Adaptive PRRT in NMM .................................... 31
      2.3.1. Structure of Use and Adaptation on PRRT ............ 31
      2.3.2. Workflow of NMM with PRRT ....................... 32

3. Adaptation of PRRT 37
   3.1. Characteristic of Wireless LAN ......................... 37
   3.2. Theoretical Background of Adaptation .............. 38
   3.3. Smoothing ............................................. 43
   3.4. Algorithm of Adaptation ................................ 48

4. Implementation : Integration of PRRT into NMM 53
   4.1. PRRT API ............................................. 53
   4.2. Socket Wrapper ......................................... 54
      4.2.1. PRRTSock ........................................ 55
      4.2.2. AdaptPRRTSock .................................. 57
   4.3. Adaptability ........................................... 57
      4.3.1. Optimized Parameter Set .......................... 58
      4.3.2. Super Class (Adaptability) .................... 59
      4.3.3. Lookup based Adaptation ....................... 60
   4.4. Transport Address ...................................... 62
   4.5. Transport Strategy ..................................... 64
      4.5.1. Registration ..................................... 64
      4.5.2. PRRTSenderStrategy ............................... 65
      4.5.3. PRRTReceiverStrategy ......................... 73
      4.5.4. Network Event Handling ....................... 75
      4.5.5. Setting up PRRT Sockets .................... 78
   4.6. Application of PRRT in NMM ......................... 80
      4.6.1. Instream Channel Mode .......................... 80
      4.6.2. Client / Server Mode ........................... 84
5. Test Results

5.1. Setup of Tests

5.1.1. Hardware and Software

5.1.2. Configuration of Setup

5.2. Results

6. Summary and Future Work

6.1. Summary

6.2. Future Work

Appendix

A. List of Source Code Files

B. PRRT Tutorial

C. NMM Tutorial

D. NETEM Tutorial
List of Figures

2.1.1.AHEC Scheme .................................................. 25
2.1.2.PRRT Module in the Kernel User Space ......................... 27
2.2.1.NMM Architecture ............................................ 28
2.2.2.NMM Flow Graph ............................................. 29
2.2.3.Unified Message System of NMM ............................. 30
2.2.4.Communication Channels ..................................... 30
2.2.5.Parallel Binding .............................................. 31
2.3.1.Structure of Using and Adapting the PRRT .................. 32
2.3.2.Workflow of NMM with PRRT in Client / Server model ...... 34
2.3.3.Workflow of NMM with PRRT in Instream Channel .......... 35
3.2.1.\(R_{1,RTT,P_e}(o)\) with Different \(RTT\) and \(P_e\) for Unicast Scenario with Data Rate of 4 Mbps .......... 42
3.2.2.\(R_{1,RTT,P_e}(o)\) with Different \(RTT\) and \(P_e\) for Multicast Scenario with Data Rate of 4 Mbps .......... 43
3.2.3.\(R_{1,RTT,P_e}(o)\) with Different \(RTT\) and \(P_e\) for Multicast Scenario with Data Rate of 4 Mbps .......... 44
3.2.4.\(R_{1,RTT,P_e}(o)\) with Different \(RTT\) and \(P_e\) for Unicast Scenario with Data Rate of 12 Mbps .......... 45
3.2.5.\(R_{1,RTT,P_e}(o)\) with Different \(RTT\) and \(P_e\) for Multicast Scenario with Data Rate of 12 Mbps .......... 46
3.2.6.\(R_{1,RTT,P_e}(o)\) with Different \(RTT\) and \(P_e\) for Multicast Scenario with Data Rate of 12 Mbps .......... 47
3.3.1.Smoothing with Function 3 and Function 4 on \(min = 3\) and \(max = 5\) and \(g = 0.0001\) ............... 48
3.3.2.Smoothing with Different Gravitation Factors on \(min = 3\) and \(max = 5\) ............... 49
3.4.1.Data Structure of Lookup Table for Storing Optimized Parameter Sets .................. 50
4.2.1.Socket Wrapper of PRRT ...................................... 56
4.2.2.Class Diagram of \(PRRT\) ...................................... 57
4.2.3.Class Diagram of \(AdaptPRRT\) ................................. 58
4.3.1.Class Diagram of \(Adaptability\) ............................... 59
4.3.2.Class Diagram of \(AdaptByLookup\) ............................ 61
4.4.1.Class Diagram of \(PRRTAddress\) ............................. 63
4.5.1.Class Diagram of \(PRRTSenderStrategy\) ..................... 90
4.5.2.Sequence Diagram of Indirect Connection Setup ............. 91
4.5.3.Class Diagram of \(PRRTReceiverStrategy\) ................... 92
4.5.4.Leader-Followers Event Handling for \(PRRTReceiverStrategy\) .................. 93
4.6.1.PRRT Packet Structure in Parallel Binding .................. 93
### List of Figures

<table>
<thead>
<tr>
<th>Figure Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.6.2. Class Diagram of PRRTNetSinkNode</td>
<td>94</td>
</tr>
<tr>
<td>4.6.3. Sequence Diagram of buffer sending in PRRTNetSinkNode</td>
<td>95</td>
</tr>
<tr>
<td>4.6.4. Class Diagram of PRRTNetSourceNode</td>
<td>95</td>
</tr>
<tr>
<td>5.1.1. Distribution of the AP and Wireless Stations</td>
<td>98</td>
</tr>
<tr>
<td>5.1.2. Flow Graph on the Server Side</td>
<td>100</td>
</tr>
<tr>
<td>5.1.3. Flow Graph on the Client Side</td>
<td>101</td>
</tr>
<tr>
<td>5.2.1. Performance of PRRT on Simulated Multicast Scenarios with Data Rate of 4 Mbps</td>
<td>104</td>
</tr>
<tr>
<td>5.2.2. Performance of PRRT on Multicast Scenarios (Stationary) with Data Rate of 4 Mbps</td>
<td>105</td>
</tr>
<tr>
<td>5.2.3. Performance of PRRT on Multicast Scenarios (Moving) with Data Rate of 4 Mbps</td>
<td>107</td>
</tr>
<tr>
<td>5.2.4. Performance of PRRT on Multicast Scenarios (Stationary) with Data Rate of 12 Mbps</td>
<td>108</td>
</tr>
</tbody>
</table>

**NOTE:** The structural class diagrams shown in figures do not necessarily depict all concrete details on the implementation. They are intended to further explain and formalize or the concepts and design issues.
Listings

4.1. run Method of AdaptByLookup ............................................ 60
4.2. setLocalAddress Method of PRRTSenderStrategy .................. 63
4.3. Registration of Transport Strategy at TSFactory .................. 64
4.4. requestConnection Method of PRRTSenderStrategy ................. 67
4.5. acceptConnection Method of PRRTSenderStrategy ................. 68
4.6. receiveOneWayBuffer Method of PRRTSenderStrategy ............. 69
4.7. run Method of PRRTSenderStrategy ....................................... 70
4.8. sendQueuedBuffer Method of PRRTSenderStrategy ................ 72
4.9. receiveSignal Method of PRRTReceiverStrategy ................. 74
4.10. receiveData Method of PRRTReceiverStrategy .................... 77
4.11. createLocalSocket Method of PRRTReceiverStrategy .......... 79
4.12. Interface Definition Language (IDL) Description in IPPRTNet.idl 85
4.13. Setting up PRRTSenderStrategy in PRRTNetSinkNode ........... 86
4.14. Handling Buffer and Signal in PRRTNetSinkNode ................. 87
4.15. Buffer Receiving and Producing in PRRTNetSinkNode ........... 88

NOTE: All code listings just give an overview of the implementation and aim at formu-
lating the realization of the design issues, therefore some concrete details and exceptions
handling are simplified and omitted. For further details please see the source code.
Abbreviations
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AHEC</td>
<td>Adaptive Hybrid Error Correction.</td>
</tr>
<tr>
<td>AP</td>
<td>Access Point.</td>
</tr>
<tr>
<td>ARQ</td>
<td>Automatic Repeat reQuest.</td>
</tr>
<tr>
<td>B</td>
<td>The message of type Buffer in NMM.</td>
</tr>
<tr>
<td>BLBP</td>
<td>Beacon-driven Leader Based Protocol.</td>
</tr>
<tr>
<td>dB</td>
<td>decibel.</td>
</tr>
<tr>
<td>DFS</td>
<td>Dynamic Frequency Selection.</td>
</tr>
<tr>
<td>DTIM</td>
<td>Delivery Traffic Indication Message.</td>
</tr>
<tr>
<td>E</td>
<td>The message of type Event in NMM.</td>
</tr>
<tr>
<td>FEC</td>
<td>Forward Error Correction.</td>
</tr>
<tr>
<td>FMI</td>
<td>Future Media Internet.</td>
</tr>
<tr>
<td>GE</td>
<td>Gilbert-Elliott.</td>
</tr>
<tr>
<td>HDTV</td>
<td>High-definition television.</td>
</tr>
<tr>
<td>HEC</td>
<td>Hybrid Error Correction.</td>
</tr>
<tr>
<td>HEC-PR</td>
<td>Hybrid Error Correction with Packet Repetition technique.</td>
</tr>
<tr>
<td>HLBP</td>
<td>Hybrid Leader Based Protocol.</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute of Electrical and Electronics Engineers.</td>
</tr>
<tr>
<td>i.i.d</td>
<td>Independent and Identically Distributed Channel Model.</td>
</tr>
<tr>
<td>I/O</td>
<td>Input and Output.</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol.</td>
</tr>
<tr>
<td>LBP</td>
<td>Leader Based Protocol.</td>
</tr>
<tr>
<td>MAC</td>
<td>Media-Access-Control.</td>
</tr>
<tr>
<td>MPEG</td>
<td>Moving Picture Experts Group.</td>
</tr>
<tr>
<td>NACK</td>
<td>Negative Acknowledgment.</td>
</tr>
<tr>
<td>NMM</td>
<td>Network-Integrated Multimedia Middleware.</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System.</td>
</tr>
<tr>
<td>PLR</td>
<td>Packet Loss Rate.</td>
</tr>
<tr>
<td>PRRT</td>
<td>Predictably Reliable Real-time Transport.</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Services.</td>
</tr>
<tr>
<td>RI</td>
<td>Redundancy Information.</td>
</tr>
<tr>
<td>RTP</td>
<td>Real-time Transport Protocol.</td>
</tr>
<tr>
<td>RTCP</td>
<td>Real-time Control Protocol.</td>
</tr>
<tr>
<td>RTT</td>
<td>Round Trip Time.</td>
</tr>
<tr>
<td>SDTV</td>
<td>Standard-definition television.</td>
</tr>
<tr>
<td>SNR</td>
<td>Signal-Noise-Ratio.</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol.</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol.</td>
</tr>
<tr>
<td>WLAN</td>
<td>Wireless Local Area Network.</td>
</tr>
</tbody>
</table>
Notations

$\alpha$ The transition probability from “Bad” state to “Bad” state in the GE channel model.

$\beta$ The transition probability from “Good” state to “Good” state in the GE channel model.

$\rho$ The correlation coefficient of the GE channel model. $(\rho = \alpha + \beta - 1)$

$k$ The number of data packets per coding block.

$D_{\text{target}}$ The target delay constraint.

$D_{\text{work}}$ The work delay is the time duration of one “complete” retransmission, i.e. from the time when the latest packet loss was detected by a receiver to the time when the receiver possibly receives the required packet.

$L$ The length of the encoding packet.

$\bar{N}_p$ The multipliers of parity packets for each transmission round.

$N_{\text{recv}}$ The number of the receivers.

$P_e$ Original link packet loss rate.

$\text{PLR}_{\text{res}}$ Residual packet loss rate, i.e. the real packet loss rate after applying the AHEC scheme.

$\text{PLR}_{\text{target}}$ The target packet loss rate requirement.

$R_d$ Multimedia data rate.

$RI_{\text{res}}$ Residual redundancy information.

$RTT$ Round Trip Time.
1. Introduction

Wireless networks become a significant part of future networking in every aspect such as home, office and academy. The expectation of the quality of their connectivity is increasing as more and more mobile and portable electronic devices e.g. smartphones and tablets are used for multimedia entertainment like streaming movies. Adapting to this surging demand, networking technologies will have to be ready to enable the best possible experience.

Digital audiovisual media transmission over wireless IP networks using off-the-shelf protocol support (e.g.: TCP, UDP and RTP) is facing the challenge of the compromise between the strict delivery time requirement and data reliability, especially in wireless multicast streaming: Transmission Control Protocol (TCP) provides reliable end-2-end transmission without scalability; User Datagram Protocol (UDP) can minimize the transmission delay and also be applicable in various multicast and broadcast scenarios but without any guarantee for data reliability; Real-Time Transport Protocol (RTP) supports error “correction” by packet loss detection and packet interpolation techniques on the application level via RTP profiles and payload formats and holds the time constraint like UDP, hence within limits to the format of delivery data. Some traditional mechanisms like Automatic Repeat reQuest (ARQ) and Forward Error Correction (FEC) are imposed to improve this situation. However, a pure ARQ or FEC scheme is still not able to efficiently handle the multimedia streaming over the wireless multicast networks by now: ARQ causes “feedback implosions” as a result of a massive increase of ACK/NACK (Acknowledgment / Negative Acknowledgment) if the multicast group becomes larger; FEC introduces a fixed delay by a fixed code rate even if no data is corrupted over the channel so that the bandwidth may be wasted by its inflexibility. Therefore an Adaptive Hybrid Error Correction (AHEC) scheme has been proposed to combine both approaches such that ARQ brings in the adaptivity while FEC resolves the scalability [Fmi10].

Furthermore, it is impossible to achieve sufficient reliability in a timely and scalable multimedia transmission via present popular protocols. The concept of “Predictable Reliability” enables to allow the tolerance of a certain delivery error. Hence, Predictably Reliable Real-time Transport (PRRT) [Gru09] has been developed and applies the AHEC architecture, which is parameterized by a statistical model. It serves the needs of applications with predictable reliability under predictable delay and minimizes the redundancy transmission at the same time. PRRT is a prototype implementation in the Linux kernel, which represents a new media transport protocol stack based on the UDP stack. Apart from the benefits of the UDP such as short delay and multicast/broadcast support, it can employ the AHEC parameters, which are estimated as the theoretic optimum based

1 RTP is not a transport layer protocol, instead it is usually regarded as an application layer protocol built upon UDP.
2 http://www.nt.uni-saarland.de/projects/prrt/.
1. Introduction

upon a statistical model, to meet the predictable delay and reliability requirements of
the application.

Nevertheless, the remaining problem is the rapidly varying channel quality in wireless
networks. Can PRRT run in an adaptive fashion under this rapid change, to offer the pre-
dictable reliability and delay for multimedia applications and to reduce the redundancy
on the network traffic as much as possible? And how can it accomplish this job without
reasonably lowering the performance of the applications? Can those AHEC parameters,
which are determined based on the statistical model from simplified architecture and
parameters in [Gor11], be put into the practice? Those questions constitute one motiva-
tion for this thesis. Answering all those doubts must be based on tests and application
scenarios. As an academic study, we integrate PRRT into a multimedia application soft-
ware, which should be an open source software and easily extensible for communication
mechanism.

The Network-Integrated Multimedia Middleware (NMM)\(^3\) is a software for realizing
diverse networked multimedia applications in different operating systems (e.g. Linux,
Windows and Mac OX) relied on the distributed middleware architecture. NMM, which
features network transparency, enables the access of various hardware components and
a multiplicity of multimedia formats. Also within the middleware and flow graph based
architecture, computing-intensive applications can be distributed to multiple PCs over
the network. Additionally, the open source C++ framework and networked middleware
architecture of NMM ease the extension of transport protocols and offer a good basis for
integration of communication technologies. On the other hand, while NMM is constantly
being sharpened for high-quality multimedia applications, it is hitched in the wireless trans-
mition by the unreliability in IEEE 802.11 multicast. Consequently, the exploration and
the exploitation of new streaming protocols are expected in NMM.

Altogether, this thesis is supposed to elaborate characteristic parameter sets for the
AHEC scheme operating in various IEEE 802.11 multicast scenarios and to integrate the
available implementation (PRRT) of the AHEC protocol stack into the NMM archite-
cture as well as to evaluate the integration with multicast real-time multimedia streaming
in the wireless network.

In the rest of this chapter, we first introduce related topics of reliable multimedia
streaming on different network scenarios. Then we summarize the problems we will
tackle in this thesis, list the contributions of our work and draw the outline for the thesis.

1.1. Related Work

Wireless media multicast has received wide investigation with the rapid development
of broadband wireless networks. Some other researches have been conducted and some
protocol solutions have been proposed to offer partial reliability under the strict delay
\(^3\) http://www.networkmultimedia.org/.

1.1. Related Work

The Stream Control Transmission Protocol (SCTP)\(^4\) was designed with some of the same service features of TCP and UDP. It is message oriented similar to UDP and provides reliable transmission like TCP. However, the SCTP maintains the reliability by introducing a fixed number of retransmission and positive acknowledgements. It’s rather a unicast protocol and suffers from the scalability issues like TCP.

[Pin93] have shown that selective NACK reliable multicast protocols are more scalable than the others which are based on the ACK for receiving data. The NACK-oriented Reliable Multicast (NORM)\(^5\) was proposed to reliably transport the data over an IP multicast network, by requesting repairs of missing data with selective NACK. But the main aim of NORM is to transfer bulk data in an efficient and scalable way. It’s not suitable for the delivery of real-time multimedia streams. The studies in [Tan06] showed that NORM can fulfill the PLR requirement of \(10^{-6}\) for DVB systems but not the delay constraint: the maximum latency can be even more than 2 seconds. On the other hand, NORM allows the use of packet-level FEC and feedback suppression to enhance the scalability. The researches of feedback suppression based on exponential timer are described in [Non98] and [Ada02].

RTP is widely used for the real-time media transmission. A modified RTP profile (RT-P/AVPF)\(^6\) is extended for Real-time Transport Control Protocol (RTCP)-based feedback, such that RTP is, to some degree, resistant to packet loss. However, the performance of RTP/AVPF is limited, as receivers send an RTCP packet “early”. Tan et al. also showed in [Tan06] that RTP/AVPF cannot reach the previous high PLR requirement within the latency of 100 milliseconds. FEC is one of the main methods to recovery the information from packet loss for real-time application. Adopting the Hybrid Error Correction (HEC) scheme, RTP Payload Format for Generic Forward Error Correction\(^7\) could become an adaptive FEC using RTCP receiver reports. But RTP restricts the usage of RTCP feedback.

Apart from the above solutions there are other approaches to deal with the reliability in real-time transmission. Luby et al. proposed to apply application layer FEC in IPTV services.[Lub08] This would take advantage of the known structural characteristics of multimedia streams. Razavi et al. further applies priority encoding of video streaming to adapt the extent of packet level interleaved FEC, in order to reduce the impact of FEC and interleaving upon delay.[Raz09] On the other hand, Exposito et al. presented a framework for communication services composition on partial order and partial reliability in [Exp09]. The framework aims at dynamically composing available transport mechanisms in order to provide the optimal transmission according to application’s QoS (Quality of Services) requirements. All those technologies and related ideas may be employed in wireless media multicast.

---

\(^4\) http://tools.ietf.org/html/rfc4960
\(^5\) http://tools.ietf.org/html/rfc5740
\(^6\) http://www.packetizer.com/rfc/rfc4585/
\(^7\) http://www.rfc-editor.org/rfc/rfc5109.txt
1. Introduction

1.2. Problem Description

As mentioned in prior, the emerging protocol PRRT needs to be evaluated with multicast streaming real-time multimedia in wireless scenarios and NMM requires the extension by a communication mechanism that supports reliable multicast in wireless transmission. Accordingly, this thesis focuses on the following problems:

- How to integrate the PRRT into the existing communication framework of NMM? The integration should be applicable not only to a distributed NMM flow graph but also to the client / server streaming approach.
- How to make the PRRT online adaptive to varying wireless network conditions and to implement the adaptation in NMM?
- Does the implementation realize real-time multimedia streaming applications in wireless multicast scenarios?

1.3. Contributions

As a result of research and implementation, this article contributes to multimedia streaming applications in wireless multicast scenarios. The main contributions are summarized below:

- **An approach of the adaptation on PRRT:** We realize the adaptation of PRRT in NMM by designing a socket wrapper that implements the functionality of the PRRT socket and encloses the adaptation.
- **An extension of the NMM communication mechanism:** We extend the communication mechanism of NMM by implementing a pair of new PRRT strategies, which utilize the socket wrapper.
- **The realization of the client / server streaming application with PRRT in NMM:** We implement two NMM nodes as a client and a server to support the streaming from non-NMM sources and to non-NMM sinks via PRRT.
- **Improvements on PRRT:** We facilitate many improvements on PRRT during the integration into a thoroughly designed software project such as NMM.
- **The evaluation of wireless multicast networks and the configuration of devices surrounding it:** In order to set up the optimized evaluation environment for the implementation we examine and configure the devices related to IEEE 802.11 multicast.
- **Statistical and practical analysis of the adaptation:** We do a statistical analysis of the adaptation including the selection of parameter sets and a smoothing method for the input statistics of the adaptation. We compare the theoretical analysis with the practical analysis by evaluating of real-time multimedia streaming in the wireless multicast environment.
1.4. Outline

The remainder of this thesis is structured as follows. We firstly introduce the fundamental knowledge of PRRT and NMM as a background for the integration of PRRT into NMM, in Chapter 2. Also a general overview of how to integrate the adaptive PRRT in NMM is provided. In Chapter 3 we present the theory of the adaptation on PRRT that is going to be developed. Chapter 4 explains the implementation details on the integration of PRRT into NMM and the adaptation. The test setup and results are discussed in Chapter 5. Finally, Chapter 6 covers the summary, conclusions and future work. Moreover, the list of source files and extensive tutorials of using the tools and applications related in the thesis are given in the appendixes.
2. Foundations: Integration of PRRT into NMM

In this section we will provide an abstract overview of the integration of PRRT into NMM before going into the details of the implementation. The basic concepts will also be presented regarding the architecture of PRRT and the communication framework of NMM. First of all, the introduction of PRRT and NMM surrounding the communication will be respectively given in section 2.1 and section 2.2. Section 2.3 will describe how the integration of PRRT with the adaptation works in NMM.

2.1. PRRT

PRRT is a new implementation paradigm of a protocol stack based on the AHEC scheme. It enhances the prevalent UDP stack to implement the AHEC mechanism and is integrated into the Linux kernel.¹ We firstly introduce the core of PRRT.

AHEC Scheme

![Figure 2.1.1: AHEC Scheme](Fmi10)

Figure 2.1.1 illustrates the highly flexible combination of NACK based ARQ and variable packet level FEC which is proposed in the AHEC scheme. This hybrid system is achieved as follows:

¹ For more details about AHEC and PRRT please refer to [Fmi10] and [Gru09]. In this thesis we just outline those architectures.
1. **Encoding and Sending:**
The packets at the sender are inserted into an interleaver and sent immediately. Once packet-level encoding of a block is finished, $N_p[0]$ parity packets (redundancy) can be sent in advance. For the retransmissions a packet cache is maintained and its size is subject to the target delay.

2. **Loss Detection and Feedback:**
The receiver detects the missing packets. It initiates a NACK regarding to the missing packets for parity packets from the sender if the current block cannot be decoded.

3. **Analysing Feedback and Updating the Coding parameters:**
The sender evaluates the current network parameters such as RTT and PLR for each receiver based on the received feedback and updates the AHEC parameters to adapt to variable network conditions.

4. **Retransmission and Repetition:**
In case of $k = 1$ original data packets are retransmitted, otherwise parity packets are sent. Depending on the current AHEC parameters applied by the sender, steps 1 to 4 are performed $r$ times within the allowed delay. Each component of the vector $\vec{N}_p$ except $N_p[0]$ defines the number of parity packets retransmitted in one retransmission round.

The AHEC scheme considers a number of network parameters. According to different application requirements it aims at optimizing its parameters such as $k$ and $\vec{N}_p$ under the constraints of the target PLR and delay. Table 2.1.1 lists the parameter set of the system.

**PRRT**

A protocol stack with the AHEC functionality, called PRRT, has been developed. PRRT is implemented as a module in the Linux kernel. This incurs less overhead of system calls, is low maintenance effort and supports run time loading. Socket based communication (local UDP sockets) is used between the module in kernel space and the application in user space. This communication mechanism based on the socket interface provides a lot of flexibility, e.g. allows applications to listen on a socket and send messages with arbitrary message length. As long as a pair of sockets for the PRRT module is set up on both spaces, an application can send and receive messages to and from the module, like using the common internet socket. Furthermore, the application can utilize a given

---

2 The RTT is determined by the sender subtracting the work delay from the interval between sending a packet and receiving a corresponding NACK for that packet, where the work delay is contained in that NACK; the PLR can be estimated by the sender calculating the ratio between the number of lost packets in one block of length $k$, where the number of lost packets is informed in a NACK. The computation of RTT and PLR is based on the implementation of the PRRT.

3 The parity packets are generated by packet-level encoding. For $1 \leq i \leq r$, $N_p[i]$ parity packets are sent in the $i$-th retransmission round if the sender receives a NACK.

4 The length is up to the maximum packet length allowed by the socket, e.g. For a UDP socket the theoretical limit is 65527 bytes (65535 bytes - 8 bytes UDP header).

5 The setup of those sockets is automatically performed when PRRT socket is created.
Table 2.1.1.: Parameter Set of AHEC

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R_d )</td>
<td>Multimedia data rate</td>
</tr>
<tr>
<td>( L )</td>
<td>Encoding packet length</td>
</tr>
</tbody>
</table>

**Network Parameters**

- \( \text{Packet loss model}^a \)

**Optimization**

- \( k \): The number of data packets per block
- \( \vec{N}_p = (N_p[0], N_p[1], ..., N_p[r]) \) defining the multipliers of parity packets for each transmission round, where \( N_p[0] \) refers to the number of parity packets sent in advance and \( N_p[1], ..., N_p[r] \) represent the number of parity packets for each retransmission round.

---

The Independent and Identically Distributed (i.i.d) Channel Model \([Fmi10]\) is used in this thesis to reduce the parameter space or the parameter table size respectively. It’s also applied for the simplified architecture and parameters in \([Gor11]\), which is the theoretical core for the future implementation of online calculating AHEC parameters.

interface to create, bind, control and destroy the PRRT sockets as well as write data on and read data from the PRRT sockets, like a socket API library. Figure 2.1.2 depicts the overview of the relation of these three parties.

![Figure 2.1.2: PRRT Module in the Kernel User Space [Gor10]](image)

**Features**

Some features of PRRT are listed as follows:

- **Predictable delay and predictable residual error rate**

  \(^6\) “Predictable” means here that a margin tolerance is allowable for both targets as they are achieved
PRRT aims for specific quality of services on the transmission such as maximum delay and residual error rate according to the application requirements.

- **In-order delivery**
  PRRT ensures in-order delivery so that the applications do not need to consider the reordering of the packets.

- **Partial reliable multicast**
  PRRT supports the partial reliability even in various IEEE 802.11 multicast scenarios.

- **Easy to use**
  The module can be inserted (removed) into (from) the kernel at run time of the Linux system. The implementation provides a socket API library to control the module.

### 2.2. NMM

NMM is a software architecture for multimedia applications, in which all diverse distributed resources can be transparently accessed, controlled and integrated into an application. It employs a flow graph based multimedia middleware technology to utilize resources over the network efficiently and to rise to the challenge of the heterogeneity of their underlying infrastructure. Below, we generally explain the detail step by step from the architecture to the communication channel, where we deploy the integration of PRRT into NMM.

**Flow Graph Based Middleware**

Figure 2.2.1.: *NMM Architecture*  
[Mot10a]

Figure 2.2.1 shows the core architecture of NMM. Firstly a flow graph based architecture enables distributing each big task into independent processing elements according

---

**based on the statistical model.**
to a logical workflow. This will be of benefit to the extension of the new functionality and to the parallel processing. The role of a distributed middleware is to simplify the development of network applications. On the one hand, it provides common programming abstractions and masks out low-level programming details. On the other hand, it hides the complex and heterogeneous environment (e.g. multifarious hardware resources, operating systems and communication protocols) and enables the interoperability.

**NMM Flow Graph**

An NMM flow graph describes a certain type of multimedia processing. In a directed graph each *Node* models a software component of NMM as the smallest functional element, which can produce data, process data or consume data. A node may have several inputs and outputs, each represented by a *Jack*. Associated with each jack are its supported *Formats* which specify the metadata of multimedia data. To interconnect nodes, an output jack of one node is connected to an input jack of another. The connection requires a match on the supported formats of both jacks. As shown in figure 2.2.2, a playback of MPEG audio is performed.

![NMM Flow Graph](Figure 2.2.2: NMM Flow Graph [Loh04])

**Unified Message System**

To control an object, an application or a node requires the *Interfaces* provided by that object, which are wrapped in the *Message* objects with the message type *Event*. The interface is specified with an NMM *Interface Definition Language* (IDL), similar to the Common Object Request Broker Architecture (CORBA). NMM employs this infrastructure to normalize the method-call semantics between objects residing either in the same address space or remote address space. NMM has two types of *Message* objects. Messages of type *Event* can be used for controlling node behavior. Another Message type is *Buffer*, used to carry multimedia data, while events can be used for controlling node behavior. Figure 2.2.3 demonstrates this message system in the communication between objects, with which all NMM communication is performed.

Buffers are forwarded only along connected jacks between nodes. An event is called an *Instream* event if it is delivered in the same way like the buffer. Those messages can be passed in *downstream* direction (i.e from an output jack to an input jack) and *upstream* direction on the opposite. Moreover, an application can interact with nodes by sending *out-of-band* events, those are mostly used to set particular parameters of the node. For example, *INode* is an interface for controlling and configuring a node and is defined in the IDL file *INode.idl*. 

29
Communication Channels

Within the NMM all communication is based on Communication Channels, each of which provides an abstraction for a bidirectional binding between two objects. A communication channel uses one or more pairs of Transport Strategies, which allow to transmit and receive data using a specific transport mechanism. For instance, PRRTSenderStrategy and PRRTReceiverStrategy, which we have been implementing, are such a pair of transport strategies that achieves the communication with the PRRT protocol in NMM. Similar to the events mentioned before, communication channels also have two cases: instream channels and out-of-band channels, like sketched in figure 2.2.4.

The instream channel supports the transmission of various message types (buffer or event), as seen in figure 2.2.3. NMM applies multiple transport mechanisms to parallelly handle those messages with different Quality of Services (QoS) requirements in regard to the reliability and the delay. However, a multimedia stream may possibly require messages to be transported in order. Therefore the framework of Parallel Binding is developed to maintain the ordering of messages and transmit them with different QoS at the same time, as illustrated in figure 2.2.5. It manages a set of sub-bindings, each
of which realizes the actual transmission of messages using a transport strategy, to meet different QoS requirements. Finally, a resynchronization mechanism is used to preserve the ordering.

2.3. Adaptive PRRT in NMM

In this section we outline the architecture of adapting PRRT in NMM and the big picture with respect to the integration of PRRT into NMM. This includes an overview of the typical structure for using and adapting PRRT, and the workflow of NMM with PRRT. Both will provide a comprehensible description for the relationship and the collaboration of all components which will be involved in the implementation.

2.3.1. Structure of Use and Adaptation on PRRT

Figure 2.3.1 illustrates the structure of how we use PRRT and adapt it. The information flow among the components is also shown in the figure. PRRT provides the main functionality of the protocol as a module residing in the Linux kernel and a socket API library which is used to control the module. The information communication between the module and API involves data transmission, the statistic of the PRRT network and the parameters applied in the module. For adapting the PRRT, we introduce a component “adaptation” as shown in figure. With the help of the API, this component can read out the current statistic of the PRRT network from the PRRT module, compute the optimized parameters according to the statistic and apply them on the module, in an online fashion. Note that the adaptation is done only at the sender side. This is due to

---

7 The statistic of the PRRT network consists of round trip time ($RTT$), packet loss rate ($P_e$) and the number of receivers ($N_{recv}$), as the subject parameters (inputs) for the adaptation. The objective parameters (outputs) of the adaptation includes the number of data packets per block ($k$) and the transmission round vector of parity packets ($\vec{N}_p$). For more details about the parameter symbol please refer to table 2.1.1. Further details will be discussed in section 4.3.
the fact that the sender can gather the global information of the network situations from different receivers and apply the appropriate parameters which benefit to all receivers. For data communication the application transmits and receives the multimedia data also through the PRRT socket API. In the later implementation the application is NMM.

![Diagram](image)

**Figure 2.3.1.: Structure of Using and Adapting the PRRT**

In contrast to this structure of figure 2.3.1, we might put the adaptation into the PRRT module. But we didn’t choose this alternative for three reasons:

- Only integer arithmetic is available in the kernel. Therefore, a floating point computation for the analysis of the AHEC parameters requires the corresponding conversion.

- The algorithms of the adaptation are at the moment not optimal, they might spend relative long CPU time on computing the optimized parameters if the parameters are estimated on the fly based on the current network conditions or they might take a lot of memory utilization if the algorithms apply a pre-computed parameter approach like looking up from a table of parameters. This will cost a lot of limited kernel resources and reduce the performance of the system.

- The detached adaptation provides the flexibility for applications. The applications can make a choice whether they use adaptation or just a normal hybrid scheme of the fixed FEC and retransmission mechanism. Furthermore the applications can also implement their own adaptation processing according to their requirements.

In this thesis we implement the adaptation component within a socket wrapper in NMM, which works as a socket for NMM to send and receive data via the PRRT protocol and owns the adaptability on PRRT.

### 2.3.2. Workflow of NMM with PRRT

To integrate a new transport protocol into NMM, a pair of (leaf) transport strategies is required as the low level components, which perform the actual transport mechanism supported by the protocol. On the other hand, the new transport strategies should provide essential functionality so that they can cooperate with other components within
2.3. Adaptive PRRT in NMM

NMM. Here we will introduce the basic concepts of a transport strategy and its related components, involved in the integration. In addition, we will offer a rough overview of the collaboration with them by means of the workflow diagrams, which also depict how PRRT is integrated into the system. Further details will be discussed in chapter 3 for the adaptation and chapter 4 for the implementation.

The first component we are presenting is a socket wrapper, called AdaptPRRTSock (see section 4.2), which implements the basic functionality of a PRRT socket in the object oriented structure within NMM. As the name suggests, it also envelops the adaptation into the socket. The adaptation is represented by the Adaptability (see section 4.3) of the PRRT socket. The AdaptPRRTSock can have different Adaptability policies to accomplish the different adaptation in need.

A pair of transport strategies (PRRT Strategies see section 4.5), respectively named PRRTSenderStrategy and PRRTReceiverStrategy, realizes the actual transmission via PRRT network using the AdaptPRRTSock. Each one implements the transport mechanism as one endpoint of the connection: PRRTSenderStrategy implements the one at the sender side and PRRTReceiverStrategy at the receiver side. Apart from this, a strategy must take a role in NMM in order to interact with other NMM components in the same address space. Those roles are specified as Transport Role Interfaces. Every NMM object can play a specific role by deriving from the corresponding interface and implementing it. For example, a sender transport strategy, e.g. the PRRTSenderStrategy, should take the role of the one-way buffer receiver, which is represented by the ITSOneWayBufferReceiver interface. Because the sender strategy receives multimedia buffers from the component that need to send data to the network and then forwards those buffers to the socket that performs the transmission of buffers over the network. The other way round, the PRRTReceiverStrategy acts as the one-way buffer sender on the receiver side, as shown in figure 2.3.2.

Figure 2.3.2 demonstrates how the PRRT strategies work in an NMM distributed application structure of the Client / Server (C/S) model (see section 4.6.2). In C/S application mode of NMM, two specialized nodes are established at this stage: the NetSinkNode and the NetSourceNode. They manage multiple pairs of transport strategies and select one of those to perform the communication with the transport mechanism the strategies support, like a multi-switcher on the site of multiple connections.

The NetSinkNode consumes all messages as a sink node in an NMM flow graph (representing an application) at the sender and conveys multimedia buffers from its predecessor node to the PRRTSenderStrategy where the actual sending operation on buffers is executed. Moreover, a simple signal synchronization mechanism is brought in the system for avoiding too much buffers queued in the strategy. The basic idea is that

---

8 The PRRT strategies can transmit only messages with the type buffer due to the partial reliability of the PRRT protocol. Messages of the type event must be transported in a reliable way.

9 In the workflow figures 2.3.2 and 2.3.3 each component or node is associated with its class name. Additionally for better illustrating the relationship among components, some components are marked with other names enclosed in parentheses, which means that they belong to those objects by inheriting those classes.

---

33
a sender strategy signals the NetSinkNode to forward more buffers to it dependent on how much data the strategy can afford. The NetSignalReceiver is a signal receiver that can receive the signals from the PRRTSenderStrategy e.g. the FREE_QUEUE SLOT, which indicates that the strategy needs more data. Once the FREE_QUEUE SLOT arrives at the NetSignalReceiver, the signal receiver will call the NetSinkNode to pass buffers to the strategy. Otherwise the buffer forwarding will get blocked between the node and the strategy. The same signal mechanism is applied at the receiver side. But now the PRRTReceiverStrategy is a signal receiver to the NetSourceNode, as one-way buffer sender and receiver are a signal receiver too.

Note that in this mode, it’s not necessary to run NMM applications at both sides. Any one-sided NMM application can comply with other applications that use the same transport protocol.\footnote{The transmission format of the multimedia data must of course be consistent on both sides too.}

In the following, another application model of NMM – the Instream Channel model (see section 4.6.1) – is being displayed with the workflow diagram, compared to the C/S model. The strategies in the instream channel mode behave similar to the case of the C/S mode. However, the collaboration with other components becomes more complicated, since there are other transport strategies being used simultaneously for various messages in the instream channel of NMM. Nodes between which the communication is undertaken by the strategies do not directly interact with those strategies, but through a series of components involved in the communication framework of NMM. The InstreamChannel is a high level concept which represents the connection between an output jack and an input jack. It’s responsible for the setup and configuration of the connection.

As illustrated in figure 2.3.3, those components are as follows and are briefly explained.
Figure 2.3.3.: Workflow of NMM with PRRT in Instream Channel

Based on the figure:

- **Jack** realizes a bidirectional instream communication. The **OutputJack** or the **InputJack** respectively sends or receives messages to or from the instream composite strategy set in that jack.

- **InstreamCompositeStrategy** is a top level transport strategy for realizing a (parallel) binding. It conveys the messages between a jack and an instream demultiplexer or an instream multiplexer owned by it.

- **InstreamDemultiplexer** and **InstreamMultiplexer** are the adaptors between a top level strategy and its leaf strategies, which are used in parallel binding to dispatch and reconstruct the ordered messages. Furthermore, the demultiplexer is a signal receiver to its leaf strategies while the multiplexer is an one-way buffer receiver.

Note that in the instream channel mode, the components surrounding our strategies are more than those in the C/S mode. But thanks to the flexibility and extensibility of the NMM framework, they haven’t been touched during the implementation (see chapter 4). The collaboration is established and performed automatically, once the new leaf strategies provide necessary functionality. Therefore, we don’t discuss them in details in order to avoid a high complexity. For more details please refer to [Loh05] and [Mot10b].
3. Adaptation of PRRT

This chapter will introduce the adaptation of PRRT and its algorithm in a theory level. First of all, we will review the characteristic of wireless LAN in section 3.1, in order to outline the motivation of such an adaptation. Section 3.2 will explain the optimization problem of the AHEC parameters as the background of the adaptation. Thereafter we will discuss a smoothing method for the unpredictably varying network conditions in section 3.3. Finally, the lookup based algorithm of the adaptation will be described in section 3.4.

3.1. Characteristic of Wireless LAN

In many practical networks, in particular wireless networks, packet losses mostly occur in a bursty way due to access collisions, overflows of certain queues within the transmission path and protocol issues [Fmi10]. Therefore, the Gilbert-Elliot channel (GE) model should be used while analysing the wireless networks.\(^1\) However, in order to keep the search space smaller for the later table-based adaptation, we apply Independent and Identically Distributed (i.i.d) channel model instead, which is a special case of GE model.\(^2\) to evaluate the AHEC parameters. The i.i.d model is also used in the simplified architecture for optimizing the AHEC parameters.\(^3\) That simplified architecture will be employed in the adaptation through online computing the optimized parameters according to the network conditions. So our research is a prelude to test the performance of the simplified architecture.

Moreover, as mentioned in prior, multicast is infeasible in IEEE 802.11 wireless networks owing to the absence of feedback mechanism. To avoid the feedback collision, 802.11 MAC layer protocol sends multicast packets in an open-loop mechanism. Three main problems caused by this legacy 802.11 multicast can be summarized as follows:

- unfairness between unicast and multicast traffic and even the collapse of a network,
- no retransmission for lost packets at the MAC layer
- and no adaptation of the physical transmission rate according to the channel situation.

Leader-based error recovery schemes like Leader Based Protocol (LBP), Beacon-driven Leader Based Protocol (BLBP) [Li08b] and Hybrid Leader Based Protocol (HLBP) [Li08a] are one solution to improve the standard 802.11 MAC layer protocol. Those

\(^1\) It has been shown that the GE model is a very good approximation for the packet loss model in a wireless channel.\(^4\) \(^5\)\n
\(^2\) The i.i.d model is the GE model with zero correlation coefficient (\(\rho = 0\)).
schemes are introduced and their performance is also evaluated via simulated experiments in [Li08a] and [Li08b]. In a nutshell, this approach proposes to select a leader among the receivers to send feedbacks to the sender like the common unicast transmission. It’s also shown that this multicast mechanism outperforms the standard 802.11 multicast in practice [Duj06].

However, wireless network uses a shared medium, which means that the more clients and the higher bandwidth applications run across the network, the less bandwidth can be shared per user, thereby decreasing the overall throughput and performance of the network. Many key technologies have been developed in the IEEE 802.11 standard family such as a, b, g and n, to mitigate this effect by increasing the coverage and throughput of wireless networks. The most significant one is 802.11n in recent years, which substantially boosts the network performance. Some major improvements are listed here to state the remarkable characteristic of 802.11n:

- **MIMO**: as the most significant functionality, greatly increases the data rate respectively by signal processing and spatial multiplexing.

- **Frame Aggregation**: cuts down the overhead of the 802.11 MAC to a great extent by combining multiple packets and sending them as one at a time.

- **802.11n’s Version of OFDM**: achieves higher data rate through more sub-carriers and short guard intervals as well as higher coding rate.

- **Channel Bonding**: as an option mode, further doubles the bandwidth by using two adjacent 20MHz channels as a single 40MHz channel.

- **Backward Compatibility**: allows the interoperability of all 802.11a/b/g/n stations over the network.

Unfortunately, some functionalities are restricted in this thesis owing to the setup environment and multicast test. Those restrictions will be explained in section 5.1 of chapter 5.

### 3.2. Theoretical Background of Adaptation

The adaptation of PRRT in this thesis is based on the lookup of all statistically optimized parameters. The performance of the lookup is strongly dependent on the size of the table. So we need to pick out the samples from all optimized parameters as the candidates in the table, which can represent the optimized parameter set for a specific criteria range. The selection should rely on the analysis of the optimization on the AHEC scheme.

The principle of the optimization on the AHEC scheme is to minimize the Redundancy Information (RI) while guaranteeing a certain packet loss rate requirement under a strict delay constraint. Hence, we firstly investigate the RI performance with the AHEC parameters under different conditions and constraints. Based on the principle we can formalize the optimization problem in the following way:
3.2. Theoretical Background of Adaptation

Minimize: \[ RI(o) \]
Subject to: \[
D(o) \leq D_{\text{target}} \\
PLR(o) \leq PLR_{\text{target}}
\]

where
\( o \in OPT \) is an AHEC parameter set – a tuple set of \( \{k; \vec{N}_p\} \) in which the parameters are already mentioned in Table 2.1.1. \( OPT \) is a set collecting all possible AHEC parameter sets;

\( RI \) is a function: \( OPT \rightarrow \mathbb{R}_{\geq 0} \) from some parameter sets \( OPT \) to the real numbers, which determines the redundancy required in the transmission under a specific parameter set \( opt \):

\[
RI(o) = \frac{1}{k} \left( N_p[0] + \sum_{i=1}^{r} (1 - (1 - Pr(R_{i-1} < k))^N_{\text{recv}}) \cdot N_p[i] \right)
\]

3: \( D \) is a function: \( OPT \rightarrow \mathbb{R}_{\geq 0} \) from some parameter sets \( OPT \) to the non-negative real numbers, which states the total delay of a packet under a specific parameter set \( o \):

\[
D(o) = \frac{RTT}{2} + k \cdot T_s + r \cdot RTT
\]

and \( PLR \) is a function: \( OPT \rightarrow \mathbb{R}_{\geq 0} \) from some parameter sets \( OPT \) to the non-negative real numbers, which denotes the residual packet loss rate under a specific parameter set \( o \):

\[
PLR(o) = \frac{1}{k} \sum_{i=1}^{k} \sum_{j=\max(n-k+1,i)}^{n-k+1} i \cdot \binom{k}{j} \binom{n-k}{j-i} \cdot P_e^j(1-P_e)^{n-j}
\]

5 The optimal solution of the problem is

\[
\text{opt} = \arg \min_{o \in OPT} RI(o)
\]

For simplification we express the optimization problem with the notation:

\[
RI^{d,plr}(o)
\]

3 For more details about the formula please refer to [Gor11]. Here we just express the meaning of the notations. \( R_i \) is a random variable indicating the number of packets collected at the receiver after i-th round. The probability of \( R_i < k \) is:

\[
Pr(R_i < k) = \sum_{j=\max(n-k+1,i)}^{n[i]} \binom{n[i]}{j} P_e^j(1-P_e)^{n[i]-j}
\]

where the sender has sent \( n[i] = k + \sum_{j=0}^{i} N_p[j] \) packets in the i-th parity transmission round among the r transmission rounds.

4 Without explicitly specified, all numbers involved in this chapter are non-negative numbers.

5 The derivations of the functions \( RI, D \) and \( PLR \) please refer to [Gor11].
3. Adaptation of PRRT

and thereupon the optimal solution of the problem is:

\[ \text{opt}^{d,plr} = \arg \min_{o \in \text{OPT}} RI^{d,plr}(o) \]

where superscripts \( d \) and \( plr \) respectively denote a specific \( D_{\text{target}} \) value (in milliseconds) and a specific \( PLR_{\text{target}} \) value. For example, \( RI^{200,10^{-6}}(o) \) states that we minimize the RI under the target delay with 200 ms and the target packet loss rate with \( 10^{-6} \) and the corresponding optimal solution \( \text{opt}^{200,10^{-6}} \) represents an AHEC parameter set for \( RI^{200,10^{-6}}(o) \).

As our scenario has various network conditions (the number of receivers \( N_{\text{recv}} \), round trip time \( RTT \) in milliseconds and original packet loss \( P_e \)), which already arise in the formula, each of those can affect our optimization problem as a coefficient. We refine the optimization problem by introducing those coefficients in the notation with subscripts:

\[ RI^{d,plr}_{N_{\text{recv}},RTT,P_e} \]

Therefore, each possible combination of all constraints (\( d \) and \( plr \) and coefficients \((N_{\text{recv}}, RTT \text{ and } P_e)\)) can represent a sub optimization problem of RI. For instance, we can use a notation \( RI^{200,10^{-6}}_{1,50,0.01} \) to denote an optimization of the AHEC scheme for a particular scenario as shown in table 3.2.1\(^6\).

Note that not every sub optimization problem has an optimal solution, i.e. an optimal solution may not be found for every arbitrary combination of all constraints and coefficients. Figure 3.2.1 shows the RI of the optimized parameter sets with different \( RTT \) and \( P_e \) as well as \( N_{\text{recv}} \) for \( RI^{200,10^{-6}}(o) \) with the data rate of 4 Mbps.

From figure 3.2.1 we learn that

- \( RTT \) vs \( RI \). Generally, the change of the \( RTT \) doesn’t constantly affect the RI much if \( P_e \) is fixed, compared to the effect of \( P_e \) on the RI in the case of a fixed \( RTT \). This is because the optimized parameter sets of two consecutive points are close or even same. For example, the optimized parameter sets with \( P_e = 0.05 \) and \( RTT = 40 \) or \( RTT = 50 \) are \( \{k = 1, \vec{N}_p = (0,1,3)\} \), that means that the AHEC system uses a pure ARQ scheme at that time. But along with the increasing \( RTT \) the RI may have a jump at some point, because the parameter set of the system in most cases changes from a pure ARQ scheme to a pure FEC scheme or a hybrid scheme. The point of \( RTT = 40 \) and \( P_e = 0.1 \) is such a point at which the RI increases from 11% at \( RTT = 30 \) to 38% and the optimized parameter set is switched from \( \{k = 1, \vec{N}_p = (0,1,1,4)\} \) to \( \{k = 58, \vec{N}_p = (22)\} \). This property implies that we can select a parameter set whose \( RTT \) is as high as possible before the RI changes much, as a “reference parameter set” for the adaptation. Since we are going to adapt the PRRT through a table-based algorithm, to minimize the table size we should choose those sample parameter sets which can represent other

---

\(^6\) Some parameters in the scenario table are not explicitly expressed in the notation such as data rate, packet length and work delay, since we regard them as constants in the optimization, even as fixed values in the experiment. A typical Standard-definition television (SDTV) has average data rate of 4 Mbps and seven TS packets per IP packet. Additionally we evaluate the case of data rate of 12 Mbps, which is the typical data rate of Digital Video Broadcasting - Satellite - Second Generation (DVB-S2).
3.2. Theoretical Background of Adaptation

Table 3.2.1.: Application Targets and Parameters for a unicast Scenario

<table>
<thead>
<tr>
<th>Notation</th>
<th>Items</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>RI$^{200,10^{-6}}$</td>
<td>Delay Constraint: $D_{\text{target}}$</td>
<td>200 ms $^a$</td>
</tr>
<tr>
<td></td>
<td>PLR Requirement: $PLR_{\text{target}}$</td>
<td>$10^{-6}$</td>
</tr>
<tr>
<td></td>
<td>Number of Receivers: $N_{\text{recv}}$</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Round Trip Time: $RTT$</td>
<td>50 ms</td>
</tr>
<tr>
<td></td>
<td>Original Loss Rate: $P_e$</td>
<td>1%</td>
</tr>
<tr>
<td></td>
<td>Data Rate: $R_d$</td>
<td>4 or 12 Mbps</td>
</tr>
<tr>
<td></td>
<td>Packet Length: $L$</td>
<td>1316 bytes</td>
</tr>
<tr>
<td></td>
<td>Work Delay$^b$: $D_{\text{work}}$</td>
<td>20 ms</td>
</tr>
<tr>
<td></td>
<td>Bandwidth Limit$^c$: $B_{\text{max}}$</td>
<td>8 or 18 Mbps</td>
</tr>
</tbody>
</table>

$^a$ The delay constraint is adjusted to 200 ms, because we observed long error bursts in the tests of $D_{\text{target}} = 100$ and $D_{\text{target}} = 150$. This might be due to the fact that the $RTT$ in wireless networks could arise up to 140 ms, as shown in test results (see section 5).

$^b$ An additional constant is considered in the delay function which is the sum of the response times at each host. In our case, the duration is 10 milliseconds respectively at the sender and the receiver.

$^c$ The optimization of the parameters incorporate a bandwidth limit of 8 and 18 Mbps, which respectively corresponds to the data rate of 4 and 12 Mbps. This might result in no optimum found under the constraints, as seen in the following.

parameter sets with similar performance as much as possible, as reference parameter sets stored in the table. For example, in the case of $P_e = 0.08$, the parameter set at the $RTT = 50$ could be used by the AHEC scheme for those cases with $RTT \leq 50$ and the one at the $RTT = 80$ for other cases with $50 < RTT \leq 80$.

- $P_e$ vs RI. A similar situation occurs for $P_e$ while $P_e$ is changing if $RTT$ is fixed. The RI also has a turning point where the system mostly switches from a pure ARQ scheme to a FEC based scheme. Therefore, all turning points regarding to $P_e$ could be used as the reference parameter sets in the adaptation. Additionally, the increase of the $P_e$ by 1% generally causes the increase of the RI by roughly from 1% to 3%. So, besides the turning points, we sample the optimized parameter sets which use an FEC based scheme by 2% and combine those cases (close one by one) with the same optimized parameter set as one sample. Furthermore, the parameters of those points using the ARQ scheme mostly have one more transmission round or one more parity packet in the last retransmission or even are same, compared to their preceding points (with the PLR from 0% to 40%), if $RTT$ is fixed. This allows us to choose the point with the highest $PLR$ as a reference to those ones with the same $RTT$. Because the actual RI increases only in need by retransmissions, even if a stronger parameter is used for the “mild” network situation.

- No solutions found. Due to the bandwidth limit, our algorithm cannot find an

$^7$ We might choose the parameter set at the $RTT = 100$ as a reference for those points with $60 \leq RTT \leq 100$ and $P_e = 0.08$. But considering the difference of the sampling points when $P_e < 0.7$, we distinguish the reference parameter sets between $RTT = 80$ and $RTT = 100$. 

41
3. Adaptation of PRRT

![Diagram](image)

**Figure 3.2.1.** $R_{I_{1,RTT,P_e}(o)}^{200,10^{-6}}$ with Different RTT and $P_e$ for Unicast Scenario with Data Rate of 4 Mbps

...optimized parameter set for those cases of $P_e \geq 0.3$, which are indicated in the figure with $RI = 1$.

Overall, we sample the optimized parameter sets for the unicast scenario mentioned in table 3.2.1, as the green squares marked in figure 3.2.1.

Then we consider the multicast scenarios for three receivers and five receivers with the same requirements in previous unicast scenarios. Their RI performance is respectively depicted in figure 3.2.2 and figure 3.2.3 with different RTT and original packet loss rate. In both multicast cases the RI has similar properties to the unicast scenario except that the curve is more smooth and there are no explicit turning points. This is due to the fact that the HEC scheme is earlier applied in the multicast scenario than in the unicast scenario. Also the reference parameter sets for both cases are marked with the green squares in the figures.

Finally, we take a look at the cases where the data rate of the multimedia stream is 12 Mbps, as illustrated in figure 3.2.4, 3.2.5 and 3.2.6. Generally, the curve properties of the RI performance in the case of a higher data rate is similar to the previous scenarios. The sampling criteria of optimized parameter sets for the data rate of 4 Mbps, as described above, can be applied for the cases with the data rate of 12 Mbps. It must be noted that

---

8 Actually, two cases ($RTT = 10$ and $RTT = 20$) at $P_e = 0.3$ have a solution. But their RI is very high and close to 1.
the non-solution problem occurs comparatively early along the increase of the PLR and the curve is not so smooth as the one of 4 Mbps data rate. This is because we limit the number of data packets per block ($k \leq 10$) while optimizing the parameters.

3.3. Smoothing

The network conditions vary rapidly in WLAN, especially the fluctuation of RTT and PLR is unpredictable and large at times. This variance would severely lower the performance of PRRT, since computing an optimized parameter set and applying it in the module are expensively in a time-intensive multimedia application. Even though the lookup based adaptation is comparatively cheap after the selection of candidates in the table, frequent switching of AHEC parameters in the module would increase the workload of the module. On the other hand, a smoothing on the input parameters of the adaptation is important if the online computing of parameters is applied later, which would be expensive. The smoothing can reduce the frequency of computing new parameters. As a result we introduce an adequate smoothing function for our adaptation.

---

9 We found that the performance of PRRT of the current version is susceptible to the WLAN when a large $k$ is used at the high data rate. Therefore we choose the ARQ scheme and the FEC scheme with small $k$ in our later tests.
Firstly we examine the properties our smoothing function should have:

- it should filter the frequent fluctuation like the behavior of a low pass filter.
- the smoothed value must not be smaller than the original value. Otherwise the optimal parameters computed with that value can not satisfy the requirements. To formalize this, we define the smoothing function $f(x) : \mathbb{R} \rightarrow \mathbb{R}$ where it always holds
  \[ \forall x \in \mathbb{R}_{\geq 0} : f(x) \geq x. \]  \hspace{1cm} (3.1)

Therefore, in the following we design a special smoothing function step by step. To guarantee the prerequisite 3.1, a low pass filter can not be applied in our case, because it filters out some high peaks and it needs to see certain past samples to filter the high frequency, consequently introduces the delay while smoothing. These two characteristics determine that a low pass filter cannot fulfil our requirement.

Note that due to this criterion the function we design is a non-smoothing function. It’s a combination out of smoothing and extrapolation. We cannot smooth the values if those are increasing. Because the network condition is greatly uncertain, values may raise abruptly. In that case we must heighten the “smoothed” value at least as high as the original value, if the last “smoothed” value is lower than that value. Therefore, our procedure is rather an extrapolation. But the difference is that our method doesn’t estimate a new value, instead reevaluate the current value.
We consider two main cases while sketching a smoothing function: if the input value increases, the output should increase too, so that the smoothed value keeps larger than or equal to the original one; if the input value decreases, the output should slowly drop so that an optimized parameter set can be chosen to reduce the RI and a weaker parameter set wouldn’t be switched back to a stronger one again if the next input value raises. According to these two considerations, we use models of the linear growth and free fall to design the smoothing as follows:

\[
 f(x_i) = \begin{cases} 
 f(x_{i-1}) + x_{i} - x_{i-1} & \text{if } x_{i} > x_{i-1} \\
 f(x_{i-1}) - g \cdot t^2 & \text{if } x_{i} < x_{i-1} \\
 f(x_{i-1}) & \text{if } x_{i} = x_{i-1} \\
 x_0 & \text{if } i = 0 
\end{cases}
\]  

(3.2)

where

- \( x_i \) is a discrete sequence of \( x \) with \( i \in \mathbb{N} \);
- \( t \) is an internal parameter of the function, which increases by 1 if \( x_{i} < x_{i-1} \) and is reset to 0 if \( x_{i} > x_{i-1} \);
- \( g \) is a gravitation factor that controls how fast the smoothed value should drop.

However, function 3.2 isn’t bounded, i.e. it will continue rising if \( x \) oscillates. Furthermore, it may not hold the prerequisite 3.1 if \( x \) decreases. Therefore, we make some modifications to the smoothing function 3.2 and get a new one as follows:
Figure 3.2.5.: $RI_{3,RTT,P_e}(o)$ with Different RTT and $P_e$ for Multicast Scenario with Data Rate of 12 Mbps

Due to the unboundedness of function 3.2, $max$ and $f(x_{i-1}) - x_i$ are introduced into function 3.3 to avoid the higher divergence between the smoothed value and the original one. At the same time $min$ and the fourth case in the function are used to let the output always be higher than the input when $x$ is decreasing. On the other hand, the first case in function 3.3 can keep the smoothed value unchanged if the last smoothed value already stays in the given range between $min$ and $max$ to the current value.

Additionally, as shown in figure 3.3.1, function 3.3 is overreacting when handling the increase of the input. The point marked with a black square in the figure shows that the output behaves with the same steep increase as the input at that point even if the last smoothed value is already greater than the current one. This may results in little
3.3. Smoothing

smoothness in some cases, as the curve from 282 s shown in figure 3.3.1. To deal with those cases, we consider two more cases when the input rises like:

$$f(x_i, \text{min}, \text{max}) = \begin{cases} 
    f(x_{i-1}) & \text{if } \text{min} < f(x_{i-1}) - x_i < \text{max} \\
    f(x_{i-1}) + x_i - x_{i-1} & \text{if } 0 < x_i - x_{i-1} < \text{max} \text{ and } f(x_{i-1}) - x_i < \text{max} \\
    x_i + \text{max} & \text{if } \text{max} \leq x_i - x_{i-1} \text{ and } f(x_{i-1}) - x_i < \text{max} \\
    f(x_{i-1}) - g \cdot t^2 & \text{if } x_i < x_{i-1} \text{ and } x_i \leq f(x_{i-1}) - g \cdot t^2 \\
    x_i + \text{min} & \text{if } f(x_{i-1}) - g \cdot t^2 < x_i < x_{i-1} \\
    f(x_{i-1}) & \text{if } x_i = x_{i-1} \\
    x_0 & \text{if } i = 0
\end{cases}$$

The new function 3.4 can prevent from steep increasing when the last smoothed value is already greater than the current value and can behave more smoothly, as shown in figure 3.3.1.

Now, our smoothing function has three factors which can adjust the smoothness of the function: \( \text{min} \), \( \text{max} \) and \( g \). Since \( \text{min} \) and \( \text{max} \) act as a threshold for smoothing in the function, they affect how apart the smoothed curve is from the original value. For simplification we choose \( \text{min} = 3 \) and \( \text{max} = 5 \) as the default values in the analysis. So we evaluate the significant factor \( g \) for the smoothness of the function. Figure 3.3.2 depicts

Figure 3.2.6.: \( RI_{5,\text{RTT},P_e}(o) \) with Different RTT and \( P_e \) for Multicast Scenario with Data Rate of 12 Mbps
3. Adaptation of PRRT

5 different gravitation factors. As the factor decreases, the corresponding curve becomes more smooth. But smoother output is more insensitive to the change of the input. This will cost a lot of RI if the PLR or the RTT drops, since an improperly stronger parameter set is applied in the system due to overestimating the current situations. Therefore we select $g = 0.0001$ for the later tests, since it can master the drop maximum by roughly 2.7% in 30 iterations.

3.4. Algorithm of Adaptation

The algorithm of the adaptation is a lookup based algorithm, i.e. we first precalculate a table and store the table in the static program storage, then we fetch the value we use from the table rather than computing it when the value is needed. The savings in term of the processing time can be significant, since retrieving a value from the memory is generally faster than an “expensive” computation.

However, the performance of a table lookup algorithm is dependent to the size of the table and the data structure used in the table. Regarding to the size we select the candidates of the table with care as described in section 3.2. And we utilize the data structure `map` from the C++ STL Container as our data structure for the lookup. Because the method `lower_bound` provided by the `map` can simplify our lookup operation: it has logarithmic complexity in the table size; it returns the first optimized parameter set whose corresponding network parameter (as a key stored with the parameter set in
3.4. Algorithm of Adaptation

Figure 3.3.2.: Smoothing with Different Gravitation Factors on $\text{min} = 3$ and $\text{max} = 5$

The map) doesn’t compare less than the current network parameter. Such an optimized parameter set fulfils the application requirements and minimizes the RI at the same time.

Figure 3.4.1 illustrates how optimized parameter sets are stored in the lookup table. As discussed before, we have four input parameters for determining the optimized parameter set we need: delay constraint ($D_{\text{target}}$), the number of receivers ($\#\text{receivers}$), round trip time ($\text{RTT}$) and packet loss rate ($\text{PLR}$). Therefore, we design four table structures for input parameters using the map. Each table of an input parameter (category) stores all elements (items) each associated with a key (label). An item can be uniquely identified by its label in a category. An element can be another table in form of a nested table. As shown in figure 3.4.1, the table of the delay constraint subsumes three tables of the number of receivers with different delay constraints (100 ms, 150 ms and 200 ms) and each table of the number of receivers can also subsume multiple tables of the round trip time and so on. A leaf table, PLR table in our case, consists of multiple optimized parameter sets, each of which represents an optimized parameter set for its associated PLR. For example, in the figure, “Optimum” with blue color states the optimized parameter set in a unicast scenario with delay constraint of 150 ms, RTT of 20 ms and PLR of 1%. To search such an optimal solution, the algorithm starts the lookup from the top table (the root of the tree), the table of delay constraint in this case, and ends it in a specific leaf table (a leaf of the tree). The complexity of such a lookup is:

11 The PLR requirement ($\text{PLR}_{\text{target}}$) is also an input parameter. We don’t take it into account in this thesis, since the PLR requirement is set to $10^{-6}$, as suggested by the Digital Video Broadcasting - Internet Protocol (DVB-IP) specification for digital TV services over IP networks.
Each Element contains a table of #receivers associated with a delay constraint.

Each Element contains a table of the RTT associated with the number of receivers.

Each Element contains a table of the PLR associated with an RTT.

Each Element contains an optimized parameter set associated with the PLR.

Figure 3.4.1: Data Structure of Lookup Table for Storing Optimized Parameter Sets

\[ O(\log(N_{\text{delay}})) + O(\log(N_{\text{recv}})) + O(\log(N_{\text{RTT}})) + O(\log(N_{\text{PLR}})) \]

where \( N_{\text{delay}}, N_{\text{recv}}, N_{\text{RTT}} \) and \( N_{\text{PLR}} \) are respectively the number of elements in a table of delay constraint, the number of receivers, RTT and PLR. Since the complexity is logarithmic, the run time of the adaptation would be acceptable, if the size of the table is well arranged. It should be noted that the setup of the table (i.e. storing the parameter sets) requires \( O(N \log(N)) \) time where \( N \) is the number of the parameter sets to be stored. Because the complexity of inserting all elements in all maps is:

\[
O(\sum_{i=1}^{N_{\text{delay}}} \log i) + O(\sum_{i=1}^{N_{\text{recv}}} \log i) + O(\sum_{i=1}^{N_{\text{RTT}}} \log i) + O(\sum_{i=1}^{N_{\text{PLR}}} \log i)
\]

\[
= O(\log(N_{\text{delay}}!)) + O(\log(N_{\text{recv}}!)) + O(\log(N_{\text{RTT}}!)) + O(\log(N_{\text{PLR}}!))
\]

\[
= O(N_{\text{delay}} \log N_{\text{delay}}) + O(N_{\text{recv}} \log N_{\text{recv}}) + O(N_{\text{RTT}} \log N_{\text{RTT}}) + O(N_{\text{PLR}} \log N_{\text{PLR}})
\]

\[
= O(N \log N)
\]

where each insert operation takes \( O(\log(n)) \) time with respect to the current size \( n \) of the map. The equal sign marked with * holds, since:

\[
\forall n \in \mathbb{N}_{>0} : \quad \left(\frac{n}{2}\right)^{\frac{n}{2}} \leq n! \leq n^n
\]

\[
\implies \log\left(\left(\frac{n}{2}\right)^{\frac{n}{2}}\right) \leq \log(n!) \leq \log(n^n)
\]

\[
\implies \frac{n}{2} \log\left(\frac{n}{2}\right) \leq \log(n!) \leq n \log(n)
\]

\[
\implies \log(n!) = \Theta(n \log(n))
\]

According to the implementation (see section 4.3.3), the table of the delay constraints is created when the algorithm object is instantiated; other tables associated with a specific
number of receivers are set up and corresponding parameter sets are stored when the first search operation for that number of receivers is performed.
4. Implementation: Integration of PRRT into NMM

In this chapter we discuss the implementation of the integration of PRRT into NMM. We firstly start with the API provided by the PRRT in section 4.1. Then we introduce the basic element of socket based transport mechanism in NMM – the Socket Wrapper – in section 4.2. Section 4.3 explains the adaptation on the PRRT socket. Section 4.4 and 4.5 describe the low level components which perform the actual transport mechanisms. Finally, the application of the PRRT in NMM will be referenced in section 4.6. To keep the thesis compact, we don’t reveal all the methods of the classes and their details in the following sections. More details can be found in the source files.

4.1. PRRT API

The PRRT API comprises a library for developing applications that perform the transmission via the PRRT protocol, just like a common BSD socket API. Actually it is rather an interface for the communication between user space and kernel space, by which the corresponding functions or methods are invoked in the PRRT kernel module, if a function or method in the interface is called by the application from the user space. This list is a summary of functions or methods supported by the PRRT socket API library:\footnote{We just show the functions and methods we used during the integration. For other functions and methods as well as further details please refer to [Gru09] and the source code of PRRT.}

- `prrt_socket()` creates a new PRRT socket and allocates system resources to it.
- `~prrt_socket()` destructs the PRRT socket and closes it.
- `bind()` associates the socket with a specific local IP address and port number. On the sender side it’s used for receiving feedbacks from receivers; on the receiver side it’s used for receiving data from the sender.
- `connect()` assigns a remote IP address and port number to the socket. The remote address and port are referred to the feedback address and port on the receiver side or the receiver address and port on the sender side.
- `send()` and `recv()` are respectively used for sending and receiving data to / from a remote PRRT socket.
- `close()` closes the socket and releases resources allocated to it.
4. Implementation : Integration of PRRT into NMM

- **setsockopt()** sets a certain socket option for the socket. Options offered by the PRRT socket at the moment are given below:
  - option_FEEDBACK enables or disables sending feedbacks through a receiver.
  - option_REPLACE enables or disables sending replacements packets through a sender.
  - option_CODING enables or disables the coding of blocks.
  - option_TIMEOUT states if the receiver stops receiving after a timeout or not.
  - option_RCVBUF is used to set the maximum socket receive buffer in bytes.

- **set_param()** sets a particular socket parameter for the socket. The PRRT socket provides the following parameters regarding to the AHEC scheme:
  - param_D_target (target delay) specifies the allowed delay for a packet in milliseconds. Within the period the decoding of that packet has to be finished. This delay includes the time of original packet transmission and all retransmission for that packet if needed.
  - param_PLR (target packet loss rate) specifies the packet loss rate required by the application in per mil.
  - param_D_FEC states the delay for coding a FEC block in milliseconds.
  - param_CodingParams determines the parameters related to the block coding e.g. the number of data packets and redundancy packets per block and redundancy packets sent in advance.
  - param_RetransmissionRounds denotes the number of retransmissions for each retransmission round in a vector form.
  - param_T_replace is used to set the replace time of the system in milliseconds, which consists of the RTT as well as work delay on the sender and the receiver. It’s supposed to override the internal RTT for replace time measurement.

- **getStats()** retrieves the current statistic of packet sending or receiving for the socket. At the moment two statistical values are available: the last group RTT (lastGRTT) and PLR (lastGPLR) among all receivers, which represent the worst cases within the group.

- **getSockfd()** retrieves the file descriptor of the PRRT socket, which can be used to check the state of a socket, e.g. to see if data can be read from the socket.

4.2. Socket Wrapper

PRRT is very handy especially for the integration in applications through its socket API. However, to integrate PRRT into NMM, we introduce a socket wrapper of PRRT, since a

---

2 All options in the PRRT socket API are represented by an option object that specifies the option to be set and the value for that option.

3 All parameters in the PRRT socket API are represented by a parameter object that specifies the parameter to be set and the value for that parameter.
functional adaptation is built into NMM to make PRRT in NMM work online adaptively over the varying wireless network conditions. This new function along with the PRRT module is encapsulated into the socket wrapper so that an adaptive PRRT on the socket level is available in NMM like a “black box”.

In addition, the introduced socket wrapper provides us other advantages in the integration of PRRT into NMM. We summarize those as follows:

- being able to develop the adaptation of PRRT.
- encapsulating the adaptation and PRRT module into a wrapper on the socket level.
- being usable on the transport strategy level (see section 4.5) of NMM. It acts as a bridge between PRRT and NMM and minimises the implementation of the transport strategy as well as prevents the strategy from handling more tasks on low-level socket other than it is responsible for while using PRRT.
- being convenient for debugging. Since the PRRT module is loaded into the kernel of the operating system, it’s inappropriate to build the debug messages in it to track the bugs.
- being easy to handle or get around some bugs during the development if those were still open in the PRRT module during implementing.

Under considering the features of Object-Oriented Programming, we divide this socket wrapper into three main units in the implementation:

- a pure wrapper of the PRRT module, which corresponds to the class PRRTSock,
- the functional module of the adaptation, which is represented by the class Adaptability,
- and an adaptive PRRT socket, which is utilised in the transport strategy of NMM to transport data between a pair of PRRT sockets. Its corresponding class, called AdaptPRRTSock, derives from PRRTSock and can own one or more different instances of Adaptability.

The class diagram in figure 4.2.1 shows the relation of those classes. In the following subsections we will explain them respectively.

### 4.2.1. PRRTSock

A socket wrapper with the class name Sock is already available in NMM for the transmission via TCP/UDP protocol. This class wraps up Berkeley socket API into a better compatible interface for NMM, by which NMM can control the socket, e.g.: create a socket, establish a connection between two sockets and set the socket options etc. Sock is derived as a main class by SockStream and SockDGram, which are respectively used as the TCP socket and the UDP socket to transport data in the TCP-based and UDP-based transport strategies. However, the PRRT socket cannot reuse Sock, because the PRRT socket API is different to the API library of Berkeley sockets, which is used in Sock:

---

4 The concepts and principles around OOP can be found in [Pre01].

5 Different subclasses can be derived from this base class dependent on the algorithms of different variants of adaptation. More detail will be discussed in the later sections.
they use different data structures;

- not all option methods in \texttt{Sock} are supported by the PRRT socket;

- some functions or methods of these two socket APIs require different parameter types.

Therefore, a new socket wrapper independent from \texttt{Sock} is created for the PRRT socket with the class name \texttt{PRRTSock}. The responsibility of this class is similar to that of \texttt{Sock}. It can

- create and release resources allocated to a PRRT socket;
- initialize the PRRT socket;
- translate the attributes (e.g.: IP address, parameters and options) between PRRT socket API and NMM;
- set and change the parameters and options of the PRRT socket;
- send and receive the data;
- and read out the feedback statistic from the PRRT socket.

The methods and their brief explanations of \texttt{PRRTSock} are shown in the class diagram of the figure 4.2.2.

4.2.2. \texttt{AdaptPRRTSock}

To make PRRT online adaptively used in NMM, a class named \texttt{AdaptPRRTSock} is implemented, which derives from \texttt{PRRTSock} and additionally has the function of dynamically optimizing the performance of PRRT. \texttt{AdaptPRRTSock} is an NMM socket of PRRT, which
4.3. Adaptability

Adaptability is implemented as a base class to perform the functional adaptation of PRRT. It’s responsible for setting and controlling of the adaptation, e.g. storing a pointer to an AdaptPRRTSock, starting or stopping the adaptability. Since the adaptation is operated under continuously analysing the instant feedback statistic and optimizing the parameters of the PRRT socket, Adaptability runs on a separate thread to perform the job in a loop. This can be done by deriving from the ThreadedObject, which is the base class for all classes that need their own thread.

However, the real implementation of the algorithm of adaptation is addressed in the subclasses of Adaptability. This pattern is of great benefit to the development of the adapting PRRT socket:

- AdaptPRRTSock can adopt different types of adaptation by selecting the modes.
- Hybrid adaptation can be available in AdaptPRRTSock through the combination of

Figure 4.2.2: Class Diagram of PRRTSock
4. Implementation: Integration of PRRT into NMM

Figure 4.2.3: Class Diagram of AdaptPRRTSock

- It's handy in the later development for testing, analysing and comparing various types of adaptability.

Figure 4.3.1 gives us an overview of the structure and the methods related to the adaptability. Note that we only implement one algorithm based on table lookup (AdaptByLookup) in this thesis, other possible adaptation would be an important piece of future work. The following sub sections will explain more details of how the adaptation on PRRT is implemented here.

4.3.1. Optimized Parameter Set

The adaptation on the PRRT is to optimize the AHEC parameters based on the network conditions and constraints and to apply the optimal parameters on the PRRT socket. We introduce a new class to represent those parameters, called OptimizedParameterSet. The OptimizedParameterSet consists of the AHEC parameters: the number of data packets per block \((k)\) and the multipliers of parity packets for each transmission round \((\vec{N}_p)\) in a vector form.\(^6\) This class can be used in the adaptation. For example, in lookup based algorithm of the adaptation, an instance of OptimizedParameterSet can be used to store in the table. Furthermore, the functionalities like the override of the comparison operator and assignment operator as well as conversion of parameters to human readable string form are very convenient for utilization and debugs in the adaptation.

\(^6\) There are another parameters contained in OptimizedParameterSet such as the redundancy packets per block and the delay of FEC, since the PRRT module requires explicit setting of those parameters at the moment.
4.3. Adaptability

Adaptability

#m_sock: AdaptPRRTSock*
#m_mutex: ThreadMutex
#m_cond: ThreadCondition

+start()
+stop()
+setSock(AdaptPRRTSock*)
#run()
+applyOptParamSet(const OptimizedParameterSet*)
+isStatsChanged() : bool
+isOptParamSetChanged(const OptimizedParameterSet*) : bool

Responsibilities
-- Base class of the classes for the adaptation.
-- provides basic control functions for the adaptive PRRT socket.

A base class for all classes that need their own thread.
This class is already available in NMM.

A pointer to an adaptive PRRT socket which owns the adaptability.
The thread mutex for protection read/write access of the socket statistics vs computing the parameters vs updating the socket setting.
A thread condition for notifying the work thread of the adaptation.
Starts/stops the running of the adaptation.
Stores the pointer to an adaptive PRRT socket.
Performs the adaptation. The real algorithm is implemented in its subclasses.

Figure 4.3.1: Class Diagram of Adaptability

4.3.2. Super Class (Adaptability)

The Adaptability is the base class for all concrete adaptation classes. It is responsible for providing the AdaptPRRTSock a generic control on all different adaptation objects e.g. starting or stopping an adaptation and setting an AdaptPRRTSock object in the adaptation, as well as for supporting the basic functions for subclasses of the adaptability e.g. applying an optimized parameter set on an adaptive PRRT socket, checking if the socket statistics of the PRRT changed and if an optimized parameter set is different to the current parameter set. Figure 4.3.1 shows the class diagram of the Adaptability. Therein applyOptParamSet, isStatsChanged and isOptParamSetChanged are defined as virtual methods, so that the subclasses may override them according to their adaptation mechanism, e.g. isStatsChanged may indicate a change only if the statistic is changed out of a specific threshold.

4.3.3. Lookup based Adaptation

A concrete lookup based adaptation on the PRRT is implemented in the class AdaptByLookup. This class implements the main thread method run to perform the adaptation in a loop. The workflow of the adaptation in AdaptByLookup is as follows:
4. Implementation : Integration of PRRT into NMM

1. updates statistics (RTT and PLR) of the PRRT socket.

2. smoothes the PLR and looks up an optimized parameter set from the precalculated table based on the smoothed PLR and RTT.

3. checks if the found parameter set is different to the current parameter set. The new parameter set is applied to the socket if the parameters have changed.

4. repeat the adaptation. If there is a change on the parameters, the adaptation will perform the repetition after the new parameters are applied. Otherwise the work thread for the adaptation will stay idle in a specific time duration and afterwards repeat the adaptation from step 1 again.

The corresponding code of the above workflow is shown in the following listing.

```cpp
void AdaptByLookup::run()
{
    // This is a main thread method for processing the adaptation in a loop.
    // Protect the state of read/write access of the statistics
    // and the update of the socket setting.
    MutexGuard mg( m_mutex );

    // Main Loop if the adaptation is still running.
    while ( m_is_running ) {
        // Update the statistic of the socket.
        m_sock->updateStats();

        // Check if the target statistics is changed and store it.
        isStatsChanged( 0, 0, 0, 0, 0, 0, 0, &m_grtt, 0, &m_gplr );

        // Look up the optimized parameter set. The smoothing is done in that method.
        const OptimizedParameterSet * opt = searchOptParamSet();

        // Check if the parameter set is changed.
        if ( isOptParamSetChanged( opt ) ) {
            // Apply the optimized parameter set on the PRRT socket.
            applyOptParamSet( opt );
        } else {
            // The working thread takes a break of 1 second, if the parameters don’t change.
            m_condition.relativeTimedWait(1,0);
        }
    }
}
```

Listing 4.1: run Method of AdaptByLookup

In addition to the operation of adapting, the AdaptByLookUp must set up the table of statistical optimized parameter sets before the adaptation. This task is performed by a class DelayCell, which can create the lookup table with the data structure described in section 3.4. Each instance of DelayCell represents an element containing a table of the number of receivers associated with the specific delay constraint. According to the characteristic of wireless LAN, we only create the table of the optimized parameters for the scenario of delay constraint with 200 ms, one receiver, two receivers and three receivers. The class diagram of the AdaptByLookup and the DelayCell is shown in figure 4.3.2.

For convenience we define three types of map below:
4.3. Adaptability

- typedef map<double, OptimizedParameterSet*> plr_map;
  A C++ STL containers map (PLR table) is used to store all optimized parameter sets each associated with the corresponding PLR.

- typedef map<int, plr_map*> rtt_map;
  It represents a map (RTT table) for storing all PLR tables each associated with a corresponding RTT.

- typedef map<int, rtt_map*> num_recv_map; This is a map (table) of the number of receivers, which consists of all RTT tables with the corresponding number of receivers.

The num_recv_map is constructed and initialized with empty container once the DelayCell is instantiated. If an element (i.e. an RTT table associated with a specific number of receivers on the network) isn’t found in the num_recv_map, that element will be created and all optimized parameter sets under that delay constraint and that specific number of receivers are stored in the table.

Figure 4.3.2: Class Diagram of AdaptByLookup

- A base class of the classes for the adaptation.
- Performs the adaptation of table lookup in a loop.
- Looks up the adapted parameter set from the pre-computed parameter sets.
- Smoothes the PLR and RTT respectively.
- Creates the table of optimized parameter sets for one receiver, two and three receivers with the data rate of 4Mbps or 12Mbps.
- Releases the resources allocated to the parameter sets.
- A map (c++ STL container) for different RTT tables associated with the number of receivers.
- The delay constraint of all optimized parameter sets contained in this object.
- Searches and returns the optimized parameter set with the given number of receivers, RTT and PLR.
- Creates the table of optimized parameter sets for one receiver, two and three receivers with the data rate of 4Mbps or 12Mbps.
- Releases the resources allocated to the parameter sets.

- typedef map<double, OptimizedParameterSet*> plr_map;
  A C++ STL containers map (PLR table) is used to store all optimized parameter sets each associated with the corresponding PLR.

- typedef map<int, plr_map*> rtt_map;
  It represents a map (RTT table) for storing all PLR tables each associated with a corresponding RTT.

- typedef map<int, rtt_map*> num_recv_map; This is a map (table) of the number of receivers, which consists of all RTT tables with the corresponding number of receivers.

The num_recv_map is constructed and initialized with empty container once the DelayCell is instantiated. If an element (i.e. an RTT table associated with a specific number of receivers on the network) isn’t found in the num_recv_map, that element will be created and all optimized parameter sets under that delay constraint and that specific number of receivers are stored in the table.
4.4. Transport Address

The addressing scheme in NMM serves two principal functions: the identification of network objects and the description of connection information. Our implementation involves two address classes as described below.

Internet Address Wrapper

The class \texttt{INetAddr} is a wrapper for an internet address in NMM, which is deployed in the socket wrapper (see section 4.2) for representing the local and remote addresses. The \texttt{INetAddr} class is convenient to use. The IP address and the port for an internet address are automatically converted from host byte order to network byte order. Additionally, it provides methods for checking the multicast address and converting into a human readable string for the address.

Transport Strategy Address

The transport strategy described in section 4.5 is the communication mechanism between objects in NMM. The transport strategy address represents the information about the connection using the transport strategy. The information covers location addressing, the description of the strategy and the means for connecting to the object, which are utilized to realize an NMM binding with its specifying strategy. NMM offers an abstract superclass \texttt{TSAddress} of the transport address. Based on different connection information, each transport strategy has its own transport address (a subclass of \texttt{TSAddress}), like in computer network the internet protocol having the internet protocol address. So does \texttt{PRRTStrategy} and the corresponding address is called \texttt{PRRTAddress}. Figure 4.4.1 illustrates the class diagram of the \texttt{PRRTAddress}.

The \texttt{PRRTAddress} is used to connect one \texttt{PRRTStrategy} to another \texttt{PRRTStrategy} i.e. \texttt{PRRTSenderStrategy} (see section 4.5.2) to \texttt{PRRTReceiverStrategy} (see section 4.5.3) or vice versa. It takes advantage of the internet address \texttt{INetAddr} to designate an IP address and a port. Deriving from the \texttt{ConnectorAddress} (a subclass of \texttt{TSAddress} too), which is used by the connection manager to perform a reliable connection setup for unreliable transport strategies, our \texttt{PRRTAddress} possesses the interface \texttt{IConnectionManager} of the connection manager in order to establish and configure the communication link between different address spaces, e.g. sending connection request and reply, sending reliable configuration events and exchanging addresses. This indirect connection setup is explained in section 4.5.2.

The following code, taken from \texttt{PRRTSenderStrategy}, shows an example how to set up a local \texttt{PRRTAddress} in the \texttt{PRRTStrategy}. First of all, it constructs and initializes a new \texttt{PRRTAddress} for the local address. Then the internet address in the local address is updated to the given address. For requesting an indirect connection setup\footnote{Different machines in the network may use different byte orderings internally for their multi-byte integers. This mismatch is solved by converting the native byte order to network byte order before the transmission and back again into host bytes order after the transmission.}, it requires

\footnote{A mechanism that provides reliable connection setup and reliable exchange of configuration events for non-reliable transport strategies. See section 4.5.2.}
4.5. Transport Strategy

The low level transport components in the communication framework of NMM are so called Transport Strategies, which provide the actual implementations of the transport mechanisms to communicate NMM messages through relevant sockets. For instance, PRRTStrategy is a new such transport component that implements the adaptive PRRT

---

Figure 4.4.1.: Class Diagram of PRRTAddress

---

Listing 4.2: setLocalAddress Method of PRRTSenderStrategy
transport mechanism in NMM using AdaptPRRTSock, which is described in section 4.2. NMM separates between leaf transport strategies and top-level transport strategies through respectively representing an endpoint of a sub-binding and a binding. TransportStrategy is the base class of the transport strategies, which realizes a transport mechanism as one endpoint of a sub-binding or a binding. PRRTStrategy is a leaf transport strategy and is implemented as two subclasses of TransportStrategy, PRRTSenderStrategy (section 4.5.2) and PRRTReceiverStrategy (section 4.5.3) as a pair of transport strategies9, which respectively implement the communication mechanisms via the PRRT protocol on the sender side and the receiver side.

4.5.1. Registration

A transport strategy can be used in NMM by registering it at the transport strategy factory when a process starts and deregistering it when a process ends. The transport strategy factory is a singleton named TSFactory which creates transport strategies. The transport strategy can be registered by a template registration object of transport strategy (TTSFactoryRegistrator), which is a wrapper for automatic construction, destruction, registration and deregistration of the transport strategy factory object (TSSFactoryObject). A code snippet from PRRTReceiverStrategy shows how to register PRRTStrategy of the receiver side as follows:

```c++
#include "nmm/comm/TTSFactoryRegistrar.hpp"
#include "nmm/comm/TTSFactoryObject.hpp"

// Register class PRRTReceiverStrategy at the TSFactory.
TTSFactoryRegistrar< TTSFactoryObject<PRRTReceiverStrategy> >
   m_prrt_recv_strategy_prototype("PRRTStrategy",
       new ConnectorAcceptor("PRRTStrategy"),
       TSDescription::ITS_ONE_WAY_BUFFER_SENDER,
       TSDescription::TS_RELIABLE |
       TSDescription::TS_ONE_TO_ONE |
       TSDescription::TS_DOWNSTREAM_CONNECT |
       TSDescription::TS_UPSTREAM_CONNECT,
       TSAddress::NEW_PRIVATE,
       TSAddress::NEW_PRIVATE);
```

Listing 4.3: Registration of Transport Strategy at TSFactory

PRRTReceiverStrategy is registered with its supported characteristics of the transport strategy (TSDescription) at the TSFactory by the construction of TTSFactoryRegistrator. The constructor of TTSFactoryRegistrator provides the following parameters:10, some of which specify the supported characteristics of PRRTReceiverStrategy like transport strategy interfaces and capabilities:

- The type name of the strategy.
- A ConnectorAcceptor object used by ConnectionManager to achieve indirect connection setup, if the strategy cannot offer reliable connection setup.

---

9 Most NMM low-level transport strategies are implemented into a pair of transport strategies, each of which realizes the transport mechanism on the sender side or the receiver side.

10 More available values and their meaning supported by the parameters can be found in Table 8, 12, 13 and 14 of [Mot10b].
4.5. Transport Strategy

- Transport role interfaces, which are determined by a bit mask of transport role interface flags the strategy supports. A sender side strategy should implement the ITS_ONE_WAY_BUFFER_RECEIVER interface and a receiver side strategy ITS_ONE_WAY_BUFFER_SENDER, as mentioned in chapter 2.

- Transport capabilities. The capabilities of a transport strategy are also represented by a bit mask of transport capability flags. As shown in the previous code PRRTStrategy offers one-to-one and one-to-many (e.g. broadcast and multicast) communication and can establish instream upstream and downstream connections.\(^\text{11}\)

- Default sharing mode if the strategy is used for out-of-band connections. The sharing mode of PRRTStrategy is defined as NEW_PRIVATE which means that it creates new private binding, since PRRTStrategy doesn’t support shared bindings at the moment.

- Default sharing mode if the strategy is used for instream connections. NEW_PRIVATE is defined for instream connections of PRRTStrategy same as that for out-of-band connections.

Within the foregoing specified information, TSFactory can find the corresponding transport strategy and then create an instance of that for the top-level transport strategies (CompositeStrategy\(^\text{12}\)).

4.5.2. PRRTSenderStrategy

PRRTStrategy on the sender side in NMM is implemented in a transport strategy class called PRRTSenderStrategy, which receives NMM Buffers from other NMM components in the same address space by taking the transport role of the one-way buffer receiver (ITSOneWayBufferReceiver) and sends them to the network via the PRRT protocol using AdaptPRRTSock. The class diagram of PRRTSenderStrategy is shown in figure 4.5.1 where most methods of the class are listed and some e.g. parameters and option methods are skipped in view of the space.

PRRTSenderStrategy derives from five classes to reach its task:

- TransportStrategy realizes the fundamental functionality of transport mechanisms.

- ITSOneWayBufferReceiver provides message passing interface as one-way buffer receiver in the local address space.

- TSAAddressOwner gets the local address of the strategy during the indirect connection setup.

- IPRRTControlImpl supports control interfaces for setting the parameters and options of the PRRT protocol.

\(^{11}\) PRRTStrategy is not a reliable strategy. The flag TS_RELIABLE is just to make it work in the parallel binding. For further details please see section 4.6.1.

\(^{12}\) CompositeStrategy is the base class of top-level transport strategies for realizing the bindings and responsible for managing the leaf transport strategies.
ThreadedObject offers a dedicated thread for the strategy in sending data.

The main implementation of PRRTSenderStrategy can be generally covered by two aspects: connection setup and sending data.

Figure 4.5.1.: Class Diagram of PRRTSenderStrategy
4.5. Transport Strategy

Connection Setup

Before a communication between objects, NMM requires transport strategies to establish and configure the connection for the communication. This process is done in a handshaking way by two methods requestConnection and acceptConnection, which are pure virtual methods of TransportStrategy and are implemented in each leaf transport strategy. In PRRTSenderStrategy, the requestConnection method requests a connection to a partner transport strategy i.e. PRRTRecieverStrategy, whereas acceptConnection accepts the connection from a partner transport strategy.

An indirect connection setup takes places in the instream channel mode. The figure 4.5.2 shows a sequence diagram of an indirect connection setup for PRRTStrategy, which demonstrates how this handshaking connection works. Two classes of transport addresses must be supported by the strategy if the strategy requires the indirect connection setup. One is the ConnectorAddress, with which the requestConnection method can initiate an indirect connection by the connection manager. The other is the subclass of ConnectorAddress named PRRTAddress (see section 4.4), which specifies the connection information for the strategy. After the initiation of a connection from requestConnection, ConnectionManager takes over the connection and then invokes the acceptConnection with a PRRTAddress on the other side of the connection. Note that in acceptConnection the local address of the requesting strategy is passed to the accepting strategy with a PRRTAddress and the local address of the accepting side is also passed to the requesting side with another PRRTAddress after the method ends. These two addresses are the arguments (denoted with the local and remote address) of the connect method in the connection manager.

An extracted implementation of the requestConnection method looks as follows:

```cpp
unsigned int PRRTSenderStrategy::requestConnection
(const TSAddress& address,
 const CEvent& local_config, const CEvent& remote_config)
{
    // Determine the type of the address.
    const PRRTAddress* prrt_address =
        dynamic_cast < const PRRTAddress* >(& address);
    const ConnectorAddress* connector_address =
        dynamic_cast < const ConnectorAddress* >(& address);

    // Create a local PRRT socket.
    if (prrt_address) {
        // Create the socket and bind it to the explicit address and port.
        createLocalSocket ( &( prrt_address -> getINetAddr ()) );
    } else {
        // Create the socket
        // and bind it to the local host address and a random port.
        createLocalSocket(0);
    }

    // Apply the local configuration.
    applyConfiguration(local_config);

    // Set up the local address with the bound address in PRRT socket,
    setLocalAddress( getAdaptPRRTSocket().getLocalAddress() );

    // Set up the remote address.
```

An indirect connection setup takes places in the instream channel mode. The figure 4.5.2 shows a sequence diagram of an indirect connection setup for PRRTStrategy, which demonstrates how this handshaking connection works. Two classes of transport addresses must be supported by the strategy if the strategy requires the indirect connection setup. One is the ConnectorAddress, with which the requestConnection method can initiate an indirect connection by the connection manager. The other is the subclass of ConnectorAddress named PRRTAddress (see section 4.4), which specifies the connection information for the strategy. After the initiation of a connection from requestConnection, ConnectionManager takes over the connection and then invokes the acceptConnection with a PRRTAddress on the other side of the connection. Note that in acceptConnection the local address of the requesting strategy is passed to the accepting strategy with a PRRTAddress and the local address of the accepting side is also passed to the requesting side with another PRRTAddress after the method ends. These two addresses are the arguments (denoted with the local and remote address) of the connect method in the connection manager.

An extracted implementation of the requestConnection method looks as follows:

```cpp
unsigned int PRRTSenderStrategy::requestConnection
(const TSAddress& address,
 const CEvent& local_config, const CEvent& remote_config)
{
    // Determine the type of the address.
    const PRRTAddress* prrt_address =
        dynamic_cast < const PRRTAddress* >(& address);
    const ConnectorAddress* connector_address =
        dynamic_cast < const ConnectorAddress* >(& address);

    // Create a local PRRT socket.
    if (prrt_address) {
        // Create the socket and bind it to the explicit address and port.
        createLocalSocket ( &( prrt_address -> getINetAddr ()) );
    } else {
        // Create the socket
        // and bind it to the local host address and a random port.
        createLocalSocket(0);
    }

    // Apply the local configuration.
    applyConfiguration(local_config);

    // Set up the local address with the bound address in PRRT socket,
    setLocalAddress( getAdaptPRRTSocket().getLocalAddress() );

    // Set up the remote address.
```
if ( prrt_address ) {
    // We have an explicit PRRTAddress. Do nothing, it’s Client/Server mode.
    // The remote address should be explicitly set by PRRTnetSinkNode.
} else {
    // We have a ConnectorAddress.
    // Set up an indirect connection by ConnectionManager.
    // Store the IConnectionManager interface.
    m_icm = dynamic_cast<IConnectionManager*>(
            connector_address->getConnectionManager()->copy());

    // Request a connection between this strategy and a partner strategy.
    SConnectionManager::getInstance().connect(m_local_address,
        remote_addr, remote_config, m_icm);

    // The exchange of local and remote address is done. Store the remote address.
    const PRRTAddress* prrt_addr = dynamic_cast<PRRTAddress*>(remote_addr);
    m_remote_address = prrt_addr->copy();

    // Store the NMM reference of the partner strategy.
    m_remote_ts_ref = m_remote_address->getTSReference();

    // Set the peer address in the PRRT socket.
    getAdaptPRRTSocket().setPeerAddress( m_remote_address->getINetAddr() );
} // The connection index is always returned zero for one-to-one
// instream connection setup.
return 0;

Listing 4.4: requestConnection Method of PRRTSenderStrategy

The initiator of a connection firstly creates a local PRRT socket for the transmission.
4.5. Transport Strategy

With the bound address and port on the created socket it sets up its local address. Then the strategy initiates a connection request to its partner strategy and stores the returned information of the connection request. Finally we tell the PRRT socket about the remote address.

The following code manifests the acceptConnection method for the above requestConnection method.

```cpp
unsigned int PRRTSenderStrategy::acceptConnection
    (const TSAddress& address, const CEvent& local_config)
{
    // The address must be a PRRTAddress, otherwise throw an exception.
    const PRRTAddress* prrt_address =
        dynamic_cast<const PRRTAddress*>(&address);
    // Create a local PRRT socket
    // and bind it to the local host address and a random port.
    createLocalSocket(0);
    // Apply the local configuration.
    applyConfiguration(local_config);
    // Set up the local address.
    setLocalAddress( getAdaptPRRTSocket().getLocalAddress() );
    // Store the remote address.
    m_remote_address = prrt_address->copy();
    // Set the remote IConnectionManager interface and reference.
    if (m_icm = dynamic_cast<IConnectionManager*>(m_remote_address->getConnectionManager())->copy()) {
        m_remote_ts_ref = m_remote_address->getTSReference();
    }
    // Set the peer address in the PRRT socket.
    getAdaptPRRTSocket().setPeerAddress( m_remote_address->getINetAddr() );
    // The connection index is always returned zero for one-to-one
    // instream connection setup.
    return 0;
}
```

Listing 4.5: acceptConnection Method of PRRTSenderStrategy

As seen in the preceding code extraction, what two methods really do are to set up the local PRRT socket and the local/remote PRRTAddress (see section 4.4) through explicitly setting the configuration or interacting with the partner strategy under a connection manager. It’s important to note that to set up an indirect connection between transport strategies by the connection manager the transport strategy requires the implementation of the getLocalAddress method derived from TSAddressOwner and the registration at the ConnectionManager with an additional ConnectorAcceptor object as shown in code listing 4.3.

Sending Data

To realize the sending part in PRRTSenderStrategy, we should be aware of how to get the sent data and how to send out the data. These two questions can be an-

---

13 Section 4.5.5 talks about the detail regarding how to set up the local PRRT socket.
answered by the implementation of the three methods: `receiveOneWayBuffer`, `run` and `sendQueuedBuffer`. As mentioned before the strategy on sender side should inherit the transport role interface (ITSOneWayBufferReceiver) to receive data from other components in the local address space. Then the strategy has to implement the `receiveOneWayBuffer` method derived from ITSOneWayBufferReceiver to accomplish this task. This method is called to pass messages to the strategy by the components that need sending those messages via that strategy.

```c++
void PRRTSenderStrategy::run()
{
    // This is a main thread method for handling messages in a loop.
    MutexGuard buffer_guard(m_buffer_mutex);
    while (m_is_sender_running) {
        // Wait until there is a message to send.
        while (m_is_sender_running && !m_send_buffer) {
            // Notify the condition to wake up the buffer sending thread.
            m_buffer_condition.notifyAll();
        }
    }
}
```

Listing 4.7: `sendQueuedBuffer` Method of PRRTSenderStrategy

There is a simple synchronization mechanism in the strategy i.e. it always keeps only a certain number of buffers (one buffer here) in the queue to be handled in each sending. The aim is to avoid excessive buffers queued in the strategy and to be able to comply with some protocol mechanism (e.g. flow control). This is done by returning `BLOCKED` in the `receiveOneWayBuffer` method, as noticed in the previous code, and signaling the sender component to pass a message again with the `FREE_QUEUE_SLOT` signal after processing the buffer. The latter one is performed in the `run` method (see listing 4.7).

But then we have to make sure that the operation of buffer sending is non-blocking, because blocking in buffer sending results in blocking in the `receiveOneWayBuffer` method or no notification about the `FREE_QUEUE_SLOT` signal, one of which causes that the sender component is also blocked which passes buffers to the strategy. Hence PRRTSenderStrategy owns a separate thread by itself through the inheritance of ThreadObject and implements the inherited method `run` to handle incoming messages in a loop. The below listing demonstrates the sample implementation of the `run` method.

```c++
void PRRTSenderStrategy::run()
{
    // This method is called by OutputJack or PRRTNetSinkNode.
    MutexGuard buffer_guard(m_buffer_mutex);
    // Enqueue the to send buffer.
    m_send_buffer = buffer;
    // Notify the condition to wake up the buffer sending thread.
    m_buffer_condition.notifyAll();
    // Return BLOCKED for synchronization.
    return BLOCKED;
}
```

Listing 4.6: `receiveOneWayBuffer` Method of PRRTSenderStrategy

Note that the size of the queue can select various units in line with the capability of the strategy, for example, number of buffers, bytes and time units.
m_buffer_condition.wait();
}

if (!m_is_sender_running) {
    // Exit message sending thread.
    return;
}

// Dequeue the buffer to send.
const Buffer* tmp_buffer = m_send_buffer;
m_send_buffer = 0;

// Unlock the buffer mutex while not using the state of sending buffer.
// Allow other threads to enqueue new buffers while sending the buffer.
// This greatly improves the performance.
buffer_guard.unlock();

// Notify the parent about the FREE_QUEUE_SLOT signal.
// Yield a different signal based on the direction of receiving buffer.
if (tmp_buffer->getDirection() == Message::DOWNSTREAM) {
    getSignalReceiver()->receiveSignal(FREE_DOWNSTR_QUEUE_SLOT);
} else if (tmp_buffer->getDirection() == Message::UPSTREAM) {
    getSignalReceiver()->receiveSignal(FREE_UPSTR_QUEUE_SLOT);
}

// Send the dequeued buffer.
sendQueuedBuffer(tmp_buffer);

// Lock the buffer mutex again.
buffer_guard.lock();
}

Listing 4.7: run Method of PRRTSenderStrategy

PRRTSenderStrategy enforces mutual exclusion to protect the state of the sending buffer while enqueuing a buffer in the receiveOneWayBuffer method and dequeuing one in the run method. This protection is undertook by the MutexGuard\(^{15}\) object with a ThreadMutex\(^{16}\) variable in those two methods. The receiveOneWayBuffer method instantiates a MutexGuard with a ThreadMutex variable to lock the buffer state while storing the pointer of a sent buffer (enqueuing) and unlock the state when the method exits. The run method locks the buffer state while dealing with the buffer (checking if a sent buffer is available and dequeuing a buffer). It should be noticed that we release the lock while sending a dequeued buffer and acquire the lock again after a buffer is sent in the run method. This enables new buffers to be enqueued while the dequeued buffers being sent. Otherwise it will heavily reduce the efficiency. Moreover, it would happen that no buffer is available to send. In this case the thread of sending data has to wait. To get around the busy waiting PRRTSenderStrategy takes advantage of the condition variable (ThreadCondition\(^{17}\)). That releases the mutex lock and lets the buffer sending thread go to sleep during the waiting period as well as notifies the thread when a new sent buffer arrives (Notifying is executed on the receiveOneWayBuffer method shown in listing 4.6). A sequence diagram in figure 4.6.3 pictures this synchronization workflow

\(^{15}\) MutexGuard is a class of NMM which guards a mutex variable (ThreadMutex) in scope, namely a locked mutex is unlocked if an instance of MutexGuard is destroyed.

\(^{16}\) ThreadMutex is a wrapper in NMM for the pthread_mutex variable of POSIX Pthread libraries on Linux.

\(^{17}\) ThreadCondition is also a wrapper for the pthread_cond_t variables. It is used to notify the state information of shared resources associated with ThreadMutex.
4. Implementation: Integration of PRRT into NMM

with the PRRTNetSinkNode, which will be explained in section 4.6.2.

PRRTSenderStrategy can handle buffers which are received from the upstream or downstream direction. Upon the direction the strategy, as a signal sender, lets the partner component get a corresponding **FREE_QUEUE_SLOT** signal, so that the component can pass buffers to the strategy. The partner component is known as a signal receiver in the strategy by calling the **getSignalReceiver** method and is set before during the setup of the strategy.

The genuine action of sending data in the strategy is placed in the **sendQueuedBuffer** method, which invokes the **send** operation of **AdaptPRRTSock** to send out the buffer data to the destination through the PRRT socket. PRRT supports sending packets within a given payload size depending on the practical scenarios. A reasonable large payload size brings greater efficiency while taking the header overhead of the protocol into account and can lessen the packet loss due to space overflow of the kernel UDP receive buffer. We chose 1316 bytes as the payload size of the PRRT packet in our experiment, which is the size of multiple MPEG-TS packets (7*188) and the size of the used data in the sent NMM buffer as well. Additionally considering fragmentation of IP packets in the transmission we limit the payload size to 1500-8-12 bytes (8 bytes for the UDP header and 12 bytes for the PRRT header), as some old devices might suffer fragmentation if a UDP packet has a larger size than 1500 bytes.

```c
void PRRTSenderStrategy::sendQueuedBuffer(const Buffer* buffer)
{
    // This method is called in run to send the dequeued buffers.
    // Allocate an array for repacking the payload with the buffer counter
    // in the case of instream channel mode.
    static char payload[1500];
    // BEGIN > Send the buffer in one packet.
    // Determine the application mode.
    if (m_icm == 0) { // Client / Server mode, since we don’t use ConnectionManager.
        getAdaptPRRTSocket().send(buffer->getData(), buffer->getUsed());
    }
    else { // Instream Channel mode, since using ConnectionManager to set up
        // an indirect connection in instream channel mode.
        // Send the buffer counter for parallel binding together with the
        // data in one PRRT packet.
        memcpy(payload, buffer->getData(), buffer->getUsed());
        int64_t * seq_nr = (int64_t *)(payload + buffer->getUsed());
        *seq_nr = buffer->getTimestamp().stream_counter;
        getAdaptPRRTSocket().send(payload, buffer->getUsed() + 8);
    }
    // END < Send the buffer in one packet.
    // Release the buffer.
    buffer->release();
}
```

18 When we set the maximum socket receive buffer size, the kernel doubles the value we set to allow space for bookkeeping overhead. A larger packet size will reduces this overhead.

19 The 1316 bytes are suggested by the DVB-IP specification.

20 In computer networks, the Maximum Transmission Unit (MTU) determines the largest size of the protocol unit. If the data size is bigger than the MTU the sender devices must fragment the data and the receiver devices must reassemble the data again. It naturally has to incur certain costs.
4.5. Transport Strategy

Listing 4.8: sendQueuedBuffer Method of PRRTSenderStrategy

As said before, PRRTStrategy in the instream channel mode requires the connection manager to set up an indirect connection between the endpoints of the communication but it doesn’t need that in the C/S mode. This implies that the ConnectionManager interface of the partner strategy isn’t instantiated in the C/S mode. Consequently we decide which mode is running by checking if the instance of connection manager has been created, like m_icm == 0 in the run method, where m_icm is a pointer to the instance of ConnectionManager interface. Different to the C/S mode, the instream channel mode requires us to pack the sequence number (8 bytes) of NMM buffers together with the actual data into the payload before a packet being sent. This is necessary for the message ordering in Parallel Binding (see section 4.6.1).

At the end of this section we introduce three methods of PRRTSenderStrategy, which allow the strategy to configure the PRRT socket with respect to the application scenarios.

- **setTargetedDelay** This method is intended to set the time limit required by the application which expresses the maximum transmission delay of a packet. The packets exceeding this limit on the transmission will be dropped by the PRRT socket.

- **setTargetedPLR** This method determines the acceptable packet loss rate to the application.

- **setAdaptationMode** The mode of the adaptation used on the PRRT is decided by this method with one of the following string flags:
  - "auto" – the default adaptation is selected. At the moment the default mode is "lookup".
  - "lookup" – the adaptation is done by looking up optimized parameters from a table of precomputed parameter sets, depending on the current network conditions.
  - "manual" – the adaptation is disabled. The default parameters or the given parameters can be applied in this case.

4.5.3. PRRTReceiverStrategy

As a pair of the realization on PRRTStrategy together with PRRTSenderStrategy, the strategy on the receiving side has the class name PRRTReceiverStrategy. In contrast to the sender strategy, the strategy is implemented to perform the functions of receiving data instead of sending data. This makes the workflow of the receiver strategy reverse in a mirror, i.e. PRRTReceiverStrategy reads out the data from the PRRT socket and then passes the data to the other NMM component in the same local address space. Again AdaptPRRTSock is utilized to accomplish the receiving of data from the network; while the strategy plays a role of the buffer producer to NMM with the transport role interface (ITSONewayBufferSender). The class diagram in figure 4.5.3 gives us an overview of the
4. Implementation : Integration of PRRT into NMM

PRRTReceiverStrategy

- scheduleDestruction()
- getStrategyType() : string

Responsibilities
- requests and accepts an indirect connection setup to a partner strategy.
- sets up the adaptive PRRT socket on the receiver side.
- handles the receiving data i.e. receives data from the PRRT socket and forwards data to other NMM component.

Figure 4.5.3.: Class Diagram of PRRTReceiverStrategy

structure and related classes involved in PRRTReceiverStrategy.

In comparison with PRRTSenderStrategy, the parent classes of PRRTReceiverStrategy have the following differences:

- ITSOneWayBufferSender. The received data in the strategy must be delivered to the actual consumer entity. So PRRTReceiverStrategy works as a buffer sender on passing buffers.

- no inheritance of the PRRT control interface (IPRRTControl). Because the controls of a PRRT socket e.g. parameters setting and the adaptation are centralized in the sender, the receiver strategy isn’t supposed to implement it.

- ThreadedNetHandler vs ThreadedObject. A sender strategy typically runs in a separate thread to ensure non-blocking on sending data, as explained before. How-
ever, in order to concurrently handle network requests from multiple sockets or multiple events on the same socket, NMM provides a special handler of network events (ThreadedNetHandler) for transport strategies in cooperation with an event listener named Network Reactor (NetReactor). More details of these two classes are described later.

The implementation of PRRTReceiverStrategy can also be separated into two parts like PRRTSenderStrategy: connection setup and receiving data. In the part of connection setup, it uses the similar manner to establish and configure the connection between two leaf strategies. So we won’t repeat it again. But the part of receiving data is much different to sending data of PRRTSenderStrategy and is the most important part in PRRTReceiverStrategy. Furthermore that part becomes much more complicated since ThreadedNetHandler is employed instead of ThreadedObject. Therefore we’re going to explain it in a single section called Network Event Handling.

We firstly see how the receiver strategy forwards a buffer of received multimedia data to other components before digging into Network Event Handling. Deriving from one-way buffer sender (transport role interface), PRRTReceiverStrategy owns two methods for setting and getting one-way buffer receiver, which is the partner object of the strategy on buffer passing in the same address space. Thus the strategy can make the receiver component to accept the buffer through running

\[
getOneWayBufferReceiver() \rightarrow \text{receiveOneWayBuffer(buffer)};
\]

if a buffer is available to be delivered. The partner component of the strategy can be determined by the component itself as the owner of the strategy with calling

\[
setOneWayBufferReceiver(this);
\]

Like the PRRTSenderStrategy, the receiver strategy should also go along with the partner component under a certain synchronization mechanism to save the buffer queue from overflow. But unlike the sender strategy, where the strategy restrains the sender component from delivering buffers too fast in order to hold a reasonable number of buffers at a time, PRRTReceiverStrategy is required not to forward buffers rashingly. Otherwise extreme excess buffers will be queued in the succeeding components. So the same interaction between the strategy and its partner is applied once more. The receiver strategy continues forwarding the buffers only when the partner returns a result of not BLOCKED in the receiveOneWayBuffer method. The BLOCKED result will pause the receiving operation of the strategy. (See Listing 4.10 of the receiveData method in section 4.5.4.) The strategy is woken up to work again if the partner dispatches a FREE_QUEUE_SLOT signal to it through the receiveSignal method. The receiveSignal method is an interface method of the receiver strategy which derives from ISignalReceiver. The receiver strategy must implement this method to receive and process signals. The sample code of the receiveSignal method is listed below.

```cpp
void PRRTReceiverStrategy::receiveSignal(Signal signal)
{
    if (signal == FREE_QUEUE_SLOT) {
        // Wake up the receiving operation, since the partner is no longer blocked.
    }
```

21 It provides an interface for receiving and handling NMM signals.
4. Implementation : Integration of PRRT into NMM

Listing 4.9: receiveSignal Method of PRRTReceiverStrategy

4.5.4. Network Event Handling

Generally it’s impossible to transmit the data stream without any discontinuity. Because of the processing time and scheduling on a computer system as well as the unpredictable delay on the network there must be some time gap between any two consequent data packets, though it’s very tiny. Hence it’s inefficient that the transport strategy receives the data from the network socket using a busy waiting thread. An ideal case is that the strategy reads out the data only if the data is available on the socket. This can be typically done by employing a thread to wait for some network events (e.g. there is data to read.), like the Linux poll() function, and then handling the event if one occurs. Within a refined scheduling this approach can be extended to efficiently handle multiple requests on multiple sockets.

The Leader-Followers Event Handling [Mot10b] in NMM is just this kind of framework that handles network events from multiple sockets in an efficient, concurrent and scalable way. Our PRRTReceiverStrategy employs this framework. It consists of the Network Reactor and the Threaded Network Event Handler.

We firstly explain the network reactor (NetReactor). The NetReactor is an network event listener and notifier with an own thread, which waits for incoming network events and triggers the corresponding network event handler to handle the events using the Linux poll-function. To poll the network events on a socket we must register that socket at the NetReactor. The following line gives an example how PRRTReceiverStrategy registers the PRRT socket at the NetReactor:

```cpp
SNetReactor::getInstance().registerSocket(getAdaptPRRTSock().getSocket(), this);
```

The unregistration looks similar to the registration as follows:

```cpp
SNetReactor::getInstance().unregisterSocket(getAdaptPRRTSock().getSocket());
```

The registration is performed during creating a PRRT socket, see section 4.5.5. The un-registration is invoked when the strategy is destructed. It should be noted that the socket must provide a file descriptor of the network socket for the NetReactor as the argument.\(^{22}\)

Once the socket is registered at the NetReactor, the events on the socket will immediately be polled and the event requests to be handled if one is detected. However, in order to handle the events notified by the network reactor, the strategy must derive from DispatchedNetHandler or ThreadedNetHandler and implement the handler functions for arose events in need. Those two network handler can dispatch the events from NetReactor and map them to the appropriate operation. For example, the event

\(^{22}\) The NetReactor employs the Linux function poll to wait for network events on the socket. We found that that poll doesn’t work properly for proc file descriptors.
EVENT_RECEIVE_DATA matches the handler method `receiveData`. ThreadedNetHandler is actually a subclass of `DispatchedNetHandler` which exploits the Leader-Followers pattern [POSA2] together with the `NetReactor` to avoid context switches between detecting and handling events. Therefore, as already seen in the class diagram of figure 4.5.3, `PRRTReceiverStrategy` inherits from `ThreadedNetHandler`.

Now let us see how the Leader-Followers pattern works in `PRRTReceiverStrategy`. The leader-followers event handling allows one thread at a time - the leader - to listen to new networks events on a set of sockets. Meanwhile, other threads - the followers - can handle the events. There are three points that should be noted. Firstly, if an event for one socket is being handled, the leader thread won’t wait for new events on that socket, which will be temporarily deactivated at the `NetReactor`. Secondly, the roles of the leader and follower can be exchanged. It means that a leader thread becomes a follower thread while it is processing the event detected by itself and a follower thread can be promoted to the leader again after it finishes the processing of the event. By the NMM thread pool (`ThreadPool`), the idle threads can be managed and reused for the dynamic leader-followers pattern. Finally, before a leader thread becomes a follower as the event handler, it must request a new leader thread to poll the events on sockets. So in this model the followers (the former leaders) and the new leader can run concurrently and no context switch is overspent between the detection and the processing of an event.

The leader-followers pattern for `PRRTReceiverStrategy` is depicted in the sequence diagram of figure 4.5.4. At the beginning, a thread for the `NetReactor` is idly waiting for an event to occur on a number of sockets. That thread is a leader in that period and there is no followers. At some moment an event on the PRRT socket occurs. The `NetReactor` notifies that event to the `ThreadedNetHandler` by calling the `receiveEvent` method of `ThreadedNetHandler`. Then the `ThreadedNetHandler` is in the process of handling the event and becomes a follower thread. At first the `ThreadedNetHandler` requests a new leader thread to take over the event listening on other sockets (except the current socket, since that PRRT socket is temporarily deactivated before handling the event on it.). Now the event is dispatched to the `PRRTReceiverStrategy` and concurrently the new leader thread is checking for new events on other sockets. Based on the type of the event `PRRTReceiverStrategy` invokes the corresponding method. In the diagram an event for reading data from the socket is caught, the handler method `receiveData` is executed to read out data and process it. If at the same time an event is detected by the new leader, the same way as before is repeated. After the receiving of data finishes, the follower becomes idle again and is retrieved by the `ThreadPool` for later reuse.

Furthermore, NMM still has one more variant in the leader-followers pattern to prevent from frequent context switching if some events of the same socket occurs frequently in a short time. There are three return values (`VOID`, `RECHECK` and `REMOVE`) in each event handling function of the `ThreadedNetHandler`. Each of them prompts the different behaviors of the `NetReactor` and the current thread which have just dealt with the socket, after the thread finishes processing an event from that socket.

- `VOID` means that nothing should be done with that socket and the thread becomes
4. Implementation: Integration of PRRT into NMM

NetReactor ThreadPool «Thread 2»

Figure 4.5.4.: Leader-Followers Event Handling for PRRTReceiverStrategy

idle.

- **RECHECK** allows the thread to wait for new events on that socket until a timeout occurs.

- **REMOVE** leads to unregistrating that socket from the NetReactor. The thread awaits tasks in the ThreadPool.

At this point, the strategy has a chance that a follower thread can go on listening for events on the same socket, from which an event was processed, and handle a new event without context switch if one is detected before a short timeout. Of course this will cost a dedicated thread, but the performance benefits if network events occur frequently.

The following code section shows an example of the handler method **receiveData** from PRRTReceiverStrategy.

```c
NetHandler::NetResult PRRTReceiverStrategy::receiveData(SocketHandle /*socket*/)
{
    // This method is called if ThreadedNetHandler dispatches an EVENT_RECEIVE_DATA.
    // Assume the max payload size 1500 due to packet fragmentation.
    uint32_t size = 1500;
    // Allocate new NMM Buffer
    Buffer* buffer = getBufferManager()->getNewBuffer(size);
    // BEGIN> Read at most @c size bytes from socket and update used size of buffer.
    // Read out the data from the PRRT socket.
    uint32_t received = getAdaptPRRTSocket().recv(buffer->getData(), size);
```
4.5. Transport Strategy

Listing 4.10: receiveData Method of PRRTReceiverStrategy

The handler method receiveData is executed if an event of receiving data is dispatched to the PRRTReceiverStrategy. What is to be done in this method is firstly to allocate a new NMM buffer for storing the received data. The interface for an external buffer manager IExternalBufferManager enables the strategy to accomplish the job. In our case the strategy uses the interface method getNewBuffer to request a new buffer from the buffer manager which can hold multimedia data at least 1500 bytes. To manipulate the data in a buffer, the getData method of the Buffer returns a pointer to the data field (a simple byte array), which we can write and access. Once the received data is saved

24 This interface allows the node to set and request a buffer manager as well as request a buffer from that buffer manager. The buffer manager provides buffer management e.g. allocation, release and reuse of a buffer. See [Mot10b] [Loh05] for more details.
into the buffer, we can set the actually used bytes of the buffer according to the size of multimedia data. As the sender packages multimedia data together with the sequence number (the last 8 bytes) in the payload of the PRRT packet in the case of instream communication, the receiver has to extract the sequence number and update the corresponding buffer information e.g. the actually used size and the time stamp.

Finally, we need to deliver the produced buffer containing multimedia data to the receiver component which further processes the buffer. This delivery is already mentioned in prior section 4.5.3. However, a remark should be made here. The handler method will affect the succeeding actions on the current thread and the NetReactor in accordance with its return values. PRRTreceiverStrategy returns RECHECK in order to detect more network events on the PRRT socket, if it gets a non-blocking signal from the receiver component, because the component is still able to accept more buffers at that time. In situations where the buffer queue of the receiver component is full i.e. the component cannot temporarily receive the buffer and BLOCKED is returned by its receiveOneWayBuffer method, the strategy cannot listen to network events until the component notifies a signal FREE_QUEUE_SLOT to it. After waiting for the free queue signal the receiver might not run any more. Then the strategy tells the current thread to be idle through returning VOID.

4.5.5. Setting up PRRT Sockets

To achieve the message transmission via our adaptive PRRT protocol, we must set up the adaptive PRRT sockets in our PRRTStrategy. This task is undertaken by the method createLocalSocket (see listing 4.11), which can be used in the strategy to set up a local PRRT socket. Thereafter the receiver strategy can receive the PRRT packets from its created socket while the sender strategy can send out the NMM buffers through the PRRT socket. Thanks to the socket wrapper AdaptPRRTSock, there is not much work for setting up a local PRRT socket. Only calling the binding method of socket wrapper (see section 4.2.1) already makes a PRRT socket with the bound address and port ready for the transmission. However, we should consider two cases depending on the application scenarios and fit the setup of the socket to them.

- **Explicit Address** An explicit address must be given to set up the sockets if the user starts a client/server program (see section 4.6.2). The program will terminate in case of the failure on binding the socket with the given IP address and port, e.g. a failure due to an engaged port by another program. For convenience we allow the socket to be bound to a “random” free port by setting the given port to 0.

- **Implicit Address** If the strategy is used in the instream channel mode (see section 4.6.1), no address (a null pointer) is passed into the method, since the setting of the strategy is automatically controlled by NMM. In this case the PRRT socket is bound to the local host address and a “random” free port.

25 The address must be an address of the local host, otherwise binding fails too.
26 We iterate all port numbers from 30000 to 40000 and select the first free one among those ports. The bind method of the PRRT socket doesn’t support binding to a random free port at the moment.
27 This case has two limitations: the local host cannot locate in the multiple subnetworks, because the
Furthermore, the receiver side of \textit{PRRTStrategy} requires two more steps for the setup of the socket, compared to the sender side. One is concerned with UDP receive buffer size, which affects the quality of our applications. An improperly large buffer size will deteriorate the latency (dependent on the data rate) in an interactive application like TV channel switching, since the application has to read through excessively obsolete data. However, a too small buffer size will also bring us problem – packet loss due to UDP buffer space overflow, because the application should be able to bear the normal variance in CPU scheduling latency. From our experiment we find that we can get around packet loss if UDP buffer sizing $200\times1316$ bytes. Another setting is with respect to network event handling described in section 4.5.4. We need to register the file descriptor of the socket at the network reactor in order to activate the event listening of the receiver on that socket.

```cpp
void PRRTReceiverStrategy::createLocalSocket(const INetAddr *address)
    throw (const ConnectionException &, const std::exception &)
{
    // This method is used in requestConnection() and acceptConnection()
    // to set up the PRRT socket with the given internet address.

    // Get the local host address.
    char buffer[256];
    ::gethostname(buffer, 256);
    string hostname(buffer);

    // Check if we have an explicit address or implicit address.
    if (address) {
        // We have an explicit address.
        INetAddr local_addr = *address;

        if (local_addr.getPort() == 0) { // Bind to "random" free port, if the port is not specified.
            getAdaptPRRTSocket().tryBind(local_addr, prrt_min_port, prrt_max_port);
        } else { // Bind to fixed port.
            getAdaptPRRTSocket().bind(local_addr);
        }
    } else { // We have an implicit address.
        INetAddr local_addr;
        // Use local host address for binding.
        local_addr.setAddress(hostname.c_str());

        // Bind to "random" free port.
        getAdaptPRRTSocket().tryBind(local_addr, prrt_min_port, prrt_max_port);
    }

    // NOTE: The following lines are only for PRRTReceiverStrategy not PRRTSenderStrategy.
    // Set receive buffer size on socket.
    setRecvBufferSize(200 * 1316);
```

The system call of \texttt{gethostname} will randomly choose one of its IP addresses; the system must support address lookup of the local host.

\textsuperscript{28} The size we use is a rough value, not a precise bound. Because it will vary according to computer performance. In our case we got packet loss if buffer sizing was $100\times1316$ bytes or less. For an interactive application like TV channel switching, buffer sizing is subject to the permitted latency up to one second i.e. maximum buffer sizing roughly $500\times1316$ bytes. Additionally, it’s worth noting that the maximum kernel UDP buffer size has its default value depending on various operating system. Thus, a change on that default maximum size should be considered for enlarging the buffer size as expected.
4. Implementation: Integration of PRRT into NMM

```cpp
// Register the socket at NetReactor.
SNetReactor::getInstance().registerSocket(getAdaptPRRTSocket().getSocket(), this);
```

Listing 4.11: `createLocalSocket` Method of `PRRTRceiverStrategy`

4.6. Application of PRRT in NMM

The new transport strategy can be used in the transparent communication infrastructure of a distributed multimedia application represented by an NMM flow graph, to undertake message transmission between nodes. We call this application mode *Instream Channel* mode (see section 4.6.1). In this mode the multimedia application and even the nodes often don’t need to care about the transmission at all, as the NMM communication channels do and automatically handle the exchange of messages with the corresponding strategy. In addition, considering the compatibility layer and some central service areas, NMM offers another application mode for the prevalent or emerging network protocols. This compatibility and specialized mode is called *Client / Server* mode (see section 4.6.2), in which the transport strategy is applied in a specific node to send / receive the messages to / from the non-NMM components via the network protocol integrated in that strategy.

Our `PRRTStrategy` fulfills those two application modes of NMM, which are respectively explained below.

4.6.1. Instream Channel Mode

Within NMM all message transmission relies on the communication channel of NMM. The communication channel between nodes is called *Instream Channel*, which determines and manages a set of transport strategies for performing the communication. This channel is created when two jacks are connected to transmit messages. Inside the channel various transport strategies can be arranged to be responsible for transmitting messages with a specific type (Buffer and Event). The type buffer of message is used to deliver multimedia data between nodes while event can be used for carrying control information. `PRRTStrategy` is only for forwarding multimedia data, since events should be transmitted in a reliable way whereas PRRT just supports partial reliability.

NMM aims at providing a communication framework for efficiently integrating a network protocol into the multimedia middleware solution. As long as a transport strategy has realized the transport mechanism of that protocol, that transport mechanism can often soon be employed in an NMM application as a “Black-Box”. This is done by instream channel adding the strategy for a sub-binding (connection), just like adding a conduit for connecting the water between two pools. So `PRRTStrategy` can be set up in instream channel and perform the transmission of multimedia data via the PRRT protocol. However, each transport mechanism has its distinct properties. To make the `PRRTStrategy` well collaborate within NMM, we have to adjust the strategy to fit the NMM application.
4.6. Application of PRRT in NMM

Target Delay

A first modification of the PRRTStrategy for instream channel mode is to speed up the transmission during the format analysing.

The background is the following. In NMM, before a binding is created between Jacks (in other words, a connection is created between nodes), a format recognition is required to determine whether an output format of data stream that a node is going to process matches an input format that another node supports. An binding is successfully set up only if the data format matches. Therefore the format being handled has to be specified before the matching is executed. In some situations where the data format is already explicitly given to the application nothing should be done but directly performing the matching of the supported format. Most often, the application has to dynamically determine the data format the application is handling by analysing the first few chunks of the data stream, because the particular format specification is usually unknown. Hence, each NMM node can internally carry out this analysing to determine the data format if the data format is unknown to that node. Once the data format between two nodes matches, the binding between them can be established. Then, this operation is performed step by step along all nodes in the flow graph. NMM utilizes an interaction mechanism on message transmission during format analysing, i.e. each sending operation is performed on a sender node only if a request for that sending is confirmed from the receiver node. Otherwise excessive amounts of data would be left in the receiver socket buffer, and more worse buffer overflow will occur and then data will get lost. In the interaction a receiver node sends a request for the next packet to the sender node just after it gets a packet. So the duration of the transmission becomes an important issue, since a long transmission results in a later delivery of the next packet. And later delivery may also lead to buffer space overflow on the sender node, if the sender receives a live stream as a queue node and cannot timely consume the arrived data stream. Moreover, format negotiation and setup of the flow graph already spends the major proportion of the interaction latency, so that the duration of the transmission is being restricted. Overall, the transmission delay has to be limited during the format analysing.

On account of that interaction mechanism, PRRTStrategy requires accelerating the whole PRRT transmission process. The most critical factor in PRRT which influences the duration of PRRT transmission is the target delay, because it directly affects the time of reading out the packet from the PRRT receiver socket. If the target delay is long, the packet “stays” longer in the PRRT module, because the module may wait longer for arriving packet before decoding it, considering that the packet may be retransmitted due to loss. But the timing for reading out packets is very vital to the performance of FEC in PRRT. Hastily reading out packets earlier will degenerate the effect of FEC or even invalidate FEC in the worst case according to the RTT, since the packet has been read out and the decoding of that block is impossible. Now we’re facing a trade-off between time and reliability. To maintain the acceptable interaction latency during the phase of setup of a flow graph, we make a slight change in the PRRTSenderStrategy. The target delay is firstly set to 1 millisecond during format analysing so that the duration of whole transmission is speeded up by earlier reading out the packet. After all format analysing of the whole flow graph, more precisely after the setup of the whole graph, the target
4. Implementation: Integration of PRRT into NMM

delay is switched back to the expected target delay. When the setup of a graph is finished
depends on how complex the graph is. Currently we keep the target delay to 1 millisecond
in the first 1000 sending operations to get around the above issue during the setup of
the flow graph. Nevertheless, this is just a temporal solution. A neat solution should be
that a switch back is signaled by the application through an out-of-band channel once
the setup of the graph is completed.

Parallel Binding

Compared to the low-level transport component (transport strategy) in NMM, a high-
level transport component is called “Binding”. A binding represents a connection between
two components as the first concept to an application in the communication framework
of NMM. It can manage and use one or more sub-bindings, each of which is realized by
a specific transport strategy and performs the corresponding transport mechanism and
provides particular transport properties, to transmit various messages through proper
sub-bindings in terms of different Quality of Services (QoS) requirements (e.g. reliability
and latency). However, a single multimedia stream can consist of diverse ordered mes-
sages and the ordering of those messages must be preserved after the parallel transmission.
Parallel Binding [Rep05], as one of the significant features in instream communication
channel of NMM, is proposed to this parallel transmission and the ordering constraint
along the transmission. The basic idea of parallel binding, as shown in figure 2.2.5, is: a
demultiplexer is used at the sender to dispatch messages of different types to the corre-
sponding sub-bindings; a multiplexer is used at the receiver to reconstruct the ordering
of messages.

The synchronization algorithm in the demultiplexer and multiplexer is described in
[Rep05], based on different combinations of sub-binding types\textsuperscript{29}. We just address one
of the cases: synchronization between two reliable sub-bindings, even though PRRT is
not a reliable transport protocol. This is due to the fact that the parallel binding in the
current open source release version (2.2.1\textsuperscript{30}) of NMM is only available for the reliable
sub-bindings, i.e. the parallel binding of that version supports merely reliable transport
strategies at the moment. Therefore our PRRTStrategy has to work as a “reliable” strategy
with an intention of tests and is equipped with the necessary mechanism for being later available in the parallel binding.

To synchronize messages from reliable sub-bindings NMM assigns each message with
a unique sequence number. In fact the sequence number is also prerequisite for the syn-
chronization of other cases. The sequence number is represented in the Timestamp of the
Message as a 64 bit stream counter (stream_counter). This counter is strictly increas-
ingly allocated to each ordered message which being dispatched in the multiplexer. The
demultiplexer reconstructs the ordering of incoming messages by comparing the sequence
numbers associated to messages. The demultiplexer can forward a message to the receiv-
ing component only if all messages whose sequence number is smaller than the one of that

\textsuperscript{29} There are three types of sub-bindings: reliable sub-bindings, unreliable sub-bindings and multimedia
sub-bindings (unreliable but subject to certain formats).

\textsuperscript{30} NMM 2.2.1 is the NMM version into which we integrate PRRT.
4.6. Application of PRRT in NMM

message have been delivered. Otherwise it must wait for the preceding messages. This mechanism wouldn’t get in an infinite wait loop, because all messages are transmitted in a reliable way, i.e. each message must be able to arrive in the multiplexer sooner or later.

So a second modification for suiting PRRTStrategy to the instream channel mode is to characterize the strategy with the capability of the reliability in NMM. This is done by adding a transport capability flag TS_RELIABLE in the registration of PRRTStrategy, as already described in section 4.5.1. 31

Eventually, we need to inform the receiving side about the sequence number of the message. Since PRRT doesn’t have format limitation on the data it carries, the application can manipulate the payload of a PRRT packet at will. A typical approach is to convey the sequence number together with its associated message in one PRRT packet. The concrete implementation is already presented in listings 4.8 and 4.10, where we simply append the 64 bits sequence number to the end of the forwarded multimedia data before transmission and extract the sequence number out of the packet after transmission. Figure 4.6.1 illustrates the PRRT packet format in the instream channel mode.

![Figure 4.6.1: PRRT Packet Structure in Parallel Binding](image)

Again, note that PRRT is not a reliable transport strategy. A proper integration of PRRT into NMM in the instream channel mode requires synchronizing messages from PRRTStrategy as a unreliable strategy in the parallel binding. Hence an unreliable strategy should offer the transmission delay to the synchronization of the parallel binding. The getTransmissionDelay method in the PRRTStrategy returns an estimated delay for that purpose. This delay is estimated by a half of RTT, which is provided by the PRRT module.

4.6.2. Client / Server Mode

Most NMM transport strategies can be used in the distributed application structure of the client / server model. This model in NMM is designed to provide multimedia services for client applications. On the other hand, it allows other non-NMM applications compatibly to join into distributed multimedia systems of NMM. The client / server application mode of transport strategies in NMM is realized by bundling all transport

---

31 Regarding to the fake absolute reliability, it might happen that packet loss occurs in the transmission through our strategy due to a bad network condition or socket buffer overflow during the setup of flow graph. We introduce a work around for those situations to get through the tests of the parallel binding. If a message arrives whose sequence number is greater than the sequence number of last message by more than one, the lost messages in the gap of those two messages are “regenerated” with the corresponding sequence numbers by duplicating the current message and then forwarded to the receiving component. The infinite waiting at the multiplexer is avoided at the price of slightly degraded quality in presence of residual loss of PRRT packets.
strategies which support client / server mode into two specialized nodes (NetSourceNode and NetSinkNode), which can be respectively used on the client side and the server side. Each of both cooperates with the corresponding part of the transport strategy pair to connect one endpoint of the communication via the protocol mechanism that the transport strategy realizes. Both nodes can manage multiple pairs of strategies and select one of those to set up the communication to any supported network needed by the application, just like a multi-switcher switching the connections. At the same time they work as a mediator too, for conveying messages between network and NMM.

We implement two new node classes for the integration of PRRTStrategy into the client / server mode and name them similarly to the NetSinkNode and the NetSourceNode but with the prefix PRRT: PRRTNetSinkNode and PRRTNetSourceNode. Both new nodes are proposed to achieve what the original nodes can do and additionally support the PRRT transmission. Note that in this section we just discuss how the strategy is used in the node to achieve a distributed multimedia application of the C/S module. More details regarding to the processing in nodes are out of scope of this thesis and please refer to [Loh05].

Controlling Interface

The PRRTNetSourceNode and PRRTNetSinkNode provide multi-protocols communications for multimedia applications. Within the proxy architecture of NMM, applications can choose which transport strategy is used in these two nodes. The communication between applications and nodes is accomplished by sending events to each other in a request-reply interaction paradigm. NMM employs an Interface Definition Language (IDL) to abstract interface methods that an object supports and that describe the corresponding events. From an IDL definition (in a file with suffix “.idl”), IDL compilers automatically generate an Interface class and an Implementation class (with suffix “Impl”) respectively for the client and the server. Note that in this context the client is the application and the server means a node. If a node inherits the implementation class and implements the interface, an application can control the node through the interface object.

Within such an object-oriented interface system, the following has been done to make the switching of strategies available to applications. We introduce an IDL file named IPRRTNet.idl, which defines a control interface on nodes for the application. PRRTNetSinkNode and PRRTNetSourceNode derive from the implementation class IPRRTNetImpl and implement the interface methods defined in the IPRRTNet.idl.

32 Naming convention: NetSourceNode represents a source node in an NMM flow graph which receives data from a specific network and produces multimedia buffers containing received data for its successor; in contrast, NetSinkNode is a sink node that consumes multimedia buffers from the predecessor and sends them to the network.

33 This will simplify the later integration of PRRTStrategy into the NetSourceNode and NetSinkNode. Furthermore, we didn’t directly integrate PRRTStrategy into both original nodes. This provided us two independent nodes, which were not affected by the new implementation, and they can offer live streaming during the tests as the queue nodes.

34 The interaction between applications and NMM objects are realized in the Proxy Design Pattern by sending events to each other. For more details please refer to [Loh05].
4.6. Application of PRRT in NMM

module NMM {

    interface IPRRTNet {

        /** Sets up parameters of PRRTStrategy. 
         * 
         * @param local_address local IP address 
         * @param local_port local port 
         * @param peer_address peer IP address 
         * @param peer_port peer port 
         */
        void setPRRTStrategy(in string local_address , in nmm_int32 local_port ,
                              in string peer_address , in nmm_int32 peer_port);

        // Other interface methods
        // ...
   );
}

Listing 4.12: Interface Definition Language (IDL) Description in IPPRTNet.idl

The above listing 4.12 extracts just one method from the IPPRTNet.idl as an example of an IDL definition. From the example the parameters of the method are additionally indicated as in parameter, which means that the parameters are only sent to the method.\(^{35}\) With these parameters information PRRTNetSinkNode and PRRTNetSourceNode can create the corresponding transport addresses (PRRTAddress see section 4.4) and then set up the transport strategy by the setTransportStrategy method. Moreover, similarly to IPPRTNet.idl, we also create the IPPRTParameter.idl file to define the interface methods related to the setting of PRRTStrategy, so that the application can configure the strategy, e.g. the setAdaptationMode method, which determines the mode of adaptation. It should be noted that the parameters setting of the PRRT module is placed in the sender side and therefore all interface methods of parameters setting are only implemented in the PRRTNetSinkNode.

Setting up the Strategy

Once the event of setPRRTStrategy from the application arrives at the PRRTNetSinkNode or PRRTNetSourceNode, the setPRRTStrategy method of PRRTNetSinkNode or PRRTNetSourceNode is executed. Then the node sets up the strategy, as shown in listing 4.13. The code list is taken from PRRTNetSinkNode and setting up a strategy in PRRTNetSourceNode is similar. The setup of the strategy is done in the following three steps: creating and storing the transport strategy addresses according to the given IP addresses; creating the corresponding transport strategy and setting the local and remote addresses of the strategy; initializing the relations between the node and the strategy (e.g. PRRTSenderStrategy is the buffer receiver to PRRTNetSinkNode and the node is a signal receiver to the strategy.)

\[
\text{void PRRTNetSinkNode:::setPRRTStrategy(} \\
\quad\text{const string& local_address, const nmm_int32& local_port,} \\
\quad\text{const string& feedback_address, const nmm_int32& feedback_port)}
\]

\(^{35}\) Other indicators out and inout respectively mean that the parameter only received from the method and that the parameter can be sent to and received from the method.
4. Implementation: Integration of PRRT into NMM

// This method is called by the application through IDL interface method.

// Create transport addresses to store the address information.
PRRTAddress local_addr(local_address, local_port);
PRRTAddress remote_addr(feedback_address, feedback_port);

// Set up the transport strategy.
setTransportStrategy(&local_addr, &remote_addr);

void PRRTNetSinkNode::setTransportStrategy(const TSAddress* local_address,
const TSAddress* remote_address)
throw(const BindingException&, const std::exception&)
{
    // This method is a help method for setting up the transport strategy.
    // Store addresses.
    m_local_address = local_address->copy();
    m_remote_address = remote_address->copy();

    // Create TransportStrategy which can connect to the given address.
    m_transport_strategy = STSFactory::getInstance().createTransportStrategy(
        local_address->getStrategyType(), TSDescription::ITS_ONE_WAY_BUFFER_RECEIVER);

    // Connect and configure transport strategy.
    CEvent empty_configuration;
    m_transport_strategy->requestConnection(
        *local_address, empty_configuration, empty_configuration);

    // Set remote address if the strategy is PRRTSenderStrategy.
    PRRTSenderStrategy *prrt_strategy =
        dynamic_cast<PRRTSenderStrategy*>(m_transport_strategy);
    if (prrt_strategy) {
        PRRTAddress *prrt_address = dynamic_cast<PRRTAddress*>(m_remote_address);
        if (prrt_address) {
            prrt_strategy->setRemoteAddress(*prrt_address);
        }
    }

    // Get the transport role interface of the strategy and
    // set the signal receiver for the strategy.
    m_buffer_receiver = dynamic_cast<ITSOneWayBufferReceiver*>(m_transport_strategy);
    m_buffer_receiver->setSignalReceiver(new PRRTNetSignalReceiver(this));
}

Listing 4.13: Setting up PRRTSenderStrategy in PRRTNetSinkNode

Sender Side

A flow graph containing PRRTNetSinkNode can act as a server to provide services for other remote clients through the PRRT protocol. However, PRRTNetSinkNode cannot yield services by itself. It is just only in charge of transmitting services. So its tasks can be summarized into two parts: receiving multimedia buffers from other NMM nodes and forwarding them to the strategy to perform a sending operation. These two tasks are implemented in the method processBuffer, which derives from the GenericSinkNode (more exactly from the GenericNode – the superclass of nodes) and provides the core functionality. At first, a buffer as the method parameter is passed from the input queue of the node into the method. Then the method calls the PRRTSenderStrategy, which is already set up during the construction of the node, to receive the buffer by the receiveOneWayBuffer of the strategy. The strategy schedules to send that buffer as explained in section 4.5.2. The code snippet of the processBuffer is listed in listing 4.14.
4.6. Application of PRRT in NMM

Message PRRTNetSinkNode::processBuffer(Buffer* buffer)
{
    // This method is internally called by the node itself.

    // Protect the processing state.
    MutexGuard mg(m_internal_mutex);

    // Forward buffer to PRRTSenderStrategy.
    Result status = m_buffer_receiver->receiveOneWayBuffer(buffer);
    if (status == BLOCKED) {
        // Get blocked and wait for FREE_QUEUE_SLOT signal.
        m_is_blocked = true;
        while (m_is_blocked) {
            m_internal_cond.wait();
        }
    }

    // A sink node always returns 0, since no output stream.
    return 0;
}

Result PRRTNetSinkNode::enableInputStream()
{
    // This method is called by the PRRTNetSignalReceiver.

    // If a FREE_QUEUE_SLOT signal is received the input stream will be enabled.
    MutexGuard mg(m_internal_mutex);
    m_is_blocked = false;
    m_internal_cond.notify();
    return SUCCESS;
}

void PRRTNetSignalReceiver::receiveSignal(Signal signal)
{
    // This method is called by the PRRTSenderStrategy.

    // Call PRRTNetSinkNode to handle a FREE_QUEUE_SLOT signal.
    if (signal == FREE_QUEUE_SLOT) {
        m_prrtnetsinknode->enableInputStream();
    }
}

Listing 4.14: Handling Buffer and Signal in PRRTNetSinkNode

Due to the synchronization mechanism described in section 4.5.2, PRRTNetSinkNode gets a BLOCKED signal and becomes idle to wait for a FREE_QUEUE SLOT signal, after it forwards the buffer to the PRRTSenderStrategy. Hence, PRRTNetSinkNode need to receive and handle the signal from the strategy. A typical way is to inherit the ISignalReceiver interface and implement the receiveSignal method. However, the ISignalReceiver interface is already used by the GenericNode to handle internal signals. In order to get around ambiguities of the base class, we introduce a dedicated signal receiver for the PRRTNetSinkNode to receive the signals from PRRTSenderStrategy. We call it PRRTNetSignalReceiver, which calls the PRRTNetSinkNode to enable the input stream if acquiring a FREE_QUEUE SLOT signal. The relation of two objects is shown in the class diagram of figure 4.6.2 and the collaboration of handling a signal from the strategy is demonstrated also in listing 4.14. The sequence diagram in figure 4.6.3 illustrates this synchronization mechanism. The processBuffer method is internally called by the sink node itself if there is a buffer in its input stream queue, which stores buffers passed from the predecessor.
4. Implementation: Integration of PRRT into NMM

Figure 4.6.2: Class Diagram of PRRTNetSinkNode

Receiver Side

At the client side, the PRRTNetSourceNode is implemented to receive data from a PRRT network with the PRRTReceiverStrategy. Figure 4.6.4 illustrates its class diagram. Similarly to PRRTNetSinkNode, the source node cannot consume the incoming data by itself, it has to forward them to its succeeding node. Therefore, we explain the implementation of the PRRTNetSourceNode in two aspects: how the source node cooperates with the receiver strategy and how it produces multimedia data for its successor.

Message* PRRTNetSourceNode::processBuffer(Buffer* /*buffer*/) {
    // This method is internally called by the node itself.
    // Protect the access of current message.
    MutexGuard current_message_guard(m_current_message_mutex);

    if (!m_current_message) {
        // ...
// Stop producing until we are notified that we have a message to send.
return 0;

// Message to return.
Message* out_message = m_current_message;

// Start producing messages again and wait for receiving more messages.
m_current_message = 0;
m_buffer_sender->receiveSignal(FREE_QUEUE_SLOT);
return out_message;

Result PRRTNetSourceNode::receiveOneWayBuffer(Buffer* buffer)
{
    // This method is called by the PRRTReceiverStrategy.
    // Protect the access of current message.
    MutexGuard current_message_guard(m_current_message_mutex);
    // Store the message.
    m_current_message = buffer;
    m_current_message_cv.notify();
    return BLOCKED;

Listing 4.15: Buffer Receiving and Producing in PRRTNetSinkNode

As mentioned before in section 4.5.4, the PRRTReceiverStrategy forwards the received data from the PRRT socket to its partner component by calling the receiveOneWayBuffer method of the partner. That partner of the strategy is the PRRTNetSourceNode in this context. Again under the synchronization the buffer forwarding promotes the producing operation in the processBuffer method of the source node. During the phase of producing, the source node signals the strategy for requesting more buffers after taking over the buffer and then passes the buffer to its following node by returning the buffer in the
4. Implementation : Integration of PRRT into NMM

Figure 4.6.4.: Class Diagram of PRRTNetSourceNode

processBuffer method. Note that the PRRTNetSourceNode has no preceding node. To trigger the internal call of the processBuffer method we set the parent of the input stream queue (StreamQueue\(^36\)) to the PRRTNetSourceNode and the partner of the queue to the PRRTReceiverStrategy during the construction of the node, since the source node reads from the queue and the strategy writes to it.

\(^36\) The class StreamQueue represents an input stream queue of a node.
5. Test Results

In this chapter we evaluate the integration of PRRT into NMM and the performance of PRRT in wireless scenarios. At first we describe the setup of the tests in section 5.1 and then discuss how the test was performed and the results in section 5.2.

5.1. Setup of Tests

This section describes our wireless testbed including the hardware and software and the placement of wireless stations and the Access Point (AP) as well as their configurations.

5.1.1. Hardware and Software

In our evaluation we use standard laptops with off-the-shelf WLAN boards as the wireless stations. A Linksys advanced wireless-N router is selected as the AP with support of IEEE 802.11n, which can offer wireless coverage throughout larger homes. Further details on the hardware and software are listed below:

- **Hardware**
  - AP: Linksys E2000 advanced wireless-N router.\(^1\)
  - Sender: one station with the Model (CPU: Duo T9300 2.5GHz, RAM: 2GB).\(^2\)
  - Receiver: two stations with the Model (CPU: Duo L2400 1.66GHz, RAM: 2GB, wireless board: Intel Pro/Wireless 3945ABG). Both are connected to the AP by wireless LAN.

- **Software**
  - Operating System: Linux kernel version 2.6.27-9-generic, 2.6.28-17-generic are installed in the different laptops.\(^3\)
  - PRRT Protocol: Revision 195 \(^4\)
  - NMM: Revision 117.
  - Streaming Server and Clients: Client/Server mode using PRRTNetSourceNode and PRRTNetSinkNode of NMM open source version 2.2.1. The corresponding graph descriptions are shown in section 5.1.2.\(^5\)

\(^1\) Only one hop multicast is evaluated in this thesis. Also roaming from one AP to another is not considered here, because it may take long delay.

\(^2\) We select the most powerful laptop among all stations as the sender, because a sender should provide constant and stable transmission of multicast packets so that the whole wireless multicast test won’t be severely affected by the sender.

\(^3\) It should be noted that PRRT requires Linux kernel version 2.6.26 or later.

\(^4\) For more details on using PRRT please refer to the PRRT tutorial in appendix B.

\(^5\) For more details on using NMM please refer to the NMM tutorial in appendix C.
5. Test Results

- Network Emulator: netem 2.6.\(^6\)
- Software for SNR recorder: supported by the drivers and Linux command `iwconfig`. The record of SNR is only available for the wireless receivers.

- **Distribution of hardware:** The placement of all wireless stations and the AP are illustrated in figure 5.1.1, in which “AP” is the location of the AP; “S” is the sender position and the sender has a wired connection to the AP; “W” is the position of the wireless receivers, one is stationary and one is moving along the line with the arrow. The number near the “W” means the test number: “1” means that the receiver participates in Test 1 (see section 5.2), “2” means that the receiver participates in Test 2 (see section 5.2) and “1/2” means that the receiver participates in both tests.

![Figure 5.1.1: Distribution of the AP and Wireless Stations](image)

5.1.2. Configuration of Setup

In this section we discuss the configuration of the setup for the evaluation of PRRT on 802.11a multicast. Since the most significant influence on the test is the characteristic of the wireless network, the configuration of the AP is the dominant one among all other configurations. But it’s important to note that some settings on the wireless end hosts may affect the results of the evaluation e.g.:

\(^6\) For more details on using netem please refer to the netem tutorial in appendix D.
- Maximum socket receive buffer size: a too small buffer size might cause artifacts in the video during live streaming, dependent on the computing performance of the stations. We set it to 200 × 1316 as default.

- sleep or power-down modes: the system of each station need to deactivate the sleep and power-down modes, because entering those modes will make the wireless board send a message to the AP so that the AP buffers the packets until the station becomes available again. [Duj06]

The following settings of the AP are configured before starting the evaluation:

- Frequency Spectrum: 5 GHz band\(^7\) is selected in our tests, since the packet loss rate in 5 GHz band is lower than that in 2.4 GHz band. We tried 2.4 GHz in a few tests, but it turned out that it was rather unstable.

- Beacon\(^8\) Interval: we set it to 20 milliseconds.\(^9\)

- DTIM\(^10\) Interval: is set to default value 1, which means that DTIM is sent in each Beacon by the AP. It should be noted that Beacon Interval and DTIM interval need to be adjusted to the proper values according to the data rate of the multicast traffic. Otherwise it will draw the battery power as if the power-save mode is disabled.

- Transmission Setting: The mixed mode with 802.11b is disabled, because the transmission rate of mere 2 Mbps in 802.11b will degrade the performance of our live streaming test.

- MIMO: MIMO functionality of 802.11n cannot be applied in multicast, because each receiver might observe a different signal path depending on its location so that the orthogonality of the signals cannot be hold in the simultaneous transmission any more.

Finally we evaluate the new functionality of PRRT in NMM with multicasting real-time multimedia streams. As NMM is a graph based application, we firstly sketch the corresponding graphs for the application on the sender and receiver, respectively illustrated in figure 5.1.2 and figure 5.1.3.

The flow graph on the server represents the server functionality which constantly receives an IP stream with the format of MPEG Transport Stream via the UDP protocol and then multicasts the stream via the PRRT protocol. The following textual description is the so called NMM “Graph Description”, which is used by \texttt{clic}\(^11\) to create the NMM

\(^{7}\) Note that the use of 5 GHz for WLANs is somewhat limited in some geographical locations for regulatory acceptance.

\(^{8}\) A Beacon is a MAC control frame that is broadcasted by the AP to synchronize the wireless network.

\(^{9}\) The 802.11a WLAN suffers from the Dynamic Frequency Selection (DFS), which is mandatory in the 5 GHz domain. We adjusted the Beacon interval to 20 ms which most probably reduced the DFS activities of the AP.

\(^{10}\) A field in Beacon, called Delivery Traffic Indication Message (DTIM), is used to wake up the receivers to receive the multicast packets on a 802.11 WLAN with enabled power-save mode. The interval of DTIM offers a trade-off between the power saving and the performance on the WLAN.

\(^{11}\) The Command Line Interaction and Configuration (clic) is an application in NMM for performing an NMM flow graph represented in a textual form. For further details please refer to [Mot10c].
5. Test Results

![Flow Graph on the Server Side](image)

Figure 5.1.2.: Flow Graph on the Server Side

% This graph description with the file name "server.gd"
% describes the NMM flow graph on the server side.

% NetSourceNode $ setRawUDPStrategy("10.1.14.139", 12358) CONSTRUCTED
% setPayloadType(33) CONSTRUCTED
!
PRRTNetSinkNode $ setPRRTStrategy("224.0.2.2", 5005, "224.0.2.2", 5004) CONSTRUCTED
% setPayloadType(33) CONSTRUCTED
% setPRRTTargetedDelay( 200 ) CONSTRUCTED
% setPRRTAdaptationMode( "auto" ) CONSTRUCTED

flow graph on the server and to perform the depicted server functionality.

In this graph description NetSourceNode calls two specific methods\(^\text{12}\) in the state CONSTRUCTED in the life-cycle of the node. The IDL method setRawUDPStrategy specifies that NetSourceNode utilizes RawUDPStrategy\(^\text{13}\) to receive the stream with the address 10.1.14.139 and the port 12358; the method setPayloadType indicates that MPEG transport stream has the payload type of 33. Furthermore, while resending the stream, PRRTNetSinkNode is told to use PRRTStrategy with the source address 224.0.2.2 and port 5005 and the destination address 224.0.2.2 and port 5004. Also the payload type of MPEG transport stream is set to 33 by setPayloadType.

The client receives the PRRT packets which carry the MPEG TS data sent by the server. This is done by PRRTNetSourceNode in NMM with the similar interface methods to those of NetSourceNode on the server. The slight difference is that setPRRTStrategy requires the destination address 224.0.2.2 and port 5005 for the feedback in addition to the source address 224.0.2.2 and port 5004. Thereafter MPEGSDemuxNode demultiplexes the received stream and delivers the video stream to MPEGVideoDecodeNode by the output jack with the jack tag \(\text{mpeg\_video0}\) and the audio stream to MPEGAudioDecodeNode by the output jack with the jack tag \(\text{mpeg\_audio0}\), respectively. Then through decoding the streams the video and audio can be played back respectively by XDisplayNode and ALSAPlaybackNode at the end. The interface method setDownstreamMaxSize is used to

\(^{12}\) In NMM a node can perform additional methods to set up its particular configuration. Those methods are specified in NMM Interface Definition Language (IDL) and are implemented in the NMM nodes if needed.

\(^{13}\) Note that RawUDPStrategy is different to UDPStrategy in NMM: the former one sends the stream data without any NMM header information in the UDP payload; whereas the payload of a UDP packet contains the specific NMM headers which might not be recognized by the third party applications.

\(^{14}\) A jack tag associated with each jack is a unique string to distinguish different input jacks or output jacks. The string "default" is used as a default jack tag if a node has only a single input jack or output jack or the node doesn’t specify other jack tag.
To start the server application, we run the graph description of the server with `clic` using the command:

```
./clic_server.gd
```

For the client application, we execute the following command instead:

```
./clic_client.gd
```

Then we see the live video in a popped up window.

### 5.2. Results

In this section we discuss the test results of our implementation and show the performance of the PRRT. First of all we describe how the tests were performed.

#### Test Description

In the performance test of the PRRT on wireless LAN, we firstly examine PRRT in an emulated scenario of WLAN, in order to compare the performance of PRRT in the simulated case and the real case. In the real WLAN test, we also take the obstacle of the wireless signal and the portability of the receiver into account. Therefore, we did the following tests:

- **Test 0**: We used “netem” to emulate a WLAN scenario with the packet loss sequence, generated for the evaluation of the smoothing function in section 3.3 and enlarge the queue size\(^{15}\) to 1000 NMM Buffers.

\(^{15}\)The queue size in NMM indicates the number of NMM buffers in the stream queue.
5. Test Results

% This graph description with the file name "client.gd"
% describes the NMM flow graph on the client side.

```
PRRTNetSourceNode $ setPRRTStrategy("224.0.2.2", 5004, "224.0.2.2", 5005) CONSTRUCTED
$ setPayloadType(33) CONSTRUCTED
{
  { 
    MPEGTSDemuxNode $ setDownstreamMaxSize(1000, "default") CONSTRUCTED
    { 
      ["mpeg_video0"]
        MPEGVideoDecodeNode $ setDownstreamMaxSize(1000, "default") CONSTRUCTED
        XDisplayNode
    } 
    { 
      ["mpeg_audio0"]
        MPEGAudioDecodeNode $ setDownstreamMaxSize(1000, "default") CONSTRUCTED
        ALSAPlaybackNode
    } 
  }
}
```

the fixed delay of 10 ms. The emulation was done at the sender by setting the packet loss and delay of the outgoing network interface every second, according to the generated sequence.

- **Test 1**: we placed one wireless receiver in a different room near the room where the AP was. The straight line distance was about 6 meters and the line of sight was obstructed by a wall between them. One receiver was placed in the same room as the AP.

- **Test 2**: One wireless receiver still stayed in a different room to the one of the AP, as in Test 1. Another wireless receiver was steadily moving back and forth along the corridor. The maximum range of the line of sight to the AP was about 9 meters during the moving.

- **Test 3**: This test is similar to Test 1, but the average data rate of the stream in the test was 12 Mbps instead of 4 Mbps.

We chose an SDTV channel in Test 0, 1 and 2 and an High-definition television (HDTV) channel in Test 3, respectively. The data rate of the streams has been variable around 4 and 12 Mbps, respectively. Figure 5.1.1 illustrates the distribution of all the hardware and the route of the portable receiver.

We started multimedia streaming with a variable data rate\textsuperscript{16}, which was provided by an IP streaming server. The sender re-streamed the multimedia to the PRRT network. After the sender started streaming, the receivers started receiving the data and playing back the audio and video. During the playback we were recording the Signal-Noise-Ratio (SNR), PLR and RTT at each receiver about every second. The results are shown in the following.

\textsuperscript{16} The data rate of multimedia streaming could oscillate.
5.2. Results

We investigate the performance regarding to different residual RI, residual PLR, RTT, PLR and SNR shown in the figures. We start with the results of the emulated scenario, illustrated in figure 5.2.1. The figure shows that the adaptation of PRRT can successfully select and apply the appropriate parameters according to the current PLR and RTT. Because the residual PLR of two stations stayed at 0 and the RI changed due to using different optimized parameters based on the varying PLR.

Figure 5.2.2 depicts the performance if the two receivers are fixed at some position. It shows that

- The receiver with no obstruct in the line of sight to the AP started earlier than the other about 10 minutes. Its statistics kept stable, even after the receiver which was placed in another room started to receive the streaming. This means that the start of receiving of one receiver should not affect other receivers in the same multicast group.

- A wall between the AP and the receiver could reduce the SNR by roughly 10 dB. However, generally, the changes of the statistics on that receiver was still roughly same as the one on the receiver with direct line of sight to the AP. This implies that the quality of the connectivity on both receivers are mainly dependent on the network and the AP, not on the receiver itself, in this case (the positions of the receivers are fixed).

- RTT / PLR in wireless LAN constantly varied and a peak occurred roughly periodically, it might be due to that the queue of the AP is overflow at times, so that both receivers have long RTT and worse PLR.

- The peak in the residual RI means that the FEC was applied at that point, because the RTT increased significantly and FEC was used to fix the PLR. And it also shows that the adaptation worked according to the network conditions and it could react in time. On the other hand the RI curves indicate the performance of adaptive PRRT: it only burdens the network traffic if in need. However, they also show that the parameters switching during the adaptation was relative frequently, since the change of the residual RI tended to that of the PLR and the RTT.

- The residual PLR was higher than the PLR requirement ($10^{-6}$), especially when the RI was high, i.e. when the parameters were switched from the ARQ scheme to the FEC scheme. It means that the PLR requirement could not be held during the adaptation. This should be improved in the future.

- The curve of the residual PLR is sawtooth. Since the residual PLR is calculated through the ratio of accumulating all lost packets and sent packets in a specific duration, it steeply increases to the peak at some point and then decrease with an inverse ratio curve.

Figure 5.2.3 compares the performance between the one fixed receiver and one moving receiver (Test 2). The SNR of the moving receiver was periodically varying, since one
Figure 5.2.1.: Performance of PRRT on Simulated Multicast Scenarios with Data Rate of 4 Mbps
5.2. Results

Figure 5.2.2.: Performance of PRRT on Multicast Scenarios (Stationary) with Data Rate of 4 Mbps
receiver was moving back and forth on the floor and the walls reduced the signal strength. In the worst case, the SNR dropped to 2 dB and the PLR increased to the peak along the decrease of the SNR. The SNR was sometimes higher than the one of the fixed receiver, because the receiver could directly see the AP during the moving. Other characteristic can be generally referenced to Test 1.

Figure 5.2.4 shows the performance of PRRT in the case of the higher data rate. PRRT works as well as in Test 1. The residual PLR exceeded the theoretical value.
5.2. Results

Figure 5.2.3.: Performance of PRRT on Multicast Scenarios (Moving) with Data Rate of 4 Mbps
5. Test Results

Figure 5.2.4.: Performance of PRRT on Multicast Scenarios (Stationary) with Data Rate of 12 Mbps
6. Summary and Future Work

This chapter will summarize the work of this article. Section 6.1 will recap the previous chapters and sections and then draw the conclusion. Finally, we will talk about the future work in section 6.2.

6.1. Summary

In this thesis we study the performance of PRRT in wireless multicast scenarios by integrating PRRT into the NMM. We implement a new pair of transport strategies in NMM which can perform the transport mechanism using the PRRT. Also we realize an adaptation of the PRRT in the NMM based on a table lookup algorithm, which can adapt the AHEC parameters in PRRT according to the varying network conditions in an on the fly manner. To evaluate the implementation we carry out tests with real-time multimedia streams in wireless multicast scenarios and analyze the results. At the same time we examine and refine the configuration of the test.

PRRT is a protocol based on the AHEC scheme, which combines ARQ and FEC technology. It provides predictable reliability under predictable delay for the delivery of real-time audio-visual media. NMM is characterized with the flow graph based middleware architecture and supports for creating networked multimedia applications in different operating systems. It allows for integrating new transport mechanism. The architectures and features of both parts are introduced in chapter 2 as the context of the thesis. Moreover, the chapter explains the adaptation of PRRT and the integration of PRRT into the NMM in an abstract level.

Chapter 3 discusses the theoretical foundations concerned with the adaptation of the PRRT. It includes the optimization of the AHEC parameters and the analysis of those optimized parameters in different network conditions, the smoothing method of the fluctuant network conditions and the lookup based algorithm of the adaptation.

PRRT is integrated into the communication framework of NMM by realizing the PRRT transport strategies. The strategies can be applied in the client / server model and a distributed NMM flow graph. The adaptation of PRRT is implemented and integrated into a new socket of NMM as a socket wrapper for the PRRT socket, which can be used by the PRRT strategies. The details of the implementation are described in chapter 4.

The performance of PRRT has been evaluated within NMM in wireless multicast scenarios. The results are presented in chapter 5. From the test results, PRRT performs well in multicast scenarios, even if the receiver is moving. The adaptation on the PRRT can timely apply the pre-computed AHEC parameters according to the network conditions.
Altogether, the integration of PRRT into the NMM is accomplished in this thesis. The corresponding evaluations of the performance are also achieved. They show that PRRT is a prospective UDP based protocol for real-time streaming and the NMM is featured with its openness, uniformness, extensibility and flexibility.

6.2. Future Work

The development and the study of the PRRT in multicast scenarios is an ongoing topic at the Telecommunications Lab of Saarland University. The implementation related to the PRRT in NMM needs to be further improved. Beyond this thesis the following points might be followed up in the future.

- More tests of the adaptation and in wireless scenarios will be performed to show the performance of the PRRT, since only a few specific test cases described in chapter 5 were done due to the time limitation. This would improve the completeness of the PRRT and refine the adaptation. For example, long time tests should also be considered, as we didn’t do some long time tests of the adaptation.

- The improvement of the lookup based adaptation is expected. For example, the reference parameter sets stored in the table should be refined. Additionally, along with considering more wireless scenarios, more optimized parameter sets should be stored in the pre-calculated table. We need analyse those parameter sets and selectively add them into the table.

- The algorithm of the adaptation on the PRRT is a lookup based algorithm. We are going to implement a simplified algorithm of the adaptation which can perform the online computing of the parameters. This will save memory utilization, since the table size becomes larger in lookup based algorithms if the algorithm considers more network scenarios.

- The advanced development and improvement of the integration of PRRT into NMM may be another goal in the future. For example, the error handling related to the integration of PRRT should be completed and improved, since no exception handling is implemented in the adaptation. The problem should be solved that the unreliable strategies cannot be destructed during the termination of the application, if they are used as the reliable strategies in the parallel binding.
Appendix

A. List of Source Code Files

The following is a list of the source files we created for the implementation.

- **Socket Level:**
  - `/nmm-2.2.1/nmm/utils/net/prrt/AdaptPRRTSock_Linux.cpp`
  - `/nmm-2.2.1/nmm/utils/net/prrt/AdaptPRRTSock.hpp`
  - `/nmm-2.2.1/nmm/utils/net/prrt/PRRTSock_Linux.cpp`
  - `/nmm-2.2.1/nmm/utils/net/prrt/PRRTSock.hpp`

- **Adaptation Level:**
  - `/nmm-2.2.1/nmm/utils/net/prrt/Adaptability.cpp`
  - `/nmm-2.2.1/nmm/utils/net/prrt/Adaptability.hpp`
  - `/nmm-2.2.1/nmm/utils/net/prrt/AdaptByLookup.cpp`
  - `/nmm-2.2.1/nmm/utils/net/prrt/AdaptByLookup.hpp`
  - `/nmm-2.2.1/nmm/utils/net/prrt/OptimizedParameterSet.cpp`
  - `/nmm-2.2.1/nmm/utils/net/prrt/OptimizedParameterSet.hpp`

- **Transport Strategy Level:**
  - `/nmm-2.2.1/nmm/comm/netstrategy/prrt/IPRRTControl.idl`
  - `/nmm-2.2.1/nmm/comm/netstrategy/prrt/PRRTAddress.cpp`
  - `/nmm-2.2.1/nmm/comm/netstrategy/prrt/PRRTAddress.hpp`
  - `/nmm-2.2.1/nmm/comm/netstrategy/prrt/PRRTRecieverStrategy.cpp`
  - `/nmm-2.2.1/nmm/comm/netstrategy/prrt/PRRTRecieverStrategy.hpp`
  - `/nmm-2.2.1/nmm/comm/netstrategy/prrt/PRRTSenderStrategy.cpp`
  - `/nmm-2.2.1/nmm/comm/netstrategy/prrt/PRRTSenderStrategy.hpp`

- **Application Level:**
  - `/nmm-2.2.1/nmm/plugins/net/prrt/IPRRTParameters.idl`
  - `/nmm-2.2.1/nmm/plugins/net/prrt/IPRRTNet.idl`
  - `/nmm-2.2.1/nmm/plugins/net/prrt/PRRTNetSinkNode.cpp`
  - `/nmm-2.2.1/nmm/plugins/net/prrt/PRRTNetSinkNode.hpp`
Appendix

- /nmm-2.2.1/nmm/plugins/net/prrt/PRRTNetSourceNode.cpp
- /nmm-2.2.1/nmm/plugins/net/prrt/PRRTNetSourceNode.hpp
B. PRRT Tutorial

PRRT (Predictably Reliable Real-time Transport) protocol is our kernel module of the AHEC scheme. It is used to evaluate the parameter sets of AHEC in various multicast scenarios in this thesis. The goal of PRRT is to reduce the packet loss rate as much as possible under the delay constraint. PRRT is approaching the release version of the prototype implementation regarding to the protocol stack with Predictable Reliability under Predictable Delay (PRPD). For more detail please see the homepage of PRRT: http://www.nt.uni-saarland.de/projects/prrt/.

The following is a short introduction to compile and use PRRT. To follow the below instructions, the system is required.

Requirements of PRRT

- Linux version 2.6.26-9-generic or later
- automake (for generating makefiles, dependencies: autoconf and autotools-dev)
- make 3.81 or later
- g++ 4.3.1 or later
- gcc 4.3.1 or later

Download PRRT

Update the nmm project repository to at least the revision 115.

Install PRRT

1. Extract the archive and change the directory into the folder to which the archive was extracted.

   ```
   tar -xjvf prrt-r195.tar.bz2 && cd prrt
   ```

2. Run the script autogen.sh if you want to regenerate the Makefile.in, aclocal.m4 and config.h.in.

   ```
   ./autogen.sh
   ```

3. Generate the makefiles.

   ```
   ./configure
   ```

   A check list of all dependencies and tools will be shown during running this command.

4. Compile and install PRRT.

   ```
   make && sudo make install
   ```
5. Done! In /usr/local/include/prrt/ you will see all developer’s headers related to PRRT and the libs are located to /usr/local/lib/.

Here is an example of all steps for installing PRRT you could follow:

- **Environments**:
  - PRRT Revision: 195
  - Linux version 2.6.27-9-generic
  - GNU automake 1.10.1
  - GNU Make 3.81
  - gcc (Ubuntu 4.3.2-1ubuntu11) 4.3.2
  - g++ (Ubuntu 4.3.2-1ubuntu11) 4.3.2

- **Steps**:

  ```
  1. svn up
  2. tar -xjvf prrt-r195.tar.bz2
  3. cd prrt/
  4. ./configure
  5. make
  6. sudo make install
  ```

**How to Use PRRT with an Example**

1. Load PRRT into the kernel with the following command under “prrt/prrt_module/”:

  ```
  sudo insmod media_transport_protocol.ko
  ```

   **NOTE**: to remove the inserted module use the command:

  ```
  sudo rmmod media_transport_protocol
  ```

2. Emulate the network environment of the SENDER with the packet loss rate 0.1% on the network interface eth0:

  ```
  sudo tc qdisc add dev eth0 root netem loss 0.1%
  ```

   **NOTE**: The previous command is to add the netem kernel with a fixed packet loss rate at the first time you emulate the network, in order to randomly drop the packets going out of the local Ethernet. To change the packet loss rate after the netem kernel being added with the above command, you use the command with the keyword ”change” like:

  ```
  sudo tc qdisc change dev eth0 root netem loss 2%
  ```

   And you can use the following command to remove the active queuing discipline (”qdisc”) for dev eth0:
B. PRRT Tutorial

1  sudo tc qdisc del dev eth0 root

For more details refer to NETEM tutorial in Appendix D.

3. Execute under the directory of the SENDER “/prrt/example/” :

1  ./prrtcat -j 4000 -k 11 -b 0 -d 30 -r 114

Some usages of prrtcat :
- \( j \) : data rate \( (R_d) \)^1
- \( k \) : data block length \( (k) \)
- \( b \) : prior redundancy \( (N_p[0]) \)
- \( d \) : overall block length \( (n) \)
- \( r \) : retransmission round \( (\vec{N}_p) \)

4. Execute under the directory of the RECEIVER “/prrt/trunk/example/” :

1  ./prrtrecv 224.0.2.2 > /dev/null

5. Get the socket number of the working PRRT socket of the RECEIVER.

1  ls /proc/prrt/

The files of each PRRT socket reside in a directory with the socket number.

6. Output the parameters on the RECEIVER (assume the socket number is 1.) :

1  ./evalscript.sh 1

The following is the snippet of the output :

1  19:23:52 1558 53 69 0 .0272 0 5 0 .03543003 29

corresponding to in order :
current time, the number of received packets, the total number of lost packets, the
total number of received redundancy packets, the total number of missed packets,
the smoothed PLR (by exponential smoothing), the last group PLR, the last group
RTT, the residual PLR and the residual RI as well as the SNR.
Note that the SNR is only available for wireless scenarios.

7. Change the parameters of PRRT kernel at runtime, e.g. the parameters corresponding
to -k -b -d and -r in prrtcat. The parameters contains: “d_target” \( (D_{target}) \),
“plr” \( (PLR_{target}) \), “codp” (coding parameters), “rtr” (retransmission rounds) and
“t_repl” (replace time).

1  echo [param_value] > /proc/prrt/[socket_no]/[parameters]

e.g.:

^1 With this option prrtcat can generate the dummy data of the given data rate.
changes the retransmission parameters that include the number of retransmission rounds and retransmission multipliers.

<table>
<thead>
<tr>
<th>The number of retransmission rounds</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multipliers in each retransmission round</td>
<td>[112]</td>
</tr>
</tbody>
</table>

In the following we’re introducing a brief example for streaming DVB with PRRT.

- **Requirements:**
  - **Sender**
    * PRRT Revision 195 or higher (compiled)
    * Linux version 2.6.27-9-generic or higher
    * dvbstream v0.6 or higher (for receiving a TS from the DVB.)
  - **Receiver**
    * PRRT Revision 195 or higher (compiled)
    * Linux version 2.6.27-9-generic or higher
    * mplayer (for playing back the TS transmitted via PRRT.)

Notes: To accomplish this example, you must have a DVB receiver in hand and install it at the sender system.

- **Steps:**
  - **Sender**
    ```
    $ echo "3" > /proc/prrt/1/rtr
    ```
    ```
    cd prrt_module/
    sudo insmod media_transport_protocol.ko
    cd ../example/
    dvbstream -f 546000 -bw 8 -qam 16 -tm 8 -gi 4 -o 0 17 545 546 | ./prrtcat -k 1 -b 0 -r 112
    ```
  - **Receiver**
    ```
    cd prrt_module/
    sudo insmod media_transport_protocol.ko
    cd ../example/
    ./prrtrecv 224.0.2.2 | mplayer -cache 1024
    ```

Now MPlayer is displaying the transmitted video on a popup GUI-window.
C. NMM Tutorial

This tutorial introduces the installation and testing of NMM. We focus only on the open source version 2.2.1 for linux in this thesis. It should be noted that some instructions below are strongly based on the thesis. A more general tutorial can be found in the NMM homepage: [http://www.motama.com/nmmdocumentation.html](http://www.motama.com/nmmdocumentation.html).

Requirements of NMM

- **necessary**:
  - automake (for generating Makefile.in from Makefile.am, if running autogen.sh.)
  - make
  - g++ (recommend 4.3.1)
  - gcc (recommend 4.3.1)
  - ltdl (for base functionality. e.g.: libltdl7-dev)
  - GNU bison (yacc programm should be GNU bison.)
  - x11 extension (for XDisplayNode. e.g.: x11proto-xext-dev, libxext-dev and libxv-dev required for adding XShm.h and Xvlib.h in x11/extensions/.)

The packages above are required for the installation of NMM. Without them errors occur.

- **optional**:
  - doxygen (for the documentation.)
  - docbook (for the documentation. e.g.: docbook-utils)
  - nasm (for integrating the assembler optimized code.)
  - dvb_headers
  - cdparanoia
  - alsa (for ALSAPlaybackNode. e.g.: libasound2-dev)
  - raw1394, e.g.: libraw1394-dev (uncertain, to check.)
  - libvorbis (for OggVorbisDecodeNode. e.g.: libvorbis-dev)
  - flex
  - lex
  - external libraries e.g.: nmm-2.2.0-optional-external-libs-linux.tar.gz (for encoding/decoding plug-ins.)

The installation of NMM without the optional packages causes one or more warning messages and then one or more functionalities of NMM are limited in later use.

**NOTE:** The name of the packages would be different in various Linux distributions.

Download NMM

Check out the latest version of our NMM project repository with the following command:
Install NMM

1. Extract the external libraries under the project directory “nmm/” :
   `tar xvfz nmm-2.2.0-optional-external-libs-linux.tar.gz`

   The external libraries are needed if you use the optional features of NMM e.g.: encoding or decoding plug-ins.

2. Set the environment variable `LD_LIBRARY_PATH`, which specifies all paths where external libraries required by NMM are installed. For bash, use
   `export LD_LIBRARY_PATH=$PWD/nmm-2.2.0-external-libs/lib:/ $LD_LIBRARY_PATH`

   or for tcsh
   `setenv LD_LIBRARY_PATH $PWD/nmm-2.2.0-external-libs/lib: $LD_LIBRARY_PATH`

   NOTE : you should consider the appropriate (absolute) path of the library. The previous command is executed under the project directory “nmm/”.

3. Create a directory (under the project directory) where NMM is to be installed :
   `mkdir nmm-2.2.1-installed/`

4. Under the directory ”nmm/nmm-2.2.1/” execute :
   `chmod +x autogen.sh`

   to make the script for automatically building system preparation executable.

5. Run the script under “nmm/nmm-2.2.1/” to generate the required preparation for the installation of NMM.
   `./autogen.sh`

6. Now you can run the configure script for the first time under the same directory of the previous step.
   `./configure --with-extra-libs=$PWD/..nmm-2.2.0-external-libs/lib/ --with-extra-includes=$PWD/..nmm-2.2.0-external-libs/include/ --prefix=$PWD/..nmm-2.2.1-installed/ --enable=all --enable-debug`

   During running you will see many messages about the status of requirements check and the config. If it terminates successfully, NMM features are listed to indicate which are either enabled or disabled.
7. To build NMM, use simply:

```
1  make
```

or

```
1  make parallel -j2
```

The later one will really speed up the compilation, if multi-core computer is available.

8. Finally, you can install NMM with:

```
1  make install
```

9. Installation done!

Here is an example of all steps for installing NMM you could follow:

- **Requirements**:
  
  - NMM Revision: 117
  
  - Ubuntu Linux version 2.6.32.1
  
  - `install_nmm_dependencies.sh` (already in the svn repository `nmm/scripts/`, which tries to install all mandatory and optional packets for installing NMM.)
  
  - NMM external libraries 2.2.0 (already in the svn repository `nmm/nmm-2.2.0-optional-external-libs-linux.tar.gz`)

- **Steps**:

  ```
  1  svn co https://projects.nt.uni-saarland.de/svn/nmm
  2  cd nmm/
  3  tar xvzf nmm-2.2.0-optional-external-libs-linux.tar.gz
  4  sudo scripts/install_nmm_dependencies.sh
  5  export LD_LIBRARY_PATH=$PWD/nmm-2.2.0-external-libs/lib/
  6  mkdir nmm-2.2.1-installed/
  7  cd nmm-2.2.1/
  8  chmod +x autogen.sh
  9  ./autogen.sh
  10  ./configure --with-extra-libs=$PWD/.../nmm-2.2.0-external-libs/lib/ --with-extra-includes=$PWD/.../nmm-2.2.0-external-libs/include/ --prefix=$PWD/.../nmm-2.2.1-installed/ --enable=all --enable-debug
  11  make
  12  make install
  ```

Notes: After running the 10th step, there are a lot of output messages on the screen. They are the check lists of the requirement for NMM. There can be still some warnings even though the dependent packets from “install_nmm_dependencies.sh” have been installed. In this case you may manually install the extra packets to eliminate the warnings if needed.
Test NMM with an Example

We assume that you installed NMM totally the same as the way we just showed above. Running this test is strongly related to the environment variable (LD_LIBRARY_PATH) you set during the installation of NMM (associated with the prefix path you specified in the configuration of NMM.) and the graph description file, called GD file, which is parsed by the tool clic to perform a specific multimedia operation, e.g. to play back a ts file. Of course you need a tested media file. In this test the environments are defined as follows according to the previous installed instructions:

- LD_LIBRARY_PATH : /home/bob/nmm/nmm-2.2.0-external-libs/lib/
- path of installed NMM : /home/bob/nmm/nmm-2.2.1-installed/
- GD file (already in svn repository) : /home/bob/nmm/nmm-2.2.1/examples/prrt/mpegtsread_demux_mpeg_av_dec_play.gd
- ts media file : /home/bob/nmm/media/test.ts

NOTE : To match the GD file, the test media file have to be a transport stream media file with .ts extension (a container format) which encapsulates the packetized elementary streams with MPEG audio and MPEG video format.

1. Change to the directory of the tool clic (in the installed NMM directory nmm/nmm-2.2.1-installed/bin).

   ```
   cd /home/bob/nmm/nmm-2.2.1-installed/bin
   ```

2. Execute clic with given GD file and input media file.

   ```
   ./clic /home/bob/nmm/nmm-2.2.1/examples/prrt/mpegtsread_demux_mpeg_av_dec_play.gd -i /home/bob/nmm/media/test.ts
   ```

Usage :

```
clic <GD file> -i <input media file>
```
D. NETEM Tutorial

This tutorial is based on some online references, e.g.

- http://www.linuxfoundation.org/collaborate/workgroups/networking/netem
- http://pupa.da.ru/tc/

“netem” offers network emulation functionality for testing protocols by emulating the properties of wide area networks: variable loss, delay, duplication, corruption, re-ordering and even more sophisticated traffic shaping such as rate control with advanced commands.

For simplicity, just only commands around this thesis were introduced here.

- **ADD**

  ```
  sudo tc qdisc add dev eth0 root netem loss 2%
  ```

  This command adds a queuing discipline (“qdisc”) to the local ethernet device (“dev eth0”) that enforces a network emulation (“netem”) with a fixed packet loss 2% to randomly drop the packets going out of the local Ethernet.

- **SHOW**

  ```
  sudo tc qdisc
  ```

  or

  ```
  sudo tc qdisc show
  ```

  These two commands show the currently active queuing discipline on all devices. To display the active queuing discipline only on some devices, take “eth0” for instance, use simply :

  ```
  sudo tc qdisc show dev eth0
  ```

- **CHANGE**

  ```
  sudo tc qdisc change dev eth0 root netem loss 10% delay 10ms
  ```

  Use “change” to modify an active queuing discipline. All outgoing packets of the local Ethernet have a fixed delay of 10 ms.

- **DELETE**

  ```
  sudo tc qdisc del dev eth0 root
  ```

  Finally, to remove an active queuing discipline, use “del” instead of “add”, “show” and “change”.
Bibliography


[Li08b] Zhao Li, Thorsten Herfet, *Beacon-driven Leader Based Protocol over a GE Channel for MAC Layer Multicast Error Control*, in International Journal of Communications, Network and System Science (IJCNS), 2008


